An analysis of seek time based on Fitts's Law $T = K_0 + Klog_2(D/S + 0.5)$ where T = time to position cursor using mouse (seek time), K_0 = constant time to adjust grasp on mouse, K = constant normalization factor (positioning device dependent), S = size of target in $pixels_2$, D = distance in screen pixels, helps explain our results because the ratio of the distance (D) to target size (S) is smaller for pie menus. The fixed target distance and increased size of targets for pie menus decreases the mean positioning time as compared with linear menus. In our experiment, the activation region for an item constitutes the target. All subjects were informed of the fact that their target was not necessarily the text, but the region containing the text target item. This was clearly understood by all participants. The font size for text items in both

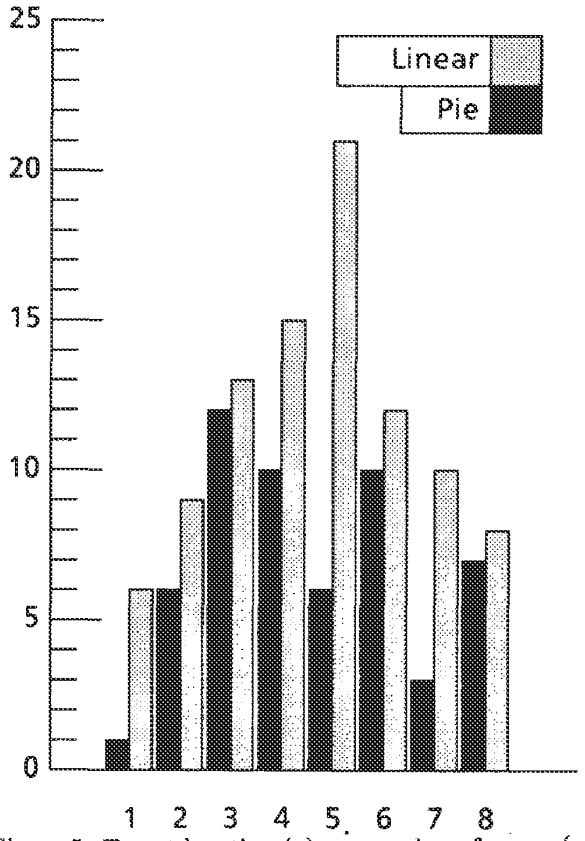


Figure 5: Target location (x) vs. number of errors (y)

menu styles was the same, yet the target region size for pie menus ($3500 - 6000 pixels^2$) was on the order of 2-3 times the size of linear menu activation region sizes ($1000 - 2000 pixels^2$). The distance from the center of a pie menu to an activation region is 10 pixels while the distance in linear menus varied from 13-200 pixels.

Figure 5 displays the target location plotted against the total number of errors across all subjects. Pie and linear menus seem to suffer from a similar phenomenon - errors are made more often on items in the central region of the menu display. These are the items with the most interaction with neighboring items [2].

	Task type			
	Pie	Linear	Unclass.	$Mean_{menu}$
Using pie menus	0.45	0.60	0.60	0.55
Using linear menus	0.88	0.73	1.24	0.95
$Mean_{task}$	0.66	0.66	0.92	

Table 4: number of errors means per cell, menu type, and task type (all observations including no errors)

Repeated measure analysis of variance results on the error rates show marginally statistically significant differences ($P = 0.087$) between pie and linear menus (Tables 4 and 5). No other statistically significant differences were observed.

Subjective results obtained in the pilot study repeated themselves in the experiment. Subjects were split on preferring one menu type over another but those who preferred linear menus had no strong conviction in this direction and most agreed that with further practice

	F	$PR > F$
Menu type	3.12	0.0869
Task type	0.93	0.4066
Menu type X Task type	1.34	0.2773

Table 5: repeated measures analysis of variance results for number of errors

they might prefer the pie menu structure. Those who preferred pie menus generally felt fairly confident in their assessment and this is reflected in the questionnaires.

One subject complained of having a problem with *menu drift* which is the phenomenon which occurs as the result of the cursor relocating to the relative screen location of the last selected target. With linear menus, this tends to "drift" the cursor towards the bottom of the screen. This may explain the higher error rate for linear menus, but the same problem occurs to a lesser degree with pie menus. This, in fact, we believe to be another positive feature of pie menus: the cursor drift distance is minimized. Most subjects had no problems coping with drift in either menu style. One area of further research is measuring the extent and effect of this problem.

CONCLUSIONS

What does this mean? Should we program pie menus

into our bitmapped window systems tomorrow and expect a 15-20% increase in productivity since users can select items slightly faster with pie menus. Pie menus seem promising, but more experiments are needed before issuing a strong recommendation.

First, this experiment only addresses fixed length menus, in particular, menus consisting of 8 items - no more, no less. Secondly, there remains the problem of increased screen real estate usage. In one trial a subject complained because the pie menu obscured his view of the target prompt message. Finally, the questionnaire showed that the subjects were almost evenly divided between pie and linear menus in subjective satisfaction. Many found it difficult to "home in on" a particular item because of the unusual activation region characteristics of the pie menu.

One assumption of this study concerns the use of a mouse/cursor control device and the use of pop-up style menus (as opposed to menus invoked from a fixed screen location or permanent menus). Certainly, pie menus can and in fact have been incorporated to use keyed input [7] and fixed "pull-down" style presentation (the pie menu becomes a *semicircle* menu). These variations are areas for further research.

One continuing issue with pie menus is the limit on the number of items that can be placed in a circular format before the size of the menu window is impractical. Perhaps, like the limiting factors in linear menus concerning their lengths, pie menus reach a similar "breaking point" beyond which other menu styles would be more useful. Hierarchical organization, arbitrarily shaped windows (Figure 6), numeric item assignment and other menu refinements as well as further analysis is contained in [7]. Pie menus offer a novel alternative worthy of further exploration.

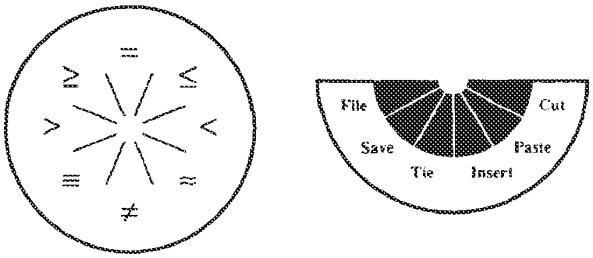


Figure 6: Advanced "pie" menus

ACKNOWLEDGEMENTS

The authors wish to thank the following people for their invaluable help in the preparation of the experiment, analysis of results and statistics, and this paper: Jim Purtilo, Nancy Anderson, Ken Norman, John Chin, Linda Weldon, Mark Feldman, Mike Gallaher, Mitch Bradley, and Glenn Pearson.

REFERENCES

[1] Adobe Systems Inc. *Postscript Reference Manual*, Palo Alto, Calif., 1985.

[2] Card, S.K. User perceptual mechanisms in the search of computer command menus, In *Proceedings - Human Factors in Computer Systems 1982* (Gaithersburg, Md., Mar. 15-17). ACM, New York, 1982, pp. 190-196.

[3] Card, S.K., Moran, T.P., and Newell, A. *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum, London, 1983.

[4] Dray, S.M., Ogden, W.G., and Vestewig, R.E. Measuring performance with a menu-selection human computer interface *Proceedings of the Human Factors Society: 25th Annual Meeting 1981* (Rochester, N.Y., Oct. 12-16). Human Factors Society, Santa Monica, Calif., 1981, pp. 746-748.

[5] Gettys, J. and Newman, R., *X Windows*. MIT, 1985.

[6] Gosling, J. *NeWS: A Definitive Approach to Window Systems* Sun Microsystems Corp., Mountain View, Calif., 1986.

[7] Hopkins, D., Callahan, J., and Weiser, M. Pies: Implementation, Evaluation and Application of Circular Menus. University of Maryland Computer Science Department Technical Report, 1988.

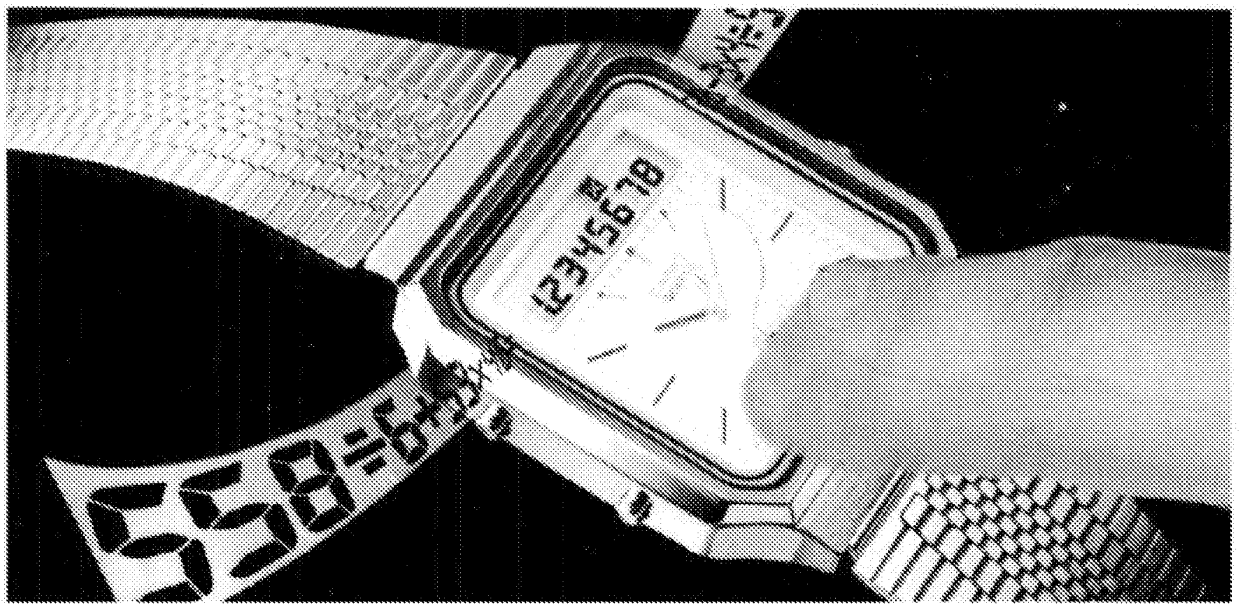
[8] Kirk, R. *Experimental Design: Procedures for the Behavioral Sciences*. Brooks-Cole, Belmont, Calif., 1968.

[9] McDonald, J.E., Stone, J.D., and Liebelt, L.S. Searching for items in menus: The effects of organization and type of target. *Proceeding of the Human Factors Society: 27th Annual Meeting 1983* (Norfolk, Virginia, Oct. 10-14). Human Factors Society, Santa Monica, Calif., 1983, pp. 834-837.

[10] Perlman, G. Making the right choices with menus. *INTERACT '84, First IFIP International Conference on Human Computer Interaction*. North-Holland, Amsterdam, 1984, pp. 291-295.

[11] Shneiderman, B. *Designing the User Interface*, Addison-Wesley, Reading, Mass., 1987.

[12] Xerox Corporation, *Interpress Electronic Printing Standard*. Stamford, Conn., 1984.



NOW... THE INVISIBLE CASIO CALCULATOR WATCH

Finger-write your figures on the watch face.

Introducing the timepiece that adds another dimension to watch technology. This new CASIO combines state-of-the-art microcomputer technology with the latest styling to give you an elegant timepiece with a multitude of functions.

And the most remarkable function of all is this... The watch face actually reads and computes math problems you trace on its face.

And there's more, much more... for less than $100.00.

ELECTRO-TOUCH TECHNOLOGY.

This handsome and superbly styled timepiece has a transparent crystal that reads finger-strokes you trace across its face. Each figure and math symbol you outline appears on the background digital display. Take your finger across twice (=) and the answer presents itself like magic.

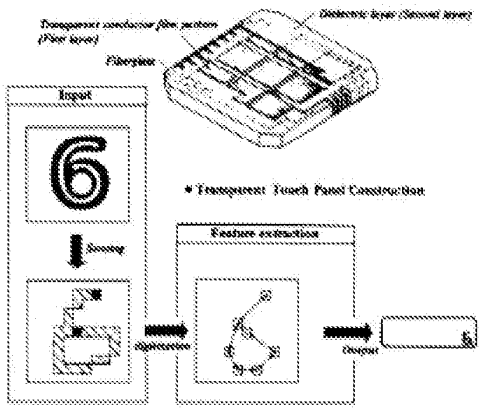
No keys, no keyboards, no need to use stylus or pen. Even the broadest fingers will work. Add, subtract, multiply, divide — perform chain and mixed calculations to eight places, plus decimal. There's even an indicator telling you which function is being performed.

DIGITAL PRECISION, ANALOG STYLE.

This handsome CASIO was created exclusively for the man who recognizes exceptional styling. And that's what you get with this sleek new timepiece. From its raised time markers to the elegant case and band of sight mesh steel or gold plate, this beauty has the special look of luxury that is never out of place... from the very casual to the most formal occasions.

You get the classic elegance and convenience of analog watch hands. Subdued in the background is the modern message of digital precision. The digital display can be set in 12-hour or 24-hour digital time. A pre-programmed calendar is set until the year 2019. It's a handsome and functional way to wear time with accuracy to 1/2 second per day.

HERE'S HOW THIS MARVEL WORKS



PRECISION ALARM AND STOPWATCH.

You can program this multi-talented wrist alarm for daily events. To wake you up. Catch your bus. Program it for any minute you choose. Or set it to beep-beep you every hour.

In addition to helping you organize your day, this gifted chronograph boasts a fiercely talented stopwatch. Record normal times and net times with accuracy to 1/10 second. A beep confirms starts and stops. Ideal for tallying cooking time, minutes on your parking meter or anything you like.

WE INTRODUCE IT AND WE GUARANTEE IT.

When we first heard the engineers at CASIO were on the brink of perfecting a finger-trace recognition calculator watch, we had hopes of being the first to offer it. And now that's a reality.

Because this innovative timepiece is now available only through On The Run, to be assured earliest delivery, please order yours now. Chrome and stainless model **AT-550** is only **$99.95** and gold-plated model **AT-550G** is **$119.95.**

See how this handsome accessory can be worn anytime, anywhere. Discover the convenience of finger-trace calculation and all the other special features of this talented timepiece. Once you see this handsome and functional timepiece, you're sure to want to keep it. If not, we guarantee your satisfaction. Simply return it in new condition within 30 days for a full and courteous refund. One-year warranty included.

CREDIT CARD HOLDERS ORDER TOLL-FREE TODAY.

To order, call toll-free number below, or send a check or money order for the total amount **plus $2.50** for the first watch, $1.00 for each additional watch for shipping and insurance. Add an additional $2.00 for UPS air delivery. ND residents add 4% tax.

800-437-4385

On The Run

107 Roberts St., Department PS-2
P.O. Box 67, Fargo, ND 58107
Telephone (701) 232-9400

Call or write for a free one-year subscription to our catalog of timepieces and high-tech items for better living.

AT-550 FP-1 © 1986 ON THE RUN

GUARANTEE CERTIFICATE

MODEL: .....

DATE OF PURCHASE: .....

OFFICIAL DEALER STAMP:

UG1183A

Dear Customer,

Congratulations on your purchase of this Casio Digital Quartz Watch. Although highly sophisticated, this watch is delightfully simple to use, and by referring to this guide you will enjoy all the features of this reliable ultra-precise timepiece.

Notice: As this watch contains precise electronic components, it is suggested that the back cover be removed only by authorized personnel.

Reading the Display
Regular timekeeping mode
(Time)
Calendar
Hour
Minute
Second
Year
Month
Day
Date
SU 6-30
Press Ⓐ.
Press Ⓐ.
Crown
Hour hand
Minute hand
Second hand
SU: Sunday MO: Monday TU: Tuesday WE: Wednesday TH: Thursday FR: Friday
SA: Saturday
2
(Calculator mode)
(Daily alarm mode)
(Stopwatch mode)
Time signal ON-mark
Alarm ON-mark
Mode indicator
Hour
Minute
Second
1/10 second
CAL 0.
8-35 AL
ST 0003-9
Press Ⓑ.
Press Ⓑ.
Press Ⓑ.
Press Ⓑ.
If the Ⓑ button is pressed after a calculation or setting daily alarm, the display reverts to the time display.
(Sound demonstration) Press and hold the Ⓑ button to sound the buzzer.
3

Setting time and calendar

(Regular timekeeping) (Second adjusting) (Hour setting) (Minute setting)

Press and hold Ⓑ in time mode to set new digital time.

Press Ⓒ on a time signal to correct seconds.*

Press Ⓓ to set hour digits. One hour advances with every push of Ⓓ.

Press Ⓔ to set minute digits. One minute advances with every push of Ⓔ.

*Precise time can be maintained by correcting the seconds once a month on a time signal from a radio, TV, telephone, etc.
(Quick digit advance) When the Ⓓ button is pressed for more than 2 seconds, the digit advances quickly. When released, the digit advance will stop.

IMPORTANT: Setting sequence MUST BE FOLLOWED when making any new setting.

(12/24-hour format setting) (Year setting) (Month setting) (Date setting)

Press Ⓓ to set 12/24-hour format. With every push of Ⓑ, the display is switched between the 12 and 24 formats.

Press Ⓒ to set year digits. One year advances with every push of Ⓒ.

Press Ⓓ to set month digits. One month advances with every push of Ⓓ.

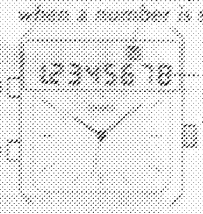
Press Ⓔ to set date digit. One date advances with every push of Ⓔ. Press Ⓑ to complete.

(Auto-retrieve function) When setting the watch, if you leave it alone for 2 to 4 minutes, the display will automatically return to the regular timekeeping mode.

Calculator operation

Deletes the displayed number by one digit for entry correction.

A function command sign (flashes when a number is set as a constant).



8-digit entry (7-digit for negatives) can be made.

Clears entry for correction.
Releases overflow or error check. Overflow is indicated by an "E" sign and stops the calculation.
Overflow occurs when the integer part of an answer, whether intermediate or final, exceeds 8 digits (7 digits for negatives).

[Before starting calculations]

Make sure the display shows 0 before starting calculations.
To input figures, write them directly on the glass with your finger-tip. (No input is possible with a fingernail or pen.)
Write figures slowly and carefully using the whole space of the glass, checking your entries on the display.
Be sure to stop your finger whenever it comes to an inflection point of a figure, and then move the finger again.
When writing a figure or symbol with multiple strokes, move your fingertip off the glass at the end of each stroke, and start the following stroke before the input recognition indicator (→→→) disappears.

Making strokes not in the specified order or directions may input a wrong figure or symbol.
"Please carefully read the "Notes on how to input".
Input may become difficult in low temperatures or when the air is too dry.
Avoid operation when moisture or dirt are present on the watch face. Wipe off with a dry, soft cloth.
If the Ⓑ button is pressed with your finger touching the glass, a figure or symbol may be input. Press Ⓐ or Ⓔ button to clear.
When calculating with the watch off your wrist, keep touch contact with the back of the case.
A special process has been used in manufacturing the crystal surface to allow the touch sensor function.
This is protected with a special coating to resist dirt and scratches.
Special care should be given to the crystal. This care should include avoiding scratches if at all possible as this can cause touch key sensor input problems.

This watch incorporates a unique new mechanism which permits you to carry out calculations by writing figures directly on the glass with your fingertip.

What is this new mechanism?
Finger-Trace (handwritten figures or symbols) recognition system. The mechanism reads the order in which strokes are made and their directions.

Notes on how to input

The following points should be observed when you write figures.

0

Align each end of the stroke. (Other writing forms)

1

Draw a straight, vertical line. (Other writing forms)

2

Clearly show the inflection point. (Other writing forms)

3

- Both upper and lower curves should be of the same size.
- The starting, inflection, and ending points should all line up.
- The figure should not lean over to the right. (Other writing forms)

4

- Clearly show the inflection point of the first stroke.
- Draw the second stroke parallel to the line drawn down to the inflection point of the first stroke. (Other writing forms)

5

- Clearly draw the line ②.
- Clearly show the inflection point. (Other writing forms)

6

Intersect the downward stroke near the center line, dividing the figure into upper and lower halves. (Other writing forms)

7

- Draw the line indicated by ②, horizontally.
- Clearly show the inflection point.
- Draw the line indicated by ③ diagonally. (Other writing forms)

8
• Start from the upper right, and align the end start points.
• Draw both upper and lower circles the same size.
(Other writing form)
9
• Make the circle a little larger.
• Align the intersection point with the starting point.
(Other writing form)
Draw a horizontal and a vertical line intersecting at right angles.
(Other writing form)
Draw a horizontal line.
Draw two diagonal lines intersecting at right angles.
(Other writing form)
Input of three strokes or more will become
• Draw two horizontal lines.
Lightly touch the glass. Don't stroke your fingertip over it.

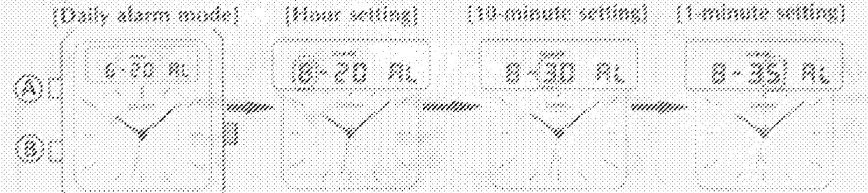
Be sure to press the Ⓒ button when starting calculations.

EXAMPLE	OPERATION	READ-OUT
Basic calculation: (12 − 0.5) × 3 ÷ 7 = 4.9285714 . . .	Ⓒ12 − 0.5 × 3 ÷ 7(=)	4.9285714
Constant calculation: 3 + 4 = 7 (4 is constant)	Ⓒ4 + + 3(=)	7
8 + 4 = 12	8(=)	12
3 − 4 = −1	Ⓒ4 − − 3(=)	−1
8 − 4 = 4	8(=)	4

EXAMPLE	OPERATION	READ-OUT
3 × 4 = 12	Ⓒ4 × × 3(=)	12
8 × 4 = 32	8(=)	32
3 ÷ 4 = 0.75	Ⓒ4 ÷ ÷ 3(=)	0.75
8 ÷ 4 = 2	8(=)	2
25^2 = 625	Ⓒ25 × × (=)	625
25^3 = 15625	(=)	15625
25^4 = 390625	(=)	390625

Setting daily alarm

If the daily alarm is set the buzzer sounds for 20 seconds at the preset time every day until cleared. To stop the buzzer while sounding, press the Ⓐ button. If the time signal is set, the watch sounds every hour on the hour.



Press and hold Ⓐ in daily alarm mode to set hours.

One hour advances with every push of Ⓑ.*

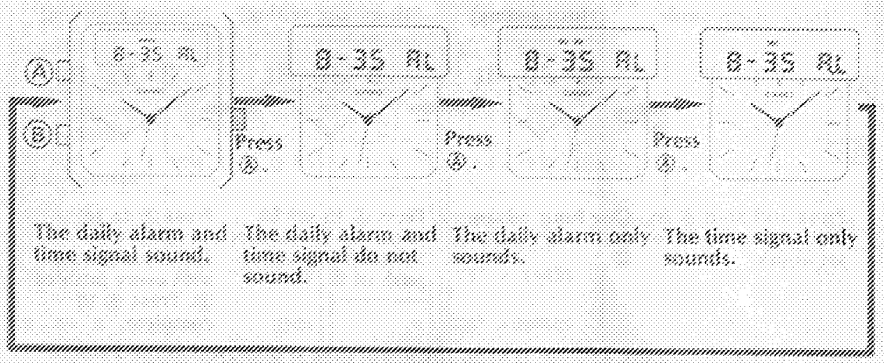
Press Ⓐ to set 10 minutes. 10 minutes advance with every push of Ⓑ.

Press Ⓐ to set minutes. One minute advances with every push of Ⓑ. Press Ⓐ to complete.

*When the watch is in the 24-hour system, the alarm time is displayed in the 24-hour system.

[ON or OFF setting of daily alarm and time signal]

[The alarm-ON-mark and time-signal-ON-mark appear.]



[The alarm-ON-mark and time-signal-ON-mark disappear.]

The daily alarm and time signal sound.

[The alarm-ON-mark only appears.]

The daily alarm and time signal do not sound.

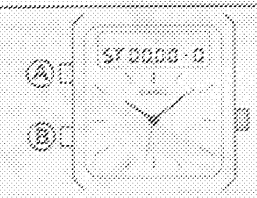
[The time-signal-ON-mark only appears.]

The daily alarm only sounds.

The time signal only sounds.

Press Ⓑ.

Stopwatch operation



Press Ⓐ to start or stop.
Press and hold Ⓐ to reset.
A signal confirms start/stop operation.
(Working range) The stopwatch display is limited to 59 minutes 59.9 seconds, for longer times reset and started again.

Setting analog timekeeping

1) Pull the crown out when the second hand is at the 12 o'clock position and the second hand stops.
2) Set the hands by turning the crown.
3) In accordance with a time signal, push the crown in.

A gain or loss of one second or less may result from properties of mechanical parts.

[How to replace the battery] Calculator's battery replacement should not be attempted without use of the correct tool.

1. Check the type of back cover 2. Removing the back cover Place the watch on soft material, like a cloth, and hold it firmly.

Screw-on type

With the Adjustable Case Opener, turn the back cover counterclockwise.

Snap-on type (A)

Insert Opener A in the recess and move from side to side to make a gap between the cover and the case. Then use the opener to pry off the cover.

Case Opener A

Snap-on type (B)

Fit Case Opener B into the notch and pry open the back cover.

Case Opener B

BATTERY LIFE: 12-month battery life starts when battery is factory installed. At first sign of power fade (dim display), renew battery at sales store or Casio distributor.

3. Replacing the battery 4. AC (ALL CLEAR) 5. Fitting the back cover

Using a screwdriver, remove screws from the battery holder.

Replace dead batteries and attach the battery holder.

As shown below, touch the AC contact and the battery (+) side with metallic tweezers. Contact should be about 2 seconds.

Using the Adjustable Case Opener, tighten the back cover.

Place the watch on a hard surface and push the back cover in gently.

CAUTION:
• Avoid touching the contact (—) of the battery.
• Never hold the tweezers with metallic tweezers.

IMPORTANT:
If Conducting AC (ALL CLEAR) is necessary, when a new battery has been put in, because the movement readings may cause erratic displays.
In such a case, pushing the light button will turn on the display.

Hold the watch horizontally and snap on the back cover.

Care of your watch

- Battery Life: 12-month battery life starts when battery is factory installed. At first sign of power fade (dim display), renew battery at sales store or Casio distributor.
- As this watch contains precise electronic components, it is suggested that the back cover be removed by authorized personnel.
- Your watch is ranked A through E according to the water resistance chart below. Check the ranking of your watch to determine proper use.

Rank	Case Designation	Splashes, rain, etc.	Swimming, car-washing, etc.	Snorkeling, diving, etc.	Scuba diving
A*	—	No	No	No	No
B	WATER RESISTANT	Yes	No	No	No
C**	50M WATER RESISTANT	Yes	Yes	No	No
D***	100M WATER RESISTANT	Yes	Yes	Yes	No
E****	200M WATER RESISTANT	Yes	Yes	Yes	Yes

*The watch is sweat-resistant but not water-resistant. So be careful not to get it wet. The pendant type does not meet sweat-resistant levels.

**50M WATER RESISTANT casing model does not permit underwater button operations.

***100M WATER RESISTANT casing model permits underwater button operations (except where buttons are countersunk). Should the watch be exposed to sea water, wash it well with fresh water and wipe dry.

****200M WATER RESISTANT casing model withstands scuba diving use (except diving using helium-oxygen gas).

- A waterproof rubber seal is used to exclude water and dust. As rubber deteriorates after long usage, the seal should be replaced periodically (every 2—3 years).
- Should water or condensation appear in the watch, immediately have it overhauled because water can corrode electronic parts inside the case.
- Avoid exposing it to extremely high and low temperatures.
- Although the watch is designed to withstand shocks under normal use, it is inadvisable to subject it to hard knocks — rough usage or drops onto hard surfaces —.
- Avoid fastening the band too tightly. You should be able to insert your finger inside the band.
- Clean the watch and bracelet with a soft cloth, dry or moistened with mild soap. To avoid surface damage, never use volatile chemicals (such as benzine, thinners, spray cleaners, etc.).
- Gold plated surfaces can be kept in good condition by regular wiping with a soft damp cloth. Discoloration can be removed with detergent. Always keep unused watches in a dry place.

* Avoid exposing the watch to strong chemicals such as gasoline, cleaning solvent, aerosol spray, adhesive agent, paints, etc. whose chemical action will destroy the seals, case and finish.

Specifications

Accuracy at normal temperature: ± 15 seconds per month
Display capacity:
* Regular timekeeping mode
Analog: Hour, minute and second hands
Digital: Hour, minute, second, am/pm, month, date, day
Time system: Changeover between 12/24-hour format
Calendar system: Auto-calendar pre-programmed until the year 2019
* Calculator mode
8 digits (7 digits for negatives)
Abilities: Four basic calculations, chain & mixed operations, constants for + (–) x ÷
Overflow check: Indicated by the "E" sign, locking the calculator mode
* Stopwatch function
Measuring capacity: 59 minutes 59.9 seconds
Measuring unit: 1/10th of a second
Measuring modes: Normal time and net time

* Daily alarm
* Hourly time signal
Battery:
One silver oxide battery (type: No. 396)
Approx. 12-month on No. 396 (under following conditions: alarm — 20 seconds/day, calculation — 10 minutes/day)

NOTE: THERE IS NO WAY that components can be damaged or malfunction, due to misoperation of buttons. If confusing information appears on the display it means entry sequence was incorrect. Please read the manual and try again.

Warranty Certificate

THIS WARRANTY CERTIFICATE IS VALID ONLY FOR SERVICE IN THE COUNTRY OF PURCHASE.

Should this watch malfunction under normal use, it will be repaired without charge for a period of one year from the date of purchase. If the watch requires service within the warranty period, request repair or adjustment at the store where purchased or the authorized Casio watch distributor, presenting the watch together with this warranty certificate. The customer shall not have any claim under this warranty for repair or adjustment expenses if:

(1) The trouble is caused by improper, rough or careless treatment.
(2) The trouble is caused by a fire or other natural calamity.
(3) The trouble is caused by improper repair or adjustment made by anyone other than the authorized Casio watch distributor or its retailers.
(4) The case, band, glass or battery is damaged or worn.
(5) This warranty certificate is not presented when requesting service.
(6) The name and address of the authorized distributor or the retailer are not stamped in the warranty certificate.
(7) The date of purchase, model name and manufacturing number are not entered in the warranty certificate.

* The above warranty applies to regions other than the United States of America.
* United Kingdom — This undertaking is in addition to consumers statutory rights and does not affect those rights in any way.

CASIO

CASIO ELECTRONIC WATCH LIMITED WARRANTY

This product, except the case (including buttons), band and battery is warranted by Casio Inc. to the original purchaser to be free from defects in material and workmanship under normal use for a period of one year from the date of purchase. During the warranty period, and upon proof of purchase, the product will be repaired or replaced with the same or similar model at our option, without charge for either parts or labor at a Casio Repair/Parts Center listed on this card. There is a $4.95 charge for handling, postage, and insurance. Please enclose your check or money order payable to Casio Inc., when returning your watch for any repair. The warranty will not apply to this product if it has been misused, abused or altered. Without limiting the foregoing, leakage of battery, bending or dropping of the unit, or visible cracking of the LCD display are presumed to be defects resulting from misuse or abuse.

NEITHER THIS WARRANTY NOR ANY OTHER WARRANTY EXPRESS OR IMPLIED, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY, SHALL EXTEND BEYOND THE WARRANTY PERIOD. NO RESPONSIBILITY IS ASSUMED FOR ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING BUT WITHOUT LIMITING THE SAME, TO MATHEMATICAL ACCURACY OF THE PRODUCT. SOME STATES DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS AND SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THAT THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

CASIO INC. 15 Gardner Road, Fairfield, NJ 07006
CASIO SERVICE CENTER
National Repair/Parts Center 175 Route 46 West, Fairfield, NJ 07006
For information on other Authorized Service Center,
please call: 201-575-5695

Estimado Cliente:
Felicitaciones por la compra de este Reloj Digital de Cuarzo Casio. Aunque se trata de un reloj altamente sofisticado, resulta fácil de usar, y remitiéndose a esta guía Ud. podrá disfrutar todas las características de esta pieza ultra-precisa.

Nota: Como este reloj cuenta con componentes electrónicos de precisión, se recomienda que la cubierta posterior sea extraída únicamente por personal autorizado.

UDC 621.371.029.55: 621.376: 621.394: 681.32

Indexing Terms: H.f. radio communication systems, Modems, Microprocessor application

Bit-slice microprocessors in h.f. digital communications

S. D. SMITH, B.Sc.,*

P. G. FARRELL, B.Sc., Ph.D., C.Eng., M.I.E.E.*,

K. R. DIMOND, B.Sc., Ph.D., C.Eng., M.I.E.E.*

Based on a paper presented at the IERE Conference on Microprocessors in Automation and Communications held in London in January 1981

SUMMARY

A 2·4 kbit/s baseband modem is being designed for use at h.f., incorporating modulation/demodulation techniques that are matched to those frequencies and the problems associated with them. Fed by a continuous serial data stream, the modulator functions are implemented wholly by a bit-slice microprocessor, and controlled by another more conventional microprocessor. Analogue output waveforms are generated in a d/a converter, which is driven by the bit-slice machine. Demodulation is performed in a similar device, using an a/d input and giving a serial output.

*Electronics Laboratories, University of Kent at Canterbury, Canterbury, Kent CT2 7NZ

1 Introduction

Over the past ten years, devices for transmission and reception of data have become more digital in their realization. Not only are these devices constructed with more digital circuitry, but also signals hitherto transmitted on analogue schemes have been modulated digitally. This mode of transmission requires a modem which will convert the baseband signal into a form suitable for transmission. Design of modulators/demodulators which convert between data streams and waveforms suitable for specific types of channel has accelerated in recent years, one such channel being h.f. radio.

This paper describes a modem of this type, which has been designed at the University of Kent for use specifically on voiceband channels at h.f, and also discusses methods of realization. The modem is fed by a serial data stream at 2·4 kbits per second, which it modulates into a 3 kHz baseband channel. In the receiver, after mixing down to baseband, the second half of the modem uses the incoming signal to synchronize, and demodulates it back into a serial data stream (Fig. 1). The modulation technique employed for this system is multi-channel four-phase differential p.s.k.,[1] both with and without pilot synchronization tones inserted in the band. Although other modulation schemes are under consideration to demonstrate the versatility of the modem, this technique is the one to be used at h.f. trials.

H.f. transmission and reception have special problems associated with them. This is because h.f. channels are usually ionospheric and therefore suffer from multi-path propagation and both man-made and natural interference, properties which can cause unpredictable loss of data and synchronization. Unless modem parameters such as data rate or bandwidth are altered, little can be done to prevent loss of data. Loss of synchronization on the other hand results in an additional increase in data errors which can to some extent be controlled. Hence synchronization and the approach for its implementation have been under careful scrutiny in the design of the demodulator.

Until quite recently, nearly all modems would have consisted largely of analogue circuitry with a digital interface to the data source or sink. Utilizing microprocessors enables the construction of modems which are completely digital with just an analogue interface to the communication channel. The most obvious advantage in this case is the increased versatility of the modem. Whereas before, to change modulation type would have needed a major reconstruction of the hardware, the microprocessor realization reduces the problem to a modification in the program which it executes.

2 Operation

In the modulator, incoming data are packed into bytes which are used two or four at a time to provide sixteen or

0033–7722/81/060299+03 $1.50/0

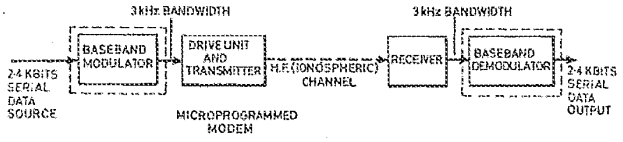


Fig. 1. Schematic diagram of the modem.

thirty-two channels of parallel information. These blocks of data are modulated by a repeating real-time programme with period τ equal to 1/16th or 1/32nd of the incoming serial data rate, into sixteen or thirty two parallel q.d.p.s.k. channels all placed side-by-side in the 3 kHz baseband. Each channel is separated from its neighbour by $2/\tau$ Hz and is also at a multiple of the frequency $2/\tau$. In the 16-channel case, eight carriers each at multiple of the frequency 300 Hz are phase modulated, carrying two bits of information on each of four 90° spaced phases (Fig. 2). In the thirty-two-channel case, sixteen carriers are modulated at a time, but the period τ is doubled too.

The individual carrier signals are generated from sine look-up tables, similar to those described in Ref. 2. These tables are sampled, scaled and summed, depending on the required frequency and phase, every 1/9600th of a second. 128 samples at each frequency of the carriers are derived from the tables at the requisite phases, and summed to obtain 128 samples for transmission. Another two or four bytes are taken from the incoming data stream and used to calculate the new phases for each carrier, so that the whole cycle may begin again. The resultant samples are clocked through a d/a converter to produce the baseband modulated waveform (Fig. 3).

The demodulator, which has to contend with synchronization and error decisions, is more complex than the modulator. (Error decisions consist of resolving the polarity of incoming data into its most likely state, and possibly implementing any error detection/correction that might have been coded into the data.) The noise-corrupted incoming signal is sampled by an a/d converter at 9·6 kbaud. Samples are used in a synchronization algorithm which is arranged to provide the start pulses to a Fast Fourier Transform (F.F.T.) routine. Output from this gives the phase and amplitude of each carrier, which may be compared with the previous phase and amplitude of the same carrier to regenerate the two bits of data.

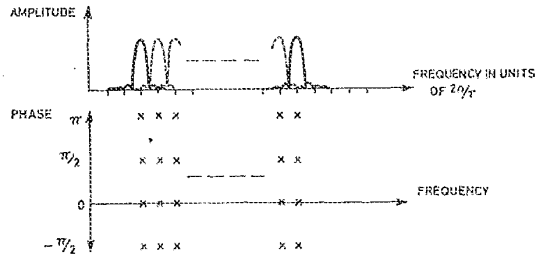


Fig. 2. Amplitude and phase spectrum for multi-channel 4-phase d.p.s.k.

Consider the sixteen carrier situation.

The incoming data from the a/d converter consist of amplitudes of an analogue waveform sampled at discrete intervals of 1/9600th of a second. Without noise, this analogue signal is a sum of sixteen sine waves of equal amplitudes at four possible discrete phases. At intervals of 1/32nd of the data rate (i.e. 75 Hz) the phase of each carrier might change by multiples of 90°, depending on the two new bits of data it carries. Assuming it is highly probable that at least one of the carriers will change phase at every discontinuity, it is possible to gain data synchronization from the phase transitions. An output from this synchronization is used to keep an F.F.T. in step with the incoming data. A double 64-point radix-2 F.F.T. routine[3,5] is applied to each block of 128 samples to produce two frequency domain samples for each carrier frequency. These are averaged and converted from complex coordinates to amplitude and phase coordinates from which not only the data may be determined, but also the rate of fading of the incoming signal and the frequency/phase shift caused by h.f. interference.

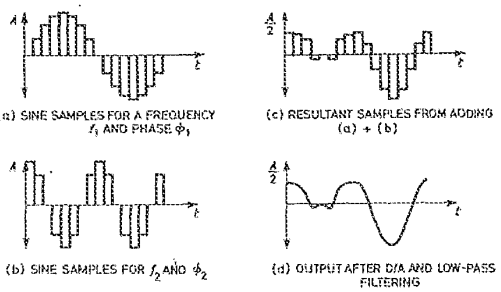


Fig. 3. Example of sine sample summation for two carriers.

Were the F.F.T. to take its 128 samples so that a phase discontinuity boundary was somewhere in the middle, the resulting data would be completely useless. In fact the errors rise fairly quickly with the number of samples at the wrong side of a phase transition, so it is essential that there is good data synchronization. This requires accurate data rate recovery from the incoming signal, which is achieved by a sliding filter algorithm in association with a local 'flywheel' clock. Whether this local clock or the generated synchronization pulses are used to synchronize the transform depends on the depth of fade or the frequency/phase error, as ascertained from previously decoded data blocks.

An additional technique is available for improving

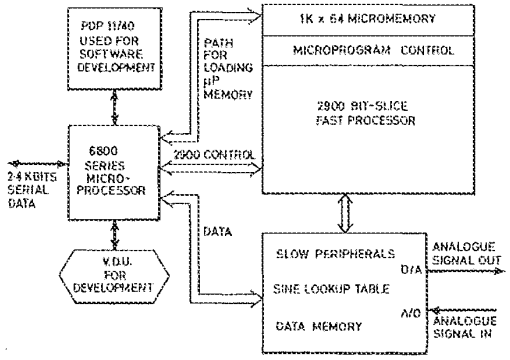


Fig. 4. Modem structure (block diagram).

synchronization, and for minimizing the possibility of performing an F.F.T. across a discontinuity. This involves spreading the carrier frequencies out so that they cover the complete bandwidth of the voice channel, rather than their frequencies being integral multiples of the data rate. The sampling frequency of the receiver is increased proportionately so that it is still an integral multiple of the carrier frequencies. However, the period between phase discontinuities in the transmitted signal remains a simple fraction of the data rate. Hence the period during which the 128 samples are taken for the F.F.T. is shorter than the time between discontinuities by approximately 16% for a data rate of 2·4 kHz in a 3 kHz bandwidth.[4] This means that there is a fairly long period of time, across the phase transition, over which no samples are taken for use in the F.F.T. This is advantageous for two reasons: (a) to allow a greater margin for synchronization error before trans-discontinuity samples cause errors in the F.F.T. algorithm, and (b) the F.F.T. does not employ samples near to the discontinuity where the 3 kHz bandlimiting causes 'rounding' of the signal on either side.

3 The Modem Structure

In both the modulator and the demodulator there are two microprocessors. A slower, one-chip microprocessor from the 6800 family is used to interface the modem to the serial data source or sink. Its responsibility is for the slower data processing, such as packing the incoming serial data into bytes and encoding it, some of the synchronization mechanism in the demodulator, and the control functions for the fast processor. (Fig. 4.)

This fast processor consists of a 2900-series bit-slice microprocessor to perform the modulation and demodulation of data, and is connected directly to the analogue port via its data bus. Its purpose is to convert data to samples of summed sine waves at the correct phases in the modulator, and to perform the F.F.T. and clock recovery in the receiver. Since as a modulator/demodulator it is repeatedly executing a dedicated routine of known duration, there is no need for macro-coding and a mapping p.r.o.m. as in the conventional bit-slice machine.[6] Hence all programming is at the microcode level. Microcode is bootstrapped into the writable microcode memory on power-up by the 6800 processor, which in turn is fed by a host mainframe computer during microprogram development. For a completed portable modem, the bootstrapping is from e.p.r.o.m.s in the 6800's memory map.

All data/address buses on the bit-slice are 12 bits wide, together with the a/d, d/a converters, while the width of the microprogram word is 64 bits. Two-level pipelining and parallel hardware stacks, together with fast data paths and devices isolated from slow data buses by registers allow minimization of processor cycle times. The bit-slice machine is connected to the slow processor by an 8-bit bidirectional data register which is directly addressable in the memory map of each machine.

4 Conclusions

This paper describes a modem which uses only digital processing to accomplish its operation. When used for h.f. trials the modem demonstrates the viability of microprocessor controlled modulation and demodulation. It also reveals its versatility to be reprogrammed with ease to a completely different modulation scheme.

5 Acknowledgments

S. D. Smith would like to acknowledge the support of the S.R.C. and of the Ministry of Defence (Procurement Executive); the authors are grateful for helpful discussions with Mr J. Pennington (A.S.W.E.)

6 References

1 Ziemer, R. E., and Tranter, W. H., 'Principles of Communication: Modulation and Noise', Sect. 7.5 (Houghton Mifflin, Boston, 1976).
2 Gorski-Popiel, J. (Ed.), 'Frequency Synthesis: Techniques and Applications' (IEEE Press, New York, 1975).
3 Rabiner, L. R., and Gold, B., 'Theory and Application of Digital Signal Processing' (Prentice Hall, Englewood Cliffs, N.J., 1975).
4 Riley, G. I., 'Error Control for Data Multiplex Systems', Ph.D. Thesis, University of Kent at Canterbury, 1975.
5 Brigham, O., 'The Fast Fourier Transform' (Prentice Hall, Englewood Cliffs, N.J., 1974)
6 'Build a Microcomputer', Advanced Micro Devices, Sunnyvale, Cal., 1979

Manuscript received by the Institution in final form on 27th March 1981 (Paper No. 1993/Comm 220)

A Novel use for Microprocessors in Designing Single and Multi-tone Generators, Ringing Generators and Inverters

Authors

Earl Rhyne
President
Permace Associates
7 Walnut Street
Millis, MA 02054

Ray Bennett
Engineering, Consultant
Permace Associates
21 Rickey Drive
Maynard, Ma 01754

ABSTRACT

This paper describes a method for producing precise single or multi-frequency tones for telephone office ringing generators, tone generators, and general purpose inverters, by using microprocessors and digital technology.

Recent technical developments have provided engineers new tools for generating the signals used for telephone equipment and for permitting remote access to the equipment for supervisory and diagnostic purposes. Figure 1 illustrates a system in which microcontrollers, counters,timers, Random Access Memory (RAM), Analog to Digital Converters (A/D), and Digital to Analog Converters (D/A) are combined to produce tone signals. These signals are then amplified to produce the required ringing or tone power.

The microcontroller is programmed with a mathematical equation to derive timing and voltage levels for the output signal. This equation is converted to digital words that are passed to the data portion of RAM and is converted to a digital word that sets a timer controlling the address portion to RAM. The RAM output is converted to an analog signal by the D/A converter. This signal is used by a power amplifier to condition the signal for use on the telephone lines.

By using a Microcontroller, the terms of the equation (i.e. frequencies and the voltage levels of each frequency independently) are input as variables, thus giving the user complete control of the output signal. The user can use a manual control (such as a keypad/readout) or a remote computer (through an RS232 port) to change the variables. The range of the signals is determined by the resolution of the D/A converter and the frequency response of the power amplifier. Because the Microcontroller is crystal controlled, frequency response and accuracy are a function of binary resolution. The output level is also a function of the binary resolution and reference voltage. Multiple frequency tones are generated by the same method. Since the quantizing frequency is much higher than the tone frequencies, simple low pass filtering is used to eliminate unwanted frequencies. Using an A/D converter, the output is monitored. This same signal may be used as a built-in diagnostic test.

The output of the power amplifier is sensed for voltage and current output. In low frequency applications, the microcontroller can monitor for inductive and capacitive loads, and make adjustments.

The versatility of the microcontroller allows a single design to cover a wide variety of uses. The power amplifier can customize the application. Other options such as zero crossing interrupting would be under control of the microcontroller.

INTRODUCTION

Currently Ringing and Tone generators consists of analog devices such as oscillators and linear amplifiers, or non-adjustable digital oscillators and bandpass filters to generate the required frequency and waveshapes. Some of the more important analog design considerations are signal linearity, symmetry, frequency stability, and temperature variations that effect all of the above. Frequency is derived from standard oscillator circuits, which contain resistors, capacitors, and/or inductors. In adjusting the frequency, fine tuning is done with variable resistors, while more coarse adjustments are done by switching capacitors and/or inductors. The frequency selective components used in ringing generators have large values and are physically large.

Some of the more important digital design elements are the master clock and count down circuits. The use of digital circuits usually solves the problem of stability and symmetry but introduces some filtering requirements because digital signals are square waves which, by definition, are rich in harmonics. Proper filter design reduces harmonics to the desired output level. Digital filters at ringing frequencies contain large capacitance and/or inductance values and, like the analog oscillators, are physically large. A resonant transformer may be used as a filter but it is physically large, and can only work with a single frequency. The filters for tones need to have high Q values in order to suppress the harmonics below the DTMF band requirements. In the current technology, interrupters are not synchronized to the tone wave shapes being interrupted.

Today's technology allows the use of Microcontrollers to generate signals. Microcontroller generate the required data and D/A converters transform the digital words to analog signals, eliminating the need for RC or LC oscillators. All the required functions are executed in firmware which calculates the sine functions for magnitude and duration of wait statements necessary to obtain the frequency. Since the microcontroller is driven by a crystal oscillator, frequency stability can be as good as a standard quartz watch (in the order of .002%). Frequency and voltage adjustments can be made by recalculating Microcontroller data. The user may input the data in many ways, such as a key pad, selector switches, or it may be down loaded via a RS232 computer/modem port.

DIGITAL SYNTHESIS

Ringing and tone analog signals are independent, continuous signals varying as a function of time. Digital signals are discrete time varying signals. A digital signal is a sequence of numbers[1].

CH2928-0/90/0000-0611$01.00 ©1990 IEEE

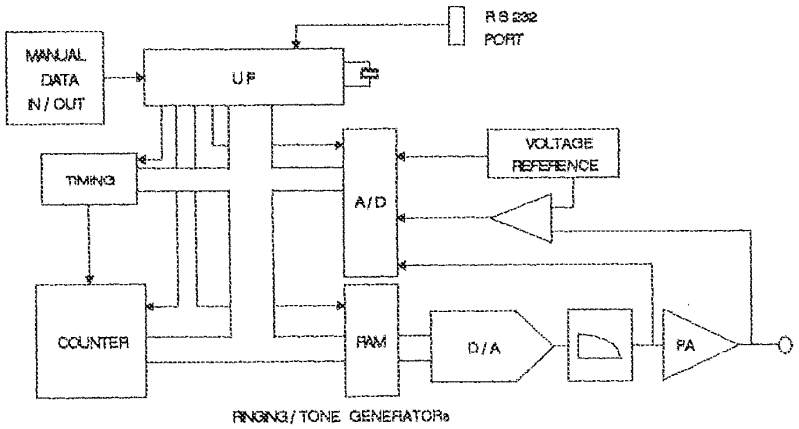


Figure 1

Each digital number represents a time variant value of an analog signal called sampled data. Sampled data is a discrete value for a sampled time. The following sampled data words are analog signals at different values in both time and magnitude. This quantized signal takes on only those values specified by the quantized levels. The quantized signal differs from the analog signal by the number of individual points taken during the duration of the analog signal. The more points taken over a given period of time, the smaller are the errors introduced by the digitizing of the signal.

Ringing and tones are repetitive, time varying signals, making their mathematical models quite simple. They lend themselves to simple calculations. Allowing the Microcontroller to generate discrete samples of data over an integral number of one or more repetitive "tone cycles".

To convert the mathematical numbers to an analog signal, a D/A converter is used. Its input is a digital word and its output is a voltage level corresponding to that word. As each new different word is applied to the D/A converter the output varies accordingly. The result is a time varying signal; but this signal is not continuous because of the quantized affect.
See figure 2.

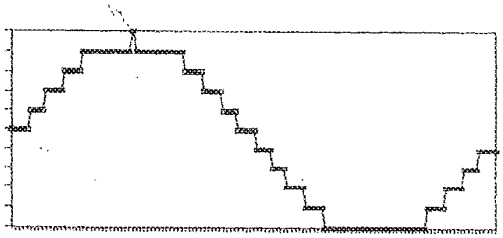


Figure 2.

The computer generated mathematical formula is turned into a digitized analog signal. This analog signal has quantized error terms that cause distortion with respect to an analog generated signal. If the correct number of points are chosen the error terms are low. These error terms contain high frequency signals that are easily filtered out with a simple bandpass filter. This process yields a signal that is amplified and applied to the telephone circuits.

Dial tone and ringback tone are produced by adding together two precise frequency tones. Ringback tones comprise 440 Hz and 480 Hz; dial tones comprise 350 hz and 440 Hz. The "Tone Cycle" is defined as follows: with all frequencies starting at zero degrees phase angle (zero volts and zero current), a "Tone Cycle" is complete when all the tones arrive at a 360° phase angle at the same time (i.e. zero volts). For a single frequency tone, quantized data is calculated for only one 360° cycle. For ringback tone, the computed data includes eleven complete 360° cycles of 440 Hz with twelve complete 360° cycles of 480 Hz to complete one "Tone Cycle". For dial tone, the computed data includes thirty five complete 360° cycles of 350 Hz with forty four complete 360° cycles of 440 Hz to complete one "Tone Cycle". After incrementing and transferring one "Tone Cycle" to the D/A converter, the Microcontroller resets to the start of the "Tone Cycle" and repeats.

Because the error terms in the digital generated dual tones are high frequency, a lowpass filter will leave only the two fundamental frequencies to be linearly amplified and distributed to the telephone circuits.
See figure 3.

Equation 1 produces a single sine wave frequency.

$$e = E*\sin(w*tq) \quad (1)$$

Where e = output signal
E = peak output voltage
w = 2*pi*f
tq = quantized time
tq = (1/f)/points

Equation 2 produces dual sine wave tones.

$$e = E_1*\sin(w_1*tq) + E_2*\sin(w_2*tq); \quad (2)$$

Where
E_1 = peak output of f_1
E_2 = peak output of f_2
$w_1 = 2*pi*f_1$
$w_2 = 2*pi*f_2$

The quantized time is

$$tq = (1/f_1)/points$$

Example:

If 512 data points are chosen and a ringing frequency of 20Hz is needed then,

$f = 20Hz$

$tq = (1/20)/512 = 98us.$

$fq = 1/98E-6 = 10.2Khz.$

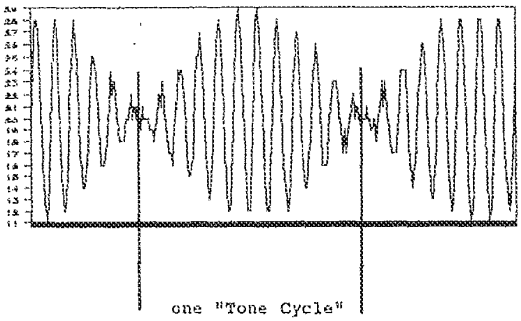


Figure 3

This quantizing frequency is twice the upper limit of the DTMF band. Filtering 10 Khz to a level of 50 dB below the fundamental can be done by a typical Chebyshev filter.

A simple listing for a single frequency is as follows:

```
 /* GET VARIABLES */
input frequency, f1;
input voltage, E1;

/* DEFINE CONSTANTS
 two_pi = 2*pi */
two_pi=6.283185307;

/* CALCULATE VARIABLES */
t1=((1/f1)/512) - overhead;
Mloc=0;
E = E1*10;

/* MAIN */
while (Mloc<=512)
   {
/* GET NEXT LOCATION */
     t=t1*Mloc;
/* OUTPUT DATA */
   P1= E*sin(two_pi*f1*t);
/* LOAD TIMER */
   TL0 = low(TICK);
   TH0 = high(TICK);
   TR0 = 1;
/* INCREMENT COUNTER */
   Mloc++;
/* START TIMER */
   while (TF0=0); TF0 = 0;
/* WAIT FOR TIMER & RESET*/
   };      /* END MAIN */
```

The value t1 is the time between quantized points. P1 is the output port attached to the D/A. Overhead is the time it takes the microcontroller to execute the instructions.

DIGITAL SYNTHESIZER

We have now identified one mathematical approach to produce a single frequency signal. One way to implement this in hardware is to use microcontrollers. The microcontroller contains timers and RAM. The instructions, also referred to as firmware, are contained in a ROM connected to the microcontroller's data bus. One data port of the controller is directly connected to the D/A converter, and the D/A converter is connected to the filter. See figure 4.

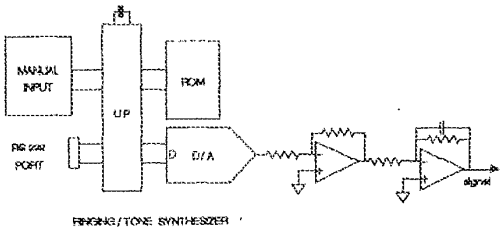


Figure 4

Installed in firmware are the calculations for a normalized sine function and the normalized timing functions. The input parameters of frequency and output level are independent functions, and are not in the main loop that generates the output signal. The input parameters are used by the calculator portion of the main loop that produces the output signal. From this, a signal containing the programmed frequency(s) and output level(s) is produced. The frequency is converted to time and is then loaded into one of the microcontroller's timers. The output value is calculated using the D/A parameters along with the loss of the filter and the gain of the output amplifier.

The firmware then takes the first normalized value from the sine calculation and algebraically adds the value of the programmed output voltage, sets the timer and passes this value to the D/A for conversion to output analog signal. The next step in the sequence, increments the count and repeats the

function. The repeated functions are put on the data bus when time t1 has elapsed. To insure t1 is strictly adhered to, the timer will interrupt the controller at the end of its programmed time. The analog value is zero and the address counter is reset to zero when the count reaches it's limit; then the cycle repeats until the controller is reset or powered down.

Multi tones may be generated by algebraically adding two computed sine functions together and passing the data to a single D/A converter. A second method is to use two independently generated sine functions and pass the individual data to two A/D converters, then sum the analog signals. This method requires the use of separate timers to keep track of the individual times of each sine function.

OUTPUT VALUES

The full scale value of a typical D/A is plus and minus five volts. In binary, the plus full scale for the D/A is 11111111 (FF in hex) and the minus full scale is 00000000 (00 in hex). The firmware must convert the value in the 128th (first peak value of the sine) location to FF (hex). The same conversion value is then used for each of the 512 sine values as they are to be loaded into the A/D.

Resolution is the function of the number of parallel data bits used by the D/A converter. For example, if an 8 bit D/A converter is used, the resolution is 1/256 times the full scale value. If the analog full scale value of the D/A converter is 5v then the resolution is equal to 5/256 or 19.5 millivolts. This is the smallest value of change allowed for this signal. The resolution is approximately 0.4%. By using a larger input D/A (i.e. 12 bits) the resolution is lowered to approximately 0.03%. The digital word for each analog value is calculated using equation 3.

$$V = N*R \qquad (3)$$

where

V = output volts
N = number of steps
R = resolution in volts

If the filter and amplifier have a combined gain of unity, the input parameters are the only multipliers.

Example:

To generate a 2.5v rms signal at the output of the 8 bit D/A converter,

Vrms/.707 = Vpk

2.5V/.707 = 3.54Vfs

2^8 = 256 steps FS

R = 5/256 = 19.53mv

N=V/R = 3.54/.0195 = 182

182 decimal is B6(hex).

Thus on the 128th count the D/A is to be loaded with B6 (hex).

Filtering is necessary because the output of the D/A converter still contains the quantizing frequency. A quantizing filter is a low pass filter with high enough value of Q to allow the desired signals to pass unattenuated, but attenuate the quantized frequency to a value at least 50 Db below the fundamental. A two or three stage Chebyshev filter is all that is required if the quantizing frequency is separated by two or three orders of magnitude.

The resulting signal is then treated as a standard analog signal and may be amplified by many means, such as a linear amplifier.

FEATURES

With a microcontroller calculating and generating the data for each step and count, it is possible to start and stop the signal on a data boundary (typically zero volts). It is also possible to modify the frequency and the output level by modifying data words, this allows for stable non-component dependent signals. If a host computer is connected to the generator it is possible for the microcontroller to collect operating data and diagnostic data (such as load peaks with respect to the time of day and active operating data). With the use of a battery backed up clock, the time of unscheduled interruption (failures etc.) may be logged, and the host computer may be used to trouble shoot the faulty equipment. Modems make it possible for remote sites to be monitored and data to be logged.

An interrupter can be included in the same package by adding appropriate hardware, and driving it by microcontrollers. Since microcontrollers are controlling both ringing generator and tone generator as well as the interrupter, it can synchronize them for zero voltage and zero current interruptions. Before the interrupter makes or breaks the signal, the microcontroller will first allow the output signal to finish its cycle to zero, then shut off the generator allowing the output voltage and current to go to zero. After waiting for transient settling, it opens or closes the interrupting relay. After the relay switching time has elapsed, the controller restarts the generator at zero phase angle and zero volts. A non-current breaking interruption has taken place.

CONCLUSIONS

Digital to analog technology is now a mature process and is supplemented with many pre-packaged circuits. The combining of functions within packages and the small physical sizes of the packages make them a viable solution to existing requirements. As the usage of these complex packages becomes more widespread, the prices reduce, and the variations increase. This gives today's designer a broad spectrum of ideas to chose from to make the telephone equipment more compatible with the present day technology.

REFERENCES

1. Lawrence R. Rabiner and Bernard Gold, "Theory and Application of Digital Signal Processing",Prentice-Hall, Inc. Englewood Cliffs, N.J.

Capacitive Impedance Readout Tactile Image Sensor

R. A. Boie

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT The transduction of mechanical forces to representative electrical signals uses a three layer sandwich structure. The top layer is columns of compliant metal strips over a central elastic dielectric sheet. The bottom layer is a flexible printed circuit board with rows of metal strips and multiplexing circuits. Electrically, the sensor is a capacitor array formed by the row and column crossings with the middle layer functioning as a dielectric spring. A readout of the capacitor values corresponds to a sampled tactile image.

The reasons for choosing this transduction method, the performance advantages of capacitive sensing and the design and integration of 64 element imagers into the fingers of a controlled compliance gripper are described.

1. INTRODUCTION

A review of touch or tactile sensor technology is given by Harmon[1]. Several sensor designs, including the one reported here, are based on measuring the thickness of an elastic layer compressed by the applied force. Resistive readout sensors of this class use conductive loading and obtain the pressure map by cross layer resistance measurement[2]. The method is inherently non linear and the materials exhibit poor elastic properties including hysteresis.

Cross layer capacitive impedance sensing is more favorable in many respects. The elastic materials need not be modified and desirable mechanical properties are generally consistent with low dielectric loss. Capacitive sensing is demonstrated to have marked advantage in terms of signal to noise ratio and measurement speed. The idea of force distribution sensing by capacitive readout and a study of suitable elastic/dielectric materials are presented in a comprehensive paper by Nicol[3]. The main contributions here are the development of a relationship for noise limited force resolution, illustrating the inherent performance of the sensing method, and the development of an appropriate robotic sensor.

2. CAPACITIVE SENSING

Figure 1 illustrates an exploded view of a sample robotic touch sensor. The topmost layer is a compliant glove that contacts objects and transmits via its elastic constant the contacting force distribution to the elastic/dielectric layer below. The lower layer is here shown rigidly supported by the printed circuit board. The glove and dielectric layer can be viewed as two springs in series under compression where the force information is obtained by measuring the displacement of the dielectric spring. The mechanical point-spread function of the glove can be narrowed, if desired, by suitably segmenting the glove material.

Orthogonal sets of conductive strips are arranged on the upper and lower surfaces of the elastic layer. A sampling of the layer thickness map is obtained by measuring the array of capacitors formed by the crossing areas, $A_{i,j}$, of row and column strips. The strip widths and spacing along with any point force spreading in the structure determine the spatial sampling and resolution. The time required to measure all capacitors determines the temporal sampling. The r.f. source, $V_0\cos(\omega_0 t)$ is connected to the lower set of strips via analog multiplexer "i". The multiplexer "j" connects pads to the amplifier input node. The pads are capacitively coupled to the upper strips via an inactive region of the elastic/dielectric layer. This contactless arrangement, due to Miller[4], is an important construction feature of this method. Cross talk signals are reduced by connecting the unselected strips and pads to ground potential. For each pair of multiplexer addresses (i,j) the r.f. source voltage is connected through the capacitance C(i,j) of strip i to strip j to the input node of the amplifier. (The strip to pad capacitance is arranged to be sufficiently large.) The output signal of the amplifier, V_A (i,j,t), is related to the strip to strip capacitance by,

$$V_A(i,j,t) = -V_D \frac{C(i,j)}{C_A} \cos \omega_0 t \qquad (1)$$

where C_A is the capacitance in feedback. C(i,j) is related to the localized layer thickness change by,

$$C(i,j) = \frac{K A}{\epsilon_0 (d_0 - x(i,j))} \qquad (2)$$

where A is the strips crossing area, K is the relative dielectric constant, ϵ_0 is the permittivity of vacuum, d_0 is the unloaded layer thickness. The local sampled force is described by the relationship,

$$F(i,j) = \lambda\, x(i,j) \qquad (3)$$

where λ is the dielectric/elastic layer spring constant.

The applied force is linearly related to measures of the reciprocal crossing capacitances with a constraint of fixed layer constants. Each crossing capacitance, independent of the shunt dielectric loss and series switch resistances, is measured in turn by phase sensitive detection during the interval T_m between sequential address advances.

Figure 2 illustrates the measurement method. The signal, V_A(i,j,t), is multiplied by the amplitude limited r.f. drive and integrated over the measurement interval, T_m. The time, T_m is synchronous with and has duration of m cycles of the r.f. drive. The integrator output is sampled and reset and the multiplexers address advanced at the end of each interval.

The sampled output is related to the strip i to strip j crossing capacitance by,

$$V_s(i,j) \propto V_d\ m\ \frac{C(i,j)}{C_A} \tag{4}$$

The force information is related to reciprocals of offset corrected capacitance measurements. Two direct reading readout methods were considered and may prove practical for some sensor designs. A conceptually simple method requires only the circuit location interchange of capacitors C(i,j) and C_A. All else remaining the same, the output provides a measurement of the crossing capacitive impedance. The impedance is linearly related to the displacement and, via the elastic constant, the force. This method requires a high performance input amplifier. The central difficulty is the large loop gain required for linear measurement response over a wide dynamic range. A more robust method is described in a paper on capacitive distance measurement[5].

3. NOISE, RESOLUTION AND DYNAMIC RANGE

Capacitive sensing of mechanical displacements is in most applications the method of choice. The low noise - high bandwidth properties of the method are well known, but little practiced. The method has the virtues of a parametric measurement, that is, the output signal is proportional to the displacement times the drive signal. Capacitors are non dissipative elements and so generate no noise. Capacitive sensing has not faired well in the robotics literature to date where it is described as inappropriate because of noise[6]. This misconception most likely results from confusing man-made interference, which can be reduced to negligible levels by proper shielding and connection, with intrinsic noise related to the basic nature of the detection process.

Figures 3 illustrate the equivalent circuits used for the performance analysis. Here a simpler receiver and filter are used to better illustrate the performance relationship. Figure 3a illustrates the strip crossing capacitance measurement. The r.f. drive or pump voltage, V_D, is connected to the input node of amplifier, A, via the crossing impedance. The peak output level of the filter with bandwidth Δf and center frequency ω_0 is the measure of the crossing capacitance and thereby the displacement of the dielectric/elastic and the force.

The diagram of Fig. 3b includes the significant parasitic circuit elements and the amplifier noise sources referred to its input. The resistances, R_{si} and R_{sj}, represent the multiplexers "on" resistances that appear as uncorrelated series noise sources. The resistor R(i,j) represents the dielectric loss, a parallel source. The generators e_n and i_n are the input equivalent series and parallel noise sources of the amplifier. The capacitors C_D, C_s and C_{gm} represent the parasitic elements of the sensor, wiring strays and the amplifier input, respectively. The noise sources and parasitic elements may be combined into equivalent noise resistances R_s and R_p and total shunt capacitance C_T, without loss of generality as shown in Fig. 3c.

The signal to noise relationship is developed in terms of the thickness change δx of the dielectric at a measured crossing. The differential signal output of the filter for a small displacement is;

$$\delta V_s = V_D\ \frac{C_0}{C_T}\ \frac{\delta x}{d_0} \tag{5}$$

where the displacement is described in relationship (2). A sensor array formed of N × N strips has parasitic capacitance C_D, which is by inspection of Fig. 4a proportional to the strip length.

$$C_D \propto N\ C_0 \tag{6}$$

The stray capacitances are not intrinsic to the design and can in practice be made relatively small. The sensor represents a capacitive source to the amplifier. The signal to noise ratio is optimized if the amplifier input element is physically scaled, while preserving its gain bandwidth product, so that C_{gm} and C_D have the same value[7]. The relationship for the optimized configuration is;

$$\delta V_s = \frac{V_D}{\alpha N}\ \frac{\delta x}{d_0} \tag{7}$$

where α is excess capacitance scaling constant. The signal improves linearly with the pump magnitude and degrades by the square root of the total number of array elements.

The mean square output signal of the uncorrelated series and parallel sources may be expressed as,

$$\overline{V}_n^2 = 4kT\ \Delta f\ \left\{ \frac{1}{\omega_0^2 C_T^2 R_p} + R_s \right\} \tag{8}$$

The first term in braces is due to the parallel source. The series term is usually dominant at the measurement frequencies and values of interest. The measurement bandwidth Δf is not of direct interest, more important is the array or frame rate F. That being the case the rms noise limited displacement resolution for a fully multiplexed sensor readout is given by,

$$\frac{\sigma_x}{d_0} = \alpha N^2 \left\{ \frac{\sqrt{4kT\ R_s\ F}}{V_D} \right\} \tag{9}$$

where σ_x is the r.m.s. displacement uncertainty. The term in braces represents the ratio of the series noise to the drive voltages. A conservative value of 1K Ohm for R_s, a drive of 10 volts and a framing rate 100 Hz yields a ratio value of 4×10^{-9}. This translates into a wide available dynamic range that may in turn be advantageously traded for relaxed layer requirements. Increasing the spring constant λ and thereby restricting the total fractional excursion, may help in reducing force dependent effects in the layer constants λ and K.

4. TACTILE IMAGING FINGERS

An 8 × 8 element tactile imager and its finger are shown in Fig. 4. The U shaped flexible circuit board is shown in the lower right of the photograph. The base of the U is the active region. The eight long strips are the driven elements and the eight short strips are the signal coupling pads. The short arm of the U supports the drive circuitry. The other supports the eight amplifiers, one for each pad, and the output multiplexer. The photograph also shows the finger structure and the assembly of the U shaped touch sensor band-aid on the robot finger. A view of the instrumented gripper is shown in Fig. 5. The low loss and backdriveable robot gripper mechanism was developed to support ultrasonic eye in the hand ranging and tactile imaging fingers with independent and variable gripping impedance[8]. The ultrasonic ranging system and the gripper control system are described elsewhere in these proceedings in papers by Miller[9] and Brown[10]. Pressures up to 50,000 dynes/cm^2 are sensed using a two thickness nylon stocking mesh elastic/dielectric layer. Each capacitor of the 64 element array is measured in turn by phase sensitive detection over eight cycles of a 200 KHz r.f. drive for a 390 Hz frame rate. Figures 6b and 6c show photographs of touch sensor raw data, $V_s(i,j)$, in response to touching a 1/4 inch diameter ball. Figure 6a shows the zero force offset image. The position directions "i" and "j" are indicated. Each of the 8 × 8 square areas shown correspond to strip crossings areas of 2.5 mm × 2.5 mm. The displacement out of the picture corresponds to increasing capacitance, C(i,j), and thereby sampled force, F(i,j). Figures 7a and 7b show touch images for the lead ends of an 8 pin dual in line package.

5. DISCUSSION

Capacitive sensing provides a robust and simple method of tactile imaging. The construction is straight forward and uses well behaved materials and catalog electronics. Structurally, the sensors are thin and conformable and are easily scaled. Static as well as dynamic images are sensed with a linear response. The temporal sampling can be made short relative to the mechanical response times of the robot system. The array readout need not be fully multiplexed, all rows may be measured during the time each column strip is driven. The spatial resolution is fundamentally limited only by strip lithography. If warranted, a 32×32 elements finger mounted single chip subsystem with composite video like output could be developed using current technology.

6. ACKNOWLEDGMENTS

I would like to thank G. L. Miller and R. A. Kubli for invaluable comments and suggestions.

References

1. Harmon, L.D., SME Technical Report MSR80-03, (1980)
2. Hillis, W.D., M.I.T. A.I. Memo 629, April (1981).
3. Nicol, K., Transducer Tempcon, (1981).
4. Miller, G. L., Private Communication, (1982).
5. G.L. Miller, R.A. Boie, P.L. Cowan, J.A. Golovchenko, R.W. Kerr, D.A.H. Robinson; A Capacitance-Based Micropositioning System for X-ray Rocking Curve Measurements; RSI, Vol. 50, No. 8, August 1979.
6. Harmon, L.D., SME Technical Report MSR82-02, (1982).
7. Gillespie, A. B., Signals, Noise and Resolution in Nuclear Counter Amplifiers. McGraw Hill. (1953).
8. Hogan, N., "Programmable Impedance Control of Industrial Manipulation", Proc. of Confr. on CAD/CAM Technology in Mechanical Engineering, M.I.T., Cambridge, Ma., March (1982).
9. G. L. Miller, R. A. Boie, M. Sibilia, "Active Damping of Ultrasonic Transducers for Robotic Application".
10. M. K. Brown, "Computer Simulation of Controlled Impedance Robot Hand".

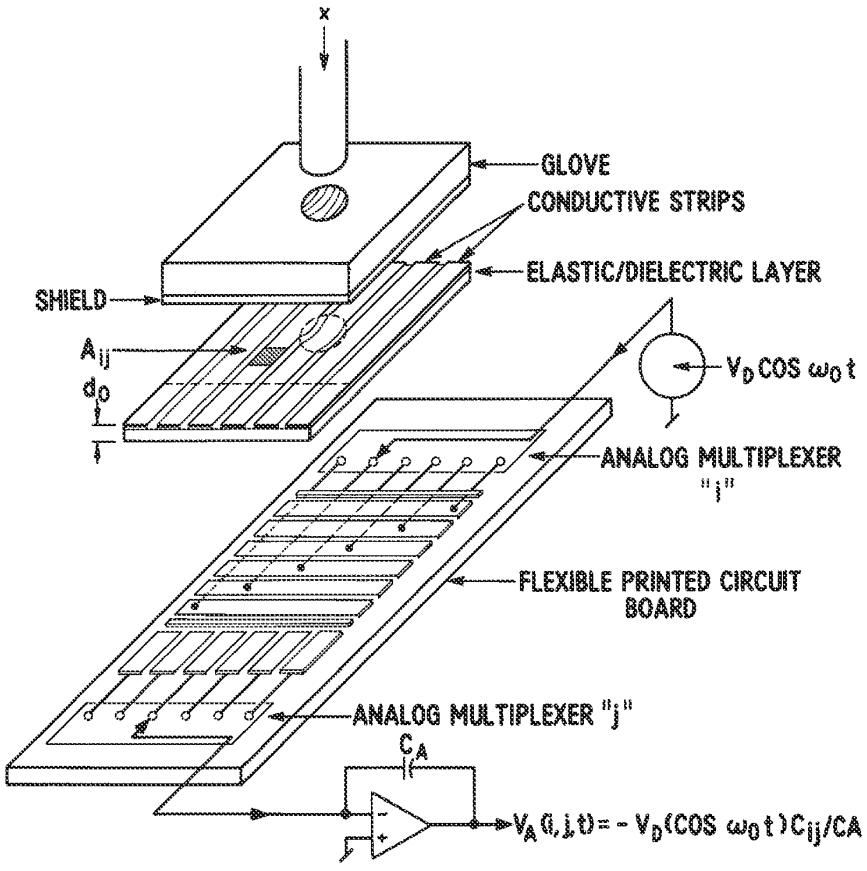


Fig. 1 Exploded view of a sample 6×6 element robotic touch sensor illustrating the layering and contactless construction. The force distribution is obtained by measurement of the cross dielectric/elastic layer capacitance.

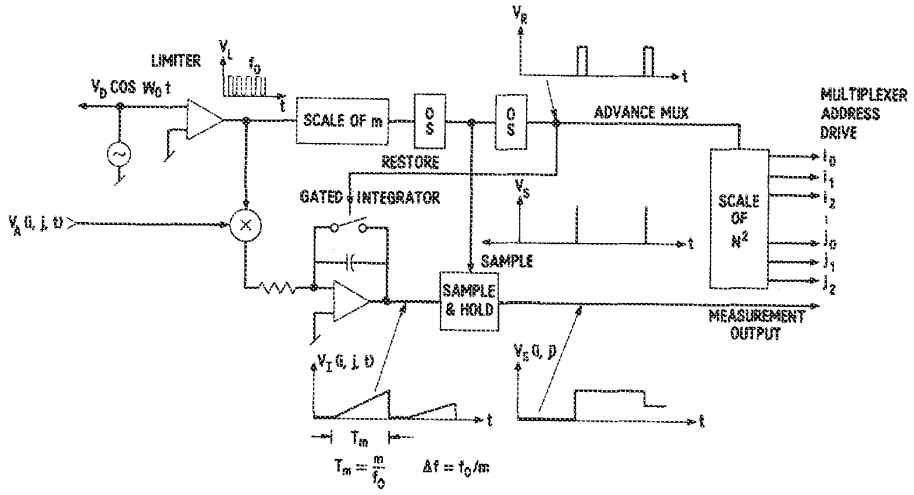
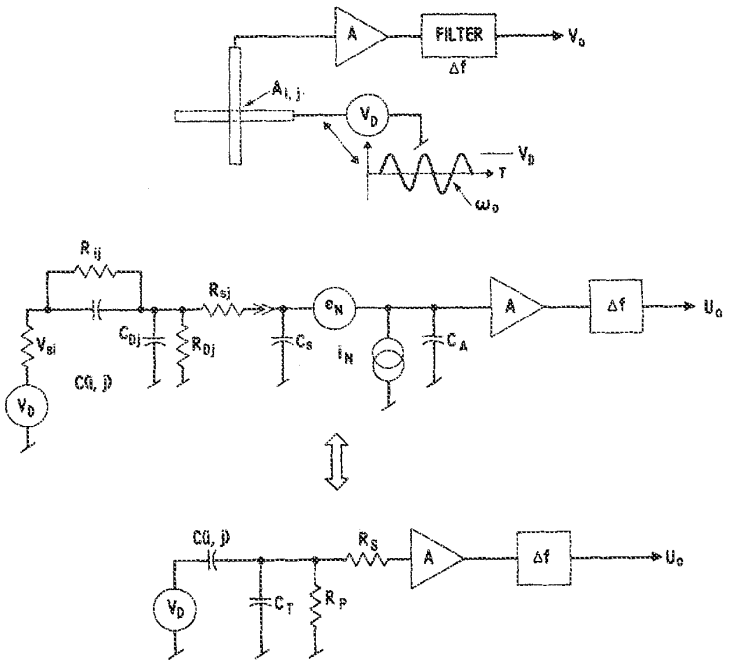


Fig. 2 Block diagram of the touch sensor control and phase sensitive receiver.



SIGNAL V_{OS}

$$CG, D \approx \frac{K}{\epsilon_o} \frac{A_{ij}}{d_o(i,j) - x(i,j)} \qquad C_T = \alpha N C_{ij} \qquad N^2 \text{ POSITION ELEMENTS}$$

$$\partial V_{os} \approx V_D \frac{C_o}{C_T} \frac{\partial_x}{d_o}$$

NOISE $\bar{V}_{ON}$ (RMS)

$$\bar{V}_{ON}^2 = \underbrace{4kT R_s \Delta f}_{\bar{V}_{ONS}^2} + \underbrace{\frac{4kT \Delta f}{R_p \omega_o^2 C_T^2}}_{\bar{V}_{ONP}^2} \approx \bar{V}_{ONS}^2$$

RESOLUTION

FRAME RATE $f_R \Longrightarrow \Delta f > N^2 f_R$

$$\frac{\sigma_x}{d_o} \approx \alpha N^2 \frac{\sqrt{4kT R_s f_R}}{V_D}$$

Fig. 3 Illustrations of equivalent circuits used in the performance analysis.

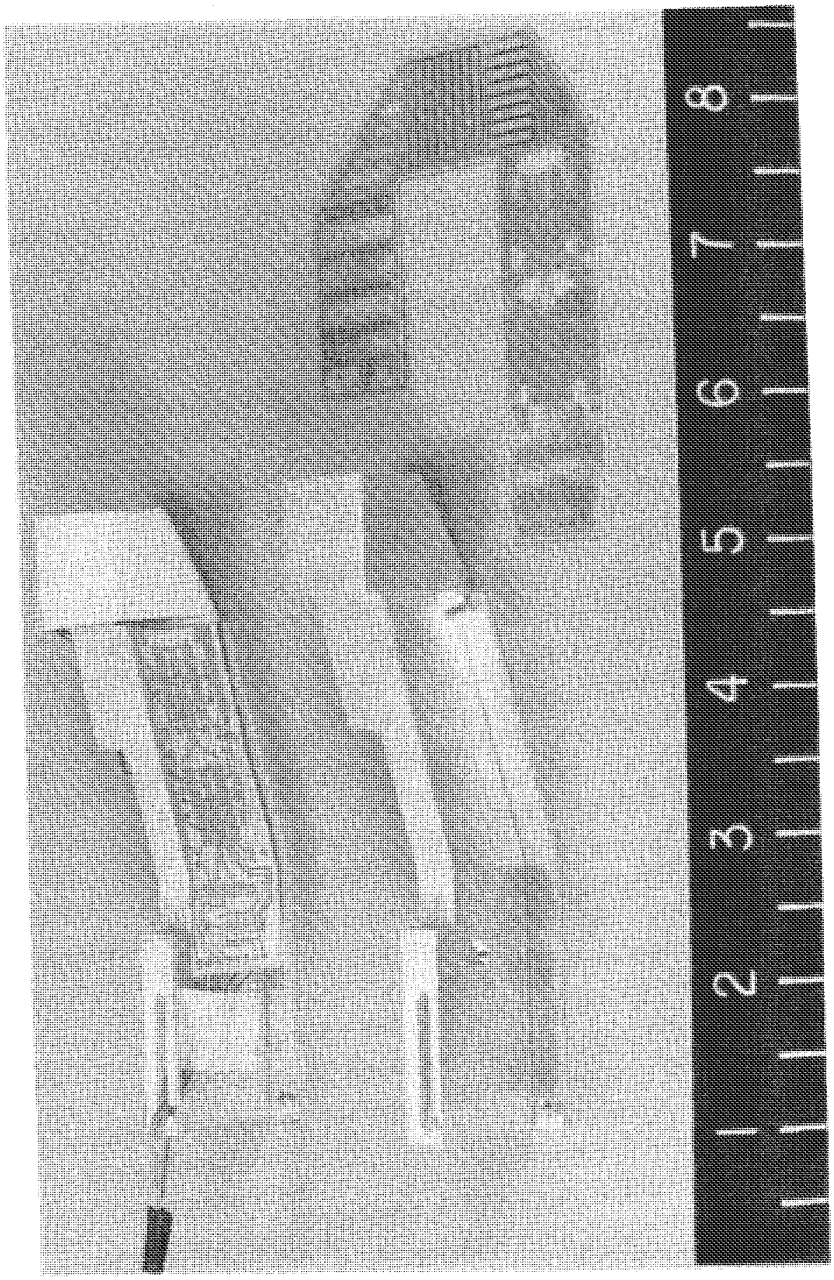


Fig. 4 Photograph showing the tactile sensor flexible printed circuit substrate, the robot finger structure and the active finger assembly.

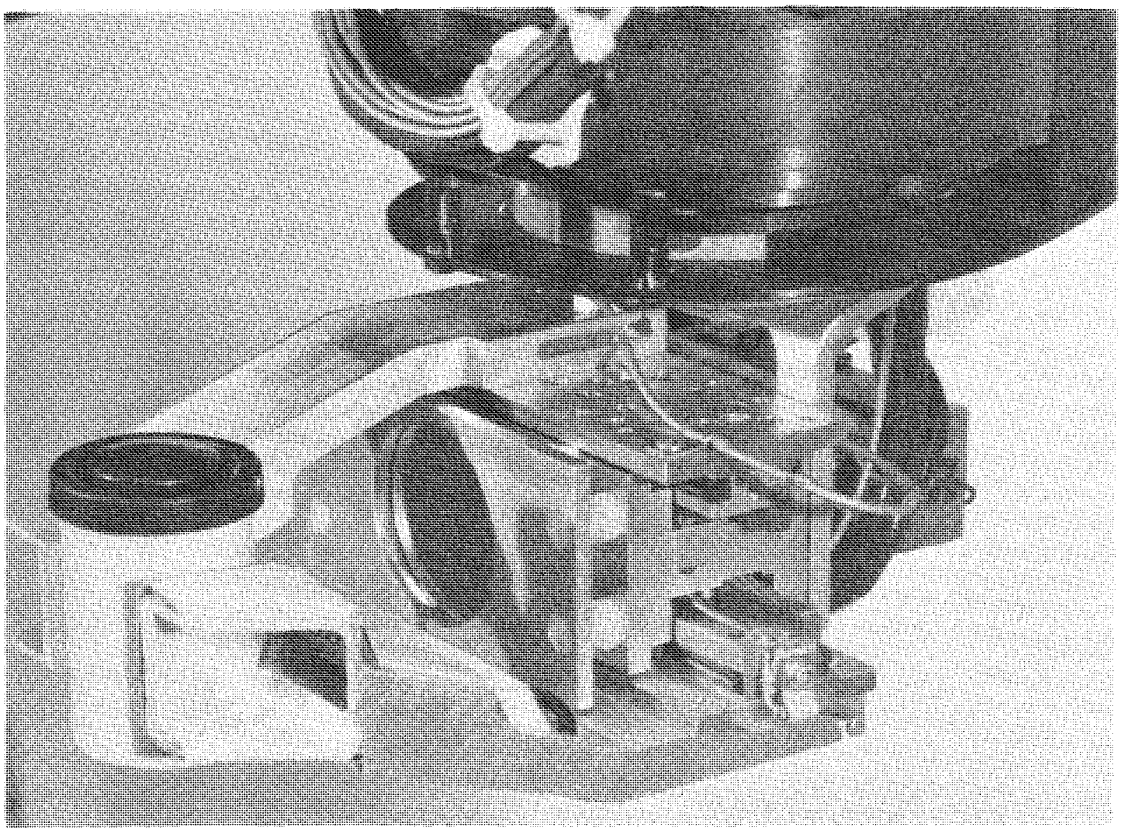


Fig. 5 A view of the instrumented gripper feeling a film container.

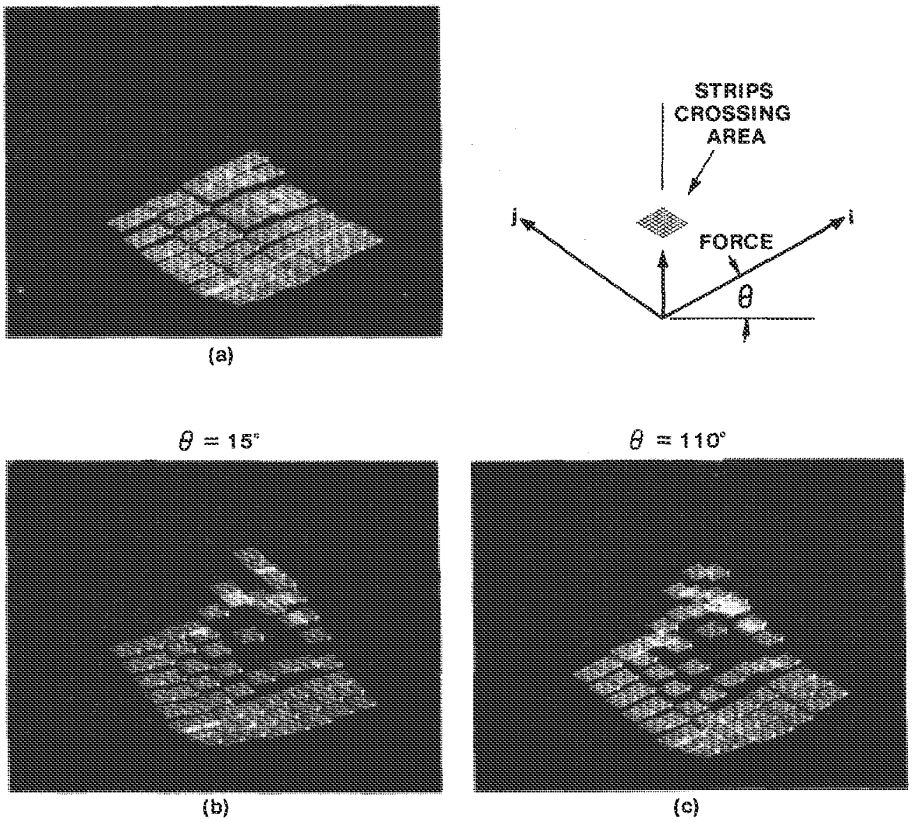


Fig. 6 Photographs of touch sensor raw data display showing zero force offset image and two views of touching a 1/4" diameter ball.

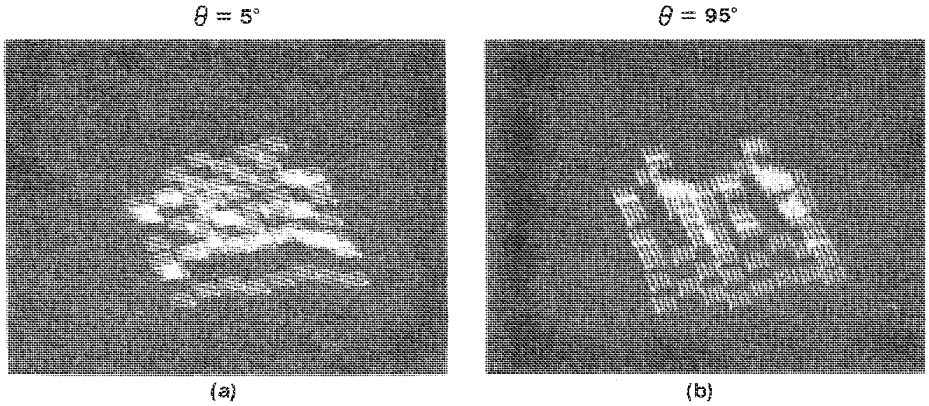


Fig. 7 Two views of touch data for lead ends of an eight pin dual in line package.

MAGAZINE
19.08

Clive Thompson on The Breakthrough Myth

By Clive Thompson ✉ July 26, 2011 | 12:00 pm | Wired August 2011

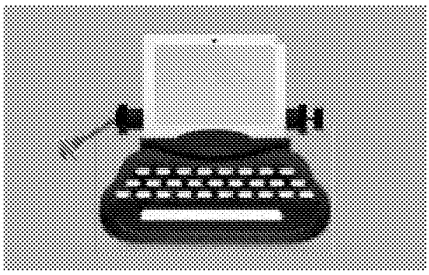


Illustration: Dev Gupta

Tech people love stories about breakthrough innovations—gadgets or technologies that emerge suddenly and take over, like the iPhone or Twitter. Indeed, there's a whole industry of pundits, investors, and websites trying feverishly to predict the Next New Big Thing. The assumption is that breakthroughs are inherently surprising, so it takes special genius to spot one coming.

But that's not how innovation really works, if you ask Bill Buxton. A pioneer in computer graphics who is now a principal researcher at Microsoft, he thinks paradigm-busting inventions are easy to see coming because they're already lying there, close at hand. "Anything that's going to have an impact over the next decade—that's going to be a billion-dollar industry—has always already been around for 10 years," he says.

Buxton calls this the "long nose" theory of innovation: Big ideas poke their noses into the world very slowly, easing gradually into view.

Can this actually be true? Buxton points to exhibit A, the pinch-and-zoom gesture that Apple introduced on the iPhone. It seemed like a bolt out of the blue, but as Buxton notes, computer designer Myron Krueger pioneered the pinch gesture on his experimental Video Place system in 1983. Other engineers began experimenting with it, and companies like Wacom introduced tablets that let designers use a pen and a puck simultaneously to manipulate images onscreen. By the time the iPhone rolled around, "pinch" was a robust, well-understood concept.

A more recent example is the Microsoft Kinect. Sure, the idea of controlling software just by waving your body seems wild and new. But as Buxton says, engineers have long been perfecting motion-sensing for alarm systems and for automatic doors in grocery stores. We've been controlling software with our bodies for years, just in a different domain.

This is why truly billion-dollar breakthrough ideas have what Buxton calls surprising obviousness. They feel at once fresh and familiar. It's this combination that lets a new gizmo take off quickly and dominate.

The iPhone was designed by Apple engineers who had learned plenty from successes and failures in the PDA market, including, of course, their own ill-fated Newton. By the time they added those pinch gestures, they'd made the obvious freshly surprising.

If you want to spot the next thing, Buxton argues, you just need to go "prospecting and mining"—looking for concepts that are already successful in one field so you can bring them to another. Buxton particularly recommends prospecting the musical world, because musicians invent gadgets and interfaces that are robust and sturdy yet creatively cool—like guitar pedals. When a team led by Buxton

developed the interface for Maya, a 3-D design tool, he heavily plundered music hardware and software. ("There's normal spec, there's military spec, and there's rock spec," he jokes.)

OK: If it's so easy to spy the future, what are Buxton's predictions? He thinks tablet computers, pen-based interfaces, and omnipresent e-ink are going to dominate the next decade. Those inventions have been slowly stress-tested for 20 years now, and they're finally ready.

Using a "long nose" analysis, I have a prediction of my own. I bet electric vehicles are going to become huge----specifically, electric bicycles. Battery technology has been improving for decades, and the planet is urbanizing rapidly. The nose is already poking out: Electric bikes are incredibly popular in China and becoming common in the US among takeout/delivery people, who haul them inside their shops each night to plug them in. (Pennies per charge, and no complicated rewiring of the grid necessary.) I predict a design firm will introduce the iPhone of electric bikes and whoa: It'll seem revolutionary!

But it won't be. Evolution trumps revolution, and things happen slowly. The nose knows.

Email clive@clivethompson.net.

Post Comment | Permalink

PREVIOUS
What Bandwidth Caps Would Mean for Internet Gluttons

NEXT
Rate This Article: What's Wrong with the Culture of Critique

RECENT ARTICLES

Storytown: Mark Norton Talks Vampires and Writing at Comic Con

Harry Potter, RIP

Five Gory Game Deaths

Surfer Geeks Build a Better Wave Pool

Soul-Crushing Realism is a Videogame Hit

Decode: Puzzles, games and harrowing mental torments

Wired Magazine RSS feed

RECENT ISSUES

19.08 - August 2011: Extreme Science
19.07 - July 2011: The Mental Machine
19.06 - June 2011: The Smartest Jobs
19.05 - May 2011: The Humor Issue
19.04 - April 2011: How To Make Stuff

SharperBike

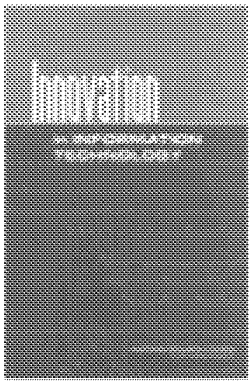
I believe the IPhone of electric bikes is already on the market and it is called the VeloMini folding electric bike. It is everything the Segway should have been as a transportation vehicle (12 miles and hour for 10 miles without pedaling) and you can purchase 7 for the price of a Segway, fold them up and put them all in a small SUV Two will fit in the trunk of a Prius. They come in IPod colors and are used by students, commuters, seniors, as well as boat, RV and private plane owners.

1 week ago Like Reply

ArizonaRider

We feel it happening. Pedego Electric Bike sales are soaring!

1 week ago Like Reply



Innovation in Information Technology

National Research Council
ISBN: 0-309-52622-1, 84 pages, 6x9, (2003)

Innovation in INFORMATION TECHNOLOGY

Computer Science and Telecommunications Board

Division on Engineering and Physical Sciences

NATIONAL RESEARCH COUNCIL
OF THE NATIONAL ACADEMIES

THE NATIONAL ACADEMIES PRESS
Washington, D.C.
www.nap.edu

THE NATIONAL ACADEMIES PRESS 500 Fifth Street, N.W. Washington, DC 20001

NOTICE: The projects that are the basis of this synthesis report were approved by the Governing Board of the National Research Council, whose members are drawn from the councils of the National Academy of Sciences, the National Academy of Engineering, and the Institute of Medicine. The members of the committees responsible for the final reports of these projects and of the board that produced this synthesis were chosen for their special competences and with regard for appropriate balance.

Support for this project was provided by the core sponsors of the Computer Science and Telecommunications Board (CSTB), which include the Air Force Office of Scientific Research, Cisco Systems, Defense Advanced Research Projects Agency, Department of Energy, Intel Corporation, Microsoft Research, National Aeronautics and Space Administration, National Institute of Standards and Technology, National Library of Medicine, National Science Foundation, and Office of Naval Research. Sponsors enable but do not influence CSTB's work. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the organizations or agencies that provide support for CSTB.

International Standard Book Number 0-309-08980-8 (book)
International Standard Book Number 0-309-52622-1 (PDF)

Copies of this report are available from the National Academies Press, 500 Fifth Street, N.W., Lockbox 285, Washington, DC 20055, (800) 624-6242 or (202) 334-3313 in the Washington metropolitan area; Internet: http://www.nap.edu.

Printed in the United States of America

THE NATIONAL ACADEMIES

Advisers to the Nation on Science, Engineering, and Medicine

The **National Academy of Sciences** is a private, nonprofit, self-perpetuating society of distinguished scholars engaged in scientific and engineering research, dedicated to the furtherance of science and technology and to their use for the general welfare. Upon the authority of the charter granted to it by the Congress in 1863, the Academy has a mandate that requires it to advise the federal government on scientific and technical matters. Dr. Bruce M. Alberts is president of the National Academy of Sciences.

The **National Academy of Engineering** was established in 1964, under the charter of the National Academy of Sciences, as a parallel organization of outstanding engineers. It is autonomous in its administration and in the selection of its members, sharing with the National Academy of Sciences the responsibility for advising the federal government. The National Academy of Engineering also sponsors engineering programs aimed at meeting national needs, encourages education and research, and recognizes the superior achievements of engineers. Dr. Wm. A. Wulf is president of the National Academy of Engineering.

The **Institute of Medicine** was established in 1970 by the National Academy of Sciences to secure the services of eminent members of appropriate professions in the examination of policy matters pertaining to the health of the public. The Institute acts under the responsibility given to the National Academy of Sciences by its congressional charter to be an adviser to the federal government and, upon its own initiative, to identify issues of medical care, research, and education. Dr. Harvey V. Fineberg is president of the Institute of Medicine.

The **National Research Council** was organized by the National Academy of Sciences in 1916 to associate the broad community of science and technology with the Academy's purposes of furthering knowledge and advising the federal government. Functioning in accordance with general policies determined by the Academy, the Council has become the principal operating agency of both the National Academy of Sciences and the National Academy of Engineering in providing services to the government, the public, and the scientific and engineering communities. The Council is administered jointly by both Academies and the Institute of Medicine. Dr. Bruce M. Alberts and Dr. Wm. A. Wulf are chair and vice chair, respectively, of the National Research Council.

www.national-academies.org

COMPUTER SCIENCE AND TELECOMMUNICATIONS BOARD

---

*Term ended June 30, 2001.
**Term ended June 30, 2002.

JON EISENBERG, Senior Program Officer
RENEE HAWKINS, Financial Associate
PHIL HILLIARD, Research Associate
MARGARET MARSH HUYNH, Senior Project Assistant
ALAN S. INOUYE, Senior Program Officer
HERBERT S. LIN, Senior Scientist
LYNETTE I. MILLETT, Program Officer
DAVID PADGHAM, Research Associate
CYNTHIA A. PATTERSON, Program Officer
JANICE SABUDA, Senior Project Assistant
BRANDYE WILLIAMS, Staff Assistant
STEVEN WOO, Dissemination Officer

NOTE: For more information on CSTB, see its Web site at <http://www.cstb.org>, write to CSTB, National Research Council, 500 Fifth Street, N.W., Washington, DC 20001, call at (202) 334-2605, or e-mail the CSTB at cstb@nas.edu.

Preface

The health of the computer science field and related disciplines has been an enduring concern of the National Research Council's Computer Science and Telecommunications Board (CSTB). From its first reports in the late 1980s, CSTB has examined the nature, conduct, scope, and directions of the research that drives innovation in information technology.

Ironically, the success of the industries that produce information technology (IT) has caused confusion about the roles of government and academia in IT research. And it does not help that research in computer science—especially research relating to software—is hard for many people outside the field to understand. This synthesis report draws on several CSTB reports, published over the course of the past decade, to explain the what and why of IT research. It was developed by members of the board, drawing on CSTB's body of work and on insights and experience from their own careers.

This synthesis is kept brief in order to highlight key points. It is paired with a set of excerpts from previous reports, chosen either for their explanation of relevant history or for their compelling development of core arguments and principles.

David D. Clark, *Chair*
Computer Science and
Telecommunications Board

Acknowledgment of Reviewers

This report has been reviewed in draft form by individuals chosen for their diverse perspectives and technical expertise, in accordance with procedures approved by the National Research Council's Report Review Committee. The purpose of this independent review is to provide candid and critical comments that will assist the institution in making its published report as sound as possible and to ensure that the report meets institutional standards for objectivity, evidence, and responsiveness to the study charge. The review comments and draft manuscript remain confidential to protect the integrity of the deliberative process. We wish to thank the following individuals for their review of this report:

Frederick P. Brooks, Jr., University of North Carolina at Chapel Hill,
Linda Cohen, University of California at Irvine,
Samuel H. Fuller, Analog Devices Inc.,
Juris Hartmanis, Cornell University,
Timothy Lenoir, Stanford University,
David G. Messerschmitt, University of California at Berkeley,
Ivan E. Sutherland, Sun Microsystems Laboratories, and
Joseph F. Traub, Columbia University.

Although the reviewers listed above have provided many constructive comments and suggestions, they were not asked to endorse the conclusions or recommendations, nor did they see the final draft of the report before its release. The review of this report was overseen by John

Hopcroft, Cornell University. Appointed by the National Research Council, he was responsible for making certain that an independent examination of this report was carried out in accordance with institutional procedures and that all review comments were carefully considered. Responsibility for the final content of this report rests entirely with the authoring board and the institution.

Contents

Summary and Recommendations

Progress in information technology (IT) has been remarkable, but the best truly is yet to come: the power of IT as a *human enabler* is just beginning to be realized. Whether the nation builds on this momentum or plateaus prematurely depends on today's decisions about fundamental research in computer science (CS) and the related fields behind IT.

The Computer Science and Telecommunications Board (CSTB) has often been asked to examine how innovation occurs in IT, what the most promising research directions are, and what impacts such innovation might have on society. Consistent themes emerge from CSTB studies, notwithstanding changes in information technology itself, in the IT-producing sector, and in the U.S. university system, a key player in IT research.

In this synthesis report, based largely on the eight CSTB reports enumerated below, CSTB highlights these themes and updates some of the data that support them. Much of the material is drawn from (1) the 1999 CSTB report *Funding a Revolution: Government Support for Computing Research*,[1] written by both professional historians and computer scientists to ensure its objectivity, and (2) *Making IT Better: Expanding Information Tech-*

[1]Computer Science and Telecommunications Board, National Research Council. 1999. *Funding a Revolution: Government Support for Computing Research.* National Academy Press, Washington, D.C.

nology Research to Meet Society's Needs,[2] the 2000 CSTB report that focuses on long-term goals for maintaining the vitality of IT research. Many of the themes achieved prominence in (3) the 1995 CSTB report *Evolving the High Performance Computing and Communications Initiative to Support the Nation's Information Infrastructure*,[3] known informally as the Brooks-Sutherland report. Other reports contributing to this synthesis include (4) *Computing the Future: A Broader Agenda for Computer Science and Engineering* (1992),[4] (5) *Building a Workforce for the Information Economy* (2001),[5] (6) *Academic Careers in Experimental Computer Science and Engineering* (1994),[6] (7) *Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers* (2001),[7] and (8) *More Than Screen Deep: Toward Every-Citizen Interfaces to the Nation's Information Infrastructure* (1997).[8] In the text that follows, these reports are cited by number as listed, for easy reference, in Box 1.

Here are the most important themes from CSTB's studies of innovation in IT:

- *The results of research*
  - America's international leadership in IT—leadership that is vital to the nation—springs from a deep tradition of research (1,3,4).
  - The unanticipated results of research are often as important as the anticipated results—for example, electronic mail and instant messaging were by-products of research in the 1960s that was aimed at making it

[2]Computer Science and Telecommunications Board, National Research Council. 2000. *Making IT Better: Expanding Information Technology Research to Meet Society's Needs*. National Academy Press, Washington, D.C.

[3]Computer Science and Telecommunications Board, National Research Council. 1995. *Evolving the High Performance Computing and Communications Initiative to Support the Nation's Information Infrastructure*. National Academy Press, Washington, D.C.

[4]Computer Science and Telecommunications Board, National Research Council. 1992. *Computing the Future: A Broader Agenda for Computer Science and Engineering*. National Academy Press, Washington, D.C.

[5]Computer Science and Telecommunications Board, National Research Council. 2001. *Building a Workforce for the Information Economy*. National Academy Press, Washington, D.C.

[6]Computer Science and Telecommunications Board., National Research Council. 1994. *Academic Careers in Experimental Computer Science and Engineering*. National Academy Press, Washington, D.C.

[7]Computer Science and Telecommunications Board, National Research Council. 2001. *Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers*. National Academy Press, Washington, D.C.

[8]Computer Science and Telecommunications Board, National Research Council. 1997. *More Than Screen Deep: Toward Every-Citizen Interfaces to the Nation's Information Infrastructure*. National Academy Press, Washington, D.C.

BOX 1
Reference Numbers for Key CSTB Titles Cited in This Report

Reference Number	*Title*
(1)	*Funding a Revolution: Government Support for Computing Research* (1999)
(2)	*Making IT Better: Expanding Information Technology Research to Meet Society's Needs* (2000)
(3)	*Evolving the High Performance Computing and Communications Initiative to Support the Nation's Information Infrastructure* (1995)
(4)	*Computing the Future: A Broader Agenda for Computer Science and Engineering* (1992)
(5)	*Building a Workforce for the Information Economy* (2001)
(6)	*Academic Careers in Experimental Computer Science and Engineering* (1994)
(7)	*Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers* (2001)
(8)	*More Than Screen Deep: Toward Every-Citizen Interfaces to the Nation's Information Infrastructure* (1997)

NOTE: Complete citations for these reports appear in footnotes 1 through 8 in this "Summary and Recommendations" section.

possible to share expensive computing resources among multiple simultaneous interactive users (1,3).

- The interaction of research ideas multiplies their impact—for example, concurrent research programs targeted at integrated circuit design, computer graphics, networking, and workstation-based computing strongly reinforced and amplified one another (1-4).

- *Research as a partnership*

- The success of the IT research enterprise reflects a complex partnership among government, industry, and universities (1-8).

- The federal government has had and will continue to have an essential role in sponsoring fundamental research in IT—largely university-based—because it does what industry does not and cannot do (1-8). Industrial and governmental investments in research reflect different

motivations, resulting in differences in style, focus, and time horizon (1-3,7,8).

- Companies have little incentive to invest significantly in activities whose benefits will spread quickly to their rivals (1,3,7). Fundamental research often falls into this category. By contrast, the vast majority of corporate research and development (R&D) addresses product and process development (1,2,4).
- Government funding for research has leveraged the effective decision making of visionary program managers and program office directors from the research community, empowering them to take risks in designing programs and selecting grantees (1,3). Government sponsorship of research especially in universities also helps to develop the IT talent used by industry, universities, and other parts of the economy (1-5).

• *The economic payoff of research*

- Past returns on federal investments in IT research have been extraordinary for both U.S. society and the U.S. economy (1,3). The transformative effects of IT grow as innovations build on one another and as user know-how compounds. Priming that pump for tomorrow is today's challenge.
- When companies create products using the ideas and workforce that result from federally sponsored research, they repay the nation in jobs, tax revenues, productivity increases, and world leadership (1,3,5).

The themes highlighted above underlie two recurring and overarching recommendations evident in the eight CSTB reports cited:

Recommendation 1 The federal government should continue to boost funding levels for fundamental information technology research, commensurate with the growing scope of research challenges (2-4,6-8). It should ensure that the major funding agencies, especially the National Science Foundation and the Defense Advanced Research Projects Agency, have strong and sustained programs for computing and communications research that are broad in scope and independent of any special initiatives that might divert resources from broadly based basic research (2,3).

Recommendation 2 The government should continue to maintain the special qualities of federal IT research support, ensuring that it complements industrial research and development in emphasis, duration, and scale (1-4,6).

This report addresses the ways that past successes can guide federal funding policy to sustain the IT revolution and its contributions to other fields.

1

Innovation in Information Technology

UNIVERSITIES, INDUSTRY, AND GOVERNMENT: A COMPLEX PARTNERSHIP YIELDING INNOVATION AND LEADERSHIP

Figure 1 illustrates some of the many cases in which fundamental research in IT, conducted in industry and universities, led 10 to 15 years later to the introduction of entirely new product categories that became billion-dollar industries. It also illustrates the complex interplay between industry, universities, and government. The flow of ideas and people—the interaction between university research, industry research, and product development—is amply evident.

Figure 1 updates Figure 4.1 from the 2002 CSTB report *Information Technology Research, Innovation, and E-Government*.[1] The originally published figure[2] produced an extraordinary response: it was used in presentations to Congress and to administration decision makers, and it was

[1]Computer Science and Telecommunications Board, National Research Council. 2002. *Information Technology Research, Innovation, and E-Government*. National Academy Press, Washington, D.C.

[2]Known informally as the "tire-tracks chart" because of its appearance, the figure was first published in *Evolving the High Performance Computing and Communications Initiative to Support the Nation's Information Infrastructure* (3; p. 2).

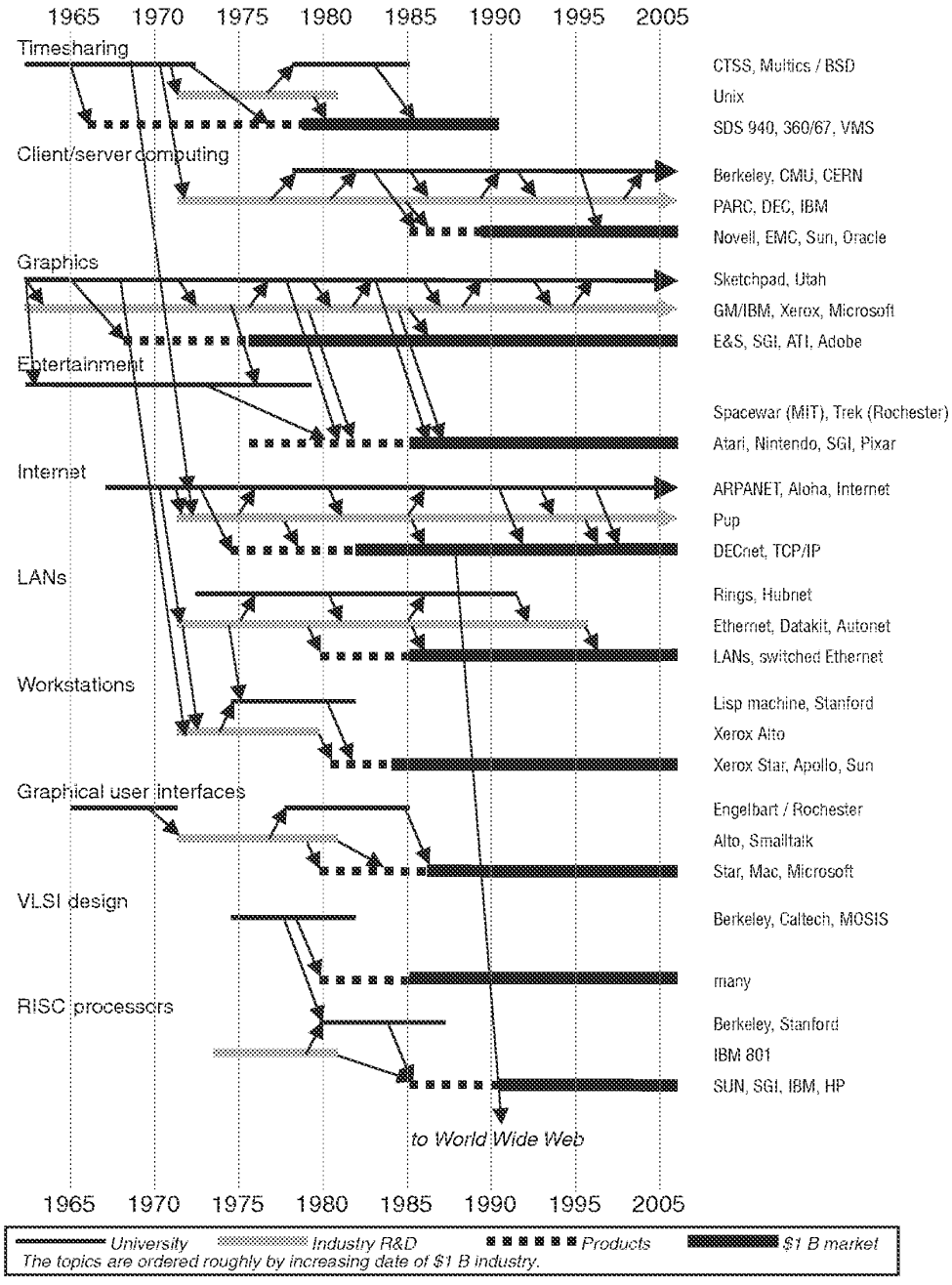
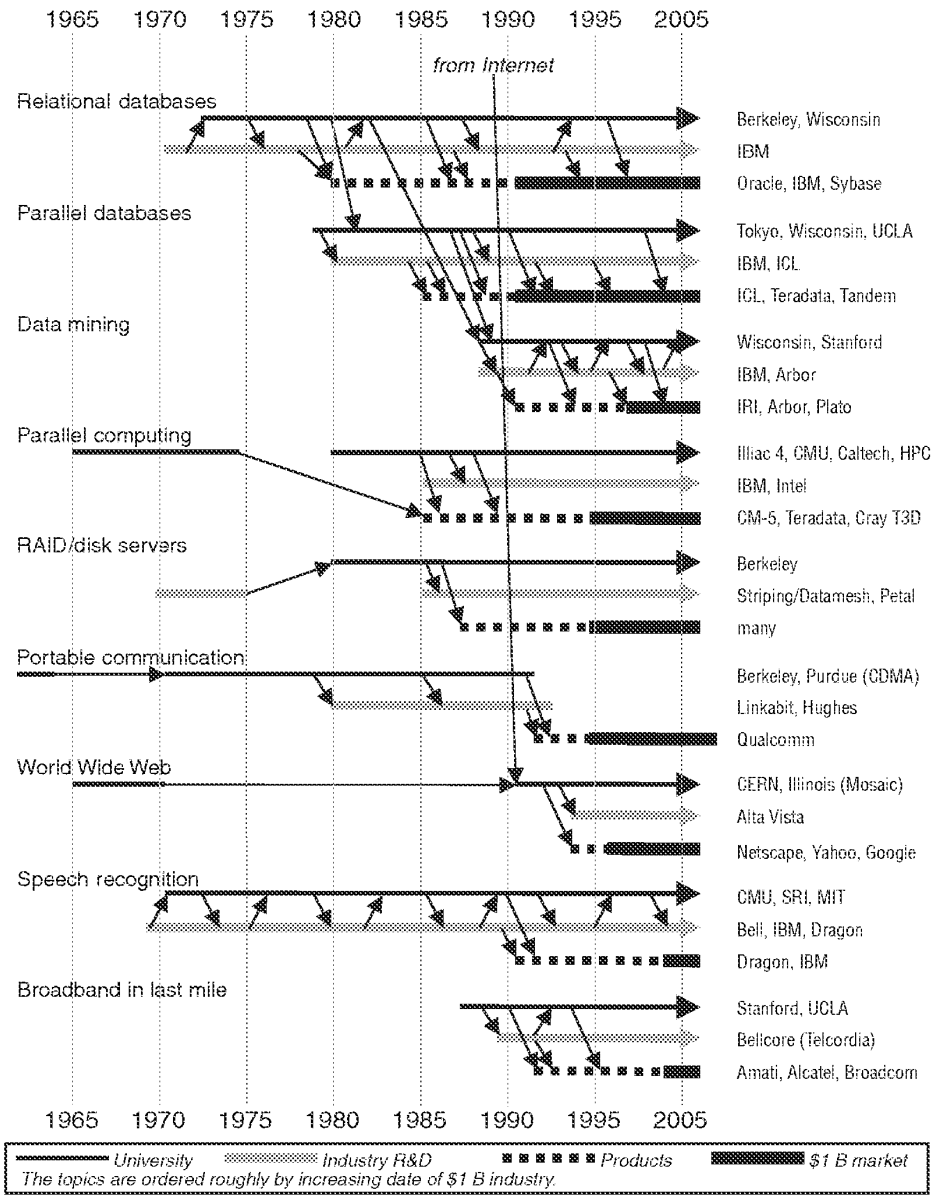


FIGURE 1 Examples of government-sponsored IT research and development in the creation of commercial products and industries. Federally sponsored research lies at the heart of many of today's multibillion-dollar information technology industries—industries that are transforming our lives and driving our economy. Ideas and people flow in complex patterns. The interaction of research ideas



multiplies their effect. The result is that the United States is the world leader in this critical arena. Although the figure reflects input from many individuals at multiple points in time, ensuring readability required making judgments about the examples to present, which should be seen as illustrative rather than exhaustive. SOURCE: 2002 update by the Computer Science and Telecommunications Board of a figure (Figure ES.1) originally published in Computer Science and Telecommunications Board, National Research Council, 1995, *Evolving the High Performance Computing and Communications Initiative to Support the Nation's Information Infrastructure,* National Academy Press, Washington, D.C.

discussed broadly in the research community. Although IT commercial success leads some policy makers to assume that industry is self-sufficient, the tire-tracks chart underscores how much industry builds on government-funded university research, sometimes through long incubation periods (1,3).

Figure 1 also illustrates—although sketchily—the interdependencies of research advances in various subfields. There is a complex research ecology at work, in which concurrent advances in multiple subfields—in particular within computer science but extending into other fields, too, from electrical engineering to psychology—are mutually reinforcing: they stimulate and enable one another.[3]

One of the most important messages of Figure 1 is the long, unpredictable incubation period—requiring steady work and funding—between initial exploration and commercial deployment (1,3). Starting a project that requires considerable time often seems risky, but the payoff from successes justifies backing researchers who have vision. It is often not clear which aspect of an early-stage research project will be the most important; fundamental research produces a range of ideas, and later developers select from among them as needs emerge. Sometimes the utility of ideas is evident well after they have been generated. For example, some early work in artificial intelligence has achieved unanticipated applicability in computer games, some of which are now being investigated for decision support and other professional uses as well as recreation.

It is important to remember that real-world requirements can change quickly. Although the end of the Cold War was interpreted by some as lessening the need for research,[4] September 11, 2001, underscored research needs in several areas: system security and robustness, automatic natural language translation, data integration, image processing, and biosensors, among others—areas in which technical problems are difficult to begin with, and may become harder when technology must be designed to both meet homeland security needs and protect civil liber-

[3]The idea that research in IT not only builds in part on research in physics, mathematics, electrical engineering, psychology, and other fields but also strongly influences them is consistent with what Donald Stokes has characterized in his four-part taxonomy as "Pasteur's Quadrant" research: use- or application-inspired basic research that pursues fundamental understanding (such as Louis Pasteur's research on the biological bases of fermentation and disease). See the discussion on pp. 26-29 in the 2000 CSTB report *Making IT Better* (2), and see Donald E. Stokes, 1997, *Pasteur's Quadrant: Basic Science and Technological Innovation*, Brookings Institution Press, Washington, D.C.

[4]Linda R. Cohen and Roger G. Noll. 1994. "Privatizing Public Research," *Scientific American* 271(3): 72-77.

ties.[5] Without fundamental research, the cupboard is bare when there is a sudden need for ideas to reduce to practice.

THE ESSENTIAL ROLE OF THE FEDERAL GOVERNMENT

Federally sponsored research played a critical role in creating the enabling technologies for each of the billion-dollar market segments illustrated in Figure 1—and for many others as well. The government role coevolved with IT industries: its organization and emphases changed to focus on capabilities not ready for commercialization and on new needs that emerged as commercial capabilities grew, both moving targets (1). As this coevolution shows, successful technology development relies on flexibility in the conduct of research and in the structure of industry.

Most often, this federal investment took the form of grants or contracts awarded to university researchers by the Defense Advanced Research Projects Agency (DARPA) and/or the National Science Foundation (NSF)—although a shifting mix of other funding agencies has been involved, reflecting changes in the missions of these agencies and their needs for IT (1,3). For example, the Department of Energy (DOE), the National Aeronautics and Space Administration (NASA), and the military services have supported high-performance computing, networking, human-computer interaction, and other kinds of research.[6]

Why has federal support been so effective in stimulating innovation in computing? As discussed below, many factors have been important.

1. *Federally funded programs have supported long-term research into fundamental aspects of computing, whose widespread practical benefits typically take years to realize* (1).

"Long-term" research refers to a long time horizon for the research effort and for its impact to be realized. Examples of innovations that required long-term research include speech recognition, packet radio, computer graphics, and internetworking. In every case illustrated in Figure 1, the time from first concept to successful market is measured in

[5]See Computer Science and Telecommunications Board, National Research Council. 2003. *Information Technology for Counterterrorism: Immediate Actions and Future Possibilities*. National Academies Press, Washington, D.C.

[6]In addition to research funding, complementary activities have been undertaken by other agencies, such as the National Institute of Standards and Technology, which often brings together people from universities and industry on issues relating to standards setting and measurement.

decades (see Box 2)—a contrast to the more incremental innovations that are publicized as evidence of the rapid pace of IT innovation.

Work on speech recognition, for example, which began in earnest in the early 1970s, took until 1997 to generate a successful product for enabling personal computers to recognize continuous speech (8). Work on packet radio also dates from the 1970s, and its realization in commercial ad hoc mobile networking also began in the late 1990s.[7] Fundamental algorithms for shading three-dimensional graphics images, which were developed with federal funding in the 1960s, saw limited use on high-performance machines until they entered consumer products in the 1990s; today these algorithms are used in a range of products in the health care, entertainment, and defense industries. The research programs behind these innovations not only were long-term but also were broad enough to accommodate within a single program the development of those unanticipated results that have in many cases provided the most significant outcomes of a project.

The benefits of a long time horizon, combined with program breadth, extend to today's challenges. This point was emphasized in CSTB's 1997 report on usability, *More Than Screen Deep* (8), which explained (at p. 192):

> Federal initiatives that emphasize long-term goals beyond the horizon of most commercial efforts and that may thus entail added risk have the potential to move the whole information technology enterprise into new modes of thinking and to stimulate discovery of new technologies for the coming century.

Because of unanticipated results and synergies, the exact course of fundamental research cannot be planned in advance, and its progress cannot be measured precisely in the short term. Even projects that appear to have failed or whose results do not seem to have immediate utility often make significant contributions to later technology development or achieve other objectives not originally envisioned. A striking example is the field of number theory (1): for hundreds of years a branch of pure mathematics without applications, it is now the basis for the public-key cryptography that underlies the security of electronic commerce.

[7]Similarly, commercial developments in broadband cellular radio (which has become essentially wireless Internet access in third-generation wireless) are built in part on many decades of federally supported research into Code Division Multiple Access technology, signal processing for antenna arrays, error-correction coding, and so on.

BOX 2
The Role of Federal Support for Fundamental Research in IT

CSTB's 1995 report *Evolving the High Performance Computing and Communications Initiative to Support the Nation's Information Infrastructure* (3) examined the payoff from several decades of federal investment in IT research. Among the conclusions of that report are these:

- *Research has kept paying off over a long period.*
- *The payoff from research takes time.* As Figure [1] shows, at least 10 years, more often 15, elapse between initial research on a major new idea and commercial success. This is still true in spite of today's shorter product cycles.
- *Unexpected results are often the most important.* Electronic mail and the "windows" interface are only two examples. . . .
- *Research stimulates communication and interaction.* Ideas flow back and forth between research programs and development efforts and between academia and industry [and between research programs with different foci that are proceeding concurrently].
- *Research trains people,* who start companies or form a pool of trained personnel that existing companies can draw on to enter new markets quickly.
- *Doing research involves taking risks.* Not all public research programs have succeeded or led to clear outcomes even after many years. But the record of accomplishments suggests that government investment in computing and communications research has been highly productive.[1]

---

[1]Computer Science and Telecommunications Board, National Research Council. 1995. *Evolving the High Performance Computing and Communications Initiative to Support the Nation's Information Infrastructure.* National Academy Press, Washington, D.C., pp. 3-4.

2. *The interplay of government-funded and industry research has been an important factor in IT commercialization* (1-8).

The examples in Figure 1 show the interplay between government-funded research and industry research and development. In some cases, such as reduced-instruction-set computing (RISC) processors, the initial ideas came from industry, but the research that was essential to advancing these ideas came from government funding to universities. RISC was conceived at IBM, but it was not commercialized until DARPA funded additional research at the University of California at Berkeley and at Stanford University as part of its Very Large Scale Integrated Circuit (VLSI) program of the late 1970s and early 1980s (1,3). The VLSI program also supported university research that gave rise to such companies as Synopsys, Cadence, and Mentor, which have acquired dozens of smaller

companies that started as spinoffs of DARPA-funded[8] university research; such research has also pushed the proverbial envelope in algorithms and user interfaces. The more than $3 billion electronic design automation industry is an essential enabler to other parts of IT.

Similarly, IBM pioneered the concept of relational databases (its System R project) but did not commercialize the technology. NSF-sponsored research at the University of California at Berkeley brought this technology to the point at which it was commercialized by several start-up companies and then by more established database companies (including IBM) (1,3). In other cases, such as timesharing, the initial ideas came from the university community, and subsequent industry research, while significant for a time, was not sustained. In none of the examples in Figure 1 did industry alone provide the necessary research.

3. *There is a complex interleaving of fundamental research and focused development* (1-3).

In the case of integrated circuit (VLSI) design tools, research innovation led to products and then to major industrial markets. A still-unfolding example is the theoretical research that yielded the algorithms behind the Web-content management technology underlying Akamai. In the case of relational databases, the introduction of products stimulated new fundamental research questions, leading to a new generation of products with capabilities vastly greater than those of their predecessors. The purpose of publicly funded research is to advance knowledge and to solve hard problems. The exploitation of that knowledge and those solutions in products is fundamentally important, but the form it takes is often unpredictable, as is the impact on future research (see Box 3).

4. *Federal support for research has tended to complement, rather than preempt, industry investments in research.*

The IT sector invests an enormous amount each year in R&D. It is critical to understand, however, that the vast majority of corporate R&D has always been focused on product and process development (2). This is what shareholders (or other investors) demand. It is harder for corporations to justify funding long-term, fundamental research. Economists

[8]In some cases, the Semiconductor Research Corporation provided the funding. For additional information, see the Web site <http://www.src.org/member/about/history.asp>. Accessed June 2, 2003.

BOX 3
The Technological Underpinnings of Electronic Commerce

Electronic commerce is becoming pervasive. It is changing many aspects of our lives, from the way we shop to the way we obtain government services.

The organizations and individuals that exploit electronic commerce employ commercial tools from companies such as Microsoft and Oracle. They may not think of themselves as the beneficiaries of federal investments in university-based IT research—but they are. Nearly every key technological component underlying electronic commerce has been shaped by this investment. For example:

- *The Internet*—Defense Advanced Research Projects Agency (DARPA) investments in the 1960s and 1970s were followed by National Science Foundation (NSF) investments in the 1980s and early 1990s, with research (supported by multiple agencies) continuing to this day (1).
- *Web browsers*—Mosaic, the first browser with a graphical user interface, was invented at the NSF-supported National Center for Supercomputer Applications at the University of Illinois (1).
- *Public-key cryptography for secure credit card transactions*—NSF sponsored university-based research in the 1970s that supported this innovation (1).
- *Back-end database and transaction processing systems*—NSF and DARPA supported key research on relational databases and transaction processing systems at the University of California at Berkeley, University of Wisconsin, and elsewhere, beginning in the early 1980s and continuing to this day (1).
- *Search engines*—Search engines grew out of federally supported university research programs, such as the ranking algorithm work at Stanford University that contributed to Google; the WebCrawler and MetaCrawler grew out of work at the University of Washington.

But the development is not complete: a range of technical challenges still exist, along with challenges for improving the fit between the technologies and the behavior and needs of the people who use them (2,8).

SOURCES: Pieces of this history are recounted in the previously cited CSTB reports (1-8) and in CSTB's series of reports on the Internet: *Toward a National Research Network* (1988), *Realizing the Information Future: The Internet and Beyond* (1990), *The Unpredictable Certainty: Information Infrastructure Through 2000* (1996), *The Internet's Coming of Age* (2001), and *Broadband: Bringing Home the Bits* (2002), all published by the National Academy Press, Washington, D.C.

have articulated the concept of "appropriability" to express the extent to which the results of an investment can be captured by the investor, as opposed to being available to all players in the market. The results of long-term, fundamental research are hard to appropriate for several reasons: they tend to be published openly and thus to become generally known; they tend to have broad value; the most important may be unpre-

dictable in advance; and they become known well ahead of the moment of realization as a product, so that many parties have the opportunity to incorporate the results into their thinking. In contrast, incremental research and product development can be performed in a way that is more appropriable: it can be done under wraps, and it can be moved into the marketplace more quickly and predictably.

Although individual industrial players may find it hard to justify research that is weakly appropriable, it is the proper role of the federal government to support this sort of endeavor (1,3). When companies create successful new products using the ideas and workforce that result from federally sponsored research, they repay the nation handsomely in jobs, tax revenues, productivity increases, and world leadership (1,3). Long-term research often has great benefits for the IT sector as a whole, although no particular company can be sure of reaping most of these benefits.

Appropriability helps to explain why the companies that have tended to provide the greatest support for fundamental research are large companies that enjoy dominant positions in their market (1). AT&T and IBM, for example, have historically made significant investments in fundamental research. Anything that advances IT as a whole benefits the dominant players—they may be capable of reaping a significant proportion of the returns on their research investments. As IT industries became more competitive, however, even these firms began to link their research more closely with corporate objectives and product development activities.[9] One of them (AT&T) has radically cut back its research effort. This process began with a government proceeding that resulted in the splitting up of functions formerly aggregated under "Ma Bell" and continued with the growth and contraction of a set of industry research and development endeavors (AT&T Research, Lucent Technologies, Agere Systems, and Bellcore [now Telcordia]) where once there was the monolithic Bell Laboratories.[10]

Several of the companies that have recently emerged as dominant in their sectors, such as Intel and Microsoft, have increased their support for fundamental research. However, many other successful companies with large market shares (e.g., Cisco, Dell, Oracle) have chosen not to invest in fundamental research to any significant extent. And even at Microsoft, just as at AT&T and IBM before it, the investment in fundamental research

[9]Elizabeth Corcoran, 1994, "The Changing Role of U.S. Corporate Research Labs," *Research-Technology Management* 37(4):14-20; Peter Coy, 1993, "R&D Scoreboard: In the Labs, the Fight to Spend Less, Get More," *Business Week*, June 28, pp. 102-124.

[10]CSTB launched a study of the future of telecommunications R&D in 2003.

represents a relatively small proportion of overall corporate R&D. In 2002, Microsoft invested roughly $5 billion in R&D, but the company's fundamental research arm is small enough to suggest that 95 percent of Microsoft's R&D investment is product-related.

Start-ups represent the other end of the spectrum. A hallmark of U.S. entrepreneurship, start-ups and start-up financing promote flexibility in industry structure and industry management. They have facilitated the development of high-risk products as well as an iconoclastic, risk-taking attitude among more traditional companies and managers in the IT business. *But they do not engage in research* (2). Thus, the wave of Internet-related and other IT start-ups of the 1990s is notable for two reasons: first, these start-ups attracted some researchers away from universities and research, and second, notwithstanding the popular labeling of those start-ups as "high-tech," they applied the fruits of past research rather than generating more. Start-ups illustrate the critical role of government funding in building the foundations for innovative commercial investments.

THE DISTINCTIVE CHARACTER OF FEDERALLY SUPPORTED RESEARCH

The most important characteristic of successful government research activities is their breadth of scope—both in their long time dimension and in their focus on activities that are potentially difficult to appropriate privately in their entirety. Two specific topic areas that illustrate these principles are large-scale IT systems and social applications of IT. Growing capabilities and broadening use of IT in the 1990s motivated CSTB recommendations for greatly increased federal support in these two categories (2) (see Boxes 4 and 5).

Prospects for progress in social applications—however difficult—are one reason for confidence that IT will improve as a human enabler. The beginnings evident in all of these areas are but crude indicators of what research may make possible.

An example of particular currency is that of cybersecurity. Stimulated by the events of September 11, CSTB issued the report *Cybersecurity Today and Tomorrow: Pay Now or Pay Later*, in early 2002. The report summarized the findings of seven CSTB reports issued over the preceding decade that had cybersecurity as a principal theme. *Cybersecurity Today and Tomorrow* concludes with the following paragraph:

> Research and development on information systems security should be construed broadly to include R&D on defensive technology (including both underlying technologies and architectural issues), organizational and sociological dimensions of such security, forensic and recovery tools, and best policies and practices. Given the failure of the market to ad-

BOX 4
Defining Large-Scale Systems and Social Applications of Information Technology

Large-scale systems are IT systems that contain many (thousands, millions, billions, or trillions or more) interacting hardware and software components. They tend to be heterogeneous—in that they are composed of many different types of components—and highly complex because the interactions among the components are numerous, varied, and complicated. They also tend to span multiple organizations (or elements of organizations) and have changing configurations. Over time, the largest IT systems have become ever larger and more complex, and at any given point in time, systems of a certain scale and complexity are not feasible or economical to design with existing methodologies.

Social applications of IT serve groups of people in shared activities. The most straightforward of these applications improve the effectiveness of geographically dispersed groups of people who are collaborating on some task in a shared context. More sophisticated applications may support the operations of a business or the functioning of an entire economy; systems for e-commerce are an example. Characteristic of social applications of IT is the embedding of IT into a large organizational or social system to form a "sociotechnical" system in which people and technology interact to achieve a common purpose—even if that purpose is not obviously social, such as efficient operation of a manufacturing line (which is a conjunction of technological automation and human workers) or rapid and decisive battlefield management (which is a conjunction of command-and-control technology and the judgment and expertise of commanders). Social applications of IT—especially those supporting organizational and societal missions—tend to be large-scale and complex, mixing technical and nontechnical design and operational elements and involving often-difficult social and policy issues such as those related to privacy and access.

SOURCE: Reprinted from Computer Science and Telecommunications Board, National Research Council. 2000. *Making IT Better: Expanding Information Technology Research to Meet Society's Needs.* National Academy Press, Washington, D.C., p. 3.

dress security challenges adequately, government support for such research is especially important.[11]

CSTB's 2001 study on networked systems of embedded computers (7) sounds a similar theme (at p. 9):

> [T]he committee (composed of people from both academia and industry) believes that while some of the questions raised in this report may

[11]Computer Science and Telecommunications Board, National Research Council. 2002. *Cybersecurity Today and Tomorrow: Pay Now or Pay Later.* National Academy Press, Washington, D.C., pp. 14-15.

BOX 5
Research on the Social Applications of Information Technology

Research on the social applications of information technology (IT) combines work in technical disciplines, such as computing and communications, with research in the social sciences to understand how people, organizations, and IT systems can be combined to most effectively perform a set of tasks. Such research can address a range of issues related to IT systems, as demonstrated by the examples below . . . :

- *Novel activities and shifts in organizational, economic, and social structures*—What will people do (at work, in school, at play, in government, and so on) when computers can see and hear better than people can? How will activities and organizations change when robotic technology is widespread and cheap? How will individual and organizational activities change when surveillance via IT becomes effectively universal? New technologies will affect all kinds of people in many ways, and they hold particular promise for those with special situations or capabilities, because they will give them broader access to social and economic activities.
- *Electronic communities*—How can IT systems be best designed to facilitate the communication and coordination of groups of people working toward a common goal? Progress requires an understanding of the sociology and dynamics of groups of users, as well as of the tasks they wish to perform. Psychologists and sociologists could offer insight for the conceptualization and refinement of these social applications, and technologists could mold their technological aspects.
- *Electronic commerce*—How can buyers and sellers be best brought together to conduct business transactions on the Internet? What kinds of security technologies will provide adequate assurances of the identities of both parties and protect the confidentiality of their transactions without imposing unnecessary burdens on either? How will electronic commerce affect the competitive advantage of firms, their business strategies, and the structure of industries (e.g., their horizontal and vertical linkages)? Such work requires the insight of economists, organizational theorists, business strategists, and psychologists who understand consumer behavior, as well as of technologists.
- *Critical infrastructures*—How can IT be better embedded into the nation's transportation, energy, financial, telecommunications, and other infrastructures to make them more efficient and effective without making them less reliable or more prone to human error? For example, how can an air traffic control system be designed to provide controllers with sufficient information to make critical decisions without overwhelming them with data? Such work requires the insight of cognitive psychologists and experts in air traffic control, as well as of technologists.
- *Complexity*—How can the benefits of IT be brought to the citizenry without the exploding complexity characteristic of professional uses of IT? Although networks, computers, and software can be assembled and configured by professionals to support the mission-critical computing needs of large organizations, the techniques that make this possible are inadequate for information appliances designed for the home, car, or individual. Research is needed to simplify and automate

continues

BOX 5
Continued

system configuration, change, and repair. Such research will require insight from technologists, cognitive psychologists, and those skilled in user interface design.

SOURCE: Reprinted from Computer Science and Telecommunications Board, National Research Council. 2000. *Making IT Better: Expanding Information Technology Research to Meet Society's Needs.* National Academy Press, Washington, D.C., p. 7.

> be answered without a concerted, publicly funded research agenda, leaving this work solely to the private sector raises a number of troubling possibilities. Of great concern is that individual commercial incentives will fail to bring about work on problems that have a larger scope and that are subject to externalities: interoperability, safety, upgradability, and so on. Moreover, a lack of government funding will slow down the sharing of the research, since the commercial concerns doing the research tend to keep the research private to retain their competitive advantage. The creation of an open research community within which results and progress are shared is vital to making significant progress in this arena.

Another example of the distinctive role that federal funding can play in computing research comes from two recent CSTB studies of the Internet. The 2001 report *The Internet's Coming of Age* examined the role of the government in funding research that leads to open standards, exemplified by the work that defined the Internet. One of the Internet's hallmarks has been its openness. Proprietary research can enhance a particular product, but research leading to open standards can create a new marketplace for products. Each company that is an Internet "player" will be tempted to diverge from the common standard if it looks possible to capture a large portion of it—we have seen this during the past decade in protocols for transport, electronic mail, instant messaging, and many other areas (see Box 6). However, a common, open standard maximizes overall social welfare as a result of the network externalities obtained from the larger market. When effective open standards are made available, they can be attractive in the marketplace and may win out over proprietary ones. The report notes:

> The government's role in supporting open standards for the Internet has not been, and should not be, to directly set or influence standards. Rather, its role should be to provide funding for the networking research community, which has led to both innovative networking ideas as well

BOX 6
The Origins of Electronic Mail and Instant Messaging

The invention of timesharing systems in the 1960s not only contributed important technical developments in hardware, software, and system security but also provided the environment that led to the development of the most useful and widespread of popular applications, namely, e-mail and instant messaging (1).

Timesharing allowed concurrent multiple users to share the power of a computer, which provided a fresh way for colleagues to interact. By 1970, programmers in federally funded research laboratories had developed both asynchronous electronic mail and facilities for real-time interaction between users, in research operating systems such as Tenex, Multics, and CalTSS.

These modalities—now widely known as e-mail and instant messaging—proved so powerful that they have spread far and wide with the availability of low-cost personal computers, public networking, and client-server computing. These popular and visible tools, as well as all of the other forms of collaborative computing, have truly transformed our work and our lives. They owe their origins to the funding of IT research by the Defense Advanced Research Projects Agency and the National Science Foundation (1,3).

as specific technologies that can be translated into new open standards.[12]

A 2002 report, *Broadband: Bringing Home the Bits,* outlines an even broader role for federally funded research to enable openness in infrastructural systems:

> Support research and development on access technologies, especially targeting the needs of nonincumbent players and other areas that are not targets of stable, private sector funding. . . . [One target area is] technologies that foster the accommodation of multiple competitive service providers over facilities. Such open access-ready systems might not be a natural research and development target of large incumbent providers but will be the preferred form for a variety of public sector or public-private deployments.[13]

Broadband: Bringing Home the Bits notes that federally funded research can complement the more proprietary-oriented industry approaches to innovation, whether in communications architecture or content. It also

[12]Computer Science and Telecommunications Board, National Research Council. 2001. *The Internet's Coming of Age.* National Academy Press, Washington, D.C., p. 18.

[13]Computer Science and Telecommunications Board, National Research Council. 2002. *Broadband: Bringing Home the Bits.* National Academy Press, Washington, D.C., p. 40.

calls for the support of research on economic, social, and regulatory factors relating to broadband technologies—nontechnical factors that interact with the design and deployment of broadband.

UNIVERSITY RESEARCH AND INDUSTRIAL R&D

Much of the government-funded research in IT has been carried out at universities.[14] Federal support has constituted roughly 70 percent of total university research funding in computer science and electrical engineering since 1976 (2). Among the many benefits of federally funded university research, the generation of new knowledge is only one (see Box 7).

Strong research institutions are recognized as being among the most critical success factors in high-tech economic development (5). In computing, electronics, telecommunications, and biotechnology, evidence of the correlation abounds—in Boston (Harvard University and the Massachusetts Institute of Technology); Research Triangle Park (Duke University, the University of North Carolina, and North Carolina State University); New Jersey (Princeton University, Rutgers University, and New York City-based Columbia University); Austin (the University of Texas); southern California (the University of California at San Diego, the University of California at Los Angeles, the California Institute of Technology, and the University of Southern California); northern California (the University of California at Berkeley, the University of California at San Francisco, and Stanford University); and Seattle (the University of Washington).

In addition to creating ideas and companies, universities often import forefront technologies to their regions (e.g., the nationwide expansion of ARPANET in the 1970s and of NSFnet in the 1980s, and the continuation of those efforts through the private Internet2 activities in the 1990s and early 2000s). Universities also serve as powerful magnets for companies seeking to relocate. These contributions are not reflected in Figure 1.

Figure 1 also does not capture the most important product of universities: people. The American research university is unique in the degree to which it integrates research with education—both undergraduate and graduate education. Not only do graduating students serve to staff industry (5,6), but they also are *by far* the most effective vehicle for technol-

[14]The concentration of research in universities is particularly true for computer science research; industry played an important role in telecommunications research before the breakup of AT&T and the original Bell Labs.

BOX 7
The Diverse Benefits of University Research

Universities have a number of important characteristics that contribute to their success as engines of innovation. Among them are the following:

- *Universities can focus on long-term research.* Focusing on long-term research is the special role of universities—one that IT companies cannot be expected to fill to any significant extent (1-3). America's IT companies are extraordinarily adept at improving current products, but the track record is at best mixed on the invention and adoption of "disruptive technologies," and corporate research in IT has been becoming more applied (2).
- *Universities provide a neutral ground for collaboration.* Universities encourage movement and collaboration among faculty through leave and sabbatical policies that allow professors to visit industry, government, and other university departments or laboratories. These uniquely valuable components of the R&D structure in the United States are not generally present in industry. Universities also provide sites at which researchers from competing companies can come together to explore technical issues. At the same time that industry people share their wisdom and experience with university researchers, they have the opportunity to learn from one another (2,6).
- *Universities integrate research and education.* Universities provide a forum for educating the skilled IT workers of the future (5). The presence of research activities in an educational setting creates very powerful synergy (2,4). IT is a rapidly changing field. Many of the specific facts and techniques that a student learns become obsolete early in his or her career. The educational foundation for continuous learning—"keeping up with the field"—is a crucial component of IT education (5). Students, even beginning undergraduates, get that education not only in the classroom, but also by serving as apprentices on leading-edge research projects, where knowledge is being discovered, not read from a book. Often, new ideas are a by-product of what goes on in the classroom: in an attempt to explain the solutions to emerging problems, teachers often deepen their own understanding, while discovering interesting research questions whose answers are as yet unknown. Additionally, students are the most powerful vehicle for technology transfer, not only from university to industry but also between university laboratories and departments, through the hiring of postdoctoral researchers and assistant professors (5).
- *Universities are inherently multidisciplinary.* University researchers are well situated to draw on experts from a variety of other fields (2). There are often cultural barriers to cross-disciplinary collaboration, but physical proximity and collegial values go a long way in enabling collaboration. The multidisciplinary nature of universities is of historic and growing importance to computer science, which interfaces with so many other fields.
- *Universities are "open."* This characteristic of universities, which is true both literally and figuratively, can pay enormous unanticipated dividends. Chance interactions in an open environment can change the world; for example, when Microsoft founders Paul Allen and Bill Gates were students at Seattle's Lakeside School in the early 1970s, they were exposed to computing and computer science at the University of Washington and a university spinoff company, Computer Center Corporation.

ogy transfer (see Box 7). Federal support for university research drives this process (1-6). In top university computer science programs, over half of all graduate students receive financial support from the federal government, mostly in the form of research assistantships. In addition, most of the funding for research equipment—that is, research infrastructure—comes from federal agencies. Industry also contributes significantly to equipment but is usually attracted by existing research excellence and collaborations. Thus, by placing infrastructure in universities, the federal government directly and indirectly makes possible hands-on learning experiences for countless young engineers and scientists, as well as enabling university researchers to continue their work (1-6).

HALLMARKS OF FEDERALLY SPONSORED IT RESEARCH

As discussed below, the hallmarks of federally sponsored IT research include scale, diversity, vision, and flexibility.

1. *Federal programs have been effective in supporting the construction of large-scale systems and testbeds that have motivated research and demonstrated the feasibility of new technological approaches* (1-3).

Some research challenges are too large and require too much research infrastructure to be carried out by small, local research groups (6). In IT research, as in other areas of scientific investigation, federal programs have played an important role in stimulating and supporting large-scale efforts. DARPA's decision to construct a packet-switched network (called the ARPANET) to link computers at its many contractor sites prompted diverse, high-impact research on networking protocols, the design of packet switches and routers, software structures for managing large networks (such as the Domain Name System), and applications (such as remote log-in, file transfer, and ultimately the Web). Moreover, by constructing a successful system, DARPA demonstrated the value of large-scale packet-switched networks, motivating subsequent deployment of other networks—such as the NSF's NSFnet, which ultimately served as the foundation of the Internet—and also a series of high-speed networking testbeds (1,3).

Much of the success of major system-building efforts derives from their ability to bring together large groups of researchers from universities and industry that develop a common vocabulary, share ideas, and create a critical mass of people who subsequently extend the technology (2,6).

2. *Computing research has benefited from diverse modes of research sponsored by different federal agencies* (1-3).

Funding for research in computing has been provided by various federal agencies—most notably DARPA and NSF, but also including other parts of the Department of Defense (DOD) besides DARPA, and other federal agencies such as NASA, DOE, and the National Institutes of Health (NIH; in particular through the National Library of Medicine). Complementary investments have supported technology transfer to industry (e.g., activities of the National Institute of Standards and Technology, or NIST). Funding agencies have continually evolved in order to match their structures better to the needs of the research and policy-making communities (1). (See Box 8.)

In supporting research, these agencies pursue different objectives and employ different mechanisms. In contrast to NSF, for example—which has a mandate to support a very broad research agenda—"mission agencies" tend to focus on topics that appear to have the greatest relevance to their specific missions. Additionally, the early DARPA programs chose to concentrate large research awards in so-called centers of excellence (many of which over time have matured into some of the nation's leading university computer science programs), while NSF and the Office of Naval Research have supported individual researchers at a more diverse set of institutions (1). NSF has been active in supporting educational and research needs more broadly, awarding graduate student fellowships and providing funding for research equipment and infrastructure.

CSTB has recognized the effective leadership of NSF and DARPA, calling on them to step up to larger roles (2; p. 11):

> The programs run by [NSF and DARPA] should complement one another and should together [do the following]:
>
> - Support both theoretical and experimental work;
> - Offer awards in a variety of sizes (small, medium, and large) to support individual investigators, small teams of researchers, and larger collaborations;
> - Investigate a range of approaches to large-scale systems problems, such as improved software design methodologies, system architecture, reusable code, and biological and economic models . . . ;
> - Attempt to address the full scope of large-scale systems issues, including scalability, heterogeneity, trustworthiness, flexibility, and predictability; and
> - Give academic researchers some form of access to large-scale systems for studying and demonstrating new approaches.

BOX 8
Federal Agency Evolution

In response to proposals by Vannevar Bush and others for an organization to fund basic research, especially in universities, the U.S. Congress established the National Science Foundation (NSF) in 1950 (1). A few years earlier, the U.S. Navy had founded the Office of Naval Research to draw on science and engineering resources in the universities.

In the early 1950s, during an intense phase of the Cold War, the military services became the preeminent funders of computing and communications research. The Soviet Union's launching of Sputnik in 1957 raised fears in Congress and the country that the Soviets had forged ahead of the United States in advanced technology. In response, the U.S. Department of Defense, pressured by the Eisenhower administration, established the Advanced Research Projects Agency (ARPA, now DARPA) to fund technological projects with military implications. In 1962 DARPA created the Information Processing Techniques Office (IPTO), whose initial research agenda gave priority to further development of computers for command-and-control systems.

With the passage of time, new organizations have emerged, and old ones have often been reformed or reinvented to respond to new national imperatives and counter bureaucratic trends (2). DARPA's IPTO has transformed itself several times to bring greater coherence to its research efforts and to respond to technological developments and changes in perceived national needs for IT.

In 1967 NSF established the Office of Computing Activities, and in 1986 it formed the Computer and Information Science and Engineering Directorate to advance and coordinate support for research, education, and infrastructure in computing (1). In the 1980s NSF, which customarily has focused on fundamental research in universities, also began to encourage joint university-industry research centers through its Engineering Research Centers program (these centers focus on research and education in the context of long-time-horizon, complex engineering challenges[1]) and its Science and Technology Center program (aimed at long-term research in areas that are new or that can bridge disciplines and/or institutions and sectors[2]).

With the growth in the IT sector and corresponding IT development together with the maturation of the field of computer science, more recent federal funding has been characterized by a series of multiagency, long-term, high-risk initiatives. The first was the High Performance Computing and Communications Initiative, which emerged in the late 1980s and broadened through the mid-1990s (1,3). By the late 1990s and the establishment of the multiagency Information Technology for the Twenty-First Century initiative (in NSF, the Information Technology Research initiative), social science research—relating IT innovation to the people who use IT—was an important complement to the science and technology research per se (3,8).

---

[1]See <http://www.eng.nsf.gov/eec/erc.htm>. Accessed June 2, 2003.
[2]See <http://www.nsf.gov/od/oia/programs/stc/>. Accessed June 2, 2003.

> Given the wide circle of agencies interested in and involved with IT research and the even wider circle coming to depend on large-scale IT systems, the NSF and DARPA should attempt to involve in their research other federal agencies . . . that operate large-scale IT systems and would benefit from advances in their design. Such involvement could provide a means for researchers to gain access to operational systems for analytical and experimental purposes.

The diversity of research funding objectives and program management styles offers many benefits (1,3). It helps ensure exploration of a diverse set of research topics and consideration of a range of applications. For example, DARPA, NASA, and NIH (in addition to NSF) have all supported work in expert systems. However, because the systems have had different applications—decision aids for pilots, tools for determining the structure of molecules on other planets, and medical diagnostics—each agency has supported different groups of researchers who tried different approaches. And no one's judgment is infallible. If one agency declines to support a particular topic, researchers have other sources of funding.

3. *Visionary program managers who were willing to take risks have been a hallmark of many of the highest-impact federal research initiatives* (1,3).

The program manager is responsible for initiating, funding, and overseeing research programs. The funding and management styles of program managers at DARPA during the 1960s and 1970s, for example, reflected an ability to marry visions for technological progress with strong technical expertise and an understanding of the uncertainties of the research process (1,3). Many of these program managers and program office directors were recruited from universities and industrial research laboratories for limited tours of duty and were themselves leading researchers. With close ties to the field, they were trusted by—and trusted—the research community. They tended to lay down broad guidelines for new research areas and to draw specific project proposals from principal investigators. They were willing to place bets—to pursue high-risk/high-gain projects.

This style of funding and management allowed researchers room to pursue new venues of inquiry. The funding style resulted in advances in areas as diverse as computer graphics, artificial intelligence, networking, and computer architecture. As that experience illustrates, because unanticipated outcomes of research are so valuable, federal mechanisms for funding and managing research need to recognize the inherent uncertainties and build in enough flexibility to accommodate midcourse changes (1,3).

LOOKING FORWARD

Federal funding agencies will have to continue to adjust their strategies and tactics as national needs and imperatives change. Today there is an escalation in concern about homeland security, the globalization of industry, a rise of commodity IT products and an IT mass market, the growing dependence of economic and social activity on networking and distributed computing capabilities, and a variety of industry retrenchments. Coevolution with industry thus means different things for federally funded computing research today than it did in the middle to late decades of the 20th century.

Challenges as well as opportunities have grown: computer science is a larger field with more subdisciplines; telecommunications is increasingly intertwined with computing while evolving across multiple media;[15] the interdisciplinary problems that engage computer science and telecommunications are broader-ranging; and the number of hard problems—reflecting growth in scale, complexity, and interactions with people—has increased. Evolving capabilities motivate a range of stretch goals that can help realize the potential of information technology as a human enabler.[16] Examples include new forms of prosthetics (beginning with systems that can hear, speak, or see as well as a person can) and better ways to observe or participate in activities from a distance (i.e., telepresence).

These circumstances imply that the challenge to federal research program managers has also grown. For example, while IT is at the core of a number of interdisciplinary programs (such as the multiagency Digital Libraries Initiative and NSF's Digital Government and Computing and Social System programs), it takes more work to review proposals for interdisciplinary work and to assure its quality. It may thus be more important to engage IT-using organizations in research projects, which may involve more work for the researchers (2). The growth in opportunities at the intersection of computing and biology, for example, or even computing and the arts—both topics of CSTB projects[17]—suggests new horizons

---

[15]Innovations are enhancing the potential of optical fiber, various forms of wireless, and even older media, such as copper.

[16]These and other problems were outlined by Jim Gray in his 1998 A.M. Turing Award lecture. See Jim Gray. 1998. "What's Next? A Few Remaining Problems in Information Technology." Available online at <http://research.Microsoft.com/~Gray/talks>. Accessed June 9, 2003.

[17]The project on computing and the arts and design was completed in early 2003. See Computer Science and Telecommunications Board, National Research Council. 2003. *Beyond Productivity: Information Technology, Innovation, and Creativity.* National Academies Press, Washington, D.C.

for IT innovation that depend on the nurturing that is available through university-based research programs.

The challenges confronting program managers underscore the need to attract talent from universities and industry to such public service positions. Past advances fostered by federal funding leveraged the energies and wisdom of people who went from universities and industry into the government, for at least a limited period. It is ironic that their success has increased the incentives for researchers to stay in universities or to try their hand in industry instead of cultivating the field as program managers.

Government support for IT research will also be shaped by categories of problems in which it has a special interest. The events of September 11, 2001, remind us that computer and communications security, constrained by market failure, has always depended on federal investments. But so, too, has research in human-computer interaction, another arena in which market forces have been limited (8) and where the rise of e-government reinforces long-standing government interest associated with its own applications.[18] The post-September 11 focus on homeland security and intelligence analysis also puts a spotlight on supercomputing architectures, numerical analysis, parallel programming languages and tools, and other areas in which IT advances have flowed from scientific and engineering computing needs within the research community at large—and in which purely commercial development was unlikely at best (1,3).

The downturn in the telecommunications industry presents opportunities for the government to stimulate new directions through its support for research. We may see a consolidation and a loss of viable competition, or a realignment of the sector boundaries to better reflect economic realities. Government funding, supporting the development of open standards, can help shape the structure of industry.[19] Given the "chicken-and-egg" tension shaping advances in infrastructure and applications, government support for exploration of new kinds of applications can have great impact.[20] The government can encourage competition by supporting the definition of critical interfaces and demonstrations of feasibil-

---

[18]Computer Science and Telecommunications Board, National Research Council. 2002. *Information Technology Research, Innovation, and E-Government*. National Academy Press, Washington, D.C.

[19]See Computer Science and Telecommunications Board, National Research Council, 2001, *The Internet's Coming of Age*, National Academy Press, Washington, D.C.; and Computer Science and Telecommunications Board, National Research Council, 2002, *Broadband: Bringing Home the Bits*, National Academy Press, Washington, D.C.

[20]This was demonstrated by the evolution of the early Internet and Web, involving development and refinement of both the underlying infrastructure and a suite of compelling

ity for open standards, and it can demonstrate new architectures through field trials and testbeds. This role was critical in the emergence of the Internet, and the relevance and importance of this sort of leadership have not waned.[21]

More generally, the 2001-2002 downturn in the economy and the crisis in the telecommunications industry caused a reduction in investment across all of IT. Spending remained down in 2003, and internal investment has dropped accordingly. Venture and equity capital has also become harder to obtain in the IT industries. In times such as these, research, especially longer-term research, is an obvious target for cost cutting. But if we as a nation do not continue to invest in the foundations of innovation, we run the risk that when an improving economy justifies an increase in investment, there may be few ideas in which to invest. For that reason this time is especially important for government-sponsored research.

Today's research investments are essential to tomorrow's world leadership in IT. From its position of leadership today—reinforced by an aggregation of universities, companies, government programs, and talent—the United States is better positioned than other nations are to make the most of nonappropriable research (and even appropriable research). Properly managed, publicly funded research in IT will continue to create important new technologies and industries, some of them unimagined today. The process will continue to take 10 to 15 years from the inception of a new idea to the creation of a billion-dollar industry. Without continued federal investment in fundamental research there would still be innovation, but the quantity and range of new ideas for U.S. industry to draw from would be greatly diminished—as would the flow of people edu-

---

applications by researchers focused not only on IT but also on other fields of science and engineering in which people used IT. The Internet probably could never have developed commercially without this phase of government-supported experimentation and refinement coordinated between infrastucture and applications. For a discussion of new opportunities in the support of applications, see Computer Science and Telecommunications Board, National Research Council, 2002, *Broadband: Bringing Home the Bits*, National Academy Press, Washington, D.C.

[21]For a discussion of the role of government in setting a vision, see Computer Science and Telecommunications Board, National Research Council, 1994, *Realizing the Information Future: The Internet and Beyond*, National Academy Press, Washington, D.C. For a discussion of government leadership and the importance of government funding of research as a policy tool, see Computer Science and Telecommunications Board, National Research Council, 1996, *The Unpredictable Certainty: Information Infrastructure Through 2000*, National Academy Press, Washington, D.C.

cated at the forefront, the most important product of the nation's research universities (1-8).

The lessons of history are clear, as many CSTB studies in the past decade have shown, and many of those lessons are relevant to 21st-century realities. A complex partnership among government, industry, and universities has made the United States the world leader in IT, and information technology has become essential to our national security and economic and social well-being. Turn-of-the-century turmoil and structural changes in IT industries have diminished their inherently limited capacity to support fundamental IT research. The role of the federal government in sponsoring fundamental research in IT—largely university-based—has been and will continue to be essential.

2

Excerpts from Earlier CSTB Reports

This section contains excerpts from three CSTB reports:

- *Making IT Better: Expanding Information Technology Research to Meet Society's Needs* (2000),
- *Funding a Revolution: Government Support for Computing Research* (1999), and
- *Evolving the High Performance Computing and Communications Initiative to Support the Nation's Information Infrastructure* (1995).

While this synthesis report is based on all the CSTB reports listed in Box 1 in the "Summary and Recommendations," the excerpts from these three reports are the most general and broad. To keep this report to a reasonable length, nothing was excerpted from the other five reports. Readers are encouraged to read all eight reports, which can be found online at <http://www.nap.edu>.

For the sake of simplicity and organizational clarity, footnotes and reference citations appearing in the original texts have been omitted from the reprinted material that follows. A bar in the margins beside the excerpted material is used to indicate that it is extracted text. Section heads show the topics addressed.

MAKING IT BETTER: EXPANDING INFORMATION TECHNOLOGY RESEARCH TO MEET SOCIETY'S NEEDS (2000)

CITATION: Computer Science and Telecommunications Board (CSTB), National Research Council. 2000. *Making IT Better: Expanding Information Technology Research to Meet Society's Needs.* National Academy Press, Washington, D.C.

The Many Faces of Information Technology Research

(From pp. 23-26): IT research takes many forms. It consists of both theoretical and experimental work, and it combines elements of science and engineering. Some IT research lays out principles or constraints that apply to all computing and communications systems; examples include theorems that show the limitations of computation (what can and cannot be computed by a digital computer within a reasonable time) or the fundamental limits on capacities of communications channels. Other research investigates different classes of IT systems, such as user interfaces, the Web, or electronic mail (e-mail). Still other research deals with issues of broad applicability driven by specific needs. For example, today's high-level programming languages (such as Java and C) were made possible by research that uncovered techniques for converting the high-level statements into machine code for execution on a computer. The design of the languages themselves is a research topic: how best to capture a programmer's intentions in a way that can be converted to efficient machine code. Efforts to solve this problem, as is often the case in IT research, will require invention and design as well as the classical scientific techniques of analysis and measurement. The same is true of efforts to develop specific and practical modulation and coding algorithms that approach the fundamental limits of communication on some channels. The rise of digital communication, associated with computer technology, has led to the irreversible melding of what were once the separate fields of communications and computers, with data forming an increasing share of what is being transmitted over the digitally modulated fiber-optic cables spanning the nation and the world.

Experimental work plays an important role in IT research. One modality of research is the design experiment, in which a new technique is proposed, a provisional design is posited, and a research prototype is built in order to evaluate the strengths and weaknesses of the design. Although much of the effect of a design can be anticipated using analytic techniques, many of its subtle aspects are uncovered only when the prototype is studied. Some of the most important strides in IT have been made through such experimental research. Time-sharing, for example, evolved

in a series of experimental systems that explored different parts of the technology. How are a computer's resources to be shared among several customers? How do we ensure equitable sharing of resources? How do we insulate each user's program from the programs of others? What resources should be shared as a convenience to the customers (e.g., computer files)? How can the system be designed so it's easy to write computer programs that can be time-shared? What kinds of commands does a user need to learn to operate the system? Although some of these tradeoffs may succumb to analysis, others—notably those involving the user's evaluation and preferences—can be evaluated only through experiment.

Ideas for IT research can be gleaned both from the research community itself and from applications of IT systems. The Web, initiated by physicists to support collaboration among researchers, illustrates how people who use IT can be the source of important innovations. The Web was not invented from scratch; rather, it integrated developments in information retrieval, networking, and software that had been accumulating over decades in many segments of the IT research community. It also reflects a fundamental body of technology that is conducive to innovation and change. Thus, it advanced the integration of computing, communications, and information. The Web also embodies the need for additional science and technology to accommodate the burgeoning scale and diversity of IT users and uses: it became a catalyst for the Internet by enhancing the ease of use and usefulness of the Internet, it has grown and evolved far beyond the expectations of its inventors, and it has stimulated new lines of research aimed at improving and better using the Internet in numerous arenas, from education to crisis management.

Progress in IT can come from research in many different disciplines. For example, work on the physics of silicon can be considered IT research if it is driven by problems related to computer chips; the work of electrical engineers is considered IT research if it focuses on communications or semiconductor devices; anthropologists and other social scientists studying the uses of new technology can be doing IT research if their work informs the development and deployment of new IT applications; and computer scientists and computer engineers address a widening range of issues, from generating fundamental principles for the behavior of information in systems to developing new concepts for systems. Thus, IT research combines science and engineering, even though the popular—and even professional—association of IT with systems leads many people to concentrate on the engineering aspects. Fine distinctions between the science and engineering aspects may be unproductive: computer science is special because of how it combines the two, and the evolution of both is key to the well-being of IT research.

Implications for the Research Enterprise

(From pp. 42-43): The trends in IT suggest that the nation needs to reinvent IT research and develop new structures to support, conduct, and manage it. . . .

As IT permeates many more real-world applications, additional constituencies need to be brought into the research process as both funders and performers of IT research. This is necessary not only to broaden the funding base to include those who directly benefit from the fruits of the research, but also to obtain input and guidance. An understanding of business practices and processes is needed to support the evolution of e-commerce; insight from the social sciences is needed to build IT systems that are truly user-friendly and that help people work better together. No one truly understands where new applications such as e-commerce, electronic publishing, or electronic collaboration are headed, but business development and research together can promote their arrival at desirable destinations.

Many challenges will require the participation and insight of the end user and the service provider communities. They have a large stake in seeing these problems addressed, and they stand to benefit most directly from the solutions. Similarly, systems integrators would benefit from an improved understanding of systems and applications because they would become more competitive in the marketplace and be better able to meet their estimates of project cost and time. Unlike vendors of component technologies, systems integrators and end users deal with entire information systems and therefore have unique perspectives on the problems encountered in developing systems and the feasibility of proposed solutions. Many of the end-user organizations, however, have no tradition of conducting IT research—or technological research of any kind, in fact—and they are not necessarily capable of doing so effectively; they depend on vendors for their technology. Even so, their involvement in the research process is critical. Vendors of equipment and software have neither the requisite experience and expertise nor the financial incentives to invest heavily in research on the challenges facing end-user organizations, especially the challenges associated with the social applications of IT. Of course, they listen to their customers as they refine their products and strategies, but those interactions are superficial compared with the demands of the new systems and applications. Finding suitable mechanisms for the participation of end users and service providers, and engaging them productively, will be a big challenge for the future of IT research.

Past attempts at public-private partnerships, as in the emerging arena of critical infrastructure protection, show it is not so easy to get the public

and private sectors to interact for the purpose of improving the research base and implementation of systems: the federal government has a responsibility to address the public interest in critical infrastructure, whereas the private sector owns and develops that infrastructure, and conflicting objectives and time horizons have confounded joint exploration. As a user of IT, the government could play an important role. Whereas historically it had limited and often separate programs to support research and acquire systems for its own use, the government is now becoming a consumer of IT on a very large scale. Just as IT and the widespread access to it provided by the Web have enabled businesses to reinvent themselves, IT could dramatically improve operations and reduce the costs of applications in public health, air traffic control, and social security; government agencies, like private-sector organizations, are turning increasingly to commercial, off-the-shelf technology.

Universities will play a critical role in expanding the IT research agenda. The university setting continues to be the most hospitable for higher-risk research projects in which the outcomes are very uncertain. Universities can play an important role in establishing new research programs for large-scale systems and social applications, assuming that they can overcome long-standing institutional and cultural barriers to the needed cross-disciplinary research. Preserving the university as a base for research and the education that goes with it would ensure a workforce capable of designing, developing, and operating increasingly sophisticated IT systems. A booming IT marketplace and the lure of large salaries in industry heighten the impact of federal funding decisions on the individual decisions that shape the university environment: as the key funders of university research, federal programs send important signals to faculty and students.

The current concerns in IT differ from the competitiveness concerns of the 1980s: the all-pervasiveness of IT in everyday life raises new questions of how to get from here to there—how to realize the exciting possibilities, not merely how to get there first. A vital and relevant IT research program is more important than ever, given the complexity of the issues at hand and the need to provide solid underpinnings for the rapidly changing IT marketplace.

(From p. 93): Several underlying trends could ultimately limit the nation's innovative capacity and hinder its ability to deploy the kinds of IT systems that could best meet personal, business, and government needs. First, expenditures on research by companies that develop IT goods and services and by the federal government have not kept pace with the expanding array of IT. The disincentives to long-term, fundamental research have become more numerous, especially in the private sector,

which seems more able to lure talent from universities than the other way around. Second, and perhaps most significantly, IT research investments continue to be directed at improving the performance of IT components, with limited attention to systems issues and application-driven needs. Neither industry nor academia has kept pace with the problems posed by the large-scale IT systems used in a range of social and business contexts—problems that require fundamental research. . . . New mechanisms may be needed to direct resources to these growing problem areas.

(From pp. 6-9): Neither large-scale systems nor social applications of IT are adequately addressed by the IT research community today. Most IT research is directed toward the *components* of IT systems: the microprocessors, computers, and networking technologies that are assembled into large systems, as well as the software that enables the components to work together. This research nurtures the essence of IT, and continued work is needed in all these areas. But component research needs to be viewed as part of a much larger portfolio, in which it is complemented by research aimed directly at improving large-scale systems and the social applications of IT. The last of these includes some work (such as computer-supported cooperative work and human-computer interaction) traditionally viewed as within the purview of computer science. Research in all three areas—components, systems, and social applications—will make IT systems better able to meet society's needs, just as in the medical domain work is needed in biology, physiology, clinical medicine, and epidemiology to make the nation's population healthier.

Research on large-scale systems and the social applications of IT will require new modes of funding and performing research that can bring together a broad set of IT researchers, end users, system integrators, and social scientists to enhance the understanding of operational systems. Research in these areas demands that researchers have access to operational large-scale systems or to testbeds that can mimic the performance of much larger systems. It requires additional funding to support sizable projects that allow multiple investigators to experiment with large IT systems and develop suitable testbeds and simulations for evaluating new approaches and that engage an unusually diverse range of parties. Research by individual investigators will not, by itself, suffice to make progress on these difficult problems.

Today, most IT research fails to incorporate the diversity of perspectives needed to ensure advances on large-scale systems and social applications. Within industry, it is conducted largely by vendors of IT components: companies like IBM, Microsoft, and Lucent Technologies. Few of the companies that are engaged in providing IT services, in integrating large-scale systems (e.g., Andersen Consulting [now Accenture], EDS, or

Lockheed Martin), or in developing enterprise software (e.g., Oracle, SAP, PeopleSoft) have significant research programs. Nor do end-user organizations (e.g., users in banking, commerce, education, health care, and manufacturing) tend to support research on IT, despite their increasing reliance on IT and their stake in the way IT systems are molded. Likewise, there is little academic research on large-scale systems or social applications. Within the IT sector, systems research has tended to focus on improving the performance and lowering the costs of IT systems rather than on improving their reliability, flexibility, or scalability (although systems research is slated to receive more attention in new funding programs). Social applications present an even greater opportunity and have the potential to leverage research in human-computer interaction, using it to better understand how IT can support the work of individuals, groups, and organizations. Success in this area hinges on interdisciplinary research, which is already being carried out on a small scale.

One reason more work has not been undertaken in these areas is lack of sufficient funding. More fundamentally, the problems evident today did not reach critical proportions until recently. . . . From a practical perspective, conducting the types of research advocated here is difficult. Significant cultural gaps exist between researchers in different disciplines and between IT researchers and the end users of IT systems.

FUNDING A REVOLUTION: GOVERNMENT SUPPORT FOR COMPUTING RESEARCH (1999)

CITATION: Computer Science and Telecommunications Board (CSTB), National Research Council. 1999. *Funding a Revolution: Government Support for Computing Research.* National Academy Press, Washington, D.C.

(From p. 1): The computer revolution is not simply a technical change; it is a sociotechnical revolution comparable to an industrial revolution. The British Industrial Revolution of the late 18th century not only brought with it steam and factories, but also ushered in a modern era characterized by the rise of industrial cities, a politically powerful urban middle class, and a new working class. So, too, the sociotechnical aspects of the computer revolution are now becoming clear. Millions of workers are flocking to computing-related industries. Firms producing microprocessors and software are challenging the economic power of firms manufacturing automobiles and producing oil. Detroit is no longer the symbolic center of the U.S. industrial empire; Silicon Valley now conjures up visions of enormous entrepreneurial vigor. Men in boardrooms and gray flannel suits are giving way to the casually dressed young founders of start-up computer and Internet companies. Many of these entrepreneurs had their early hands-on computer experience as graduate students conducting federally funded university research.

As the computer revolution continues and private companies increasingly fund innovative activities, the federal government continues to play a major role, especially by funding research. Given the successful history of federal involvement, several questions arise: Are there lessons to be drawn from past successes that can inform future policy making in this area? What future roles might the government play in sustaining the information revolution and helping to initiate other technological developments?

Lessons from History

(From pp. 5-13): Why has federal support been so effective in stimulating innovation in computing? Although much has depended on the unique characteristics of individual research programs and their participants, several common factors have played an important part. Primary among them is that federal support for research has tended to *complement*, rather than preempt, industry investments in research. Effective federal research has concentrated on work that industry has limited incentive to pursue: long-term, fundamental research; large system-building efforts that require the talents of diverse communities of scientists and engi-

neers; and work that might displace existing, entrenched technologies. Furthermore, successful federal programs have tended to be organized in ways that accommodate the uncertainties in scientific and technological research. Support for computing research has come from a diversity of funding agencies; program managers have formulated projects broadly where possible, modifying them in response to preliminary results; and projects have fostered productive collaboration between universities and industry. The lessons below expand on these factors. The first three lessons address the complementary nature of government- and industry-sponsored research; the final four highlight elements of the organizational structure and management of effective federally funded research programs. . . .

1. Government supports long-range, fundamental research that industry cannot sustain.

Federally funded programs have been successful in supporting long-term research into fundamental aspects of computing, such as computer graphics and artificial intelligence, whose practical benefits often take years to demonstrate. Work on speech recognition, for example, which was begun in the early 1970s (some started even earlier), took until 1997 to generate a successful product for enabling personal computers to recognize continuous speech. Similarly, fundamental algorithms for shading three-dimensional graphics images, which were developed with defense funding in the 1960s, entered consumer products only in the 1990s, though they were available in higher-performance machines much earlier. These algorithms are now used in a range of products in the health care, entertainment, and defense industries.

Industry does fund some long-range work, but the benefits of fundamental research are generally too distant and too uncertain to receive significant industry support. Moreover, the results of such work are generally so broad that it is difficult for any one firm to capture them for its own benefit and also prevent competitors from doing so. . . . Not surprisingly, companies that have tended to support the most fundamental research have been those, like AT&T Corporation and IBM Corporation, that are large and have enjoyed a dominant position in their respective markets. As the computing industry has become more competitive, even these firms have begun to link their research more closely with corporate objectives and product development activities. Companies that have become more dominant, such as Microsoft Corporation and Intel Corporation, have increased their support for fundamental research.

2. Government supports large system-building efforts that have advanced technology and created large communities of researchers.

In addition to funding long-term fundamental research, federal programs have been effective in supporting the construction of large systems that have both motivated research and demonstrated the feasibility of new technological approaches. The Defense Advanced Research Projects Agency's (DARPA's) decision to construct a packet-switched network (called the ARPANET) to link computers at its many contractor sites prompted considerable research on networking protocols and the design of packet switches and routers. It also led to the development of structures for managing large networks, such as the domain name system, and development of useful applications, such as e-mail. Moreover, by constructing a successful system, DARPA demonstrated the value of large-scale packet-switched networks, motivating subsequent deployment of other networks, like the National Science Foundation's NSFnet, which formed the basis of the Internet.

Efforts to build large systems demonstrate that, especially in computing, innovation does not flow simply and directly from research, through development, to deployment. Development often precedes research, and research rationalizes, or explains, technology developed earlier through experimentation. Hence attempts to build large systems can identify new problems that need to be solved. Electronic telecommunications systems were in use long before Claude Shannon developed modern communications theory in the late 1940s, and the engineers who developed the first packet switches for routing messages through the ARPANET advanced empirically beyond theory. Building large systems generated questions for research, and the answers, in turn, facilitated more development.

Much of the success of major system-building efforts derives from their ability to bring together large groups of researchers from academia and industry who develop a common vocabulary, share ideas, and create a critical mass of people who subsequently extend the technology. Examples include the ARPANET and the development of the Air Force's Semi-Automatic Ground Environment (SAGE) project in the 1950s. Involving researchers from MIT, IBM, and other research laboratories, the SAGE project sparked innovations ranging from real-time computing to core memories that found widespread acceptance throughout the computer industry. Many of the pioneers in computing learned through hands-on experimentation with SAGE in the 1950s and early 1960s. They subsequently staffed the companies and laboratories of the nascent computing and communications revolution. The impact of SAGE was felt over the course of several decades.

3. Federal research funding has expanded on earlier industrial research.

In several cases, federal research funding has been important in advancing a technology to the point of commercialization after it was first explored in an industrial research laboratory. For example, IBM pioneered the concept of relational databases but did not commercialize the technology because of its perceived potential to compete with more-established IBM products. National Science Foundation (NSF)-sponsored research at UC-Berkeley allowed continued exploration of this concept and brought the technology to the point that it could be commercialized by several start-up companies—and more-established database companies (including IBM). This pattern was also evident in the development of reduced instruction set computing (RISC). Though developed at IBM, RISC was not commercialized until DARPA funded additional research at UC-Berkeley and Stanford University as part of its Very Large Scale Integrated Circuit (VLSI) program of the late 1970s and early 1980s. A variety of companies subsequently brought RISC-based products to the marketplace, including IBM, the Hewlett-Packard Company, the newly formed Sun Microsystems, Inc., and another start-up, MIPS Computer Systems. For both relational databases and VLSI, federal funding helped create a community of researchers who validated and improved on the initial work. They rapidly diffused the technology throughout the community, leading to greater competition and more rapid commercialization.

4. Computing research has benefited from diverse sources of government support.

Research in computing has been supported by multiple federal agencies, including the Department of Defense (DOD)—most notably the Defense Advanced Research Projects Agency and the military services—the National Science Foundation, National Aeronautics and Space Administration (NASA), Department of Energy (DOE), and National Institutes of Health (NIH). Each has its own mission and means of supporting research. DARPA has tended to concentrate large research grants in so-called centers of excellence, many of which over time have matured into some of the country's leading academic computer departments. The Office of Naval Research (ONR) and NSF, in contrast, have supported individual researchers at a more diverse set of institutions. They have awarded numerous peer-review grants to individual researchers, especially in universities. NSF has also been active in supporting educational and research needs more broadly, awarding graduate student fellowships and providing funding for research equipment and infrastructure. Each of these organizations employs a different set of mechanisms to support research,

from fundamental research to mission-oriented research and development projects, to procurement of hardware and software.

Such diversity offers many benefits. It not only provides researchers with many potential sources of support, but also helps ensure exploration of a diverse set of research topics and consideration of a range of applications. DARPA, NASA, and NIH have all supported work in expert systems, for example, but because the systems have had different applications—decision aids for pilots, tools for determining the structure of molecules on other planets, and medical diagnostics—each agency has supported different groups of researchers who tried different approaches.

Perhaps more importantly, no single approach to investing in research is by itself a sufficient means of stimulating innovation; each plays a role in the larger system of innovation. Different approaches work in concert, ensuring continued support for research areas as they pass through subsequent stages of development. Organizations such as NSF and ONR often funded seed work in areas that DARPA, with its larger contract awards, later magnified and expanded. DARPA's Project MAC, which gave momentum to time-shared computing in the 1960s, for example, built on earlier NSF-sponsored work on MIT's Compatible Time-Sharing System. Conversely, NSF has provided continued support for projects that DARPA pioneered but was unwilling to sustain after the major research challenges were resolved. For example, NSF funds the Metal Oxide Semiconductor Implementation Service (MOSIS)—a system developed at Xerox PARC and institutionalized by DARPA that provides university researchers with access to fast-turnaround semiconductor manufacturing services. Once established, this program no longer matched DARPA's mission to develop leading-edge technologies, but it did match NSF's mission to support university education and research infrastructure. Similarly, NSF built on DARPA's pioneering research on packet-switched networks to construct the NSFnet, a precursor to today's Internet.

5. Strong program managers and flexible management structures have enhanced the effectiveness of computing research.

Research in computing, as in other fields, is a highly unpredictable endeavor. The results of research are not evident at the start, and their most important contributions often differ from those originally envisioned. Few expected that the Navy's attempt to build a programmable aircraft simulator in the late 1940s would result in the development of the first real-time digital computer (the Whirlwind); nor could DARPA program managers have anticipated that their early experiments on packet switching would evolve into the Internet and later the World Wide Web.

The potential for unanticipated outcomes of research has two implications for federal policy. First, it suggests that measuring the results of federally funded research programs is extremely difficult. Projects that appear to have failed often make significant contributions to later technology development or achieve other objectives not originally envisioned. Furthermore, research creates many intangible products, such as knowledge and educated researchers whose value is hard to quantify. Second, it implies that federal mechanisms for funding and managing research need to recognize the uncertainties inherent in computing research and to build in sufficient flexibility to accommodate mid-course changes and respond to unanticipated results.

A key element in agencies' ability to maintain flexibility in the past has been their program managers, who have responsibility for initiating, funding, and overseeing research programs. The funding and management styles of program managers at DARPA during the 1960s and 1970s, for example, reflected an ability to marry visions for technological progress with strong technical expertise and an understanding of the uncertainties of the research process. Many of these program managers and office directors were recruited from academic and industry research laboratories for limited tours of duty. They tended to lay down broad guidelines for new research areas and to draw specific project proposals from principal investigators, or researchers, in academic computer centers. This style of funding and management resulted in the government stimulating innovation with a light touch, allowing researchers room to pursue new avenues of inquiry. In turn, it helped attract top-notch program managers to federal agencies. With close ties to the field and its leading researchers, they were trusted by—and trusted in—the research community.

This funding style resulted in great advances in areas as diverse as computer graphics, artificial intelligence, networking, and computer architectures. Although mechanisms are clearly needed to ensure accountability and oversight in government-sponsored research, history demonstrates the benefits of instilling these values in program managers and providing them adequate support to pursue promising research directions.

6. Collaboration between industry and university researchers has facilitated the commercialization of computing research and maintained its relevance.

Innovation in computing requires the combined talents of university and industry researchers. Bringing them together has helped ensure that industry taps into new academic research and that university researchers

understand the challenges facing industry. Such collaboration also helps facilitate the commercialization of technology developed in a university setting. All of the areas described in this report's case studies—relational databases, the Internet, theoretical computer science, artificial intelligence, and virtual reality—involved university and industry participants. Other projects examined, such as SAGE, Project MAC, and very large scale integrated circuits, demonstrate the same phenomenon.

Collaboration between industry and universities can take many forms. Some projects combine researchers from both sectors on the same project team. Other projects involve a transition from academic research laboratories to industry (via either the licensing of key patents or the creation of new start-up companies) once the technology matures sufficiently. As the case studies demonstrate, effective linkages between industry and universities tended to emerge from projects, rather than being thrust upon them. Project teams assembled to build large systems included the range of skills needed for a particular project. University researchers often sought out productive avenues for transferring research results to industry, whether linking with existing companies or starting new ones. Such techniques have often been more effective than explicit attempts to encourage collaboration, many of which have foundered due to the often conflicting time horizons of university and industry researchers.

7. Organizational innovation and adaptation are necessary elements of federal research support.

Over time, new government organizations have formed to support computing research, and organizations have continually evolved in order to better match their structure to the needs of the research and policy-making communities. In response to proposals by Vannevar Bush and others that the country needed an organization to fund basic research, especially in the universities, for example, Congress established the National Science Foundation in 1950. A few years earlier, the Navy founded the Office of Naval Research to draw on science and engineering resources in the universities. In the early 1950s during an intense phase of the Cold War, the military services became the preeminent funders of computing and communications. The Soviet Union's launching of Sputnik in 1957 raised fears in Congress and the country that the Soviets had forged ahead of the United States in advanced technology. In response, the U.S. Department of Defense, pressured by the Eisenhower administration, established the Advanced Research Projects Agency (ARPA, now DARPA) to fund technological projects with military implications. In 1962 DARPA created the Information Processing Techniques Office (IPTO), whose initial re-

search agenda gave priority to further development of computers for command-and-control systems.

With the passage of time, new organizations have emerged, and old ones have often been reformed or reinvented to respond to new national imperatives and counter bureaucratic trends. DARPA's IPTO has transformed itself several times to bring greater coherence to its research efforts and to respond to technological developments. NSF in 1967 established the Office of Computing Activities and in 1986 formed the Computer and Information Sciences and Engineering (CISE) Directorate to couple and coordinate support for research, education, and infrastructure in computer science. In the 1980s NSF, which customarily has focused on basic research in universities, also began to encourage joint academic-industrial research centers through its Engineering Research Centers program. With the relative increase in industrial support of research and development in recent years, federal agencies such as NSF have rationalized their funding policies to complement short-term industrial R&D. Federal funding of long-term, high-risk initiatives continues to have a high priority.

As this history suggests, federal funding agencies will need to continue to adjust their strategies and tactics as national needs and imperatives change. The Cold War imperative shaped technological history during much of the last half-century. International competitiveness served as a driver of government funding of computing and communications during the late 1980s and early 1990s. With the end of the Cold War and the globalization of industry, the U.S. computing industries need to maintain their high rates of innovation, and federal structures for managing computing research may need to change to ensure that they are appropriate for this new environment.

Sources of U.S. Success

(From pp. 27-28): That the United States should be the leading country in computing and communications was not preordained. Early in the industry's formation, the United Kingdom was a serious competitor. The United Kingdom was the home of the Difference Engine and later the Analytical Engine, both of which were programmable mechanical devices designed and partially constructed by Charles Babbage and Ada, Countess of Lovelace, in the 19th century. Basic theoretical work defining a universal computer was the contribution of Alan Turing in Cambridge just before the start of World War II. The English defense industry—with Alan Turing's participation—conceived and constructed vacuum tube computers able to break the German military code. Both machines and their accomplishments were kept secret, much like the efforts and suc-

cesses of the National Security Agency in this country. After the war, English universities constructed research computers and developed computer concepts that later found significant use in U.S. products. Other European countries, Germany and France in particular, also made efforts to gain a foothold in this new technology.

How then did the United States become a leader in computing? The answer is manifold, and a number of external factors clearly played a role. The state of Europe, England in particular, at the end of World War II played a decisive role, as rebuilding a country and industry is a more difficult task than shifting from a war economy to a consumer economy. The movement of people among universities, industry, and government laboratories at the end of World War II in the United Kingdom and the United States also contributed by spreading the experience gained during the war, especially regarding electronics and computing. American students and scholars who were studying in England as Fulbright Scholars in the 1950s learned of the computer developments that had occurred during the war and that were continuing to advance.

Industrial prowess also played a role. After World War II, U.S. firms moved quickly to build an industrial base for computing. IBM and Remington Rand recognized quite early that electronic computers were a threat to their conventional electromechanical punched-card business and launched early endeavors into computing. . . . Over time, fierce competition and expectations of rapid market growth brought billions in venture money to the industry's inventors and caused a flowering of small high-tech innovators. Rapid expansion of the U.S. marketplace for computing equipment created buyers for new computing equipment. The rapid post-World War II expansion of civilian-oriented industries and financial sources created new demands for data and data processing. Insurance companies and banks were at the forefront of installing early computers in their operations. New companies, such as Engineering Research Associates, Datamatic, and Eckert-Mauchly, as well as established companies in the data processing field, such as IBM and Sperry Rand, saw an opportunity for new products and new markets. The combination of new companies and established ones was a powerful force. It generated fierce competition and provided substantial capital funds.

These factors helped the nation gain an early lead in computing that it has maintained. While firms from other nations have made inroads into computing technology—from memory chips to supercomputers—U.S. firms have continued to dominate both domestic and international markets in most product categories. This success reflects the strength of the nation's innovation system in computing technology, which has continually developed, marketed, and supported new products, processes, and services.

Research and Technological Innovation

(From pp. 28-31): Innovation is generally defined as the process of developing and putting into practice new products, processes, or services. It draws upon a range of activities, including research, product development, manufacturing, and marketing. Although often viewed as a linear, sequential process, innovation is usually more complicated, with many interactions among the different activities and considerable feedback. It can be motivated by new research advances or by recognition of a new market need. Government, universities, and industry all play a role in the innovation process.

Research is a vital part of innovation in computing. In dollar terms, research is just a small part of the innovation process, representing less than one-fifth of the cost of developing and introducing new products in the United States, with preparation of product specifications, prototype development, tooling and equipment, manufacturing start-up, and marketing start-up comprising the remainder. Indeed, computer manufacturers allocated an average of just 20 percent of their research and development budgets to research between 1976 and 1995, with the balance supporting product development. Even in the largest computer manufacturers, such as IBM, research costs are only about 1 to 2 percent of total operating expenses. Nevertheless, research plays a critical role in the innovation process, providing a base of scientific and technological knowledge that can be used to develop new products, processes, and services. This knowledge is used at many points in the innovation process—generating ideas for new products, processes, or services; solving particular problems in product development or manufacturing; or improving existing products, for example. . . .

Traditionally, research expenditures have been characterized as either basic or applied. The term "basic research" is used to describe work that is exploratory in nature, addressing fundamental scientific questions for which ready answers are lacking; the term "applied research" describes activities aimed at exploring phenomena necessary for determining the means by which a recognized need may be met. These terms, at best, distinguish between the motivations of researchers and the manner in which inquiries are conducted, and they are limited in their ability to describe the nature of scientific and technological research. Recent work has suggested that the definition of basic research be expanded to include explicitly both basic scientific research and basic technological research. This definition recognizes the value of exploratory research into basic technological phenomena that can be used in a variety of products. Examples include research on the blue laser, exploration of biosensors, and much of the fundamental work in computer engineering.

(From pp. 21-23): Clearly, the future of computing will differ from the history of computing because both the technology and environmental factors have changed. Attempts by companies to align their research activities more closely with product development processes have influenced the role they may play in the innovation process. As the computing industry has grown and the technology has diffused more widely throughout society, government has continued to represent a proportionally smaller portion of the industry.

The Benefits of Public Support of Research

(From pp. 46-47): The development of scientific and technological knowledge is a cumulative process, one that depends on the prompt disclosure of new findings so that they can be tested and, if confirmed, integrated with other bodies of reliable knowledge. In this way open science promotes the rapid generation of further discoveries and inventions, as well as wider practical exploitation of additions to the stock of knowledge.

The economic case for public funding of what is commonly referred to as basic research rests mainly on that insight, and on the observation that business firms are bound to be considerably discouraged by the greater uncertainties surrounding investment in fundamental, exploratory inquiries (compared to commercially targeted R&D), as well as by the difficulties of forecasting when and how such outlays will generate a satisfactory rate of return.

The proposition at issue here is quantitative, not qualitative. One cannot adequately answer the question "Will there be enough?" merely by saying, "There will be some." Economists do not claim that without public patronage (or intellectual property protection), basic research will cease entirely. Rather, their analysis holds that there will not be enough basic research—not as much as would be carried out were individual businesses (like society as a whole) able to anticipate capturing all the benefits of this form of investment. Therefore, no conflict exists between this theoretical analysis and the observation that R&D-intensive companies do indeed fund some exploratory research into fundamental questions. Their motives for this range from developing a capability to monitor progress at the frontiers of science, to identifying ideas for potential lines of innovation that may be emerging from the research of others, to being better positioned to penetrate the secrets of their rivals' technological practices.

Nevertheless, funding research is a long-term strategy, and therefore sensitive to commercial pressures to shift research resources toward advancing existing product development and improving existing processes,

rather than searching for future technological options. Large organizations that are less asset constrained, and of course the public sector, are better able to take on the job of pushing the frontiers of science and technology. Considerations of these kinds are important in addressing the issue of how to find the optimal balance for the national research effort between secrecy and disclosure of scientific and engineering information, as well as in trying to adjust the mix of exploratory and applications-driven projects in the national research portfolio.

(From p. 137): Quantifying the benefits of federal research support is a difficult, if not impossible, task for several reasons. First, the output of research is often intangible. Most of the benefit takes the form of new knowledge that subsequently may be instantiated in new hardware, software, or systems, but is itself difficult to measure. At other times, the benefits take the form of educated people who bring new ideas or a fresh perspective to an organization. Second, the delays between the time a research program is conducted and the time the products incorporating the research results are sold make measurement even more difficult. Often, the delays run into decades, making it difficult to tell midcourse how effective a particular program has been. Third, the benefits of a particular research program may not become visible until other technological advances are made. For example, advances in computer graphics did not have widespread effect until suitable hardware was more broadly available for producing three-dimensional graphical images. Finally, projects that are perceived as failures often provide valuable lessons that can guide or improve future research. Even if they fail to reach their original objectives, research projects can make lasting contributions to the knowledge base.

Maintaining University Research Capabilities

(From pp. 139-140): Federal funding has . . . maintained university research capabilities in computing. Universities depend largely on federal support for research programs in computer science and electrical engineering, the two academic disciplines most closely aligned with computing and communications. Since 1973, federal agencies have provided roughly 70 percent of all funding for university research in computer science. In electrical engineering, federal funding has declined from its peak of 75 percent of total university research support in the early 1970s, but still represented 65 percent of such funding in 1995. Additional support has come in the form of research equipment. Universities need access to state-of-the-art equipment in order to conduct research and train students. Although industry contributes some equipment, funding for uni-

versity research equipment has come largely from federal sources since the 1960s. Between 1981 and 1995, the federal government provided between 59 and 76 percent of annual research equipment expenditures in computer science and between 64 and 83 percent of annual research equipment expenditures in electrical engineering. Such investments have helped ensure that researchers have access to modern computing facilities and have enabled them to further expand the capabilities of computing and communications systems.

Universities play an important role in the innovation process. They tend to concentrate on research with broad applicability across companies and product lines and to share new knowledge openly. Because they are not usually subject to commercial pressures, university researchers often have greater ability than their industrial counterparts to explore ideas with uncertain long-term payoffs. Although it would be difficult to determine how much university research contributes directly to industrial innovation, it is telling that each of the case studies and other major examples examined in [the source] report—relational databases, the Internet, theoretical computer science, artificial intelligence, virtual reality, SAGE, computer time-sharing, very large scale integrated circuits, and the personal computer—involved the participation of university researchers. Universities play an especially effective role in disseminating new knowledge by promoting open publication of research results. They have also served as a training ground for students who have taken new ideas with them to existing companies or started their own companies. Diffusion of knowledge about relational databases, for instance, was accelerated by researchers at the University of California at Berkeley who published the source code for their Ingres system and made it available free of charge. Several of the lead researchers in this project established companies to commercialize the technology or brought it back to existing firms where they championed its use.

Creating Human Resources

(From pp. 140-141): In addition to supporting the creation of new technology, federal funding for research has also helped create the human resources that have driven the computer revolution. Many industry researchers and research managers claim that the most valuable result of university research programs is educated students—by and large, an outcome enabled by federal support of university research. Federal support for university research in computer science grew from $65 million to $350 million between 1976 and 1995, while federal support for university research in electrical engineering grew from $74 million to $177 million (in constant 1995 dollars). Much of this funding was used to support gradu-

ate students. Especially at the nation's top research universities, the studies of a large percentage of graduate students have been supported by federal research contracts. Graduates of these programs, and faculty researchers who received federal funding, have gone on to form a number of companies, including Sun Microsystems, Inc. (which grew out of research conducted by Forest Baskett and Andy Bechtolsheim with sponsorship from DARPA) and Digital Equipment Corporation (founded by Ken Olsen, who participated in the SAGE project). Graduates also staff academic faculties that continue to conduct research and educate future generations of researchers.

Furthermore, the availability of federal research funding has enabled the growth and expansion of computer science and computer engineering departments at U.S. universities, which increased in number from 6 in 1965 to 56 in 1975 and to 148 in 1995. The number of graduate students in computer science also grew dramatically, expanding more than 40-fold from 257 in 1966 to 11,500 in 1995, with the number of Ph.D. degrees awarded in computer science increasing from 19 in 1966 to over 900 in 1995. Even with this growth in Ph.D. production, demand for computing researchers still outstrips the supply in both industry and academia.

Beyond supporting student education and training, federal funding has also been important in creating networks of researchers in particular fields—developing communities of researchers who could share ideas and build on each other's strengths. Despite its defense orientation, DARPA historically encouraged open dissemination of the results of sponsored research, as did other federal agencies. In addition, DARPA and other federal agencies funded large projects with multiple participants from different organizations. These projects helped create entire communities of researchers who continued to refine, adopt, and diffuse new technology throughout the broader computing research community. Development of the Internet demonstrates the benefits of this approach: by funding groups of researchers in an open environment, DARPA created an entire community of users who had a common understanding of the technology, adopted a common set of standards, and encouraged their use broadly. Early users of the ARPANET created a critical mass of people who helped to disseminate the technology, giving the Internet Protocol an important early lead over competing approaches to packet switching.

The Organization of Federal Support: A Historical Review

(From pp. 85-86): Rather than a single, overarching framework of support, federal funding for research in computing has been managed by a set of agencies and offices that carry the legacies of the historical periods in which they were created. Crises such as World War II, Korea, Sputnik,

Vietnam, the oil shocks, and concerns over national competitiveness have all instigated new modes of government support. Los Alamos National Laboratory, for example, a leader in supercomputing, was created by the Manhattan Project and became part of the Department of Energy. The Office of Naval Research and the National Science Foundation emerged in the wake of World War II to continue the successful contributions of wartime science. The Defense Advanced Research Projects Agency (DARPA) and the National Aeronautics and Space Administration (NASA) are products of the Cold War, created in response to the launch of Sputnik to regain the nation's technological leadership. The National Bureau of Standards, an older agency, was transformed into the National Institute of Standards and Technology in response to . . . concerns about national competitiveness. Each organization's style, mission, and importance have changed over time; yet each organization profoundly reflects the process of its development, and the overall landscape is the result of numerous layers of history.

Understanding these layers is crucial for discussing the role of the federal government in computing research. [The following sections briefly set] out a history of the federal government's programmatic involvement in computing research since 1945, distinguishing the various layers in the historical eras in which they were first formed. The objective is to identify the changing role the government has played in these different historical periods, discuss the changing political and technological environment in which federal organizations have acted, and draw attention to the multiplicity, diversity, and flexibility of public-sector programs that have stimulated and underwritten the continuing stream of U.S. research in computing and communications since World War II. In fulfilling this charge, [the following text] reviews a number of prominent federal research programs that exerted profound influence on the evolving computing industry. These programs are illustrative of the effects of federal funding on the industry at different times. Other programs, too numerous to describe here, undoubtedly played key roles in the history of the computing industry but are not considered here.

1945-1960: Era of Government Computers

(From pp. 86-87): In late 1945, just a few weeks after atomic bombs ended World War II and thrust the world into the nuclear age, digital electronic computers began to whir. The ENIAC (Electronic Numerical Integrator and Computer), built at the University of Pennsylvania and funded by the Army Ballistics Research Laboratory, was America's first such machine. The following 15 years saw electronic computing grow from a laboratory technology into a routine, useful one. Computing hard-

ware moved from the ungainly and delicate world of vacuum tubes and paper tape to the reliable and efficient world of transistors and magnetic storage. The 1950s saw the development of key technical underpinnings for widespread computing: cheap and reliable transistors available in large quantities, rotating magnetic drum and disk storage, magnetic core memory, and beginning work in semiconductor packaging and miniaturization, particularly for missiles. In telecommunications, American Telephone and Telegraph (AT&T) introduced nationwide dialing and the first electronic switching systems at the end of the decade. A fledgling commercial computer industry emerged, led by International Business Machines (IBM) (which built its electronic computer capability internally) and Remington Rand (later Sperry Rand), which purchased Eckert-Mauchly Computer Corporation in 1950 and Engineering Research Associates in 1952. Other important participants included Bendix, Burroughs, General Electric (GE), Honeywell, Philco, Raytheon, and Radio Corporation of America (RCA).

In computing, the technical cutting edge, however, was usually pushed forward in government facilities, at government-funded research centers, or at private contractors doing government work. Government funding accounted for roughly three-quarters of the total computer field. A survey performed by the Army Ballistics Research Laboratory in 1957, 1959, and 1961 lists every electronic stored-program computer in use in the country (the very possibility of compiling such a list says a great deal about the community of computing at the time). The surveys reveal the large proportion of machines in use for government purposes, either by federal contractors or in government facilities.

The Government's Early Role

(From pp. 87-88): Before 1960, government—as a funder and as a customer—dominated electronic computing. Federal support had no broad, coherent approach, however, arising somewhat ad hoc in individual federal agencies. The period was one of experimentation, both with the technology itself and with diverse mechanisms for federal support. From the panoply of solutions, distinct successes and failures can be discerned, from both scientific and economic points of view. After 1960, computing was more prominently recognized as an issue for federal policy. The National Science Foundation and the National Academy of Sciences issued surveys and reports on the field.

If government was the main driver for computing research and development (R&D) during this period, the main driver for government was the defense needs of the Cold War. Events such as the explosion of a Soviet atomic bomb in 1949 and the Korean War in the 1950s heightened

international tensions and called for critical defense applications, especially command-and-control and weapons design. It is worth noting, however, that such forces did not exert a strong influence on telecommunications, an area in which most R&D was performed within AT&T for civilian purposes. Long-distance transmission remained analog, although digital systems were in development at AT&T's Bell Laboratories. Still, the newly emergent field of semiconductors was largely supported by defense in its early years. During the 1950s, the Department of Defense (DOD) supported about 25 percent of transistor research at Bell Laboratories.

However much the Cold War generated computer funding, during the 1950s dollars and scale remained relatively small compared to other fields, such as aerospace applications, missile programs, and the Navy's Polaris program (although many of these programs had significant computing components, especially for operations research and advanced management techniques). By 1950, government investment in computing amounted to $15 million to $20 million per year.

All of the major computer companies during the 1950s had significant components of their R&D supported by government contracts of some type. At IBM, for example, federal contracts supported more than half of the R&D and about 35 percent of R&D as late as 1963 (only in the late 1960s did this proportion of support trail off significantly, although absolute amounts still increased). The federal government supported projects and ideas the private sector would not fund, either for national security, to build up human capital, or to explore the capabilities of a complex, expensive technology whose long-term impact and use was uncertain. Many federally supported projects put in place prototype hardware on which researchers could do exploratory work.

Establishment of Organizations

(From pp. 88-95): The successful development projects of World War II, particularly radar and the atomic bomb, left policymakers asking how to maintain the technological momentum in peacetime. Numerous new government organizations arose, attempting to sustain the creative atmosphere of the famous wartime research projects and to enhance national leadership in science and technology. Despite Vannevar Bush's efforts to establish a new national research foundation to support research in the nation's universities, political difficulties prevented the bill from passing until 1950, and the National Science Foundation (NSF) did not become a significant player in computing until later in that decade. During the 15 years immediately after World War II, research in computing and communications was supported by mission agencies of the federal government, such as DOD, the Department of Energy (DOE), and NASA. In

retrospect, it seems that the nation was experimenting with different models for supporting this intriguing new technology that required a subtle mix of scientific and engineering skill.

Military Research Offices

Continuity in basic science was provided primarily by the Office of Naval Research (ONR), created in 1946 explicitly to perpetuate the contributions scientists made to military problems during World War II. In computing, the agency took a variety of approaches simultaneously. First, it supported basic intellectual and mathematical work, particularly in numerical analysis. These projects proved instrumental in establishing a sound mathematical basis for computer design and computer processing. Second, ONR supported intellectual infrastructure in the infant field of computing, sponsoring conferences and publications for information dissemination. Members of ONR participated in founding the Association for Computing Machinery in 1947.

ONR's third approach to computing was to sponsor machine design and construction. It ordered a computer for missile testing through the National Bureau of Standards from Raytheon, which became known as the Raydac machine, installed in 1952. ONR supported Whirlwind, MIT's first digital computer and progenitor of real-time command-and-control systems. John von Neumann built a machine with support from ONR and other agencies at Princeton's Institute for Advanced Study, known as the IAS computer. The project produced significant advances in computer architecture, and the design was widely copied by both government and industrial organizations.

Other military services created offices on a model similar to that of ONR. The Air Force Office of Scientific Research was established in 1950 to manage U.S. Air Force R&D activities. Similarly, the U.S. Army established the Army Research Office to manage and promote Army programs in science and technology.

National Bureau of Standards

Arising out of its role as arbiter of weights and measures, the National Bureau of Standards (NBS) had long had its own laboratories and technical expertise and had long served as a technical advisor to other government agencies. In the immediate postwar years, NBS sought to expand its advisory role and help U.S. industry develop wartime technology for commercial purposes. NBS, through its National Applied Mathematics Laboratory, acted as a kind of expert agent for other government agencies, selecting suppliers and overseeing construction and delivery of

new computers. For example, NBS contracted for the three initial Univac machines—the first commercial, electronic, digital, stored-program computers—one for the Census Bureau and two for the Air Materiel Command.

NBS also got into the business of building machines. When the Univac order was plagued by technical delays, NBS built its own computer in-house. The Standards Eastern Automatic Computer (SEAC) was built for the Air Force and dedicated in 1950, the first operational, electronic, stored-program computer in this country. NBS built a similar machine, the Standards Western Automatic Computer (SWAC) for the Navy on the West Coast. Numerous problems were run on SEAC, and the computer also served as a central facility for diffusing expertise in programming to other government agencies. Despite this significant hardware, however, NBS's bid to be a government center for computing expertise ended in the mid-1950s. Caught up in postwar debates over science policy and a controversy over battery additives, NBS research funding was radically reduced, and NBS lost its momentum in the field of computing.

Atomic Energy Commission

Nuclear weapons design and research have from the beginning provided impetus to advances in large-scale computation. The first atomic bombs were designed only with desktop calculators and punched-card equipment, but continued work on nuclear weapons provided some of the earliest applications for the new electronic machines as they evolved. The first computation job run on the ENIAC in 1945 was an early calculation for the hydrogen bomb project "Super." In the late 1940s, the Los Alamos National Laboratory built its own computer, MANIAC, based on von Neumann's design for the Institute for Advanced Study computer at Princeton, and the Atomic Energy Commission (AEC) funded similar machines at Argonne National Laboratory and Oak Ridge National Laboratory.

In addition to building their own computers, the AEC laboratories were significant customers for supercomputers. The demand created by AEC laboratories for computing power provided companies with an incentive to design more powerful computers with new designs. In the early 1950s, IBM built its 701, the Defense Calculator, partly with the assurance that Los Alamos and Livermore would each buy at least one. In 1955, the AEC laboratory at Livermore, California, commissioned Remington Rand to design and build the Livermore Automatic Research Computer (LARC), the first supercomputer. The mere specification for LARC advanced the state of the art, as the bidding competition required the use of transistors instead of vacuum tubes. IBM developed improved

ferrite-core memories and supercomputer designs with funding from the National Security Agency, and designed and built the Stretch supercomputer for the Los Alamos Scientific Laboratory, beginning it in 1956 and installing it in 1961. Seven more Stretch supercomputers were built. Half of the Stretch supercomputers sold were used for nuclear weapon research and design.

The AEC continued to specify and buy newer and faster supercomputers, including the Control Data 6600, the STAR 100, and the Cray 1 (although developed without AEC funds), practically ensuring a market for continued advancements. AEC and DOE laboratories also developed much of the software used in high-performance computing including operating systems, numerical analysis software, and matrix evaluation routines. In addition to stimulating R&D in industry, the AEC laboratories also developed a large talent pool on which the computer industry and academia could draw. In fact, the head of IBM's Applied Science Department, Cuthbert Hurd, came directly to IBM in 1949 from the AEC's Oak Ridge National Laboratory. Physicists worked on national security problems with government support providing demand, specifications, and technical input, as well as dollars, for industry to make significant advances in computing technology.

Private Organizations

Not all the new organizations created by the government to support computing were public. A number of new private organizations also sprang up with innovative new charters and government encouragement that held prospects of initial funding support. In 1956, at the request of the Air Force, the Massachusetts Institute of Technology (MIT) created Project Lincoln, now known as the Lincoln Laboratory, with a broad charter to study problems in air defense to protect the nation from nuclear attack. The Lincoln Laboratory then oversaw the construction of the Semi-Automatic Ground Environment (SAGE) air-defense system. In 1946, the Air Force and Douglas Aircraft created a joint venture, Project RAND, to study intercontinental warfare. In the following year RAND separated from Douglas and became the independent, nonprofit RAND Corporation.

RAND worked only for the Air Force until 1956, when it began to diversify to other defense and defense-related contractors, such as the Advanced Research Projects Agency and the Atomic Energy Commission, and provided, for a time, what one researcher called "in some sense the world's largest installation for scientific computing [in 1950]." RAND specialized in developing computer systems, such as the Johnniac, based on the IAS computer, which made RAND the logical source for the pro-

gramming on SAGE. While working on SAGE, RAND trained hundreds of programmers, eventually leading to the spin-off of RAND's Systems Development Division and Systems Training Program into the Systems Development Corporation. Computers made a major impact on the systems analysis and game theoretic approaches that RAND and other similar think tanks used in attempts to model nuclear and conventional warfighting strategies.

Engineering Research Associates (ERA) represented yet another form of government support: the private contractor growing out of a single government agency. With ERA, the Navy effectively privatized its wartime cryptography organization and was able to maintain civilian expertise through the radical postwar demobilization. ERA was founded in St. Paul, Minnesota, in January 1946 by two engineers who had done cryptography for the Navy and their business partners. The Navy moved its Naval Computing Machine Laboratory from Dayton to St. Paul, and ERA essentially became the laboratory. ERA did some research, but it primarily worked on task-oriented, cost-plus contracts. As one participant recalled, "It was not a university atmosphere. It was 'Build stuff. Make it work. How do you package it? How do you fix it? How do you document it?'" ERA built a community of engineering skill, which became the foundation of the Minnesota computer industry. In 1951, for example, the company hired Seymour Cray for his first job out of the University of Minnesota.

As noted earlier, the RAND Corporation had contracted in 1955 to write much of the software for SAGE owing to its earlier experience in air defense and its large pool of programmers. By 1956, the Systems Training Program of the RAND Corporation, the division assigned to SAGE, was larger than the rest of the corporation combined, and it spun off into the nonprofit Systems Development Corporation (SDC). SDC played a significant role in computer training. As described by one of the participants, "Part of SDC's nonprofit role was to be a university for programmers. Hence our policy in those days was not to oppose the recruiting of our personnel and not to match higher salary offers with an SDC raise." By 1963, SDC had trained more than 10,000 employees in the field of computer systems. Of those, 6,000 had moved to other businesses across the country.

Observations

(From pp. 95-96): In retrospect, the 1950s appear to have been a period of institutional and technological experimentation. This diversity of approaches, while it brought the field and the industry from virtually nothing to a tentative stability, was open to criticisms of waste, duplica-

tion of effort, and ineffectiveness caused by rivalries among organizations and their funding sources. The field was also driven largely by the needs of government agencies, with relatively little input from computer-oriented scientists at the highest levels. Criticism remained muted during the decade when the military imperatives of the Cold War seemed to dominate all others, but one event late in the decade opened the entire system of federal research support to scrutiny: the launch of Sputnik in 1957. Attacks mounted that the system of R&D needed to be changed, and they came not only from the press and the politicians but also from scientists themselves.

1960-1970: Supporting a Continuing Revolution

(From p. 96): Several significant events occurred to mark a transition from the infancy of information technology to a period of diffusion and growth. Most important of these was the launching of Sputnik in 1957, which sent convulsions through the U.S. science and engineering world and redoubled efforts to develop new technology. President Eisenhower elevated scientists and engineers to the highest levels of policy making. Thus was inaugurated what some have called the golden age of U.S. research policy. Government support for information technology took off in the 1960s and assumed its modern form. The Kennedy administration brought a spirit of technocratic reform to the Pentagon and the introduction of systems analysis and computer-based management to all aspects of running the military. Many of the visions that set the research agendas for the following 15 years (and whose influence remains today) were set in the early years of the decade.

Maturing of a Commercial Industry

(From pp. 96-97): Perhaps most important, the early 1960s can be defined as the time when the commercial computer industry became significant on its own, independent of government funding and procurement. Computerized reservation systems began to proliferate, particularly the IBM/American Airlines SABRE system, based in part on prior experience with military command-and-control systems (such as SAGE). The introduction of the IBM System/360 in 1964 solidified computer applications in business, and the industry itself, as significant components of the economy.

This newly vital industry, dominated by "Snow White" (IBM) and the "Seven Dwarfs" (Burroughs, Control Data, GE, Honeywell, NCR, RCA, and Sperry Rand), came to have several effects on government-supported R&D. First, and most obvious, some companies (mostly IBM) became

large enough to conduct their own in-house research. IBM's Thomas J. Watson Research Center was dedicated in 1961. Its director, Emanuel Piore, was recruited from ONR, and he emphasized basic research. Such laboratories not only expanded the pool of researchers in computing and communications but also supplied a source of applied research that allowed or, conversely, pushed federal support to focus increasingly on the longest-term, riskiest ideas and on problems unique to government. Second, the industry became a growing employer of computer professionals, providing impetus to educational programs at universities and making computer science and engineering increasingly attractive career paths to talented young people.

These years saw turning points in telecommunications as well. In 1962, AT&T launched the first active communications satellite, Telstar, which transmitted the first satellite-relay telephone call and the first live transatlantic television signal. That same year, a less-noticed but equally significant event occurred when AT&T installed the first commercial digital-transmission system. Twenty-four digital speech channels were time multiplexed onto a repeatered digital transmission line operating at 1.5 megabits per second. In 1963, the first Stored Program Control electronic switching system was placed into service, inaugurating the use of digital computer technology for mainstream switching.

The 1960s also saw the emergence of the field called computer science, and several important university departments were founded during the decade, at Stanford and Carnegie Mellon in 1965 and at MIT in 1968. Hardware platforms had stabilized enough to support a community of researchers who attacked a common set of problems. New languages proliferated, often initiated by government and buoyed by the needs of commercial industry. The Navy had sponsored Grace Hopper and others during the 1950s to develop automatic programming techniques that became the first compilers. John Backus and a group at IBM developed FORTRAN, which was distributed to IBM users in 1957. A team led by John McCarthy at MIT (with government support) began implementing LISP in 1958, and the language became widely used, particularly for artificial intelligence programming, in the early 1960s. In 1959, the Pentagon began convening a group of computer experts from government, academia, and industry to define common business languages for computers. The group published a specification in 1959, and by 1960 RCA and Remington Rand Univac had produced the first COBOL compilers. By the beginning of the 1960s, a number of computer languages, standard across numerous hardware platforms, were beginning to define programming as a task, as a profession, and as a challenging and legitimate subject of intellectual inquiry.

The Changing Federal Role

(From pp. 98-107): The forces driving government support changed during the 1960s. The Cold War remained a paramount concern, but to it were added the difficult conflict in Vietnam, the Great Society programs, and the Apollo program, inaugurated by President Kennedy's 1961 challenge. New political goals, new technologies, and new missions provoked changes in the federal agency population. Among these, two agencies became particularly important in computing: the new Advanced Research Projects Agency and the National Science Foundation.

The Advanced Research Projects Agency

The founding of the Advanced Research Projects Agency (ARPA) in 1958, a direct outgrowth of the Sputnik scare, had immeasurable impact on computing and communications. ARPA, specifically charged with preventing technological surprises like Sputnik, began conducting long-range, high-risk research. It was originally conceived as the DOD's own space agency, reporting directly to the Secretary of Defense in order to avoid interservice rivalry. Space, like computing, did not seem to fit into the existing military service structure. ARPA's independent status not only insulated it from established service interests but also tended to foster radical ideas and keep the agency tuned to basic research questions: when the agency-supported work became too much like systems development, it ran the risk of treading on the territory of a specific service.

ARPA's status as the DOD space agency did not last long. Soon after NASA's creation in 1958, ARPA retained essentially no role as a space agency. ARPA instead focused its energies on ballistic missile defense, nuclear test detection, propellants, and materials. It also established a critical organizational infrastructure and management style: a small, high-quality managerial staff, supported by scientists and engineers on rotation from industry and academia, successfully employing existing DOD laboratories and contracting procedures (rather than creating its own research facilities) to build solid programs in new, complex fields. ARPA also emerged as an agency extremely sensitive to the personality and vision of its director.

ARPA's decline as a space agency raised questions about its role and character. A new director, Jack Ruina, answered the questions in no uncertain terms by cementing the agency's reputation as an elite, scientifically respected institution devoted to basic, long-term research projects. Ruina, ARPA's first scientist-director, took office at the same time as Kennedy and McNamara in 1961, and brought a similar spirit to the

agency. Ruina decentralized management at ARPA and began the tradition of relying heavily on independent office directors and program managers to run research programs. Ruina also valued scientific and technical merit above immediate relevance to the military. Ruina believed both of these characteristics—independence and intellectual quality—were critical to attracting the best people, both to ARPA as an organization and to ARPA-sponsored research. Interestingly, ARPA's managerial success did not rely on innovative managerial techniques per se (such as the computerized project scheduling typical of the Navy's Polaris project) but rather on the creative use of existing mechanisms such as "no-year money," unsolicited proposals, sole-source procurement, and multiyear forward funding.

ARPA and Information Technology. From the point of view of computing, the most important event at ARPA in the early 1960s, indeed in all of ARPA's history, was the establishment of the Information Processing Techniques Office, IPTO, in 1962. The impetus for this move came from several directions, including Kennedy's call a year earlier for improvements in command-and-control systems to make them "more flexible, more selective, more deliberate, better protected, and under ultimate civilian authority at all times." Computing as applied to command and control was the ideal ARPA program—it had no clearly established service affinity; it was "a new area with relatively little established service interest and entailed far less constraint on ARPA's freedom of action," than more familiar technologies. Ruina established IPTO to be devoted not to command and control but to the more fundamental problems in computing that would, eventually, contribute solutions.

Consistent with his philosophy of strong, independent, and scientific office managers, Ruina appointed J.C.R. Licklider to head IPTO. The Harvard-trained psychologist came to ARPA in October 1962, primarily to run its Command and Control Group. Licklider split that group into two discipline-oriented offices: Behavioral Sciences Office and IPTO. Licklider had had extensive exposure to the computer research of the time and had clearly defined his own vision of "man-computer symbiosis," which he had published in a landmark paper of 1960 by the same name. He saw human-computer interaction as the key, not only to command and control, but also to bringing together the then-disparate techniques of electronic computing to form a unified science of computers as tools for augmenting human thought and creativity. Licklider formed IPTO in this image, working largely independently of any direction from Ruina, who spent the majority of his time on higher-profile and higher-funded missile defense issues. Licklider's timing was opportune: the 1950s had produced a stable technology of digital computer hardware, and the big systems

projects had shown that programming these machines was a difficult but interesting problem in its own right. Now the pertinent questions concerned how to use "this tremendous power . . . for other than purely numerical scientific calculations." Licklider not only brought this vision to IPTO itself, but he also promoted it with missionary zeal to the research community at large. Licklider's and IPTO's success derived in large part from their skills at "selling the vision" in addition to "buying the research."

Another remarkable feature of IPTO, particularly during the 1960s, was its ability to maintain the coherent vision over a long period of time; the office director was able to handpick his successor. Licklider chose Ivan Sutherland, a dynamic young researcher he had encountered as a graduate student at MIT and the Lincoln Laboratory, to succeed him in 1964. Sutherland carried on Licklider's basic ideas and made his own impact by emphasizing computer graphics. Sutherland's own successor, Robert Taylor, came in 1966 from a job as a program officer at NASA and recalled, "I became heartily subscribed to the Licklider vision of interactive computing." While at IPTO, Taylor emphasized networking. The last IPTO director of the 1960s, Lawrence Roberts, came, like Sutherland, from MIT and Lincoln Laboratory, where he had worked on the early transistorized computers and had conducted ARPA research in both graphics and communications.

During the 1960s, ARPA and IPTO had more effect on the science and technology of computing than any other single government agency, sometimes raising concern that the research agenda for computing was being directed by military needs. IPTO's sheer size, $15 million in 1965, dwarfed other agencies such as ONR. Still, it is important to note, ONR and ARPA worked closely together; ONR would often let small contracts to researchers and serve as a talent agent for ARPA, which would then fund promising projects at larger scale. ARPA combined the best features of existing military research support with a new, lean administrative structure and innovative management style to fund high-risk projects consistently. The agency had the freedom to administer large block grants as well as multiple-year contracts, allowing it the luxury of a long-term vision to foster technologies, disciplines, and institutions. Further, the national defense motivation allowed IPTO to concentrate its resources on centers of scientific and engineering excellence (such as MIT, Carnegie Mellon University, and Stanford University) without regard for geographical distribution questions with which NSF had to be concerned. Such an approach helped to create university-based research groups with the critical mass and stability of funding needed to create significant advances in particular technical areas. But although it trained generations of young researchers in those areas, ARPA's funding style did little to help them pursue the

same lines of work at other universities. As an indirect and possibly unintended consequence, the research approaches and tools and the generic technologies developed under ARPA's patronage were disseminated more rapidly and widely, and so came to be applied in new nonmilitary contexts by the young M.S. and Ph.D. graduates who had been trained in that environment but could not expect to make their research careers within it.

ARPA's Management Style. To evaluate research proposals, IPTO did not employ the peer-review process like NSF, but rather relied on internal reviews and the discretion of program managers as did ONR. These program managers, working under office managers such as Licklider, Sutherland, Taylor, and Roberts, came to have enormous influence over their areas of responsibility and became familiar with the entire field both personally and intellectually. They had the freedom and the resources to shape multiple R&D contracts into a larger vision and to stimulate new areas of inquiry. The education, recruiting, and responsibilities of these program managers thus became a critical parameter in the character and success of ARPA programs. ARPA frequently chose people who had training and research experience in the fields they would fund, and thus who had insight and opinions on where those fields should go.

To have such effects, the program managers were given enough funds to let a large enough number of contracts and to shape a coherent research program, with minimal responsibilities for managing staffs. Program budgets usually required only two levels of approval above the program manager: the director of IPTO and the director of ARPA. One IPTO member described what he called "the joy of ARPA. . . . You know, if a program manager has a good idea, he has got two people to convince that that is a good idea before the guy goes to work. He has got the director of his office and the director of ARPA, and that is it. It is such a short chain of command."

Part of ARPA's philosophy involved aiming at radical change rather than incremental improvement. As Robert Taylor put it, for example, incremental innovation would be taken care of by the services and their contractors, but, ARPA's aim was "an order of magnitude difference." ARPA identified good ideas and magnified them. This strategy often necessitated funding large, group-oriented projects and institutions rather than individuals. Taylor recalled, "I don't remember a single case where we ever funded a single individual's work. . . . The individual researcher who is just looking for support for his own individual work could [potentially] find many homes to support that work. So we tended not to fund those, because we felt that they were already pretty well covered. Instead, we funded larger groups—teams." NSF's peer-review process worked

well for individual projects, but was not likely to support large, team-oriented research projects. Nor did it, at this point in history, support entire institutions and research centers, like the Laboratory for Computer Science at MIT. IPTO's style meshed with its emphasis on human-machine interaction, which it saw as fundamentally a systems problem and hence fundamentally team oriented. In Taylor's view, the university reward structure was much more oriented toward individual projects, so "systems research is most difficult to fund and manage in a university." This philosophy was apparent in ARPA's support of Project MAC, an MIT-led effort on time-shared computing. . . .

ARPA, with its clearly defined mission to support DOD technology, could also afford to be elitist in a way that NSF, with a broader charter to support the country's scientific research, could not. "ARPA had no commitment, for example, to take geography into consideration when it funded work." Another important feature of ARPA's multiyear contracts was their stability, which proved critical for graduate students who could rely on funding to get them through their Ph.D. program. ARPA also paid particular attention to building communities of researchers and disseminating the results of its research, even beyond traditional publications. IPTO would hold annual meetings for its contract researchers at which results would be presented and debated. These meetings proved effective not only at advancing the research itself but also at providing valuable feedback for the program managers and helping to forge relationships between researchers in related areas. Similar conferences were convened for graduate students only, thus building a longer-term community of researchers. ARPA also put significant effort into getting the results of its research programs commercialized so that DOD could benefit from the development and expansion of a commercial industry for information technology. ARPA sponsored conferences that brought together researchers and managers from academia and industry on topics such as time-sharing, for example.

Much has been made of ARPA's management style, but it would be a mistake to conclude that management per se provided the keys to the agency's successes in computing. The key point about the style, in fact, was its light touch. Red tape was kept to a minimum, and project proposals were turned around quickly, frequently into multiple-year contracts. Typical DOD research contracts involved close monitoring and careful adherence to requirements and specifications. ARPA avoided this approach by hiring technically educated program managers who had continuing research interests in the fields they were managing. This reality counters the myth that government bureaucrats heavy-handedly selected R&D problems and managed the grants and contracts. Especially during the 1960s and 1970s, program managers and office directors were not

bureaucrats but were usually academics on a 2-year tour of duty. They saw ARPA as a pulpit from which to preach their visions, with money to help them realize those visions. The entire system displayed something of a self-organizing, self-managing nature. As Ivan Sutherland recalled, "Good research comes from the researchers themselves rather than from the outside."

National Science Foundation

While ARPA was focusing on large projects and systems, the National Science Foundation played a large role in legitimizing basic computer science research as an academic discipline and in funding individual researchers at a wide range of institutions. Its programs in computing have evolved considerably since its founding in 1950, but have tended to balance support for research, education, and computing infrastructure. Although early programs tended to focus on the use of computing in other academic disciplines, NSF subsequently emerged as the leading federal funder of basic research in computer science.

NSF was formed before computing became a clearly defined research area, and it established divisions for chemistry, physics, and biology, but not computing. NSF did provide support for computing in its early years, but this support derived more from a desire to promote computer-related activities in other disciplines than to expand computer science as a discipline, and as such was weighted toward support for computing infrastructure. For example, NSF poured millions of dollars into university computing centers so that researchers in other disciplines, such as physics and chemistry, could have access to computing power. NSF noted that little computing power was available to researchers at American universities who were not involved in defense-related research and that "many scientists feel strongly that further progress in their field will be seriously affected by lack of access to the techniques and facilities of electronic computation." As a result, NSF began supporting computing centers at universities in 1956 and, in 1959, allocated a budget specifically for computer equipment purchases. Recognizing that computing technology was expensive, became obsolete rapidly, and entailed significant costs for ongoing support, NSF decided that it would, in effect, pay for American campuses to enter the computer age. In 1962, it established its first office devoted to computing, the program for Computers and Computing Science within the Mathematical Sciences Division. By 1970, the Institutional Computing Services (or Facilities) program had obligated $66 million to university computing centers across the country. NSF intended that use of the new facilities would result in trained personnel to fulfill increasing needs for computer proficiency in industry, government, and academia.

NSF provided some funding for computer-related research in its early years. Originally, such funding came out of the mathematics division in the 1950s and grew out of an interest in numerical analysis. By 1955, NSF began to fund basic research in computer science theory with its first grants for the research of recursion theory and one grant to develop an analytical computer program under the Mathematical Sciences Program. Although these projects constituted less than 10 percent of the mathematics budget, they resulted in significant research.

In 1967, NSF united all the facets of its computing support into a single office, the Office of Computing Activities (OCA). The new office incorporated elements from the directorates of mathematics and engineering and from the Facilities program, unifying NSF's research and infrastructure efforts in computing. It also incorporated an educational element that was intended to help meet the radically increasing demand for instruction in computer science. The OCA was headed by Milton Rose, the former head of the Mathematical Sciences Section, and reported directly to the director of NSF.

Originally, the OCA's main focus was improving university computing services. In 1967, $11.3 million of the office's $12.8 million total budget went toward institutional support. Because not all universities were large enough to support their own computing centers but would benefit from access to computing time at other universities, the OCA also began to support regional networks linking many universities together. In 1968, the OCA spent $5.3 million, or 18.6 percent of its budget, to provide links between computers in the same geographic region. In the 1970s, the computer center projects were canceled, however, in favor of shifting emphasis toward education and research.

Beginning in 1968, through the Education and Training program, the OCA began funding the inauguration of university-level computer science programs. NSF funded several conferences and studies to develop computer science curricula. The Education and Training program obligated $12.3 million between 1968 and 1970 for training, curricula development, and support of computer-assisted instruction.

Although the majority of the OCA's funding was spent on infrastructure and education, the office also supported a broad range of basic computer science research programs. These included compiler and language development, theoretical computer science, computation theory, numerical analysis, and algorithms. The Computer Systems Design program concentrated on computer architecture and systems analysis. Other programs focused on topics in artificial intelligence, including pattern recognition and automatic theory proving.

1970-1990: Retrenching and International Competition

(From p. 107): Despite previous successes, the 1970s opened with computing at a critical but fragile point. Although produced by a large and established industry, commercial computers remained the expensive, relatively esoteric tools of large corporations, research institutions, and government. Computing had not yet made its way to the common user, much less the man in the street. This movement would begin in the mid-1970s with the introduction of the microprocessor and then unfold in the 1980s with even greater drama and force. If the era before 1960 was one of experimentation and the 1960s one of consolidation and diffusion in computing, the two decades between 1970 and 1990 were characterized by explosive growth. Still, this course of events was far from clear in the early 1970s.

Accomplishing Federal Missions

(From pp. 141-142): In addition to supporting industrial innovation and the economic benefits that it brings, federal support for computing research has enabled government agencies to accomplish their missions. Investments in computing research by the Department of Energy (DOE), the National Aeronautics and Space Administration (NASA), and the National Institutes of Health (NIH), as well as the Department of Defense (DOD), are ultimately based on agency needs. Many of the missions these agencies must fulfill depend on computing technologies. DOD, for example, has maintained a policy of achieving military superiority over potential adversaries not through numerical superiority (i.e., having more soldiers) but through better technology. Computing has become a central part of information gathering, management, and analysis for commanders and soldiers alike.

Similarly, DOE and its predecessors would have been unable to support their mission of designing nuclear weapons without the simulation capabilities of large supercomputers. Such computers have retained their value to DOE as its mission has shifted toward stewardship of the nuclear stockpile in an era of restricted nuclear testing. Its Accelerated Strategic Computing Initiative builds on DOE's earlier success by attempting to support development of simulation technologies needed to assess nuclear weapons, analyze their performance, predict their safety and reliability, and certify their functionality without testing them. In addition, NASA could not have accomplished its space exploration or its Earth observation and monitoring missions without reliable computers for controlling spacecraft and managing data. New computing capabilities, including the World Wide Web, have enabled the National Library of Medicine to expand access to medical information and have provided tools for researchers who are sequencing the human genome.

EVOLVING THE HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS INITIATIVE TO SUPPORT THE NATION'S INFORMATION INFRASTRUCTURE (1995)

CITATION: Computer Science and Telecommunications Board (CSTB), National Research Council. 1995. *Evolving the High Performance Computing and Communications Initiative to Support the Nation's Information Infrastructure.* National Academy Press, Washington, D.C.

Continued Federal Investment Is Necessary to Sustain Our Lead

(From pp. 23-25): What must be done to sustain the innovation and growth needed for enhancing the information infrastructure and maintaining U.S. leadership in information technology? Rapid and continuing change in the technology, a 10- to 15-year cycle from idea to commercial success, and successive waves of new companies are characteristics of the information industry that point to the need for a stable source of expertise and some room for a long-term approach. Three observations seem pertinent.

1. *Industrial R&D cannot replace government investment in basic research.* Very few companies are able to invest for a payoff that is 10 years away. Moreover, many advances are broad in their applicability and complex enough to take several engineering iterations to get right, and so the key insights become "public" and a single company cannot recoup the research investment. Public investment in research that creates a reservoir of new ideas and trained people is repaid many times over by jobs and taxes in the information industry, more innovation and productivity in other industries, and improvements in the daily lives of citizens. This investment is essential to maintain U.S. international competitiveness. . . .

Because of the long time scales involved in research, the full effect of decreasing investment in research may not be evident for a decade, but by then, it may be too late to reverse an erosion of research capability. Thus, even though many private-sector organizations that have weighed in on one or more policy areas relating to the enhancement of information infrastructure typically argue for a minimal government role in commercialization, they tend to support a continuing federal presence in relevant basic research.

2. *It is hard to predict which new ideas and approaches will succeed.* Over the years, federal support of computing and communications research in universities has helped make possible an environment for exploration and experimentation, leading to a broad range of diverse ideas from which the marketplace ultimately has selected winners and losers. . . . [I]t is

difficult to know in advance the outcome or final value of a particular line of inquiry. But the history of development in computing and communications suggests that innovation arises from a diversity of ideas and some freedom to take a long-range view. It is notoriously difficult to place a specific value on the generation of knowledge and experience, but such benefits are much broader than sales of specific systems.

3. *Research and development in information technology can make good use of equipment that is 10 years in advance of current "commodity" practice.* When it is first used for research, such a piece of equipment is often a supercomputer. By the time that research makes its way to commercial use, computers of equal power are no longer expensive or rare. . . .

The large-scale systems problems presented both by massive parallelism and by massive information infrastructure are additional distinguishing characteristics of information systems R&D, because they imply a need for scale in the research effort itself. In principle, collaborative efforts might help to overcome the problem of attaining critical mass and scale, yet history suggests that there are relatively few collaborations in basic research within any industry, and purely industrial (and increasingly industry-university or industry-government) collaborations tend to disseminate results more slowly than university-based research.

The government-supported research program . . . is small compared to industrial R&D . . . but it constitutes a significant portion of the research component, and it is a critical factor because it supports the exploratory work that is difficult for industry to afford, allows the pursuit of ideas that may lead to success in unexpected ways, and nourishes the industry of the future, creating jobs and benefits for ourselves and our children. The industrial R&D investment, though larger in dollars, is different in nature: it focuses on the near term—increasingly so, as noted earlier—and is thus vulnerable to major opportunity costs. The increasing tendency to focus on the near term is affecting the body of the nation's overall R&D. Despite economic studies showing that the United States leads the world in reaping benefits from basic research, pressures in all sectors appear to be promoting a shift in universities toward near-term efforts, resulting in a decline in basic research even as a share of university research. Thus, a general reduction in support for basic research appears to be taking place.

It is critical to understand that there are dramatic new opportunities that still can be developed by fundamental research in information technology—opportunities on which the nation must capitalize. These include high-performance systems and applications for science and engineering; high-confidence systems for applications such as health care, law enforcement, and finance; building blocks for global-scale information utilities (e.g., electronic payment); interactive environments for applica-

tions ranging from telemedicine to entertainment; improved user interfaces to allow the creation and use of ever more sophisticated applications by ever broader cross sections of the population; and the creation of the human capital on which the next generation's information industries will be based. Fundamental research in computing and communications is the key to unlocking the potential of these new applications.

How much federal research support is proper for the foreseeable future and to what aspects of information technology should it be devoted? Answering this question is part of a larger process of considering how to reorient overall federal spending on R&D from a context dominated by national security to one driven more by other economic and social goals. It is harder to achieve the kind of consensus needed to sustain federal research programs associated with these goals than it was under the national security aegis. Nevertheless, the fundamental rationale for federal programs remains:

> That R&D can enhance the nation's economic welfare is not, by itself, sufficient reason to justify a prominent role for the federal government in financing it. Economists have developed a further rationale for government subsidies. Their consensus is that most of the benefits of innovation accrue not to innovators but to consumers through products that are better or less expensive, or both. Because the benefits of technological progress are broadly shared, innovators lack the financial incentive to improve technologies as much as is socially desirable. Therefore, the government can improve the performance of the economy by adopting policies that facilitate and increase investments in research. [Linda R. Cohen and Roger G. Noll. 1994. "Privatizing Public Research," *Scientific American* 271(3): 73]

What Is CSTB?

As a part of the National Research Council, the Computer Science and Telecommunications Board (CSTB) was established in 1986 to provide independent advice to the federal government on technical and public policy issues relating to computing and communications. Composed of leaders from industry and academia, CSTB conducts studies of critical national issues and makes recommendations to government, industry, and academia. CSTB also provides a neutral meeting ground for consideration of complex issues where resolution and action may be premature. It convenes discussions that bring together principals from the public and private sectors, assuring consideration of key perspectives. The majority of CSTB's work is requested by federal agencies and Congress, consistent with its National Academies context.

A pioneer in framing and analyzing Internet policy issues, CSTB is unique in its comprehensive scope and effective, interdisciplinary appraisal of technical, economic, social, and policy issues. Beginning with early work in computer and communications security, cyber-assurance and information systems trustworthiness have been a cross-cutting theme in CSTB's work. CSTB has produced several reports known as classics in the field, and it continues to address these topics as they grow in importance.

To do its work, CSTB draws on some of the best minds in the country and from around the world, inviting experts to participate in its projects as a public service. Studies are conducted by balanced committees without direct financial interests in the topics they are addressing. Those

committees meet, confer electronically, and build analyses through their deliberations. Additional expertise is tapped in a rigorous process of review and critique, further enhancing the quality of CSTB reports. By engaging groups of principals, CSTB gets the facts and insights critical to assessing key issues.

The mission of CSTB is to

- *Respond to requests* from the government, nonprofit organizations, and private industry for advice on computer and telecommunications issues and from the government for advice on computer and telecommunications systems planning, utilization, and modernization;
- *Monitor and promote the health of the fields* of computer science and telecommunications, with attention to issues of human resources, information infrastructure, and societal impacts;
- *Initiate and conduct studies* involving computer science, technology, and telecommunications as critical resources; and
- *Foster interaction* among the disciplines underlying computing and telecommunications technologies and other fields, at large and within the National Academies.

CSTB projects address a diverse range of topics affected by the evolution of information technology. Recently completed reports include *Beyond Productivity: Information Technology, Innovation, and Creativity; Cybersecurity Today and Tomorrow: Pay Now or Pay Later; Youth, Pornography, and the Internet; Broadband: Bringing Home the Bits; The Digital Dilemma: Intellectual Property in the Information Age; IDs—Not That Easy: Questions About Nationwide Identity Systems; The Internet Under Crisis Conditions: Learning from September 11;* and *IT Roadmap to a Geospatial Future.* For further information about CSTB reports and active projects, see <http://cstb.org>.

Issues and Techniques in Touch-Sensitive Tablet Input

William Buxton
Ralph Hill
Peter Rowley

Computer Systems Research Institute
University of Toronto
Toronto, Ontario
Canada M5S 1A4

(416) 978-6320

Abstract

Touch-sensitive tablets and their use in human-computer interaction are discussed. It is shown that such devices have some important properties that differentiate them from other input devices (such as mice and joysticks). The analysis serves two purposes: (1) it sheds light on touch tablets, and (2) it demonstrates how other devices might be approached. Three specific distinctions between touch tablets and one button mice are drawn. These concern the *signaling of events*, *multiple point sensing* and the use of *templates*. These distinctions are reinforced, and possible uses of touch tablets are illustrated, in an example application. Potential enhancements to touch tablets and other input devices are discussed, as are some inherent problems. The paper concludes with recommendations for future work.

CR Categories and Subject Descriptors: I.3.1 [**Computer Graphics**]: Hardware Architecture: Input Devices. I.3.6 [**Computer Graphics**]: Methodology and Techniques: Device Independence, Ergonomics, Interaction Techniques.

General Terms: Design, Human Factors.

Additional Keywords and Phrases: touch sensitive input devices.

1. Introduction

Increasingly, research in human-computer interaction is focusing on problems of input [Foley, Wallace & Chan 1984; Buxton 1983; Buxton 1985]. Much of this attention is directed towards input technologies. The ubiquitous Sholes keyboard is being replaced and/or complemented by alternative technologies. For example, a major focus of the marketing strategy for two recent personal computers, the Apple Macintosh and Hewlett-Packard 150, has been on the input devices that they employ (the mouse and touch-screen, respectively).

Now that the range of available devices is expanding, how does one select the best technology for a particular application? And once a technology is chosen, how can it be used most effectively? These questions are important, for as Buxton [1983] has argued, the ways in which the user *physically* interacts with an input device have a marked effect on the type of user interface that can be effectively supported.

In the general sense, the objective of this paper is to help in the selection process and assist in effective use of a specific class of devices. Our approach is to investigate a specific class of devices: touch-sensitive tablets. We will identify touch tablets, enumerate their important properties, and compare them to a more common input device, the mouse. We then go on to give examples of transactions where touch tablets can be used effectively. There are two intended benefits for this approach. First, the reader will acquire an understanding of touch tablet issues. Second, the reader will have a concrete example of how the technology can be investigated, and can utilize the approach as a model for investigating other classes of devices.

2. Touch-Sensitive Tablets

A touch-sensitive tablet (touch tablet for short) is a flat surface, usually mounted horizontally or nearly horizontally, that can sense the location of a finger pressing on it. That is, it is a tablet that can sense that it is being touched, and where it is being

touched. Touch tablets can vary greatly in size, from a few inches on a side to several feet on a side. The most critical requirement is that the user is not required point with some manually held device such as a stylus or puck.

What we have described in the previous paragraph is a *simple* touch tablet. Only one point of contact is sensed, and then only in a binary, touch/no touch, mode. One way to extend the potential of a simple touch tablet is to sense the degree, or pressure, of contact. Another is to sense multiple points of contact. In this case, the location (and possibly pressure) of several points of contact would be reported. Most tablets currently on the market are of the "simple" variety. However, Lee, Buxton and Smith [1985], and Nakatani [private communication] have developed prototypes of multi-touch, multi-pressure sensing tablets.

We wish to stress that we will restrict our discussion of touch technologies to touch tablets, which can and should be used in ways that are different from touch screens. Readers interested in touch-screen technology are referred to Herot & Weinsapfel [1978], Nakatani & Rohrlich [1983] and Minsky [1984]. We acknowledge that a flat touch screen mounted horizontally is a touch tablet as defined above. This is not a contradiction, as a touch screen has exactly the properties of touch tablets we describe below, as long as there is no attempt to mount a display below (or behind) it or to make it the center of the user's visual focus.

Some sources of touch tablets are listed in Appendix A.

3. Properties of Touch-Sensitive Tablets

Asking "Which input device is best?" is much like asking "How long should a piece of string be?" The answer to both is: it depends on what you want to use it for. With input devices, however, we are limited in our understanding of the relationship between device properties and the demands of a specific application. We will investigate touch tablets from the perspective of improving our understanding of this relationship. Our claim is that other technologies warrant similar, or even more detailed, investigation.

Touch tablets have a number of properties that distinguish them from other devices:

- They have no mechanical intermediate device (such as stylus or puck). Hence they are useful in hostile environments (e.g., classrooms, public access terminals) where such intermediate devices can get lost, stolen, or damaged.
- Having no puck to slide or get bumped, the tracking symbol "stays put" once placed, thus making them well suited for pointing tasks in environments subject to vibration or motion (e.g., factories, cockpits).
- They present no mechanical or kinesthetic restrictions on our ability to indicate more than one point at a time. That is, we can use two hands or more than one finger simultaneously on a single tablet. (Remember, we can manually control at most two mice at a time: one in each hand. Given that we have ten fingers, it is conceivable that we may wish to indicate more than two points simultaneously. An example of such an application appears below).
- Unlike joysticks and trackballs, they have a very low profile and can be integrated into other equipment such as desks and low-profile keyboards (e.g., the Key Tronic Touch Pad, see Appendix A). This has potential benefits in portable systems, and, according to the Keystroke model of Card, Newell and Moran [1980], reduces homing time from the keyboard to the pointing device.
- They can be molded into one-piece constructions thus eliminating cracks and grooves where dirt can collect. This makes them well suited for very clean environments (eg. hospitals) or very dirty ones (eg., factories).
- Their simple construction, with no moving parts, leads to reliable and long-lived operation, making them suitable for environments where they will be subjected to intense use or where reliability is critical.

They do, of course, have some inherent disadvantages, which will be discussed at the close of the paper.

In the next section we will make three important distinctions between touch tablets and mice. These are:

- Mice and touch tablets vary in the number and types of events that they can transmit. The difference is especially pronounced when comparing to simple touch tablets.
- Touch tablets can be made that can sense multiple points of contact. There is no analogous property for mice.
- The surface of a tablet can be partitioned into regions representing a collection of independent "virtual" devices. This is analogous to the partitioning of a screen into "windows" or virtual displays. Mice, and other devices that transmit "relative change" information, do not lend themselves to this mode of interaction without consuming display real estate for visual feedback. With conventional tablets and touch tablets, graphical, physical or virtual templates can be placed over the input device to delimit regions. This allows valuable screen real estate to be preserved. Physical templates, when combined with touch sensing, permit the operator to sense the regions without diverting the eyes from the primary display during visually demanding tasks.

After these properties are discussed, a simple finger painting program is used to illustrate them in the context of a concrete example. We wish to stress that we do not pretend that the program represents a viable paint program or an optimal interface. It is simply a vehicle to illustrate a variety of transactions in an easily understandable context.

Finally, we discuss improvements that must be made to current touch tablet technology, many of which we have demonstrated in prototype form. Also, we suggest potential improvements to other devices, motivated by our experience with touch technology.

4. Three Distinctions Between Touch Tablets and Mice[1]

The distinctions we make in this section have to do with suitability of devices for certain tasks or use in certain configurations. We are only interested in showing that there are some uses for which touch tablets are not suitable, but other devices are, and vice versa. We make no quantitative claims or comparisons regarding performance.

Signaling

Consider a rubber-band line drawing task with a one button mouse. The user would first position the tracking symbol at the desired starting point of the line by moving the mouse with the button released. The button would then be depressed, to signal the start of the line, and the user would manipulate the line by moving the mouse until the desired length and orientation was achieved. The completion of the line could then be signaled by releasing the button.[2]

Figure 1 is a state diagram that represents this interface. Notice that the button press and release are used to signal the beginning and end of the rubber-band drawing task. Also note that in states 1 and 2 both motion and signaling (by pressing or releasing the button, as appropriate) are possible.

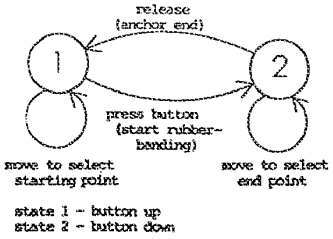


Figure 1. State diagram for rubber-banding with a one-button mouse.

Now consider a simple touch tablet. It can be used to position the tracking symbol at the starting point of the line, but it cannot generate the signal needed to initiate rubber-banding. Figure 2 is a state diagram representation of the capabilities of a simple touch tablet. In state 0, there is no contact with the tablet.[3] In this state only one action is possible: the user may touch the tablet. This causes a change to state 1. In state 1, the user is pressing on the tablet, and as a consequence position reports are sent to the host. There is no way to signal a change to some other state, other than to release (assuming the exclusion of temporal or spatial cues, which tend to be clumsy and difficult to learn). This returns the system to state 0. This signal could not be used to initiate rubber-banding, as it could also mean that the user is pausing to think, or wishes to initiate some other activity.

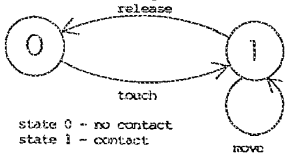


Figure 2. Diagram for showing states of simple touch-tablet.

This inability to signal while pointing is a severe limitation with current touch tablets, that is, tablets that do not report pressure in addition to location. (It is also a property of trackballs, and joysticks without "fire" buttons). It renders them unsuitable for use in many common interaction techniques for which mice are well adapted (e.g., selecting and dragging objects into position, rubber-band line drawing, and pop-up menu selection); techniques that are especially characteristic of interfaces based on *Direct Manipulation* [Shneiderman 1983].

One solution to the problem is to use a separate function button on the keyboard. However, this usually means two-handed input where one could do, or, awkward co-ordination in controlling the button and pointing device with a single hand. An alternative solution when using a touch tablet is to provide some level of pressure sensing. For example, if the tablet could report two levels of contact pressure (i.e., hard and soft), then the transition from soft to hard pressure, and vice versa, could be used for signaling. In effect, pressing hard is equivalent to pressing the button on the mouse. The state diagram showing the rubber-band line drawing task with this form of touch tablet is shown in Figure 3.[4]

As an aside, using this pressure sensing scheme would permit us to select options from a menu, or

---

[1] Although we are comparing touch tablets to one button mice throughout this section, most of the comments apply equally to tablets with one-button pucks or (with some caveats) tablets with styli.

[2] This assumes that the interface is designed so that the button is held down during drawing. Alternatively, the button can be released during drawing, and pressed again, to signal the completion of the line.

[3] We use state 0 to represent a state in which no location information is transmitted. There no analogous state for mice, and hence no state 0 in the diagrams for mice. With conventional tablets, this corresponds to "out of range" state.
At this point the alert reader will wonder about difficulty in distinguishing between hard and soft pressure, and friction (especially when pressing hard). Taking the last first, hard is a relative term. In practice friction need not be a problem (see Inherent Problems, below).

[4] One would conjecture that in the absence of button clicks or other feedback, pressure would be difficult to regulate accurately. We have found two levels of pressure to be easily distinguished, but this is a ripe area for research. For example, Stu Card [private communication] has suggested that the threshold between soft and hard should be reduced (become "softer") while hard pressure is being maintained. This suggestion, and others, warrant formal experimentation.

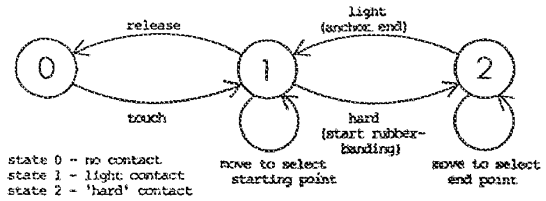


Figure 3. State diagram for rubber-banding with pressure sensing touch tablet.

activate light buttons by positioning the tracking symbol over the item and "pushing". This is consistent with the gesture used with a mouse, and the model of "pushing" buttons. With current simple touch tablets, one does just the opposite: position over the item and then lift off, or "pull" the button.

From the perspective of the signals sent to the host computer, this touch tablet is capable of duplicating the behaviour of a one-button mouse. This is not to say that these devices are equivalent or interchangeable. They are not. They are physically and kinesthetically very different, and should be used in ways that make use of the unique properties of each. Furthermore, such a touch tablet can generate one pair of signals that the one-button mouse cannot — specifically, press and release (transition to and from state 0 in the above diagrams). These signals (which are also available with many conventional tablets) are very useful in implementing certain types of transactions, such as those based on character recognition.

An obvious extension of the pressure sensing concept is to allow continuous pressure sensing. That is, pressure sensing where some large number of different levels of pressure may be reported. This extends the capability of the touch tablet beyond that of a traditional one button mouse. An example of the use of this feature is presented below.

Multiple Position Sensing

With a traditional mouse or tablet, only one position can be reported per device. One can imagine using two mice or possibly two transducers on a tablet, but this increases costs, and two is the practical limit on the number of mice or tablets that can be operated by a single user (without using feet). However, while we have only two hands, we have ten fingers. As playing the piano illustrates, there are some contexts where we might want to use several, or even all of them, at once.

Touch tablets need not restrict us in this regard. Given a large enough surface of the appropriate technology, one could use all fingers of both hands simultaneously, thus providing ten separate units of input. Clearly, this is well beyond the demands of many applications and the capacity of many people, however, there are exceptions. Examples include chording on buttons or switches, operating a set of slide potentiometers, and simple key roll-over when touch typing. One example (using a set of slide potentiometers) will be illustrated below.

Multiple Virtual Devices and Templates

The power of modern graphics displays has been enhanced by partitioning one physical display into a number of virtual displays. To support this, display window managers have been developed. We claim (see Brown, Buxton and Murtagh [1985]) that similar benefits can be gained by developing an input window manager that permits a single physical input device to be partitioned into a number of virtual input devices. Furthermore, we claim that multi-touch tablets are well suited to supporting this approach.

Figure 4a shows a thick cardboard sheet that has holes cut in specific places. When it is placed over a touch tablet as shown in Figure 4b, the user is restricted to touching only certain parts of the tablet. More importantly, the user can *feel* the parts that are touchable, and their shape. Each of the "touchable" regions represents a separate virtual device. The distinction between this template and traditional tablet mounted menus (such as seen in many CAD systems) is important.

Traditionally, the options have been:

a) Save display real estate by mounting the menu on the tablet surface. The cost of this option is eye diversion from the display to the tablet, the inability to "touch type", and time consuming menu changes.

b) Avoid eye diversion by placing the menus on the display. This also make it easier to change menus, but still does not allow "touch typing", and consumes display space.

Touch tablets allow a new option:

c) Save display space and avoid eye diversion by using templates that can be felt, and hence, allow "touch typing" on a variety of virtual input devices. The cost of this option is time consuming menu (template) changes.

It must be remembered that for each of these options, there is an application for which it is best. We have contributed a new option, which makes possible new interfaces. The new possibilities include more elaborate virtual devices because the improved kinesthetic feedback allows the user to concentrate on providing input, instead of staying in the assigned region. We will also show (below) that its main cost (time consuming menu changes) can be reduced in some applications by eliminating the templates.

5. Examples of Transactions Where Touch Tablets Can Be Used Effectively

In order to reinforce the distinctions discussed in the previous section, and to demonstrate the use of touch tablets, we will now work through some examples based on a toy paint system. We wish to stress again that we make no claims about the quality of the example as a paint system. A paint system is a common and easily understood application, and thus, we have chosen to use it simply as a vehicle for discussing interaction techniques that use touch tablets.

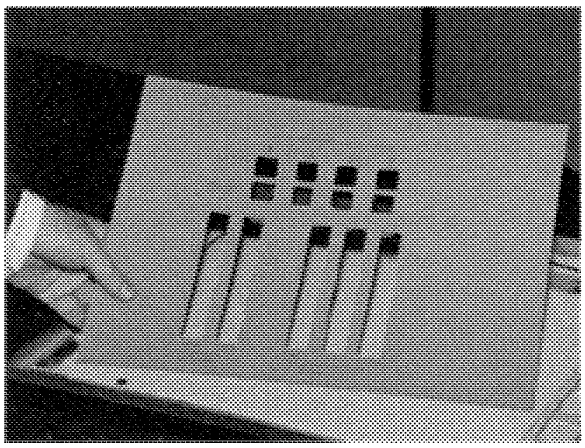


Figure 4a. Sample template.

Figure 5. Main display for paint program.

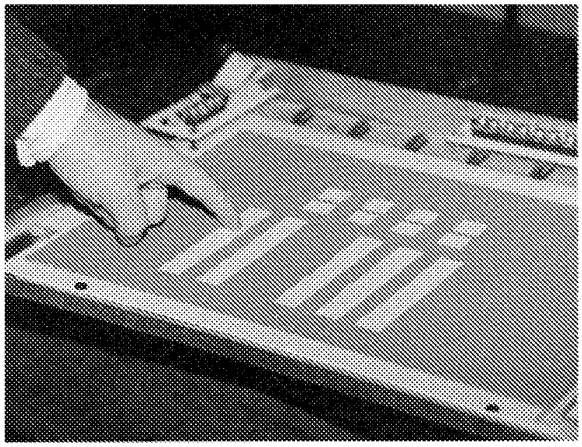


Figure 4b. Sample template in use.

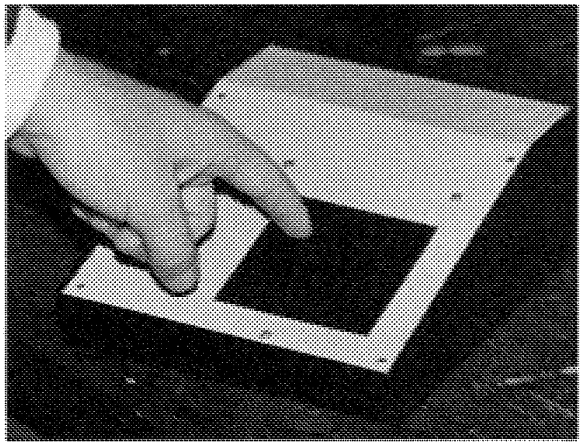


Figure 6. Touch tablet used in demonstrations.

The example paint program allows the creation of simple finger paintings. The layout of the main display for the program is shown in Figure 5. On the left is a large drawing area where the user can draw simple free-hand figures. On the right is a set of menu items. When the lowest item is selected, the user enters a colour mixing mode. In switching to this mode, the user is presented with a different display that is discussed below. The remaining menu items are "paint pots". They are used to select the colour that the user will be painting with.

In each of the following versions of the program, the input requirements are slightly different. In all cases an 8 cm x 8 cm touch tablet is used (Figure 6), but the pressure sensing requirements vary. These are noted in each demonstration.

5.1. Painting Without Pressure Sensing

This version of the paint program illustrates the limitation of having no pressure sensing. Consider the paint program described above, where the only input device is a touch tablet without pressure sensing. Menu selections could be made by pressing down somewhere in the menu area, moving the tracking symbol to the desired menu item and then selecting by releasing. To paint, the user would simply press down in the drawing area and move (see Figure 7 for a representation of the signals used for painting with this program).

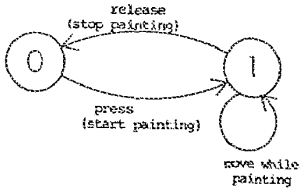


Figure 7. State diagram for drawing portion of simple paint program.

There are several problems with this program. The most obvious is in trying to do detailed drawings. The user does not know where the paint will appear until it appears. This is likely to be too late. Some form of feedback, that shows the user where the brush is, without painting, is needed. Unfortunately, this cannot be done with this input device, as it is not possible to signal the change from tracking to painting and vice versa.

The simplest solution to this problem is to use a button (e.g., a function key on the keyboard) to signal state changes. The problem with this solution is the need to use two hands on two different devices to do one task. This is awkward and requires practice to develop the co-ordination needed to make small rapid strokes in the painting. It is also inefficient in its use of two hands where one could (and normally should) do.

Alternatively, approaches using multiple taps or timing cues for signalling could be tried, however, we have found that these invariably lead to other problems. It is better to find a direct solution using the properties of the device itself.

5.2. Painting with Two Levels of Pressure

This version of the program uses a tablet that reports two levels of contact pressure to provide a satisfactory solution to the signaling problem. A low pressure level (a light touch by the user) is used for general tracking. A heavier touch is used to make menu selections, or to enable painting (see Figure 8 for the tablet states used to control painting with this program). The two levels of contact pressure allow us to make a simple but practical one finger paint program.

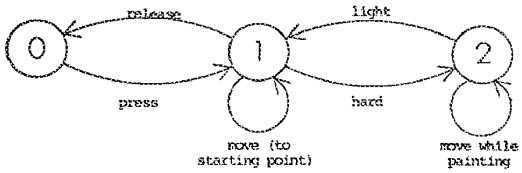


Figure 8. State diagram for painting portion of simple paint program using pressure sensing touch tablet.

This version is very much like using the one button mouse on the Apple Macintosh with MacPaint [Williams, 1984]. Thus, a simple touch tablet is not very useful, but one that reports two levels of pressure is similar in power (but not feel or applicability) to a one button mouse.[5]

5.3. Painting with Continuous Pressure Sensing

In the previous demonstrations, we have only implemented interaction techniques that are common using existing technology. We now introduce a technique that provides functionality beyond that obtainable using most conventional input technologies.

In this technique, we utilize a tablet capable of sensing a continuous range of touch pressure. With this additional signal, the user can control both the width of the paint trail and its path, using only one finger. The new signal, pressure, is used to control width. This is a technique that cannot be used with any mouse that we are aware of, and to our knowledge, is available on only one conventional tablet (the GTCO Digipad with pressure pen [GTCO 1982]).

We have found that using current pressure sensing tablets, the user can accurately supply two to three bits of pressure information, after about 15 minutes practice. This is sufficient for simple doodling and many other applications, but improved pressure resolution is required for high quality painting.

5.4. "Windows" on the Tablet: Colour Selection

We now demonstrate how the surface of the touch tablet can be *dynamically* partitioned into "windows" onto virtual input devices. We use the same basic techniques as discussed under templates (above), but show how to use them without templates. We do this in the context of a colour selection module for our paint program. This module introduces a new display, shown in Figure 9.

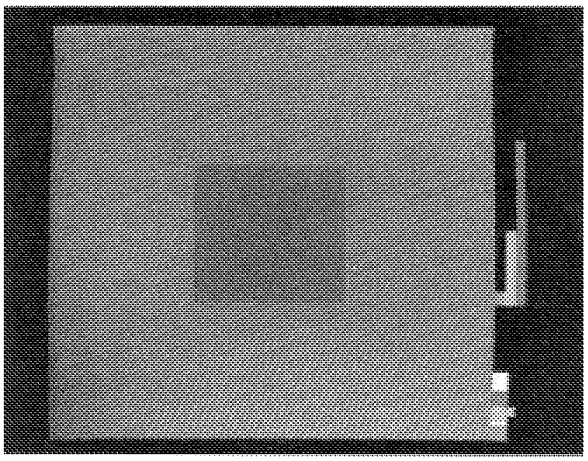


Figure 9. Colour mixing display.

In this display, the large left side consists of a colour patch surrounded by a neutral grey border. This is the patch of colour the user is working on. The right side of the display contains three bar graphs with two light buttons underneath. The primary function of the bar graphs is to provide feedback, representing relative proportions of red, green and blue in the colour patch. Along with the light buttons below, they also serve to remind the user of the current layout of the touch tablet.

In this module, the touch tablet is used as a "virtual operating console". Its layout is shown (to scale) in Figure 10. There are 3 valuators (corresponding to the bar graphs on the screen) used to control

[5] Also, there is the problem of friction, to be discussed below under "Inherent Problems".

colour, and two buttons: one, on the right, to bring up a pop-up menu used to select the colour to be modified, and another, on the left, to exit.

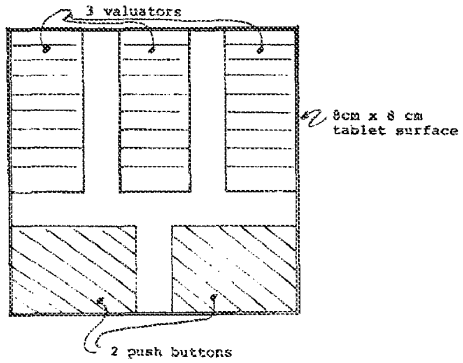


Figure 10. Layout of virtual devices on touch tablet.

The single most important point to be made in this example is that a single *physical* device is being used to implement 5 *virtual* devices (3 valuators and 2 buttons). This is analogous to the use of a display window system, in its goals, and its implementation.

The second main point is that there is nothing on the tablet to delimit the regions. This differs from the use of physical templates as previously discussed, and shows how, in the absence of the need for a physical template, we can instantly change the "windows" on the tablet, without sacrificing the ability to touch type.

We have found that when the tablet surface is small, and the partioning of the surfaces is not too complex, the users very quickly (typically in one or two minutes) learn the positions of the virtual devices relative to the edges of the tablet. More importantly, they can use the virtual devices, practically error free, without diverting attention from the display. (We have repeatedly observed this behaviour in the use of an application that uses a 10 cm square tablet that is divided into 3 sliders with a single button across the top).

Because no template is needed, there is no need for the user to pause to change a template when entering the colour mixing module. Also, at no point is the user's attention diverted from the display. These advantages cannot be achieved with any other device we know of, without consuming display real estate.

The colour of the colour patch is manipulated by *dragging* the red, green and blue values up and down with the valuators on the touch tablet. The valuators are implemented in relative mode (i.e., they are sensitive to changes in position, not absolute position), and are manipulated like one dimensional mice. For example, to make the patch more red, the user presses near the left side of the tablet, about half way to the top, and slides the finger up (see Figure 11). For larger changes, the device can be repeatedly stroked (much like stroking a mouse). Feedback is provided by changing the level in the bar graph on the screen and the colour of the patch.

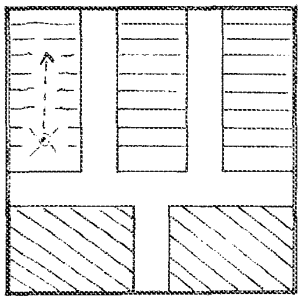


Figure 11. Increasing red content, by pressing on red valuator and sliding up.

Using a mouse, the above interaction could be approximated by placing the tracking symbol over the bars of colour, and dragging them up or down. However, if the bars are narrow, this takes acuity and concentration that distracts attention from the primary task — monitoring the colour of the patch. Furthermore, note that the touch tablet implementation does not need the bars to be displayed at all, they are only a convenience to the user. There are interfaces where, in the interests of maximizing available display area, there will be no items on the display analogous to these bars. That is, there would be nothing on the display to support an interaction technique that allows values to be manipulated by a mouse.

Finally, we can take the example one step further by introducing the use of a touch tablet that can sense multiple points of contact (e.g., [Lee, et al. 1985]). With this technology, all three colour values could be changed at the same time (for example, fading to black by drawing all three sliders down together with three fingers of one hand). This simultaneous adjustment of colours could *not* be supported by a mouse, nor any single commercially available input device we know of. Controlling several valuators with one hand is common in many operating consoles, for example: studio light control, audio mixers, and throttles for multi-engine vehicles (e.g., aircraft and boats). Hence, this example demonstrates a cost effective method for providing functionality that is currently unavailable (or available only at great cost, in the form of a custom fabricated console), but has wide applicability.

5.5. Summary of Examples

Through these simple examples, we have demonstrated several things:

- The ability to sense at least two levels of pressure is a virtual necessity for touch tablets, as without it, auxiliary devices must be used for signaling, and "direct manipulation" interfaces cannot be effectively supported.
- The extension to continuous pressure sensing opens up new possibilities in human-computer interaction.

- Touch tablets are superior to mice and tablets when many simple devices are to be simulated. This is because: (a) there is no need for a mechanical intermediary between the fingers and the tablet surface, (b) they allow the use of templates (including the edges of the tablet, which is a trivial but useful template), and (c) there is no need for positional feedback that would consume valuable display space.
- The ability to sense multiple points of contact radically changes the way in which users may interact with the system. The concept of multiple points of contact does not exist for, nor is it applicable to, current commercially available mice and tablets.

6. Inherent Problems with Touch Tablets

A problem with touch tablets that is annoying in the long term is friction between the user's finger and the tablet surface. This can be a particularly severe problem if a pressure sensitive tablet is used, and the user must make long motions at high pressure. This problem can be alleviated by careful selection of materials and care in the fabrication and calibration of the tablet.[6] Also, the user interface can be designed to avoid extended periods of high pressure.

Perhaps the most difficult problem is providing good feedback to the user when using touch tablets. For example, if a set of push-on/push-off buttons are being simulated, the traditional forms of feedback (illuminated buttons or different button heights) cannot be used. Also, buttons and other controls implemented on touch tablets lack the kinesthetic feel associated with real switches and knobs. As a result, users must be more attentive to visual and audio feedback, and interface designers must be freer in providing this feedback. (As an example of how this might be encouraged, the input "window manager" could automatically provide audible clicks as feedback for button presses).

7. Potential Enhancements to Touch Tablets (and other devices)

The first problem that one notices when using touch tablets is "jitter" when the finger is removed from the tablet. That is, the last few locations reported by the tablet, before it senses loss of contact, tend to be very unreliable.

This problem can be eliminated by modifying the firmware of the touch tablet controller so that it keeps a short FIFO queue of the samples that have most recently be sent to the host. When the user releases pressure, the oldest sample is retransmitted, and the queue is emptied. The length of the queue depends on the properties of the touch tablet (e.g., sensitivity, sampling rate). We have found that determining a suitable value requires only a few minutes of experimentation.

A related problem with most current tablet controllers (not just touch tablets) is that they do not inform the host computer when the user has ceased pressing on the tablet (or moved the puck out of range). This information is essential to the development of certain types of interfaces. (As already mentioned, this signal is not available from mice). Currently, one is reduced to deducing this event by timing the interval between samples sent by the tablet. Since the tablet controller can easily determine when pressure is removed (and must if it is to apply a de-jittering algorithm as above), it should share this information with the host.

Clearly, pressure sensing is an area open to development. Two pressure sensitive tablets have been developed at the University of Toronto [Sasaki, et al. 1981; Lee, et al. 1985]. One has been used to develop several experimental interfaces and was found to be a very powerful tool. They have recently become available from Elographics and Big Briar (see Appendix A). Pressure sensing is not only for touch tablets. Mice, tablet pucks and styli could all benefit by augmenting switches with strain gauges, or other pressure sensing instruments. GTCO, for example, manufactures a stylus with a pressure sensing tip [GTCO 1982], and this, like our pressure sensing touch tablets, has proven very useful.

8. Conclusions

We have shown that there are environments for which some devices are better adapted than others. In particular, touch tablets have advantages in many hostile environments. For this reason, we suggest that there are environments and applications where touch tablets may be the most appropriate input technology.

This being the case, we have enumerated three major distinctions between touch tablets and one button mice (although similar distinctions exist for multi-button mice and conventional tablets). These assist in identifying environments and applications where touch tablets would be most appropriate. These distinctions concern:

- limitation in the ability to signal events,
- suitability for multiple point sensing, and
- the applicability of tactile templates.

These distinctions have been reinforced, and some suggestions on how touch tablets may be used have been given, by discussing a simple user interface. From this example, and the discussion of the distinctions, we have identified some enhancements that can be made to touch tablets and other input devices. The most important of these are pressure sensing and the ability to sense multiple points of contact.

We hope that this paper motivates interface designers to consider the use of touch tablets and shows some ways to use them effectively. Also, we hope it encourages designers and manufacturers of input devices to develop and market input devices with the enhancements that we have discussed.

[6] As a bad example, one commercial "touch" tablet requires so much pressure for reliable sensing that the finger cannot be smoothly dragged across the surface. Instead, a wooden or plastic stylus must be used, thus loosing many of the advantages of touch sensing.

The challenge for the future is to develop touch tablets that sense continuous pressure at multiple points of contact and incorporate them in practical interfaces. We believe that we have shown that this is worthwhile and have shown some practical ways to use touch tablets. However, interface designers must still do a great deal of work to determine where a mouse is better than a touch tablet and vice versa.

Finally, we have illustrated, by example, an approach to the study of input devices, summarized by the credo: "Know the interactions a device is intended to participate in, and the strengths and weaknesses of the device." This approach stresses that there is no such thing as a "good input device," only good interaction task/device combinations.

9. Acknowledgements

The support of this research by the Natural Sciences and Engineering Research Council of Canada is gratefully acknowledged. We are indebted to Kevin Murtagh and Ed Brown for their work on virtual input devices and windowing on input. Also, we are indebted to Elographics Corporation for having supplied us with the hardware on which some of the underlying studies are based.

We would like to thank the referees who provided many useful comments that have helped us with the presentation.

10. References

Brown, E. Buxton, W. Murtagh, K. 1985 — **Windows on Tablets as a Means of Achieving Virtual Input Devices.** Submitted for publication.

Buxton, W. 1983 — **Lexical and Pragmatic Considerations of Input Structures.** *Computer Graphics* 17.1. Presented at the SIGGRAPH Workshop on Graphical Input Techniques, Seattle, Washington, June 1982.

Buxton, W. 1985 — **There is More to Interaction Than Meets the Eye: Some Issues in Manual Input.** (in) Norman, D.A. and Draper, S.W. (Eds.), *User Centered System Design: New Perspectives on Human-Computer Interaction.* Hillsdale, N.J.: Lawrence Erlbaum and Associates. Publication expected late 1985.

Buxton, W. Fiume, E. Hill, R. Lee, A. Woo, C. 1983 — **Continuous Hand-Gesture Driven Input.** *Proceedings Graphics Interface '83*: pp. 191-195. May 9-13, 1983, Edmonton, Alberta.

Card, S.K. Moran, T.P. Newell, A. Jul 1980 — **The Keystroke-Level Model for User Performance Time with Interactive Systems.** *Communications of the ACM* 23.7: pp. 396-409.

Foley, J.D. Wallace, V.L. Chan, P. Nov 1984 — **The Human Factors of Computer Graphics Interaction Techniques.** *IEEE Computer Graphics and Applications* 4.11: pp. 13-48.

GTCO 1982 — **DIGI-PAD 5 User's Manual.** GTCO Corporation, 1055 First Street, Rockville, MD 20850.

Herot, C.F. Weinzapfel, G. Aug 1978 — **One-Point Touch Input of Vector Information for Computer Displays.** *Computer Graphics* 12.3: pp. 210-216. SIGGRAPH'78 Conference Proceedings, August 23-25, 1978, Atlanta, Georgia.

Lee, S. Buxton, W. Smith, K.C. 1985 — **A Multi-Touch Three Dimensional Touch-Sensitive Tablet.** *Human Factors in Computer Systems*: pp. 21-25. (CHI'85 Conference Proceedings, April 14-18, 1985, San Fransisco).

Minsky, M.R. Jul 1984 — **Manipulating Simulated Objects with Real-world Gestures using a Force and Position Sensitive Screen.** *Computer Graphics* 18.3: pp. 195-203. (SIGGRAPH'84 Conference Proceedings, July 23-27, 1984, Minneapolis, Minnesota).

Nakatani, L.H. Rohrlich, J.A. Dec 1983 — **Soft Machines: A Philosophy of User-Computer Interface Design.** *Human Factors in Computing Systems*: pp. 19-23. (CHI'83 Conference Procedings, December 12-15, 1983, Boston).

Sasaki, L. Fedorkow, G. Buxton, W. Retterath, C. Smith, K.C. 1981 — **A Touch Sensitive Input Device.** *Proceedings of the 5th International Conference on Computer Music.* North Texas State University, Denton Texas, November 1981.

Shneiderman, B. Aug 1983 — **Direct Manipulation: A Step Beyond Programming Languages.** *Computer* 16.8: pp. 57-69.

Williams, G. Feb 1984 — **The Apple Macintosh Computer.** *Byte* 9.2: pp. 30-54.

Appendix A: Touch Tablet Sources

Big Briar: 3 by 3 inch continuous pressure sensing touch tablet

Big Briar, Inc.
Leicester, NC
28748

Chalk Board Inc.: "Power Pad", large touch table for micro-computers

Chalk Board Inc.
3772 Pleasantdale Rd.,
Atlanta, GA 30340

Elographics: various sizes of touch tablets, including pressure sensing

Elographics, Inc.
105 Randolph Toad
Oak Ridge, Tennessee
37830
(615)-482-4100

Key Tronic: Keyboard with touch pad.

Keytronic
P.O. Box 14687
Spokane, WA 99214
(509)-928-8000

KoalaPad Technologies: Approx. 5 by 7 inch touch tablet for micro-computers

Koala Technologies
3100 Patrick Henry Drive
Santa Clara, California
95050

Spiral Systems: Trazor Touch Panel, 3 by 3 inch touch tablet

Spiral System Instruments, Inc.
4853 Cordell Avenue, Suite A-10
Bethesda, Maryland
20814

TASA: 4 by 4 inch touch tablet (relative sensing only)

Touch Activated Switch Arrays Inc.
1270 Lawrence Stn. Road, Suite G
Sunnyvale, California
94089

Lexical and Pragmatic Considerations of Input Structures

William Buxton
Computer Systems Research Group
University of Toronto
Toronto, Ontario
Canada M5S 1A4

Introduction

Increased access to computer-based tools has made only too clear the deficiencies in our ability to produce effective user interfaces [1]. Many of our current problems are rooted in our lack of sufficiently powerful theories and methodologies. User interface design remains more of a creative art than a hard science.

Following an age-old technique, the point of departure for much recent work has been to attempt to impose some structure on the problem domain. Perhaps the most significant difference between this work and earlier efforts is the weight placed on considerations falling outside the scope of conventional computer science. The traditional problem-reduction paradigm is being replaced by a holistic approach which views the problem as an integration of issues from computer science, electrical engineering, industrial design, cognitive psychology, psychophysics, linguistics, and kinesthetics.

In the main body of this paper, we examine some of the taxonomies which have been proposed and illustrate how they can serve as useful structures for relating studies in user interface problems. In so doing, we attempt to augment the power of these structures by developing their ability to take into account the effect of gestural and positional factors on the overall effect of the user interface.

Two Taxonomies

One structure for viewing the problem domain of the user interface is provided by Foley and Van Dam [12]. They describe the space in terms of the following four layers:

- conceptual
- semantic
- syntactic
- lexical

The *conceptual* level incorporates the main concepts of the system as seen by the user. Therefore, Foley and Van Dam see it as being equivalent to the *user model.* The *semantic* level incorporates the functionality of the system: what can be expressed. The *syntactic* level defines the grammatical structure of the tokens used to articulate a semantic concept. Finally, the *lexical* component defines the structure of these tokens.

One of the benefits of such a taxonomy is that it can serve as the basis for systems analysis in the design process. It also helps us categorize various user interface studies so as to avoid "apples and bananas" type of comparisons. For example, the studies of Ledgard, Whiteside, Singer and Seymour [16] and Barnard, Hammond, Morton and Long [3] both address issues at the syntactic level. They can, therefore, be compared (which is quite interesting since they give highly contradictory results[1]). On the other hand, by recognizing the "keystroke" model of Card, Moran and Newell [7] as addressing the lexical level, we have a good way of understanding its limitations and comparing it to related studies (such as Embley, Lan, Leinbaugh and Nagy, [8]), or relating it to studies which address different levels (such as the two studies in syntax mentioned above).

While the taxonomy presented by Foley and Van Dam has proven to be a useful tool, our opinion is that it has one major shortcoming. That is, the grain of the lexical level is too coarse to permit the full benefit of the model to be derived. As defined, the authors lump together issues as diverse as:

- how tokens are spelt (for example "add" *vs* "append" *vs* "a" *vs* some graphical icon)

[1] Barnard *et al* invalidate Ledgard *et al's* main thesis that the syntax of natural language is necessarily the best suited for command languages. They demonstrate cases where fixed-field format is less prone to user error than the direct object — indirect object syntax of natural language. A major problem of the paper of Ledgard *et al* is that they did not test many of the interesting cases and then drew conclusions that went beyond what their results supported.

- where items are placed spatially on the display (both in terms of the layout and number of windows, and the layout of data within those windows)
- where devices are placed in the work station
- the type of physical gesture (as determined by the transducer employed) used to articulate a token (pointing with a joystick *vs* a lightpen *vs* a tablet *vs* a mouse, for example)

These issues are sufficiently different to warrant separate treatment. Grouping them under a single heading has the danger of generating confusion comparable to that which could result if no difference was made between the semantic and syntactic levels. Therefore, taking our cue from work in language understanding research in the AI community, we chose to subdivide Foley and Van Dam's lexical level into the following two components:

- lexical: issues having to do with spelling of tokens (*i.e.*, the ordering of lexemes and the nature of the alphabet used — symbolic or iconic, for example).
- pragmatic: issues of gesture, space and devices.

To illustrate the distinction, in the Keystroke model the number of key pushes would be a function of the *lexical* structure while the homing time and pointing time would be a function of *pragmatics*.

Factoring out these two levels helps us focus on the fact that the issues affecting each are different, as is their influence on the overall effect of the user interface. This is illustrated in examples which are presented later in this paper.

It should be pointed out that our isolation of what we have called pragmatic issues is not especially original. We see a similar view in the Command Language Grammar of Moran [18], which is the second main taxonomy which we present. Moran represents the domain of the user interface in terms of three components, each of which is sub-divided into two levels. These are as follows:

- Conceptual Component
  - —task level
  - —semantic level
- Communication Component
  - —syntactic level
  - —interaction level
- Physical Component
  - —spatial level
  - —device level

The *task level* encompasses the set of tasks which the user brings to the system and for which it is intended to serve as a tool. The *semantic level* lays out the conceptual entities of the system and the conceptual operations upon them. As with the Foley and Van Dam model, the *syntactic level* then incorporates the structure of the language within which the semantic level is embedded. The *interaction level* relates the user's physical actions to the conventions of the interactions in the dialogue. The *spatial level* then encompasses issues related to how information is laid out on the display, while the *device level* covers issues such as what types of devices are used and their properties (for example, the effect on user performance if the locator used is a mouse *vs* an isometric joystick *vs* step-keys). (A representative discussion of such issues can be found in Card, English and Burr, [5].)

One subtle but important emphasis in Moran's paper is on the point that it is the effect of the user interface *as a whole* (that is, all levels combined) which constitutes the user's model. The other main difference of his taxonomy, when compared to that of Foley and Van Dam, is his emphasis on the importance of the physical component. A shortcoming, however, lies in the absence of a slot which encapsulates the lexical level as we have defined it above. Like the lexical level (as defined by Foley and Van Dam), the interaction level of Moran appears a little too broad in scope when compared to the other levels in the taxonomy.

Pragmatics

In examining the two studies discussed above, one quickly recognizes that the effect of the pragmatic level on the user interface, and therefore on the user model, is given very little attention. Moran, for example, points out that the physical component exists and that it is important, but does not discuss it further. Foley and Van Dam bury these issues within the lexical level. Our main thesis is that since the primary level of contact with an interactive system is at the level of pragmatics, this level *has one of the strongest effects on the user's perception of the system.* Consequently, the models which we adopt in order to specify, design, implement, compare and evaluate interactive systems *must* be sufficiently rich to capture and communicate the system's properties at this level. This is clearly not the case with most models, and this should be cause for concern. To illustrate this, let us examine a few case studies which relate the effect of pragmatics to:

- pencil-and-paper tests of query languages
- ease of use with respect to action language grammars
- device independence

Pencil-and-Paper Tests

As an aid to the design of effective data base query languages, Reisner [19] has proposed the use of pencil-and-paper tests. Subjects were taught a query language in a class-room environment and then tested as to their

ability to formulate and understand queries. Different control groups were taught different languages. By comparing the test results of the different groups, Reisner drew conclusions as to the relative "goodness" of structure and ease of learning of the different languages. She then made the argument that the technique could be used to find weaknesses in new languages before they are implemented, thereby shortening their development cycle.

While the paper makes some important points, it has a serious defect in that it does not point out the limitations of the technique. The approach does tell us something about the cognitive burden involved in the learning of a query language. But it does not tell us everything. In particular, the technique is totally incapable of taking into account the effect that the means and medium of doing something has on our ability to remember how to do it. To paraphrase McLuhan, the medium does affect the message.

Issues of syntax are not independent of pragmatics, but pencil-and-paper tests cannot take such dependencies into account. For example, consider the role of "muscle memory" in recalling how to perform various tasks. The strength of its influence can be seen in my ability to type quite effectively, even though I am incapable of telling you where the various characters are on my QWERTY keyboard, or in my ability to open a lock whose combination I cannot recite. Yet, this effect will never show up in a pencil-and-paper test. Another example is seen in the technique's inability to take into account the contribution that appropriate feedback and help mechanisms can provide in developing mnemonics and other memory and learning aids.

We are not trying to claim that such pencil-and-paper tests are not of use (although Barnard *et al*, [3], point out some important dangers in using such techniques). We are simply trying to illustrate some of their limitations, and demonstrate that lack of adequate emphasis on pragmatics can result in readers (and authors) drawing false or misleading conclusions from their work. Furthermore, we conjecture that if pragmatics were isolated as a separate level in a taxonomy such as that of Foley and Van Dam, they would be less likely to be ignored.

Complexity and Chunking

In another study, Reisner [20] makes an important contribution by showing how the analysis of the grammar of the "action language" of an interactive system can provide valuable metrics for predicting the ease of use and proneness to error of that system. Thus, an important tool for system design, analysis and comparison is introduced.

The basis of the technique is that the complexity of the grammar is a good metric for the cognitive burden of learning and using the system. Grammar complexity is measured in terms of number of productions and production length. There is a problem, however, which limits our ability to reap the full benefits of the technique. This has to do with the technique's current inability to take into account what we call *chunking*. By this we mean the phenomenon where two or more actions fuse together into a single gesture (in a manner analogous to the formation of a compound word in language). In many cases, the cognitive burden of the resulting aggregate may be the equivalent of a single token. In terms of formal language theory, a non-terminal *when effected by an appropriate compound gesture* may carry the cognitive burden of a single terminal.

Such chunking may be either sequential, parallel or both. Sequentially, it should be recognized that some actions have different degrees of *closure* than others. For example, take two events, each of which is to be triggered by the change of state of a switch. If a foot-switch similar to the high/low beam switch in some cars is used, the down action of a down/up gesture triggers each event. The point to note is that there is no kinesthetic connection between the gesture that triggers one event and that which triggers the other. Each action is complete in itself and, as with driving a car, the operator is free to initiate other actions before changing the state of the switch again.

On the other hand, the same binary function could be controlled by a foot pedal which functions like the sustain pedal of a piano. In this case, one state change occurs on depression, a second on release. Here, the point to recognize is that the second action is a direct consequent of its predecessor. The syntax is implicit, and the cognitive burden of remembering what to do after the first action is minimal.

There are many cases where this type of kinesthetic connectivity can be bound to a sequence of tokens which are logically connected. One example given by Buxton [4] is in selecting an item from a graphics menu and "dragging" it into position in a work space. A button-down action (while pointing at an item) "picks it up." For as long as the button is depressed, the item tracks the motion of the pointing device. When the button is released, the item is anchored in its current position. Hence, the interface is designed to force the user to follow proper syntax: select then position. There is no possibility for syntactic error, and cognitive resources are not consumed in trying to remember "what do I do next?". Thus, by recognizing and exploiting such cases, interfaces can be constructed which are "natural" and easy to learn.

There is a similar type of chunking which can take place when two or more gestures are articulated at one time. Again we can take an example from driving a car, where in changing gears the actions on the clutch, accelerator and gear-shift reinforce one another and are coordinated into a single gesture. Choosing appropriate gestures for such coordinated actions can accelerate their bonding into what the user thinks of as a single act,

thereby freeing up cognitive resources to be applied to more important tasks. What we are arguing here is that by matching appropriate gestures with tasks, we can help render complex skills routine and gain benefits similar to those seen at different level in Card, Moran and Newell [6].

In summary, there are three main points which we wish to make with this example:

- there is an important interplay between the syntactic-lexical levels and the pragmatic level
- that this interplay can be exploited to reduce the cognitive burden of learning and using a system
- that this cannot be accomplished without a better understanding of pragmatic issues such as chunking and closure.

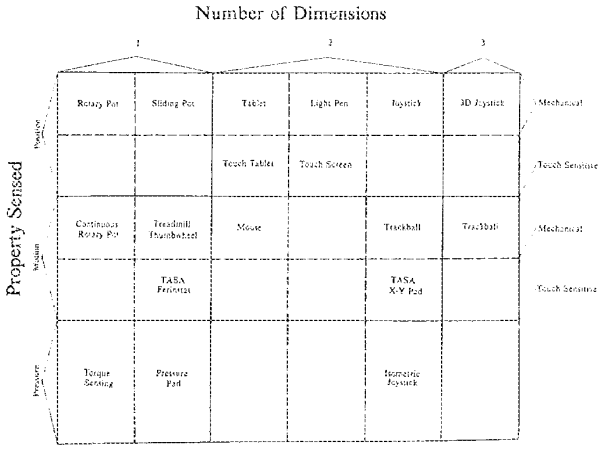
Pragmatics and Device Independence

We began by declaring the importance of being able to incorporate pragmatic issues into the models which we use to specify, design, compare and evaluate systems. The examples which followed then illustrated some of the reasons for this belief. When we view the CORE proposal [13, 14] from this perspective, however, we see several problems. The basis of how the CORE system approaches input is to deal with user actions in terms of abstractions, or logical devices (such as "locators" and "valuators"). The intention is to facilitate software portability. If all "locators," for example, utilized a common protocol, then user A (who only had a mouse) could easily implement software developed by B (who only had a tablet). From the application programmer's perspective, this is a valuable feature. However, for the purposes of specifying systems from the user's point of view, these abstractions are of very limited benefit. As Baecker [2] has pointed out, the effectiveness of a particular user interface is often due to the use of a *particular* device, and that effectiveness will be lost if that device were replaced by some other of the same logical class. For example, we have a system [10] whose interface depends on the simultaneous manipulation of four joysticks. Now in spite of tablets and joysticks both being "locator" devices, it is clear that they are not interchangeable in this situation. We cannot simultaneously manipulate four tablets. Thus, for the full potential of device independence to be realized, such pragmatic considerations must be incorporated into our overall specification model so that appropriate equivalencies can be determined in a methodological way. (That is, in specifying a generic device, we must also include the required pragmatic attributes. But to do so, we must develop a taxonomy of such attributes, just as we have developed a taxonomy of virtual devices.)

A Taxonomy of Devices

In view of the preceding discussion, we have attempted to develop a taxonomy which helps isolate relevant characteristics of input devices. The tableau shown in Figure 1 summarizes this effort in a two dimensional representation. The remainder of this section presents the details and motivation for this tableau's organization.

Figure 1. Tableau of Continuous Input Devices



To begin with, the tableau deals only with continuous hand-controlled devices. (Pedals, for example, are not included for simplicity's sake.) Therefore the first (but implicit) questions in our structure are:

- continuous *vs* discrete?
- agent of control (hand, foot, voice, ...)?

The table is divided into a matrix whose rows and columns delimit

- what is being sensed (position, motion or pressure), and
- the number of dimensions being sensed (1, 2 or 3),

respectively. These primary partitions of the matrix are delimited by solid lines. Hence, both the rotary and sliding potentiometer fall into the box associated with one-dimensional position-sensitive devices (top left-hand corner).

Note that the primary rows and columns of the matrix are sub-divided, as indicated by the dotted lines. The sub-columns exist to isolate devices whose control motion is roughly similar. These groupings can be seen in examining the two-dimensional devices. Here the tableau implies that tablets and mice utilize similar types of hand control and that this control is different from that shared in using a light-pen or touch-screen. Furthermore, it is shown that joysticks and trackballs share a common control motion which is, in turn, different than the other sub-classes of two-dimensional devices.

The rows for *position* and *motion* sensing devices are subdivided in order to differentiate between transducers which sense potential *via* mechanical *vs* touch-sensitive means. Thus, we see that the light-pen and touch-screen are closely related, except that the light-pen employs a mechanical transducer. Similarly, we see that trackball and TASA touch-pad[2] provide comparable signals from comparable gestures (the 4" by 4" dimensions of the TASA device compare to a 3 1/2" diameter trackball).

The tableau is useful for many purposes by virtue of the structure which it imposes on the domain of input devices. First, it helps in finding appropriate equivalences. This is important in terms of dealing with some of the problems which arose in our discussion of device independence. For example, we saw a case where four tablets would not be suitable for replacing four joysticks. By using the tableau, we see that four trackballs will probably do.

The tableau makes it easy to relate different devices in terms of metaphor. For example, a tablet is to a mouse what a joystick is to a trackball. Furthermore, if the taxonomy defined by the tableau can suggest new transducers in a manner analogous to the periodic table of Mendeleev predicting new elements, then we can have more confidence in its underlying premises. We make this claim for the tableau and cite the "torque sensing" one-dimensional pressure-sensitive transducer as an example. To our knowledge, no such device exists commercially. Nevertheless it is a potentially useful device, an approximation of which has been demonstrated by Herot and Weinzapfel [15].

Finally, the tableau is useful in helping quantify the generality of various physical devices. In cases where the work station is limited to one or two input devices, then it is often in the user's interest to choose the least constraining devices. For this reason, many people claim that tablets are the preferred device since they can emulate many of the other transducers (as is demonstrated by Evans, Tanner and Wein, [9]). The tableau is useful in determining the degree of this generality by "filling in" the squares which can be adequately covered by the tablet.

Before leaving the topic of the tableau, it is worth commenting on why a primary criterion for grouping devices was whether they were sensitive to position, motion or pressure. The reason is that what is sensed has a *very* strong effect on the nature of the dialogues that the system can support with any degree of fluency. As an example, let us compare how the user interface of an instrumentation console can be affected by the choice of whether motion or position sensitive transducers are used. For such consoles, one design philosophy follows the traditional model that for every function there should be a device. One of the rationales behind this approach is to avoid the use of "modes" which result when a single device must serve for more than one function. Another philosophy takes the point of view that the number of devices required in a console need only be in the order of the control bandwidth of the human operator. Here, the rationale is that careful design can minimize the "mode" problem, and that the resulting simple consoles are more cost-effective and less prone to breakdown (since they have fewer devices).

One consequence of the second philosophy is that the same transducer must be made to control different functions, or parameters, at different times. This context switching introduces something known as the *nulling problem*. The point which we are going to make is that this problem can be completely avoided if the transducer in question is motion rather than position sensitive. Let us see why.

Imagine that you have a sliding potentiometer which controls parameter A. Both the potentiometer and the parameter are at their minimum values. You then raise A to its maximum value by pushing up the position of the potentiometer's handle. You now want to change the value of parameter B. Before you can do so using the same potentiometer, the handle of the potentiometer must be repositioned to a position corresponding to the current value of parameter B. The necessity of having to perform this normalizing function is the nulling problem.

Contrast the difficulty of performing the above interaction using a position-sensitive device with the ease of doing so using one which senses motion. If a thumb-wheel or a treadmill-like device was used, the moment that the transducer is connected to the parameter it can be used to "push" the value up or "pull" it down. Furthermore, the same transducer can be used to simultaneously change the value of a group of parameters, all of whose instantaneous values are different.

Horizontal vs Vertical Strata

The above example brings up one important point: the different levels of the taxonomies of Foley and Van Dam or of Moran are not orthogonal. By describing the user interface in terms of a horizontal structure, it is very easy to fall into the trap of believing that the effect of modifications at one level will be isolated. This is clearly not true as the above example demonstrated: the choice of transducer type had a strong effect on syntax.

The example is not isolated. In fact, just as strong an argument could be made for adopting a model based on a vertical structure as the horizontal ones which we have discussed. Models based on interaction techniques such as those described in Martin [17] and Foley, Wallace and Chan [11] are examples. With them, the primary gestalt is the transaction, or interaction. The user model is described in terms of the set and style of

[2] The TASA X-Y 360 is a 4" by 4" touch sensitive device which gives 60 units of delta modulation in 4 inches of travel. The device is available from TASA, 2346 Walsh Ave., Santa Clara CA, 95051.

the interactions which take place over time. Syntactic, lexical and pragmatic questions become sub-issues.

Neither the horizontal or vertical view is "correct." The point is that *both* must be kept in mind during the design process. A major challenge is to adapt our models so that this is done in a well structured way. That we still have problems in doing so can be seen in Moran's taxonomy. Much of the difficulty in understanding the model is due to problems in his approach in integrating vertically oriented concepts (the interaction level) into an otherwise horizontal structure.

In spite of such difficulties, both views must be considered. This is an important cautionary bell to ring given the current trend towards delegating personal responsibilities according to horizontal stratification. The design of a system's data-base, for example, has a very strong effect on the semantics of the interactions that can be supported. If the computing environment is selected by one person, the data-base managed by another, the semantics or functional capability by another, and the "user interface" by yet another, there is an inherent danger that the decisions of one will adversely affect another. This is not to say that such an organizational structure cannot work. It is just imperative that we be aware of the pitfalls so that they can be avoided. Decisions made at all levels affect one another and *all* decisions potentially have an effect on the user model.

Summary and Conclusions

Two taxonomies for describing the problem domain of the user interface were described. In the discussion it was pointed out that the outer levels of the strata, those concerning lexical, spatial, and physical issues were neglected. The notion of pragmatics was introduced in order to facilitate focusing attention on these issues. Several examples were then examined which illustrated why this was important. In so doing, it was seen that the power of various existing models could be extended if we had a better understanding of pragmatic issues. As a step towards such an understanding, a taxonomy of hand controlled continuous input devices was introduced. It was seen that this taxonomy made some contribution towards addressing problems which arose in the case studies. It was also seen, however, that issues at this outer level of devices had a potentially strong effect on the other levels of the system. Hence, the danger of over-concentration on horizontal stratification was pointed out.

The work reported has made some contribution towards an understanding of the effect of issues which we have called pragmatics. It is, however, a very small step. While there is a great deal of work still to be done right at the device level, perhaps the biggest challenge is to develop a better understanding of the interplay among the different levels in the strata of a system. When we have developed a methodology which allows us to determine the gesture that best suits the expression of a particular concept, then we will be able to build the user interfaces which today are only a dream.

Acknowledgements

The ideas presented in this paper have developed over a period of time and owe much to discussions with our students and colleagues. In particular, a great debt is owed to Ron Baecker who was responsible for helping formulate many of the ideas presented. In addition, we would like to acknowledge the contribution of Alain Fournier, Russel Kirsch, Eugene Fiume and Ralph Hill in the intellectual development of the paper, and the help of Monica Delange in the preparation of the manuscript. Finally, we gratefully acknowledge the financial support of the National Sciences and Engineering Research Council of Canada.

References

1. Baecker, R. Human-computer interactive systems: a state-of-the-art review. In P. Kolers, E. Wrolfiad & H. Bouma, Eds., *Processing of Visible Language II*, New York: Plenum, (1980), 423–444.
2. Baecker, R. Towards an effective characterization of graphical interaction. In R. A. Guedj, P. Ten Hagen, F. Hopgood, H. Tucker & D. Duce, Eds., *Methodology of Interaction*, Amsterdam: North-Holland, (1980), 127–148.
3. Barnard, P., Hammond, N., Morton, J., and Long, J. Consistency and compatibility in human-computer dialogue. *International Journal of Man-Machine Studies 15*, (1981), 87–134.
4. Buxton, W. An informal study of selection-positioning tasks. *Proceedings of Graphics Interface '82*, Toronto, (1982), 323–328.
5. Card, S., English, W., and Burr, B. Evaluation of mouse, rate-controlled isometric joystick, step keys, and text keys for text selection on a CRT. *Ergonomics 8*, (1978), 601–613.
6. Card, S., Moran, T., and Newell, A. Computer text editing: an information-processing analysis of a routine cognitive skill. *Cognitive Psychology 12*, (1980),32–74.
7. Card, S., Moran, T., and Newell, A. The keystroke-level model for user performance time with interactive systems. *Communications of the ACM 23*, 7 (1980), 396–410.
8. Embley, D., Lan, M., Leinbaugh, D., and Nagy, G. A procedure for predicting program editor performance from the user's point of view. *International Journal of Man-Machine Studies 10*, (1978), 639–650.

9. Evans, K., Tanner, P., and Wein, M. Tablet-based valuators that provide one, two, or three degrees of freedom. *Computer Graphics 15*, 3 (1981), 91–97.

10. Fedorkow, G., Buxton, W., and Smith, K. C. A computer controlled sound distribution system for the performance of electroacoustic music. *Computer Music Journal 2*, 3 (1978), 33–42.

11. Foley, J., Wallace, V., and Chan, P. The human factors of interaction techniques. Technical Report GWU-IIST-81-03, Washington: The George Washington University, Institute for Information Science and Technology, (1981).

12. Foley, J. and Van Dam, A. *Fundamentals of Interactive Computer Graphics.* Reading, MA: Addison Wesley, (1982).

13. GSPC. Status Report of the Graphics Standards Planning Committee. *Computer Graphics 11*, (1977).

14. GSPC. Status Report of the Graphics Standards Committee. *Computer Graphics 13*, 3 (1979).

15. Herot, C. and Weinzapfel, G. One-point touch input of vector information for computer displays. *Computer Graphics 12*, 3 (1978), 210–216.

16. Ledgard, H., Whiteside, J., Singer, A., and Seymour, W. The natural language of interactive systems. *Communications of the ACM 23*, 10 (1980), 556–563.

17. Martin, J. *Design of Man-Computer Dialogues.* Engelwood Cliffs, NJ: Prentice-Hall, (1973).

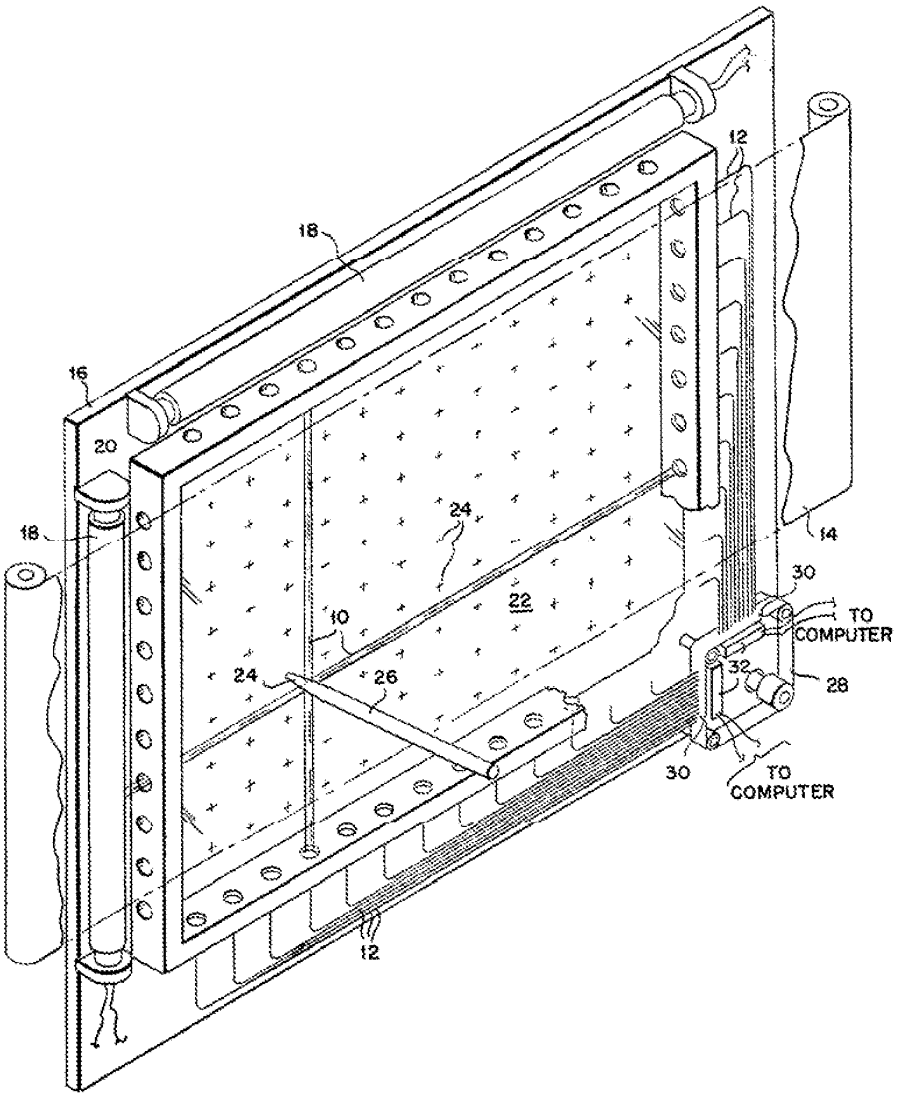
18. Moran, T. The command language grammar: a representation for the user interface of interactive computer systems. *International Journal of Man-Machine Studies 15*, (1981), 3–50.

19. Reisner, P. Use of psychological experimentation as an aid to development of a query language. *IEEE Transactions on Software Engineering 3*, 3 (1977), 218–229.

20. Reisner, P. Formal grammar and human factors design of an interactive graphics system. *IEEE Transactions on Software Engineering 7*, 2 (1981), 229–240.

Light Beam Matrix Input Terminal

This display and computer input device consists of a rectangular matrix of light beams 10 and associated photosensitive devices 12 overlaying document 14. Mount 16 contains a pair of light sources 18 at right angles to each other. Beams 10 are formed by holes in frame 20 and image on optical fibers 12 opposite sources 18. Thus, a light beam matrix is formed. The frame assembly is spaced slightly above document 14 by thin, clear screen 22 having response holes 24 at each intersection of beams 10. When probe 26 or the finger is placed in a hole of screen 22, intersecting beams are interrupted. Fibers 12 are merged to moving belt 28 having light detectors 32 at its underside. Fibers 12 are so arranged that slots 30 scan them serially. Document 14 can be one of a plurality on a roll.


12
18
16
20
18
24
14
30
22
TO
COMPUTER
10
24
26
32
28
30
TO
COMPUTER
12

Multi-Touch Systems that I Have Known and Loved

Bill Buxton
Microsoft Research
Original: Jan. 12, 2007
Version: March 21st, 2011

Keywords / Search Terms

Multi-touch, multitouch, input, interaction, touch screen, touch tablet, multi-finger input, multi-hand input, bi-manual input, two-handed input, multi-person input, interactive surfaces, soft machine, hand gesture, gesture recognition .

 This page is also available in Belorussian, thanks to the translation by Martha Ruszkowski.

Preamble

Since the announcements of the *iPhone* and Microsoft*'s Surface* (both in 2007), an especially large number of people have asked me about multi-touch. The reason is largely because they know that I have been involved in the topic for a number of years. The problem is, I can't take the time to give a detailed reply to each question. So I have done the next best thing (I hope). That is, start compiling my would-be answer in this document. The assumption is that ultimately it is less work to give one reasonable answer than many unsatisfactory ones.

Multi-touch technologies have a long history. To put it in perspective, my group at the University of Toronto was working on multi-touchin 1984 (Lee, Buxton & Smith, 1985), the same year that the first Macintosh computer was released, and we were not the first. Furthermore, during the development of the iPhone, Apple was very much aware of the history of multi-touch, dating at least back to 1982, and the use of the pinch gesture, dating back to 1983. This is clearly demonstrated by the bibliography of the PhD thesis of Wayne Westerman, co-founder of FingerWorks, a company that Apple acquired early in 2005, and now an Apple employee

> Westerman, Wayne (1999). *Hand Tracking,Finger Identification, and Chordic Manipulation on a Multi-Touch Surface.* U of Delaware PhD Dissertation: http://www.ee.udel.edu/~westerma/main.pdf

In making this statement about their awareness of past work, I am not criticizing Westerman, the iPhone, or Apple. It is simply good practice and good scholarship to know the literature and do one's homework when embarking on a new product. What I *am* pointing out, however, is that "new" technologies - like multi-touch - do not grow out of a vacuum. While marketing tends to like the "great invention" story, real innovation rarely works that way. In short, the evolution of multi-touch is a text-book example of what I call "the long-nose of innovation."

So, to shed some light on the back story of this particular technology, I offer this brief and incomplete summary of some of the landmark examples that I have been involved with, known about and/or encountered over the years. As I said, it is incomplete and a work in progress (so if you come back a second time, chances are there will be more and better information). I apologize to those that I have missed. I have erred on the side of timeliness *vs* thoroughness. Other work can be found in the references to the papers that I do include.

Note: for those note used to searching the HCI literature, the primary portal where you can search for and download the relevant literature, including a great deal relating to this topic (including the citations in the Westerman thesis), is the ACM Digital Library: http://portal.acm.org/dl.cfm. One other relevant source of interest, should you be interested in an example of the kind of work that has been done studying gestures in interaction, see the thesis by Hummels:

> http://id-dock.com/pages/overig/caro/publ_caro.htm

While not the only source on the topic by any means, it is a good example to help gauge what might be considered new or obvious.

Please do not be shy in terms of sending me photos, updates, etc. I will do my best to integrate them.

For more background on input, see also the incomplete draft manuscript for my book on input tools, theories and techniques:

> http://www.billbuxton.com/inputManuscript.html

For more background on input devices, including touch screens and tablets, see my directory at:

- http://www.billbuxton.com/inputSources.html

I hope this helps.

Some Dogma

There is a lot of confusion around touch technologies, and despite a 25 year history, very little information or experience with multi-touch interaction. I have three comments to set up what is to follow:

1. Remember that it took 30 years between when the mouse was invented by Engelbart and English in 1965 to when it became ubiquitous, on the release of Windows 95. Yes, it was released commercially on the Xerox Star and PERQ workstations in 1982, and I used my first one in 1972 at the National Research Council of Canada. But statistically, that doesn't matter. It took 30 years to hit the tipping point. So, by that measure, multi-touch technologies have 5 years to go before they fall behind.
2. Keep in mind one of my primary axioms: *Everything is best for something and worst for something else.* The trick is knowing what is what, for what, when, for whom, where, and most importantly, why. Those who try the replace the mouse play a fool's game. The mouse is great for many things. Just not everything. The challenge with new input is to find devices that work together, simultaneously with the mouse (such as in the other hand), or things that are strong where the mouse is weak, thereby complimenting it.
3. To significantly improve a product by a given amount, it probably takes about two more orders of magnitude of cost, time and effort to improve the display as to get the same amount of improvement on input. Why? Because we are ocular centric, and displays are therefore much more mature. Input is still primitive, and wide open for improvement. So it is a good thing that you are looking at this stuff. What took you so long?

Some Framing

I don't have time to write a treatise, tutorial or history. What I can do is warn you about a few traps that seem to cloud a lot of thinking and discussion around this stuff. The approach that I will take is to draw some distinctions that I see as meaningful and relevant. These are largely in the form of contrasts:

- **Touch-tablets *vs* Touch screens:** In some ways these are two extremes of a continuum. If, for example, you have paper graphics on your tablet, is that a display (albeit more-or-less static) or not? What if the "display" on the touch tablet is a tactile display rather than visual? There are similarities, but there are real differences between touch-sensitive display surfaces, *vs* touch pads or tablets. It is a difference of *directness.* If you touch exactly where the thing you are interacting with is, let's call it a touch screen or touch display. If your hand is touching a surface that is not overlaid on the screen, let's call it a touch tablet or touch pad.

- **Discrete *vs* Continuous:** The nature of interaction with multi-touch input is highly dependent on the nature of discrete *vs* continuous actions supported. Many conventional touch-screen interfaces are based discrete items such as pushing so-called "light buttons", for example. An example of a multi-touch interface using such discrete actions would be using a soft graphical QWERTY keyboard, where one finger holds the shift key and another pushes the key for the upper-case character that one wants to enter. An example of two fingers doing a coordinated continuous action would be where they are stretching the diagonally opposed corners of a rectangle, for example. Between the two is a continuous/discrete situation, such as where one emulates a mouse, for example, using one finger for indicating continuous position, and other fingers, when in contact, indicate mouse button pushes, for example.

- **Degrees of Freedom:** The richness of interaction is highly related to the richness/numbers of degrees of freedom (DOF), and in particular, continuous degrees of freedom, supported by the technology. The conventional GUI is largely based on moving around a single 2D cursor, using a mouse, for example. This

results in 2DOF. If I am sensing the location of two fingers, I have 4DOF, and so on. When used appropriately, these technologies offer the potential to begin to capture the type of richness of input that we encounter in the everyday world, and do so in a manner that exploits the everyday skills that we have acquired living in it. This point is tightly related to the previous one.

- **Size matters:** Size largely determines what muscle groups are used, how many fingers/hands can be active on the surface, and what types of gestures are suited for the device.

- **Orientation Matters - Horizontal *vs* Vertical:** Large touch surfaces have traditionally had problems because they could only sense one point of contact. So, if you rest your hand on the surface, as well as the finger that you want to point with, you confuse the poor thing. This tends not to occur with vertically mounted surfaces. Hence large electronic whiteboards frequently use single touch sensing technologies without a problem.

- **There is more to touch-sensing than contact and position:** Historically, most touch sensitive devices only report that the surface has been touched, and where. This is true for both single and multi touch devices. However, there are other aspects of touch that have been exploited in some systems, and have the potential to enrich the user experience:

    1. **Degree of touch / pressure sensitivity:** A touch surfaces that that can independently and continuously sense the degree of contact for each toouch point has a far higher potential for rich interaction. Note that I use "degree of contact" rather than pressure since frequently/usually, what passes for pressure is actually a side effect – as you push harder, your finger tip spreads wider over the point of contact, and what is actually sensed is amount/area of contact, not pressure, *per se*. Either is richer than just binary touch/no touch, but there are even subtle differences in the affordances of pressure *vs* degree.

    2. **Angle of approach:** A few systems have demonstrated the ability to sense the angle that the finger relative to the screen surface. See, for example, McAvinney's *Sensor Frame*, below. In effect, this lgives the finger the capability to function more-or-less as a virtual joystick at the point of contact, for example. It also lets the finger specify a vector that can be projected into the virtual 3D space behind the screen from the point of contact - something that could be relevant in games or 3D applications.

    3. **Force vectors:** Unlike a mouse, once in contact with the screen, the user can exploit the friction between the finger and the screen in order to apply various force vectors. For example, without moving the finger, one can apply a force along any vector parallel to the screen surface, including a rotational one. These techniques were described as early as 1978, as shown below, by Herot, C. & Weinzapfel, G. (1978). Manipulating Simulated Objects with Real-World Gestures Using a Force and Position Sensitive Screen, *Computer Graphics*, 18(3), 195-203.].

    Such historical examples are important reminders that it is human capability, not technology, that should be front and centre in our considerations. While making such capabilities accessible at reasonable costs may be a challenge, it is worth remembering further that the same thing was also said about multi-touch. Furthermore, note that multi-touch dates from about the same time as these other touch innovations.

- **Size matters II:** The ability of to sense the size of the area being touched can be as important as the size of the touch surface. See the Synaptics example, below, where the device can sense the difference between the touch of a finger (small) *vs* that of the cheek (large area), so that, for example, you can answer the phone by

holding it to the cheek.

- **Single-finger *vs* multi-finger:** Although multi-touch has been known since at least 1982, the vast majority of touch surfaces deployed are single touch. If you can only manipulate one point, regardless of with a mouse, touch screen, joystick, trackball, etc., you are restricted to the gestural vocabulary of a fruit fly. We were given multiple limbs for a reason. It is nice to be able to take advantage of them.

- **Multi-point *vs* multi-touch:** It is really important in thinking about the kinds of gestures and interactive techniques used if it is peculiar to the technology or not. Many, if not most, of the so-called "multi-touch" techniques that I have seen, are actually "multi-point". Think of it this way: you don't think of yourself of using a different technique in operating your laptop just because you are using the track pad on your laptop (a single-touch device) instead of your mouse. Double clicking, dragging, or working pull-down menus, for example, are the same interaction technique, independent of whether a touch pad, trackball, mouse, joystick or touch screen are used.

- **Multi-hand *vs* multi-finger:** For much of this space, the control can not only come from different fingers or different devices, but different hands working on the same or different devices. A lot of this depends on the scale of the input device. Here is my analogy to explain this, again referring back to the traditional GUI. I can point at an icon with my mouse, click down, drag it, then release the button to drop it. Or, I can point with my mouse, and use a foot pedal to do the clicking. It is the same dragging technique, even though it is split over two limbs and two devices. So a lot of the history here comes from a tradition that goes far beyond just multi-touch.

- **Multi-person *vs* multi-touch:** If two points are being sensed, for example, it makes a huge difference if they are two fingers of the same hand from one user *vs* one finger from the right hand of each of two different users. With most multi-touch techniques, you do *not* want two cursors, for example (despite that being one of the first thing people seem to do). But with two people working on the same surface, this may be exactly what you *do* want. And, insofar as multi-touch technologies are concerned, it may be valuable to be able to sense which person that touch comes from, such as can be done by the *Diamond Touch* system from MERL (see below).

- **Points *vs* Gesture:** Much of the early relevant work, such as Krueger (see below) has to do with sensing the pose (and its dynamics) of the hand, for example, as well as position. That means it goes way beyond the task of sensing multiple points.

- **Stylus and/or finger:** Some people speak as if one must make a choice between stylus *vs* finger. It certainly is the case that many stylus systems will not work with a finger, but many touch sensors work with a stylus or finger. It need not be an either or question (although that might be the correct decision – it depends on the context and design). But any user of the Palm Pilot knows that there is the potential to use either. Each has its own strengths and weaknesses. Just keep this in mind: if the finger was the ultimate device, why didn't Picasso and Rembrandt restrict themselves to finger painting? On the other hand, if you want to sense the temperature of water, your finger is a better tool than your pencil.

- **Hands and fingers *vs* Objects:** The stylus is just one object that might be used in multi-point interaction. Some multi-point / multi-touch systems can not only sense various different objects on them, but what object it is, where it is, and what its orientation is. See Andy Wilson's work, below, for example. And, the objects,

stylus or otherwise, may or may not be used in conjunction and simultaneously with fingers.

- **Different *vs* The Same:** When is something the same, different or obvious? In one way, the answer depends on if you are a user, programmer, scientist or lawyer. From the perspective of the user interface literature, I can make three points that would be known and assumed by anyone skilled in the art:

    1. *Device-Independent Graphics:* This states that the same technique implemented with an alternative input device is still the same technique. For example, you can work your GUI with a stylus, touch screen, mouse, joystick, touchpad, or trackball, and one would still consider techniques such as double-clicking, dragging, dialogue boxes as being "the same" technique;

    2. *The Interchange of devices is not neutral from the perspective of the user:* While the skill of using a GUI with a mouse transfers to using a touchpad, and the user will consider the interface as using the same techniques, nevertheless, the various devices have their own idiomatic strengths and weaknesses. So, while the user will consider the techniques the "same", their performance (speed, accuracy, comfort, preference, etc.) will be different from device to device. Hence, the interactive experience is not the same from device to device, despite using the same techniques. Consequently, it is the norm for users and researchers alike to swap one device for another to control a particular technique.

Some Attributes

As I stated above, my general rule is that everything is best for something and worst for something else. The more diverse the population is, the places and contexts where they interact, and the nature of the information that they are passing back in forth in those interactions, the more there is room for technologies tailored to the idiosyncrasies of those tasks.

The potential problem with this, is that it can lead to us having to carry around a collection of devices, each with a distinct purpose, and consequently, a distinct style of interaction. This has the potential of getting out of hand and our becoming overwhelmed by a proliferation of gadgets – gadgets that are on their own are simple and effective, but collectively do little to reduce the complexity of functioning in the world. Yet, traditionally our better tools have followed this approach. Just think of the different knives in your kitchen, or screwdrivers in your workshop. Yes there are a great number of them, but they are the "right ones", leading to an interesting variation on an old theme, namely, "more is less", i.e., more (of the right) technology results is less (not more) complexity. But there are no guarantees here.

What touch screen based "soft machines" offer is the opposite alternative, "less is more". Less, but more generally applicable technology results in less overall complexity. Hence, there is the prospect of the multi-touch soft machine becoming a kind of chameleon that provides a single device that can transform itself into whatever interface that is appropriate for the specific task at hand. The risk here is a kind of "jack of all trades, master of nothing" compromise.

One path offered by touch-screen driven appliances is this: instead of making a device with different buttons and dials mounted on it, soft machines just draw a picture of the devices, and let you interact with them. So, ideally, you get far more flexibility out of a single device. Sometimes, this can be really good. It can be especially good if, like physical devices, you can touch or operate more than one button, or virtual device at a time. For an example of where using more than one button or device at a time is important in the physical world, just think of having to type

without being able to push the SHIFT key at the same time as the character that you want to appear in upper case. There are a number of cases where this can be of use in touch interfaces.

Likewise, multi-touch greatly expands the types of gestures that we can use in interaction. We can go beyond simple pointing, button pushing and dragging that has dominated our interaction with computers in the past. The best way that I can relate this to the everyday world is to have you imagine eating Chinese food with only one chopstick, trying to pinch someone with only one fingertip, or giving someone a hug with – again – the tip of one finger or a mouse. In terms of pointing devices like mice and joysticks are concerned, we do everything by manipulating just one point around the screen – something that gives us the gestural vocabulary of a fruit fly. One suspects that we can not only do better, but as users, deserve better. Multi-touch is one approach to accomplishing this – but by no means the only one, or even the best. (How can it be, when I keep saying, everything is best for something, but worst for something else).

There is no Free Lunch.

- **Feelings**: The adaptability of touch screens in general, and multi-touch screens especially comes at a price. Besides the potential accumulation of complexity in a single device, the main source of the downside stems from the fact that you are interacting with a picture of the ideal device, rather than the ideal device itself. While this may still enable certain skills from the specialized physical device transfer to operating the virtual one, it is simply not the same. Anyone who has typed on a graphical QWERTY keyboard knows this.

  User interfaces are about look *and* feel. The following is a graphic illustration of how this generally should be written when discussing most touch-screen based systems:

  # Look and Feel

  Kind of ironic, given that they are "touch" screens. So let's look at some of the consequences in our next points.

- **If you are blind you are simply out of luck. p.s., we are all blind at times** - such as when lights are out, or our eyes are occupied elsewhere – such as on the road). On their own, soft touch screen interfaces are nearly all "eyes on". You cannot "touch type", so to speak, while your eyes are occupied elsewhere (one exception is so-called "heads-up" touch entry using single stroke gestures such as Graffiti that are location independent). With an all touch-screen interface you generally cannot start, stop, or pause your MP3 player, for example, by reaching into your pocket/purse/briefcase. Likewise, unless you augment the touch screen with speech recognition for all functions, you risk a serious accident trying to operate it while driving. On the other hand, MP3 players and mobile phones mechanical keys can to a certain degree be operated eyes free – the extreme case being some 12-17 year old kids who can text without looking!

- **Handhelds that rely on touch screens for input virtually all require two hands to operate:** one to hold the device and the other to operate it. Thus, operating them generally requires both eyes *and* both hands.
- **Your finger is not transparent:** The smaller the touch screen the more the finger(s) obscure what is being pointed at. Fingers do not shrink in the same way that chips and displays do. That is one reason a stylus is sometimes of value: it is a proxy for the finger that is very skinny, and therefore does not obscure the screen.

- **There is a reason we don't rely on finger painting:** Even on large surfaces, writing or drawing with the finger is generally not as effective as it is with a brush or stylus. On small format devices it is virtually useless to try and take notes or make drawings using a finger rather than a stylus. If one supports good digital ink and an appropriate stylus and design, one can take notes about as fluently as one can with paper. Note taking/scribble functions are notably absent from virtually all finger-only touch devices.
- **Sunshine:** We have all suffered trying to read the colour LCD display on our MP3 player, mobile phone and digital camera when we are outside in the sun. At least with these devices, there are mechanical controls for some functions. For example, even if you can't see what is on the screen, you can still point the camera in the appropriate direction and push the shutter button. With interfaces that rely exclusively on touch screens, this is not the case. Unless the device has an outstanding reflective display, the device risks being unusable in bright sunlight.

Does this property make touch-devices a bad thing? No, not at all. It just means that they are distinct devices with their own set of strengths and weaknesses. The ability to completely reconfigure the interface on the fly (so-called "soft interfaces") has been long known, respected and exploited. But there is no free lunch and no general panacea. As I have said, everything is best for something and worst for something else. Understanding and weighing the relative implications on use of such properties is necessary in order to make an informed decision. The problem is that most people, especially consumers (but including too many designers) do not have enough experience to understand many of these issues. This is an area where we could all use some additional work. Hopefully some of what I have written here will help.

An Incomplete Roughly Annotated Chronology of Multi-Touch and Related Work

In the beginning .... Typing & N-Key Rollover (IBM and others).

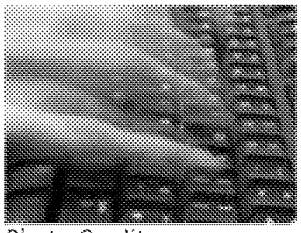
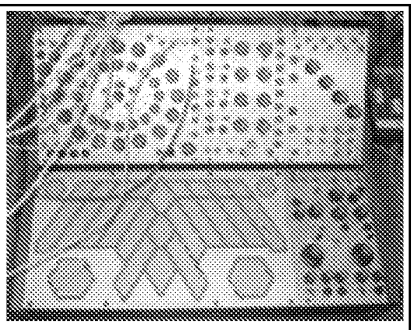


Photo Credit

- While it may seem a long way from multi-touch screens, the story of multi-touch starts with keyboards.
- Yes they are mechanical devices, "hard" rather than "soft" machines. But they do involve multi-touch of a sort.
- First, most obviously, we see sequences, such as the SHIFT, Control, Fn or ALT keys in combination with others. These are cases where we *want* multi-touch.
- Second, there are the cases of unintentional, but inevitable, multiple simultaneous key presses which we want to make proper sense of, the so-called question of n-key rollover (where you push the next key before releasing the previous one).

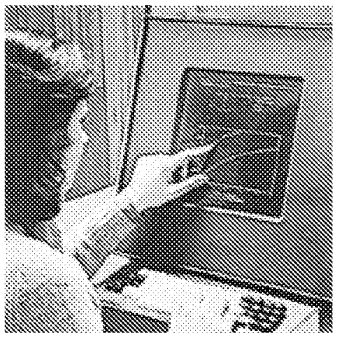
Electroacoustic Music: The Early Days of Electronic Touch Sensors (Hugh LeCaine , Don Buchla & Bob Moog).
http://www.hughlecaine.com/en/instruments.html.



- The history of touch-sensitive control devices pre-dates the age of the PC
- A number of early synthesizer and electronic music instrument makers used touch-sensitive capacitance-sensors to control the sound and music being made.
- These were touch pads, rather than touch screens
- The tradition of innovating on touch controls for musical purposes continued/continues, and was the original basis for the University of Toronto multitouch surface, as well as the CMU Sensor Frame.

1972: PLATO IV Touch Screen Terminal (Computer-based Education Research Laboratory, University of Illinois, Urbana-Champain)
http://en.wikipedia.org/wiki/Plato_computer

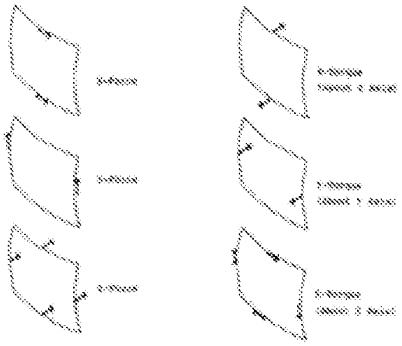
- Touch screens started to be developed in the second half of the 1960s.
- Early work was done at the IBM, the University of Illinois, and Ottawa Canada.



By 1971 a number of different techniques had been disclosed

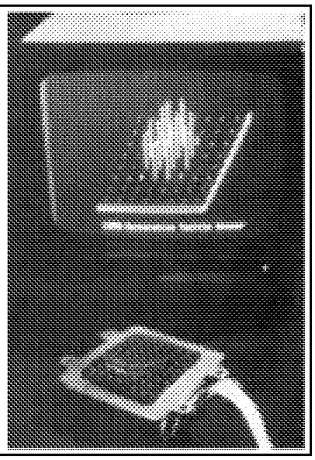
- All were single-touch and none were pressure-sensitive
- One of the first to be generally known was the terminal for the PLATO IV computer assisted education system, deployed in 1972.
- As well as its use of touch, it was remarkable for its use of real-time random-access audio playback, and the invention of the flat panel plasma display.
- the touch technology used was a precursor to the infrared technology still available today from CarrollTouch.
- The initial implementation had a 16 x 16 array of touch-sensitive locations

1978: One-Point Touch Input of Vector Information (Chris Herot & Guy Weinzapfel, Architecture Machine Group, MIT).



- The screen demonstrated by Herot & Weinzapfel could sense 8 different signals from a single touch point: position in X & Y, force in X, Y, & Z (i.e., sheer in X & Y & Pressure in Z), and torque in X, Y & Z.
- While we celebrate how clever we are to have multi-touch sensors, it is nice to have this reminder that there are many other dimensions of touch screens that can be exploited in order to provide rich interaction
- See: Herot, C. & Weinzapfel, G. (1978). One-Point Touch Input of Vector Information from Computer Displays, *Computer Graphics*, 12(3), 210-216.
- For a video demo, see: http://www.youtube.com/watch?v=vMkYfd0sOLM
- For similar work, see also: Minsky, M. (1984). Manipulating Simulated Objects with Real-World Gestures Using a Force and Position Sensitive Screen, *Computer Graphics*, 18(3), 195-203.

1981: Tactile Array Sensor for Robotics (Jack Rebman, Lord Corporation).



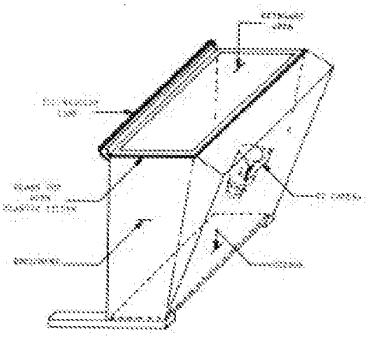
- A multi-touch sensor designed for robotics to enable sensing of shape, orientation, etc.
- Consisted of an 8 x 8 array of sensors in a 4" x 4" square pad
- Usage described in: Wolfeld, Jeffrey A. (1981). *Real Time Control of a Robot Tactile Sensor*. MSc Thesis. Philadelphia: Moore School of Electrical Engineering.
- The figure to the right shows a computer display of the tactile impression of placing a round object on the tactile sensor, shown in the foreground. Groover, M.P., Weiss, M., Nagel, R.N. & Odrey, N. (1986). *Industrial Robots.* New York: McGraw-Hill, p.152.)
- A US patent (4,521,685) was issued for this work to Rebman in 1985.

1982: Flexible Machine Interface (Nimish Mehta , University of Toronto).

- The first multi-touch system that I am aware of designed for human input to a computer system.
- Consisted of a frosted-glass panel whose local optical properties were such that

when viewed behind with a camera a black spot whose size depended on finger pressure appeared on an otherwise white background. This with simple image processing allowed multi touch input picture drawing, etc. At the time we discussed the notion of a projector for defining the context both for the camera and the human viewer.

- Mehta, Nimish (1982), *A Flexible Machine Interface,* M.A.Sc. Thesis, Department of Electrical Engineering, University of Toronto supervised by Professor K.C. Smith.

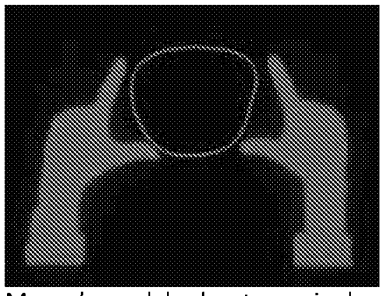


1983: Soft Machines (Bell Labs, Murray Hill)

- This is the first paper that I am aware of in the user interface literature that attempts to provide a comprehensive discussion the properties of touch-screen based user interfaces, what they call "soft machines".
- While not about multi-touch specifically, this paper outlined many of the attributes that make this class of system attractive for certain contexts and applications.
- Nakatani, L. H. & Rohrlich, John A. (1983). Soft Machines: A Philosophy of User-Computer Interface Design. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'83)*, 12-15.

1983: Video Place / Video Desk (Myron Krueger)

- A vision based system that tracked the hands and enabled multiple fingers, hands, and people to interact using a rich set of gestures.
- Implemented in a number of configurations, including table and wall.
- Didn't sense touch, per se, so largely relied on dwell time to trigger events intended by the pose.
- On the other hand, in the horizontal desktop configuration, it inherently *was* touch based, from the user's perspective.
- Essentially "wrote the book" in terms of unencumbered (i.e., no gloves, mice, styli, etc.) rich gestural interaction.
- Work that was more than a decade ahead of its time and hugely influential, yet not as acknowledged as it should be.
- His use of many of the hand gestures that are now starting to emerge can be clearly seen in the following 1988 video, including using the pinch gesture to scale and translate objects: http://youtube.com/watch?v=dmmxVA5xhuo
- There are many other videos that demonstrate this system. Anyone in the field should view them, as well as read his books:
- Krueger, Myron, W. (1983). *Artificial Reality*. Reading, MA:Addison-Wesley.
- Krueger, Myron, W. (1991). *Artificial Reality II*. Reading, MA: Addison-Wesley.
- Krueger, Myron, W., Gionfriddo, Thomas., &Hinrichsen, Katrin (1985). VIDEOPLACE - An Artificial Reality, *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI'85), 35 - 40.



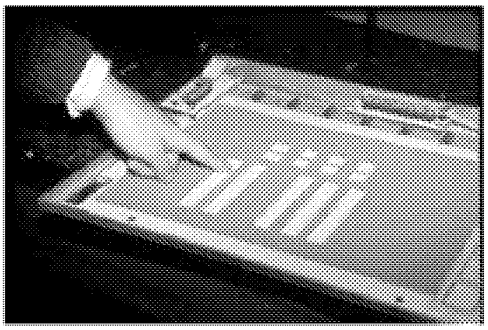
Myron's work had a staggeringly rich repertoire of gestures, muti-finger, multi-hand and multi-person interaction.

1984: Multi-Touch Screen (Bob Boie, Bell Labs, Murray Hill NJ)

- A multi-touch touch *screen*, not tablet.
- The first muti-touch screen that I am aware of.
- Used a transparent capacitive array of touch sensors overlaid on a CRT. Could manipulate graphical objects with fingers with excellent response time

- Developed by Bob Boie, but was shown to me by Lloyd Nakatani (see above), who invited me to visit Bell Labs to see it after he saw the presentation of our work at SIGCHI in 1985
- Since Boie's technology was transparent and faster than ours, when I saw it, my view was that they were ahead of us, so we stopped working on hardware (expecting that we would get access to theirs), and focus on the software and the interaction side, which was our strength. Our assumption (false, as it turned out) was that the Boie technology would become available to us in the near future.
- Around 1990 I took a group from Xerox to see this technology it since I felt that it would be appropriate for the user interface of our large document processors. This did not work out.
- There was other multi-touch work at Bell Labs around the time of Boie's. See the 1984 work by Leonard Kasday, (US Patent 4484179), which used optical techniques

1985: Multi-Touch Tablet (Input Research Group, University of Toronto):
http://www.billbuxton.com/papers.html#anchor1439918



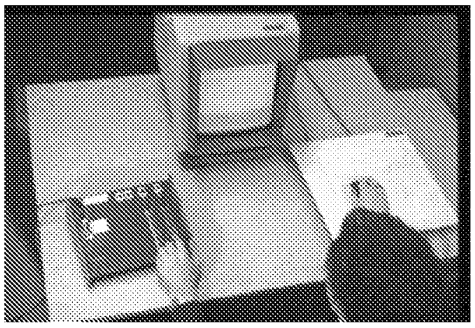
- Developed a touch tablet capable of sensing an arbitrary number of simultaneous touch inputs, reporting both location and degree of touch for each.
- To put things in historical perspective, this work was done in 1984, the same year the first Macintosh computer was introduced.
- Used capacitance, rather than optical sensing so was thinner and much simpler than camera-based systems.
- A Multi-Touch Three Dimensional Touch-Sensitive Tablet (1985). Video at: http://www.billbuxton.com/buxtonIRGVideos.html
- Issues and techniques in touch-sensitive tablet input.(1985). Video at: http://www.billbuxton.com/buxtonIRGVideos.html

1985: Sensor Frame (Carnegie Mellon University)

- This is work done by Paul McAvinney at Carengie-Mellon University
- The device used optical sensors in the corners of the frame to detect fingers.
- At the time that this was done, miniature cameras were essentially unavailable. Hence, the device used DRAM IC's with glass (as opposed to opaque) covers for imaging.
- It could sense up to three fingers at a time fairly reliably (but due to optical technique used, there was potential for misreadings due to shadows.
- In a later prototype variation built with NASA funding, the Sensor Cube, the device could also could detect the angle that the finger came in to the screen.
    - McAvinney, P. (1986). *The Sensor Frame - A Gesture-Based Device for the Manipulation of Graphic Objects*. Carnegie-Mellon University.
    - McAvinney, P. (1990). Telltale Gstures: 3D applications need 3D input. Byte Magazine, 15(7), 237-240.
    - http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19940003261_1994003261.pdf

1986: Bi-Manual Input (University of Toronto)

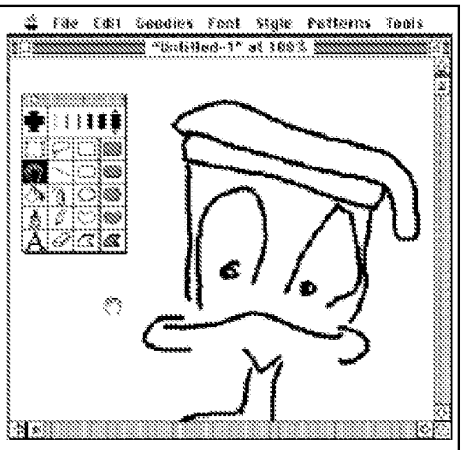
- In 1985 we did a study, published the following year, which examined the benefits of two different compound bi-manual tasks that involved continuous control with each hand
- The first was a positioning/scaling task. That is, one had to move a shape to a particular location on the screen with one hand, while adjusting its size to match a particular target with the other.
-



The second was a selection/navigation task. That is, one had to navigate to a particular location in a document that was currently off-screen, with one hand, then select it with the other.

- Since bi-manual continuous control was still not easy to do (the ADB had not yet been released - see below), we emulated the Macintosh with another computer, a PERQ.
- The results demonstrated that such continuous bi-manual control was both easy for users, and resulted in significant improvements in performance and learning.
- See Buxton, W. & Myers, B. (1986). A study in two-handed input. *Proceedings of CHI '86,* 321-326.[video]
- Despite this capability being technologically and economically viable since 1986 (with the advent of the ADB - see below - and later USB), there are still no mainstream systems that take advantage of this basic capability. Too bad.
- This is an example of techniques developed for multi-device and multi-hand that can easily transfer to multi-touch devices.

1986: Apple Desktop Bus (ADB) and the Trackball Scroller Init (Apple Computer / University of Toronto)



- The Macintosh II and Macintosh SE were released with the Apple Desktop Bus. This can be thought of as an early version of the USB.
- It supported plug-and-play, and also enabled multiple input devices (keyboards, trackballs, joysticks, mice, etc.) to be plugged into the same computer simultaneously.
- The only downside was that if you plugged in two pointing devices, by default, the software did not distinguish them. They both did the same thing, and if a mouse and a trackball were operate at the same time (which they could be) a kind of tug-of-war resulted for the tracking symbol on the screen.
- My group at the University of Toronto wanted to take advantage of this multi-device capability and contacted friends at Apple's Advanced Technology Group for help.
- Due to the efforts of Gina Venolia and Michael Chen , they produced a simple "init" that could be dropped into the systems folder called the *trackballscroller init.*
- It enabled the mouse, for example, to be designated the pointing device, and a trackball, for example, to control scrolling independently in X and Y. See, for example, Buxton, W. (1990). The Natural Language of Interaction: A Perspective on Non-Verbal Dialogues. In Laurel, B. (Ed.). *The Art of Human-Computer Interface Design,* Reading, MA: Addison-Wesley. 405-416.
- They also provided another init that enabled us to grab the signals from the second device and use it to control a range of other functions. See fr example, Kabbash, P., Buxton, W.& Sellen, A. (1994). Two-Handed Input in a

Compound Task. *Proceedings of CHI '94*, 417-423.

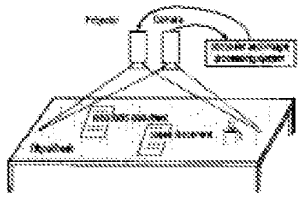
- In short, with this technology, we were able to deliver the benefits demonstrated by Buxton & Myers (see above) on standard hardware, without changes to the operating system, and largely, with out changes even to the applications.
- This is the closest that we came, without actually getting there, of supporting multi-point input - such as all of the two-point stretching, etc. that is getting so much attention now, 20 years later. It was technologically and economically viable then.
- To our disappointment, Apple never took advantage of this - one of their most interesting - innovations.

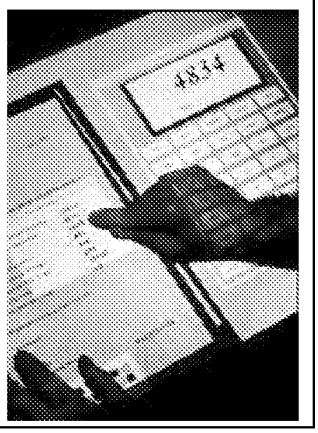
1991: Bidirectional Displays (Bill Buxton & Colleagues , Xerox PARC)

- First discussions about the feasibility of making an LCD display that was also an input device, i.e., where pixels were input as well as output devices. Led to two initiatives. (Think of the paper-cup and string "walkie-talkies" that we all made as kids: the cups were bidirectional and functioned simultaneously as both a speaker and a microphone.)
- Took the high res 2D a-Si scanner technology used in our scanners and adding layers to make them displays. The bi-directional motivation got lost in the process, but the result was the dpix display (http://www.dpix.com/about.html);
- The Liveboard project. The rear projection Liveboard was initially conceived as a quick prototype of a large flat panel version that used a tiled array of bi-directional dpix displays.

1991: Digital Desk(Pierre Wellner, Rank Xerox EuroPARC, Cambridge)

- A classic paper in the literature on augmented reality.
- Wellner, P. (1991). The Digital Desk Calculator: Tactile manipulation on a desktop display. *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology (UIST '91)*, 27-33.
- An early front projection tablet top system that used optical and acoustic techniques to sense both hands/fingers as well as certain objects, in particular, paper-based controls and data.
- Clearly demonstrated multi-touch concepts such as two finger scaling and translation of graphical objects, using either a pinching gesture or a finger from each hand, among other things.
- For example, see segment starting at 6:30 in the following 1991 video demo: http://video.google.com/videoplay?docid=5772530828816089246





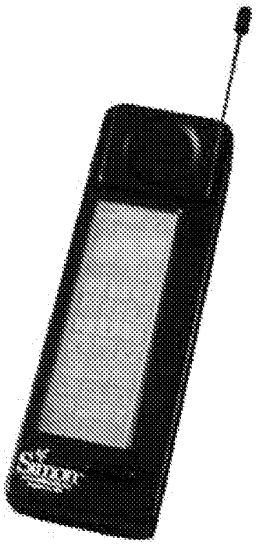
1992: Flip Keyboard(Bill Buxton, Xerox PARC): www.billbuxton.com

- A multi-touch pad integrated into the bottom of a keyboard. You flip the keyboard to gain access to the multi-touch pad for rich gestural control of applications.
- Combined keyboard / touch tablet input device (1994). Click here for video (from 2002 in conjunction with Tactex Controls).

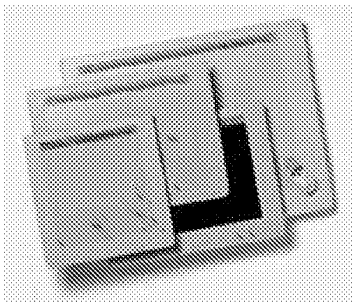
Sound
Synthesizer
Audio Mixer
Graphics on multi-touch surface defining controls for various virtual devices.

1992: Simon (IBM & Bell South)

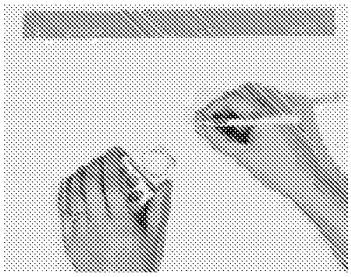
- IBM and Bell South release what was arguably the world's first smart phone, the *Simon.*
- What is of historical interest is that the Simon, like the iPhone, relied on a touch-screen driven "soft machine" user interface.
- While only a single-touch device, the Simon foreshadows a number of aspects of what we are seeing in some of the touch-driven mobile devices that we see today.
- Sidebar: my two working Simons are among the most prized pieces in my collection of input devices.



1992: Wacom (Japan)



- In 1992 Wacom introduced their UD series of digitizing tablets. These were special in that they had mutli-device / multi-point sensing capability. They could sense the position of the stylus and tip pressure, as well as simultaneously sense the position of a mouse-like puck. This enabled bimanual input.
- Working with Wacom, my lab at the University of Toronto developed a number of ways to exploit this technology to far beyond just the stylus and puck. See the work on Graspable/Tangible interfaces, below.
- Their next two generations of tablets, the Intuos 1 (1998) and Intuos 2 (2001) series extended the multi-point capability. It enabled the sensing of the location of the stylus in x and y, plus tilt in x and tilt in y (making the stylus a location-sensitive joystick, in effect), tip pressure, and value from a side-mounted dial on their airbrush

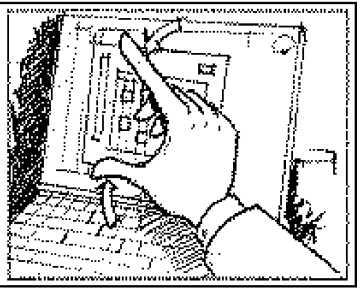


stylus. As well, one could simultaneously sense the position and rotation of the puck, as well as the rotation of a wheel on its side. In total, one was able to have control of 10 degrees of freedom using two hands.

- While this may seem extravagant and hard to control, that all depended on how it was used. For example, all of these signals, coupled with bimanual input, are needed to implement any digital airbrush worthy of the name. With these technologies we were able to do just that with my group at Alias|Wavefront, again, with the cooperation of Wacom.
- See also: Leganchuk, A., Zhai, S.& Buxton, W. (1998). Manual and Cognitive Benefits of Two-Handed Input: An Experimental Study. *Transactions on Human-Computer Interaction,* 5(4), 326-359.

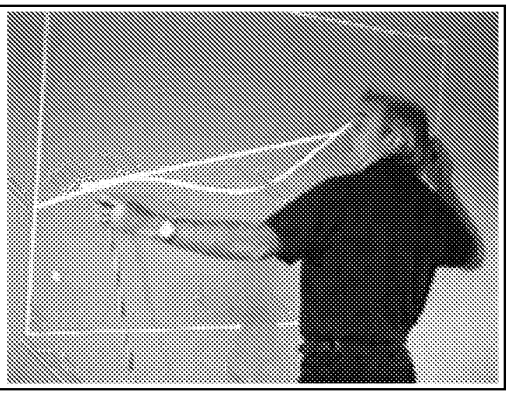
1992: Starfire (Bruce Tognazinni, SUN Microsystems)

- Bruce Tognazinni produced an future envisionment film, Starfire, that included a number of multi-hand, multi-finger interactions, including pinching, etc.



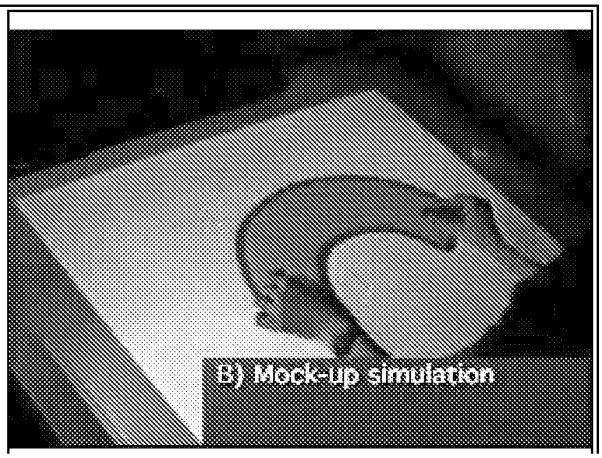
1994-2002: Bimanual Research (Alias|Wavefront, Toronto)

- Developed a number of innovative techniques for multi-point / multi-handed input for rich manipulation of graphics and other visually represented objects.
- Only some are mentioned specifically on this page.
- There are a number of videos can be seen which illustrate these techniques, along with others: http://www.billbuxton.com/buxtonAliasVideos.html
- Also see papers on two-handed input to see examples of multi-point manipulation of objects at: http://www.billbuxton.com/papers.html#anchor1442822



1995: Graspable/Tangible Interfaces (Input Research Group, University of Toronto)

- Demonstrated concept and later implementation of sensing the identity, location and even rotation of multiple physical devices on a digital desk-top display and using them to control graphical objects.
- By means of the resulting article and associated thesis introduced the notion of what has come to be known as "graspable" or "tangible" computing.
- Fitzmaurice, G.W., Ishii, H. & Buxton, W. (1995). Bricks: Laying the foundations for graspable user interfaces. *Proceedings of the ACMSIGCHI Conference on Human Factors in Computing Systems (CHI'95),* 442–449.

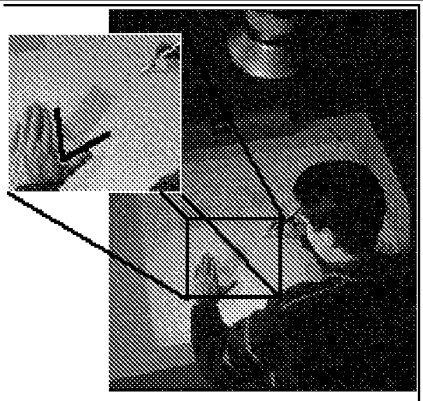


1995: DSI Datotech (Vancouver BC)

- In 1995 this company made a touch tablet, the *HandGear*, capable of multipoint sensing. They also developed a software package, *Gesture Recognition Technology (GRT)*, for recognizing hand gestures captured with the tablet.
- The company went out of business around 2002

1995/97: Active Desk (Input Research Group / Ontario Telepresence Project,University of Toronto)

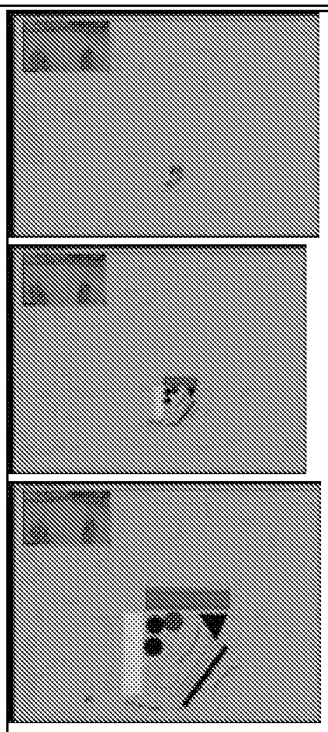
- Around 1992 we made a drafting table size desk that had a rear-projection data display, where the rear projection screen/table top was a translucent stylus controlled digital graphics tablet (Scriptel). The stylus was operated with the dominant hand. Prior to 1995 we mounted a camera bove the table top. It tracked the position of the non-dominant hand on the tablet surface, as well as the pose (open angle) between the thumb and index finger. The non-dominant hand could grasp and manipulate objects based on what it was over and opening and closing the grip on the virtual object. This vision work was done by a student, Yuyan Liu.
- Buxton,W. (1997). Living in Augmented Reality: Ubiquitous Media and Reactive Environments. In K. Finn, A. Sellen & S. Wilber (Eds.). *Video Mediated Communication. Hillsdale, N.J.: Erlbaum, 363-384. An earlier version of this chapter also appears in Proceedings of Imagina '95, 215-229.*

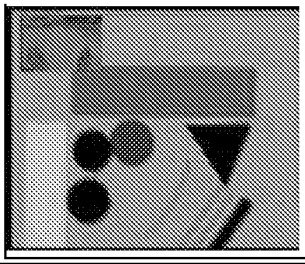


Simultaneous bimanual and multi-finger interaction on large interactive display surface

1997: T3 (Alias|Wavefront, Toronto)

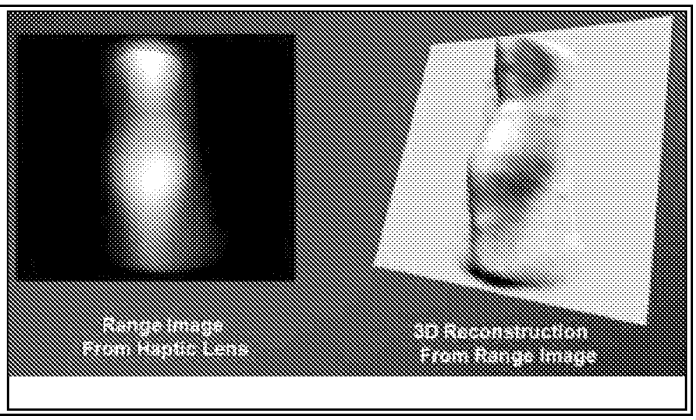
- T3 was a bimanual tablet-based system that utilized a number of techniques that work equally well on multi-touch devices, and have been used thus.
- These include, but are not restricted to grabbing the drawing surface itself from two points and scaling its size (i.e., zooming in/out) by moving the hands apart or towards each other (respectively). Likewise the same could be done with individual graphical objects that lay on the background. (Note, this was simply a multi-point implementation of a concept seen in Ivan Sutherland's Sketchpad system.)
- Likewise, one could grab the background or an object and rotate it using two points, thereby controlling both the pivot point and degree of the rotation simultaneously. Ditto for translating (moving) the object or page.
- Of interest is that one could combine these primitives, such as translate and scale, simultaneously (ideas foreshadowed by Fitzmaurice's graspable interface work – above).
- Kurtenbach, G., Fitzmaurice, G., Baudel, T. & Buxton, W. (1997). The design and evaluation of a GUI paradigm based on tabets, two-hands, and transparency. *Proceedings of the 1997 ACM Conference on Human Factors in Computing Systems, CHI '97,* 35-42. [Video].





1997: The Haptic Lens (Mike Sinclair, Georgia Tech / Microsoft Research)

- The Haptic Lens, a multi-touch sensor that had the feel of clay, in that it deformed the harder you pushed, and resumed it basic form when released. A novel and very interesting approach to this class of device.
- Sinclair, Mike (1997). The Haptic Lens. *ACM SIGGRAPH 97 Visual Proceedings: The art and interdisciplinary programs of SIGGRAPH '97,* Page: 179

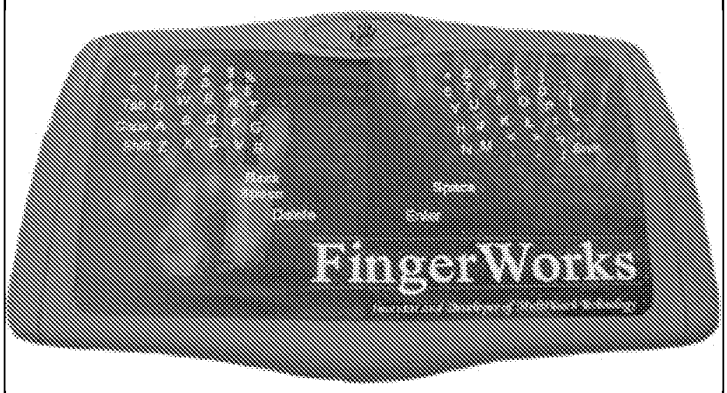


1998: Tactex Controls (Victoria BC) http://www.tactex.com/

- Kinotex controller developed in 1998 and shipped in Music Touch Controller, the MTC Express in 2000.
- See video at: http://www.billbuxton.com/flip_keyboard_s.mov

~1998: Fingerworks (Newark, Delaware).

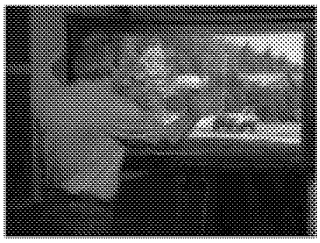
- Made a range of touch tablets with multi-touch sensing capabilities, including the *iGesture Pad*. They supported a fairly rich library of multi-point / multi-finger gestures.
- Founded by two University of Delaware academics, John Elias and Wayne Westerman
- Product largely based on Westerman's thesis: Westerman, Wayne (1999). *Hand Tracking,Finger Identification, and Chordic Manipulation on a Multi-Touch Surface.* U of Delaware PhD Dissertation: http://www.ee.udel.edu/~westerma/main.pdf
- Note that Westerman's work was solidly built on the above work. His thesis cites Matha's 1982 work which introduced multi-touch, as well as Krueger's work, which introduced - among other things - the pinch gesture. Of the 172 publications cited, 34 (20%) are authored or co-authored by me an/or my students.
- The company was acquired in early 2005 by Apple Computer.
- Elias and Westerman moved to Apple.
- Fingerworks ceased operations as an independent



company.
- However, it left a lot of fans, and documentation, including tutorials and manuals are still downloadable from: http://www.fingerworks.com/downloads.html

1999: Portfolio Wall (Alias|Wavefront,Toronto On, Canada)

- A product that was a digital cork-board on which images could be presented as a group or individually. Allowed images to be sorted, annotated, and presented in sequence.
- Due to available sensor technology, did not us multi-touch; however, its interface was entirely based on finger touch gestures that went well beyond what typical touch screen interfaces were doing at the time, and which are only now starting to appear on some touch-based mobile devices.
- For example, to advance to the next slide in a sequence, one flicked to the right. To go back to the previous image, one flicked left.
- The gestures were much richer than just left-right flicks. One could instigate different behaviours, depending on which direction you moved your finger.
- In this system, there were eight options, corresponding to the 8 main points of the compass. For example, a downward gesture over a video meant "stop". A gesture up to the right enabled annotation. Down to the right launched the application associated with the image. etc.
- They were self-revealing, could be done eyes free, and leveraged previous work on "marking menus."
- See a number of demos at: http://www.billbuxton.com/buxtonAliasVideos.html

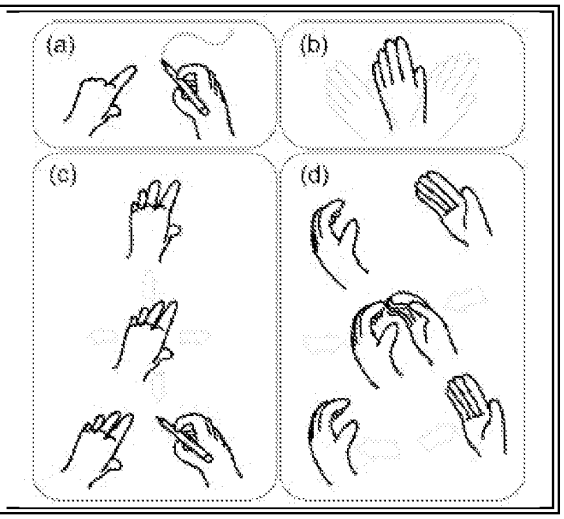


Touch to open/close image
Flick right = next
Flick left = previous

Portfolio Wall (1999)

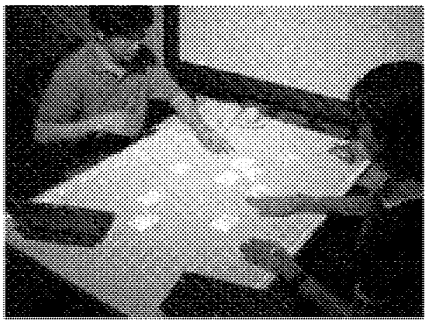
2001: Diamond Touch (Mitsubishi Research Labs, Cambridge MA)
http://www.merl.com/

- example capable of distinguishing which person's fingers/hands are which, as well as location and pressure
- various gestures and rich gestures.
- http://www.diamondspace.merl.com/



2002: Jun Rekimoto Sony Computer Science Laboratories (Tokyo)
http://www.csl.sony.co.jp/person/rekimoto/smartskin/

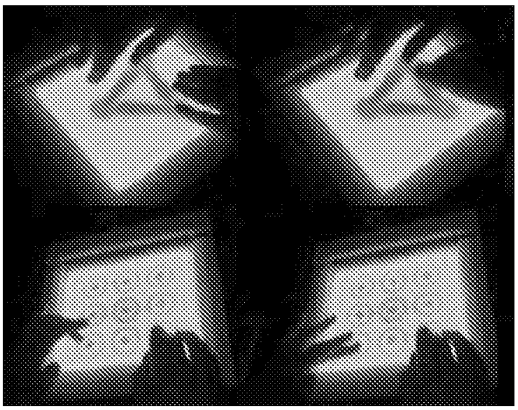
- *SmartSkin*: an architecture for making interactive surfaces that are sensitive to human hand and finger gestures. This sensor recognizes multiple hand positions and their shapes as well as calculates the distances between the hands and the surface by using capacitive sensing and a mesh-shaped antenna. In contrast to camera-based gesture recognition systems, all sensing elements can be integrated within the surface, and this method does not suffer from lighting and



occlusion problems.

- SmartSkin: An Infrastructure for Freehand Manipulation on Interactive Surfaces. *Proceedings of ACM SIGCHI*.
- Kentaro Fukuchi and Jun Rekimoto, Interaction Techniques for SmartSkin, ACM UIST2002 demonstration, 2002.
- SmartSkin demo at Entertainment Computing 2003 (ZDNet Japan)
- Video demos available at website, above.

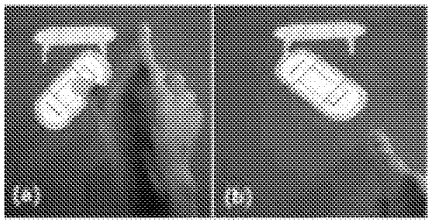
2002: Andrew Fentem (UK) http://www.andrewfentem.com/



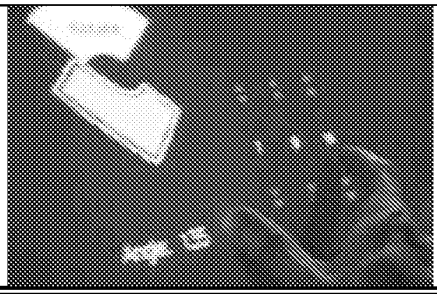
- States that he has been working on multi-touch for music and general applications since 2002
- However, appears not to have published any technical information or details on this work in the technical or scientific literature.
- Hence, the work from this period is not generally known, and - given the absence of publications - has not been cited.
- Therefore it has had little impact on the larger evolution of the field.
- This is one example where I am citing work that I have *not* known and loved for the simple reason that it took place below the radar of normal scientific and technical exchange.
- I am sure that there are several similar instances of this. Hence I include this as an example representing the general case.

2003: University of Toronto (Toronto)

- paper outlining a number of techniques for multi-finger, multi-hand, and multi-user on a single interactive touch display surface.
- Many simpler and previously used techniques are omitted since they were known and obvious.
- Mike Wu, Mike & Balakrishnan, Ravin (2003). Multi-Finger and Whole Hand Gestural Interaction Techniques for Multi-User Tabletop Displays. *CHI Letters*

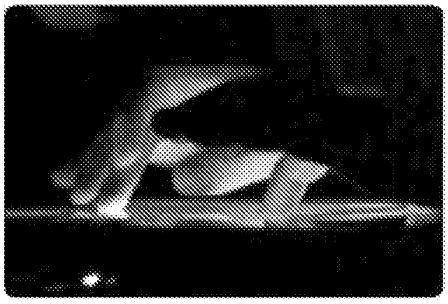


Freeform rotation. (a) Two fingers are used to rotate an object. (b) Though the pivot finger is lifted, the second finger can continue the rotation.



This parameter adjustment widget allows two-fingered manipulation.

2003: Jazz Mutant (Bordeaux France) http://www.jazzmutant.com/
Stantum: http://stantum.com/

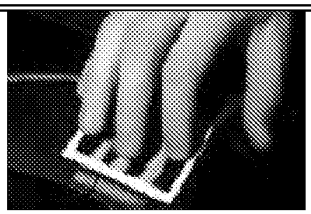


- Make one of the first transparent multi-touch, one that became – to the best of my knowledge – the first to be offered in a commercial product.
- The product for which the technology was used was the *Lemur,*a music controller with a true multi-touch screen interface.
- An early version of the Lemur was first shown in public in LA in August of 2004.
- Jazz Mutant is the company that sells the music product, while Stantum is the sibling company set up to sell the underlying multi-touch technology to other

2004: TouchLight (Andy Wilson, Microsoft Research): http://research.microsoft.com/~awilson/

- TouchLight (2004). A touch screen display system employing a rear projection display and digital image processing that transforms an otherwise normal sheet of acrylic plastic into a high bandwidth input/output surface suitable for gesture-based interaction. Video demonstration on website.
- Capable of sensing multiple fingers and hands, of one or more users.
- Since the acrylic sheet is transparent, the cameras behind have the potential to be used to scan and display paper documents that are held up against the screen .

2005: Blaskó and Steven Feiner (Columbia University):
http://www1.cs.columbia.edu/~gblasko/



- Using pressure to access virtual devices accessible below top layer devices
- Gábor Blaskó and Steven Feiner (2004). Single-Handed Interaction Techniques for Multiple Pressure-Sensitive Strips,
  Proc. ACM Conference on Human Factors in Computing Systems (CHI 2004) Extended Abstracts, 1461-1464

2005: PlayAnywhere (Andy Wilson, Microsoft Research):
http://research.microsoft.com/~awilson/

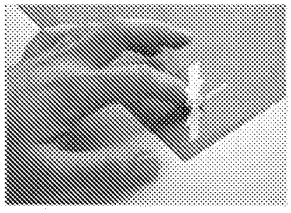
- PlayAnywhere (2005). Video on website
- Contribution: sensing and identifying of objects as well as touch.
- A front-projected computer vision-based interactive table system.
- Addresses installation, calibration, and portability issues that are typical of most vision-based table systems.
- Uses an improved shadow-based touch detection algorithm for sensing both fingers and hands, as well as objects.
- Object can be identified and tracked using a fast, simple visual bar code scheme. Hence, in addition to manual mult-touch, the desk supports interaction using various physical objects, thereby also supporting graspable/tangible style interfaces.
- It can also sense particular objects, such as a piece of paper or a mobile phone, and deliver appropriate and desired functionality depending on which..

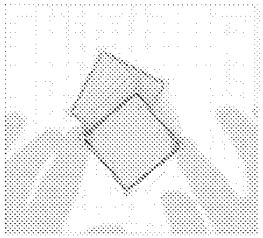
2005: Jeff Han (NYU): http://www.cs.nyu.edu/~jhan/

2006: (Perceptive Pixel: http://www.perceptivepixel.com/)

- Very elegant implementation of a number of techniques and applications on a table format rear projection surface.
- Multi-Touch Sensing through Frustrated Total Internal Reflection (2005). Video on website.
- Formed Perceptive Pixel in 2006 in order to further develop the technology in the private sector
- See the more recent videos at the Perceptive Pixel site: http://www.perceptivepixel.com/

2005: Tactiva (Palo Alto) http://www.tactiva.com/





- Have announced and shown video demos of a product called the TactaPad.
- It uses optics to capture hand shadows and superimpose on computer screen, providing a kind of immersive experience, that echoes back to Krueger (see above)
- Is multi-hand and multi-touch
- Is tactile touch tablet, i.e., the tablet surface feels different depending on what virtual object/control you are touching

2005: Toshiba Matsusita Display Technology (Tokyo)

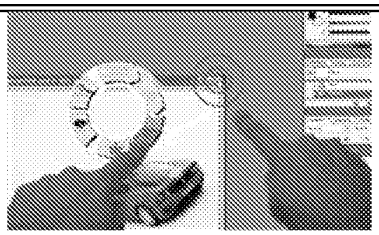
- Announce and demonstrate LCD display with "Finger Shadow Sensing Input" capability
- One of the first examples of what I referred to above in the 1991 Xerox PARC discussions. It will not be the last.
- The significance is that there is no separate touch sensing transducer. Just as there are RGB pixels that can produce light at any location on the screen, so can pixels detect shadows at any location on the screen, thereby enabling multi-touch in a way that is hard for any separate touch technology to match in performance or, eventually, in price.
- http://www3.toshiba.co.jp/tm_dsp/press/2005/05-09-29.htm

2005: Tomer Moscovich & collaborators (Brown University)

- a number of papers on web site: http://www.cs.brown.edu/people/tm/
- T. Moscovich, T. Igarashi, J. Rekimoto, K. Fukuchi, J. F. Hughes. "A Multi-finger Interface for Performance Animation of Deformable Drawings." Demonstration at *UIST 2005 Symposium on User Interface Software and Technology*, Seattle, WA, October 2005. (video)

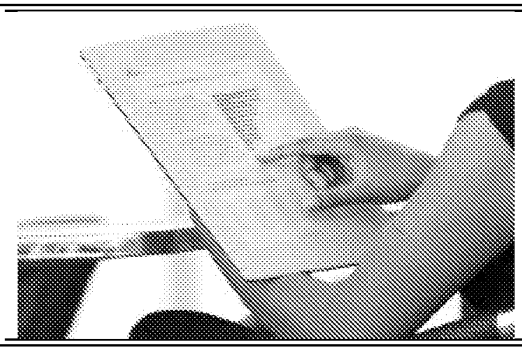
2006: Benko & collaborators (Columbia University & Microsoft Research)

- Some techniques for precise pointing and selection on muti-touch screens
- Benko, H., Wilson, A. D., and Baudisch, P. (2006). Precise Selection Techniques for Multi-Touch Screens. *Proc. ACM CHI 2006 (CHI'06: Human Factors in Computing Systems,* 1263–1272
- video



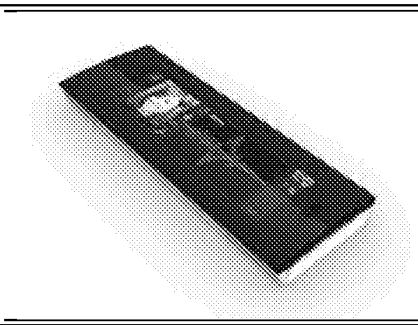
2006: Plastic Logic (Cambridge UK)

- A flexible e-ink display mounted over a multi-point touch pad, thereby creating an interactive multi-touch display.
- Was an early prototype of their ill-fated QUE e-reader



2006: Synaptics & Pilotfish (San Jose) http://www.synaptics.com

- Jointly developed Onyx, a soft multi-touch mobile phone concept using transparent Synaptics touch sensor. Can sense difference of size of contact. Hence, the difference between finger (small) and cheek (large), so you can answer the phone just by holding to cheek, for example.
- http://www.synaptics.com/onyx/

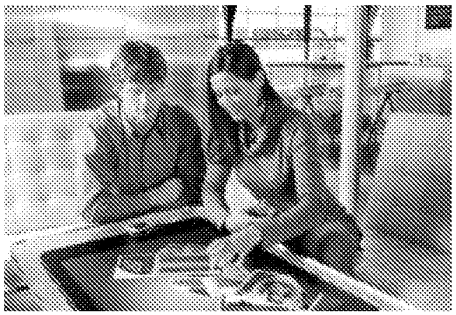


2007: Apple iPhone http://www.apple.com/iphone/technology/

- Like the 1992 Simon (see above), a mobile phone with a soft touch-based interface.
- Outstanding industrial design and very smooth interaction.
- Employed multi-touch capability to a limited degree
- Uses it, for example, to support the "pinching" technique introduced by Krueger, i.e., using the thumb and index finger of one hand to zoom in or out of a map or photo.
- Works especially well with web pages in the browser
- Uses Alias Portfolio Wall type gestures to flick forward and backward through a sequence of images.
- Did not initially enable use of multi-touch to hold shift key with one finger in order to type an upper case character with another with the soft virtual keyboard. This did not get implemented until about a year after its release.

2007: Microsoft Surface Computing http://www.surface.com

- Interactive table surface
- Capable of sensing multiple fingers and hands
- Capable of identifying various objects and their position on the surface
- Commercial manifestation of internal research begun in 2001 by Andy Wilson (see above) and Steve Bathiche
- Image is displayed by rear-projection and input is captured opticaly via cameras
- A key indication of this technology making the transition from research, development and demo to mainstream commercial applications.
- See also ThinSight and Surface 2.0

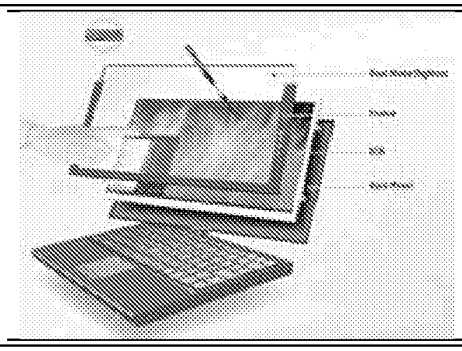


2007: ThinSight, (Microsoft Research Cambridge,UK)
http://www.billbuxton.com/UISTthinSight.pdf

- Thin profile multi-touch technology that can be used with LCD displays.
- Hence, can be accommodated by laptops, for example
- Optical technology, therefore capable of sensing both fingers and objects
- Therefore, can accommodate both touch and tangible styles of interaction
- Research undertaken and published by Microsoft Research
- see also Surface 2.0

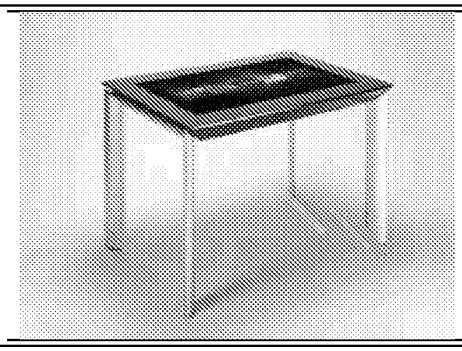
2008: N-trig http://www.n-trig.com/

- Commercially multi-touch sensor
- Can sense finger and stylus simultaneously
- unlike most touch sensors that support a stylus, this incorporates specialized stylus sensor
- result is much higher quality digital ink from stylus
- Incorporated into some recent Tablet-PCs
- Technology scales to larger formats, such as table-top size



2011: Surface 2.0 (Microsoft & Samsung) http://www.microsoft.com/surface/

- 4" thick version of Surface
- Rear projection and projectors replaced by augmented LCD technology
- builds on research such as ThinSight
- result is more that just a multi-touch surface
- since pixels have integrated optical sensors, the whole display is also an imager
- hence, device can "see" what is placed on it, including shapes, bar-codes, text, drawings, etc. - and yes - fingers



One-Point Touch Input of Vector Information for Computer Displays

Christopher F. Herot*
Guy Weinzapfel
Architecture Machine Group
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

The finger as a graphical stylus enjoys a coefficient of friction with glass sufficient to provide input of direction and torque as well as position from a single point. This report describes a pressure-sensitive digitizer (PSD) capable of accepting these force inputs, and discusses a set of five simple input applications used to assess the capabilities of this device. These applications include techniques for specifying vectors, and pushing, pulling, dispersing and reorienting objects with a single touch. Experience gained from these applications demonstrates that touch and pressure sensing open a rich channel for immediate and multi-dimensional interaction.

Key Words: Touch Input, Pressure Sensing, Force Input, Tactile Input, Kinesthetic Input, Pressure Sensitive Digitizer, Touch Sensitive Digitizer.

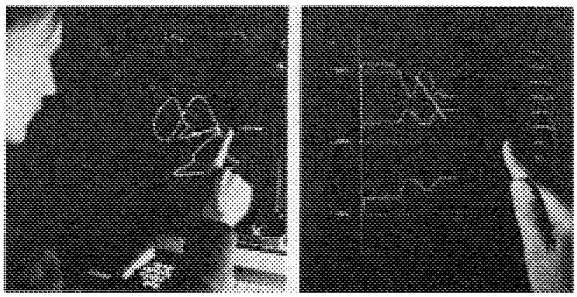
1.0 INTRODUCTION

It is a central thesis of the Architecture Machine Group, that work places as opposed to work stations, are a necessary ingredient for the amplification of creativity. (1) Work places are defined as having a multiplicity of interactive media which encourage a high degree of motor involvement - tactile participation. By austere comparison, work stations are characterized by the all-too-prevalent black and white CRT with its keyboard and occasional light pen or other stylus. The need for multimedia is based on the assertion that, regardless of task, information relating to creative performance is best perceived through a variety of senses including at the least sight, sound, and touch. While several of our current projects explore the integration of multiple media (2,3,4), this paper reports on one effort to develop a channel of tactile input.

Recently, interest has grown around a class of instruments known as touch sensitive digitizers (TSDs). Using a variety of technologies (5), these devices are capable of determining the X, Y position of a finger's touch without resorting to an intermediate physical stylus.

The work reported herein was conducted between July 1, 1977, and October 31, 1977, under Army Research Institute Grant number DAHC19-77-G-0014, Nicholas Negroponte, principal investigator.

* Mr. Herot's current address: Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts.



The excitement generated by TSDs derives directly from their ability to provide a more natural input path to the computer. The umbilical cord attached to the conventional stylus is removed; in fact the entire notion of a physical stylus is voided. Also, dislocations caused by separate input and presentation surfaces can be circumvented by superimposing transparent TSDs directly over display surfaces.

While the potentials for more natural, coincident and even multi-finger input are obvious and are being developed by other programs(6), little exploration has been undertaken in the area of multi-dimensional input - the sensing of pressure as well as location parameters(7,8,9). Yet this domain offers a rich potential for man-machine interaction. The work described in the following pages was designed to explore that potential - to test the ability of the human finger to input variable pressure and direction from a single touch.

1.1 OUR LABORATORY'S TSD.

In April, 1976, the Architecture Machine Group acquired a TSD from Instronics, Ltd. of Ontario, Canada. This device consists of a sheet of clear glass with piezoelectric transducers mounted on two adjacent edges. The glass is doubly curved to match the face of a display tube(10). The transducers are used to induce acoustic waves in the surface of the glass. These waves are reflected back to their source by fingers touched to the glass surface. The location of the touch is determined by ranging those echos(11).

It was hoped that the TSD would enable users to sweep their fingers over the display surface, thus drawing, even "fingerpainting," with the computer. It was found, however, that in order to insure proper input readings, users had to press the TSD with a force that generated friction between finger and glass sufficient to prevent smooth, sweeping gestures. As a result, the device seemed better suited to pointing than to drawing or painting.

This reality, however, opened the possibility of using the finger-glass friction to unique advantage. Namely, the TSD could be mounted on the display with strain gauges such that forces induced by the finger could be used to input pressures both normal to and parallel with the input surface. In this way, the device could become a pressure-(as well as touch-) sensitive digitizer - a TSD/PSD.

Such a configuration was implemented (as described in Section 3.0) and provided the basis for a four month research program designed to evaluate the characteristics of pressure sensitive input. The following section discusses the methods used to conduct that evaluation.

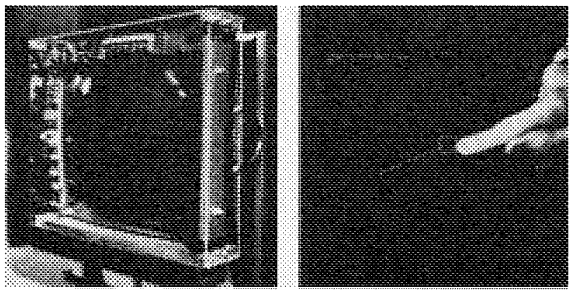
2.0 APPLICATIONS.

Five input routines were developed to assess the input characteristics of the PSD. These included:

1. Force Cursor,
2. Vector History,
3. Pushing/Pulling,
4. Dispersion, and
5. Rotation

In addition, an attempt was made to utilize the X and Y torques to determine the position of the finger, so as to eliminate the need for a TSD altogether.

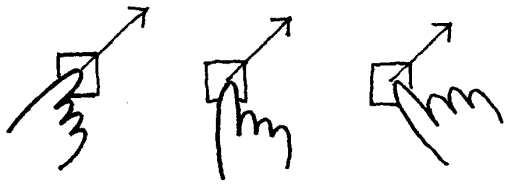
Due to the short duration of the project, evaluation of the device was limited to informal use of the five input routines by a diverse user population, consisting of the laboratory staff and the many visitors which the laboratory attracts from computer science, the arts and various industries. No attempt was made to quantify improvements in throughput, productivity, or task enjoyment resulting from use of the device. However, all users agreed that improvements were indicated in each of these areas.



2.1 FORCE CURSOR.

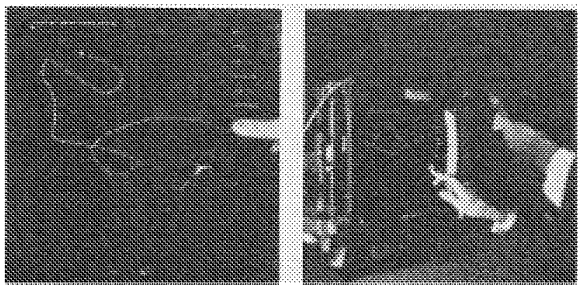
The initial routine provides the pressure sensing equivalent of a conventional cursor - that is, a graphic feedback mechanism which shows the user what is being input. The routine does this by displaying a vector, or arrow, whose origin coincides with the touch point, whose head lies in the direction of the force being exerted by the finger, and whose length is proportional to that force. At the same time, the z force (pressure normal to the face of the screen) is reported as a square, whose size is proportional to that force.

Use of the force cursor has produced some surprising results. While its function is obvious to all who observe it, many people experience initial difficulty making it behave as they expect. Most notably, novice users have difficlty making the vector point in the directions they desire. This difficulty derives not from the equipment, but from the fact that people do not always press in the direction which their finger appears to indicate. Typically, this problem is encountered with the user's first vector. The novice will press the surface of the device, causing an arrow to appear in proper alignment with the finger; but as the finger is rotated, the direction of the vector often fails to follow. Close observation has revealed that this results from the fact that the user actually maintains pressure in the original direction though the finger changes orientation.



Fortunately, the learning curve with this routine is quite steep. This most certainly has to do with the fact that the device takes advantage of the user's existing eye-hand coordination skills. Following some initial difficulty, most users are able to control the direction of their vectors with less than a minute's practice. In fact, many users, realizing that the orientation of their fingers is irrelevant to the direction of the vector, are able to manipulate the cursor from a single, natural hand position.

Beyond this initial training problem, there was a more chronic difficulty: placing the tip of the arrow with acceptable accuracy. This was expecially true as greater extensions (and hence larger forces) were attempted. This problem is similar to that of using a long pointer at a blackboard; the vector bobbed and wobbled at its greatest extension. To counteract this drawback, a damping effect was added to the cursor routine to filter out minor pressure fluctuatons. This filter proved a sufficient solution, as users are now able to point at specific targets (e.g., the menu labels) from origins well across the screen. The force cursor demonstrates that the PSD can be used for reasonably accurate inputs of direction and magnitude.



2.2 VECTOR HISTORY.

The second routine was designed to evaluate the potential for guiding a cursor from a stationary input position. In this case, the cursor scribed a path as it moved under control of the finger's pressure. This routine underwent two implementations. In the first, the speed of the cursor was constant; only its direction was controlled through the PSD. A later implementation allowed the speed to be controlled as well.

Most users, having trained with the force vector, encountered little difficulty in directing the mobile cursor. For example, many people were able to write their names on their first attempt.

Surprisingly, though, the variable-speed version was more difficult to use. This results from the fact that as the cursor deviates from an intended path, most people's reaction is to press harder on the input surface. Since this does not necessarily change the cursor's direction but does increase its speed, "errors" are exaggerated.

Nonetheless, the process of controlling a mobile cursor from a single point on the screen appears to be an engaging and successful use of the device. Real world applications (such as navigating about a map display) can easily be imagined for its use.

2.3 PUSHING/PULLING.

To explore the PSD's potential for moving objects other than a cursor, a routine was implemented which allows users to move objects about the screen. This routine differs from the previous capability, as a specific object is indicated simultaneously with the input of force. Here, the user points to an object and gives it a push. The touch is used to identify the object, the pressure to impart a direction and speed. Thereafter, the user does not need to track the object with the finger but can direct its movement from a static position. Use has demonstrated this routine to be a viable means for directing the movement of selected objects, as users are able to reposition objects with considerable ease.

It was hoped that this routine might also provide users with a sense for the relative "weights" of displayed objects. To test this potential, the routine was elaborated to incorporate parameters for differentially weighted objects. That is, the routine would cause a "lighter" object to move in response to a lighter touch than that required for a "heavier" object.

However, the routine failed to provide the desired perception of weighted objects. This failure was attributed to the absence of an essential mode of feedback from the input device - namely, the tactile/kinesthetic sensation of the object's physical displacement. When a person pushes an object in the natural environment, the weight of the object is reported not only by the pressure returned to the finger, but the movement which is both seen by the eyes and felt by the finger. In short, several feedback channels coalesce to impart a coherent perception of the object's physical properties(12). In the case of the pressure-sensitive device, there is a conflict between the kinesthetic response of the real object (the glass surface), which the finger reports as stationary, and the virtual object which the eyes

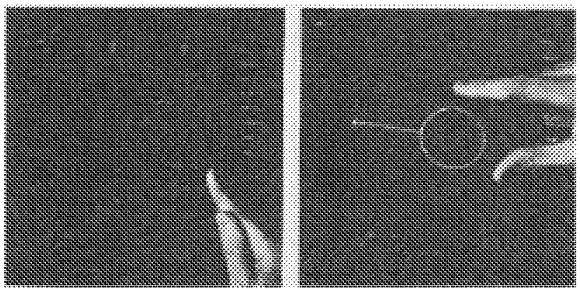
report as moving. This conflict is sufficient to impair the appraisal of the object's weight reported by the finger-sensed pressure. This is not to say that users gained no perception of weightings, for it was clear to all users that some objects moved more easily than others. But no one was able to say that one object was "twice as heavy" as another.

Nonetheless, it should be emphasized that the limited perceptions of weight did not impair the user's ability to manipulate the objects. Most users were equally comfortable using either routine to relocate objects.

2.4 DISPERSION.

In perhaps the most engaging of all the PSD applications, a graphic "shooting gallery" was devised to test the device's ability to accommodate inputs which disperse numbers of elements in various directions. This routine causes small, BB-like circles to emanate from the user's finger tip as it is pressed on the screen's surface. The number, speed, and direction of the BB's is controlled by the pressure of the user's finger. A procession of moving targets (in fact, small ducks) is played across the top of the screen to test the accuracy of the users "shots."

Interestingly enough, even users who had experienced some difficulties with the previous routines adapted to the requirements of this application quite rapidly. In fact, some "hunters" advanced to the point where selected ducks could be felled with single shots. This calls for very accurate control indeed.



2.5 ROTATION.

The fifth routine was designed to evaluate the PSD's ability to measure torque inputs and to use those measurements to advantage in interaction. For this purpose, a simple knob is displayed on the screen with an arrow indicating its angular position. It was hoped that torque about the z axis could be measured with sufficient sensitivity that even minute twists of a single finger could be used to turn the displayed knob. However, when the device was tuned to a level sensitive enough to measure these subtle inputs, the user's intentions were overshadowed by vibrations in the room and in the equipment itself. Once the sensitivity of the z torque pickup was lowered, it became possible for users to turn the knob with two fingers. In fact, the position of the knob can be adjusted to within 5 degrees of rotation with little difficulty. Further tuning of the algorithm and the hardware might permit even greater accuracy.

Though a single, rather sizeable knob was used for this application, the success achieved opens numerous additional possibilities. For example, specific machine parts in a complex display could be identified and reoriented via simple, direct manipulation, thus obviating the need for multiple commands for object selection and action specification.

2.6 POSITION DETECTION.

In addition to the more elaborate capabilities described above, it was hoped that a means of detecting the position of a surface touch could be accomplished directly by the PSD without using the echo ranging of the Instronics device. The algorithm used for this measurement divided the X and Y torques by the Z force. The results of this function were normalized for the direction of forces parallel to the input surface and amplified to produce the location of the finger. This approach produced a calculated touch point with a resolution equal to that of the TSD, but the locus of the point was influenced by the force and direction of the touch. The source of this influence was never adaquately understood, and no solution was conceived in the course of the study (13).

2.7 EXTENSIBILITY.

It should be noted that the applications described above were selected because they could all be accomplished in the time available. It was clear from the onset how each capability should work, and the amount of programming required for each was quite limited. In short, the routines were appropriately matched to the four month duration of the research.

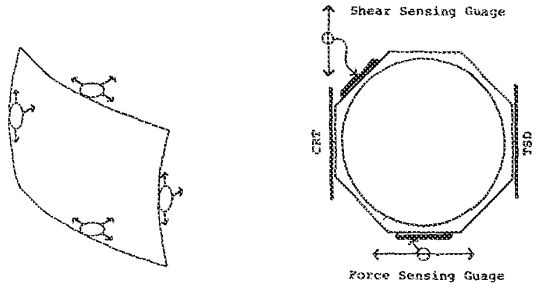
It is not difficult, however, to conceive of more elaborate uses for a pressure sensitive device. For example, a three dimensional dynamic modeling system could use the PSD for tactile manipulation of machine parts, building volumes, and the like. It is easy to imagine turning a machine part by twisting its representation on the screen, or rotating a building display by pushing on a corner. In short, the potentials for tactile involvement and physical feedback from such a device were only hinted at by this brief exploratory work.

3.0 PRINCIPLES OF OPERATION.

The PSD employs eight strain gauges, two each secured to mounting rings centered on the four sides of the TSD. Of the two gauges secured to each ring, one measures force perpendicular to the glass and the other measures shear parallel to the glass. These eight measurements are then used to derive the three force and three torque outputs which are used by the routines described in the previous section.

3.1 MOUNTING AND STRAIN GAUGES.

The TSD is secured to the CRT by means of four specially machined, octagonal, aluminum rings. All forces exerted on the TSD are transmitted to these rings, thus causing deformations which in turn flex the strain gauges secured to them. The two gauges are cemented to each ring as shown in the adjacent figure. Their placement insures that the forces which they sense are orthogonal to one another.

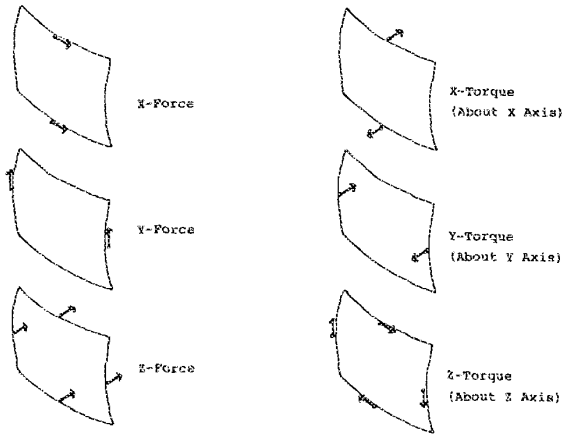


It happens that the thickness, and hence flexibility of these rings is critical to the sensitivity of the gauge's measurements. Unfortunately, the rings machined for this implementation were designed to accommodate very subtle pressures; the fact that the TSD necessitates high finger pressures was not taken into account in their design. Nor was the vibration from nearby machinery foreseen as a problem. As a result, development of the five input routines was somewhat hampered by vibration and pressures which exceeded the output range of the gauges and related circuitry. Were the equipment to be rebuilt, heavier rings would greatly improve its performance. Alternatively, load cells, rather than strain gauges might be used. Load cells measure pressure without deformation. However, these devices are significantly more expensive than the strain gauges used for this implementation.

3.2 ELECTRICAL DESCRIPTION.

The PSD utilized nine BLH (SPB3-35-500) semiconductor strain gauges. Semiconductor gauges were selected because of their sensitivity to miniscule strains. However, as semiconductor devices, they are also very sensitive to changes in temperature. Accordingly, a ninth gauge mounted such that no strain could be exerted upon it is employed to provide a reference output to which all other gauges can be compared. The gauge outputs, which vary between plus and minus 10 millivolts peak to peak, are each connected to preamplifiers which impart a gain of 50; the resultant "raw" output is .5 volts peak to peak.

The "raw" voltages from the strain gauge preamplifiers are combined by sum and difference networks to produce outputs which correspond to X force, Y force, X moment, Y moment, and Z moment. The sums of opposite torque gauges are used to provide the torques about each axis.



The six force and torque outputs are converted to digital signals by a Burr-Brown SDM853 data acquisition system (DAS). The inputs to the DAS are limited to 6.2 volts to prevent overloading the A/D converters. The DAS produces a 12 bit output for each of the 6 analogue inputs.

3.3 DIGITAL INTERFACE.

The outputs from the DAS are stored in a buffer, allowing the DAS to assemble the next sample while waiting for the computer to read the current values.

The computer interface allows program selection of either byte or halfword mode. In byte mode, only 8 most significant bits of each force and torque are used, allowing fast and easy access to the data. In halfword mode, the programing is a bit more complicated, but all of the data bits

are available. Due to the influence of vibration on low order bits, the device was operated primarily in byte mode for the experiments described here.

3.4 SYSTEMS SOFTWARE.

The PSD is equipped to interrupt the computer when data is available. However, since the PSD is always used in conjunction with the TSD, the interrupt circuitry of that device was used. When the TSD detects the finger touch, the program reads the position from the TSD and the forces and torques from the PSD. Since hysteresis of the strain rings and uncompensated temperatre drift often cause the untouched PSD to produce non-zero readings, it is additonally important that the TSD interrupt be used. When the software detects that the device is not being touched, it reads the values of the forces/torques so as to use them as zero references the next time the PSD is touched.

Drift due to temperature changes generated problems for the initial input routines. This was overcome by adding software to sample the force and torque readings when the TSD was not being touched. The latest readings, then, were used as offsets for subsequent inputs. However, this software compensation was made at the expense of the system's overall response range: the offsets biased the device unpredictably. A zeroing circuit was designed to correct for temperature drift in hardware. This circuit was not installed due to the short duration of the study and the anticipated cost associated with its installation.

4.0 CONCLUSIONS.

Development of the PSD and related input routines was undertaken in order to determine answers to several questions. First, we wished to know if it was technically feasible to measure finger pressures on a sheet of glass and to decompose those pressures into their X, Y, Z force and torque components. That question has been answered in the affirmative.

Second, the work was conducted to determine if force and torque inputs could be applied with sufficient accuracy and control to be useful for man-machine communication. All of the input routines indicate that accuracy presents no serious problem, especially where continuous, real-time, graphic feedback is provided (as in the Force Cursor and Rotation routines). Vector History indicates that flexible, easily controlled interaction is possible as well. However, this routine also shows that force input is more suited to the modulation of velocities than for the control of accelerations.

Third, the input routines were used to determine if a pressure-sensitive device could convey more natural perceptions of virtual objects. While limited success was achieved in conveying the differential weights of objects, the quality of such perceptions is only marginally improved by the use of the PSD. It would be misleading to rely upon the device as a mechanism for providing passive force feedback.

Finally, the PSD and its routines were developed to explore any unforeseen benefits which might accrue from the implementation of such a device. Here, two definite advantages can be identified. First, the PSD/TSD combination affords engaging and facile interaction which attracts and maintains the participation of all who witness its use. Second, the device has proven innately simple to use. By capitalizing on natural skills, the PSD enables users to take advantage of virtually all its capabilities within minutes. At a recent open house it was astounding to see four-and five-year-old children pointing at words with the vector, turning the knob about and shooting ducks with obvious glee.

Of course, the PSD's ultimate advantage is its ability to collapse activities which otherwise require several disjoint commands into single, natural, tactile actions.

ACKNOWLEDGMENTS.

The work has been very much a group project, initially launched by experiments with touch sensitive display (TSD) devices, conducted by Richard Bolt and under ARPA contract number MDA-903-76-C-0261, April 1, 1976, to September 30, 1976. During that period, William Donelson, a graduate student, speculated that translational and Z-forces could be sensed by strain gauges. William Kelley, assisted by Robert Hoffman, John Soltes, and Harry Boadwee, constructed the hardware outlined in Section 3.0. Peter Clay, building upon earlier TSD software by Rimas Ignaitis, implemented the input routines outlined in Section 2.0. Finally, Michael Naimark made the film which is to be shown at SIGGRAPH 1978. This long story attributes the conjoint efforts of our laboratory, as encouraged by Dr. Frank Moses of the Army Research Institute.

FOOTNOTES/REFERENCES

1. Nicholas Negroponte. On being creative with computer aided design. In Information Processing 77, B. Gilchrist, Editor, IFIP, North-Holland Publishing Co., New York, 1977.

2. Dr. Richard A. Bolt. Spatial data - management - interim report. Architecture Machine Group, M. I. T. Cambridge, Massachusetts, November, 1977.

3. William Donelson. Spatial management of information. SIGGRAPH '78, Proceedings of the Fifth Annual Conference on Computer Graphics and Interactive Techniques. Atlanta, 1978.

4. Guy Weinzapfel. Mapping by yourself. Proceedings of the conference Interactive Techniques in Computer Aided Design, Bologna, Italy, 1978.

5. A discussion of various TSD technologies is provided by Dr. Richard A. Bolt. Touch sensitive displays. Architecture Machine Group, M.I.T. Cambridge, Massachusetts, September, 1976.

6. TSD work is continuing under the aegis of ONR Contract Number N00014-75-C-0460 with Elographics Corporation furnishing a transparent sensing medium laminated to a Tektronix 650 display.

7. Several "kinesthetic" systems, that is systems incorporating force feedback, are summarized by Frederick P. Brooks, Jr. The computer "scientist" as toolsmith - studies in interactive graphics. In Information Processing 77, B. Gilchrist, Editor, IFIP, North-Holland Publishing Co., New York, 1977. However, none of the systems summarized in this paper correlates physical and grahical feedback at a common locus.

8. P.J. Kilpatrick. The use of a kinesthetic supplement in an interactive graphics system. Ph.D. dissertation, University of North Carolina, Chapel Hill, N.C., 1976.

9. A.M. Noll. Man-machine tactile communication. Ph.D. dissertation, Polytechnic Institute of Brooklyn, Brooklyn, N.Y., 1971.

10. The display used for this project is an IMLAC PDS-1 dynamic CRT. This dislay may be driven by any of the laboratory's several Interdata minicomputers (models 70, 85 or 7/32). The operating system (MAGIC) and the display software are both of Architecture Machine Group design.

11. A more complete technical description of the TSD is provided in the reference cited in (5) above.

12. A general discussion of task performance related to diminished and augmented feedback is provided in Paul M. Fitts and Michael I. Posner. Human Performance. Brooks/Cole Publishing Co., Belmont, California, 1967.

13. Subsequent to this study, it was learned that Robert Anderson and Ivan Sutherland, working at Rand Corporation had explored a pressure-sensing device to locate the x,y position of a touch. Though the Rand device used a considerably different mounting configuration, the calculated touch point migrated as the touch pressure was varied. To minimize this problem, positions were calculated using a low pressure threshold.

Department of Computer & Information Science

Technical Reports (CIS)

University of Pennsylvania *Year* 1981

Real Time Control of a Robot Tacticle Sensor

Jeffrey A. Wolfeld
University of Pennsylvania

UNIVERSITY OF PENNSYLVANIA
THE MOORE SCHOOL OF ELECTRICAL ENGINEERING
SCHOOL OF ENGINEERING AND APPLIED SCIENCE

REAL TIME CONTROL OF A ROBOT TACTILE SENSOR

Jeffrey A. Wolfeld

Philadelphia, Pennsylvania

August, 1981

A thesis presented to the Faculty of Engineering and Applied Science in partial fulfillment of the requirements for the degree of Master of Science in Engineering for graduate work in Computer and Information Science.

_____
Ruzena Bajcsy

_____
Aravind K. Joshi

The work reported here was supported in part by NSF grant number MCS-78-07466.

Jeffrey A. Wolfeld
Masters Thesis

REAL TIME CONTROL OF A ROBOT TACTILE SENSOR

Jeffrey A. Wolfeld

Philadelphia, Pennsylvania

August 1981

Abstract

The goal of the Experimental Sensory Processor project is to build a system which employs both visual and tactile senses, and then explore their interaction in a robotic environment. Here we describe the software involved in the low level control of the tactile branch of this system, and present results of some simple experiments performed with a prototype tactile sensor.

Acknowledgments

I would like to thank the following people:

Jim Korein, my office mate, with whom I had many fascinating discussions between 9:00 and 5:00 on weekdays;

Gerry Radack, who occasionally dragged me away from my terminal in order to play music;

Clayton Dane, who helped keep my feet on the ground;

Jeff Shrager, without whom I might never have gotten past the Abstract;

Taylor Adair, who kept the computer running when it really wanted to crash;

Ira Winston, who served as the local oracle;

Jack Rebman of the Lord Corporation, without whom this thesis would have been entirely speculation;

David Brown, who got me into this mess in the first place;

and my advisor, Ruzena Bajcsy, mother to us all.

Table of Contents

Chapter 1: Introduction

1.1 Motivation

Artificial Intelligence researchers have worked extensively with vision systems in an attempt to give computers, and eventually robots, a sense of sight. A great deal of this research has been directed toward overcoming certain basic inadequacies in our current technology. For example, imperfect light sensors dictate that noise must be eliminated or tolerated. Insufficient spatial resolution requires routines which will interpolate below the pixel level.

One of the most important problems is that a camera produces a two-dimensional image of a three-dimensional scene. This invalidates an assumption which one would like to rely upon -- that two adjacent points in the image are adjacent in the scene. Therefore, substantial effort has been devoted to reproducing 3-D data from one or several visual images. Tactile sensors can be used to aid the process.

An imaging tactile sensor, by its very nature, does not have the problem. Since it produces a two-dimensional image

of a two-dimensional scene, it does not provide as much information, but it yields useful information clearly, without the need for complicated heuristics.

We can take this one step further. Suppose a tactile sensor is mounted on some kind of computer controlled 3-D positioning device. Then, by moving the sensor to different points on a target object, the computer can actually obtain 3-D data directly, and much more selectively. If this information is used to supplement and augment visual data, a great deal of processing may be avoided.

One can come up with many other uses for varying kinds of tactile sensors. Briot [BRIOT-79] demonstrated that tactile sensors mounted on the fingers of a robot hand can be used to determine the position, orientation, and perhaps even the identity of an object which it has grasped. He also showed that a grid of pressure sensitive sites on a table can tell a robot the location, orientation, and again, the identity of a part. It should be possible with multi-valued pressure sensors, as opposed to binary sensors, to determine the mass of the object. When the angle is small, a tactile sensor can be used to compute the angle between it and the object being grasped, possibly with a view toward improving the grip. Also, if the device is sensitive enough, it can be an invaluable aid to a robot attempting to grasp a fragile object without breaking it. Finally, a tactile sensor makes it possible to incorporate the properties of surface texture

and resilience into the object recognition process.

1.2 Project Overview

The design and development of the tactile system has proceeded with two different sensors in mind. Unfortunately, there are so many disparities between the two that we had difficulty keeping the system general enough to handle both. Let this serve as a demonstration of the variety of characteristics that must be considered for a given application.

The first sensor is about five inches long, with an octagonal cross section about 3/4 inches in diameter. Each of the eight rectangular faces is connected to a tapered piece, which is in turn connected to a common tip piece. There are a total of 133 sensitive sites -- 16 on each main face, one on each alternate taper, and one on the tip. Because of the the vague resemblance, we will refer to this sensor as the Finger.

The second sensor, the Pad, is a flat rubber square about two and one half inches on a side. An 8 x 8 grid of conical protrusions identify the 64 pressure sensitive sites. The pad is mounted on a square metal piece, about three and one half inches on a side, which is in turn connected to another similar piece by four metal posts. These posts have strain gauges on them which measure the force parallel to the object's surface.

Initially, we only considered the finger. Because of its shape and organization, the sensor is best suited to applications involving probing and tracing. This includes testing for resiliency, examining surface texture, and tracing cross-sections of an object. In our view, texture would be thought of as a kind of microscopic contour, while the cross-section tracings would yield a macroscopic contour. Taken together, we would be able to acquire an extremely detailed description of very selective parts of the object in question.

Unfortunately, this rather vague idea has not been developed. We have instead dealt with the two descriptions independently with the assumption that they can both be incorporated into a general object recognition system.

For his Master's Thesis, David Brown [BROWN-80] developed a three-dimensional positioning device for the finger. Basically, it is a square horizontal metal frame mounted on four legs. Moving forward and backward on this is a second, vertical square frame. A vertical track rides left to right on that, and a rod moves up and down in the track. The finger would be mounted with its tip downward at the bottom of the rod.

Thus, we have three degrees of freedom -- the X, Y and Z axes -- each positioned by a stepper motor driving a lead screw. This gives us the capability of examining, from the top, any object or objects placed on a table below the

horizontal frame, in a total working volume of about 18 cubic inches. Since the degrees of freedom are strictly positional, as opposed to rotational, we are not capable of reaching under an overhanging lip, or sideways below a covering section. This places certain restrictions on the kind of object we can examine. If we think of the horizontal axes as X and Y, then the object must be describable as a strict function of those two variables. Needless to say, this is not a robot arm, but we felt it would suffice, temporarily at least, for our research.

The positioning device and tactile sensor are directly controlled by a pair of Z80 microprocessors, which are in turn under the command of a PDP-11/60 minicomputer. Of the Z80's, one (the Motor Control Processor, MCP) is responsible for driving and positioning the stepper motors, and the other (the Tactile Sensing Processor, TSP) is dedicated to tactile data acquisition and compression. The MCP and TSP communicate with each other via a 14-bit wide parallel data path. The PDP-11/60 issues high level commands, and receives positional information, through a serial connection to the MCP. Finally, tactile data is passed to the 11/60 through a DMA link from the TSP.

One of the aforementioned high level commands would request the microprocessors to trace the cross-section of an object in any arbitrary plane in space, passing the sequence of 3-D coordinates back to the host computer. A great deal

of thought went into the implementation of this command, and it is, to some extent, responsible for the architecture described above. The procedure will be described in detail in a later section. It is a good example of how tactile sensory feedback can be used in a real time, closed loop fashion.

The finger was designed and fabricated at L.A.A.S., the major robotics establishment of the French government. Because of a severe lack of communication, many of the finger's details were not known to us when the software was being designed. This had a positive affect in that we were forced to be as general as possible. However, due to a number of unexpected delays, we still do not have the finger in our possession.

We arranged to borrow the pad sensor from Lord Corporation in Erie, Penna.* They traditionally deal with blending rubbers and bonding rubber to metal. This sensor, still in the prototype stage, is an attempt to expand their business.

At any rate, we had the pad sensor in our possession for three very long days. In preparation for that ordeal, we planned a number of different experiments. The Lord people were very helpful in this, and they provided us with the appropriate wooden test objects.

---

* Lord has since moved to Cary, South Carolina.

The characteristics of the pad sensor are very different than those of the finger. In particular, there is only one sensitive face. This makes the pad much less suited to contour tracing. We therefore decided to concentrate on some of the other aspects of tactile sensing -- dynamic texture analysis, static pattern recognition, and measurement of small angles between the object and sensor surfaces.

The ensuing sections will describe in detail the work performed.

Chapter 2: The Proposed Microprocessor Software

In anticipation of the arrival of the finger, a great deal of software was planned. Then, when the delays became apparent, work on those aspects not directly applicable to the pad sensor screeched to a halt. As a result, some of the design described here has not yet been implemented. In a later section we will discuss in detail exactly what the existing software does.

One of the important features of the Experimental Sensor Processor is its delegation of low level tasks to other processors. This helps to diminish the computational load on the host pdp-11/60. The tactile branch, in keeping with this principle, would have a set of commands which could be invoked by the host to perform various I/O and timing intensive operations, or functions involving real time feedback. Following are some of the commands that were considered:

1. Reset the machine.
2. Move to absolute coordinates (x, y, z), stop on collision with an object. This can be used as a "find something in this direction" command.

3. Scan Cross-section -- Trace the contour of an object in an arbitrary plane in 3-space. Returns to the host a list of step vectors describing the finger's path.

4. Local Texture -- Trace around a small circle on the surface of an object and produce a description of the texture. This could be in terms of degree of roughness, degree of compliance, or something as crude as a list of pressure values for each point in the path.

5. Search (in an as yet unspecified manner) for either a concave or a convex edge. It is assumed that the finger is already in contact with a surface.

6. Follow the contour of a concave or convex edge. Passes a list of step vectors to the host describing the finger's path.

The first command, Reset, is trivial. It simply involves the reinitialization of variables. The move command, due to its fundamental nature, has been implemented for use with the pad sensor. The cross-sectional scan command has received a great deal of attention, but has not been completely implemented because of its incompatibility with a single-face sensor. The final three commands, Local Texture, Find Edge, and Follow Edge, have to date received very little serious consideration. They are quite tentative, and may never be implemented.

2.1 Processors

As described in other sections of this thesis, the tactile branch consists of two microprocessors, the Tactile Sensing Processor (TSP), and the Motor Control Processor (MCP). A different program runs in the firmware of each

processor. Both are entirely interrupt driven using the Z-80 vectored interrupt system. From the host computer's point of view, the TSP provides data for texture analysis, and the MCP provides data for contour analysis.

2.1.1 Tactile Sensing Processor

The TSP program consists of a single loop in which each of the sensors is interrogated for its 8-bit pressure value. Each value is thrown into one of three categories with respect to a low and a high threshold. the category indicates whether the sensor is not touching anything, is in contact with an object, or is pressing the object too hard.* The sensors are then grouped by finger face, and a face status is computed for each face using the following rules:

> If any sensor is over range, the face is over range;
> If all sensors are below range, the face is below range;
> Otherwise, the face is within range.

If there were any face status changes since the last pass, the Motor Control Processor is informed.

It is worth noticing that this condensation algorithm is independent of the particular organization of the finger. The number of faces, the faces' orientations, and even the

---

* We hope that the sensors have enough compliance of their own so we can arrange the thresholds successfully. We would like to guarantee that for any movement toward an object, there is at least one position in which the leading sensor is "in contact" before it exceeds the upper threshold.

mapping of sensor number to face number are stored in tabular form, and may be altered according to the parameters of a different sensor. It will be obvious later that the more faces we have, the easier it is to keep in contact with an object. In the ideal case, we would like a hemispherical finger with many sensors, each on its own face. Such an organization can be accommodated just as well as the current finger.

In addition to providing this condensed status information for the sister processor, the TSP must send some data to the host, for the texture analysis. How much data does the host need? If we send it all we can -- 133 8-bit bytes per step, 125 steps per second -- we would need the equivalent of 20 9600 baud serial communication lines to handle the load! The bottleneck is removed by using a Direct Memory Access (DMA) interface. But even so, we cannot expect the PDP-11/60 to analyze data arriving at such an incredible rate, and still be able to keep up with the other sensory branches, and perform the higher level recognition tasks at the same time. It simply does not have the computational power.

The answer, of course, is to filter or condense the data before sending it. We have several possibilities in mind. First, a sensor is only considered valid if its pressure value is "within range". This filter is always in effect. Other possibilities include averaging sensor

readings over time and only reporting after a fixed number of steps, or combining somehow the readings from all sensors on each face which is "within range" to produce a single face pressure value. A final possibility is to arrive at some kind of measure of roughness for the surface under consideration, and only pass that number back to the host computer. This decision has not been made.

2.1.2 Motor Control Processor

The Motor Control Processor's basic job is to control and coordinate the three stepping motors which position the finger. When it is necessary that the host computer know the path that the finger follows during the execution of a command, the MCP provides it.

Steps are taken in a synchronous fashion. That is, if the step rate is set to 125 steps per second (the default case), the processor is interrupted every eight milliseconds to determine which motors are to be stepped, and in which direction.

So, after each interval, the MCP may pulse any combination of the three motors, and each can be in one of two directions. This leads to 26 possible directions in which a single step can move (ignoring the case where no step is taken at all). We represent this direction as a

6-bit "step vector", organized as follows:

bit 5	4	3	2	1	0
Z direction	Z step	Y direction	Y step	X direction	X step

Since this fits easily in an 8-bit byte, it is very convenient now for the MCP to give a path to the host computer. It simply sends a one-byte step vector over the serial line for each step taken. The host collects the sequence of step vectors in a buffer, and the exact path can be reconstructed very quickly at any time.

There are, of course, situations in which it is necessary to give an absolute coordinate. For example, when the absolute move command is aborted due to collision with an object, it is necessary to inform the host what the new position is. A mechanism is provided for this, too.

Notice that the MCP returns (effectively) a sequence of points. It does not try to fit them to curves, surface patches, generalized cylinders, etc. This is left to the host computer. It is unreasonable to expect an 8-bit microprocessor which lacks even a multiply instruction to do these in real time.

When moving from one position to another in 3-space, it is desirable to do so in a straight line. This requires varying the speeds of the individual motors so that they all arrive at their destinations simultaneously. The following

example shows how we would like to arrange the steps in a sample situation.

	A steps	B desired time between steps
X	17	9.88 milliseconds
Y	21	8.00 milliseconds
Z	5	33.6 milliseconds

The values in column B were arrived at by dividing the column A values into the greatest column A value, and multiplying the result by 8 millisecs. (8 millisecs is the speed at which we would like the fastest motor to operate).

This is a lot of work for an 8-bit microprocessor to perform. Also, if the precision of these calculations is not great enough, it becomes virtually impossible to predict exactly where the finger will be at any given point in time.

Fortunately, the synchronous stepping scheme makes matters much simpler. The overall line of motion is a line in 3-space. This is described and stored in terms of three direction components. There are also two accumulating counters, one for the mid direction, and one for the min direction. (The mid direction is the dimension which has the second-largest number of steps to take. Min direction is defined similarly.) Both are preset to zero.

After each 8-millisecond interval, a step vector is created, and the motors are stepped accordingly. The max direction is always stepped. For each of the other two

directions, the accumulating counter is incremented by the corresponding direction component value, and the result is taken modulo the max direction component. If an overflow occured, a step is taken.

Applying the algorithm to the above example results in the following sequence of steps.

Step	X	Y	Z	Step	X	Y	Z
1		*		11		*	
2	*	*		12	*	*	
3	*	*		13	*	*	*
4	*	*		14	*	*	
5	*	*	*	15	*	*	
6		*		16		*	
7	*	*		17	*	*	*
8	*	*		18	*	*	
9	*	*	*	19	*	*	
10	*	*		20	*	*	
				21	*	*	*

When a step is taken, two corollary actions occur. First, if the MCP is providing path information, the step vector is sent to the host. Second, a termination test is made. For the absolute move command, termination occurs when the finger reaches its destination.

This command also terminates if the Tactile Sensing Processor indicates that the finger has come in contact with an object. Primarily, this is to protect the finger from damage. However, it also makes it possible for the host to say, "look in this direction for an object." In that sense, this command can be used as an object finder.

2.2 Cross-Sectional Scan Command

This command is invoked by the host to trace the contour of an object's cross-section in any arbitrary plane in 3-space.* The arguments include the coefficients a, b and c in the equation of the plane $ax + by + cz = 0$, and a pair of special 3-D points which define the search volume. The finger must already be touching an object, and the plane is assumed to pass through the finger's current position.

Consider a conical object and a slicing plane parallel to the x-y plane. The MCP will drive the finger in the plane such that it remains in contact with the surface of the cone. All the while, it passes its path back to the host. Later, the host will analyze the path, and discover that it describes a circle.

The search volume is included to limit the finger's range of motion. Suppose, for example, the host wanted to construct a 3-D bicubic surface patch. It could do this by requesting four cross-sectional scans using vertical planes whose y-z projection is a rectangle. Then it could fit curves to each of the four point sequences, and perhaps fit a patch to these four curves.

---

* My terms will be very confusing unless I define them at the outset. "Plane" generally refers to the arbitrary cross-sectional plane given by the host. "Surface" is the (possibly curved) surface of the object. "Face" refers to one of the faces of the finger on which sensors are mounted. "Search volume" means the physical volume in which the finger is allowed to move.

Unless we provide some mechanism for limiting the search space, there is no way to prevent the finger from doing a complete scan of the object's cross-section, when only a small portion of that scan is needed.

The search volume is a rectangular parallelepiped with diagonally opposed corners defined by two arbitrary points in 3-space. The arbitrary points are chosen by the host computer and passed to the MCP as arguments to this command. Very often, the points may contain special coordinate values of 0 or 'max'. These may be used to effectively leave one or more dimensions completely unconstrained.

In the surface patch example, we would like to constrain the x and y position to the projection of the four slicing planes onto the x-y plane. The z position should not be constrained at all. Thus, the two arbitrary points might be (X1, Y1,0) and (X2, Y2, max).*

The scan will terminate when the finger either exceeds one of the bounds, or returns to its initial position. This second termination condition is useful if the host is interested in producing a contour map of the object. It could do this by requesting a series of scans, using cross-section planes parallel to the x-y plane, but at varying z values. In this case we would like the finger to

---

* In addition to this constraint, there is an implicit maximum search volume given by the dimensions of the device.

completely circumscribe the object, continuing until it returns to its starting point.

A problem which has not yet been mentioned is that of keeping in contact with the surface of an object. It turns out that in most situations, this is relatively simple. The method requires three kinds of information.

As described earlier, the finger has a number of distinct faces. The present structure of the positioning device does not allow for rotation or re-orientation of any kind. Hence, except for possible translation, these faces are fixed. Their equations, as well as those of the planes perpendicular to them, are predefined as constants in the MCP program.

Second, we have the equation of the cross-sectioning plane. All motion of the finger is to be restricted to that plane. By intersecting this plane with either the plane of a face of the plane perpendicular to a face, we can calculate a line of motion. This can then be fed to the absolute move routine to effect the movement.

Finally, there is the data from the Tactile Sensing Processor. This indicates whether each face is below range, within range, or above range. Typically, there will be only one face which is within range. This is labelled the "active face," because it is the one which is in contact with the surface. There are exceptions, and we will see

shortly how we can account for them.

The objective in keeping in contact with a surface is to keep the active face within range. Recalling that by definition of the command, the active face is initially within range, we have the following cases:

(1) Active face is within range;
(2) Active face is below range;
(3) Active face is above range; and
(4) A second face comes within or above range.

In case (1), the finger is in contact with the surface. Our best estimate of the shape of the object at this point is a plane parallel to the active face. Calculate the line of motion (if it has not been calculated already) as the intersection between the active face and the cross-sectioning plane. Send the current position to the host, and take a step.

In cases (2) and (3), the finger either has lost contact, or is pressing the surface too hard. Calculate a line of motion as the intersection between the cross-sectioning plane and the plane perpendicular to the active face. Then take a step along it away from or toward the finger's center, respectively. Do not send this step vector to the host, because it is not part of the surface contour.

Case (4) could result from several different situations. Take the scenario in which the finger hit a

concave corner. In this case, the appropriate action is to make the new face the active face, and then act according to its status.

Another scenario in which case (4) could occur involves reaching either a convex corner, or a point at which the surface curves away from the currently active face. Again, the appropriate action is to declare the new face as the active face, and act according to its status.

There are a number of other situations in which a second face could come within or above range. The appropriate action is not always the same as above. In fact, one could imagine situations in which a third and perhaps a fourth face must be considered. Though these cases have not yet been adequately resolved, we do not expect them to be overly troublesome.

Chapter 3: The Implemented Software

We noted earlier that although the software was designed for the finger, it was eventually implemented for the pad sensor. The most notable difference between design and implementation was the fact that in the end, we only used one microprocessor. All those commands which required multiple face sensing -- trace contour, follow edge, etc. -- were eliminated because the pad sensor in fact has only one face. It happened that these commands coincided with the ones which required real time feedback. Therefore, the requirements of the tactile data acquisition software became almost trivial, and could be handled easily and much more simply by the Motor Control Processor.

3.1 Environmental Details

The microprocessor software is written in Z80 assembly language. It resides on the PDP-11/60, which runs under the RSX-11M operating system. We use a primitive Z80 assembler, written in C, which produces Intel hex-format object code. This we download to the microprocessor via the 1200 baud serial line which connects the two systems. As it turned

out, 1200 baud was as fast as the 11/60 could reliably receive and store data.

The microprocessor system is made up of a California Computer Systems S-100 bus and mainframe, 8K of RAM, and a Cromemco Single Card Computer (SCC) with 1K RAM and room for 8K of PROM, 1K of which is taken up by a modified form of Cromemco's power-on monitor. The SCC has five timers, three parallel ports (input/output), and a serial port. Since the A/D converter built into the pad sensor produced CMOS output levels, we decided to temporarily add our own converter, a Cromemco D+7A board.

In the following sections we give a complete description of the software as it currently stands.

3.2 Command Format and Interpretation

The command language was to be a permanent part of the software. It would be used initially by a human user to control the pad sensor's movement and data acquisition. Eventually, however, it would become the Experimental Sensory Processor's way of driving its tactile branch.

Thus we had three goals in mind. First, the command language should be versatile. It should be able to handle the commands described in the previous chapter as well as the simple placement and data acquisition commands we needed for the pad sensor experiments. Second, it should be

concise enough, and easy enough to interpret, to be used for interprocessor communication. Finally, it had to be legible, so that the user could issue commands from his keyboard.

We settled on a syntax with mnemonic, single character commands, optionally preceded by an ascii-coded positive or negative integer which defaults to +1 if omitted, and optionally followed by any special arguments required by the command. The preceding integer is decoded by the parser. It generally refers to the multiplicity, though its interpretation is up to the individual command routines. The trailing arguments are parsed and interpreted completely by the individual command routines.

Commands may be strung together to form a command sequence. Execution will not begin until a carriage return is received. The sequence is, of course, stored in a buffer until execution is complete. A key advantage to this is that it makes loops possible. In the syntax, a subsequence may be grouped by parentheses, which in turn may optionally be preceded by a multiplicity M. The entire subsequence will be repeated M times. Subsequences may be nested to any reasonable depth.

There is one more rather important feature. While the command sequence is incomplete, the Motor Control Processor completely disables interrupts. Since the motors are driven by periodic timer interrupts, all movement must stop.

Similarly, characters coming from the serial line during command execution are ignored. This generally does not matter, because execution will have terminated before a new command sequence arrives. However, should it become necessary for the host computer (or user) to abort execution, it (he) may send an ESCape character. This causes a non-local subroutine return to the command sequence input routine, which immediately disables interrupts.

The following is a list of the commands currently available.

H	Home -- return to inner, upper left corner, and reset the current position to (0,0,0).
nX	Move n steps in the X direction (n may be positive or negative, and defaults to +1 if omitted).
nY	Move n steps in the Y direction.
nZ	Move n steps in the Z direction.
@x,y,z	Move to absolute position (x,y,z).
n(	Begin nest.
)	End nest.
=	Return current position as x,y,z coordinates, ascii-coded decimal values separated by commas.
Q	Quit the program -- return to power-on monitor.
1S	Take a snapshot of the sensor, store data in memory, increment frame count.
-1S	Take as many snapshots as possible until the completion of the current motor step.
0S	Clear the frame memory.
G	Send the contents of the frame memory to the host, beginning with the frame count. All data is in ascii-coded hexadecimal. Then clear the frame memory.
space	Null operation.

These commands are obviously very simple. However, they can be very powerful when grouped together. For example, the sequence

```
@100,100,100 50( 3( 20X 20Z S -20Z) 20Y -60X) G
```

takes 150 snapshots, in a 50 by 3 grid, beginning at (100,100,100), then sends all the collected data to the host computer. Since optical limit switches prevent the motors from moving past the ends of travel, one could find the maximum limits in all directions by issuing

@10000,10000,10000 =

(the actual range is roughly 1200 steps per axis). This would move the sensor to the corner opposite the home position and report the actual coordinates.

This list will eventually be enhanced to include the commands described in the previous chapter. We expect to be able to continue to denote each command with one mnemonic character.

3.3 Motor Control

It is not surprising that the most complicated task performed by the Motor Control Processor is, in fact, motor control. The complexity arises for two reasons. First, it is intended to be a permanent part of the MCP software, and is therefore very general in design. Second and most important, the step service routines effectively and completely insulate the higher level command execution processes from the hardware.

At the top level, an individual command routine uses

the step services in the following fashion:

```
Set the direction components in LINE
Call SCFILL to fill the step control table
Do until termination-condition:
    Call STEP to initiate a step when ready
    Call NEWPOS to update current position
    Call NEXTPO to prepare the next step
End
```

Note that it does not concern itself with timing in any way, nor does it have to take into account the physical limits of the device. The STEP routine guarantees a minimum pulse width (maximum step rate), and even modifies the step request if such an action would drive a motor past its end of travel.

Also note that the routine must actively request that a step be taken. If, for some reason, the evaluation of the termination condition is very time consuming, the motors will simply run slower. This has another advantage. Should the program be damaged by an unusually high incidence of cosmic rays, the motors will not go out of control. They will simply stop, because nothing is calling the STEP routine.

Before we take a closer look at these routines, we must discuss the data structures involved. The first one that was mentioned is LINE. It takes three numbers to define the direction of a line in 3-space: delta-x, delta-y, and delta-z. These are the line's direction components. Simply put, when we take delta-x steps in the x direction, we must

also take delta-y steps in the y direction, and delta-z steps in the z direction. Within the MCP, these values are stored and manipulated as unrestricted 16-bit integers. However, should it later become necessary to compare line directions, these may have to be restricted to relatively prime integers. LINE is a three word array which defines the desired path to the step routines.*

A commonly accepted canonical form for these values is a list of direction cosines. This requires that the values be real numbers, and that the sum of their squares equal unity. Fortunately, we have not found this form necessary.

The second data structure is the Step Control Table (SCTAB). This 15-byte table is basic to the operation of the step service routines. Following is a layout of its contents.

```
SCTAB+ 0: (byte)  Next port image
       1: (byte)  Port image skeleton (direction bits)
       2: (word)  Max direction component
       4: (word)  Mid direction component
       6: (word)  Min direction component
       8: (word)  Mid accumulating counter
      10: (word)  Min accumulating counter
      12: (byte)  Max direction's motor pulse and power bits
      13: (byte)  Mid direction's motor pulse and power bits
      14: (byte)  Min direction's motor pulse and power bits
```

Let us digress a moment before we explain SCTAB. Instructions are passed to the stepper motors via an 8-bit

---

* The Z80, of course, does not really have any distinct concept of a "word." However, being an old PDP-11 man, I always have and always will refer to a 2-byte quantity as a word.

output port, which looks like this:

bit 7	6	5	4	3	2	1	0
Z dir	Z step	Y-Z power	Y dir	Y step	X power	X dir	X step

The three direction bits indicate which direction the corresponding motor is to move. One implies the negative direction, zero implies the positive. The step bits, when pulsed, cause their corresponding motors to take a step in the indicated direction. Due to a low-pass filter which is applied to these bits for noise immunization purposes, there is a minimum pulse width. The MCP uses a separate timer for this, as will be described later.

Finally, the power bits, when on, cause drive power to be applied to the corresponding motors. For now, the reader need only understand that a motor must have power in order to operate.

Now we should be able to make sense out of the Step Control Table. The first item, the "next port image" is exactly that -- the 8-bit quantity that is to be sent by the STEP subroutine to the motor drive output port at the next opportunity. It is very important to note that this value is, in general, calculated concurrently with the previous step, by a call to NEXTPO.

The second item, the "port image skeleton," contains the three direction bits. These bits are applied with every

step. The SCFILL routine sets them according to the signs of the three direction components in LINE, and they do not change again until a new line is chosen.

The next three items, the Max, Mid and Min direction components, are actually the magnitudes of the numbers that appeared in the LINE array, but in sorted order. These are used in conjunction with the Mid and Min accumulating counters to determine which motors to step at the next timing interval.

Finally, the mapping from the sorted order to the x-y-z order is given by the last three items. Each of these bytes has exactly two bits set, corresponding to the appropriate motor's step and power bits.

The NEXTPO routine first decides which motors are to be stepped, and then adds together the corresponding mapping bytes, along with the direction bits from the skeleton. The resulting value is the next motor port image.

Let us now return to the high level control loop given at the beginning of this section. First of all, note that the values passed in the LINE array indicate a direction only. They do not completely describe a line segment in 3-space. It is assumed that the line of motion will begin at the current position, and the control loop is responsible for knowing when to stop.

Once the LINE table is set, SCFILL is called to fill

the Step Control Table. All values are calculated independent of the previous contents. The NEXTPO routine is then called automatically to use the new table to compute the first port image and place it in the zeroth location.

Since a step is never taken unless specifically requested by the control loop, it is perfectly reasonable to completely change direction at any time by simply changing LINE and calling SCFILL, before calling STEP again. One need not be concerned with the timing considerations.

Within the control loop itself, the first action is a call to the STEP routine. This routine waits, if necessary, for the previous step to complete. Then it calls CHECK to check the optical end-of-travel limit switches and, if necessary, modify the candidate port image. Finally, the routine outputs the image to the motor port and returns to the calling control loop.

Internally, one of the five on-board timers is also set to cause an interrupt after a time equal to half the minimum step pulse width has elapsed. The routine which handles that interrupt will clear the motor step bits and set the timer to interrupt again after another equal interval. At that point, an entire step has completed. The STEP routine, if it is waiting, is allowed to proceed with another step. In this way, something like an open ended square wave is generated on the motor pulse bits.

This brings us to the other subroutine calls in the main control loop. During the timing delays, the CPU is free to do quite a substantial amount of processing. Recall that the STEP routine has the power to modify the candidate port image. This modified image is returned to the control loop, where it is passed again to the NEWPOS routine. NEWPOS, based on the direction and step bits which were actually sent, updates the current coordinate counters.

The calculation of the next port image is then accomplished by a call to NEXTPO, which proceeds as follows.

1. Begin with the motor port skeleton, which defines the direction bits.
2. Add in the Max direction's pulse and power bits. That motor is to move at the maximum rate, and will therefore always take a step.
3. Add the Mid direction component to the Mid accumulating counter, and take the result modulo the Max direction component. If there was an overflow, we want to step the Mid motor. Add in its pulse and power bits.
4. Repeat step 3 for the Min direction.

The resulting value is placed in the first byte of the Step Control Table. An example of this algorithm in operation was given in chapter 2.

There is one final item to discuss. Conceptually, a stepper motor has a series of magnetic coils arranged in a circle around an iron core. As steps are taken, each coil in succession is energized, drawing the core around the circle. During normal operation, a given coil is only

energized for a brief period before its successor takes over. However, when the motor is standing still, one coil is energized continuously for a long period of time. It can generate quite a bit of heat -- enough, perhaps, to burn itself out.*

To solve this problem we implemented the following scheme. Every time a motor is stepped, its power is automatically turned on. At the same time, its corresponding usage counter is reset to some constant. Periodically, another of the on-board timers interrupts the processor to decrement all the usage counters. When any one reaches zero, the corresponding power bit is turned off.

The effect of this is to power down any motor that has not been stepped in the last two seconds. The action is so completely transparent to the higher level control software that we refer to it as the "burnout protection demon."

3.4 Tactile Data Acquisition

Due to its temporary status, the tactile data acquisition is perhaps the least important part of the software. As soon as the finger arrives, these routines will be removed from the Motor Control Processor and rewritten completely for the Tactile Sensing Processor,

---

* I don't know whether motors would actually burn out, but when I found I could fry eggs on them, I did not want to take chances.

according to the plans given in chapter 2. Therefore, as might be expected, the current code is far from general. It is entirely driven by the S and G commands described earlier. Nothing happens asynchronously.

The entire unused portion of the MCP's memory board is used as a buffer for tactile data. Upon MCP initialization, the frame count is reset to zero. Then, each time a snapshot is requested, the data record is placed in the next position in the buffer, and the frame count is incremented. When the readout is requested (via the G command), the program simply types it all out, one line per record, beginning with a line consisting solely of the frame count. The information is transmitted in ascii coded hexadecimal, as an optimization of both transmission time and coding time.

Chapter 4: Experiments and Results

In this chapter we will discuss the experiments which were actually performed using the pad sensor. We will consider the methods, the goals, the problems, and the results. When possible and appropriate, we will refer to figures which illustrate the results.

4.1 Calibration

The pad sensor consists of an 8 x 8 array of sensitive sites whose analog output values are fed into an analog multiplexer, and finally into an analog to digital converter. All this circuitry is part of the sensing device. Unfortunately, since the A/D converter emits CMOS voltage levels, and our parallel ports use TTL inputs, we had to bypass the internal A/D and use our own. This resolved the incompatibility, but gave vent to another problem. The pressure signals coming out of the multiplexer ranged roughly from +2.0 to +2.5 volts, and our A/D converter expected a range of -2.5 to +2.5. As a result, the digital pressure readings never went below about 235, out of a maximum 255.

In other words, the fact that we can exhibit only a

little over four bits of precision is not a reflection on the device, but on the interface. With the right interface, we would estimate upwards of six bits of valid data.

Each of the 64 pressure sensitive sites puts out a slightly different range of voltage levels. They therefore required individual calibration. The most straightforward way of doing this is to press the sensor down hard on a flat surface, take a snapshot, release the sensor entirely, and take another snapshot. This yields a matrix of minimum and maximum pressure values, to which all subsequent data would be scaled in a linear transformation.

Of course, nothing is ever so simple. Each pressure sensitive site requires roughly 1.3 pounds of pressure to completely depress it. Multiplying that by 64 sites, we find that we need over 80 pounds of pressure to acquire the maximum readings. Our Z-axis motor is not capable of this.

The solution was to depress each site individually, and then combine the data into a single matrix of maximum pressure values. Fortunately, the Motor Control Processor's command language was flexible and powerful enough to do this painlessly in one command sequence, with two loops for X and Y positioning.

Once the minima and maxima were obtained, it was a simple matter to map all input data into a uniform range of 0 - 255. It is worth mentioning here that throughout the

entire testing period, these ranges never changed more than one unit. In addition, we never had any problem with spurious data being generated where there was no contact. Those points always mapped to zero. We were quite impressed with the robustness of the pad sensor.

4.2 Static Tactile Image Analysis

4.2.1 Single Image

The obvious first step in analyzing tactile images is to lay the sensor down on a known object, take a snapshot, and see whether it is recognizable. This we did, and the results are depicted in fig. 1.

In fig. 1f we used a one inch square, set off-center, but oriented orthogonally with the sensor's grid axes. There is no question as to the identity of that object. A simple threshold operation would clearly distinguish it from the background.

Fig. 1e and fig. 1d show the same square rotated counterclockwise 30 degrees and 45 degrees, respectively. Fig. 1c shows an equilateral triangle, point downward, and fig. 1b depicts the same triangle rotated clockwise about 75 degrees. Notice how some pixels are much lighter than others in the images with non-orthogonal edges. This phenomenon arises when the object covers less than half the area of a site. Since the site is conical in shape, the

edge must be pressing on the wall of the cone. It cannot depress the cone as far as it could if it were pressing on the apex.

In theory, it should be possible in some cases to determine exactly how much of the cone is actually covered by the object. However, we must assume the following: 1) that the object surface, particularly the edge in question, is smooth, 2) that the object surface is in a plane parallel to that of the pad sensor, 3) that the individual sites on the sensor are in fact conical, with bases that meet the bases of their neighbors, and 4) that we know how to calculate the actual depression as a function of output pressure value.

Unfortunately, neither of the last two assumptions are valid in our case. The cones are actually cut off before they reach the apex,* and we do not have the data to perform the depression calculation.

Finally, fig. 1a shows a one inch diameter circle. Notice that it appears to be identical to the square in fig. 1c. This is a question of resolution. Clearly, if the spatial resolution were doubled or quadrupled, the distinction would be obvious.

---

* My office-mate tells me that the technical term for this shape is "frustum."

4.2.2 Spatial Resolution

How do variations in sensor resolution effect the image? The simplest way to tackle this question is to vary the size of the features on the test objects. We used a set of disks with raised concentric circles projecting from them in relief. The variations consisted of two amplitudes and three frequencies, totalling six disks.

Fig. 2 shows the images obtained. As might be expected, those disks in which the spacing between the circles approach the spacing between the sensitive sites (figs. 2a and 2d) are clear. As the frequency increases, the shape becomes less obvious, until it is completely unintelligible at the highest frequency.

The effect of amplitude is also fairly predictable. At low amplitude, the circles are wider, and therefore more sites are in contact with the surface. This can be seen most clearly (again) in figs. 2a and 2d. Also, the inner circle is more distinct in fig. 2e than in fig. 2b. This is because at the lower amplitude, the depth of a trough is considerably less than the height of a conical site, and therefore some trough sites come in contact with the surface.

Theoretically, it should be possible to compare pressure values and determine where the troughs and crests occur. However, here we run into the limitation in our 3-D

positioning device which we alluded to in the Calibration section. The Z-axis motor, which supplies the normal force, is a bit too weak for this pad sensor. Each sensitive site requires a certain amount of force to depress it, and the motor must be able to exert the sum of these forces in order to obtain a reliable reading. Therefore, as more sites contact the surface, each one receives less pressure. Furthermore, if the surface is not uniform, neither are the reductions in pressure.

4.2.3 Multiple Images

How can we improve the spatial resolution with the equipment available to us? One simple way to double the number of data points on each dimension is to take a reading at each of the four corners of a small square, whose sides are half the length of the distance between sites. This we did, using the same six disks, and the results are visible in fig. 3.

The images are slightly clearer, but not as much as we had hoped. Again, the disappointment is indirectly caused by the deficient Z-axis motor. When taking a snapshot, we try to depress the sensitive cones as much as possible, since we are not capable of depressing any of them completely. To do this, we simply instruct the Motor Control Processor to lower the Z-axis motor until it won't go any further.

This works quite well in general. However, consider the following hypothetical case. Suppose the test object is a single sine wave and the sensor is a single cone. First, we lower the cone onto the crest of the wave as far as it will go, and take a snapshot. Then we move the cone to the trough and repeat the operation. The two images look identical! In both cases, the cone was depressed as far as it would go, and it is in fact the cone depression which determines the image. This, we believe, is the root of the multiple image problem.*

The solution, of course, is to strengthen the Z-axis motor. Then, instead of simply lowering the sensor until it stops, we would lower it to a consistent Z-coordinate. The resulting set of images would be much clearer.

4.2.4 Large Objects

Can we examine objects which are much larger than the sensor? For this experiment we used a flat surface about 12 inches long and three inches wide -- slightly wider than the sensor pad itself. A set of eight grooves were cut into this surface in order to form a pattern of diverging lines (see fig. 4a). By taking a series of snapshots at successive lengthwise positions, we should be able to reconstruct the entire image, in spite of the fact that it

---

* Or, "Aye, there's the rub!"

is much longer than the sensor.

The Motor Control Processor's command language again made this a simple task. We took fifty images, stepping about five millimeters between each. The reconstruction, shown in fig. 4b, was accomplished by superimposing the images in the appropriate positions relative to each other. As before, when the distance between features approaches the distance between sensitive sites, the pattern becomes clearer.

Can we use our multiple image trick to improve the resolution? We repeated the same procedure, except that this time we took three snapshots, four millimeters apart widthwise, for each of the fifty steps lengthwise. The reconstruction, fig. 4c, shows the angled edges much more clearly at lower frequencies than does fig. 4b. At higher frequencies, however, both reconstructions are equally unintelligible. Once again, we blame the failure on the Z-axis motor, and our method of maximizing pressure.

4.2.5 Small Angle Measurement

When a robot hand grasps an object, does it have a good grip? Very often, a "good grip" is one in which the flat surfaces of the object are wholly in contact with the flat faces of the fingers. The question can then be answered very simply by measuring, for each finger, the angle between these two planes.

This experiment proved to be extremely successful. Using the one inch square as our test object, we took four snapshots. In the first image we layed the pad sensor flat on the square, as usual, giving us a zero degree standard. For the three subsequent images, we lowered the left end of the table by 1.0, 1.25, and 1.5 inches respectively, producing angles of 3.3, 4.1, and 4.9 degrees.

The results are shown in table 1. For each image we arrived at a single number describing the slant. The number was calculated simply by averaging all the pressure differences between horizontally adjacent sites. In theory the ratio of the third slant value to the second should be 1.25,* and the fourth to the the second should be 1.5. This was not the case.

However, the first image, whose slant should have been zero, did exhibit a small slant value. If we take this as an error, we can produce a correction factor by dividing it by the slant value for the second frame. When that percentage is subtracted from each of the two ratios arrived at earlier, we get remarkable results. The corrected ratios differ from the expected values by less than two percent!

---

* Proof is obvious from the geometry, as long as we assume a linear relationship between depression distance and output value.

4.3 Dynamic Texture Analysis

We believe that until tactile sensors can be fabricated with extremely fine resolution, information about the texture of a surface would best be obtained by moving the sensor along the surface, and examining the changes in pressure readings, as opposed to the pressure readings themselves.

Toward this end, we tried several times to make the positioning device drag the pad sensor along different surfaces, but failed each time. The sensitive cones, because they were designed to grasp an object without allowing it to slip, were made out of high friction rubber. This, of course, directly hindered the experiment. The stepper motors were not powerful enough to pull the sensor and still maintain enough contact pressure to yield a significant reading.

In the end we performed a singularly unscientific experiment. We dismounted the pad sensor from the positioning device and dragged it by hand along a flat wooden surface, taking 100 snapshots over a period of about five seconds. This may not have been so bad, except that we neglected to measure the exact distance traversed, or anything that could directly or indirectly give us the velocity.

The analysis is interesting, though quite inconclusive.

The sensor is made up of an 8 by 8 grid of sensitive cones. Let us define a column as the series of cones lined up in the X-direction, and a row as the cones lined up in the Y-direction. Given that the sensor was dragged in the positive X-direction, we contend that there should be some aspect of the data which is consistent down a column, but different across a row. Furthermore, there should be a small but constant time delay between the features exhibited by one site and those exhibited by the next site down the column.

The motivation for this hypothesis is as follows. Picture a textured surface as a terrain of bumps and ridges. As the sensor grid passes over this terrain, the cones across a row will collect entirely unrelated data. However, those down a column will encounter the exact same bumps and ridges that were encountered by their predecessors, but a little bit later. Thus we have eight instances of eight-fold redundant data. We should be able to find some consistency somewhere.

Initially, we plotted the raw pressure data from each of the 64 cones as a function of time. Fig. 5 is a reproduction of this, with each plot placed in the same grid position as the corresponding cone. We expect to be able to look down a column and see some consistency that does not occur across a row. Unfortunately, no such consistencies were immediately obvious.

The next step was to try to home in on the changes in pressure, as opposed to the pressures themselves. However, a simple pairwise difference derivative (see fig. 6) was no more enlightening than the raw data.

Well, what about the Fourier transform? Surely the frequency domain is closer to our goal than the time domain. Unfortunately, applying this transform meant giving up our time delay information, which we needed for comparing curves.

What we really needed was some smooth measure of frequency as a function of time. A colleague* suggested the following procedure. First, take the pairwise difference derivative. Then, pass a window along the time axis. For each point in time, count the number of zero crossings in the window, and divide by the width of the window. A window n units wide would have a maximum of n zero crossings, and thus the ratio would be unity. No crossings would produce a ratio of zero. Note that the operator is valid, and produces the same range of values, independent of the window size. The only difference is in the precision.

We used a window with an odd number of points, so it could be symmetric about the point under consideration. If the distance to one margin or the other was smaller than half the window size, the window was shrunk accordingly, so

---

* Thank you, Gerry Radack.

that symmetry was maintained. We tried various window sizes in order to obtain the smoothest curve possible without losing too many features. The optimal size was about 25 units (out of 100), shown in fig. 7a. A 15 unit window is shown in fig. 7b for comparison.

There are (finally) some definitely visible similarities among the resultant curves of fig. 7a. Examine, for example, the troughs in rows 6, 7 and 8 of column 1. Notice how similar they are, and how a small, constant time delay occurs between each curve and its successor. The same phenomenon is visible in rows 1, 3, 5 and 6 of the third column, and in rows 1 and 3 of column 7.

As one looks up and down a column, there seems to be some kind of topological similarity. This is exactly what we want to find. However, identifying it mathematically is no simple task. The obvious operator to apply would be the cross correlation. This compares two graphs and produces a number describing the closeness of the match, then shifts one graph relative to the other and repeats the calculation. One correlation value is generated for each possible shift. The resulting curve shows not only how well the two graphs match, but at what time delay value the match is optimal.

Unfortunately, the results were very disappointing. No matter which pair of graphs we compared, the cross correlation never went substantially higher than zero, and the best match always occured at zero shift. Needless to

say, at least one more level of processing is called for.

4.4 Conclusions

First, it is clear that an 8 by 8 grid of pressure sensitive sites is generally not enough for pattern recognition of single static images. In most real applications, either the objects will be larger than the pad, or the features will be below the pad's resolution.

With reasonably good positioning equipment, the resolution can be significantly improved, and the size of the area under consideration considerably increased, by taking multiple images. However, this is often too time consuming, and therefore infeasible.

The straightforward solution is to increase the spatial resolution, the number of sites, or both. We have shown that when feature dimensions are comparable to resolution, shape recognition can be quite simple. This has also been demonstrated by Hillis [HILLIS-81], using a sensor recently developed at the MIT A.I. Laboratory, and of course by Briot [BRIOT-79], who used an array of binary sensors. One typical application for this might be the table sensor which was described in the introduction.

A more novel approach might be to build multijointed fingers for the robot gripper, such as the three fingered hand developed by Ken Salisbury [SALISBURY-81] at the

Stanford A.I. Laboratory. This would enable the robot to manipulate the object while transporting it, in such a way that it becomes not only feasible, but a matter of course to take multiple tactile images.

In the experiment concerning measurement of small angles, we obtained impressive results. The computed values were even more accurate than we had hoped. From this we conclude that a tactile sensor with properties similar to those of the pad sensor is eminently suited to applications involving small angle measurement, such as grip improvement.

As far as texture analysis is concerned, we believe our approach is a good one. Visually, it is apparent that we are on the right track. However, the experiment must be repeated in a much more controlled fashion, and different surfaces must be examined and compared. Then, we hope we will eventually be able to manipulate the data in such a way that we can use it to identify the surface.

Chapter 5: Further Work

As was mentioned earlier, the pad sensor was in our possession for only a short time, by no means long enough for exhaustive experimentation. In fact, many of the more interesting ideas occured to us after the sensor was returned, when we began to analyze the data.

It should be possible to calculate the coefficient of friction between various surfaces and the rubber face of the sensor. First, one must know the force as a function of digital output for each sensitive site, as well as for the strain gauges on the metal posts. Then, one would drag the sensor along the surface in question, and take force measurements. The normal force N is simply the sum of the forces on all the sites, and the frictional force F is derived from the horizontal forces given by the strain gauges. By plugging these numbers into the equation $F = uN$ one can calculate u, the coefficient of friction. This might be usable as a distinguishing characteristic between surfaces.

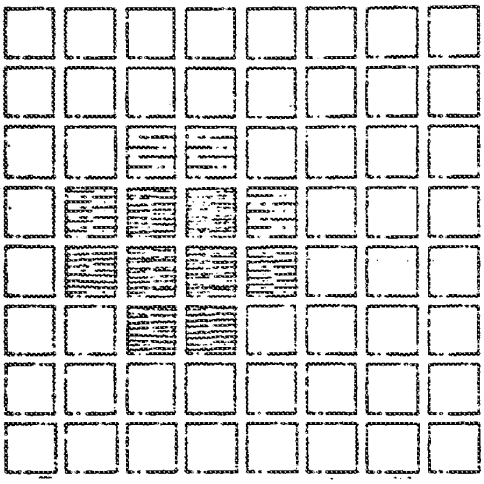
It might also be useful to measure granularity. This could be done simply by placing the sensor onto the surface

and counting the number of sensitive sites which exhibit significant pressure. Of course, the grains in the test surfaces must be comparable in size to the resolution of the sensor.

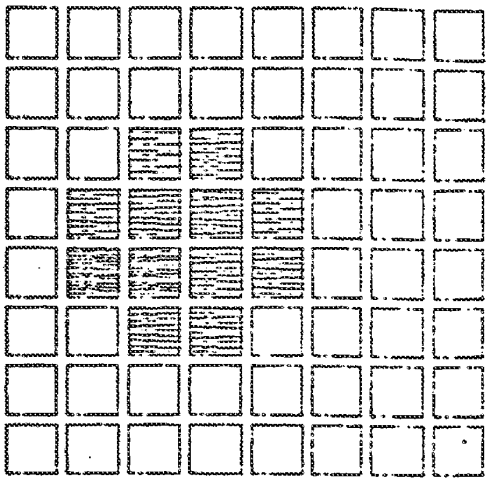
Certainly the dynamic texture analysis tests should be repeated and extended. Once that data has been hashed out, it should be possible to identify surfaces based on pressure response to friction.

Finally, there are two aspects of tactile sensing which we have not experimented with because they are better suited to the finger than the pad sensor. First, the finger should be capable of poking a surface and comparing predicted pressure with actual pressure in order to measure of surface resilience. Second, there is the whole question of tracing cross sections and producing, essentially, a 3-D description of the contour of an object.

Thus we have shape based on both static images and contour descriptions, granularity, coefficient of friction, and surface resilience and texture. These features, when they are better understood, should be incorporated as distinguishing characteristics into the Experimental Sensory Processor.

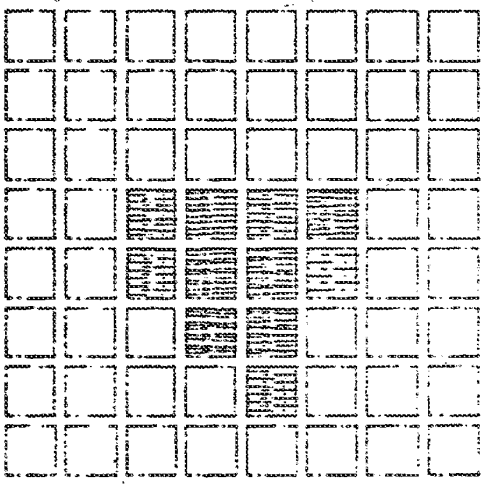


a) 1" diameter circle



d) 1" square rotated 45°

b) 1.5" triangle



e) 1" square rotated 30°

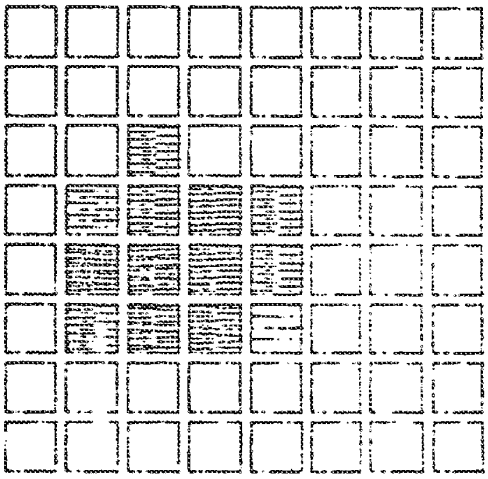
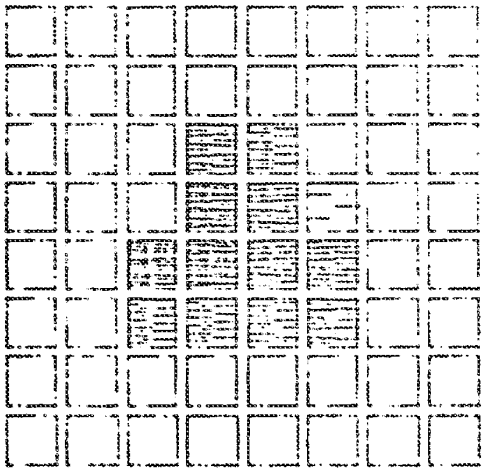
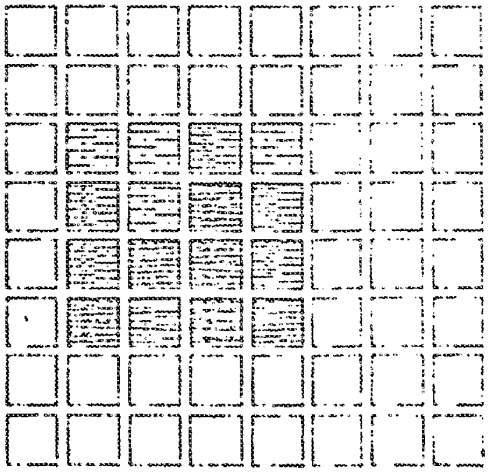


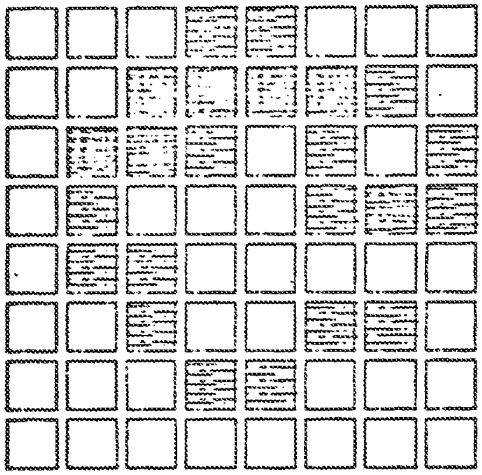
Fig. 1. Single Image Shape Recognition

c) 1.5" triangle rotated 75°

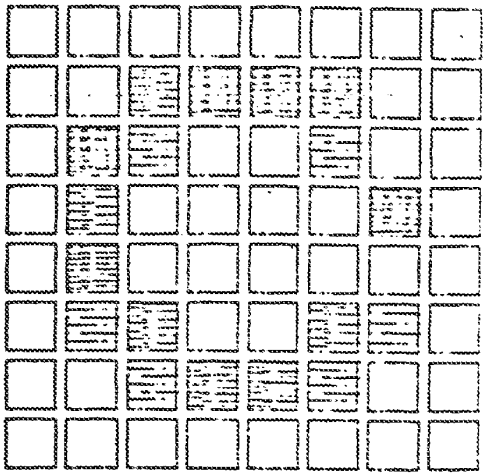


f) 1" square



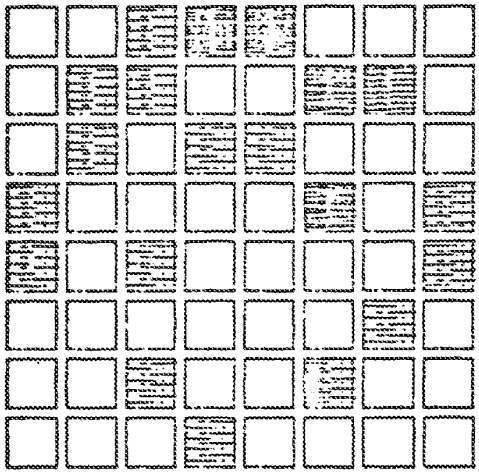


a) One circle, low amplitude

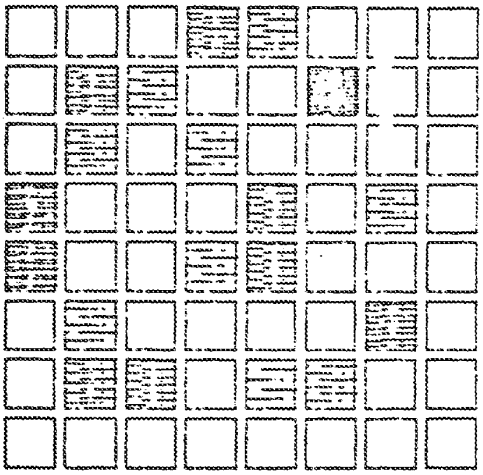


d) One circle, high amplitude

Fig. 2. Single Image Recognition Varying Frequency and Amplitude

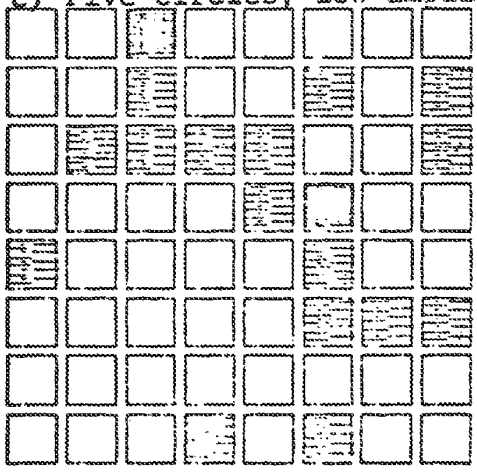


b) Two circles, low amplitude

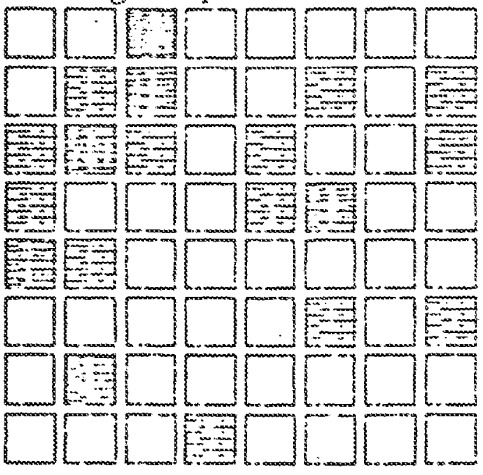


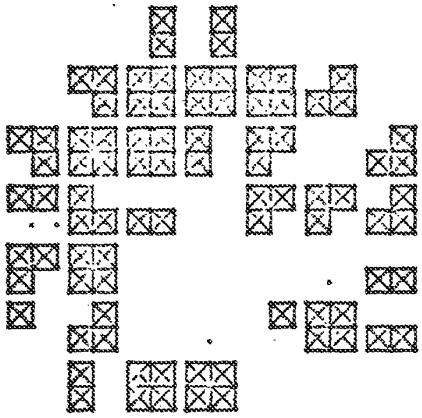
e) Two circles, high amplitude

c) Five circles, low amplitude

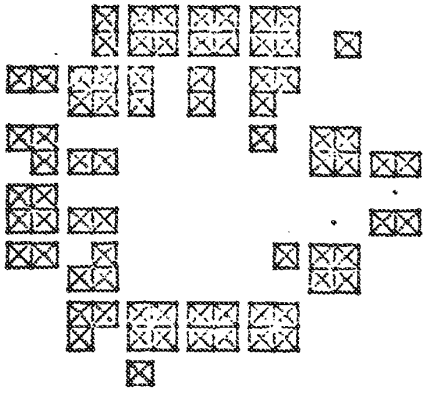


f) Five circles, high amplitude



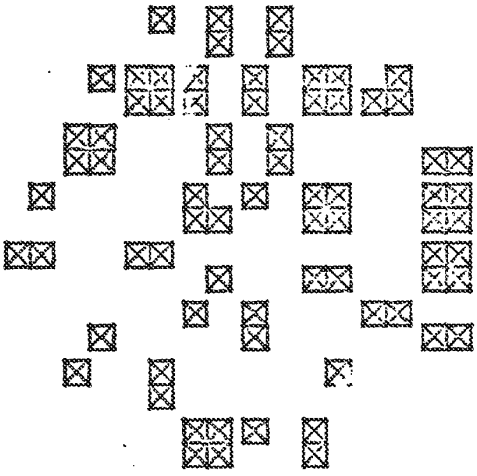


a) One circle, low amplitude

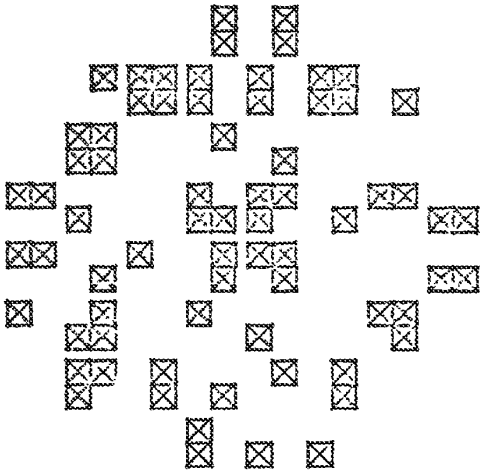


d) One circle, high amplitude

Fig. 3. Multiple Image Recognition
Varying Frequency and Amplitude

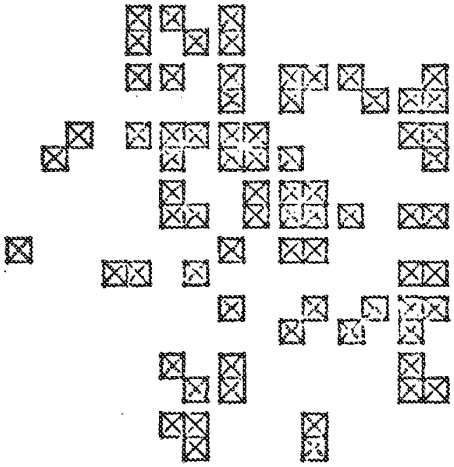


b) Two circles, low amplitude

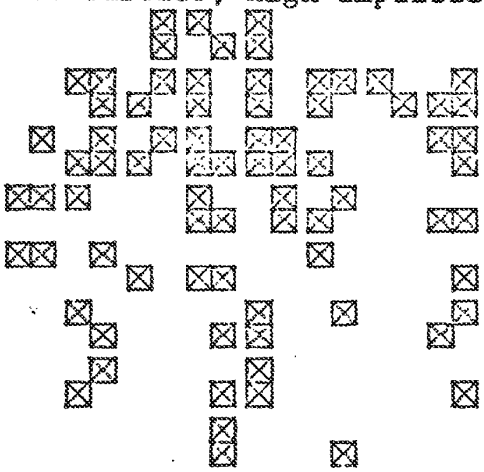


e) Two circles, high amplitude

c) Five circles, low amplitude



f) Five circles, high amplitude



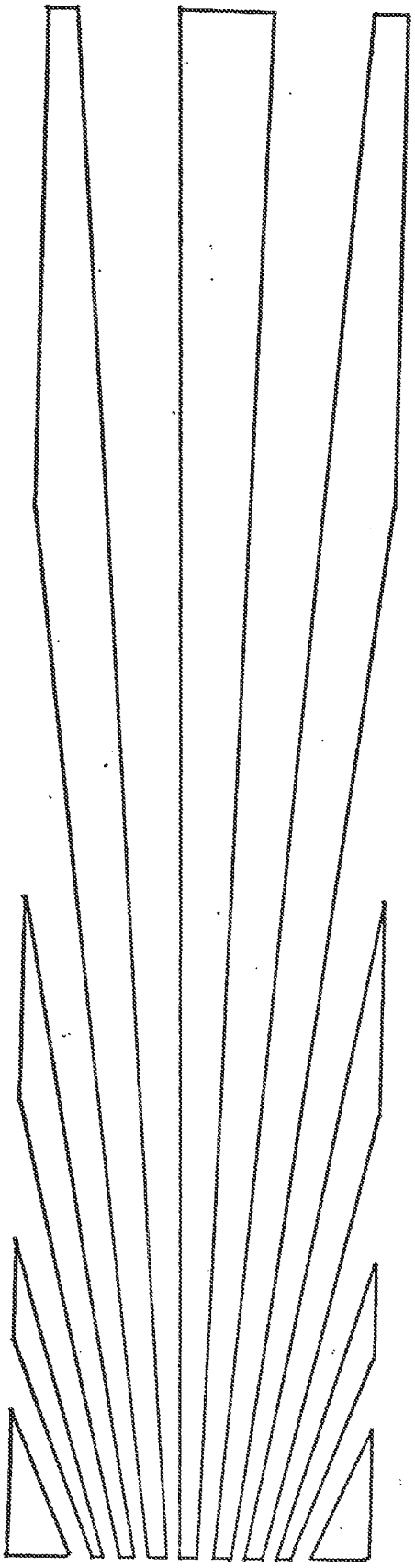


Fig. 4a.
Drawing of the
Large Test Object

Fig. 4b
Using 50 successive frames

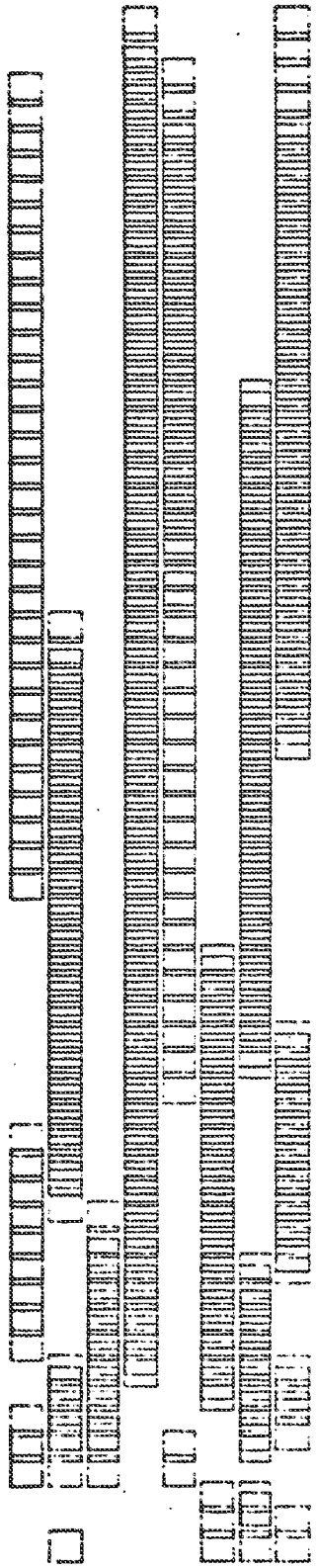


Fig. 4c
Using 3 x 50 matrix of frames

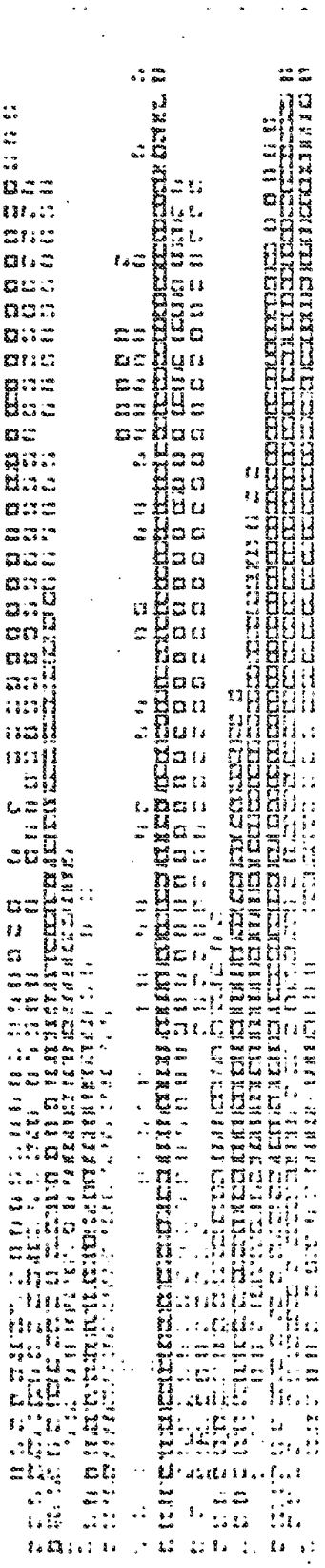


TABLE 1 -- Measurement of Small Angles

Table Slant	Data	Horiz. Difference	Avg. Diff.	Ratio*
0"	45 64 80	19 16	12.625	
	42 48 64	6 16		
	48 60 75	12 15		
	34 56 51	22 -5		
1"	15 64 160	49 96	78	1.00
	28 80 192	52 112		
	16 75 195	59 120		
	17 85 153	68 68		
1.25"	48 160	112	114	1.23
	48 192	144		
	60 180	120		
	56 136	80		
1.5"	48 160	112	142	1.53
	48 240	192		
	60 225	165		
	71 170	99		

* Ratio is calculated as the vertical average divided by the vertical average at 1" slant, multiplied by one minus the ratio of the 1" slant to the 0" slant. The closer this value is to the table slant, the better the results. As the reader can see, the results are exceedingly good.

Fig. 5. Raw pressure data as a function of time. Each plot is in the same grid position as the corresponding cone. Sensor was dragged toward +X, with Row 1 in the lead.

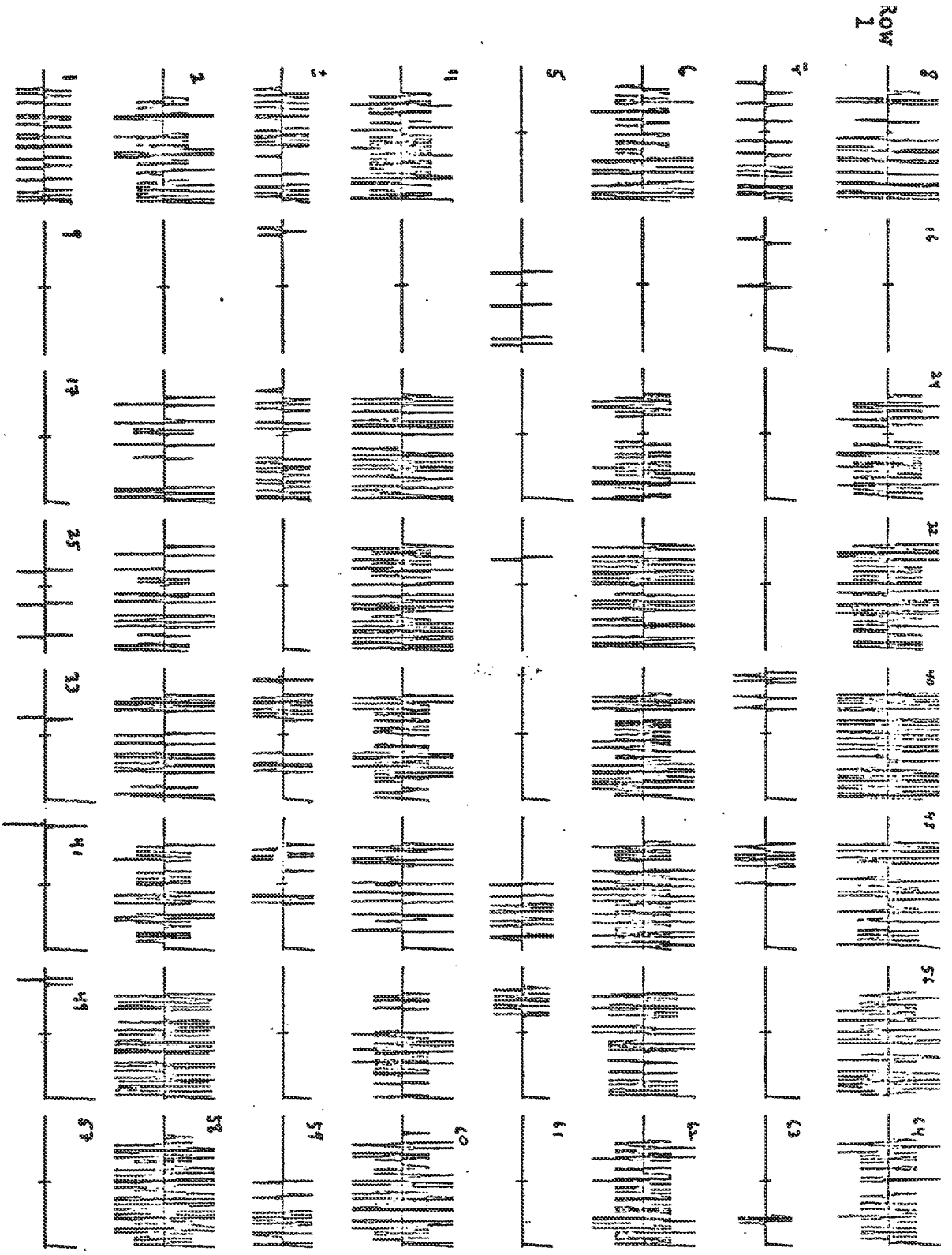


Fig. 6. Pairwise Difference Derivative

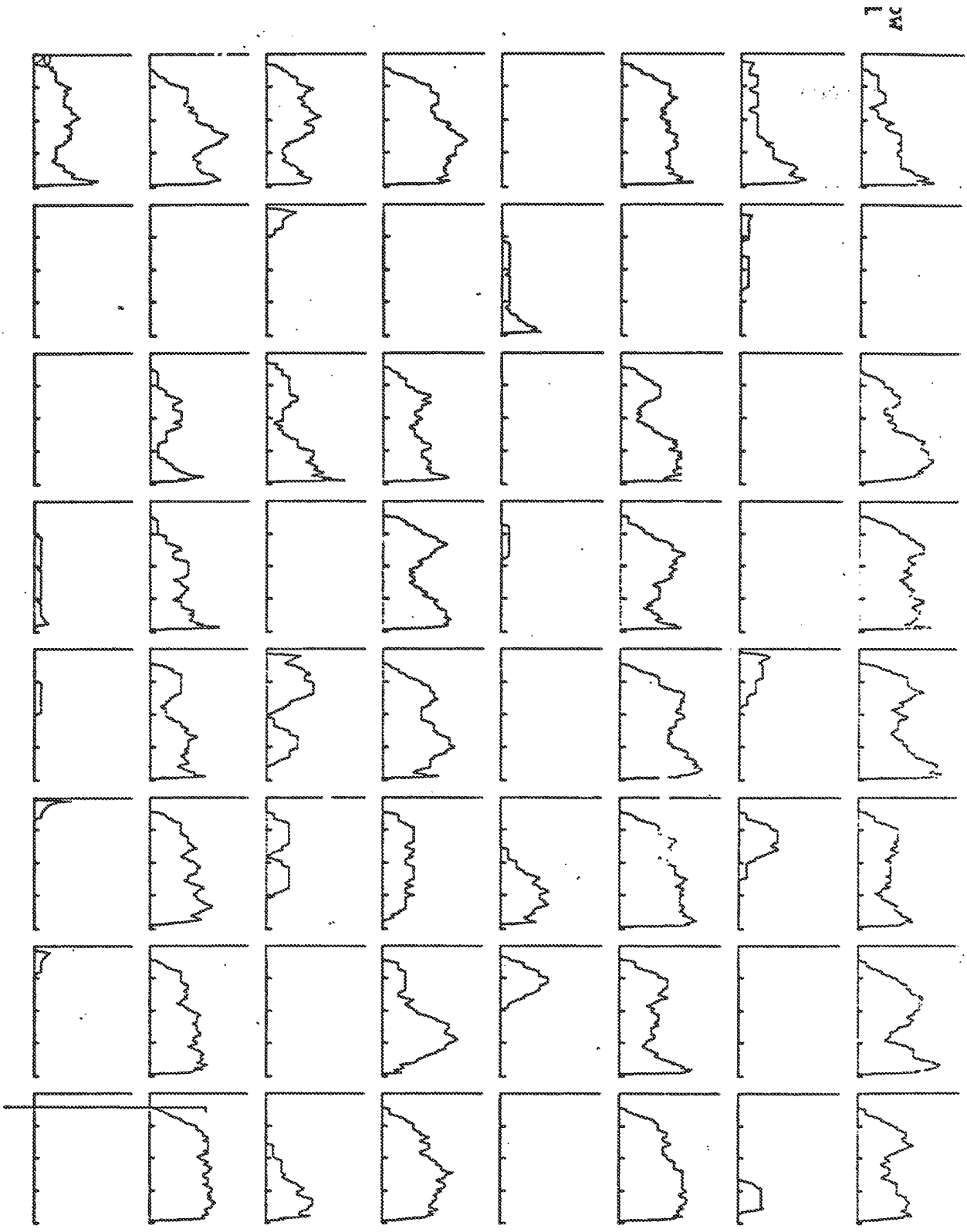


Fig. 7a. Frequency as a Function of Time
Window = 25 Units

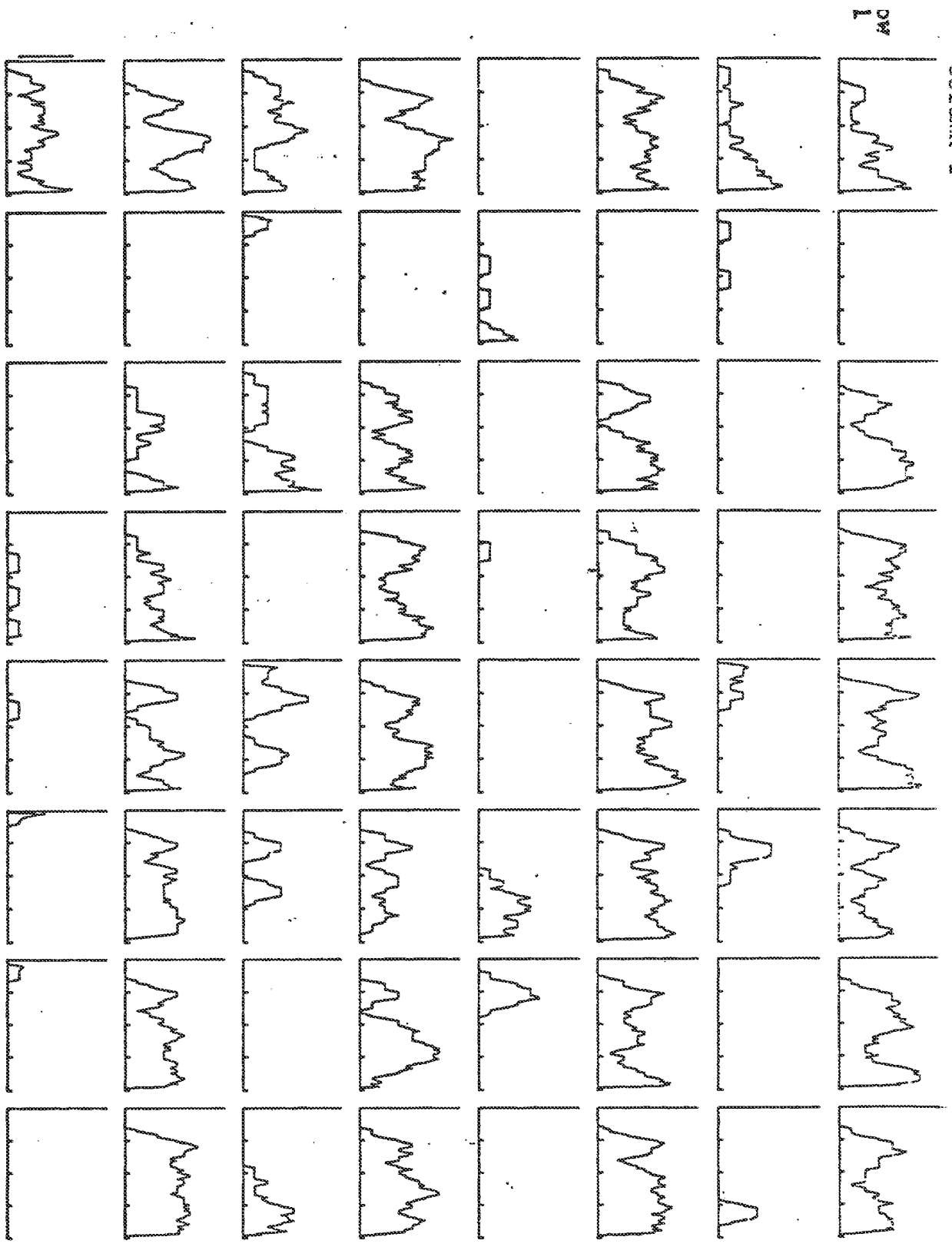


Fig. 7b. Frequency as a Function of Time
Window = 15 Units

60a

References

[BOYKIN-80] Boykin, W. H., and Diaz, Gary., "The Application of Robotic Sensors -- a Survey and Assessment," ASME Century 2 Conference, August 12-15, 1980.

[BRIOT-79] Briot, Maurice, "Utilization of an 'Artificial Skin' Sensor for the Identification of Solid Objects," Proc. of 9th International Symposium on Industrial Robots, Washington, D.C., March 12-15, 1979.

[BROWN-80] Brown, David J., "Computer Architecture for Object Recognition and Sensing," Master's Thesis, Department of Computer and Information Science, University of Pennsylvania, December, 1980.

[DANE-81] Dane, Clayton, Forthcoming PhD. Dissertation, Department of Computer and Information Science, University of Pennsylvania, 1981.

[HILL-73] Hill, John W., and Sword, Antony J., "Touch Sensors and Control," in Remotely Manned Systems -- Exploration and Operation in Space, ed. by Ewald Heer, California Institute of Technology Press, Pasadena, California, 1973.

[HILLIS-81] Hillis, William Daniel, "Active Touch Sensing," Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 629, April, 1981.

[IVANCEVIC-74] Ivancevic, Nebojsa S., "Stereometric Pattern Recognition by Artificial Touch," Pattern Recognition, Vol. 6 pp. 77-83, 1974.

[KINOSHITA-75] Kinoshita, Gen-ichiro, "A Pattern Classification by Dynamic Tactile Sense Info. Processing," Pattern Recognition, Vol. 7 pp. 243-251, 1975.

[NITZAN-80] Nitzan, David, "Assessment of Robotic Sensors," Workshop on the Research Needed to Advance the State of Knowledge In Robotics, April 15-17 1980.

[OKADA-77] Okada, T., and Tsuchiya, S., "Object Recognition by Grasping," Pattern Recognition, Vol. 9 pp. 111-119, 1977.

[PURBRICK-81] Purbrick, John A., "A Force Transducer Employing Conductive Silicone Rubber," Proc. 1st International Conference on Robot Vision and Sensory Controls, Stratford-upon-Avon, UK., IFS (Publications) Ltd., April 1-3, 1981.

[SALISBURY-81] Salisbury, Ken, Stanford Artificial Intelligence Laboratory, Personal Communication, May, 1981, and, Proc. of 1981 Joint Automatic Control Conference, Charlotsville, Virginia, June, 1981.

REAPING THE BENEFITS OF MODERN USABILITY EVALUATION: THE SIMON STORY

James R. Lewis
IBM Human Factors Group
P. O. Box 1328
Boca Raton, FL 33429-1328
Tel: +1 (407) 443-1066
Fax: +1 (407) 443-2778
E-mail: JIMLEWIS@VNET.IBM.COM

product usability, usability evaluation methods; personal communicators

Simon (TM-BellSouth Corp.) is a commercially available personal communicator (PC), combining features of a PDA (personal digital assistant) with a full suite of communications features. This paper describes the involvement of human factors engineering in the development of Simon, and summarizes the various approaches to usability evaluation employed during its development. Simon has received a considerable amount of praise from the industry and won several industry awards, with recognition both for its innovative engineering and its usability.

INTRODUCTION

The Simon is a cellular telephone, designed with a 36 x 115 mm touch screen (CGA resolution) replacing the standard telephone key area. Research in the usability of cellular telephones (Tsoi, 1993) has shown that many of the problems people have using cellular telephones are the result of inflexible control labeling and limited feedback. Replacing the standard key/display area with a touch screen allowed the Simon developers to create a simpler user interface for cellular telephone tasks. It also allowed the development of a suite of applications in addition to the cellular telephone, including an appointment calendar, an address book, a to-do list, a world clock, a note pad, a sketch pad, sending and receiving electronic mail, sending and receiving faxes, reception of pages, file management, a calculator, access to system settings, and security.

My first contact with the Simon development group came as a request to answer an apparently simple question: How small can a touch screen button be, and still be usable? Fortunately, I had just completed a literature review covering the results of human factors studies of touch screens from 1980 to 1992 (Lewis, 1992), so I was able to convey to Simon development that the answer to this simple question was actually somewhat complex and depended on the touch selection strategy (Sears and Schneiderman, 1989). From this start, I spent the next two years as a part of the Simon team, conducting studies and providing usability guidance. The approaches to usability engineering and assessment applied during Simon development illustrate the broad spectrum of modern usability methods, and the resulting product demonstrates the effectiveness of these modern methods. The descriptions appear in rough order of occurrence, but the activities overlapped considerably.

APPROACHES TO USABILITY ENGINEERING AND ASSESSMENT IN THE DEVELOPMENT OF SIMON

Focus Groups

After preliminary design work, an independent agency conducted several focus groups with different types of cellular telephone and computer users to help define the appropriate goals for the product.

Daily Design Meetings

Before writing any significant amount of code, the software team (including a human factors engineer and graphic designer) worked out more specific details about how to achieve the design goals. These meetings lasted for several hours every morning over a period of several months. After each meeting, the individual designers worked on their assignments, which typically involved detailed functional and task analyses. During the meetings, the designers presented their analyses and the rest of the team proposed scenarios for testing the task flows. Determination of problems with task flows in these meetings led to additional refinement of task analyses, which led to refinement of design concepts.

Literature Reviews

Literature reviews of human factors studies of touch screens (Lewis, 1992) and cellular telephone usability provided early, valuable guidance to Simon development. It is often tempting to skip the tedium inherent in a literature review, but keep in mind that it would be foolish to spend three months in the laboratory to obtain information available with an investment of three hours in a library.

Expert Evaluations of Competitive Products

Using an approach similar to Nielsen's (1992) heuristic evaluations, I conducted several expert evaluations of competitive products, both defining the sequence of steps required to perform key tasks and making note of probable problem areas. These evaluations revealed opportunities for improved design in such diverse areas as battery installation and removal, display contrast adjustment, key definition as a function of mode, setting calendar alarms, effective setting and removal of repeating meetings, and clear procedures for setting passwords and locking units.

Development of Test Scenarios

Considering the focus groups, daily design meetings, and expert evaluations of competitive products, the team developed an initial set of 38 test scenarios. By the end of iterative testing, there were 54 scenarios. As suggested by Lewis, Henry, and Mack (1990), some scenarios focused on tasks within a single application, while others evaluated work that crossed application boundaries. We used the scenarios for both gathering competitive performance and satisfaction benchmarks and for iterative problem discovery studies with development-level versions of Simon.

Competitive Usability Benchmarking

One application of the test scenarios was the determination of competitive usability benchmarks for both user performance (scenario completion times and success rates) and satisfaction. We used the After-Scenario Questionnaire (ASQ) to assess user satisfaction following each scenario, and the Post-Study System Usability Questionnaire (PSSUQ) to assess more global usability satisfaction following the completion of all scenarios (Lewis, 1995a). Figure 1 shows the PSSUQ benchmarks established during the competitive usability benchmarking. We collected data from three products regarded as the most likely competitors of Simon. Analysis of the problems discovered during these evaluations provided additional opportunities for improved design in Simon.

Iterative Usability Studies

We conducted three fairly extensive problem discovery studies at different stages during Simon development (early 1992 prototype, first design with reasonably comprehensive function, and the design immediately preceding the final design). Our philosophy for these studies was that measurement of scenario performance and preference variables were important, but that problem discovery was more important. As long as you have competitive benchmarks, scenario measurements give you an idea about where you are relative to your competition, but provide no real guidance about what to do when your product fails to measure up. Analysis of usability problems, on the other hand, provides strong guidance for product redesign. We used the methods described in Lewis (1994b) to determine appropriate sample sizes for these studies. As a consequence of this process of iterative problem identification and design improvement, each iteration showed significant improvement in both user performance and satisfaction. Figure 1 shows the PSSUQ scale ratings for the final iteration (showing means and 95% confidence intervals), with the competitive PSSUQ benchmarks for reference. (A lower PSSUQ score is better than a higher one.) As Figure 1 shows, Simon significantly exceeded its benchmarks for all PSSUQ scales.

Icon Assessment

Most icons that appear on Simon include a descriptive label. There are four icons, however, that appear on every Simon screen. Because these icons appear on every screen, we had a design goal to provide small icons that did not require labels (conserving valuable screen space). We assessed these icons using a battery of icon assessment methods including a matching and confidence task, icon production task, and a semantic differential (Lewis, 1988; Lin, 1992). The outcome of the study indicated a problem with recognition of the icon representing access to the non-phone office tools, and led to re-representation of the function with a focus on its access to a mobile office.

Language Guidelines and Automated Readability Measures

An often neglected area of usability design and evaluation is that of language. Even modern, otherwise usable, systems often contain complicated terms for which there are much more common names. On-line messages and other documentation contain numerous sentences in the passive voice that it would be easy to recast in active voice. These considerations might seem trivial, except that psycholinguistic research has shown that (1) frequency of occurrence of a word in a language significantly affects the speed of human lexical access (Forster, 1990) and (2) it is harder to extract meaning from a passive sentence relative to its active counterpart (Bailey, 1989). To promote clarity and consistency in terminology, I provided the Simon developers with a set of language guidelines, and iteratively reviewed messages and documentation against the guidelines. Our source book for determining the best word to use when considering several synonyms was The Living Word Vocabulary (Dale and O'Rourke, 1981). I also selected random text samples from competitors' documents and developed competitive readability benchmarks for text cloudiness (a measure based on the number of specifically identified abstract words and passivized verbs in a passage divided by the number of words in the passage). At the end of Simon development, measurements taken from a random sample of texts from Simon's documentation showed that the Simon texts had a significantly lower (lower is better) text cloudiness than any of its competitors. Furthermore, using data collected during competitive usability benchmarking and iterative usability studies, Simon had a significantly better PSSUQ Information Quality rating (Lewis, 1995b) than any of its competitors.

Statistical Modeling

Because Simon had a relatively small display area, it was necessary to provide some simple statistical modeling for the size of calendar entries (Lewis, 1993a) and name lengths (Lewis, 1993b) to provide guidance to the calendar and address book developers. The calendar entry research indicated that: (1) managers use computer calendars more than non-managers, (2) managers have more entries per day than

calendar-using nonmanagers; and (3) for user-generated entries, the 95th percentile for the number of characters in an entry was 253. The name length research showed that the mean name length in the United States was about 14 characters, and that a touch-screen button that could show 20 characters would show a person's complete name 99.2% of the time (in the United States).

Designed Experiments

On occasion, it was necessary to conduct designed experiments to answer questions that arose during development. One such experiment (Lewis, 1994a) explored different screen designs for setting dates and times. Although such settings seem straightforward, users have conflicting direction stereotypes that appear to preclude the use of arrows alone for setting times and dates. Two other experiments (Lewis, Allard, and Hudson, 1994; Lewis, 1995a) evaluated different aspects of Simon's predictive keyboard. A predictive keyboard is an on-screen keyboard that contains fewer buttons than a standard keyboard, and uses linguistic probabilities to predict which letters a user will most likely want to type next. These most-likely letters appear in the keyboard's buttons. Lewis, Allard, and Hudson (1994) studied the effects of different word populations, number of displayed letters, and number of trigraph tables on the likelihood that the desired next letter would appear on the predictive keyboard. Lewis (1995a) studied input rates and user preference for the three Simon data input methods (tapping on a small on-screen standard keyboard, tapping on the predictive keyboard, and handwriting on the sketch pad). The results showed that the most effective and preferred input method was tapping on the standard keyboard. In conducting these experiments, the experimental designs described in Lewis (1993c) were quite useful.

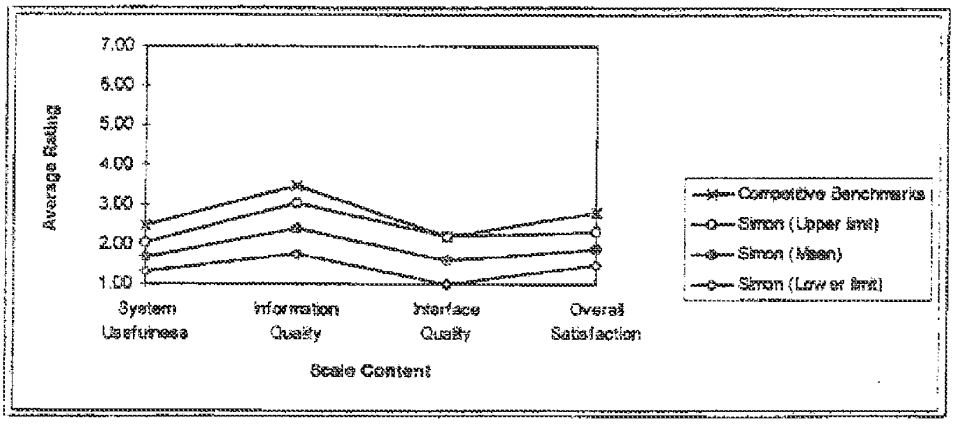


Figure 1. PSSUQ scale scores for Simon and competition

INDUSTRY RECOGNITION

One indication of the success of Simon's design is that it won the Best of Show award at Comdex '93, won an Award of Distinction in the 1994 BYTE awards (BYTE, January 1995), and was a Grand Award winner in the 7th Annual Best of What's New awards (Popular Science, December, 1994). The following quotations from reviews of Simon in trade journals also reflect the success of the usability effort.

"It looks and feels like a product you already know how to use, rather than a new religion you must immerse yourself in." (O'Malley, 1994)

"I hope that Simon is the first in a long series of personal communications tools, but even as a first generation product, Simon is a joy to use." (Nelson, 1995)

"Simon is not the first personal communicator product I've demoed, but it is by far the most comprehensive, well-designed, and easiest to use." (Carter-Lome, 1994)

DISCUSSION

This paper has described the broad range of usability evaluation methods applied to the development of Simon. The industry recognition for Simon stands as evidence for the success of the application of modern usability evaluation methods in this case. The breadth of methods also suggests that professional usability practitioners need to be fluent with a wide array of usability techniques because different development situations demand the application of different usability methods. Some of these methods come from traditional experimental psychology (statistical modeling, designed experiments, literature reviews), and others are more recent techniques (heuristic evaluations, competitive usability benchmarking, scenario-based usability problem discovery studies). All of these techniques have potential application in product development, and deserve a place in the toolbox of the professional usability practitioner.

ACKNOWLEDGMENTS

Successful product development is a team effort, and a human factors engineer is one member of a team. I gratefully acknowledge the contributions of Simon management, software development and hardware development to the final usability of Simon. Also, the efforts of Suvit Nopachai, who served with us as a graduate intern in human factors during Simon development, were invaluable.

REFERENCES

BAILEY, R. W. (1989). Human performance engineering: Using human factors/ergonomics to achieve computer system usability. (Prentice Hall, Englewood Cliffs, NJ).

BYTE. (January 1995). 1994 BYTE awards. BYTE, vol. 20, 49-60.

CARTER-LOME, M. (1994). A Simon for our times. CM: Cellular Marketing, 6.

DALE, E., and O'ROURKE, J. (1981). The living word vocabulary. (World Book, Chicago).

FORSTER, K. (1990). Lexical processing. In Osherson, D. N., and Lasnik, H. (Eds.), Language (pp. 95-131). (MIT Press, Cambridge, MA).

LEWIS, J. R. (1988). A review of symbol test methodologies (Tech. Report 54.475). (IBM Corp., Boca Raton, FL).

LEWIS, J. R. (1992). Literature review of touch-screen research from 1980 to 1992 (Tech. Report 54.694). (IBM Corp., Boca Raton, FL).

LEWIS, J. R. (1993a). Calendar entry statistics for computer calendar users (Tech. Report 54.754). (IBM Corp., Boca Raton, FL)

LEWIS, J. R. (1993b). Name length statistics for touch-screen buttons (Tech. Report 54.810). (IBM Corp., Boca Raton, FL).

LEWIS, J. R. (1993c). Pairs of Latin squares that produce digram-balanced Greco-Latin designs: A BASIC program. Behavior Research Methods, Instruments, and Computers, vol. 25, 414-415.

LEWIS, J. R. (1994a). Direction stereotypes for setting dates and times (Tech. Report 54.867). (IBM Corp., Boca Raton, FL).

LEWIS, J. R. (1994b). Sample sizes for usability studies: Additional considerations. Human Factors, vol. 36, 368-378.

LEWIS, J. R. (1995a). IBM computer usability satisfaction questionnaires: Psychometric evaluation and instructions for use. International Journal of Human-Computer Interaction, vol. 7, 57-78.

LEWIS, J. R. (1995b). Input rates and user preference for three small-screen input methods: Standard keyboard, predictive keyboard and handwriting (Tech. Report 54.889). (IBM Corp., Boca Raton, FL).

LEWIS, J. R., ALLARD, D. J., and HUDSON, H. D. (1994). Predictive keyboard design study: Effects of different word populations, number of displayed letters, and number of trigraph tables (Tech. Report 54.846). (IBM Corp., Boca Raton, FL).

LEWIS, J. R., HENRY, S. C., and MACK, R. L. (1990). Integrated office software benchmarks: A case study. In Human-Computer Interaction -- INTERACT '90 (pp. 337-343). (Elsevier, London, England).

LIN, R. (1992). An application of the semantic differential to icon design. In Proceedings of the Human Factors Society 36th Annual Meeting (pp. 336-340). (Human Factors Society, Santa Monica, CA).

NELSON, M. W. (March/April 1995). The Simon personal communicator. PDA Developer, vol. 3.2, 13-16.

NIELSEN, J. (1992). Finding usability problems through heuristic evaluation. In Proceedings of the Association for Computing Machinery CHI '92 Conference (pp. 373-380). (ACM, Monterey, CA).

O'MALLEY, C. (December, 1994). Simonizing the PDA. BYTE, 145-147.

POPULAR SCIENCE. (December 1994). Best of what's new: The year's 100 greatest achievements in science & technology. Popular Science, 50-76.

SEARS, A., and SCHNEIDERMAN, B. (1989). High precision touchscreens: Design strategies and comparisons with a mouse (Tech. Report CS-TR-2268). (University of Maryland, College Park, MD).

TSOI, K. C. (April 1993). User interface issues for cellular phones. Cellular Business, vol. 10, 32-43.

Soft Machines: A Philosophy of User-Computer Interface Design

Lloyd H. Nakatani
Bell Laboratories, Murray Hill, New Jersey 07974

John A. Rohrlich
Bell Laboratories, Whippany, New Jersey 07981

ABSTRACT

Machines and computer systems differ in many characteristics that have important consequences for the user. Machines are special-purpose, have forms suggestive of their functions, are operated with controls in obvious one-to-one correspondence with their actions, and the consequences of the actions on visible objects are immediately and readily apparent. By contrast, computer systems are general-purpose, have inscrutable form, are operated symbolically via a keyboard with no obvious correspondence between keys and actions, and typically operate on invisible objects with consequences that are not immediately or readily apparent. The characteristics possessed by machines, but typically absent in computer systems, aid learning, use and transfer among machines. But "hard," physical machines have limitations: they are inflexible, and their complexity can overwhelm us. We have built in our laboratory "soft machine" interfaces for computer systems to capitalize on the good characteristics of machines and overcome their limitations. A soft machine is implemented using the synergistic combination of real-time computer graphics to display "soft controls," and a touch screen to make soft controls operable like conventional hard controls.

INTRODUCTION

The juxtaposition of the terms "soft" and "machine" connotes the essence of a philosophy for the design of user-computer interfaces to interactive computer systems. "Machine" connotes an interface which is machine-like in appearance and operation. Such interfaces, we believe, can make computer systems as obvious, easy and efficient to use as well-designed conventional machines. "Soft" connotes a machine realized through computer generated images of controls on a high resolution color display with a touch-sensitive screen for actuating the controls. This software realization gives us the flexibility and power to overcome the limitations of conventional machines.

From our experience in building prototypes of soft machines in our laboratory, we have become aware of principles underlying the design and use of machines. We hope here to make some of these principles explicit, and to indicate how soft machines based on these principles can lead to better user-computer interfaces. We conclude with thoughts on how a collection of soft machines might be organized.

MACHINES AND COMPUTERS

We are struck by how easy most conventional machines—stoves, tape recorders and calculators—are to use, and how troublesome computer systems are to use by comparison. Machines and computers seem to contrast most sharply on the following aspects of their use[1]:

- *Learning* — We can learn how to use many machines by "playing around" and seeing what happens; learning is usually casual and easy. By contrast, we learn computer systems by reading instruction manuals and seeking help; learning is deliberate and often effortful. The recent flowering of human factors is reflective of this fact.
- *Transfer* — Having mastered a machine, say a copier, we can usually switch to another copier in a matter of minutes. Transfer between machines is generally so easy that we take it for granted and are surprised when it is hard. Having mastered a computer system, say a text editor, we find it relatively hard to learn another text editor. Transfer between computer systems can be so troublesome that ads for word processing personnel specify brands of equipment.
- *Efficiency* — Machines have specialized controls optimized for efficient operation; multi-purpose computers have unspecialized keyboards. We are

1. What follows are broad generalizations. Exceptions and counterexamples can be found, but we feel that the generalizations capture important differences between machines and computer systems which help explain why specialized computer systems are usually more machine-like in design and operation than general-purpose computer systems, and why microprocessor-based consumer products retain their machine-like character.

seeing, however, that as computer systems become more specialized, they acquire specialized, machine-like controls optimized for the functions they perform. For example, dedicated word processors have special function keys, and home computers used for games have joysticks or trackballs that are superior for pointing (Card, English & Burr, 1978; Albert, 1982). And, we observe that dumb machines that acquire microprocessor brains continue to be operated like machines rather than computers. These trends suggest that typical user interfaces to computer systems represent a step backward in interface design compared to the control panel of machines.

For ease of learning, transfer of knowledge and efficiency of operation well-designed hard machines seem better than computers. True, any single machine is not asked to do the wide variety of tasks that we perform with computers, but the superior usability of machines for their intended task is attributable to some intrinsic characteristics of machines that can exploited even for computers intended for multiple functions. What are these intrinsic characteristics?

HARD MACHINES AND HARD CONTROLS

By "hard" machines and controls, we mean conventional machines such as stoves, radios and copiers operated with knobs, switches, keys, pushbuttons and other familiar controls. Hard machines have many characteristics that make for ease of learning, efficiency of operation and ease of transfer, but they are ultimately limited by their "hardness."

Modularity

The modularity of hard machines, most typically mechanical machines, is a natural consequence of design constrained by size, complexity and cost. Modularity is obvious in the kitchen where different machines perform different functions: a stove for cooking, a mixer for mixing, and so on. Modularity keeps complexity within manageable limits, and also provides a "big picture" for organizing the bits of knowledge in learning and using a machine.

Form Follows Function

In machine design, form follows function; in its use, insight follows form. Form encompasses the overall shape of the machine, the control panel, and the individual controls with their labels and markings. Scrutiny of the form leads to conjectures about what the machine does and how it is operated. The conjectures are tested by operating the controls and observing what happens. By "playing around," we discover the what and how of the machine.

One-to-One Mapping of Controls and Operations

The success of "playing around" depends critically on the mapping between the controls on a machine's panel and operations or actions that the machine performs. Ideally, the mapping is one-to-one; that is, corresponding to each machine operation is a control which causes the operation to happen. Then if a machine has N controls, we know that the machine is capable of N and only N operations. This limits the possible conjectures to a reasonable number, and testing each conjecture is a trivial matter of actuating a control and observing what happens. Contrast this with the case where two controls have to be actuated in sequence to get each machine operation to happen. Now there are N×N possible things the machine could do, and N×N possible control sequences. We are unlikely to discover such a machine by playing around because the possibilities are too numerous and testing too tedious. Tape recorders turn this fact into a safety feature by requiring two controls to be actuated in sequence to make a recording. The improbability of discovering the proper sequence makes accidental erasure unlikely by a naive user.

Manual Operation

Machines are operated manually rather than symbolically. *Manual operations* conform to a universal language based on physical laws that govern the interactions between physical objects. Knowledge of this language enables us to cope with novel situations and tasks, usually without instruction or training. For example, if we want to toast bread, from the nature of the bread, toaster and the toasting process, it should be discoverable—if not immediately, then eventually after some trial-and-error—a procedure that will accomplish the task. By contrast, *symbolic operations* require languages which by definition are human inventions. The existence of English and Chinese, FORTRAN and Pascal, and different command languages makes clear that there is no universal language for symbolic operations. The multitude of languages and their arbitrariness is bound to render us illiterate and helpless when faced with a computer that speaks a language we do not know. Suppose, for example, that the toaster was operated by an unknown command language. We are unlikely to discover by trial-and-error how to operate such a toaster.

Immediate Feedback

It goes almost without saying that being able to observe immediately the consequences of our actions is important for evaluating whether our conjectures were correct or not, and for stimulating further conjectures.

The Language of Controls

Over centuries of machine design, a subtle language of controls (Chapanis, 1972) has evolved that we learn from our experience with machines. Designers of machines can use this language to tell us what the machine does and how to use it. Of course, the existence of this language does not guarantee good design, but we believe that a design which does not speak this language is likely to be bad. Some of the important messages in this language follow:

- *Presence* — The presence and absence of controls tell us what the machine can and cannot do. For example, the presence of controls labeled "LIGHTER" and "DARKER" on a copier tell us that we can make copies lighter or darker than the original.
- *Labels* — Good labels, whether text or symbols, tell us what the controls do and thereby what the machine as a whole can do.

- *Type* — The type of control suggests the nature of the thing controlled. For example, a toggle switch controls something with only two states, and a knob controls something that varies continuously.
- *Clustering* — Distinct clusters of controls often correspond to the distinct subfunctions of a machine. A copier may have, for example, a cluster of controls to specify the number of copies, and another to start and stop copying.
- *Arrangement* — The proper arrangement of controls can make labels superfluous. In a car, for example, a rectangular arrangement of power window switches on the center console makes obvious without labels the correspondence between switches and windows.
- *Movement* — Controls operate according to well-established conventions. For example, we flip a light switch up to turn the lights on, and turn the volume knob clockwise to make the music louder.
- *Status* — The settings of the controls can tell us what the machine is doing and what state it is in. On a toaster, for example, the position of the lever tells us that bread is toasting.
- *Graphics* — Graphic cues such as a frame around a group of controls or lines connecting controls can indicate the relationship between controls. On a control panel for a model train layout, for example, a line connects switches controlling points on a common section of track.

Limitations of Hard Controls

The physical and mechanical properties of hard controls make them nice to use. They can be felt and operated without looking, their distinctive movements provide kinesthetic feedback, and their sounds confirm their actuation. Unfortunately, the "hardness" of hard controls is also the source of many limitations.

- *Inflexibility* — The inflexibility of hard controls is the root of other limitations. Hard controls can't appear or disappear, move around, or change their appearance. Inflexible hard controls make for inflexible machines. We are now in an awkward situation where the functionality of machines is easily changed by software, but the inflexibility of hard controls severely limits the changes that can be accommodated without changing the hardware or compromising the operability of the machine. For example, if a keyboard does not provide special cursor positioning keys, we have to make do with controls intended for other uses; most likely, cursor positioning will be more awkward as a consequence.
- *Management of complexity* — Some machines are already too complex for many people, and the use of microprocessors which allow the easy addition of "bells and whistles" will lead to more complexity. The complex electronic calculator compared to the simple mechanical adding machine is an example of this trend. With hard controls it is difficult to keep the complexity from overwhelming us because the progressive disclosure of controls is difficult to achieve. Some machines, television sets for example, hide infrequently used controls behind panels to simplify their appearance and use. The problem of too many controls is aggravated by the compactness made possible by microelectronics. There may be no room on compact machines for controls which are large enough and spaced widely enough to be easily operable. Digital watches indicate the problem. The inflexibility of hard controls limit the complexity that can be easily managed with machines to far below their potential promised by microelectronics.

SOFT MACHINES AND SOFT CONTROLS

Definition and Antecedents

A soft machine can have practically all the advantages of hard machines without the disadvantages that accrue from hardware implementations. A soft machine is implemented by software which simulates hard machines in two important respects. First, a soft machine is made to *look* like a hard machine by graphics software that generates images of controls such as keys, pushbutton switches, and slide potentiometers on the screen of a color video display. The screen serves as a tabula rasa upon which computer systems are visually represented as soft machines through images of their control panels. Second, a soft machine is made to *operate* like a hard machine by covering the display screen with a touch-sensitive position sensor, or touch screen for short. The touch screen enables us to touch and operate the controls in the display as if they were physical controls. And we are not limited to pointing. We can, for example, drag our finger to activate "slide" switches, and forthcoming force-discriminating touch screens will make possible soft controls regulated by pressure. This mode of direct operation of controls by touch rather than through some intermediary pointing device such as a light pen or mouse gives soft machine users a sense of immediacy they would otherwise not have.

The basic ideas underlying soft machines were first articulated by Ken Knowlton (1977) who explored how the inflexibility of hard controls could be overcome partially by optically superimposing computer-generated labels on hard keys. Keys were made to disappear visually and logically by eliminating labels and voiding their operations. Computer graphics and color were used to indicate the clusters of related keys and their proper sequencing.

The first commercial realization of a soft machine to our knowledge is the XEROX 5700 Electronic Printing System (Schuyten, 1980). All the controls for the 5700 appear on a black-and-white video display with a touch screen for operating the controls.

More recently, Schmandt (1981) described a soft machine for editing speech recordings. Like us, Schmandt used color graphics and a touch screen to implement his soft machine. Mirrer (1982) developed in our laboratory a similar but more elaborate soft machine for making hybrid speech documents consisting of a speech recording and associated text outline. We have also developed soft machines for displaying colored speech spectrograms, and

for spreadsheet analysis.

A Calculated Example

An example should make clearer how a soft machine retains the attributes of hard machines that lead to ease in learning and transfer while taking advantage of the power and flexibility of computers to manage complexity.

Our example will be a calculator. The forerunner of the calculator is the adding machine, a hard machine with one purpose and a form suggestive of that purpose. A simple adding machine has few keys, and a complex one has many more. There is a one-to-one correspondence between keys and functions. The close resemblance of an electronic calculator to the adding machine enables us to use a calculator for simple calculations with a bit of exploration and without reading a manual.

In design, today's complex, multifunction calculator is no more than a shrunken adding machine with extra capabilities. It offers some aids to help us manage complexity, but its appearance, except for more labels, reveals little about its added capabilities. Its operation is obscured by keys with multiple labels and mode-dependent actions that require many-to-one mapping of controls onto functions, and by invisible stacks and memory registers that hide their contents.

A calculator implemented as a soft machine makes obvious much more of its functionality and current state while maintaining a simple appearance. The "soft calculator" appears on the screen as a simple four function calculator augmented with keys to access the more complex functions, memory registers and on-line instructions. The placement of the extra keys off to one side and their labels hint at their purpose. Touching one of these keys labeled "STATISTICAL FUNCTIONS" causes it to light up and another group of keys to appear. These new keys enable us to do statistical calculations easily. Touching the "STATISTICAL FUNCTIONS" key again causes it to go dim and the evoked keys to disappear. We can achieve the ideal of a one-to-one mapping between keys and functions regardless of the number of functions the calculator may have because there is ample room on the screen, and keys can disappear when no longer needed. Additional displays are created on demand to store and show intermediate results and useful constants. Such numbers are entered into further calculations simply by touching the corresponding displays. Touching a key labeled "MEMORY" evokes keys to store, recall and accumulate numbers in memory registers with corresponding displays showing their contents.

The calculator is troublesome to represent as a computer system using other interface designs. A calculator operated with a command language could not be learned without consulting a manual. A menu interface would be extremely tedious. Rapid entry of numbers would be difficult by selecting soft keys with a mouse in see-and-point interfaces like those of the XEROX Star (Smith, Irby, Kimball & Verplank, 1982) and Apple Lisa[1] (Ehardt, 1983) professional workstations. Of course, such interfaces will be ideal in other situations and applications. We hope that this calculator example shows clearly and convincingly that a soft machine interface is qualitatively different from command languages, menus and see-and-point interfaces, and that there are circumstances where a soft machine offers obvious advantages.

Operability with Flexibility

A soft machine, properly designed, preserves the essential properties of hard machines that make them easy to use: the global properties—modularity, revealing form, a one-to-one mapping between controls and operations, etc.—and the local properties—presence, labels, type, etc.—that are the language of controls.

A soft machine, furthermore, is flexible. Graphics software enables soft controls to appear and disappear on demand, to move about the screen, and to change appearance so that the form of the machine acquires a dynamic character indicative of the ever changing state of the soft machine. This flexibility gives the designer of soft machines the power to manage the complexity of computer systems to keep us from being overwhelmed. A complex soft machine can be composed of many simpler soft machines, each serving one of the subfunctions of the whole machine. Then to accomplish the overall function, we need deal with only one simple machine at a time. This strategy for managing complexity is essentially identical to the notion of progressive disclosure that characterizes the XEROX[2] Star interface (Smith et al., 1982). This layered approach also overcomes the problem of overcrowding of controls on complex hard machines. Since only those controls relevant to a subfunction need be present at any given time, the limited space on the display screen can be shared among many controls. Hence the space available for controls on a soft machine is practically limitless.

Primitive Operations: Sow's Ears and Silk Purses

A well-designed machine, hard or soft, is comprised of primitive operations which are comprehensible and complete. By comprehensible, we mean that the nature of the operations themselves and how they should be combined and sequenced to accomplish some larger task are easily understood, learned and remembered. By complete, we mean that the operations are sufficient to do all the tasks we demand of the machine. Soft machines represent a way to organize, present and actuate the primitive operations, but leave unanswered an important question in machine design: How do we determine a good set of primitive operations, and rules for combining and sequencing the operations? A sow's ear of a design will not yield a silk purse of a machine—hard or soft. Soft machines are no panacea for bad design, but they do give the designer the flexibility and power to make a good design even better.

1. Apple is a registered trademark, and Lisa a trademark, of Apple Computer.
2. XEROX is a registered trademark of the Xerox Corporation.

ORGANIZING A COLLECTION OF SOFT MACHINES

Work on any substantial project will entail working with a collection of soft machines. We want the collection organized so that we have easy access to all the machines needed for the project with no unneeded machines cluttering our work environment. We propose that our work environment be organized into parallel three-level structures of *tools* (a soft machine is an instance of a tool) and *data* (e.g., documents, spreadsheets and databases.)

For tools, the three levels are tool bin, workshop and workbench. The *tool bin* is the entire collection of tools available on a particular computer. The *workshop* is a work environment specialized for a particular type of work or task such as document preparation or programming, and containing all and only those tools needed to accomplish the task. The tools in the workshop are simply copies to those found in the tool bin. The *workbench* is analogous to a work surface or counter in the workshop where the actual work is done. On the workbench are tools needed just for the current task. These tools are temporary copies which are "put away" when the work is done. These three levels correspond naturally to a houseware store, kitchen, and kitchen counter.

For data, the three levels are file, folder and paper as in the Star and Lisa systems. As in our traditional office environment, *files* contain relatively inactive data, *folders* contain data for an active project, and *papers* represent the aspect of the project that is being actively worked on. Files reside in some independent space, but folders reside in workshops, and papers on workbenches. The analogy to the traditional office environment is clear.

Our houses have evolved special work environments such as the kitchen, bathroom and woodshop to make activities more efficient and to eliminate unwanted interference between activities. We believe that computers should be organized for similar reasons into specialized work environments with both the tools and data needed for particular tasks conveniently and simultaneously on hand.

REFERENCES

1. A. F. Albert, "The Effect of Graphic Input Devices on Performance in a Cursor Positioning Task," *Proceedingsof the Human Factors Society*, 1982, pp. 54-58.
2. S. K. Card, W. K. English and B. J. Burr, "Evaluation of Mouse, Rate-Controlled Isometric Joystick, Step Keys, and Text Keys for Text Selection on a CRT," *Ergonomics*, 21, 1978, pp. 601-613.
3. A. Chapanis, "Design of Controls," in H. P. Van Cott and R. G. Kinkade, *Human Engineering Guide to Equipment Design*, McGraw-Hill, 1972, pp. 346-379.
4. J. L. Ehardt, "Apple's Lisa: A Personal Office System," *The Seybold Report on Office Systems*, 6, January 1983.
5. K. C. Knowlton, "Computer Displays Optically Superimposed on Input Devices," *Bell System Technical Journal*, 56, March 1977, pp. 367-383.
6. B. J. Mirrer, "An Interactive, Graphical, Touch-Oriented Speech Editor," MIT Master's Thesis, 1982.
7. C. Schmandt, "The Intelligent Ear—A Graphical Interface to Digital Audio," *Proceedings of the 1981 International Conference on Cybernetics and Society*, 1981, pp. 393-397.
8. P. J. Schuyten, "Xerox Introduces 'Intelligent Copier,'" *The New York Times*, Thursday, September 25, 1980, p. D4.
9. D. C. Smith, C. Irby, R. Kimball and B. Verplank, "Designing the Star User Interface," *Byte*, April 1982, pp. 242-282.

The Automatic Recognition of Gestures

Dean Harris Rubine

December, 1991

CMU–CS–91–202

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science at Carnegie Mellon University.

Thesis Committee:

Roger B. Dannenberg, Advisor
Dario Giuse
Brad Myers
William A. S. Buxton, University of Toronto

Abstract

Gesture-based interfaces, in which the user specifies commands by simple freehand drawings, offer an alternative to traditional keyboard, menu, and direct manipulation interfaces. The ability to specify objects, an operation, and additional parameters with a single intuitive gesture makes gesture-based systems appealing to both novice and experienced users.

Unfortunately, the difficulty in building gesture-based systems has prevented such systems from being adequately explored. This dissertation presents work that attempts to alleviate two of the major difficulties: the construction of gesture classifiers and the integration of gestures into direct-manipulation interfaces. Three example gesture-based applications were built to demonstrate this work.

Gesture-based systems require classifiers to distinguish between the possible gestures a user may enter. In the past, classifiers have often been hand-coded for each new application, making them difficult to build, change, and maintain. This dissertation applies elementary statistical pattern recognition techniques to produce gesture classifiers that are trained by example, greatly simplifying their creation and maintenance. Both single-path gestures (drawn with a mouse or stylus) and multiple-path gestures (consisting of the simultaneous paths of multiple fingers) may be classified. On a 1 MIPS workstation, a 30-class single-path recognizer takes 175 milliseconds to train (once the examples have been entered), and classification takes 9 milliseconds, typically achieving 97% accuracy. A method for classifying a gesture as soon as it is unambiguous is also presented.

This dissertation also describes GRANDMA, a toolkit for building gesture-based applications based on Smalltalk's Model/View/Controller paradigm. Using GRANDMA, one associates sets of gesture classes with individual views or entire view classes. A gesture class can be specified at runtime by entering a few examples of the class, typically 15. The semantics of a gesture class can be specified at runtime via a simple programming interface. Besides allowing for easy experimentation with gesture-based interfaces, GRANDMA sports a novel input architecture, capable of supporting multiple input devices and multi-threaded dialogues. The notion of virtual tools and semantic feedback are shown to arise naturally from GRANDMA's approach.

ii

Acknowledgments

First and foremost, I wish to express my enormous gratitude to my advisor, Roger Dannenberg. Roger was always there when I needed him, never failing to come up with a fresh idea. In retrospect, I should have availed myself more than I did. In his own work, Roger always addresses fundamental problems, and his solutions are always simple and elegant. I try to follow Roger's example in my own work, usually falling far short. Roger, thank you for your insight and your example. Sorry for taking so long.

I was incredibly lucky that Brad Myers showed up at CMU while I was working on this research. His seminar on user interface software gave me the knowledge and breadth I needed to approach the problem of software architectures for gesture-based systems. Furthermore, his extensive comments on drafts of this document improved it immensely. Much of the merit in this work is due to him. Thank you, Brad. I am also grateful to Bill Buxton and Dario Giuse, both of whom provided valuable criticism and excellent suggestions during the course of this work.

It was Paul McAvinney's influence that led me to my thesis topic; had I never met him, mine would have been a dissertation on compiler technology. Paul is an inexhaustible source of ideas, and this thesis is really the *second* idea of Paul's that I've spent multiple years pursuing. Exploring Paul's ideas could easily be the life's work of hundreds of researchers. Thanks, Paul, you madman you.

My wife Ruth Sample deserves much of the credit for the existence of this dissertation. She supported me immeasurably, fed me and clothed me, made me laugh, motivated me to finish, and lovingly tolerated me the whole time. Honey, I love you. Thanks for everything.

I could not have done it with the love and support of my parents, Shirley and Stanley, my brother Scott, my uncle Donald, and my grandma Bertha. For years they encouraged me to be a doctor, and they were not the least bit dismayed when they found out the kind of doctor I wanted to be. They hardly even balked when "just another year" turned out to be six. Thanks, folks, you're the best. I love you all very much.

My friends Dale Amon, Josh Bloch, Blaine Burks, Paul Crumley, Ken Goldberg, Klaus Gross, Gary Keim, Charlie Krueger, Kenny Nail, Eric Nyberg, Barak Pearlmutter, Todd Rockoff, Tom Neuendorffer, Marie-Helene Serra, Ellen Siegal, Kathy Swedlow, Paul Vranesevic, Peter Velikonja, and Brad White all helped me in innumerable ways, from technical assistance to making life worth living. Peter and Klaus deserve special thanks for all the time and aid they've given me over the years. Also, Mark Maimone and John Howard provided valuable criticism which helped me prepare for my oral examination. I am grateful to you all.

I wish to also thank my dog Dismal, who was present at my feet during much of the design, implementation, and writing efforts, and who concurs on all opinions. Dismal, however, strongly objects to this dissertation's focus on *human* gesture.

I also wish to acknowledge the excellent environment that CMU Computer Science provides; none of this work would have been possible without their support. In particular, I'd like to thank Nico Habermann and the faculty for supporting my work for so long, and my dear friends Sharon Burks, Sylvia Berry, Edith Colmer, and Cathy Copetas.

Contents

List of Figures

List of Tables

to Grandma Bertha

Chapter 1

Introduction

People naturally use hand motions to communicate with other people. This dissertation explores the use of human gestures to communicate with computers.

Random House [122] defines "gesture" as "the movement of the body, head, arms, hands, or face that is expressive of an idea, opinion, emotion, etc." This is a rather general definition, which characterizes well what is generally thought of as gesture. It might eventually be possible through computer vision for machines to interpret gestures, as defined above, in real time. Currently such an approach is well beyond the state of the art in computer science.

Because of this, the term "gesture" usually has a restricted connotation when used in the context of human-computer interaction. There, gesture refers to hand markings, entered with a stylus or mouse, which function to indicate scope and commands [109]. Buxton [14] gives a fine example, reproduced here as figure 1.1. In this dissertation, such gestures are referred to as *single-path* gestures.

Recently, input devices able to track the paths of multiple fingers have come into use. The Sensor Frame [84] and the DataGlove [32, 130] are two examples. The human-computer interaction community has naturally extended the use of the term "gesture" to refer to hand motions used to indicate commands and scope, entered via such multiple finger input devices. These are referred to here as *multi-path* gestures.

Rather than defining gesture more precisely at this point, the following section describes an

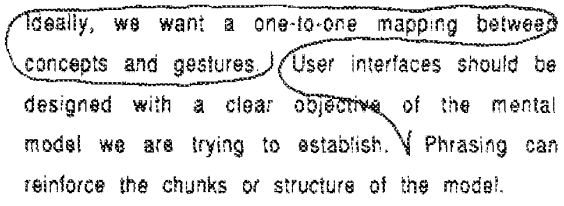


Figure 1.1: Proofreader's Gesture (from Buxton [15])

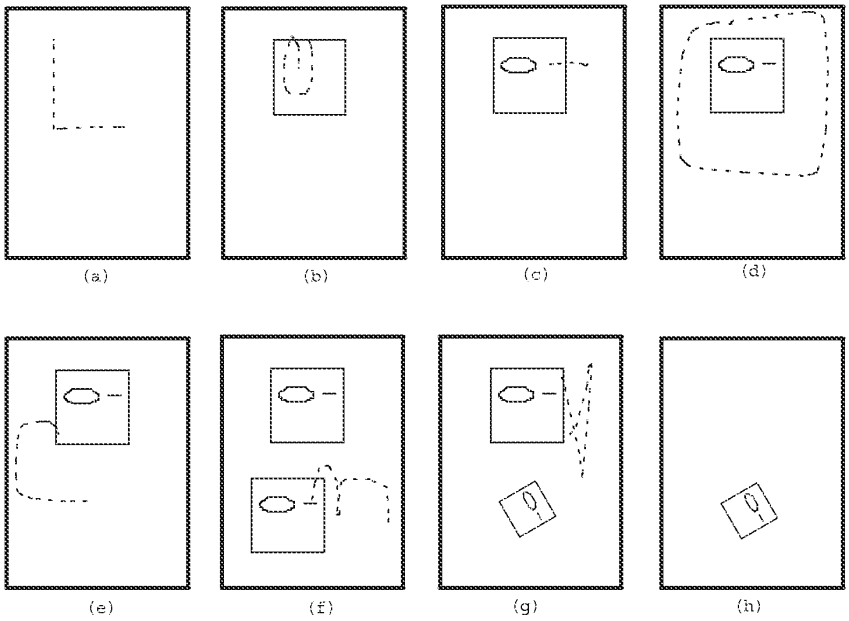


Figure 1.2: GDP, a gesture-based drawing program

example application with a gestural interface. A more technical definition of gesture will be presented in section 1.6.

1.1 An Example Gesture-based Application

GRANDMA is a toolkit used to create gesture-based systems. It was built by the author and is described in detail in the pages that follow. GRANDMA was used to create GDP, a gesture-based drawing editor loosely based on DP [42]. GDP provides for the creation and manipulation of lines, rectangles, ellipses, and text. In this section, GDP is used as an example gesture-based system. GDP's operation is presented first, followed by a description of how GRANDMA was used to create GDP's gestural interface.

1.1.1 GDP from the user's perspective

GDP's operation from a user's point of view will now be described. (GDP's design and implementation is presented in detail in Section 8.1.) The intent is to give the reader a concrete example of a gesture-based system before embarking on a general discussion of such systems. Furthermore, the description of GDP serves to illustrates many of GRANDMA's capabilities. A new interaction technique, which combines gesture and direct manipulation in a single interaction, is also introduced in the description.

Figure 1.2 shows some snapshots of GDP in action. When first started, GDP presents the user with a blank window. Panel (a) shows the rectangle gesture being entered. This gesture is drawn like an "L."[1] The user begins the gesture by positioning the mouse cursor and pressing a mouse button. The user then draws the gesture by moving the mouse.

The gesture is shown on the screen as is being entered. This technique is called *inking* [109], and provides valuable feedback to the user. In the figure, inking is shown with dotted lines so that the gesture may be distinguished from the objects in the drawing. In GDP, the inking is done with solid lines, and disappears as soon as the gesture has been recognized.

The end of the rectangle gesture is indicated in one of two ways. If the user simply releases the mouse button immediately after drawing "L" a rectangle is created, one corner of which is at the start of the gesture (where the button was first pressed), with the opposite corner at the end of the gesture (where the button was released). Another way to end the gesture is to stop moving the mouse for a given amount of time (0.2 seconds works well), while still pressing the mouse button. In this case, a rectangle is created with one corner at the start of the gesture, and the opposite corner at the current mouse location. As long as the button is held, that corner is dragged by the mouse, enabling the size and shape of the rectangle to be determined interactively.

Panel (b) of figure 1.2 shows the rectangle that has been created and the ellipse gesture. This gesture creates an ellipse with its center at the start of the gesture. A point on the ellipse tracks the mouse after the gesture has been recognized; this gives the user interactive control over the size and eccentricity of the ellipse.

Panel (c) shows the created ellipse, and a line gesture. Similar to the rectangle and the ellipse, the start of the gesture determines one endpoint of the newly created line, and the mouse position after the gesture has been recognized determines the other endpoint, allowing the line to be rubberbanded.

Panel (d) shows all three shapes being encircled by a pack gesture. This gesture packs (groups) all the objects which it encloses into a single composite object, which can then be manipulated as a unit. Panel (e) shows a copy gesture being made; the composite object is copied and the copy is dragged by the mouse.

Panel (f) shows the rotate-and-scale gesture. The object is made to rotate around the starting point of the gesture; a point on the object is dragged by the mouse, allowing the user to interactively determine the size and orientation of the object.

Panel (g) shows the delete gesture, essentially an "X" drawn with a single stroke. The object at the gesture start is deleted, as shown in panel (h).

This brief description of GDP illustrates a number of features of gesture-based systems. Perhaps the most striking feature is that each gesture corresponds to a high-level operation. The class of the gesture determines the operation; attributes of the gesture determine its scope (the operands) and any additional parameters. For example, the delete gesture specifies the object to be deleted, the pack gesture specifies the objects to be combined, and the line gesture specifies the endpoints of the line.

[1] It is often convenient to describe single-path gestures as if they were handwritten letters. This is not meant to imply that gesture-based systems can only recognize alphabetic symbols, or even that they usually recognize alphabetic symbols. The many ways in which gesture-based systems are distinct from handwriting-recognition systems will be enumerated in section 1.8.

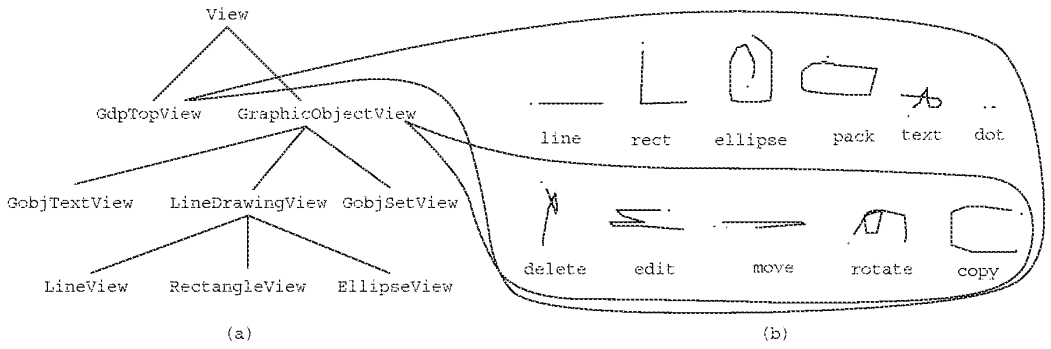


Figure 1.3: GDP's `View` class hierarchy and associated gestures
A period indicates the first point of each gesture.

It is possible to control more than positional parameters with gestural attributes. For example, one version of GDP uses the length (in pixels) of the line gesture to control the thickness of the new line.

Note how gesturing and direct manipulation are combined in a new two-phase interaction technique. The first phase, the collection of the gesture, ends when the user stops moving the mouse while holding the button. At that time, the gesture is recognized and a number of parameters to the application command are determined. After recognition, a manipulation phase is entered during which the user can control additional parameters interactively.

In addition to its gestural interface, GDP provides a more traditional click-and-drag interface. This is mainly used to compare the two styles of interface, and is further discussed in Section 8.1. The gestural interface is grafted on top of the click-and-drag interface, as will be explained next.

1.1.2 Using GRANDMA to Design GDP's Gestures

In the current work, the *gesture designer* creates a gestural interface to an application out of an existing click-and-drag interface to the application. Both the click-and-drag interface and the application are built using the object-oriented toolkit GRANDMA. The gesture designer only modifies the way input is handled, leaving the output mechanisms untouched.

A system built using GRANDMA utilizes the object-oriented programming paradigm to represent windows and the graphics objects displayed in windows. For example, figure 1.3a shows GDP's `View` class hierarchy.[2] This hierarchy shows the relationship of the classes concerned with output. The task of the gesture designer is to determine which of these classes are to have associated gestures, and for each such view class, to design a set of gestures that intuitively expresses the allowable operations on the view. Figure 1.2b shows the sets of gestures associated with GDP's `GraphicObjectView` and `GdpTopView` classes. The `GraphicObjectView` collectively

[2]For expositional purposes, the hierarchy shown is a simplified version of the actual hierarchy. Some of the details that follow have also been simplified. Section 8.1 tells the truth in gory detail.

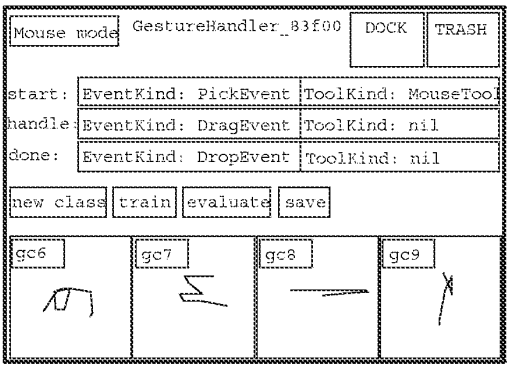


Figure 1.4: Manipulating gesture handlers at runtime

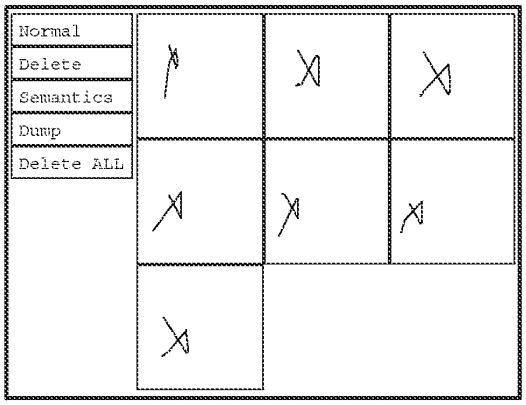


Figure 1.5: Adding examples of the delete gesture

refers to the line, rectangle, and ellipse shapes, while `GdpTopView` represents the window in which GDP runs.

GRANDMA is a Model/View/Controller-like system [70]. In GRANDMA, an input event handler (a "controller" in MVC terms) may be associated with a view class, and thus shared between all instances of the class (including instances of subclasses). This adds flexibility while eliminating a major overhead of Smalltalk MVC, where one or more controller objects are associated with each view object that expects input.

The gesture designer adds gestures to GDP's initial click-and-drag interface at runtime. The first step is to create a new gesture handler and associate it the `GraphicObjectView` class, easily done using GRANDMA. Figure 1.4 shows the gesture handler window after a number of gestures have been created (using the "new class" button), and figure 1.5 shows the window in which examples of the delete gesture have been entered. Fifteen examples of each gesture class typically suffice. If a gesture is to vary in size and/or orientation, the examples should reflect that.

Clicking on the "Semantics" button brings up a window that the designer uses to specify the semantics of each gesture in the handler's set. The window is a structured editing and browsing interface to a simple Objective-C [28] interpreter, and the designer enters three expressions: `recog`, evaluated when the gesture is first recognized; `manip`, evaluated on subsequent mouse points; and done, evaluated when the mouse button is released. In this case, the delete semantics simply change the mouse cursor to a delete cursor, providing feedback to the user, and then delete the view at which the gesture was aimed. The expressions entered are[3]

[3]Objective C syntax is used throughout. `[view delete]` sends the `delete` message to the object referred to by the variable `view`. `[handler mousetool:DeleteCursor]` sends the `mousetool:` message to the object referred to by the variable `handler` passing the value of the variable `DeleteCursor` as an argument. See Section 6.3 for more information on Objective C notation.

```
recog = [_Seq :[handler mousetool:DeleteCursor]
              :[view delete]];
manip = nil;
done = nil;
```

The designer may now immediately try out the delete gesture, as in figure 1.2g.

The designer repeats the process to create a gesture handler for the set of gestures associated with class GdpTopView, the view that refers to the window in which GDP runs. This handler deals with the gestures that create graphic objects, the pack gesture (which creates a set out of the enclosed graphic objects), the dot gesture (which repeats the last command), and the gestures also handled by GraphicObjectView's gesture handler (which when made at a GdpTopView change the cursor without operating directly on a graphic object).

The attributes of the gesture are directly available for use in the gesture semantics. For example, the semantics of the line gesture are:

```
recog = [Seq :[handler mousetool:LineCursor]
             :[[view createLine]
                      setEndpoint:0 x:<startX> y:<startY>]];
manip = [recog setEndpoint:1 x:<currentX> y:<currentY>];
done = nil;
```

The semantic expressions execute in a rich environment in which, for example, view is bound to the view at which the gesture was directed (in this case a GdpTopView) and handler is bound to the current gesture handler. Note that Seq executes its arguments sequentially, returning the value of the last, in this case the newly created line. This is bound to recog for later use in the manip expression.

The example shows how the gesture attributes, shown in angle brackets, are useful in the semantic expressions. The attributes <startX> and <startY>, the coordinates of the first point in the gesture, are used to determine one endpoint of the line, while <currentX> and <currentY>, the mouse coordinates, determine the other endpoint.

Many other gesture attributes are useful in semantics. The line semantics could be augmented to control the thickness of the line from the maximum speed or total path length of the gesture. The rectangle semantics could use the initial angle of the rectangle gesture to determine the orientation of the rectangle. The attribute <enclosed> is especially noteworthy: it contains a list of views enclosed by the gesture and is used, for example, by the pack gesture (figure 1.2d). When convenient, the semantics can simulate input to the click-and-drag interface, rather than communicating directly with application objects or their views, as shown above.

When the first point of a gesture is over more than one gesture-handling view, the union of the set of gestures recognized by each handler is used, with priority given to the foremost views. For example, any gesture made at a GDP GraphicObjectView is necessarily made over the GdpTopView. A delete gesture made at a graphic object would be handled by the GraphicObjectView while a line gesture at the same place would be handled by the GdpTopView. Set union also occurs when gestures are (conceptually) inherited via the view class hierarchy. For example, the gesture designer might create a new gesture handler for the GobjSetView class containing an unpack gesture. The set of gestures recognized by

GobjSetViews would then consist of the unpack gesture as well as the five gestures handled by GraphicObjectView.

1.2 Glossary

This section defines and clarifies some terms that will be used throughout the dissertation. It may safely be skipped and referred back to as needed. Some of the terms (click, drag) have their common usage in the human-computer interaction community, while others (pick, move, drop) are given technical definitions solely for use here.

class In this dissertation, "class" is used in two ways. "Gesture class" refers to a set of gestures all of which are intended to be treated the same, for example, the class of delete gestures. (In this dissertation, the names of gesture classes will be shown in sans serif typeface.) The job of a gesture recognizer is, given an example gesture, to determine its class (see also "gesture"). "Class" is also used in the object-oriented sense, referring to the type (loosely speaking) of a software object. It should be clear from context which of these meanings is intended.

click A click consists of positioning the mouse cursor and then pressing and releasing a mouse button, with no intervening mouse motion. In the Macintosh, a click is generally used to select an object on the screen.

click-and-drag A click-and-drag interface is a direct-manipulation interface in which objects on the screen are operated upon using mouse clicks, drags, and sometimes double-clicks.

direct manipulation A direct-manipulation interface is one in which the user manipulates a graphic representation of the underlying data by pointing at and/or moving them with an appropriate device, such as a mouse with buttons.

double-click A double-click is two clicks in rapid succession.

drag A drag consists of locating the mouse cursor and pressing the mouse button, moving the mouse cursor while holding the mouse button, and then releasing the mouse button. Drag interactions are used in click-and-drag interfaces to, for example, move objects around on the screen.

drop The final part of a drag (or click) interaction in which the mouse button is released.

eager recognition A kind of gesture recognition in which gestures are often recognized without the end of the gesture having to be explicitly signaled. Ideally, an eager recognizer will recognize a gesture as soon as enough of it has been seen to determine its class unambiguously.

gesture Essentially a freehand drawing used to indicate a command and all its parameters. Depending on context, the term maybe used to refer to an example gesture or a class of gestures, *e.g.* "a delete gesture" means an example gesture belonging to the class of delete gestures. Usually "gesture" refers to the part of the interaction up until the input is recognized as one

of a number of possible gesture classes, but sometimes the entire interaction (which includes a manipulation phase after recognition) is referred to as a gesture.

move The component of drag interaction during which the mouse is moved while a mouse button is held down. It is the presence of a move that distinguishes a click from a drag.

multi-path A multi-path gesture is one made with an input device that allows more than one position to be indicated simultaneously (multiple pointers). One may make multi-path gestures with a Sensor Frame, a multiple-finger touch pad, or a DataGlove, to name a few such devices.

off-line Considering an algorithm to be a sequence of operations, an off-line algorithm is one which examines subsequent operations before producing output for the current operation.

on-line An on-line algorithm is one in which the output of an operation is produced before any subsequent operations are read.

pick The initial part of a drag (or click) interaction consisting of positioning the mouse cursor at the desired location and pressing a mouse button.

press refers to the pressing of a mouse button.

real-time A real-time algorithm is an on-line algorithm in which each operation is processed in time bounded by a constant.

release refers to the releasing of a mouse button.

segment A segment is an approximately linear portion of a stroke. For example, the letter "L" is two segments, one vertical and one horizontal.

single-path A single-path gesture is one drawn by an input device, such as a mouse or stylus, capable of specifying only a single point over time. A single-path gesture may consist of multiple strokes (like the character "X").

single-stroke A single-stroke gesture is a single-path gesture that is one stroke. Thus drawing "L" is a single-stroke gesture, while "X" is not. In this dissertation the only single-path gestures considered are single-stroke gestures.

stroke A stroke is an unbroken curve made by a single movement of a pen, stylus, mouse, or other instrument. Generally, strokes begin and end with explicit user actions (*e.g.* , pen down/pen up, mouse button down/mouse button up).

1.3 Summary of Contributions

This dissertation makes contributions in four areas: new interaction techniques, new algorithms for gesture recognition, a new way of integrating gestures into user interfaces, and a new architecture for input in object-oriented toolkits.

The first new interaction technique is the two-phase combination of single-stroke gesture collection followed by direct manipulation, mentioned previously. In the GDP example discussed above, the boundary between the two phases is an interval of motionlessness. Eager recognition, the second new interaction technique, eliminates this interval by recognizing the single-stroke gesture and entering the manipulation phase as soon as enough of the gesture has been seen to do so unambiguously, making the entire interaction very smooth. A third new interaction technique is the two-phase interaction applied to multi-path gestures: after a multi-path gesture has been recognized, individual paths (*i.e.* fingers, possibly including additional fingers not involved in making the recognized gesture) may be assigned to manipulate independent application parameters simultaneously.

The second contribution is a new trainable, single-stroke recognition algorithm tailored for recognizing gestures. The classification is based on meaningful features, which in addition to being useful for recognition are also suitable for passing to application routines. The particular set of features used has been shown to be suitable for many different gesture sets, and is easily extensible. When restricted to features that can be updated incrementally in constant time per input point, arbitrarily large gestures may be handled. The single-stroke recognition algorithm has been extended to do eager recognition (eager recognizers are automatically generated from example gestures), and also to multi-path gesture recognition.

Third, a new paradigm for creating gestural interfaces is also propounded. As seen in the example, starting from a click-and-drag implementation of an interface, gestures are associated with classes of views (display objects), with the set of gestures recognized at a particular screen location dynamically determined by the set of overlapping views at the location, and by inheritance up the class hierarchy of each such view. The classification and attributes of gestures map directly to application operations and parameters. The creation, deletion, and manipulation of gesture handlers, gesture classes, gesture examples, and gesture semantics all occur at runtime, enabling quick and easy experimentation with gestural interfaces.

Fourth, GRANDMA, as an object-oriented user interface toolkit, makes some contributions to the area of input handling. Event handler objects are associated with particular views or entire view classes. A single event handler may be shared between many different objects, eliminating a major overhead of MVC systems. Multiple event handlers may be associated with a single object, enabling the object to support multiple interaction techniques simultaneously, including the use of multiple input devices. Furthermore, a single mechanism handles both mouse tools (*e.g.* a delete cursor that deletes clicked-upon objects) and virtual tools (*e.g.* a delete icon that is dragged around and dropped upon objects to delete them). Additionally, GRANDMA provides support for semantic feedback, and enables the runtime creation and manipulation of event handlers.

1.4 Motivation for Gestures

At this point, the reader should have a good idea of the scope of the work to be presented in this dissertation. Stepping back, this section begins a general discussion of gestures by examining the motivation for using and studying gesture-based interfaces. Much of the discussion is based on that of Buxton [14].

Computers get faster, bitmapped displays produce ever increasing information rates, speech and

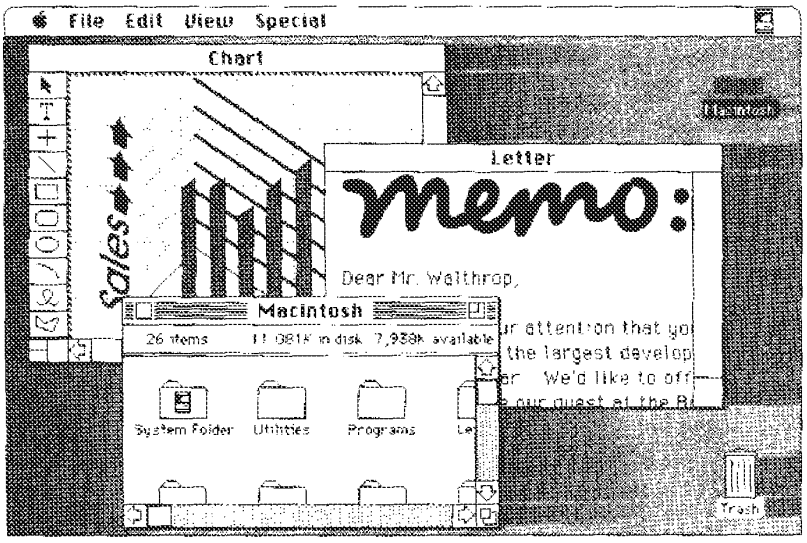


Figure 1.6: Macintosh Finder, MacDraw, and MacWrite (from Apple [2])

music can be generated in real-time, yet input just seems to plod along with little or no improvement. This is regrettable because, in Paul McAvinney's words [84], most of the useful information in the world resides in humans, not computers. Most people who interact with computers spend most of their time entering information [22]. Due to this input bottleneck, the total time to do many tasks would hardly improve even if computers became infinitely fast. Thus, improvements in input technology are a major factor in improving the productivity of computer users in general.

Of course, progress has been made. Input has progressed from batch data entry, to interactive line editors, to two-dimensional screen editors, to mouse-based systems with bitmapped displays. Pointing with a mouse has proved a useful interaction technique in many applications. "Click and drag" interfaces, where the user directly manipulates graphic objects on the screen with a mouse, are often very intuitive to use. Because of this, direct manipulation interfaces have become commonplace, despite being rather difficult to build.

Consider the Macintosh [2], generally regarded as having a good direct-manipulation interface. As shown in figure 1.6, the screen has on it a number of graphic objects, including file icons, folder icons, sliders, buttons, and pull-down menu names. Each one is generally a rectangular region, which may be clicked, sometimes double-clicked, and sometimes dragged. The Macintosh Finder, which may be used to access all Macintosh applications and documents, is almost entirely controlled via these three interaction techniques.[4]

The click and double-click interactions have a single object (or location) as parameter. The drag

[4]Obviously this discussion ignores keyboard entry of text and commands.

interaction has two parameters: an object or location where the mouse button is first pressed, and another object or location at the release point. Having only these three interaction techniques is one reason the Macintosh is simple to operate. There is, however, a cost: both the application and the user must express all operations in terms of these three interaction techniques.

An application that provides more than three operations on any given object (as many do) has several design alternatives. The first, exemplified by the Finder, relies heavily on *selection*. In the Finder, a click interaction selects an object, a double-click opens an object (the meaning of which depends upon the object's type), and a drag moves an object (the meaning of which is also object-type specific). Opening an object by a double-click is a means for invoking the most common operation on the object, *e.g.* opening a MacWrite document starts the MacWrite application on the document. Dragging is used for adjusting sliders (such as those which scroll windows), changing window size or position, moving files between folders, and selecting menu items.

All other operations are done in at least two steps: first the object to be operated upon is selected, and then the desired operation is chosen from a menu. For example, to print an object, one selects it (click) then chooses "Print" from the appropriate menu (drag); to move some text, one selects it (drag), chooses "Cut" (drag), selects an insertion point (click), and chooses "Paste" (drag). The cost of only having three interaction techniques is that some operations are necessarily performed via a sequence of interactions. The user must adjust her mental model so that she thinks in terms of the component operations.

An alternative to the selection-based click-and-drag approach is one based on modes. Consider MacDraw [2], a drawing program. The user is presented with a palette offering choices such as line, text, rectangles, circles, and so on. Clicking on the "line" icon puts the program into line-drawing mode. The next drag operation in the drawing window cause lines to be drawn. In MacDraw, after the drag operation the program reverts back to selection mode. DP, the program upon which GDP is based, is similar except that it remains in its current mode until it is explicitly changed. Mistakes occur when the user believes he is in one mode but is actually in another. The claim that direct manipulation interfaces derive their power from being modeless is not really true. Good direct manipulation interfaces simply make the modes very visible, which helps to alleviate the problems of modal interfaces.

By mandating the sole use of click, double-click, and drag interactions, the Macintosh interface paradigm necessarily causes conceptually primitive tasks to be divided into a sequence of primitive interactions. The intent of gestural interfaces is to avoid this division, by packing the basic interaction with all the parameters necessary to complete the entire transaction. Ideally, each primitive task in the user's model of the application is executed with a single gesture. Such interfaces would have less modeness than the current so-called modeless interfaces.

The Macintosh discussion in the previous section is somewhat oversimplified. Many applications allow variations on the basic interaction techniques; for example "shift-click" (holding the shift key while clicking the mouse) adds an object to the current set of selected objects. Other computer systems allow different mouse buttons to indicated different operations. There is a tradeoff between having a small number and a large number of (consistently applied) interaction techniques. The former results in a system whose primitive operations are easy to learn, perform, and recall, but a single natural chunk may be divided into a sequence of operations. In the latter case, the primitive

operations are harder to learn (because there are more of them), but each one can potentially implement an entire natural chunk.

The motivation for gestural interfaces may also apply to interfaces which combine modalities (*e.g.* speech and pointing). As with gestures, one potential benefit of multi-modal interfaces is that different modalities allow many parameters to be specified simultaneously, thus eliminating the need for modes. The "Put-That-There" system is one example [12].

1.5 Primitive Interactions

The discussion thus far has been vague as to what exactly may be considered a "primitive" interaction technique. The Macintosh has three: click, double-click, and drag. It is interesting to ask what criteria can be used for judging the "primitiveness" of proposed interaction techniques.

Buxton [14] suggests physical tension as a criterion. The user, starting from a relaxed state, begins a primitive interaction by tensing some muscles. The interaction is over when the user again relaxes those muscles. Buxton cites evidence that "such periods of tension are accompanied by a heightened state of attentiveness and improved performance." The three Macintosh interaction techniques all satisfy this concept of primitive interaction. (Presumably the user remains tense during a double-click because the time between clicks is short.)

Buxton likens the primitive interaction to a musical phrase. Each consists of a period of tension followed by a return to a state where a new phrase may be introduced. In human-computer interaction, such a phrase is used to accomplish a chunk of a task. The goal is to make each of these chunks a primitive task in the user's model of the application domain. This is what a gesture-based interface attempts to do.

1.6 The Anatomy of a Gesture

In this section a technical definition of gesture is developed, and the syntactic and semantic properties of gestures are then discussed. The dictionary definition of gesture, "expressive motion," has already been seen. How can the notion of gesture in a form suitable for sensing and processing by machine be captured?

1.6.1 Gestural motion

The motion aspect of gesture is formalized as follows: a gesture consists of the paths of multiple points over time. The points in question are (conceptually) affixed to the parts of the body which perform the gesture. For hand gestures, the points tracked might include the fingertips, knuckles, palm, and wrist of each hand. Over the course of a gesture, each point traces a path in space. Assuming enough points (attached to the body in appropriate places), these paths contain the essence of the gestural motion. A computer with appropriate hardware can rapidly sample positions along the paths, thus conveniently capturing the gesture.

The idea of gesture as the motion of multiple points over time is a generalization of pointing. Pointing may be considered the simplest gesture: it specifies a single position at an instance of

time. This is generalized to allow for the movement of the point over time, *i.e.* a path. A further generalization admits multiple paths, *i.e.* the movement of multiple points over time.[5]

Current gesture-sensing hardware limits both the number of points which may be tracked simultaneously and the dimensionality of the space in which the points travel. Gestures limited to the motion of a single point are referred to here as *single-path* gestures. Most previous gestural research has focused upon gestures made with a stylus and tablet, mouse, or single-finger touch pad. The gestures which may be made with such devices are two-dimensional, single-path gestures.

An additional feature of existing hardware is that the points are not tracked at all times. For example, a touch pad can only determine finger position when the finger is touching the pad. Thus, the path of the point will have a beginning (when the finger first makes contact) and an end (when the finger is lifted). This apparent limitation of certain gesture-sensing hardware may be used to delineate the start and possibly the end of each gesture, a necessary function in gesture-based systems. Mouse buttons may be used to similar effect.[6]

In all the work reported here, a gesture (including the manipulation phase after recognition) is always a primitive interaction. A gesture begins with the user going from a relaxed state to one of muscular tension, and ends when the user again relaxes. It is further assumed that the tension or relaxation of the user is directly indicated by some aspect of the sensing hardware. For mouse gestures, the user is considered in a state of tension if and only if a mouse button is pressed. Thus, in the current work a double-click is not considered a gesture. This is certainly a limitation, but one that could be removed, for example by having a minimum time that the button needs to be released before the user is considered to have relaxed. This added complication has not been explored here.

The space in which the points of the gesture move is typically physical space, and thus a path is represented by a set of points (x, y, z, t) consisting of three spatial Cartesian coordinates and time. However, there are devices which measure non-spatial gestural parameters; hence, gestures consisting of paths through a space where at least some of the coordinates are not lengths are possible. For example, some touch pads can sense force, and for this hardware a gesture path might consist of a set of points (x, y, f, t), f being the force measurement at time t.

The formalization of gesture as multiple paths is just one among many possible representations. It is a good representation because it coincides nicely with most of the existing gesture-sensing hardware, and it is a useful form for efficient processing. The multiple-snapshot representation, in which each snapshot gives the position of multiple points at a single instant, is another possibility, and in some sense may be considered the dual of multiple paths. Such a representation might be more suitable for gestural data derived from hardware (such as video cameras) which are not considered in this dissertation.

1.6.2 Gestural meaning

In addition to the physical aspect of a gesture, there is the content or meaning of the gesture to consider. Generally speaking, a gesture contains two kinds of information: *categorical* and

[5] A configuration of multiple points at a single instance of time may be termed *posture*. Posture recognition is commonly used with the DataGlove.

[6] Buxton [17] presents a model of the discrete signaling capabilities of various pointing devices and a list of the signaling requirements for common interaction techniques.

parametric. Consider the different motions between people meaning "come here" (beckoning gestures), "stop" (prohibiting gestures), and "keep going" (encouragement gestures). These are different categories, or *classes*, of gestures. Within each class, a gesture also can indicate parametric data. For example, a parameter of the beckoning gesture is the urgency of the request: "hurry up" or "take your time." In general, the category of the gesture must be determined before the parameters can be interpreted.

Parametric information itself comes in two forms. The first is the kind of information that can be culled at the time the gesture is classified. For example, the position, size and orientation of the gesture fall into this category. The second kind of parametric gestural information is manipulation information. After the gesture is recognized, the user can use this kind of parametric information to continuously communicate information. An example would be the directional information communicated by the gestures of a person helping a driver to back up a truck. An example from GDP (see Section 1.1) is the rubberbanding of a line after it is created, where the user continuously manipulates one endpoint.

The term "gesture" as used here does not exactly correspond to what is normally thought of as gesture. Many gestures cannot currently be processed by machine due to limitations of existing gesture-sensing hardware. Also, consider what might be referred to as "direct-manipulation gestures." A person turning a knob would not normally be considered to be gesturing. However, a similar motion used to manipulate the graphic image of a knob drawn on a computer display is considered to be a gesture. Actually, the difference here is more illusory than real: a person might make the knob-turning gesture at another person, in effect asking the latter to turn the knob. The intent here is simply to point out the very broad class of motions considered herein to be gesture.

While the notion of gesture developed here is very general (multiple paths), in practice, machine gestures have hitherto almost always been limited to finger and/or hand motions. Furthermore, the paths have largely been restricted to two dimensions. The concentration on two-dimensional hand gesturing is a result of the available gesture-sensing hardware. Of course, such hardware was built because it was believed that hand and fingers are capable of accurate and diverse gesturing, yet more amenable to practical detection than facial or other body motions. With the appearance of new input devices, three (or more) dimensional gesturing, as well as the use of parts of the body other than the hand, are becoming possible. Nonetheless, this dissertation concentrates largely on two-dimensional hand gestures, assuming that by viewing gesture simply as multiple paths, the work described may be applied to non-hand gestures, or generalized to apply to gestures in three or more dimensions.

1.7 Gesture-based systems

A gesture-based interface, as the term is used here, is one in which the user specifies commands by gesturing. Typically, gesturing consists of drawing or other freehand motions. Excluded from the class of gesture-based interfaces are those in which input is done solely via keyboard, menu, or click-and-drag interactions. In other words, while pointing is in some sense the most basic gesture, those interfaces in which pointing is the only form of gesture are not considered here to be gesture-based interfaces. A gesture-based system is a program (or set of programs) with which the user interacts via a gesture-based interface.

In all but the simplest gesture-based systems, the user may enter a gesture belonging to one of several different gesture categories or classes; the different classes refer to different commands to the system. An important component of gesture-based systems is the gesture *recognizer* or *classifier*, the module whose job is to classify the user's gesture as the first step toward inferring its meaning. This dissertation addresses the implementation of gesture recognizers, and their incorporation into gesture-based systems.

1.7.1 The four states of interaction

User interaction with the gesture-based systems considered in this dissertation may be described using the following four state model. The states--WAIT, COLLECT, MANIPULATE, EXECUTE--usually occur in sequence for each interaction.

- The WAIT state is the quiescent state of the system. The system is waiting for the user to initiate a gesture.
- The COLLECT state is entered when the user begins to gesture. While in this state, the system collects gestural data from the input hardware in anticipation of classifying the gesture. For most gesturing hardware, an explicit start action (such as pressing a mouse button) indicates the beginning of each gesture, and thus causes the system to enter this state.
- The MANIPULATE state is entered once the gesture is classified. This occurs in one of three ways:

  1. The end of the gesture is indicated explicitly, *e.g.* by releasing the mouse button;
  2. the end of the gesture is indicated implicitly, *e.g.* by a timeout which indicates the user has not moved the mouse for, say, 200 milliseconds; or
  3. the system initiates classification because it believes it has now seen enough of the gesture to classify it unambiguously (eager recognition).

  When the MANIPULATE state is entered, the system should provide feedback to the user as to the classification of the gesture and update any screen objects accordingly. While in this state, the user can further manipulate the screen objects with his motions.
- The EXECUTE state is entered when the user has completed his role in the interaction, and has indicated such (*e.g.* by releasing the mouse button). At this point the system performs any final actions as implied by the user's gesture. Ideally, this state lasts only a very short time, after which the display is updated to reflect the current state of the system, and the system reverts back to the WAIT state.

This model is sufficient to describe most current systems which use pointing devices. (For simplicity, keyboard input is ignored.) Depending on the system, the COLLECT or MANIPULATE state may be omitted from the cycle. A handwriting interface will usually omit the MANIPULATE state, classifying the collected characters and executing the resulting command. Conversely, a

direct-manipulation system will omit the COLLECT state (and the attendant classification). The GDP example described above has both COLLECT and MANIPULATE phases. The result is the new two-phase interaction technique mentioned earlier.

1.8 A Comparison with Handwriting Systems

In this section, the frequently asked question, "how do gesture-based systems differ from handwriting systems?" is addressed.

Handwriting systems may broadly be grouped into two classes: on-line and off-line. On-line handwriting recognition simply means characters are recognized as they are drawn. Usually, the characters are drawn with a stylus on a tablet, thus the recognition process takes as input a list of successive points or line segments. The problem is thus considerably different than off-line handwriting recognition, in which the characters are first drawn on paper, and then optically scanned and represented as two-dimensional rasters. Suen, Berthod, and Mori review the literature of both on-line and off-line handwriting systems [125], while Tappert, Suen, and Wakaha [129] give a recent review of on-line handwriting systems. The intention here is to contrast gesture-based systems with on-line handwriting recognition systems, as these are the most closely related.

Gesture-based systems have much in common with systems which employ on-line handwriting recognition for input. Both use freehand drawing as the primary means of user input, and both depend on recognizers to interpret that input. However, there are some important differences between the two classes of systems, differences that illustrate the merits of gesture-based systems:

- Gestures may be motions in two, three, or more dimensions, whereas handwriting systems are necessarily two-dimensional. Similarly, single-path and multiple-path gestures are both possible, whereas handwriting is always a single path.

- The alphabet used in a handwriting system is generally well-known and fixed, and users will generally have lifelong experience writing that alphabet. With gestures, it is less likely that users will have preconceptions or extensive experience.

- In addition to the command itself, a single gesture can specify parameters to the command. The proofreader's gesture (figure 1.1) discussed above, is an excellent example. Another example, also due to Buxton [21], and used in GSCORE (Section 8.2), is a musical score editor, in which a single stroke indicates the location, pitch, and duration of a note to be added to the score.

- As stated, a command and all its parameters may be specified with a single gesture. The physical relaxation of the user when she completes a gesture reinforces the conceptual completion of a command [14].

- Gestures of a given class may vary in both size and orientation. Typical handwriting recognizers expect the characters to be of a particular size and oriented in the usual manner (though successful systems will necessarily be able to cope with at least small variations in size and orientation). However, some gesture commands may use the size and orientation to specify

parameters; gesture recognizers must be able to recognize such gestures in whatever size and orientation they occur. Kim [67] discusses augmenting a handwriting recognition system so as to allow it to recognize some gestures independently of their size and orientation. Chapter 3 discusses the approach taken here toward the same end.

- Gestures can have a dynamic component. Handwriting systems usually view the input character as a static picture. In a gesture-based system, the same stroke may have different meanings if drawn left-to-right, right-to-left, quickly, or slowly. Gesture recognizers may use such directional and temporal information in the recognition process.

In summary, gestures may potentially deal in dimensions other than the two commonly used in handwriting, be drawn from unusual alphabets, specify entire commands, vary in size and orientation, and have a dynamic component. Thus, while ideas from on-line handwriting recognition algorithms may be used for gesture recognition, handwriting recognizers generally rely on assumptions that make them inadequate for gesture recognition. The ideal gesture recognition algorithm should be adaptable to new gestures, dimensions, additional features, and variations in size and orientation, and should produce parametric information in addition to a classification. Unfortunately, the price for this generality is the likelyhood that a gesture recognizer, when used for handwriting recognition, will be less accurate than a recognizer built and tuned specifically for handwriting recognition.

1.9 Motivation for this Research

In spite of the potential advantages of gesture-based systems, only a handful have been built. Examples include Button Box [86], editing using proofreader's symbols [25], the Char-rec note-input tool [21], and a spreadsheet application built at IBM [109]. These and other gesture-based systems are discussed in section 2.2. Gesture recognition in most existing systems has been done by writing code to recognize the particular set of gestures used by the system. This code is usually complicated, making the systems (and the set of gestures accepted) difficult to create, maintain, and modify. These difficulties are the reasons more gesture-based systems have not been built.

One goal of the present work is to eliminate hand-coding as the way to create gesture recognizers. Instead, gesture classes are specified by giving examples of gestures in the class. From these examples, recognizers are automatically constructed. If a particular gesture class is to be recognized in any size or orientation, its examples of the class should reflect that. Similarly, by making all of the examples of a given class the same size or orientation, the system learns that gestures in this class must appear in the same size or orientation as the examples. The first half of this dissertation is concerned with the automatic construction of gesture recognizers.

Even given gesture recognition, it is still difficult to build direct-manipulation systems which incorporate gestures. This is the motivation for the second half of this dissertation, which describes GRANDMA–Gesture Recognizers Automated in a Novel Direct Manipulation Architecture.

1.10 Criteria for Gesture-based Systems

The goal of this research was to produce tools which aid in the construction of gesture-based systems. The efficacy of the tools may be judged by how well the tools and resulting gesture-based systems satisfy the following criteria.

1.10.1 Meaningful gestures must be specifiable

A meaningful gesture may be rather complex, involving simultaneous motions of a number of points. These complex gestures must be easily specifiable. Two methods of specification are possible: specification by example, and specification by description. In the former, each application has a training session in which examples of the different gestures are submitted to the system. The result of the training is a representation for all gestures that the system must recognize, and this representation is used to drive the actual gesture recognizer that will run as part of the application. In the latter method of specification, a description of each gesture is written in a gesture description language, which is a formal language in which the "syntax" of each gesture is specified. For example, a set of gestures may be specified by a context-free grammar, in which the terminals represent primitive motions (e.g. "straight line segment") and gestures are non-terminals composed of terminals and other non-terminals.

All else being equal, the author considers specification by example to be superior to specification by description. In order to specify gestures by description, it will be necessary for the specifier to learn a description language. Conversely, in order to specify by example, the specifier need only be able to gesture. Given a system in which gestures are specified by example, the possibility arises for end users to train the system directly, either to replace the existing gestures with ones more to their liking, or to have the system improve its recognition accuracy by adapting to the particular idiosyncrasies of a given user's gestures.

One potential drawback of specification by example is the difficulty in specifying the allowable variations between gestures of a given class. In a description language, it can be made straightforward to declare that gestures of a given class may be of any size or of any orientation. The same information might be conveyed to a specify-by-example system by having multiple examples of a single class vary in size or orientation. The system would then have to *infer* that the size or orientation of a given gesture class was irrelevant to the classification of the gesture. Also, training classifiers may take longer, and recognition may be less accurate, when using examples as specifications, though this is by no means necessarily so. Similar issues arise in demonstrational interfaces [97].

1.10.2 Accurate recognition

An important characterization of a gesture recognition system will be the frequency with which gestures fail to be recognized or are recognized incorrectly. Obviously it is desirable that these numbers be made as small as possible. Questions pertaining to the amount of inaccuracy acceptable to people are difficult to answer objectively. There will likely be tradeoffs between the complexity of gestures, the number of different gestures to be disambiguated, the time needed for recognition, and the accuracy of recognition.

In speech recognition there is the problem that the accuracy of recognition decreases as the user population grows. However the analogous problem in gesture recognition is not as easy to gauge. Different people speak the same words differently due to inevitable differences in anatomy and upbringing. The way a person says a word is largely determined before she encounters a speech recognition system. By contrast, most people have few preconceptions of the way to gesture at a machine. People will most likely be able to adapt themselves to gesturing in ways the machine understands. The recognition system may similarly adapt to each user's gestures. It would be interesting, though outside the scope of this dissertation, to study the fraction of incorrectly recognized gestures as a function of a person's experience with the system.

1.10.3 Evaluation of accuracy

It should be possible for a gesture-based system to monitor its own performance with respect to accuracy of recognition. This is not necessarily easy, since in general it is impossible to know which gesture the user had intended to make. A good gesture-based system should incorporate some method by which the user can easily inform the system when a gesture has been classified incorrectly. Ideally, this method should be integrated with the undo or abort features of the systems. (Lerner [78] gives an alternative in which subsequent user actions are monitored to determine when the user is satisfied with the results of system heuristics.)

1.10.4 Efficient recognition

The goal of this work is to enable the construction of applications that use gestures as input, the idea being that gesture input will enhance human/computer interaction. Speed of recognition is very important–a slow system would be frustrating to use and hinder rather than enhance interaction

Speed is a very important factor in the success or failure of user interfaces in general. Baecker and Buxton [5] state that one of the chief determinants of user satisfaction with interactive computer systems is response time. Poor performance in a direct-manipulation system is particularly bad, as any noticeable delay destroys the feeling of directness. Rapid recognition is essential to the success of gesture as a medium for human-computer interaction, even if achieving it means sacrificing certain features or, perhaps, a limited amount of recognition accuracy.

1.10.5 On-line/real-time recognition

When possible, the recognition system should attempt to match partial inputs with possible gestures. It may also be desirable to inform the user as soon as possible when the input does not seem to match any possible gesture. An on-line/real-time matching algorithm has these desirable properties. The gesture recognition algorithms discussed in Chapters 3, 4, and 5 all do a small, bounded amount of work given each new input point, and are thus all on-line/real-time algorithms.

1.10.6 General quantitative application interface

An application must specify what happens when a gesture is recognized. This will often take the form of a callback to an application-specific routine. There is an opportunity here to relay the

parametric data contained in the gesture to the application. This includes the parametric data which can be derived when the gesture is first recognized, as well as any manipulation data which follows.

1.10.7 Immediate feedback

In certain applications, it is desirable that the application be informed immediately once a gesture is recognized but before it is completed. An example is the turning of a knob: once the system recognizes that the user is gesturing to turn a knob it can monitor the exact details of a gesture, relaying quantitative data to the application. The application can respond by immediately and continuously varying the parameter which the knob controls (for example the volume of a musical instrument).

1.10.8 Context restrictions

A gesture sensing system should be able, within a single application, to sense different sets of gestures in different contexts. An example of a context is a particular area of the display screen. Different areas could respond to different sets of gestures. The set of gestures to which the application responds should also be variable over time–the application program entering a new mode could potentially cause a different set of gestures to be sensed.

The idea of contexts is closely related to the idea of using gestures to manipulate graphic objects. Associated with each picture of an object on the screen will be an area of the screen within which gestures refer to the object. A good gesture recognition system should allow the application program to make this association explicit.

1.10.9 Efficient training

An ideal system would allow the user to experiment with different gesture classes, and also adapt to the user's gestures to improve recognition accuracy. It would be desirable if the system responded immediately to any changes in the gesture specifications; a system that took several hours to retrain itself would not be a good platform for experimentation.

1.10.10 Good handling of misclassifications

Misclassifications of gestures are a fact of life in gesture-based systems. A typical system might have a recognition rate of 95% or 99%. This means one out of twenty or one out of one hundred gestures will be misunderstood. A gesture-based system should be prepared to deal with the possibility of misclassification, typically by providing easy access to abort and undo facilities.

1.10.11 Device independence

Certain assumptions about the form of the input data are necessary if gesture systems are to be built. As previously stated, the assumption made here is that the input device will supply position as a function of time for each input "path" (or supply data from which it is convenient to calculate such positions). (A path may be thought of as a continuous curve drawn by a single finger.) This form of

data is supplied by the Sensor Frame, and (at least for the single finger case) a mouse and a clock can be made to supply similar data. The recognition systems should do their recognition based on the position versus time data; in this way other input devices may also benefit from this research.

1.10.12 Device utilization

Each particular brand of input hardware used for gesture sensing will have characteristics that other brands of hardware will not have. It would be unfortunate not to take advantage of all the special features of the hardware. For example, the Sensor Frame can compute finger angle and finger velocity.[7] While for device independence it may be desirable that the gesture matching not depend on the value of these inputs, there should be some facility for passing these parameters to the application specific code, if the application so desires. Baecker [4] states the case strongly: "Although portability is facilitated by device-independence, interactivity and usability are enhanced by *device dependence*."

1.11 Outline

The following chapter describes previous related work in gesture-based systems. This is divided into four sections: Section 2.1 discusses various hardware devices suitable for gestural input. Section 2.2 discusses existing gesture-based systems. Section 2.3 reviews the various approaches to pattern recognition in order to determine their potential for gesture recognition. Section 2.4 examines existing software systems and toolkits that are used to build direct-manipulation interfaces. Ideas from such systems will be generalized in order to incorporate gesture recognition into such systems.

Everything after Chapter 2 focuses on various aspects of the gesture-based interface creation tool built by the author. Such a tool makes it easy to 1) specify and create classifiers, and 2) associate gestures classes and their meanings with graphic objects. The former goal is addressed in Chapters 3, 4, and 5, the latter in 6 and 7.

The discussion of the implementation of gesture recognition begins in Chapter 3. Here the problem of classifying single-path, two-dimensional gestures is tackled. This chapter assumes that the start and end of the gesture are known, and uses statistical pattern recognition to derive efficient gesture classifiers. The training of such classifiers from example gestures is also covered.

Chapter 3 shows how to classify single-path gestures; Chapter 4 shows *when*. This chapter addresses the problem of recognizing gestures while they are being made, without any explicit indication of the end of the gesture. The approach taken is to define and construct another classifier. This classifier is intended solely to discriminate between ambiguous and unambiguous subgestures.

Chapter 5 extends the statistical approach to the recognition of multiple-path gestures. This is useful for utilizing devices that can sense the positions of multiple fingers simultaneously, in particular the Sensor Frame.

Chapter 6 presents the architecture of an object-oriented toolkit for the construction of direct-manipulation systems. Like many other systems, this architecture is based on the Model-View-

[7] This describes the Sensor Frame as originally envisioned. The hardware is capable of producing a few bits of finger velocity and angle information, although to date this has not been attempted.

Controller paradigm. Compared to previous toolkits, the input model is considerably generalized in preparation for the incorporation of gesture recognition into a direct-manipulation system. The notion of virtual tools, through which input may be generated by software objects in the same manner as by hardware input devices, is introduced. Semantic feedback will be shown to arise naturally from this approach.

Chapter 7 shows how gesture recognizers are incorporated into the direct-manipulation architecture presented in Chapter 6. A gesture handler may be associated with a particular view of an object on the screen, or at any level in the view hierarchy. In this manner, different objects will respond to different sets of gestures. The communication of parametric data from gesture handler to application is also examined.

Chapter 8 discusses three gesture-based systems built using these techniques: GDP, GSCORE, and MDP. The first two, GDP and GSCORE, use mouse gestures. GDP, as already mentioned, is the drawing editor based on DP. GSCORE is a musical score editor, based on Buxton's SSSP work [21]. MDP is also a drawing editor, but it operates using multi-path gestures made with a Sensor Frame. The design and implementation of each system is discussed, and the gestures for each shown.

Chapter 9 evaluates a number of aspects of this work. The particular recognition algorithms are tested for recognition accuracy. Measurements of the performance of the gesture classifiers used in the applications is presented. Then, an informal user study assessing the utility of gesture-based systems is discussed.

Finally, Chapter 10 concludes this dissertation. The contributions of this dissertation are discussed, as are the directions for future work.

1.12 What Is Not Covered

This dissertation attempts to cover many topics relevant to gesture-based systems, though by no means all of them. In particular, the issues involved in the ergonomics and suitability of gesture-based systems applied to various task domains have not been studied. It is the opinion of the author that such issues can only be studied after the tools have been made available which allow easy creation of and experimentation with such systems. The intent of the current work is to provide such tools. Future research is needed to determine how to use the tools to create the most usable gesture-based systems possible.

Of course, choices have had to be made in the implementation of such tools. By avoiding the problem of determining which kind of gesture-based systems are best, the work opens itself to charges of possibly "throwing the baby out with the bath-water." The claim is that the general system produced is capable of implementing systems comparable to many existing gesture-based systems; the example applications implemented (see Chapter 8) support this claim. Furthermore, the places where restrictive choices have been made (*e.g.* two-dimensional gestures) have been indicated, and extensible and scalable methods (*e.g.* linear discrimination) have been used wherever possible.

There are two major limitations of the current work. The first is that single-path multi-stroke gestures (*e.g.* handwritten characters) are not handled. Most existing gesture-based systems use single-path multi-stroke gestures. The second limitation is that the start of a gesture must be

explicitly indicated. This rules out (at least at first glance) using devices such as the DataGlove which lack buttons or other explicit signaling hardware. However, one result of the current work is that these apparent limitations give rise to certain advantages in gestural interfaces. For example, the limitations enforce Buxton's notion of tension and release mentioned above.

Gestural output, *i.e.* generating a gesture in response to a query, is also not covered. For an example of gestural output, ask the author why he has taken so long to complete this dissertation.

Chapter 2

Related Work

This chapter discusses previous work relevant to gesture recognition. This includes hardware devices suitable for gestural input, existing gesture-based systems, pattern recognition techniques, and software systems for building user interfaces.

Before delving into details, it is worth mentioning some general work that attempts to define gesture as a technique for interacting with computers. Morrel-Samuels [87] examines the distinction between gestural and lexical commands, and then further discusses problems and advantages of gestural commands. Wolf and Rhyne [140] integrate gesture into a general taxonomy of direct manipulation interactions. Rhyne and Wolf [109] discuss in general terms human-factors concerns of gestural interfaces, as well as hardware and software issues.

The use of gesture as an interaction technique is justified in a number of studies. Wolf [139] performed two experiments that showed gestural interfaces compare favorably to keyboard interfaces. Wolf [141] showed that many different people naturally use the same gestures in a text-editing context. Hauptmann [49] demonstrated a similar result for an image manipulation task, further showing that people prefer to combine gesture and speech rather than use either modality alone.

2.1 Input Devices

A number of input devices are suitable for providing input to a gesture recognizer. This section concentrates on those devices which provide the position of one or more points over time, or whose data is easily converted into that representation. The intention is to list the types of devices which can potentially be used for gesturing. The techniques developed in this dissertation can be applied, directly or with some generalization, to the devices mentioned.

A large variety of devices may be used as two-dimensional, single-path gesturing devices. Some graphical input devices, such as mice [33], tablets and styli, light pens, joysticks, trackballs, touch tablets, thumb-wheels, and single-finger touch screens [107, 124], have been in common use for years. Less common are foot controllers, knee controllers, eye trackers [12], and tongue-activated joysticks. Each may potentially be used for gestural input, though ergonomically some are better suited for gesturing than others. Baecker and Buxton [5], Buxton [14], and Buxton, Hill and Rowley [18] discuss the suitability of many of the above devices for various tasks. Buxton further points out

that two different joysticks, for example, may have very different properties that must be considered with respect to the task.

For gesturing, as with pointing, it is useful for a device to have some signaling capability in addition to the pointer. For example, a mouse usually has one or more buttons, the pressing of which can be used to indicate the start of a gesture. Similarly, tablets usually indicate when the stylus makes or breaks contact with the tablet (though with a tablet it is not possible to carefully position the screen cursor before contact). If a device does not have this signaling capacity, it will be necessary to simulate it somehow. Exactly how this is done can have a large impact on whether or not the device will be suitable for gesturing.

The 3SPACE Isotrack system, developed by Polhemus Navigation Sciences Division of McDonnel Douglas Electronics Company [32], is a device which measures the position and orientation of a stylus or a one-inch cube using magnetic fields. The Polhemus sensor, as it is often called, is a full six-degree-of-freedom sensor, returning x, y, and z rectangular coordinates, as well as azimuth, altitude, and roll angles. It is potentially useful for single path gesturing in three positional dimensions. By considering the angular dimensions, 4, 5, or 6 dimensional gestures may be entered. It is also possible to use one of the angular dimensions for signaling purposes.

Bell Laboratories has produced prototypes of a clear plate capable of detecting the position and pressure of many fingers [10, 99]. The position information is two-dimensional, and there is a third dimension as well: finger pressure. The author has seen the device reliably track 10 fingers simultaneously. The pressure detection may be used for signaling purposes, or as a third dimension for gesturing. The inventor of the multi-finger touch plate has invented another device, the Radio Drum [11], which can sense the position of multiple antennae in three dimensions. To date, the antennae have been embedded in the tips of drum sticks (thus the name), but it would also be possible to make a glove containing the antenna which would make the device more suitable for detecting hand gestures.

The Sensor Frame [84] is a frame mounted on a workstation screen (figure 2.1). It consists of a light source (which frames the screen) and four optical sensors (one in each corner). The Sensor Frame computes the two-dimensional positions of up to three fingertips in a plane parallel to, and slightly above the screen. The net result is similar to a multi-finger touch screen. The author has used the Sensor Frame to verify the multi-finger recognition algorithm described in Chapter 5. The Sensor Cube [85] is a device similar to the Sensor Frame but capable of sensing finger positions in three dimensions. It is currently under construction. The VideoHarp [112, 111] is a musical instrument based on the same sensing technology, and is designed to capture parametric gestural data.

The DataGlove [32, 130] is a glove worn on the hand able to produce the positions of multiple fingers as well as other points on the hand in three dimensions. By itself it can only output relative positions. However, in combination with the Polhemus sensor, absolute finger positions can be computed. Such a device can translate gestures as complex as American Sign Language [123] into a multi-path form suitable for processing. The DataGlove, the similar Dexterous Hand Master from Exos, and the Power Glove from Mattel, are shown in figure 2.2.

The DataGlove comes with hardware which may be trained to recognize certain static configurations of the glove. For example, the DataGlove hardware might be trained to recognize a fist,

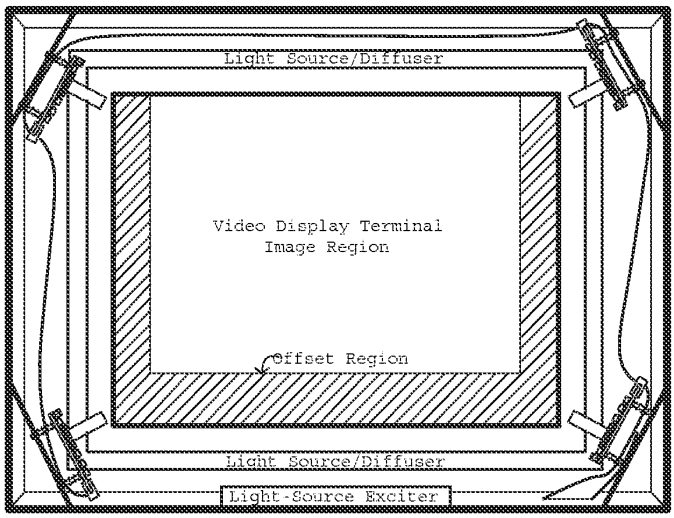


Figure 2.1: The Sensor Frame

The Sensor Frame is a frame mounted on a computer display consisting of a rectangular light source and four sensors, one in each corner. It is capable of detecting up to three fingers its field of view. (Drawing by Paul McAvinney.)

Figure 2.2: The DataGlove, Dexterous Hand Master, and PowerGlove (from Eglowstein [32])

The DataGlove, Dexterous Hand Master, and PowerGlove are three glove-like input devices capable of measuring the angles of various hand and finger joints.

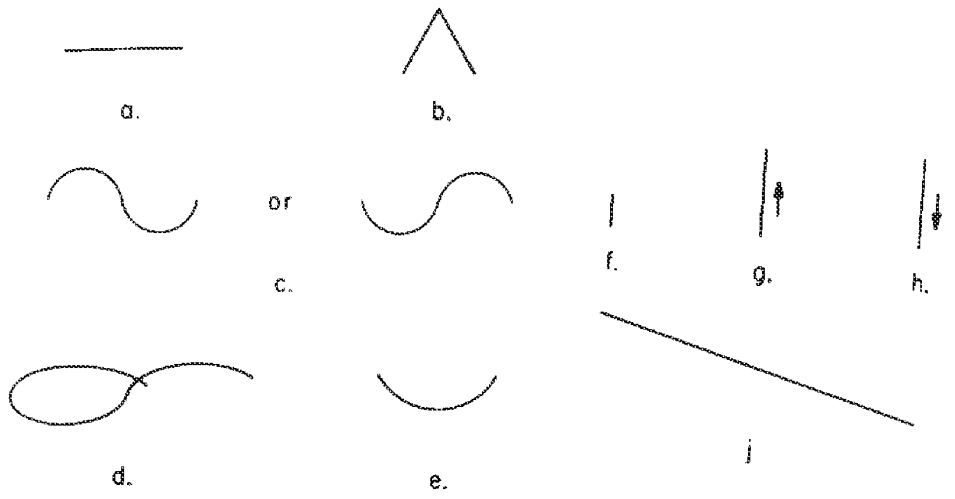


Figure 2.3: Proofreading symbols (from Coleman [25])
The operations intended by each are as follows: a) delete text (from a single line), b) insert text, c) swap text, d) move text, e) join (delete space), f) insert space, g) scroll up, h) scroll down, and i) delete multiple lines of text. Many of the marks convey additional parameters to the operation, e.g. *the text to be moved or deleted.*

signaling the host computer whenever a fist is made. These static hand positions are not considered to be gestures, since they do not involve motion. The glove hardware recognizes "posture" rather than gesture, the distinction being that posture is a static snapshot (a pose), while gesture involves motion over time. Nonetheless, it is a rather elegant way to add signaling capability to a device without buttons or switches.

The Videodesk [71, 72] is an input device based on a constrained form of video input. The Videodesk consists of a translucent tablecloth over a glass top. Under the desk is a light source, over the desk a video camera. The user's hands are placed over the desk. The tablecloth diffuses the light, the net effect being that the camera receives an image of the silhouette of the hands. Additional hardware is used to detect and track the user's fingertips.

Some researchers have investigated the attachment of point light sources to various points on the body or hand to get position information as a function of time. The output of a camera (or pair of cameras for three dimensional input) can be used as input to a gesture sensor.

2.2 Example Gesture-based Systems

This section describes a number of existing gesture-based systems that have been described in the literature. A system must both classify its gestural input and use information other than the class (*i.e.* parametric information) to be included in this survey. The order is roughly chronological.

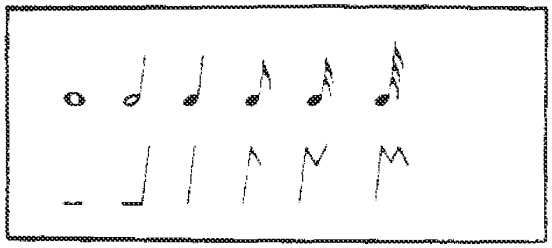


Figure 2.4: Note gestures (from Buxton [21])
A single gesture indicates note duration (from the shape of the stroke as shown) as well as pitch and starting time, both of which are determined from the position of the start of the gesture.

Coleman [25] has created a text editor which used hand-drawn proofreader's symbols to specify editing commands (figure 2.3). For example, a sideways "S" indicated that two sets of characters should be interchanged, the characters themselves being delimited by the two halves of the "S." The input device was a touch tablet, and the gesture classification was done by a hand-coded discrimination net (*i.e.* a loop-free flowchart).[1]

Buxton [21] has built a musical score editor with a small amount of gesture input using a mouse (figure 2.4). His system used simple gestures to indicate note durations and scoping operations. Buxton considered this system to be more a character recognition system than a gesture-based system, the characters being taken from an alphabet of musical symbols. Since information was derived not only from the classification of the characters, but their positions as well, the author considers this to be a gesture-based system in the true sense. Buxton's technique was later incorporated into Notewriter II, a commercial music scoring program. Lamb and Buckley [76] describe a gesture-based music editor usable by children.

Margaret Minsky [86] implemented a system called Button Box, which uses gestures for selection, movement, and path specification to provide a complete Logo programming environment (figure 2.5). Her input device was a clear plate mounted in front of a display. The device sensed the position and shear forces of a single finger touching the plate. Minksy proposed the use of multiple fingers for gesture input, but never experimented with an actual multiple-finger input device.

In Minsky's system, buttons for each Logo operation were displayed on the screen. Tapping a button caused it to execute; touching a button and dragging it caused it to be moved. The classification needed to distinguish between a touch and a tap was programmed by hand. There were buttons used for copying other buttons and for grouping sets of buttons together. A path could be drawn through a series of buttons--touching the end of a path caused its constituent buttons to execute sequentially.

VIDEOPLACE [72] is a system based on the Videodesk. As stated above, the silhouette of the user's hands are monitored. When a hand is placed in a pointing posture, the tip of the index finger

[1]Curiously, this research was done while Coleman was a graduate student at Carnegie Mellon. Coleman apparently never received a Ph.D. from CMU, and it would be twenty years before another CMU graduate student (me) would go near the topic of gesture recognition.

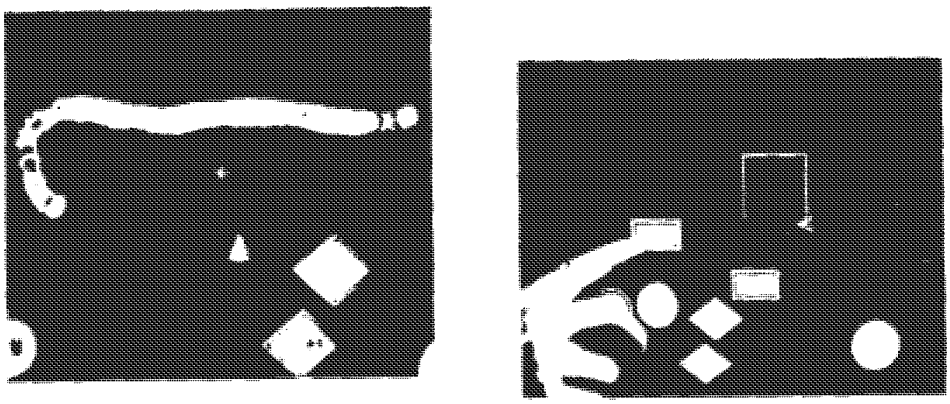


Figure 2.5: Button Box (from Minksy [86])

Tapping a displayed button causes it to execute its assigned function while touching a button and dragging it causes it to be moved.

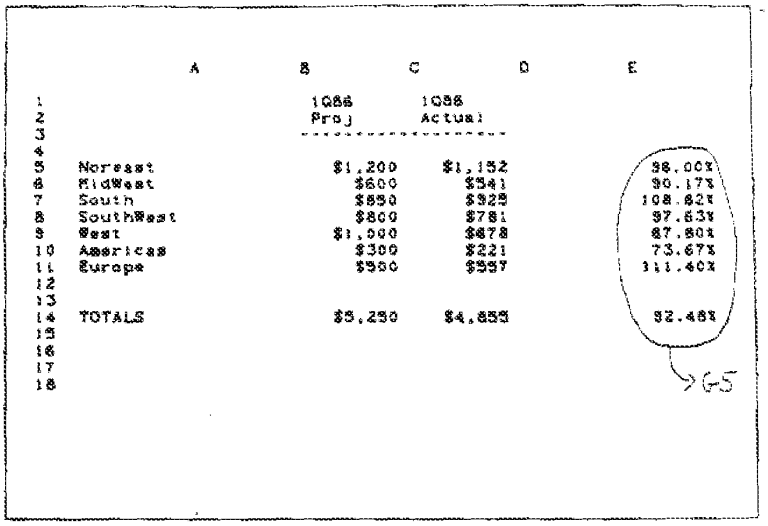


Figure 2.6: A gesture-based spreadsheet (from Rhyne and Wolf [109])

The Paper-Like Interface project produces systems which combine gesture and handwriting. The input shown here selects a group of cells and requests they be moved to the cell beginning at location "G5."

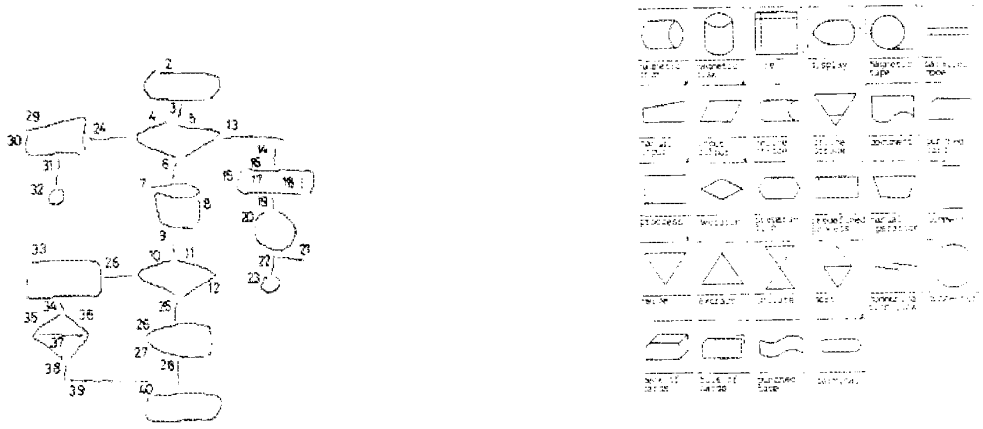


Figure 2.7: Recognizing flowchart symbols

Recognizing flowchart symbols (from Murase and Wakahara [89]). The system takes an entire freehand drawing of a flowchart (left) and recognizes the individual flowchart symbols (right), producing an internal representation of the flowchart (as nodes and edges) and a flowchart picture in which the freehand symbols are replaced by machine generated line-drawings drawn from the alphabet of symbols. This system shows a style of interface in which pattern recognition is used for something other than the detection of gestures or characters.

may be used for menu selection. After selection, the fingertips may be used to manipulated graphic objects, such as the controlling points of a spline curve.

A group at IBM doing research into gestural human-computer systems has produced a gesture-based spreadsheet application [109]. Somewhat similar to Coleman's editor, the user manipulates the spreadsheet by gesturing with a stylus on a tablet (figure 2.6). For example, deletion is done by drawing an "X" over a cell, selection by an "0", and moving selected cells by an arrow, the tip of which indicates the destination of the move. The application is interesting in that it combines handwriting recognition (isolated letters and numbers) with gesturing. For example, by using handwriting the user can enter numbers or text into a cell without using a keyboard. The portion of the recognizer which classifies letters, numbers, and gestures of a fixed size and orientation has (presumably) been trained by example using standard handwriting recognition techniques. However, the recognition of gestures which vary in size or orientation requires hand coding [67].

Murase and Wakahara [89] describe a system in which freehand-drawn flowcharts symbols are recognized by machine (figure 2.7). Tamura and Kawasaki [128] have a system which recognizes sign-language gestures from video input (figure 2.8).

HITS from MCC [55] and Artkit from the University of Arizona [52] are both systems that may be used to construct gesture-based interfaces. The author has seen a system built with HITS similar

Figure 2.8: Sign language recognition (from Tamura [128])
This system processes an image from a video camera in order to recognize a form of Japanese sign language.

to that of Murase; in it an entire control panel is drawn freehand, and then the freehand symbols are segmented, classified and replaced by icons. (Similar work is discussed by Martin, *et. al* [82], also from MCC.) Artkit has much in common with the GRANDMA system described in this dissertation, and will be mentioned again later (Sections 4.1 and 6.8). Artkit systems tend to be similar to those created using GRANDMA, in that gesture commands are executed as soon as they are entered.

Kurtenbach and Buxton [75] have implemented a drawing program based on single-stroke gestures (figure 2.9). They have used the program to study, among other things, issues of scope in gestural systems. To the present author, GEdit's most interesting attribute is the use of compound gestures, as shown in the figure. GEdit's gesture recognizer is hand-coded.

The Glove-talk system [34] uses a DataGlove to control a speech synthesizer (figure 2.10). Like Artkit and the work described in Chapter 4, Glove-talk performs eager recognition: a gesture is recognized and acted upon without its end being indicated explicitly. Weimer and Ganapathy [136] describe a system combining DataGlove gesture and speech recognition.

The use of the circling gesture as an alternative means of selection is considered in Jackson and Roske-Hofstrand [61]. In their system, the start of the circling gesture is detected automatically, *i.e.* the mouse buttons are not used. Circling is also used for selection in the JUNO system from Xerox Corporation [142].

A number of computer products offer a stylus and tablet as their sole or primary input device. These systems include GRID Systems Corp.'s GRIDPad [50], Active Book Company's new portable [43], Pencept Inc.'s computer [59], Scenario's DynaWriter, Toshiba's PenPC, Sony's Palmtop, Mometa's laptop, MicroSlate's Datalite, DFM System's TraveLite, Agilis Corp.'s system, and Go Corp.'s PenPoint system [81, 24]. While details of the interface of many of these systems are hard to find (many of these systems have not yet been released), the author suspects that many use gestures. For further reading, please see [16, 106, 31].

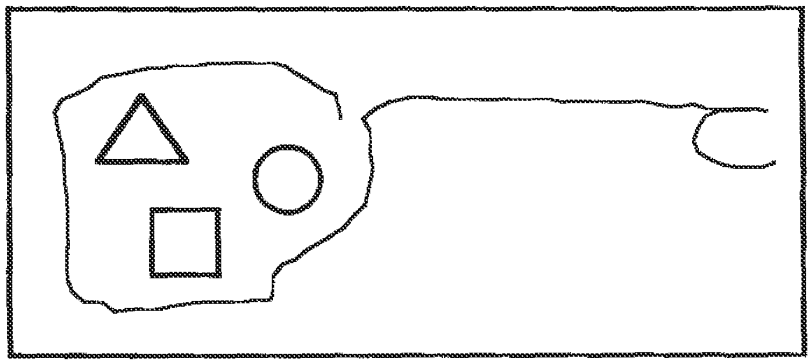


Figure 2.9: Copying a group of objects in GEdit (from Kurtenbach and Buxton [75])
Note the compound gesture: the initial closed curve does selection, and the final "C" indicates the data should be copied rather than moved.

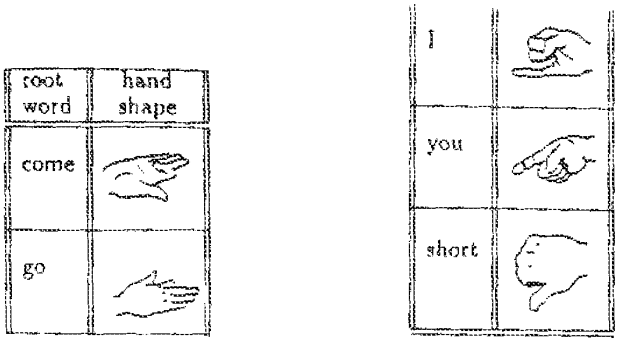


Figure 2.10: GloveTalk (from Fels and Hinton [34])
GloveTalk connects a DataGlove to a speech synthesizer through several neural networks. Gestures indicate root words (shown) and modifiers. Reversing the direction of the hand motion causes a word to be emitted from the synthesizer as well as indicating the start of the next gesture.

Tap	Select/Invoke
Press-hold	Initiate drag (move, wipe-through)
Tap-hold	Initiate drag (copy)
Flick (four directions)	Scroll/Browse
Cross out	Delete
Scratch out	Delete
Circle	Edit
Check	Options
Caret	Insert
Brackets	Select object, adjust selection
Pigtail (vertical)	Delete character
Down-right	Insert space

Figure 2.11: Basic PenPoint gestures (from Carr [24])

Recently, prototypes of Go Corporation's PenPoint system have been demonstrated. Each consists of a notebook-sized computer with a flat display. The sole input device is a stylus, which is used for gestures and handwriting on the display itself. Figure 2.11 shows the basic gestures recognized; depending on the context, additional gestures and handwriting can also be recognized. As can be seen, PenPoint gestures may consist of multiple strokes. Although it seems that trainable recognition algorithms are used internally, at the present time the user cannot add any new gestures to the existing set. The hardware is able to sense pen *proximity* (how near the stylus is to the tablet), which is used to help detect the end of multi-stroke gestures and characters. PenPoint applications include a drawing program, a word processor, and a form-based data entry system.

Many of the above systems combine gesture and direct manipulation in the same interface. GEdit, for example, appears to treat mouse input as gestural when begun on the background window, but drags objects when mouse input begins on the object. Almost none combine gesture and direct manipulation in the same interaction. One exception, PenPoint, uses the dot gesture (touching the stylus to the tablet and then not moving until recognition has been indicated) to drag graphic objects. Button Box does something similar for dragging objects. Artkit [52] uses eager recognition, more or less crediting the idea to me.

2.3 Approaches for Gesture Classification

Fu [40] states that "the problem of pattern recognition usually denotes a discrimination or classification of a set of processes or events." Clearly gesture recognition, in which the input is considered to be an event to be classified as one of a particular set of gestures, is a problem of pattern recognition.

In this dissertation, known techniques of pattern recognition are applied to the problem of sensing gestures.

The general pattern recognition problem consists of two subproblems: pattern representation and decision making [40]. This implies that the architecture of the general pattern recognition consists of two main parts. First, the *representer* takes the raw pattern as input and outputs the internal representation of the pattern. Then, the *decider* takes as input the output of the representer, and outputs a classification (and/or a description) of the pattern.

This section reviews the pattern recognition work relevant to gesture recognition. In particular, the on-line recognition of handwritten characters is discussed whenever possible, since that is the closest solved problem to gesture recognition. For a good overview of handwriting systems in general, see Suen *et. al.* [125] or Tappert *et. al.* [129].

The review is divided into two parts: alternatives for representers and alternatives for deciders. Each alternative is briefly explained, usually by reference to an existing system which uses the approach. The advantages and disadvantages of the alternative are then discussed, particularly as they apply to single-path gesture recognition.

2.3.1 Alternatives for Representers

The representer module takes the raw data from the input device and transforms it into a form suitable for classification by the decider. In the case of single-path gestures, as with on-line handprint, the raw data consists of a sequence of points. The representer outputs *features* of the input pattern.

Representers may be grouped in terms of the kinds of features which they output. The major kinds of features are: templates, global transformations, zones, and geometric features. While a single representer may combine different kinds of features, representers are discussed here as if each only outputs one kind of feature. This will make clearer the differences between the kinds of features. Also, in practice most representers do depend largely on a single kind of feature.

Templates.

Templates are the simplest features to compute: they are simply the input data in its raw form. For a path, a template would simply consist of the sequence of points which make up the path. Recognition systems based on templates require the decider to do the difficult work; namely, matching the template of the input pattern to stored example templates for each class.

Templates have the obvious advantage that the features are simple to compute. One disadvantage is that the size of the feature data grows with the size of the input, making the features unsuitable as input to certain kinds of deciders. Also, template features are very sensitive to changes in the size, location, or orientation of the input, complicating classifiers which attempt to allow for variations of these within a given class. Examples of template systems are mentioned in the discussion of template matching below.

Global Transformations.

Some of the problems of template features are addressed by global transformations of the input data. The transformations are often mathematically defined so as to be invariant under *e.g.* rotation, translation, or scaling of the input data. For example, the Fourier transform will result in features invariant with respect to rotation of the input pattern [46]. Global transformations generally output a fixed number of features, often smaller than the input data.

A set of fixed features allows for a greater variety in the choice of deciders, and obviously the invariance properties allow for variations within a class. Unfortunately, there is no way to "turn off" these invariances in order to disallow intra-class variation. Also, the global transformations generally take as input a two-dimensional raster, making the technique awkward to use for path data (it would have to first be transformed into raster data). Furthermore, the computation of the transformation may be expensive, and the resulting features do not usually have a useful parametric interpretations (in the sense of Section 1.6.2), requiring a separate pass over the data to gather parametric information.

Zones.

Zoning is a simple way of deriving features from a path. Space is divided into a number of zones, and an input path is transformed into the sequence of zones which the path traverses [57]. One variation on this scheme incorporates the direction each zone is entered into the encoding [101]. As with templates, the number of features are not fixed; thus only certain deciders may be used. The major advantage of zoning schemes are their simplicity and efficiency.

If the recognition is to be size invariant, zoning schemes generally require the input to be normalized ahead of time. Making a zoning scheme rotationally invariant is more difficult. Such normalizations make it impossible to compute zones incrementally as the input data is received. Also, small changes to a pattern might cause zones to be missed entirely, resulting in misclassification. And again, the features do not usually hold any useful parametric information.

Geometric Features.

Geometric features are the most commonly used in handwriting recognition [125]. Some geometric features of a path (such as its total length, total angle, number of times it crosses itself, *etc.*) represent global properties of the path. Local properties, such as the sequence of basic strokes, may also be represented.

It is possible to use combinations of geometric features, each invariant under some transformations of the input pattern but not others. For example, the initial angle of a path may be a feature, and all other features might be invariant with respect to rotation of the input. In this fashion, classifiers may potentially be created which allow different variations on a per-class basis.

Geometric features often carry useful parametric information, *e.g.* the total path length, a geometric feature, is potentially a useful parameter. Also, geometric features can be fed to deciders which expect a fixed number of features (if only global geometric features are used), or to deciders which expect a sequence of features (if local features are used).

Geometric features tend to be more complex to compute than the other types of features listed. With care, however, the computation can be made efficient and incremental. For all these reasons, the current work concentrates on the use of global geometric features for the single-path gesture recognition in this dissertation (see Chapter 3).

2.3.2 Alternatives for Deciders

Given a vector or sequence of features output by a representer, it is the job of the decider to determine the class of the input pattern with those features. Seven general methods for deciders may be enumerated: template-matching, dictionary lookup, a discrimination net, statistical matching, linguistic matching, connectionism, and *ad hoc*. Some of the methods are suitable to only one kind of representer, while others are more generally applicable.

Template-matching.

A template-matching decider compares a given input template to one or more prototypical templates of each expected class. Typically, the decider is based on a function which measures the similarity (or dissimilarity) between pairs of templates. The input is classified as being a member of the same class as the prototype to which it is most similar. Usually there is a similarity threshold, below which the input will be rejected as belonging to none of the possible classes.

The similarity metric may be computed as a correlation function between the input and the prototype [69]. Dynamic programming techniques may be used to efficiently warp the input in order to better match up points in the input template to those in the prototype [133, 60, 9].

Template systems have the advantage that the prototypes are simply example templates, making the system easy to train. In order to accommodate large variations, for example in the orientation of a given gesture, a number of different prototypes of various orientation must be specified. Unfortunately, a large number of prototypes can make the use of template matching prohibitively expensive, since the input pattern must be compared to every template.

Lipscomb [80] presents a variation on template matching used for recognizing gestures. In his scheme, each training example is considered at different resolutions, giving rise to multiple templates per example. (The algorithm is thus similar to multiscale algorithms used in image processing [138].) Lipscomb has applied the multiscale technique to stroke data by using an angle filter, in which different resolutions correspond to different thresholds applied to the angles in the gestures. To represent a gesture at a given resolution, points are discarded so that the remaining angles are all below the threshold. To classify an input gesture, first its highest resolution representation is (conceptually) compared to each template (at every resolution). Successively lower resolutions of the input are tried in turn, until an exact match is found. Multiple matches are decided in favor of the template whose resolution is closest to the current resolution of the input.

Dictionary lookup.

When the input features are a sequence of tokens taken from a small alphabet, lookup techniques may be used. This is often how zoning features are classified [101]. The advantage is efficient

recognition, since binary search (or similar algorithms) may be used to lookup patterns in the dictionary. Often some allowance is made for non-exact matches, since otherwise classification is sensitive to small changes in the input. Even with such allowances, dictionary systems are often brittle, due to the features employed (*e.g.* sequences of zones). Of course, a dictionary is initially created from example training input. It is also a simple matter to add new entries for rejected patterns; thus the dictionary system can adapt to a given user.

Discrimination nets.

A discrimination net (also called a decision tree) is basically a flowchart without loops. Each interior node contains a boolean condition on the features, and is connected to two other nodes (a "true" branch and a "false" branch). Each leaf node is labeled with a class name. A given feature set is classified by starting at the root note, evaluating each condition encountered and taking the appropriate branch, stopping and outputting the classification when a leaf node is reached.

Discrimination nets may be created by hand [25], or derived from example inputs [8]. They are more appropriate to classifying fixed-length feature vectors, rather than sequences of arbitrary length, and often result in accurate and efficient classifiers. However, discrimination nets trained by example tend to become unwieldy as the number of examples grows.

Statistical matching.

In statistical matching, the statistics of example feature vectors are used to derive classifiers. Typically, statistical matchers operate only on feature vectors, not sequences. Some typical statistics used are: average feature vector per class, per-class variances of the individual features, and per-class correlations within features. One method of statistical matching is to compute the distance of the input feature vector to the average feature vector of each class, choosing the class which is the closest. Another method uses the statistics to derive per-class discrimination functions over the features. Discrimination functions are like evaluation functions: each discrimination function is applied to the input feature vector, the class being determined by the largest result. Fisher [35] showed how to create discrimination functions which are simply linear combinations of the input features, and thus particularly efficient. Arakawa *et al.* [3] used statistical classification of Fourier features for on-line handwriting recognition; Chapter 3 of the present work uses statistical classification of geometric features.

Some statistical classifiers, such as the Fisher classifier, make assumptions about the distributions of features within a class (such as multivariate normality); those tend to perform poorly when the assumptions are violated. Other classifiers [48] make no such assumptions, but instead attempt to estimate the form of the distribution from the training examples. Such classifiers tend to require many training examples before they function adequately. The former approach is adopted in the current work, with the feature set carefully chosen so as to not violate assumptions about the underlying distribution too drastically.

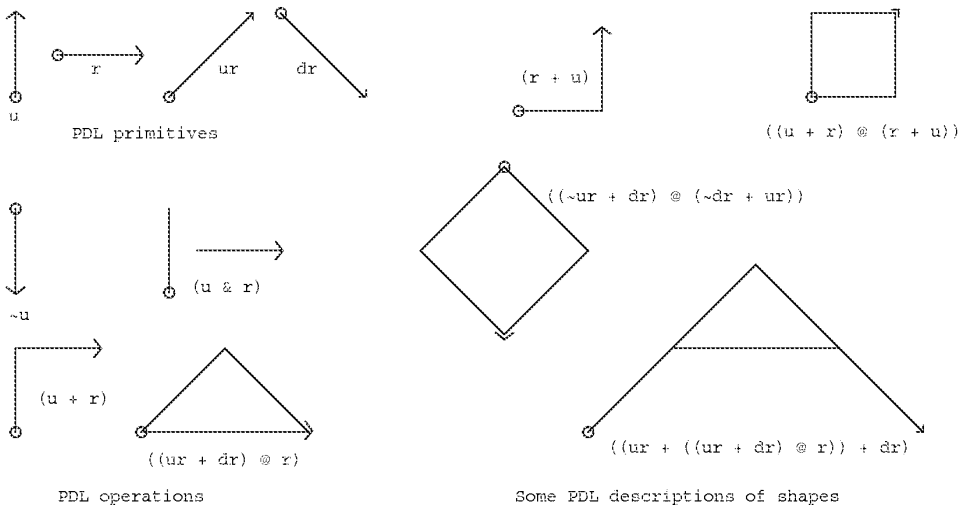


Figure 2.12: Shaw's Picture Description Language

PDL enables line drawings to be coded in string form, making it possible to apply textual parsing algorithms to the recognition of line drawings. The component line segments and combining operations are shown on the left; the right shows how the letter "A" can be described using these primitives.

Linguistic matching.

The linguistic approach attempts to apply automata and formal language theory to the problem of pattern recognition [37]. The representer outputs a sequence of tokens which is composed of a set of pattern primitives and composition operators representing the relation between the primitives. The decider has a grammar for each possible pattern class. It takes as input the sentence and attempts to parse it with respect to each pattern class grammar. Ideally, exactly one of the parses is successful and the pattern is classified thus. A useful side effect of the syntax analysis is the parse tree (or other parse trace) which reveals the internal structure of the pattern.

Linguistic recognizers may be classified based on the form of the representer output. If the output is a string then standard language recognition technology, such as regular expressions and context-free grammars, may be used to parse the input. An error-correcting parser may be used in order to robustly deal with errors in the input. Alternatively, the output of the representer may be a tree or graph, in which case the decider could use graph matching algorithms to do the parse.

The token sequence could come from a zoning representer, a representer based on local geometric properties, or from the output of a lower-level classifier. The latter is a hybrid approach–where, for example, statistical recognition is used to classify paths (or path segments), and linguistic recognition is used to classify based on the relationships between paths. This approach is similar to that taken by Fu in a number of applications [40, 39, 38].

Shaw's picture description language (PDL, see figure 2.12) has been used successfully to describe and classify line drawings [116, 40]. In another system, Stallings [120, 37] uses the composition

operators *left-of*, *above*, and *surrounds* to describe the relationships between strokes of Chinese characters.

A major problem with linguistic recognizers is the necessity of supplying a grammar for each pattern class. This usually represents considerably more effort than simply supplying examples for each class. While some research has been done on automatically deriving grammars from examples, this research appears not to be sufficiently advanced to be of use in a gesture recognition system. Also, linguistic systems are best for patterns with substantial internal structure, while gestures tend to be atomic (but not always [75]).

Connectionism.

Pattern recognition based on neural nets has received much research attention recently [65, 104, 132, 134]. A neural net is a configuration of simple processing elements, each of which is a super-simplified version of a neuron. A number of methods exist for training a neural network pattern recognizer from examples. Almost any of the different kinds of features listed above could serve as input to a neural net, though best results would likely be achieved with vectors of quantitative features. Also, some statistical discrimination functions may be implemented as simple neural networks.

Neural nets have been applied successfully to the recognition of line drawings [55, 82], characters [47], and DataGlove gestures [34]. Unfortunately, they tend to require a large amount of processing power, especially to train. It now appears likely that neural networks will, in the future, be a popular method for gesture recognition. The chief advantage is that neural nets, like template-based approaches, are able to take the raw sensor data as input. A neural network can learn to extract interesting features for use in classification. The disadvantage is that many labeled examples (often thousands) are needed in training.

The statistical classification method discussed in this dissertation may be considered a one-level neural network. It has the advantage over multilayer neural networks, in that it may be trained quickly using relatively few examples per class (typically 15). Rapid training time is important in a system that is used for prototyping gesture-based systems, since it allows the system designer to easily experiment with different sets of gestures for a given application.

Ad hoc methods.

If the set of patterns to be recognized is simple enough, a classifier may be programmed by hand. Indeed, this was the case in many of the gesture-based systems mentioned in Section 2.2. Even so, having to program a recognizer by hand can be difficult and makes the gesture set difficult to modify. The author believes that the difficulty of creating recognizers is one major reason why more gesture-based systems have not been built, and why there is a dearth of experiments which study the effect of varying the gestures in those systems which have been built. The major goal of this dissertation is to make the building of gesture-based systems easy by making recognizers specifiable by example, and incorporating them into an easy-to-use direct manipulation framework.

2.4 Direct Manipulation Architectures

A direct manipulation system is one in which the user manipulates graphical representations of objects in the task domain directly, usually with a mouse or other pointing device. In the words of Shneiderman [117],

> the central ideas seemed to be visibility of the object of interest; rapid, reversible, incremental actions; and replacement of complex command language syntax by direct manipulation of the object of interest–hence the term "direct manipulation."

As examples, he mentions display editors, Visicalc, video games, computer-aided design, and driving an automobile, among others.

For many application domains, the direct manipulation paradigm results in programs which are easy to learn and use. Of course there are tasks for which direct manipulation is not appropriate, due to the fact that the abstract nature of the task domain is not easily mapped onto concrete graphical objects [58]. For example, direct manipulation systems for the abstract task of programming have been rather difficult to design, though much progress has been made [98].

It is not intended here to debate the merits and drawbacks of direct manipulation systems. Instead, it is merely noted that direct manipulation has become an increasingly important and popular style of user interface. Furthermore, all existing gesture-based systems may be considered direct-manipulation systems. The reason is that graphical objects on the screen are natural targets of gesture commands, and updating those objects is an intuitive way of feeding back to the user the effect of his gesturing. In this section, existing approaches for constructing direct manipulation systems are reviewed. In Chapters 6 and 7 it is shown how some of these approaches may be extended to incorporate gestural input.

While direct manipulation systems are easy to use, they are among the most difficult kinds of interface to construct. Thus, there is a great interest in software tools for creating such interfaces. Myers [96] gives an excellent overview of the various tools which have been proposed for this purpose. Here, it is sufficient to divide user-interface software tools into three levels.

The lowest software level potentially seen by the direct manipulation system programmer is usually the *window manager*. Example window managers include X [113], News [127], Sun Windows [126], and Display Postscript [102]; see Myers [94] for an overview. For current purposes, it is sufficient to consider the window manager as providing a set of routines (*i.e.* a programming interface) for both output (textual and graphical) and input (keyboard and mouse or other device). Programming direct manipulation interfaces at the window manager level is a usually avoided, since a large amount of work will likely need to be redone for each application (*e.g.* menus will have to be implemented for each). Building from scratch this way will probably result in different and inconsistent interfaces for each application, making the total system difficult to recall and use.

The next software level is the *user interface toolkit*. Toolkits come in two forms: non-object-oriented and object-oriented. A toolkit provides a set of procedures or objects for constructing menus, scroll bars, and other standard interaction techniques. Most of the toolkits come totally disassembled, and it is up to the programmer to decide how to use the components. Some toolkits, notably MacApp [115] and GWUIMS [118], come partially assembled, making it easier for the programmer to customize the structure to fit the application. For this reason, some authors have

referred to these systems as User Interface Management Systems, though here they are grouped with the other toolkits.

A non-object-oriented toolkit is simply a set of procedures for creating and manipulating the interaction techniques. This saves the programmer the effort involved in programming these interaction techniques directly, and has the added benefit that all systems created using a single toolkit will look and act similarly. One problem with non-object-oriented toolkits is that they usually do not give much support for the programmer who wishes to create new interaction techniques. Such a programmer typically cannot reuse any existing code and thus finds himself bogged down with many low-level details of input and screen management.

Instead of procedures, object-oriented toolkits provide a class (an object type) for each of the standard interaction techniques. To use one of the interaction techniques in an interface, the programmer creates an instance of the appropriate class. By using the inheritance mechanism of the object-oriented programming language, the programmer can create new classes which behave like existing classes except for modifications specified by the programmer. This subclassing gives the programmer a method of customizing each interaction technique for the particular application. It also provides assistance to the programmer wishing to create new interaction techniques–he can almost always subclass an existing class, which is usually much easier than programming the new technique from scratch. One problem with object-oriented toolkits is their complexity; often the programmer needs to be familiar with a large part of the class hierarchy before he can understand the functionality of a single class.

User Interface Management Systems (UIMSs) form the software level above toolkits [96]. UIMSs are systems which provide a method for specifying some aspect of the user interface that is at a higher level than simply using the base programming language. For example, the RAPID/USE system [135] uses state transition diagrams to specify the structure of user input, the Syngraph system [64] uses context-free grammers similarly, and the Cousin system [51] uses a declarative language. Such systems encourage or enforce a strict separation between the user interface specification and the application code. While having modularity advantages, it is becoming increasingly apparent that such a separation may not be appropriate for direct manipulation interfaces [110].

UIMSs which employ *direct graphical specification* of interface components are becoming increasingly popular. In these systems, the UIMS is itself a direct manipulation system. The user interface designer thus uses direct manipulation to specify the components of the direct manipulation interface he himself desires to build. The NeXT Interface Builder [102] and the Andrew Development Environment Workbench (ADEW) [100] allow the placement and properties of existing interface components to be specified via direct manipulation. However, new interface components must be programmed in the object-oriented toolkit provided. In addition to the direct manipulation of existing interface components, Lapidary [93] and Peridot [90] enable new interface components to be created by direct graphical specification.

UIMSs are generally built on top of user interface toolkits. The UIMSs that support the construction of direct manipulation interfaces, such as the ones which use direct graphical specification, tend to be built upon object-oriented toolkits. Since object-oriented toolkits are currently the preferred vehicle for the creation of direct manipulation systems, this dissertation concentrates upon the problem of integrating gesture into such toolkits. In preparation for this, the architectures of

several existing object-oriented toolkits are now reviewed.

2.4.1 Object-oriented Toolkits

The object-oriented approach is often used for the construction of direct manipulation systems. Using object-oriented programming techniques, graphical objects on the screen can be made to correspond quite naturally with software objects internal to the system. The ways in which a graphic object can be manipulated correspond to the messages to which the corresponding software object responds. It is assumed that the reader of this dissertation is familiar with the concepts of object-oriented programming. Cox [27, 28], Stefik and Bobrow [121], Horn [56], Goldberg and Robson [44], and Schmucker [115] all present excellent overviews of the topic.

The Smalltalk-80 system [44] was the first object-oriented system that ran on a personal computer with a mouse and bitmapped display. From this system emerged the Model-View-Controller (MVC) paradigm for developing direct manipulation interfaces. Though MVC literature is only now beginning to appear in print [70, 63, 68], the MVC paradigm has directly influenced every object-oriented user interface architecture since its creation. For this reason, the review of object-oriented architectures for direct manipulation systems begins with a discussion of the use of the MVC paradigm in the Smalltalk-80 system.

The terms "model," "view," and "controller" refer to three different kinds of objects which play a role in the representation of single graphic object in a direct manipulation interface. A *model* is an object containing application specific data. Model objects encapsulate the data and computation of the task domain, and generally make no reference to the user interface.

A *view* object is responsible for displaying application data. Usually, a view is associated with a single model, and communicates with the model in order to acquire the application data that it will render on the screen. A single model may have multiple views, each potentially displaying different aspects of the model. Views implement the "look" of a user interface.

A *controller* object handles user interaction (*i.e.* input). Depending on the input, the controller may communicate directly with a model, a view, or both. A controller object is generally paired with a view object, where the controller handles input to a model and the view handles output. Internally, the controller and view objects typically contain pointers to each other and the associated model, and thus may directly send messages to each other and the model. Controllers implement the "feel" of a user interface.

When the application programmer codes a model object, for modularity purposes he does not generally include references to any particular view(s). The result is a separation between the application (the models) and the user interface (the views and controllers). There does however need to be some connection from a model to a view–otherwise how can the view be notified when the state of the model changes? This connection is accomplished in a modular fashion through the use of *dependencies*.

Dependencies work as follows: Any object may register itself as a dependent of any other object. Typically, a view object, when first created, registers as a dependent of a model object. Generally, there is a list of dependents associated with an object; in this way multiple views may be dependent on a single model. When an object that potentially has dependents changes its state, it sends itself the message `[self changed]`. Each dependent `d` of the object will then get sent the message `[d`

`update`, informing it that an object upon which it is dependent has changed. Thus, dependencies allow a model to communicate to its views the fact that it has changed, without referring to the views explicitly.

Many views display rectangular regions on the screen. A view may have subviews, each of which typically results in an object displayed within the rectangular region of the parent view. The subviews may themselves have subviews, and so on recursively, giving rise to the *view hierarchy*. Typically, a subview's display is clipped so as to wholly appear within the rectangular region of its parent. A subview generally occludes part of its parent's view.

A common criticism of the MVC paradigm is that two objects (the view and controller) are needed to implement the user interface for a model where one would suffice. This, the argument goes, is not only inefficient, but also not modular. Why implement the look and feel separately when in practice they always go together?

The reply to this criticism states that it is useful (often or occasionally) to control look and feel separately [68]. Knolle discusses the usefulness of a single view having several interchangeable controllers; implementing different user abilities (*i.e.* beginning, intermediate, and advanced) with different controllers, and having the system adapt to the user's ability at runtime is one example. While Knolle's examples may not be very persuasive, there is an important application of separating views from controllers, namely, the ability to handle multiple input devices. Chapters 6 and 7 explore further the benefits accrued from the separation of views and controllers.

Nonetheless, there is a simplicity to be had by combining views and controllers into a single object, giving rise to object-oriented toolkits based on the Data-View (DV) paradigm. Though the terminology varies, MacApp [115, 114], the Andrew Toolkit [105], the NeWS Development environment [108], and InterViews [79] all use the DV paradigm. In this paradigm, data objects contain application specific data (and thus are identical to MVC models) while view objects combine the functionality of MVC view and controller objects. In DV systems, the look and feel of an object are very tightly coupled, and detailed assumptions about the input hardware (*e.g.* a three button mouse) get built into every view.

Object-oriented toolkits also vary in the method by which they determine which controller objects get informed of a particular input event, and also in the details of that communication. Typically, input events (such as mouse clicks) are passed down the view hierarchy, with a view querying its subviews (and so on recursively) to see if one of them wishes to handle the event before deciding to handle the event itself. Many variations on this scheme are possible.

Controllers may be written to have methods for messages such as `leftButtonDown`. This style, while convenient for the programmer, has the effect of wiring in details of the input hardware all throughout the system [115, 68]. The NeXT AppKit [102], passes input events to the controller object in a more general form. This is generalized even further in Chapters 6 and 7.

Controllers are a very general mechanism for handling input. Garnet [92], a modern MVC-based system, takes a different approach, called *interactors* [95, 91]. The key insight behind interactors is that there are only several different kinds of interactive behavior, and a (parameterizable) interactor can be built for each. The user-interface designer then needs only to choose the appropriate interactor for each interaction technique he creates.

Gestural input is not currently handled by the existing interactors. It would be interesting to see

if the interactor concept in Garnet is general enough to handle a gesture interactor. Unfortunately, the author was largely unaware of the Garnet project at the time he began the research described in Chapters 6 and 7. Had it been otherwise, a rather different method for incorporating gestures into direct manipulation systems than the one described here might have been created.

The Artkit system [52] has a considerably more general input mechanism than the MVC systems discussed thus far. Like the GRANDMA system discussed in this dissertation, Artkit integrates gesture into an object-oriented toolkit. Though developed simultaneously and independently, Artkit and GRANDMA have startlingly similar input architectures. The two systems will be compared in more detail in chapter 6.

Chapter 3

Statistical Single-Path Gesture Recognition

3.1 Overview

This chapter address the problem of recognizing single-path gestures. A single-path gesture is one that can be input with a single pointer, such as a mouse, stylus, or single-finger touch pad. It is further assumed that the start and end of the input gesture are clearly delineated. When gesturing with a mouse, the start of a gesture might be indicated by the pressing of a mouse button, and the end by the release of the button. Similarly, contact of the stylus with the tablet or of a finger with the touch screen could be used to delineate the endpoints of a gesture.

Baecker and Buxton[5] warn against using a mouse as a gestural input device for ergonomic reasons. For the research described in this chapter, the author has chosen to ignore that warning. The mouse was the only pointing device readily available when the work began. Furthermore, it was the only pointing device that is widely available--an important consideration as it allows others to utilize the present work. In addition, it is probably the case that any trainable recognizer that works well given mouse input could be made to work even better on devices more suitable for gesturing, such as a stylus and tablet.

The particular mouse used is labeled DEC Model VS10X-EA, revision A3. It has three buttons on top, and a metal trackball coming out of the bottom. Moving the mouse on a flat surface causes its trackball to roll. Inside the mouse, the trackball motion is mechanically divided into x and y components, and the mouse sends a pulse to the computer each time one of its components changes by a certain amount. The windowing software on the host implements mouse acceleration, meaning that the faster the mouse is moved a given distance, the farther the mouse cursor will travel on the screen. The metal mouseball was rolled on a Formica table, resulting it what might be termed a "hostile" system for studying gestural input.

All the work described in this chapter was developed on a Digital Equipment Corporation MicroVAX II.[1] The software was written in C [66] and runs on top of the MACH operating system

[1]MicroVAX is trademark of Digital Equipment Corporation.

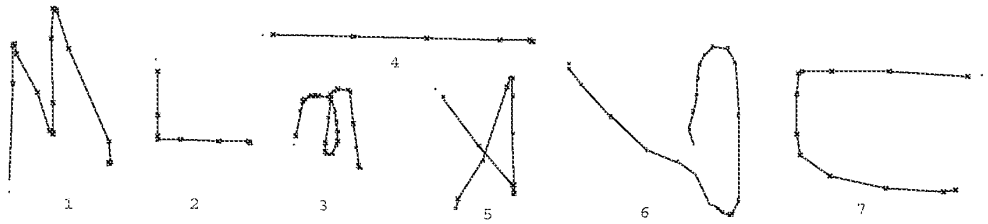


Figure 3.1: Some example gestures

The period indicates the start of the gesture. The actual mouse points that make up the gestures are indicated as well.

[131], which is UNIX[2] 4.3 BSD compatible. X10 [113] was the window system used, though there is a layer of software designed to make the code easy to port to other window systems.

3.2 Single-path Gestures

The gestures considered in this chapter consist of the two-dimensional path of a single point over time. Each gesture is represented as an array g of P time-stamped sample points:

$$g_p = (x_p, y_p, t_p) \qquad 0 \leq p < P.$$

The points are time stamped (the t_p) since the typical interface to many gestural input devices, particularly mice, does not deliver input points at regular intervals. In this dissertation, only two-dimensional gestures are considered, but the methods described may be generalized to the three-dimensional case.

When an input point is very close to the previous input point, it is ignored. This simple preprocessing of the input results in features that are much more reliable, since much of the jiggle, especially at the end of a gesture, is eliminated. The result is a large increase in recognition accuracy.

For the particular mouse used for the majority of this work, "very close" meant within three pixels. This threshold was empirically determined to produce an optimal recognition rate on a number of gesture sets.

Similar, but more complicated preprocessing was done by Leedham, *et. al.*, in their Pittman's shorthand recognition system[77]. The difference in preprocessing in Leedham's system and the current work stems largely from the difference in input devices (Leedham used an instrumented pen), indicating that preprocessing should be done on a per-input-device basis.

Figure 3.1 shows some example gestures used in the GDP drawing editor. The first point (g_0) in each gesture is indicated by a period. Each subsequent point (g_p) is connected by a line segment to the previous point (g_{p-1}). The time stamps are not shown in the figure.

The gesture recognition problem is stated as follows: There is a set of C gesture classes, numbered 0 through $C-1$. The classes may be specified by description, or, as is done in the present

[2]UNIX is a trademark of Bell Laboratories.

work, by example gestures for each class. Given an input gesture g, the problem is to determine the class to which g belongs (*i.e.* the class whose members are most like g). Some classifiers have a reject option: if g is sufficiently different so as not to belong to any of the gesture classes, it should be rejected.

3.3 Features

Statistical gesture recognition is done in two steps. First, a set of features is extracted from the input gesture. This set is represented as a feature vector, $\mathbf{f} = [f_1, \ldots, f_F]'$. (Here and throughout, the prime denotes vector transpose.) The feature vector is then classified as one of the possible gesture classes.

The set of features used was chosen according to the following criteria:

The number of features should be small. In the present scheme, the amount of time it takes to classify a gesture given the feature vector is proportional to the product of the size of the feature vector (*i.e.* the number of features) and the number of different gesture classes. Thus, for efficiency reasons, the number of features should be kept as small as possible while still being able to distinctly represent the different classes.

Each feature should be calculated efficiently. It is essential that the calculation of the feature vector itself not be too expensive: the amount of time to update the value of a feature when an input point g_p is received should be bounded by a constant. In particular, features that require all previous points to be examined for each new input point are disallowed. In this manner, very large gestures (those consisting of many points) are recognized as efficiently as smaller gestures.

In practice, this incremental calculation of features is often achieved by computing auxiliary features not used in classification. For example, if one feature is the average x value of the input points, an auxiliary feature consisting of the sum of the x values might be computed. This would require constant time (one addition) per input point. When the feature vector is needed (for classification) the average x value feature is computed in constant time by dividing the above sum by the number of input points.

Each feature should have a meaningful interpretation. Unlike simple handwriting systems, the gesture-based systems built here use the features not only for classification, but also for parametric information. For example, a drawing program might use the initial angle of a gesture to orient a newly created rectangle. While it is possible to extract such gestural attributes independent of classification, it is potentially less efficient to do so.

Meaningful features also provide useful information to the designer of a set of gesture classes for a particular application. By understanding the set of features, the designer has a better idea of what kind of gestures the system can and cannot distinguish; she is thus more likely to design gestures that can be classified accurately.

Individual features should have Gaussian-like distributions. The classifier described in this chapter is optimal when, among other things, within a given class each feature has a Gaussian distribution. This is because a class is essentially represented by its mean feature vector, and classification of an example takes place, to a first approximation, by determining the class whose mean feature vector is closest to the example's. Classification may suffer if a given feature in a given class has, for example, a bimodal distribution, whereby it tends toward one of two different values.

This requirement is satisfied when the feature is *stable*, meaning a small change in the input gesture results in a small change in the value of the feature. In general, this rules out features that are small integers, since presumably some small change in a gesture will cause a discrete unit step in the feature. When possible, features that depend on thresholds should also be avoided for similar reasons. Ideally, a feature is a real-valued continuous function of the input points.

Note that the input preprocessing is essentially a thresholding operation, and does have the effect that a seemingly small change in the gesture can cause big changes in the feature vector. However, eliminating this preprocessing would allow the noise inherent in the input device to seriously affect certain features. Thus, thresholding should not be ruled out per-se, but the tradeoffs must be considered. Another alternative is to use multiple thresholds to achieve a kind of multiscale representation of the input, thus avoiding problems inherent in using a single threshold [80].

The particular set of features used here evolved over the creation of two classifiers, the first being for a subset of GDP gestures, the second being a recognizer of upper-case letters, as handwritten by the author. In the current version of the recognition program, thirteen features are employed. Figure 3.2 depicts graphically the values used in the feature calculation.

The features are:

Cosine and sine of initial angle with respect to the X axis:

$$f_1 = \cos\alpha = (x_2 - x_0)/d$$
$$f_2 = \sin\alpha = (y_2 - y_0)/d$$

where $d = \sqrt{(x_2 - x_0)^2 + (y_2 - y_0)^2}$.

Length of the bounding box diagonal:

$$f_3 = \sqrt{(x_{max} - x_{min})^2 + (y_{max} - y_{min})^2}$$

where $x_{max}, x_{min}, y_{max}, y_{min}$ are the maximum and minimum values for x_p and y_p respectively.

Angle of the bounding box:

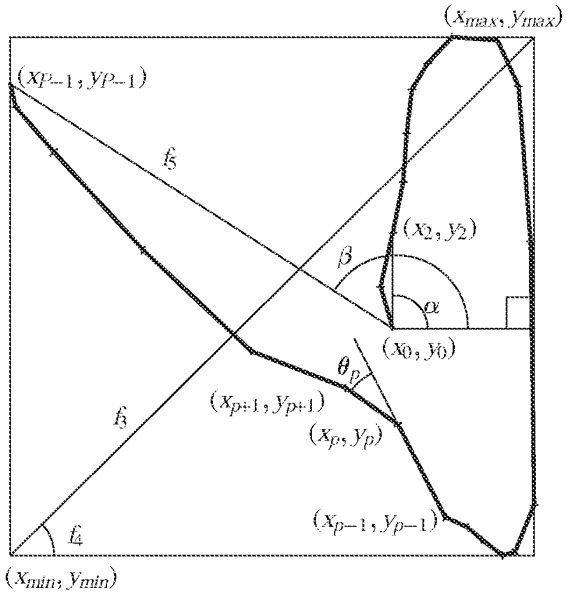


Figure 3.2: Feature calculation

Gesture 6 of figure 3.1 is shown with its relevant lengths and angles labeled with the intermediate variables used to compute features or the features themselves where possible.

$$f_4 = \arctan \frac{y_{max} - y_{min}}{x_{max} - x_{min}}$$

Distance between first and last point:

$$f_5 = \sqrt{(x_{P-1} - x_0)^2 + (y_{P-1} - y_0)^2}$$

Cosine and sine of angle between first and last point:

$$f_6 = \cos\beta = (x_{P-1} - x_0)/\, f_5$$
$$f_7 = \sin\beta = (y_{P-1} - y_0)/\, f_5$$

Total gesture length:

$$\text{Let } \Delta x_p = x_{p+1} - x_p$$
$$\Delta y_p = y_{p+1} - y_p$$

$$f_8 = \sum_{p=0}^{P-2} \sqrt{\Delta x_p^2 + \Delta y_p^2}$$

Total angle traversed (derived from the dot and cross product definitions[73]):

$$\theta_p = \arctan \frac{\Delta x_p \Delta y_{p-1} - \Delta x_{p-1} \Delta y_p}{\Delta x_p \Delta x_{p-1} + \Delta y_p \Delta y_{p-1}}$$

$$f_9 = \sum_{p=1}^{P-2} \theta_p$$

$$f_{10} = \sum_{p=1}^{P-2} |\theta_p|$$

$$f_{11} = \sum_{p=1}^{P-2} \theta_p^2$$

Maximum speed (squared):

$$\Delta t_p = t_{p+1} - t_p$$

$$f_{12} = \max_{p=0}^{P-2} \frac{\Delta x_p^2 + \Delta y_p^2}{\Delta t_p^2}$$

Path duration:

$$f_{13} = t_{P-1} - t_0$$

Features f_{12} and f_{13} allow the gesture recognition to be based on temporal factors; thus gestures have a dynamic component and are not simply static pictures.

Some features (f_1, f_2, f_6, and f_7) are sines or cosines of angles, while others (f_5, f_{10}, f_{11}, f_{12}) depend on angles directly and thus require inverse trigonometric functions to compute. A four-quadrant arctangent is needed to compute θ_p; the arctangent function must take the numerator and denominator as separate parameters, returning an angle between $-\pi$ and π. For efficient recognition, it would be desirable to use just a single feature to represent an angle, rather than both the sine and cosine. However, the recognition algorithm requires that each feature have approximately a Gaussian distribution; this poses a problem when a small change in a gesture causes a large change in angle measurement due to the discontinuity when near $\pm\pi$. This mattered for initial angle, and the angle between the start and end point of the gesture, so each of these angles is represented by its sine and cosine. The bounding box angle is always between 0 and $\pi/2$ so there was no discontinuity problem for it.

For features dependent on θ_i, the angle between three successive input points, the discontinuity only occurs when the gesture stroke turns back upon itself. In practice, likely due to the few gestures used which have such changes, the recognition process has not been significantly hampered by the potential discontinuity (but see Section 9.1.1). The feature f_9 is a measure of the total angle traversed; in a gesture consisting of two clockwise loops, this feature might have a value near 4π. If the gesture was a clockwise loop followed by a counterclockwise loop, f_9 would be close to zero. The feature f_{10} accumulates the absolute value of instantaneous angle; in both loop gestures, its value would be near 4π. The feature f_{11} is a measure of the "sharpness" of gesture.

Figure 3.3 shows the value of some features as a function of p, the input point, for gestures 1 and 2 of figure 3.1. Note in particular how the value for f_{11} (the sharpness) increases at the angles of the gesture. The feature values at the last (rightmost) input point are the ones that are used to classify the gesture. The intent of the graph is to show how the features change with each new input point.

All the features can be computed incrementally, with a constant amount of work being done for each new input point. By utilizing table lookup for the square root and inverse trig functions, the amount of computation per input point can be made quite small.

A number of features were tried and found not to be as good as the features used. For example, instead of the sharpness metric f_{11}, initially a count of the number of times θ_p exceeded a certain threshold was used. The idea was to count sharp angles. While this worked fairly well, the more continuous measure of sharpness was found to give much better results. In general, features that are discrete counts do not work as well as continuous features that attempt to quantify the same phenomena. The reason for this is probably that continuous features more closely satisfy the normality criterion. In other words, an error or deviation in a discrete count tends to be much more significant than an error or deviation in continuous metric.

Appendix A shows the C code for incrementally calculating the feature vector of a gesture.

3.4 Gesture Classification

Given the feature vector $\mathbf{x}$ computed for an input gesture g, the classification algorithm is quite simple and efficient. Associated with each gesture class is a linear evaluation function over the features. Gesture class c has weights $w_i^{\hat{c}}$ for $0 \le i \le F$, where F is the number of features, currently 13. (Per-class variables will be written using superscripts with hats to indicate the class. These are not and should not to be confused with exponentiation.) The evaluation functions are calculated as follows:

$$v^{\hat{c}} = w_0^{\hat{c}} + \sum_{i=1}^{F} w_i^{\hat{c}} x_i \qquad 0 \le c < C \tag{3.1}$$

The value $v^{\hat{c}}$ is the evaluation of class c. The classifier simply determines the c for which $v^{\hat{c}}$ is a maximum; this c is the classification of the gesture g. The possibility of rejecting g is discussed in Section 3.6.

Practitioners of pattern recognition will recognize this classifier as the classic linear discriminator [35, 30, 62, 74]. With the correct choice of weights $w_i^{\hat{c}}$, the linear discriminator is known to be optimal when (1) within a class the feature vectors have a multivariate normal distribution, and (2)

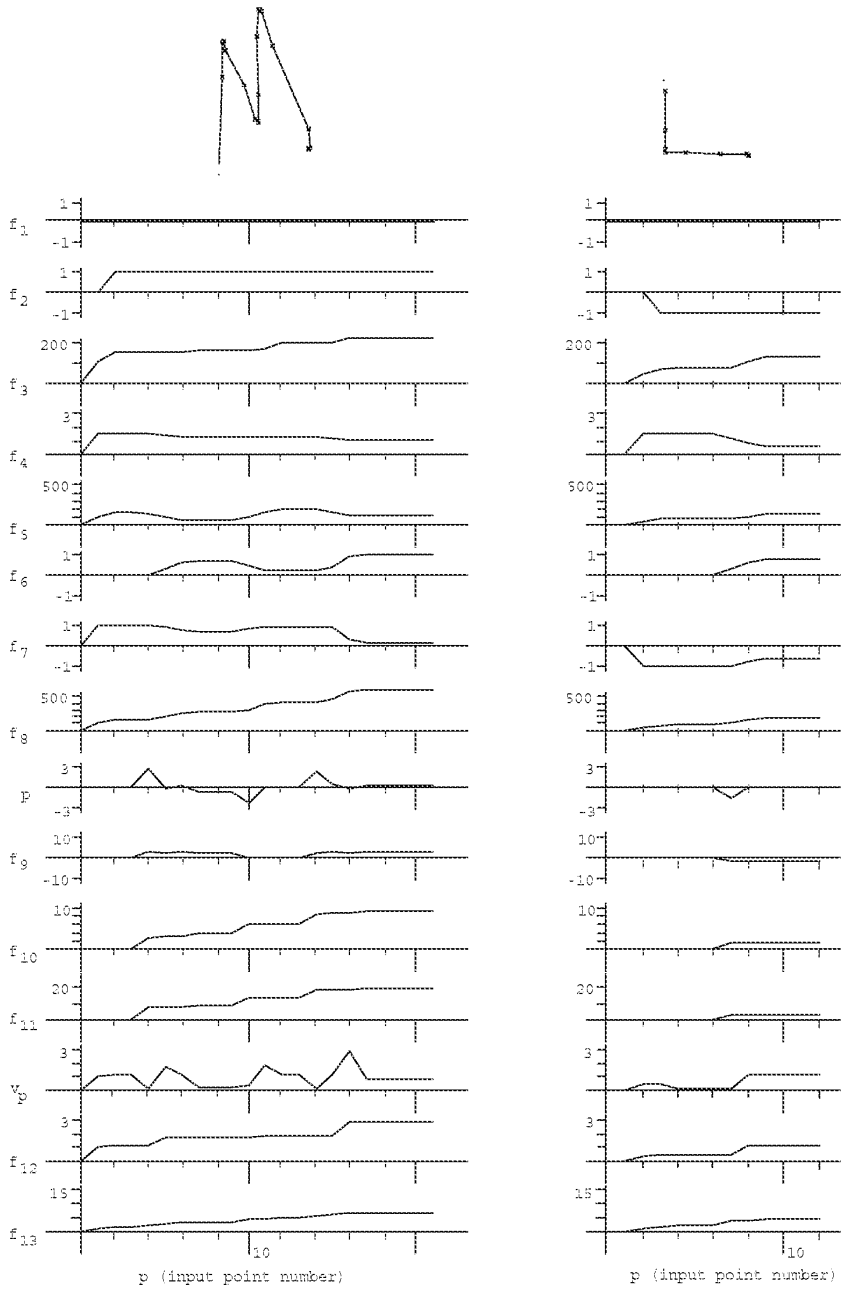


Figure 3.3: Feature vector computation

These graphs show how feature vectors change with each new input point. The left graphs refer to features of gesture 1 of 3.1 (an "M"), the right graphs to gesture 2 (an "L"). The final values of the features ($p = 21$ for gesture 1, $p = 12$ for gesture 2) are the ones used for classification. The instantaneous angle θ_p and velocity v_p have been included in the figure, although they are not part of the feature vector.

the per class feature covariance matrices are equal. (Exactly what this means is discussed in the next section. Other continuous distributions for which linear discriminant functions are optimal are investigated by Cooper [26].) These conditions do not hold for most sets of gesture classes given the feature set described; thus weights calculated assuming these conditions will not be optimal, even among linear classifiers (and even the optimal linear classifier can be outperformed by some non-linear classifiers if the above conditions are not satisfied). However, given the above set of features, linear discriminators computed as if the conditions are valid have been found to perform quite acceptably in practice.

3.5 Classifier Training

Once the decision has been made to use linear discriminators, the only problem that remains is the determination of the weights from example gestures of each class. This is known as the training problem.

Two methods for computing the weights were tried. The first was the multiclass perceptron training procedure described in Sklansky and Wassel[119]. The hope was that this method, which does not depend on the aforementioned conditions to choose weights, might perform better than methods that did. In this method, an initial guess of the weights was made, which are then used to classify the first example. For each class whose evaluation function scored higher than the correct class, each weight is reduced by an amount proportional to the corresponding feature of the example, while the correct class has its weights increased by the same amount. This is similar to back-propagation learning procedures in neural nets [34]. In this manner, all the examples are tried, multiple times if desired.

This method has the advantage of being simple, as well as needing very few example gestures to achieve reasonable results. However, the behavior of the classifier depends on the order in which the examples are presented for training, and good values for the initial weights and the constant of proportionality are difficult to determine in advance but have a large effect on the success and training efficiency of the method. The number of iterations of the examples is another variable whose optimum value is difficult to determine. Perhaps the most serious problem is that a single bad example might seriously corrupt the classifier.

Eventually, the perceptron training method was abandoned in favor of the plug-in estimation method. The plug-in estimation method usually performs approximately equally to the best perceptron-trained classifiers, and has none of the vagueness associated with perceptron training. In this method, the means of the features for each class are estimated from the example gestures, as is the common feature covariance matrix (all classes are assumed to have the same one). The estimates are then used to approximate the linear weights that would be optimal assuming the aforementioned conditions were true.

3.5.1 Deriving the linear classifier

The derivation of the plug-in classifier is given in detail in James [62]. James' explanation of the derivation is particularly good, though unfortunately the derivation itself is riddled with typos and

other errors. Krzanowski [74] gives a similar derivation (with no errors), as well as a good general description of multivariate analysis. The derivation is summarized here for convenience.

Consider the class of "L" gestures, drawn starting from the top-left. One example of this class is gesture 2 in figure 3.1. It is easy to generate many more examples of this class. Each one gives rise to a feature vector, considered to be a column vector of F real numbers $(f_0, \ldots, f_{F-1})'$.

Let f be the random vector (*i.e.* a vector of random variables) representing the feature vectors of a given class of gestures, say "L" gestures. Assume (for now) that f has a *multivariate normal* distribution. The multivariate normal distribution is a generalization to vectors of the normal distribution for a single variable. A single variable (univariate) normal distribution is specified by its mean value and variance. Analogously, a multivariable normal distribution is specified by its mean vector, $\overline{\mu}$, and covariance matrix, Σ. In a multivariate normal distribution, each vector element (feature) has a univariate normal distribution, and the mean vector is simply a vector consisting of the means of the individual features. The variance of the features form the diagonal of the covariance matrix; the off-diagonal elements represent correlations between features.

The univariate normal distribution has a density function which is the familiar bell-shaped curve. The analog in the two variable (bivariate) case is a three-dimensional bell shape. In this case, the lines of equal probability (cross sections of the bell) are concentric ellipses. The axes of the ellipses are parallel to the feature axes if and only if the variables are uncorrelated. By analogy, in the higher dimensional cases, the distribution has a hyper-bell shape, and the equiprobability hypersurfaces are ellipsoids.

A more in-depth discussion of the properties of the multivariate normal distribution would take us too far afield here. The reader unfamiliar with the subject is asked to rely on the analogy with the univariate case, or to refer to a good text, such as Krzanowski [74].

The multivariate normal probability density function is the multivariate analog to the bell-shaped curve. It is written here as a conditional probability density, *i.e.* the density of the probability of getting vector $\mathbf{x}$ given $\mathbf{x}$ comes from multivariate distribution L with F variables, mean $\overline{\mu}$, and covariance matrix Σ.

$$p(\mathbf{x} \mid L) = (2\pi)^{-F/2} |\Sigma|^{-1/2} e^{-\frac{1}{2}(\mathbf{x}-\overline{\mu})' \Sigma^{-1} (\mathbf{x}-\overline{\mu})} \tag{3.2}$$

Note that this expression involves both the determinant and the inverse of the covariance matrix. The interested reader should verify that it reduces to the standard bell-shaped curve in the univariate case ($F = 1, \Sigma = [\sigma^2]$).

In the univariate case, to determine the probability that the value of a random variable will lie within a given interval, simply integrate the probability density function over that interval. Analogously in the multivariate case, given an interval for each of the variables (*i.e.* a hypervolume) perform a multiple integral, integrating each variable over its interval to determine the probability a random vector is within the hypervolume.

All this is preparation of the derivation of the linear classifier. Assume an example feature vector $\mathbf{x}$ to be classified is given. Let $C^{\hat{c}}$ denote the event that a random feature vector $\mathbf{X}$ is in class c and $\mathbf{x}$, when used as an event, denote the event that the random feature vector $\mathbf{X}$ has value $\mathbf{x}$. We are interested in $P(C^{\hat{c}} \mid \mathbf{x})$, the probability that the particular feature vector $\mathbf{x}$ is in group $C^{\hat{c}}$. A reasonable classification rule is to assign $\mathbf{x}$ to the class i whose probability $P(C^{\hat{i}} \mid \mathbf{x})$ is greater than that of the other classes, *i.e.* $P(C^{\hat{i}} \mid \mathbf{x}) > P(C^{\hat{j}} \mid \mathbf{x})$ for all $j \neq i$. This rule, which assigns the example

to the class with the highest conditional probability, is known as *Bayes' rule*.

The problem is thus to determine $P(C^{\hat{c}} \mid \mathbf{x})$ for all classes c. Bayes' theorem tells us

$$P(C^{\hat{c}} \mid \mathbf{x}) = \frac{P(\mathbf{x} \mid C^{\hat{c}})P(C^{\hat{c}})}{\sum_{\text{all } k} P(\mathbf{x} \mid C^{\hat{k}})P(C^{\hat{k}})} \tag{3.3}$$

Substituting, the assignment rule now becomes: assign $\mathbf{x}$ to class i if $P(\mathbf{x} \mid C^{\hat{i}})P(C^{\hat{i}}) > P(\mathbf{x} \mid C^{\hat{j}})P(C^{\hat{j}})$ for all $j \neq i$.

The terms of the form $P(C^{\hat{c}})$ are the *a priori* probabilities that a random example vector is in class c. In a gesture recognition system, these prior probabilities would depend on the frequency that each gesture command is likely to be used in an application. Lacking any better information, let us assume that all gestures are equally likely, resulting in the rule: assign $\mathbf{x}$ to class i if $P(\mathbf{x} \mid C^{\hat{i}}) > P(\mathbf{x} \mid C^{\hat{j}})$ for all $j \neq i$.

A conditional probability of the form $P(\mathbf{x} \mid C^{\hat{c}})$ is known as the *likelihood* of $C^{\hat{c}}$ with respect to $\mathbf{x}$ [30]; assuming equal priors essentially replaces Bayes' rule with one that gives the maximum likelihood.

Assume now that each $C^{\hat{c}}$ is multivariate normal, with mean vector $\overline{\mu}^{\hat{c}}$, and covariance matrix $\Sigma_{\hat{c}}$. Substituting the multivariate normal density functions (equation 3.2) for the probabilities gives the assignment rule: assign $\mathbf{x}$ to class i if, for all $j \neq i$,

$$(2\pi)^{-F/2}|\Sigma_{\hat{i}}|^{-1/2}e^{-\frac{1}{2}(\mathbf{x}-\overline{\mu}^{\hat{i}})'\Sigma_{\hat{i}}^{-1}(\mathbf{x}-\overline{\mu}^{\hat{i}})} > (2\pi)^{-F/2}|\Sigma_{\hat{j}}|^{-1/2}e^{-\frac{1}{2}(\mathbf{x}-\overline{\mu}^{\hat{j}})'\Sigma_{\hat{j}}^{-1}(\mathbf{x}-\overline{\mu}^{\hat{j}})}$$

Taking the natural log of both sides, canceling, and multiplying through by -1 (thus reversing the inequality) gives the rule: assign $\mathbf{x}$ to class i if, for all $j \neq i$,

$$d^{\hat{i}}(\mathbf{x}) < d^{\hat{j}}(\mathbf{x}), \text{ where } d^{\hat{c}}(\mathbf{x}) = \ln|\Sigma_{\hat{c}}| + (\mathbf{x} - \overline{\mu}^{\hat{c}})'\Sigma_{\hat{c}}^{-1}(\mathbf{x} - \overline{\mu}^{\hat{c}}) \tag{3.4}$$

$d^{\hat{c}}(\mathbf{x})$ is the discrimination function for class c applied to $\mathbf{x}$. This is *quadratic* discrimination, since $d^{\hat{c}}(\mathbf{x})$ is quadratic in elements of $\mathbf{x}$ (the features). The discriminant computation involves the weighted sum of the pairwise products of features, as well as terms linear in the features, and a constant term.

Making the further assumption that all the per-class covariances matrices are equal, *i.e.* $\Sigma_{\hat{i}} = \Sigma_{\hat{j}} = \Sigma$, the assignment rule takes the form: assign $\mathbf{x}$ to class i if, for all $j \neq i$,

$$\ln|\Sigma| + (\mathbf{x} - \overline{\mu}^{\hat{i}})'\Sigma^{-1}(\mathbf{x} - \overline{\mu}^{\hat{i}}) < \ln|\Sigma| + (\mathbf{x} - \overline{\mu}^{\hat{j}})'\Sigma^{-1}(\mathbf{x} - \overline{\mu}^{\hat{j}}).$$

Distributing the subtractions and multiplying through by $-\frac{1}{2}$ gives the rule: assign $\mathbf{x}$ to class i if, for all $j \neq i$,

$$\nu^{\hat{i}}(\mathbf{x}) > \nu^{\hat{j}}(\mathbf{x}), \text{ where } \nu^{\hat{c}}(\mathbf{x}) = (\overline{\mu}^{\hat{c}})'\Sigma^{-1}\mathbf{x} - \tfrac{1}{2}(\overline{\mu}^{\hat{c}})'\Sigma^{-1}\overline{\mu}^{\hat{c}} \tag{3.5}$$

Note that the discrimination functions $\nu^{\hat{c}}(\mathbf{x})$ are linear in the features (*i.e.* the elements of $\mathbf{x}$), the weights being $(\overline{\mu}^{\hat{c}})'\Sigma^{-1}$ and the constant term being $-\frac{1}{2}(\overline{\mu}^{\hat{j}})'\Sigma^{-1}\overline{\mu}^{\hat{j}}$.

Comparing equations 3.5 and 3.1 it is seen that to have the optimum classifier (given the assumptions) we take

$$w_j^{\hat{c}} = \sum_{i=1}^{F} \Sigma_{ij}^{-1} \bar{\mu}_i^{\hat{c}} \qquad 1 \le j \le F$$

and

$$w_0^{\hat{c}} = -\frac{1}{2} \sum_{i=1}^{F} w_i^{\hat{c}} \bar{\mu}_i^{\hat{c}}$$

for all classes c. It is not possible to know the $\bar{\mu}^{\hat{c}}$ and Σ; these must be estimated from the examples as described in the next section. The result will be that the $w_i^{\hat{c}}$ will be estimates of the optimal weights.

The possibility of a tie for the largest discriminant has thus far neglected. If $\nu^{\hat{i}}(\mathbf{x}) = \nu^{\hat{k}}(\mathbf{x}) > \nu^{\hat{j}}(\mathbf{x})$ for all $j \neq i$ and $j \neq k$, it is clear that the classifier may arbitrarily choose i or k as the class of $\mathbf{x}$. However, this is a prime case for rejecting the gesture $\mathbf{x}$ altogether, since it is ambiguous. This kind of rejection is generalized in Section 3.6.

3.5.2 Estimating the parameters

The linear classifier just derived is optimal (given all the assumptions) in the sense that it maximizes the probability of correct classification. However, the parameters needed to operate the classifier, namely the per-class mean vectors $\bar{\mu}^{\hat{c}}$ and the common covariance matrix Σ, are not known *a priori*. They must be estimated from the training examples. The simplest approach is to use the plug-in estimates for these statistics. Since the equations that follow actually need to be programmed, the matrix notation is discarded in favor of writing the sums out explicitly in terms of the components.

Let $f_{ei}^{\hat{c}}$ be the i^{th} feature of the e^{th} example of gesture class c, $0 \le e < E^{\hat{c}}$, where $E^{\hat{c}}$ is the number of training examples of class c. The plug-in estimate of $\bar{\mu}^{\hat{c}}$, the mean feature vector per class, is denoted $\bar{f}_i^{\hat{c}}$. It is simply the average of the features in the class:

$$\bar{f}_i^{\hat{c}} = \frac{1}{E^{\hat{c}}} \sum_{e=0}^{E^{\hat{c}}-1} f_{ei}^{\hat{c}}$$

$s_{ij}^{\hat{c}}$ is the plug-in estimate of $\Sigma_{\hat{c}}$, the feature covariance matrix for class c:

$$s_{ij}^{\hat{c}} = \sum_{e=0}^{E^{\hat{c}}-1} (f_{ei}^{\hat{c}} - \bar{f}_i^{\hat{c}})(f_{ej}^{\hat{c}} - \bar{f}_j^{\hat{c}})$$

(For convenience in the next step, the usual $1/(E^{\hat{c}} - 1)$ factor has not been included in $s_{ij}^{\hat{c}}$.) The $s_{ij}^{\hat{c}}$ are averaged to give s_{ij}, an estimate of the common covariance matrix Σ.

$$s_{ij} = \frac{\sum_{c=0}^{C-1} s_{ij}^{\hat{c}}}{-C + \sum_{c=0}^{C-1} E^{\hat{c}}} \tag{3.6}$$

The plug-in estimate of the common covariance matrix s_{ij} is then inverted, the result of which is denoted $(s^{-1})_{ij}$.

The $v^{\hat{c}}$ are estimates of the optimal evaluation functions $v^{\hat{c}}(\mathbf{x})$. The weights $w_j^{\hat{c}}$ are computed from the estimates as follows:

$$w_j^{\hat{c}} = \sum_{i=1}^{F} (s^{-1})_{ij} \bar{f}_i^{\hat{c}} \qquad 1 \le j \le F$$

and

$$w_0^{\hat{c}} = -\frac{1}{2} \sum_{i=1}^{F} w_i^{\hat{c}} \bar{f}_i^{\hat{c}}$$

As mentioned before, it is assumed that all gesture classes are equally likely to occur. The constant terms $w_0^{\hat{c}}$ may be adjusted if the *a priori* probabilities of each gesture class are known in advance, though the author has not found this to be necessary for good results. If the derivation of the classifier is carried out without assuming equal probabilities, the net result is, for each class, to add $\ln P(C^{\hat{c}})$ to $w_0^{\hat{c}}$. A similar correction may be made to the constant terms if differing per-class costs for misclassification must be taken into account [74].

Estimating the covariance matrix involves estimating its $F(F+1)/2$ elements. The matrix will be singular if, for example, less than approximately F examples are used in its computation. Or, a given feature may have zero variance in every class. In these cases, the classifier is underconstrained. Rather then give up (which seems an inappropriate response when underconstrained) an attempt is made to fix a singular covariance matrix. First, any zero diagonal element is replaced by a small positive number. If the matrix is still singular, then a search is made to eliminate unnecessary features.

The search starts with an empty set of features. At each iteration, a feature i is added to the set, and a covariance matrix based only on the features in the set is constructed (by taking the singular $F \times F$ covariance matrix and using only the rows and columns of those features in the set). If the constructed matrix is singular, feature i is removed from the set, otherwise i is kept. Each feature is tried in turn. The result is a covariance matrix (and its inverse) of dimensionality smaller than $F \times F$. The inverse covariance matrix is expanded to size $F \times F$ by adding rows and columns of zeros for each feature not used. The resulting matrix is used to compute the weights.

Appendix A shows C code for training classifiers and classifying feature vectors.

3.6 Rejection

Given an input gesture g the classification algorithm calculates the evaluation $v^{\hat{c}}$, for each class c. The class k whose evaluation $v_{\hat{k}}$ is larger than all other $v^{\hat{c}}$ is presumed to be the class of g. However, there are two cases that might cause us to doubt the correctness of the classifier. The gesture g may be *ambiguous*, in that it is similar to the gestures of more than one class. Also, g may be an *outlier*, different from any of the expected gesture classes.

It would be desirable to get an estimate of how sure the classifier is that the input gesture is unambiguously in class i. Intuitively, one might expect that if some $v^{\hat{m}}$, $m \neq i$, is close to $v^{\hat{i}}$, then

the classifier is unsure of its classification, since it almost picked m instead of i. This intuition is borne out in the expression for the probability that the feature vector $\mathbf{x}$ is in class $\hat{\imath}$. Again assuming normal features, equal covariances, and equal prior probabilities, substitute the multivariate normal density function (equation 3.2) into Bayes' Theorem (equation 3.3).

$$P(\hat{\imath} \mid \mathbf{x}) = \frac{e^{-\frac{1}{2}(\mathbf{x}-\overline{\mu}^{\hat{\imath}})' \Sigma^{-1} (\mathbf{x}-\overline{\mu}^{\hat{\imath}})}}{\sum_{j=0}^{C-1} e^{-\frac{1}{2}(\mathbf{x}-\overline{\mu}^{\hat{\jmath}})' \Sigma^{-1} (\mathbf{x}-\overline{\mu}^{\hat{\jmath}})}}$$

The common factor $(2\pi)^{-F/2}|\Sigma|^{-1/2}$ has been canceled from the numerator and denominator. We may further factor out and cancel $e^{-\frac{1}{2}\mathbf{x}'\Sigma^{-1}\mathbf{x}}$ and substitute equation 3.5, yielding

$$P(\hat{\imath} \mid \mathbf{x}) = \frac{e^{v^{\hat{\imath}}(\mathbf{x})}}{\sum_{j=0}^{C-1} e^{v^{\hat{\jmath}}(\mathbf{x})}}$$

Substituting the estimates $v^{\hat{c}}$ for the $v^{\hat{c}}(\mathbf{x})$ and incorporating the numerator into the denominator yields an estimate for the probability that i is the correct class for $\mathbf{x}$:

$$\tilde{P}(\hat{\imath} \mid \mathbf{x}) = \frac{1}{\sum_{j=0}^{C-1} e^{(v^{\hat{\jmath}} - v^{\hat{\imath}})}}$$

This value is computed after recognition and compared to a threshold T_P. If below the threshold, instead of accepting g as being in class i, g is rejected. The effect of varying T_P will be evaluated in Chapter 9. There is a tradeoff between wanting to reject as many ambiguous gestures as possible and not wanting to reject unambiguous gestures. Empirically, $T_P = 0.95$ has been found to be a reasonable value for a number of gesture sets (see Section 9.1.2).

The expression for $\tilde{P}(\hat{\imath} \mid \mathbf{x})$ bears out the intuition that if two or more classes evaluate to near the same result the gesture is ambiguous. In such cases the denominator will be significantly larger than unity. Note that the denominator is always at least unity due to the $j = i$ term in the sum. Also note that all the other terms the exponents $(v^{\hat{\jmath}} - v^{\hat{\imath}})$ for $j \neq i$ will always be negative, because $\mathbf{x}$ has been classified as class i by virtue of the fact that $v^{\hat{\imath}} > v^{\hat{\jmath}}$ for $j \neq i$.

$\tilde{P}(\hat{\imath} \mid \mathbf{x})$ may be computed efficiently by using table-lookup for the exponentiation. The table need not be very extensive, since any time $v^{\hat{\jmath}} - v^{\hat{\imath}}$ is sufficiently negative (less than -6, say) the term is negligible. In practice this will be the case for almost all j.

A linear classifier will give no indication if g is an outlier. Indeed, most outliers will be considered unambiguous by the above measure of $\tilde{P}$. To test if g is an outlier, a separate metric is needed to compare g to the typical gesture of class k. An approximation to the Mahalanobis distance [74] works well for this purpose.

Given a gesture with feature vector $\mathbf{x}$, the Mahalanobis distance between $\mathbf{x}$ and class i is defined as

$$\delta^2 = (\mathbf{x} - \overline{\mu}^{\hat{\imath}})' \Sigma^{-1} (\mathbf{x} - \overline{\mu}^{\hat{\imath}})$$

Note that δ^2 is used in the exponent of the multivariate normal probability density function (equation 3.2). It plays the role that $((x-\mu)/\sigma)^2$ plays in the univariate normal distribution: the Mahalanobis distance δ^2 essentially measures the (square of the) number of standard deviations that $\mathbf{x}$ is away from the mean $\bar{\mu}^{\hat{i}}$.

If Σ^{-1} happens to be the identity matrix, the Mahalanobis distance is equivalent to the Euclidean distance. In general, the Mahalanobis distance normalizes the effects of different scales for the different features, since these presumably show up as different magnitudes for the variances s_{ii}, the diagonal elements of the common covariance matrix. The Mahalanobis distance also normalizes away the effect of correlations between pairs of features, the off-diagonal elements of the covariance matrix.

As always, it is only possible to approximate the Mahalanobis distance between a feature vector $\mathbf{x}$ and a class i. Substituting the plug-in estimators for the population statistics and writing out the matrix multiplications explicitly gives

$$\hat{d}^2 = \sum_{j=1}^{F}\sum_{k=1}^{F} \bar{s}^{-1}_{jk}(x_j - \bar{\hat{i}}_j)(x_k - \bar{\hat{i}}_k).$$

In order to reject outliers, compute $\hat{d}^2$, an approximation of the Mahalanobis distance from the feature vector $\mathbf{x}$ to its computed class i. If the distance is greater than a certain threshold T_{d^2} the gesture is rejected. Section 9.1.2 evaluates various settings of T_{d^2}; here it is noted that setting $T_{d^2} = \frac{1}{2}F^2$ is a good compromise between accepting obvious outliers and rejecting reasonable gestures.

Now that the underlying mechanism of rejection has been explained, the question arises as to whether it is desirable to do rejections at all. The answer depends upon the application. In applications with easy to use undo and abort facilities, the reject option should probably be turned off completely. This is because in either failure mode (rejection or misclassification) the user will have to redo the gesture (probably about the same amount of work in both cases) and turning on rejection merely increases the number of gestures that will have to be redone.

In applications in which it is deemed desirable to do rejection, the question arises as to how the interface should behave when a gesture is rejected. The system may prompt the user with an error message, possibly listing the top possibilities for the class (judging from the discriminant functions) and asking the user to pick. Or, the system may choose to ignore the gesture and any subsequent input until the user indicates the end of the interaction. The proper response presumably depends on the application.

3.7 Discussion

One goal of the present research was to enable the implementor of a gesture-based system to produce gesture recognizers without the need to resort to hand-coding. The original plan was to try a number of pattern recognition techniques of increasing complexity until one powerful enough to recognize gestures was found. The author was pleasantly surprised when the first technique he tried, linear discrimination, produced accurate and efficient classifiers.

Figure 3.4: Two different gestures with identical feature vectors

The efficiency of linear recognition is a great asset: gestures are recognized virtually instantaneously, and the system scales well. The incremental feature calculation, with each new input point resulting in a bounded (and small) amount of computation, is also essential for efficiency, enabling the system to handle large gestures as efficiently as small ones.

3.7.1 The features

The particular feature set reported on here has worked fine discriminating between the gestures used in three sample applications: a simple drawing program, the uppercase letters in the alphabet, and a simple score editor. Tests using the gesture set of the score editor application are the most significant, since the recognizer was developed and tested on the other two. Chapter 9 studies the effect of training set size and number of classes on the performance of the recognizer. A classifier which recognizes thirty gestures classes had a recognition rate of 96.8% when trained with 100 examples per class, and a rate of 95.6% when trained with 10 examples per class. The misclassifications were largely beyond the control of the recognizer: there were problems using the mouse as a gesturing device and problems using a user process in a non-real-time system (UNIX) to collect the data.

It would be desirable to somehow show that the feature set was adequate for representing differences between all gestures likely to be encountered in practice. The measurements in Chapter 9 show good results on a number of different gestures sets, but are by no means a proof of the adequacy of the features. However, the mapping from gestures (sequences of points) to feature vectors is not one-to-one. In fact, it can easily be demonstrated that there are apparently different gestures that give rise to the same feature vector. Figure 3.4 shows one such pair of gestures. Since none of the features in the feature set depend on the order in which the angles in the gesture are encountered, and the two gestures are alike in every other respect, they have identical feature vectors. Obviously, any classifier based on the current feature set will find it impossible to distinguish between these gestures.

Of course, this particular deficiency of the feature set can be fixed by adding a feature that does depend on the order of the angles. Even then, it would be possible to generate two gestures which have the same angles in the same order, which differ, say, in the segment lengths between the angles, but nontheless give rise to the same feature vector. A new feature could then be added to handle this case, but it seems that there is still no way of being sure that there do not exist two different gestures giving rise to the same feature vector.

Nonetheless, adding features is a good way to deal with gesture sets containing ambiguous

classes. Eventually, the number of features might grow to the point such that the recognizer performs inefficiently; if this happens, one of the algorithms that chooses a good subset of features could be applied [62, 103]. (Though not done in the present work, the contribution of individual features for a given classifier can be found using the statistical techniques of principle components analysis and analysis of variance [74].) However, given the good coverage that can be had with 13 features, 20 features would make it extremely unlikely that grossly different gestures with similar feature vectors would be encountered in practice. Since recognition time is proportional to the number of features, it is clear that a 20 feature recognizer does not entail a significant processing burden on modern hardware, even for large (40 class) gesture sets. There still may be good reason to employ fewer features when possible; for example, to reduce the number of training examples required.

The problem of detecting when a classifier has been trained on ambiguous classes is of great practical significance, since it determines if the classifier will perform poorly. One method is to run the training examples through the classifier, noting how many are classified incorrectly. Unfortunately, this may fail to find ambiguous classes since the classifier is naturally biased toward recognizing its training examples correctly. An alternative is to compute the pairwise Mahalanobis distance between the class means; potentially ambiguous classes will be near each other.

3.7.2 Training considerations

There is a potential problem in the training of classifiers, even when the intended classes are unambiguous. The problem arises when, within a class, the training examples do not have sufficient variability in the features that are irrelevant to the recognition of that class.

For example, consider distinguishing between two classes: (1) a rightward horizontal segment and (2) an upward vertical segment. Suppose all the training examples of the rightward segment class are short, and all those of the upward segment class are long. If the resulting classifier is asked to classify a long rightward segment, there is a significant probability of misclassification.

This is not surprising. Given the training examples, there was no way for classifier to know that being a rightward segment was the important feature of class (1), but that the length of the segment was irrelevant. The same training examples could just as well have been used to indicate that all elements of class (1) are short segments.

The problem is that, by not varying the length of the training examples, the trainer does not give the system significant information to produce the desired classifier. It is not clear what can be done about this problem, except perhaps to impress upon the people doing the training that they need to vary the irrelevant features of a class.

3.7.3 The covariance matrix

An important problem of linear recognition comes from the assumption that the covariance matrices for each class are identical. Consider a classifier meant to distinguish between three gestures classes named **C**, **U**, and **I** (figure 3.5). Examples of class **C** all look like the letter "C", and examples of class **U** all look like the letter "U." Assume that example **C** and **U** gestures are drawn similarly

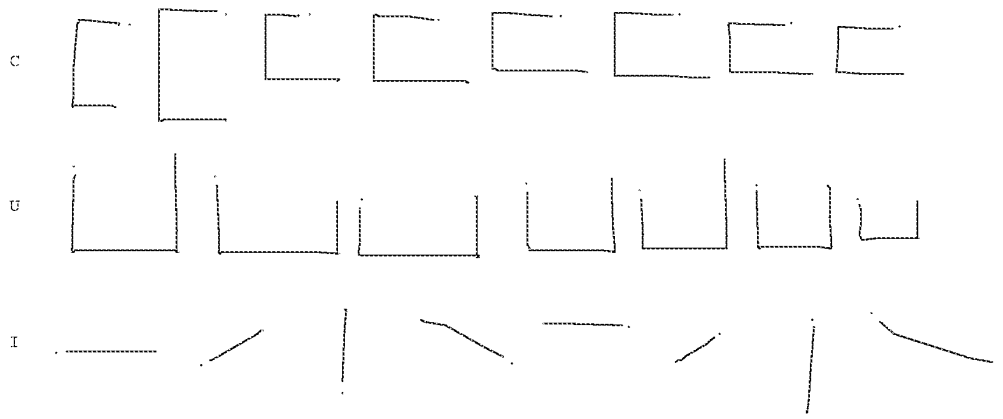


Figure 3.5: A potentially troublesome gesture set

This figure contains examples of three classes: **C**, **U**, *and* **I**. **I** *varies in orientation while* **C** *and* **U** *depend upon orientation to be distinguished. Theoretically, there should be a problem recognizing gestures in this set with the current algorithm, but in practice this has been shown not to be the case.*

except for the initial orientation. Examples of class **I**, however, are strokes which may occur in any initial orientation.

The point of this set of gesture classes is that initial orientation is essential for distinguishing between **C** and **U** gestures, but must be ignored in the case of **I** gestures. This information is contained in the per-class covariance matrices $s_{ij}^{\mathbf{C}}$, $s_{ij}^{\mathbf{U}}$, and $s_{ij}^{\mathbf{I}}$. In particular, consider the variance of the feature f_1, which, for each class c, is proportional to $s_{11}^{\hat{c}}$. Since the initial angle is almost the same for each example C gesture, $s_{11}^{\mathbf{C}}$ will be close to zero. Similarly, $s_{11}^{\mathbf{U}}$ will also be close to zero. However, since the examples of class **I** have different orientations, $s_{11}^{\mathbf{I}}$ will be significantly non-zero.

Unfortunately, the information on the variance of f_1 is lost when the per-class covariance matrix estimates $s_{ij}^{\hat{c}}$ are averaged to give an estimate of the common covariance matrix s_{ij} (equation 3.6). Initially, it was suspected this would cause a problem resulting in significantly lowered recognition rates, but in practice the effect has not been too noticeable. The classifier has no problem distinguishing between the above gestures correctly.

A more extensive test where some gestures vary in size and orientation while others depend on size and orientation to be recognized is presented in Section 9.1.4. The recognition rates achieved show the classifier has no special difficulty handling such gesture sets. Had there been a real problem, the plan was to experiment with improving the linear classifier, say by a few iterations of the perceptron training method [119]. Had this not worked, using a quadratic discriminator (equation 3.4) was another possible area of exploration.

3.8 Conclusion

This chapter discussed how linear statistical pattern recognition techniques can be successfully applied to the problem of classifying single-path gestures. By using these techniques, implementors of gesture-based systems no longer have to write application-specific gesture-recognition code. It is hoped that by making gesture recognizers easier to create and maintain, the promising field of gesture-based systems will be more widely explored in the future.

Chapter 4

Eager Recognition

4.1 Introduction

In Chapter 3, an algorithm for classifying single-path gestures was presented. The algorithm assumes that the entire input gesture is known, *i.e.* that the start and end of the gesture are clearly delineated. For some applications, this restriction is not a problem. For others, however, the need to indicate the end of the gesture makes the user interface more awkward than it need be.

Consider the use of mouse gestures in the GDP drawing editor (Section 1.1). To create a rectangle, the user presses a mouse button at one corner of the rectangle, enters the "L" gesture, stops (while still holding the button), waits for the rectangle to appear, and then positions the other corner. It would be much more natural if the user did not have to stop; *i.e.* if the system recognized the rectangle gesture *while the user was making it*, and then created the rectangle, allowing the user to drag the corner. What began as a gesture changes to a rubberbanding interaction with no explicit signal or timeout.

Another example, mentioned previously, is the manipulation of the image of a knob on the screen. Let us suppose that the knob responds to two gestures: it may be turned or it may be tapped. It would be awkward if the user, in order to turn the knob, needed to first begin to turn the knob (entering the turn gesture), then stop turning it (asking the system to recognize the turn gesture), and then continue turning the knob, now getting feedback from the system (the image of the knob now rotates). It would be better if the system, as soon as enough of the user's gesture has been seen so as to unambiguously indicate her intention of turning the knob, begins to turn the knob.

The author has coined the term *eager recognition* for the recognition of gestures as soon as they are unambiguous. Henry et. al. [52] mention that Artkit, a system similar to GRANDMA, can be used to build applications that perform eager recognition of mouse gestures. There is currently no information published as to how gesture recognition or eager recognition is implemented using Artkit. GloveTalk [34] does something similar in the recognition of DataGlove gestures. GloveTalk attempts to use the deceleration of the hand to indicate that the gesture in progress should be recognized. It utilizes four neural networks: the first recognizes the deceleration, the last three classify the gesture when indicated to do so by the first.

Eager recognition is the automatic recognition of the end of a gesture. For many applications, it

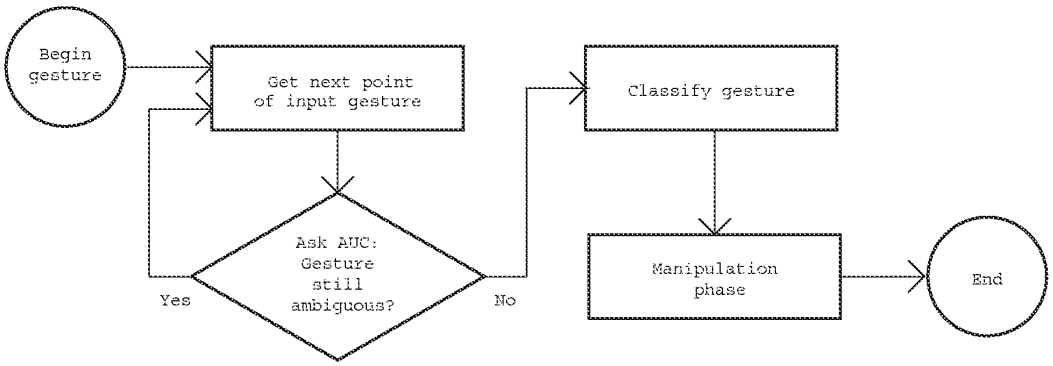


Figure 4.1: Eager recognition overview

Eager recognition works by collecting points until the gesture is unambiguous, at which point the gesture is classified by the techniques of the previous chapter and the manipulation phase is entered. The determination as to whether the gesture seen so far is ambiguous is done by the AUC, i.e. *the* ambiguous/unambiguous *classifier.*

is not a problem to indicate the start of a gesture explicitly, by pressing a mouse button for example. In the present work, no attempt is made to solve the problem of determining the start of a gesture. Recognizing the start of a gesture automatically is especially important for gesture-based systems that use input devices without any explicit signaling capability (*e.g.* the Polhemus sensor or the DataGlove). For such a device, sudden changes in speed or direction might be used to indicate the start of a gesture. More complex techniques for determining the start of a gesture are outside the scope of this dissertation.

There has been some work on the automatic recognition of the start of gestures. Jackson and Roske-Hofstrand's system [61] recognizes the start of a circling gesture without an explicit indication. In GloveTalk, the user is always gesturing; thus the end of one gesture indicates the start of another. Also related is the automatic segmentation of characters in handwriting systems [125, 13], especially the online recognition of cursive writing [53].

4.2 An Overview of the Algorithm

In order to implement eager recognition, a module is needed that can answer the question "has enough of the gesture being entered been seen so that it may be unambiguously classified?" (figure 4.1). The insight here is to view this as a classification problem: classify a given gesture in progress (called a *subgesture* below) as an ambiguous or unambiguous gesture prefix. This is essentially the approach taken independently in GloveTalk. Here, the recognition techniques developed in the previous chapter are used to build the ambiguous/unambiguous classifier (AUC).

Two main problems need to be solved with this approach. First, training data is needed to train the AUC. Second, the AUC must be powerful enough to accurately discriminate between ambiguous and unambiguous subgestures.

In GloveTalk, the training data problem was solved by explicitly labeling snapshots of a gesture in progress. Each gesture was made up of an average of 47 snapshots (samples of the DataGlove and Polhemus sensors). For each of 638 gestures, the snapshot indicating the time at which the system should recognize the gesture had to be indicated. This is clearly a significant amount of work for the trainer of the system.

In order to avoid such tedious tasks, the present system constructs training examples for the AUC from the gestures used to train the main gesture recognizer. The system considers each subgesture of each example gesture, labels it either ambiguous or not, and uses the labeled subgestures as training data. It seems there is a chicken-and-egg problem here: in order to create the training data, the system needs to perform the very task for which it is trying to create a classifier. However, during the creation of the training data, the system has access to a crucial piece of information that makes the problem tractable: to determine if a given subgesture is ambiguous the system can examine the entire gesture from which the subgesture came.

Once the training data has been created, a classifier must be constructed. In GloveTalk this presented no particular difficulty, for two reasons. There, the classifier was trained to recognize decelerations that, as indicated by the sensor data, were similar between different gesture classes. Also, neural networks with hidden layers are better suited for recognizing classes with non-Gaussian distributions.

In the present system, the training data for the AUC consists of two sets: unambiguous subgestures and ambiguous subgestures. The distribution of feature vectors within the set of unambiguous subgestures will likely be wildly non-Gaussian, since the member subgestures are drawn from many different gesture classes. For example, in GDP the unambiguous delete subgestures are very different from the unambiguous pack gestures, etc., so there will be a multimodal distribution of feature vectors in the unambiguous set. Similarly, the distribution of feature vectors in the ambiguous set will also likely be non-Gaussian. Thus, a linear discriminator of the form developed in the previous chapter will surely not be adequate to discriminate between two classes ambiguous and unambiguous subgestures. What must be done is to turn this two-class problem (ambiguous or unambiguous) into a multi-class problem. This is done by breaking up the ambiguous subgestures into multiple classes, each of which has an approximately normal distribution. The unambiguous subgestures must be similarly partitioned.

The details of the creation of the training data and the construction of the classifier are now presented. First a failed attempt at the algorithm is considered, during which the aforementioned problems were uncovered. Then a working version of the algorithm is presented.

4.3 Incomplete Subgestures

As in the last chapter, we are given a set of C gesture classes, and a number of examples of each class, $g_e^{\hat{c}}$, $0 \leq c < C$, $0 \leq e < E^{\hat{c}}$, where $E^{\hat{c}}$ is the number of examples of class c. The algorithm described in this chapter produces a function $\mathcal{D}$ which when given a subgesture returns a boolean indicating whether the subgesture is unambiguous with respect to the C gesture classes. When the function indicates that the subgesture is unambiguous, the recognition algorithm described in the previous chapter is used to classify the gesture.

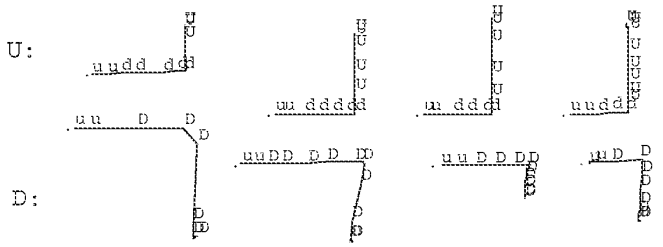


Figure 4.2: Incomplete and complete subgestures of U and D

The character indicates the classification (by the full classifier) of each subgesture. Uppercase characters indicate complete subgestures, meaning that the subgesture and all larger subgestures are correctly classified. Note that along the horizontal segment (where the subgestures are ambiguous) some subgestures are complete while others are not.

The classification algorithm of the previous chapter showed how, given a gesture g, to calculate a feature vector $\mathbf{x}$. A linear discriminator was then used to classify $\mathbf{x}$ as a class c. For much of this chapter, the classifier can be considered to be a function $\mathcal{C}$: $c = \mathcal{C}(g)$. In other words, $\mathcal{C}(g)$ is the class of g as computed by the classifier of Chapter 3.

The function $\mathcal{C}$ was produced from the statistics of the example gestures of each class c, g_e^c. The algorithms described in this chapter work best if only the example gestures that are in fact classified correctly by the computed classifier are used. Thus, in this chapter it is assumed that $\mathcal{C}(g_e^c) = c$ for all example gestures g_e^c. In practice this is achieved by ignoring those very few training examples that are incorrectly classified by $\mathcal{C}$.

Denote the number of input points in a gesture g as $|g|$, and the particular points as $g_p = (x_p, y_p, t_p)$, $0 \leq p < |g|$. The *i^{th} subgesture* of g, denoted $g[i]$, is defined as a gesture consisting of the first i points of g. Thus, $g[i]_p = g_p$ and $|g[i]| = i$. The subgesture $g[i]$ is simply a prefix of g, and is undefined when $i > |g|$. The term "full gesture" will be used when it is necessary to distinguish the full gesture g from its proper subgestures $g[i]$ for $i < |g|$. The term "full classifier" will be used to refer to $\mathcal{C}$, the classifier for full gestures.

For each example gesture of class c, $g = g_e^c$, some subgestures $g[i]$ will be classified correctly by the full classifier $\mathcal{C}$, while others likely will not. A subgesture $g[i]$ is termed *complete* with respect to gesture g, if, for all $j, i \leq j < |g|, \mathcal{C}(g[j]) = \mathcal{C}(g)$. The remaining subgestures of g are *incomplete*. A complete subgesture is one which is classified correctly by the full classifier, and all larger subgestures (of the same gesture) are also classified correctly.

Figure 4.2 shows examples of two gestures classes, U and D. Both start with a horizontal segment, but U gestures end with an upward segment, while D gestures end with a downward segment. In this simple example, it is clear that the subgestures which include only the horizontal segment are ambiguous, but subgestures which include the corner are unambiguous. In the figure, each point in the gesture is labeled with a character indicating the classification of the subgesture which ends at the point. An upper case label indicates a complete subgesture, lower case an incomplete subgesture. Notice that incomplete subgestures are all ambiguous, all unambiguous subgestures are complete, but there are complete subgestures that are ambiguous (along the horizontal segment of

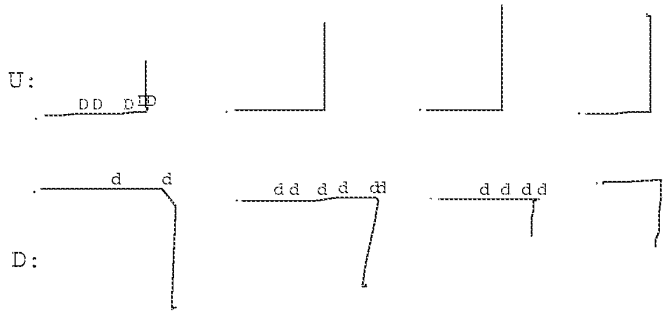


Figure 4.3: A first attempt at determining the ambiguity of subgestures
A two-class classifier was built to distinguish incomplete and complete subgestures, with the hope that those classified as complete are unambiguous and those classified as incomplete are ambiguous. The characters indicate where the resultant classifier differed from its training examples. The horizontal segment of the D *gestures were classified as incomplete (a fortuitous error), but the horizontal segment of the first* U *gesture was classified as complete. The latter is a grave mistake as the gestures are ambiguous along the horizontal segment and it would be premature for the full classifier to attempt to recognize the gesture at such points.*

the D examples).

4.4 A First Attempt

For eager recognition, subgestures that are unambiguous must be recognized as the gesture is being made. As stated above, the approach is to build an AUC, *i.e.* a classifier which distinguishes between ambiguous and unambiguous subgestures. Notice that the set of incomplete and complete subgestures approximate the set of ambiguous and unambiguous subgestures, respectively. The author's first, rather naive attempt at eager recognition was to partition the subgestures of all the example gestures into two classes, incomplete and complete. A linear classifier was then produced using the method described in Chapter 3. This classifier attempts to discriminate between complete and incomplete subgestures. The function $\mathcal{D}(g)$ then simply returns **false** whenever the above classifier reports that g is incomplete, and **true** whenever the classifier claims g is complete.

Figure 4.3 shows the output of the computed classifier for examples of U and D. Points corresponding to subgestures are labeled only when the classifier has made an error, in the sense that the classification does not agree with the training data (shown in figure 4.2). The worst possible error is for the classifier to indicate a complete gesture which happens to still be incomplete, which occurred along the right stroke of the first U gesture.

This approach to eager recognition was not very successful. That it is inadequate was indicated even more strongly by its numerous errors when tried on an example containing six gesture classes. It does however contain the germ of a good idea: that statistical classification may be used to determine if a gesture is ambiguous. A detailed examination of the problems of this attempt is instructive, and leads to a working eager recognition algorithm.

This first attempt at eager recognition has a number of problems:

- The distinction between incomplete and complete subgestures does not exactly correspond with the distinction between ambiguous and unambiguous subgestures. In the U and D example, subgestures consisting only of points along the right stroke are complete for gestures which eventually turn out to be D, and incomplete for gestures that turn out to be U. Yet, these subgestures have essentially identical features. Training a classifier on such conflicting data is bound to give poor results. In the example, as long as the right stroke is in progress the gesture is ambiguous. That it happens to be a complete D gesture is an artifact of the classifier $\mathcal{C}$ (it happens to choose D given only a right stroke).

- All the subgestures of examples were placed in one of only two categories: complete or incomplete. In the case of multiple gesture classes, within each of the two categories the subgestures are likely to form further clusters. For example, the complete U subgestures will cluster together, and be apart from the complete D subgestures. When more gesture classes are used, even more clustering will occur. Thus, the distribution of the complete subgestures is not likely to be normal. Furthermore, it is likely that incomplete subgestures will be more similar to complete gestures of the same class than to incomplete subgestures of other classes. (A similar remark holds for complete subgestures.) It is thus not likely that a linear discriminator will give good results separating complete and incomplete subgestures.

- The classifier, once computed, may make errors. The most severe error is reporting that a gesture is complete when it is in fact still ambiguous. The final classifier must be tuned to avoid such errors, even at the cost of making the recognition process less eager than it otherwise might be.

4.5 Constructing the Recognizer

Based on consideration of the above problems, a four step approach was adopted for the construction of classifiers able to distinguish unambiguous from ambiguous gestures.

Compute complete and incomplete sets.

Partition the example subgestures into $2C$ sets. These sets are named I-c and C-c for each gesture class c. A complete subgesture $g[i]$ is placed in the class C-c, where $c = \mathcal{C}(g[i]) = \mathcal{C}(g)$. An incomplete subgesture $g[i]$ is placed in the class I-c, where $c = \mathcal{C}(g[i])$ (and it is likely that $c \neq \mathcal{C}(g)$). The sets I-c are termed incomplete sets, and the sets C-c, complete sets. Note that the class in each set's name refers to the full classifier's classification of the set's elements. In the case of incomplete subgestures, this is likely not the class of the example gesture of which the subgesture is a prefix.

Figure 4.4 shows pseudocode to perform this step. Figure 4.2, already seen, shows the result of this step, with the subgestures in class I-D labeled d, class I-U labeled u, class C-D labeled D, and class C-U labeled u. The practice of labeling incomplete subgestures with lowercase

```
for c := 0 to C − 1 { /* initialize the 2C sets */
        incomplete_c := ∅ /* This is the set I-c */
        complete_c := ∅ /* This is the set C-c */
}
for c := 0 to C − 1 { /* every class c */
        for e := 0 to E^c − 1 { /* every training example in c */
                p := |g_e^c| /* subgestures, largest to smallest */
                while p > 0 ∧ C(g_e^c[p]) = C(g_e^c) {
                        complete_{C(g_e^c[p])} := complete_{C(g_e^c[p])} ∪ {g_e^c[p]}
                        p := p − 1
                }
                /* Once a subgesture is misrecognized by the full classifier, */
                /* it and its subgestures are all incomplete. */
                while p > 0 {
                        incomplete_{C(g_e^c[p])} := incomplete_{C(g_e^c[p])} ∪ {g_e^c[p]}
                        p := p − 1
                }
        }
}
```

Figure 4.4: Step 1: Computing complete and incomplete sets

letters and complete subgestures with uppercase letters will be continued throughout the chapter.

Move accidentally complete elements.

Measure the distance of each subgesture $g[i]$ in each complete set to the mean of each incomplete set. If $g[i]$ is sufficiently close to one of the incomplete sets, it is removed from its complete set, and placed in the close incomplete set. In this manner, an example subgesture that was accidentally considered complete (such as a right stroke of a D gesture) is grouped together with the other incomplete right strokes (class I-D in this case). Figure 4.5 shows pseudocode to perform this operation.

Quantifying exactly what is meant by "sufficiently close" turned out to be rather difficult. Using the Mahalanobis distance as a metric turns out not to work well if applied naively. The problem is that it depends on the estimated average covariance matrix, which in turn depends upon the covariance matrix of the individual classes. However, some of the classes are malformed, which is why this step of moving accidentally complete elements is necessary in the first place. For example, the C-D class has accidentally complete subgestures in it, so its covariance matrix will indicate large standard deviations in a number of features (total angle, in this case). The effect of using the inverse of this covariance matrix to measure distance is

that large differences between such features will map to small distances. Unfortunately, it is these very features that are needed to decide which subgestures are accidentally complete.

Alternatives exist. The average covariance matrix of the full gesture set (which does not include any subgestures) might be used. It would also be possible to use only the average covariance matrix of the incomplete classes. Or an attempt might be made to scale away the effect of different sized units of the features, and then apply a Euclidean metric. Or, the entire regrouping problem might be approached from a different direction, for example by applying a clustering algorithm to the training data [74]. The first alternative, using the average covariance matrix of the full gesture set (the same one used in the creation of the gesture classifier of Chapter 3) was chosen, since that matrix was easily available, and seems to work.

Once the metric has been chosen (Mahalanobis distance using the covariance matrix of the full gesture set), deciding when to move a subgesture from a complete class to an incomplete class is still difficult. The first method tried was to measure the distance of the subgesture to its current (complete) class, *i.e.* its distance from the mean of its class. The subgesture was moved to the closest incomplete class if that distance was less than the distance to its current class. This resulted in too few moves, as the mean of the complete class was biased since it was computed using some accidentally complete subgestures.

Instead, a threshold is computed, and if the distance of the complete subgesture to an incomplete class is below that threshold, the subgesture is moved. A fixed threshold does not work well, so the threshold is computed as follows: The distance of the mean of each full gesture class to the mean of each incomplete subgesture class is computed, and the minimum found. However, distances less than another threshold, F^2, are not included in the minimum calculation to avoid trouble when an incomplete subgesture looked like a full gesture of a different class. (This is the case if, in addition to U and D, there is a third gesture class consisting simply of a right stroke.) The threshold used is 90% of that minimum.

The complete subgestures of a full gesture were tested for accidental completeness from largest (the full gesture) to smallest. Once a subgesture was determined to be accidentally complete, it, and the remaining (smaller) complete subgestures are moved to the appropriate incomplete classes.

Figure 4.6 shows the classes of the subgestures in the example after the accidentally complete subgestures have been moved. Note that now the incomplete subgestures (lowercase labels) are all ambiguous.

Build the AUC, a classifier which attempts to discriminate between the partition sets.

Now that there is training data containing C complete classes, indicating unambiguous subgestures, and C incomplete classes, indicating ambiguous subgestures, it is a simple matter to run the algorithm in the previous chapter to create a classifier to discriminate between these $2C$ classes. This classifier will be used to compute the function $\mathcal{D}$ as follows: if this classifier places a subgesture s in any incomplete class, $\mathcal{D}(s)$ = **false**, otherwise the s is judged to be

```
for c := 0 to C − 1 {
    ∀ g ∈ complete_c /* each complete subgesture */{
        m := 0 /* m is the class of the incomplete set closest to g */
        for i := 1 to C − 1 {
            if distance(g, incomplete_i) < distance(g, incomplete_m)
                m := i
        }
        if distance(g, incomplete_m) < threshold {
            complete_c := complete_c − {g}
            incomplete_c := incomplete_c ∪ {g}
        }
    }
}
```

Figure 4.5: Step 2: Moving accidentally complete subgestures

The distance function and threshold value are described in the text. Though not apparent from the above code, the distance function to an incomplete set does not change when elements are added to the set.

in one of the complete classes, in which case $\mathcal{D}(s)$ = **true**. Figure 4.7 shows pseudocode for building this classifier.

Evaluate and tweak the classifier.

It is very important that subgestures not be judged unambiguous wrongly. This is a case where the cost of misclassification is unequal between classes: a subgesture erroneously classified ambiguous will merely cause the recognition not to be as eager as it could be, whereas a subgesture erroneously classified unambiguous will very likely result in the gesture recognizer misclassifying the gesture (since it has not seen enough of it to classify it unambiguously). To avoid this, the constant terms of the evaluation function of the incomplete classes i, w_{i0}, are incremented by a small amount, $\ln(M)$, where M is the relative cost of two kinds of misclassification. A reasonable value is $M = 5$, *i.e.* misclassifications as unambiguous are five times more costly than misclassifications as ambiguous. The effect is to bias the classifier so that it believes that ambiguous gestures are five times more likely than unambiguous gestures, so it is much more likely to choose an ambiguous class when unsure.

Each incomplete subgesture is then tested on the new classifier. Any time such a subgesture is classified as belonging to a complete set (a serious mistake), the constant term of the evaluation function corresponding to the complete set is adjusted automatically (by just enough plus a little more) to keep this from happening.

Figure 4.9 shows the classification by the final classifier of the subgestures in the example. A larger example of eager recognizers is presented in section 9.2.

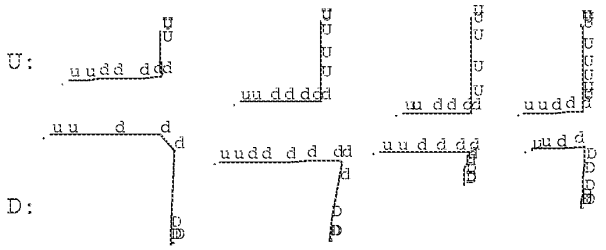


Figure 4.6: Accidentally complete subgestures have been moved

Comparing this to figure 4.2 it can be seen that the subgestures along the horizontal segment of the D *gestures have been made incomplete. Unlike before, after this step all ambiguous subgestures are incomplete.*

```
s := sNewClassifier()
for c := 0 to C − 1 {
        ∀ g ∈ complete_c
                sAddExample(s, g, "C − "c)
        ∀ g ∈ incomplete_c
                sAddExample(s, g, "I − "c)
}
sDoneAdding(s)
```

Figure 4.7: Step 3: Building the AUC

The functions called to build a classifier are `sNewClassifier()`*, which returns a new classifier object,* `sAddExample`*, which adds an example of a class, and* `sDoneAdding`*, called to generate the per-class evaluation functions after all examples have been added. These functions are described in detail in appendix A. The notation* `"C-"c` *indicates the generation a class name by concatenating the string* `"C-"` *with the value of c.*

4.6 Discussion

The algorithm just described will determine whether a given subgesture is ambiguous with respect to a set of full gestures. Presumably, as soon as it is decided that the subgesture is unambiguous it will be passed to the full classifier, which will recognize it, and then up to the application level of the system, which will react accordingly.

How well this eager recognition works depends on a number of things, the most critical being the gesture set itself. It is very easy to design a gesture set that does not lend itself well to eager recognition; for example, there would be no benefit trying to use eager recognition on Buxton's note gestures [21] (figure 2.4). This is because the note gestures for longer notes are subgestures of the note gestures for shorter notes, and thus would always be considered ambiguous by the eager recognizer. Designing a set of gestures for a given application that is both intuitive and amenable to eager recognition is in general a hard problem.

```
/* Add a small constant to the constant term of the evaluation function for */
/* each incomplete class in order to bias the classifier toward erring conservatively. */
for i := 0 to C − 1
        sIncrementConstantTerm(s, "I-"i, ln(M))
/* Make sure that no ambiguous training example is ever classified as complete. */
for i := 0 to C − 1
        ∀ g ∈ incomplete_i
                while ∃ c|sClassify(s, g) = "C − "c
                        sIncrementConstantTerm(s,"C − "c, −ε)
```

Figure 4.8: Step 4: Tweaking the classifier

First, a small constant is added to the constant term of every incomplete class (the ambiguous subgestures), to bias the classifier toward being conservative, rather than eager. Then every ambiguous subgestures is classified, and if any is accidentally classified as complete, the constant term of the evaluation function for that complete class is adjusted to avoid this.

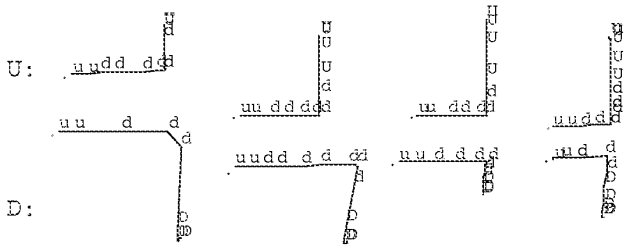


Figure 4.9: Classification of subgestures of U and D

This shows the results of running the AUC on the training examples. As can been seen, the AUC performs conservatively, never indicating that a subgesture is unambiguous when it is not, but sometimes indicating ambiguity of an unambiguous subgesture.

The training of the eager recognizer is between one and two orders of magnitude more costly than the training of the corresponding classifier for full gestures. This is largely due to the number of training examples: each full gesture example typically gives rise to ten or twenty subgestures. The amount of processing per training example is also large. In addition to computing the feature vector of each training example, a number of passes must be made over the training data: first to classify the subgestures as incomplete or complete, then to move the accidentally complete subgestures, again to build the AUC, and again to ensure the AUC is not over-eager. While a full classifier takes less than a second to train, the eager recognizer might take a substantial portion of a minute, making it less satisfying to experiment with interactively. As will be seen (Chapter 7), a full classifier may be trained the first time a user gestures at a display object. One possibility would be to use the full classifier (with no eagerness) while training the AUC in the background, activating eager recognition when ready.

The running time for the eager recognizer is also more costly than the full classifier, though not prohibitively so. A feature vector needs to be calculated for every input point; this eliminates any benefit that using auxiliary features (Section 3.3) might have bought. Of course, the AUC needs to be run at every data point; this takes about $2CF$ multiply-adds (since the AUC has $2C$ classes). Since input points do not usually come faster than one every 30 milliseconds, and $2CF$ is typically at most 1000, this computational load is not usually a problem for today's typical workstation class machine. In the current system, the multiply-adds are done in floating point, though this is probably not necessary for the recognition to work well.

One slight defect of the algorithm used to construct the AUC is that it relies totally upon the full classifier. In particular, a subgesture will never be considered unambiguous unless it is classified correctly by the full classifier. To see where this might be suboptimal, consider a full classifier that recognizes two classes, GDP's single segment line gesture and three segment delete gesture. The full classifier would likely classify any subgesture that is the initial segment of a delete as a line. It *may* also classify some two segment subgestures of delete as line gestures, even though the presence of two segments implies the gesture is unambiguously delete. The resulting eager recognizer will then not be as eager as possible, in that it will not classify the input gesture as unambiguously delete immediately after the second segment of the gesture is begun.

Two classifiers are used for eager recognition: the AUC, which decides when a subgesture is unambiguous, and the full classifier, which classifies the unambiguous subgesture. It may seem odd to use two classifiers given the implementation of the AUC, in which a subgesture is not only classified as unambiguous, but unambiguously in a given class (*i.e.* classified as C-c for some c). Why not just return a classification of c without bothering to query the full classifier? There are two main reasons. First, the full classifier, having only C classes to discriminate between, will perform better than the AUC and its $2C$ classes. Second, the final tweaking step of the AUC adjusts constant terms to assure that ambiguous gestures are never classified as unambiguous, but makes no attempt to assure that when classified as unambiguously c, c is the correct class. The adjustment of the constant terms typically degrades the AUC in the sense that it makes it more likely that c will be incorrect.

It is likely that within a decade it will be practical for neural networks to be used for gesture recognition. When this occurs, the part of this chapter concerned with building a $2C$ class linear classifier will be obsolete, since a two-class neural network could presumably do the same job. However, the part of the chapter which shows how to construct training examples for the classifier from the full gestures will still be useful, since it eliminates the hand labeling that otherwise might be necessary.

4.7 Conclusion

An eager recognizer is able to classify a gesture as soon as enough of the gesture has been seen to conclude that the gesture is unambiguous. This chapter presents an algorithm for the automatic construction of eager recognizers for single-path gestures from examples of the full gestures. It is hoped that such an algorithm will make gesture-based systems more natural to use.

Chapter 5

Multi-Path Gesture Recognition

Chapters 3 and 4 discussed the recognition of single-path gestures such as those made with a mouse or stylus. This chapter addresses the problem of recognizing multi-path gestures, *e.g.* those made using an input device, such as the DataGlove, capable of tracking the paths of multiple fingertips. It is assumed that the start and end of the multi-path gesture are known. Eager recognition of multi-path gestures has been left for future work.

The particular input device used to test the ideas in this chapter is the Sensor Frame. The Sensor Frame, as discussed in Section 2.1, is a frame which is mounted on a CRT display. The particular Sensor Frame used was mounted on the display of a Silicon Graphics IRIS Personal Workstation. The Frame detects the XY positions of up to three fingertips in a plane approximately one half inch in front of the display.

By defining the problem as "multiple-path gesture recognition", it is quite natural to attempt to apply algorithms for single-path gesture recognition (*e.g.* those developed in Chapter 3). Indeed, the recognition algorithm described in this chapter combines information culled from a number of single-path classifiers, and a "global feature" classifier in order to classify a multiple-path gesture. Before the particular algorithm is discussed, the issue of mapping the raw data returned from the particular input sensors into a form suitable for processing by the recognition algorithm must be addressed. For the Sensor Frame, this processing consisted of two stages, path tracking and path sorting.

5.1 Path Tracking

The Sensor Frame, as it currently operates, delivers the X and Y coordinates of all fingers in its plane of view each time it is polled, at a maximum rate of 30 snapshots per second. No other information is supplied; in particular the correspondence between fingers in the current and the previous snapshots is not communicated. For example, when the previous snapshot indicated one finger and the current snapshot two, it is left to the host program to determine which of the two fingers (if any) is the same finger as the previously seen one, and which has just entered the field of view. Similarly, if both the previous and current snapshots indicate two fingers, the host program must determine which finger in the current snapshot is the same as the first finger in the previous snapshot, and so on. This

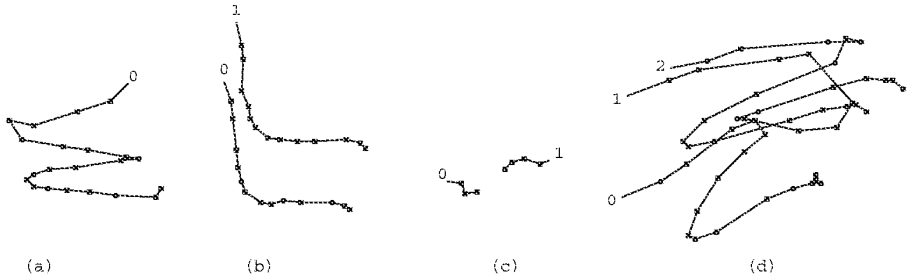


Figure 5.1: Some multi-path gestures

Shown are some MDP gestures made with a Sensor Frame. The start of each path is labeled with a path index, indicating the path's position in a canonical ordering. Gesture (a) is MDP's edit *gesture, an "E" made with a single finger. Gesture (b), parallel "L"s, is two finger* parallelogram *gesture, (c) is MDP's two finger* pinch *gesture (used for moving objects), and (d) is MDP's three finger* undo *gesture, three parallel "Z"s. The finger motions were smooth, and some noise due to the Sensor Frame's position detection can be seen in the examples.*

problem is known as *path tracking*, since it groups the raw input data into a number of paths which exist over time, each path having a definite beginning and end.

The path tracking algorithm used is quite straightforward. When a snapshot is first read, a triangular distance matrix, containing the Euclidean distance squared between each finger in the current snapshot and each in the previous, is computed. Then, for each possible mapping between current and previous fingers, an error metric, consisting of the sum of the squared distances between corresponding fingers, is calculated. The mapping with the smallest error metric is then chosen.

For efficiency, for each possible number of fingers in the previous snapshot and the current snapshot, a list of all the possible mappings are precomputed. Since the Sensor Frame detects from zero to three fingers, only 16 lists are needed. When the symmetry between the previous and current snapshots is considered, only eight lists are needed.

The low level tracking software labels each finger position with a *path identifier*. When there are no fingers in the Sensor Frame's field of view, the `next_path_identifier` variable is set to zero. A finger in the current snapshot which was not in the previous snapshot (as indicated by the chosen mapping) has its path identifier set to the value of `next_path_identifier` which is then incremented. It is thus possible for a single finger to generate multiple paths, since it will be assigned a new path identifier each time it leaves and reenters the field of view of the Sensor Frame, and those identifiers will increase as long as another finger remains in the field of view of the Frame.

The simple tracking algorithm described here was found to work very well. The anticipated problem of mistracking when finger paths crossed did not arrive very often in practice. (This was partly because all gestures were made with the fingers of a single hand, making it awkward for finger paths to cross.) Enhancements, such as using the velocity and the acceleration of each finger in the previous snapshot to predict where it is expected in the current snapshot, were not needed. Examples of the tracking algorithm in operation are shown in figure 5.1. In the figure, the start of

each path is labeled with its path index (as defined in the following section), and the points in the path are connected by line segments. Figure 5.1d shows an uncommon case where the path tracking algorithm failed, causing paths 1 and 2 to be switched.

5.2 Path Sorting

The multi-path recognition algorithm, to be described below, works by classifying the first path in the gesture, then the second, and so on, then combining the results to classify the entire gesture. It would be possible to use a single classifier to classify all the paths; this option is discussed in Section 5.7. However, since classifiers tend to work better with fewer classes, it makes sense to create multiple classifiers, one for the first path of the gesture, one for the second, and so on. This however raises the question of which path in the gesture is the first path, which is the second, etc. This is the *path sorting* problem, and the result of this sorting assigns a number to each path called its *path index*.

The most important feature of a path sorting technique is consistency. Between similar multi-path gestures, it is essential that corresponding paths have the same index. Note that the path identifiers, discussed in the previous section, are not adequate for this purpose, since they are assigned in the order that the paths first appear. Consider, for example, a "pinching" gesture, in which the thumb and forefinger of the right hand are held apart horizontally and then brought together, the thumb moving right while the forefinger moves left. Using the Sensor Frame, the thumb path might be assigned path identifier zero in one pinching gesture, since it entered the view plane of the Frame first, but assigned path identifier one in another pinching gesture since in this case it entered the view plane a fraction of a second after the forefinger. In order for multi-path gesture recognition using of multiple classifiers to give good results, it is necessary that the all thumb motions be sent to the same classifier for training and recognition, thus using path identifiers as path indices would not give good results.

For multi-path input devices which are actually attached to the hand or body, such as the DataGlove, there is no problem determining which path corresponds to which finger. Thus, it would be possible to build one classifier for thumb paths, another for forefinger paths, etc. The characteristics of the device are such that the question of path sorting does not arise.

However, the Sensor Frame (and multifinger tablets) cannot tell which of the fingers is the thumb, which is the forefinger, and so on. Thus there is no *a priori* solution to the path sorting. The solution adopted here was to impose an ordering relation between paths. The consistency property is required of this ordering relation: the ordering of corresponding paths in similar gestures must be the same.

The primary ordering criterion used was the path starting time. However, to avoid the aforementioned timing problem, two paths which start within 200 milliseconds are considered simultaneous, and the secondary ordering criteria is used. A path which starts more than 200 msec before another path will be considered "less than" the other path, and show up before the other path in the sorting.

The secondary ordering criterion is the initial x coordinate. There is a window of 150 Sensor Frame length units (about one inch) within which two paths will be considered to start at the same x coordinate, causing the tertiary ordering criterion to be applied. Outside this window, the path with

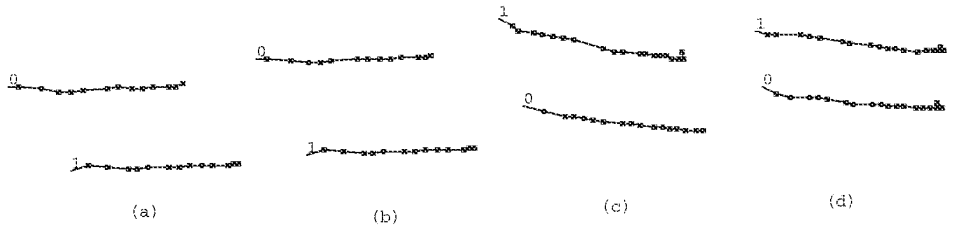


Figure 5.2: Inconsistencies in path sorting

The intention of the path sorting is that corresponding paths in two similar gestures should have the same path index. Here are four similar gestures for which this does not hold: between (b) and (c) the path sorting has changed.

the smaller initial x coordinate will appear before the other path in the sorting (assuming apparent simultaneity).

The tertiary ordering criterion is the initial y coordinate. Again, a window of 150 Sensor Frame length units is applied. Outside this window, the path whose y coordinate is less will appear earlier in the path ordering. Finally, if both the initial x and y coordinate differ by less than 150 units, the coordinate whose difference is the largest is used for ordering, and the path whose coordinate is smaller appears earlier in the path ordering.

Figure 5.2 shows the sorting for some multi-path gestures by labeling the start of each path with its index. Note that the consistency criteria is not maintained between panels (b) and (c), since the "corresponding" paths in the two gestures have different indices. The order of the paths in (b) was determined by the secondary ordering criterion (since the paths began almost simultaneously), while the ordering in (c) was determined by the tertiary ordering criterion (since the paths began simultaneously and had close x coordinates). Generally, *any* set of ordering rules which depend solely on the initial point of each path can be made to generate inconsistent sortings.

In practice, the possibility of inconsistencies has not been much of a problem. The ordering rules are set up so as to be stable for near-vertical and near horizontal finger configurations; they become unstable when the angle between (the initial points of) two fingers causes the 150 unit threshold to be crossed.[1] Knowing this makes it easy to design gesture sets with consistent path orderings. A more robust solution might be to compute a path ordering relation based on the actual gestures used to train the system.

As stated above, some multiple finger sensing devices, such as the DataGlove, do not require any path sorting. To use the DataGlove as input to the multi-path gesture recognizer described below, one approach that could be taken is to compute the paths (in three-space over time) of each fingertip, using the measured angles of the various hand joints. This will result in five sorted paths (one for each finger) which would be suitable as input into the multi-path recognition algorithm. (Of course, the lack of explicit signaling in the DataGlove still leaves the problem of determining the start and

[1] In retrospect, the 150 unit windows make the sorting more complicated then it need be. Using the coordinate whose difference is the largest (for simultaneous paths) makes the algorithm more predictable: it will become inconsistent when the initial points of two paths form an angle close to $-45°$ from the horizontal.

end of the gesture.)

5.3 Multi-path Recognition

Like the single path recognizers described in Chapter 3, the multi-path recognizer is trained by specifying a number of examples for each gesture class. The recognizer consists of a number of single-path classifiers, and a global feature classifier. These classifiers all use the statistical classification algorithm developed in Chapter 3. The differences are mainly in the sets of features used, as described in Section 5.5.

Each single-path classifier discriminates between gestures of a particular sorting index. Thus, there is a classifier for the first path of a gesture, another for the second path, and another for the third path. (The current implementation ignores all paths beyond the third, although it takes the actual number of paths into account.) When a multi-path gesture is presented to the system for classification, the paths are sorted (as described above) and the first path is classified using the first path classifier, and so on, resulting in a sequence of single-path classes.

The sequence of path classes which results is then submitted to a *decision tree*. The root node of the tree has slots pointing to subnodes for each possible class returned by the first path classifier. The subnode corresponding to the class of the first path is chosen. This node has slots pointing to subnodes for each possible class returned by the second path classifier. Some of these slots may be null, indicating that there is no expected gesture whose first and second path classes are the ones computed. In this case the gesture is rejected. Otherwise, the subnode corresponding to the class of the second path is chosen. The process is repeated for the third path class, if any.

Once the entire sequence of path classes is considered there are three possibilities. If the sequence was unexpected, the multi-path gesture is rejected since no node corresponding to this sequence exists in the decision tree. If the node does exist, the multi-path classification may be unambiguous, meaning only one multi-class gesture corresponds to this particular sequence of single-path classes. Or, there may be a number of multi-path gestures which correspond to this sequence of path classes. In this case, a global feature vector (one which encompasses information about all paths) is computed, and then classified by the global feature classifier. This class is used to choose a further subnode in the decision tree, which will result in the multi-path gesture either being classified individually or rejected. The intent is that, if needed, the global feature class is essentially appended to a sequence of path classes; some care is thus necessary to insure that the global feature classes are not confused with path classes.

Figure 5.3 shows an example of the use of a decision tree to classify multi-path gestures. The multi-path classifier recognizes four classes. Each class is composed of two paths. There are only two possible classes for the first path (path 0), since classes P, Q, and S all have similar first paths. Similarly, Q and S have similar second paths, so there are only three distinct possibilities for path 1. Since Q and S have identical path components, the global classifier is used to discriminate between these two, adding another level in the decision tree. The classification of the example input is indicated by dotted lines.

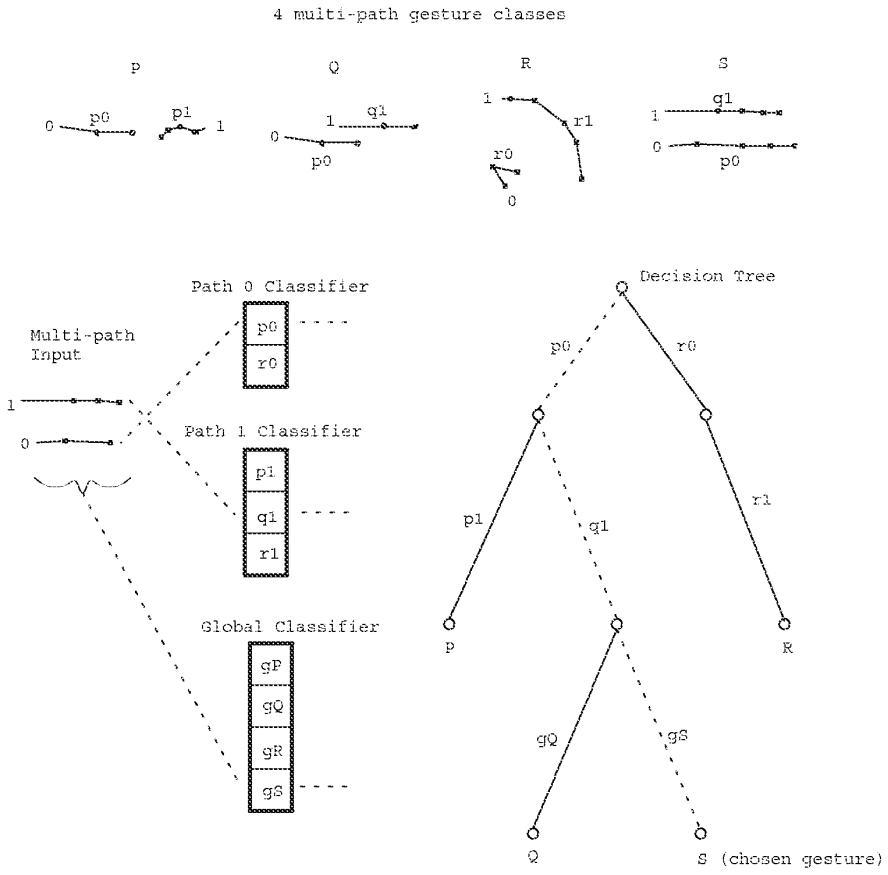


Figure 5.3: Classifying multi-path gestures

At the top are examples of four two-path gestures expected by this classifier, and at the left a two-path gesture to be classified. Path 0 of this gesture is classified (by the path 0 classifier) as path p0*, and path 1 as* q1*. These path classifications are used to traverse the decision tree, as shown by the dotted lines. The tree node reached is ambiguous (having children* Q *and* S*) so global features are used to resolve the discrepancy, and the gesture is recognized as class* S*.*

5.4 Training a Multi-path Classifier

The training algorithm for a multi-path classifier uses examples of each multi-path gesture class (typically ten to twenty examples of each class) to create a classifier. The creation of a multi-path classifier consists of the creation of a global classifier, a number of path classifiers, and a decision tree.

5.4.1 Creating the statistical classifiers

The path classifiers and the global classifiers are created using the statistical algorithm described in Chapter 3. The paths of each example are sorted, the paths for a given sorting index in each class forming a class used to train the path classifier for that index.

For example, consider training a multi-path classifier to discriminate between two multi-path gesture classes, A and B, each consisting of two paths. Gesture class A consists of two path classes, A_1 and A_2, the subscript indicating the sorting indices of the paths. Similarly, class B consists of path classes B_1 and B_2. The first path in all the A examples form the class A_1, and so on. The examples are used to train path classifier 1 to discriminate between A_1 and B_1, and path classifier 2 to discriminate between A_2 and B_2. The global features of A and B are used to create the global classifier, nominally able to discriminate between two classes of global features, A_G and B_G.

Within a given sorting index, it is quite possible and legitimate for paths from different gesture classes to be indistinguishable. For example, path classes A_1 and B_1 may both be straight right strokes. (Presumably A and B are distinguishable by their second paths or global features.) In this case it is likely that examples of class A_1 will be misclassified as B_1 or vice versa. It is desirable to remove these ambiguities from the path classifier by combining all classes which could be mistaken for each other into a single class.

A number of approaches could be taken for detecting and removing ambiguities from a statistical classifier. One possible approach would be to compute the Mahalanobis distance between each pair of classes, merging those below a given threshold. Another approach involves applying a clustering algorithm [74] to all the examples, merging those classes whose members are just as likely to cluster with examples from other classes as their own. A third approach is to actually evaluate the actual performance of a classifier which attempts to distinguish between possibly ambiguous classes; the misclassifications of the classifier then indicate which classes are to be merged. The latter approach was the one pursued here.

A naive approach for evaluating the performance of a classifier would be to construct the classifier using a set of examples, and then testing the performance of the classifier on those very same examples. This approach obviously underestimates the ambiguities of the classes since the classifier will be biased toward correctly classifying its training examples [62]. Instead, a classifier is constructed using only a small number of the examples (typically five per class) and then uses the remaining examples to evaluate the constructed classifiers. Misclassifications of the examples then indicate classes which are ambiguous and should be merged. In practice, thresholds must be established so that a single or very small percentage of misclassifications does not cause a merger.

Mathematically, combining classes is a simple operation. The mean vector of the combined class is computed as the average of the mean vectors of the component classes, each weighted by

the relative number of examples in the class. A similar operation computes a composite average covariance matrix from the covariance matrices of the classes being combined.

The above algorithm, which removes ambiguities by combining classes, is applied to each path classifier as well as the global classifier. It remains now only to construct the decision tree for the multi-path classifier.

5.4.2 Creating the decision tree

A decision tree node has two fields: `mclass`, a pointer to a multi-path gesture class, and `next`, a pointer to an array of pointers to its subnodes. To construct the decision tree, a root node is allocated. Then, during the *class phase*, each multi-path gesture class is considered in turn. For each, a sequence of path classes (in sort index order), with its global feature class appended, is constructed. Nodes are created in the decision tree in such a way that by following the sequence a leaf node whose `mclass` value is the current multi-path gesture class is reached. This creates a decision tree which will correctly classify all multi-class gesture whose component paths and global features are correctly classified.

Next, during the *example phase*, each example gesture is considered in turn. The paths are sorted and classified, as are the global features. A sequence is constructed and the class of the gesture is added to the decision tree at the location corresponding to this sequence as before. Normally, the paths and global features of the gesture will have been classified correctly, so there would already be a node in the tree corresponding to this sequence. However, if one of the paths or the global feature vector of the gesture was classified incorrectly, a new node may be created in the decision tree, and thus the same classification mistake in the future will still result in a correct classification for the gesture.

When attempting to add a class using a sequence whose components are misclassifications, it is possible that the decision tree node reached already has a non-null `mclass` field referring to a different multi-path gesture class than the one whose example is currently being considered. This is a *conflict* and is resolved by ignoring the current example (though a warning message is printed). Ignoring all but the first instance of a sequence insures that the sequences generated during the class phase will take precedence over those generated during the example phase. Of course, a conflict occurring during the class phase indicates a serious problem, namely a pair of gesture classes between which the multi-path classifier is unable to discriminate.

During decision tree construction, nodes that have only one global feature class entry with a subnode have their `mclass` value set to the same gesture class as the `mclass` value of that subnode. In other words, sequences that can be classified without referring to their global feature class are marked as such. This avoids the extra work (and potential for error) of global feature classification.

5.5 Path Features and Global Features

The classification of the individual paths and of the global features of a multi-path gesture are central to the multi-path gesture recognition algorithm discussed thus far. This section describes the particular feature vectors used in more detail.

The classification algorithm used to classify paths and global features is the statistical algorithm discussed in Chapter 3, thus the criteria for feature selection discussed in section 3.3 must be addressed. In particular, only features with Gaussian-like distributions that can be calculated incrementally are considered.

The path features include all the features mentioned in Chapter 3. One additional feature was added: the starting time of the path relative to the starting time of the gesture. Thus, for example, a gesture consisting of two fingers, one above the other, which enter the field of view of the Sensor Frame simultaneously and move right in parallel can be distinguished from a gesture in which a single finger enters the field first, and while it is moving right a second finger is brought into the viewfield and moves right. In particular, the classifier (for the second sorting index) would be able to discriminate between a path which begins at the start of the gesture and one which begins later. The path start time is also used for path sorting, as described in section 5.2.

The main purpose of the global feature vector is to discriminate between multi-path gesture classes whose corresponding individual component paths are indistinguishable. For example, two gestures both consisting of two fingers moving right, one having the fingers oriented vertically, the other horizontally. Or, one having the fingers about one half inch apart, the other two inches apart.

The global features are the duration of the entire gesture, the length of the bounding box diagonal, the bounding box diagonal angle (always between 0 and $\pi/2$ so there are no wrap-around problems), the length, sine and cosine between the first point of the first path and the first point of the last path (referring to the path sorting order), and the length, sine, and cosine between the first point of the first path and the last point of the last path.

Another multi-path gesture attribute, which may be considered a global feature, is the actual number of paths in the gesture. The number of paths was not included in the above list, since it is not included in the vector input to the statistical classifier. Instead, it is required that all the gestures of a given class have the same number of paths. The number of paths must match exactly for a gesture to be classified as a given class. This restriction has an additional advantage, in that knowing exactly the number of paths simplifies specifying the semantics of the gesture (see Section 8.3.2).

The global features, crude as they might appear, in most cases enable effective discrimination between gesture classes which cannot be classified solely on the basis of their constituent paths.

5.6 A Further Improvement

As mentioned, the multi-path classifier has a path classifier for each sorting index. The path classifier for the first path needs to distinguish between all the gestures consisting only of a single path, as well as the first path in those gestures having two or more paths. Similarly, the second path classifier must discriminate not only between the second path of the two-path gestures, but also the second path of the three path gestures, and so on. This places an unnecessary burden on the path classifiers. Since gesture classes with different numbers of paths will never be confused, there is no need to have a path classifier able to discriminate between their constituent paths. This observation leads to a further improvement in the multi-path recognizer.

The improvement is instead of having a single multi-path recognizer for discriminating between multi-path gestures with differing numbers of paths, to have one multi-path gesture recognizer, as

described above, for each possible number of paths. There is a multi-path recognizer for gestures consisting of only one path, another for two-path gestures, and so on, up until the maximum number of paths expected. Each path classifier now deals only with those paths with a given sorting index from those gestures with a given number of paths. The result is that many of the path classifiers have fewer paths to deal with, and improve their recognition ability accordingly.

Of course, for input devices in which the number of paths is fixed, such as the DataGlove, this improvement does not apply.

5.7 An Alternate Approach: Path Clustering

The multi-path gesture recognition implementation for the Sensor Frame relies heavily on path sorting. Path sorting is used to decide which paths are submitted to which classifiers, as well as in the global feature calculation. Errors in the path sorting (*i.e.* similar gestures having their corresponding paths end up in different places in the path ordering) are a potential source of misclassifications. Thus, it was thought that a multi-path recognition method that avoided path sorting might potentially be more accurate.

5.7.1 Global features without path sorting

The first step was to create a global feature set which did not rely on path sorting. As usual, a major design criterion was that a small change in a gesture should result in a small change in its global features. Thus, features which depend largely upon the precise order that paths begin cannot be used, since two paths which start almost simultaneously may appear in either order. However, such features can be weighted by the difference in starting times between successive paths, and thus vary smoothly as paths change order. Another approach which avoids the problem is to create global features which depend on, say, every pair of paths; these too would be immune to the problems of path sorting.

The global features are based on the previous global features discussed. However, for each feature which relied on path ordering there, two features were used here. The first was the previous feature weighted by path start time differences. For example, one feature is the length from the first point of the first path to the first point of the last path, multiplied by the difference between the start times of the first and second path, and again multiplied by the difference between the start times of the last and next to last path. The second was the sum of the feature between every pair path, such as the sum of the length between the start points of every pair of paths. For the sine and cosine features, the sum of the absolute values was used.

5.7.2 Multi-path recognition using one single-path classifier

Path sorting allows there to be a number of different path classifiers, one for the first path, one for the second, and so on. To avoid path sorting, a single classifier is used to classify all paths. Referring to the example in Section 5.4, a single classifier would be used to distinguish between A_1, A_2, B_1, and B_2.

Once all the paths in a gesture are classified, the class information needs to be combined to produce a classification for the gesture as a whole. As before, a decision tree is used. However, since path sorting has been eliminated, there is now no apparent order of the classes which will make up the sequence submitted to the decision tree. To remedy this, each path class is assigned an arbitrary distinct integer during training. The path class sequence is sorted according to this integer ranking (the global feature classification remains last in the sequence) and then the decision tree is examined. The net result is that each node in the decision tree corresponds to a set (rather than a sequence) of path classifications. (Actually, as will be explained later, each node corresponds to a multiset.)

In essence, the recognition algorithm is very simple: the lone path classifier determines the classes of all the paths in the gesture; this set of path classes, together with the global feature class, determines the class of the gesture. Unfortunately, this explanation glosses over a serious conceptual difficulty: In order to train the path classifier, known instances of each path class are required. But, without path sorting, how is it possible to know which of the two paths in an instance of gesture class A is A_1 and which is A_2? One of the paths of the first A example can arbitrarily be called A_1. Once this is done, which of the paths in each of the other examples of class A are in A_1?

Once asked, the answer to this question is straightforward. The path in the second instance of A which is similar to the path previously called A_1 should also be called A_1. If a gesture class has N paths, the goal is to divide the set of paths used in all the training examples of the class into N groups, each group containing exactly one path from each example. Ideally, the paths forming a group are similar to each other, or, in other words, they correspond to one another.

Note that path sorting produces exactly this set of groups. Within all the examples of a given gesture class, all paths with the same sorting index form a group. However, if the purpose of the endeavor is to build a multi-path recognizer which does not use path sorting, it seems inappropriate to resort to it during the training phase. Errors in sorting the example paths would get built into the path classifier, likely nullifying any beneficial effects of avoiding path sorting during recognition.

Another way to proceed is by analogy. Within a given gesture class, the paths in one example are compared to those of another example, and the corresponding paths are identified. The comparisons could conceivably be based on the feature of the path as well as the location and timing of the path. This approach was not tried, though in retrospect it seems the simplest and most likely to work well.

5.7.3 Clustering

Instead, the grouping of similar paths was attempted. The definition of similarity here only refers to the feature vector of the path. In particular, the relative location of the paths to one another was ignored. To group similar paths together solely on the basis of their feature vectors, a statistical procedure known as *hierarchical cluster analysis* [74] was applied.

The first step in cluster analysis is to create a triangular matrix containing the distance between every pair of samples, in this case the samples being every path of every example of a given class. The distance was computed by first normalizing each feature by dividing by the standard deviation. (The typical normalization step of first subtracting out the feature mean was omitted since it has no effect on the difference between two instances of a feature.) The distance between each pair of

example path feature vectors was then calculated as the sum of the squared differences between the normalized features.

From this matrix, the clustering algorithm produces a cluster tree, or *dendrogram*. A dendrogram is a binary tree with an additional linear ordering on the interior nodes. The clustering algorithm initially considers each individual sample to be in a group (cluster) of its own, the distance matrix giving distances between every pair of groups. The two most similar groups, *i.e.* the pair corresponding to the smallest entry in the matrix, are combined into a single group, and a node representing the new group is created in the dendrogram, the subnodes of which refer to the two constituent groups. The distance matrix is then updated, replacing the two rows and columns of the constituent groups with a single row and column representing the composite group.

The distance of the composite group to each other group is calculated as a function of the distances of the two constituents to the other group. Many such combining functions are possible; the particular one used here is the *group average* method, which computes the distance of the newly formed group to another group as the average (weighted by group size) of the two constituent groups to the other group. After the matrix is updated, the process is repeated: the smallest matrix element is found, the two corresponding groups combined, and the matrix updated. This continues until there is only one group left, representing the entire sample set. The order of node creation gives the linear order on the dendrogram nodes, nodes created early having subnodes whose groups are more similar than nodes created later.

Figure 5.4 shows the dendrogram for the paths of 10 3-path clasp gestures, where the thumb moves slightly right while the index and middle fingers move left. The leaves of the dendrogram are labeled with the numbers of the paths of the examples. Notice how all the right strokes cluster together (one per example), as do all the left strokes (two per example).

Using the dendrogram, the original samples can be broken into an arbitrary (between one and the number of samples) number of groups. To get N groups, one simply discards the top $N-1$ nodes of the dendrogram. For example, to get two groups, the root node is discarded, and the two groups are represented by the two branches of the root node.

Turning back now to the problem of finding corresponding paths in examples of the same multi-path gesture class, the first step is to compute the dendrogram of all the paths in all examples of the gesture. The dendrogram is then traversed in a bottom-up (post-order) fashion, and at each node a histogram that indicates the count of the number of paths for each example is computed. The computation is straightforward: for each leaf node (*i.e.* for each path) the count is zero for all examples except the one the path came from; for each interior node, each element of the histogram is the sum of the corresponding elements of the subnode's histogram.

Ideally, there will be nodes in the tree whose histogram indicates that all the paths below this node come from different examples, and that each example is represented exactly once. In practice, however, things do not work out this nicely. First, errors in the clustering sometimes group two paths from the same example together before grouping one path from every example. This case is easily handled by setting a threshold, *e.g.* by accepting nodes in which paths from all but two examples appear exactly once in the cluster.

The second difficulty is more fundamental. It is possible that two or more paths in a single gesture are quite similar (remember that relative path location is being ignored). This is actually

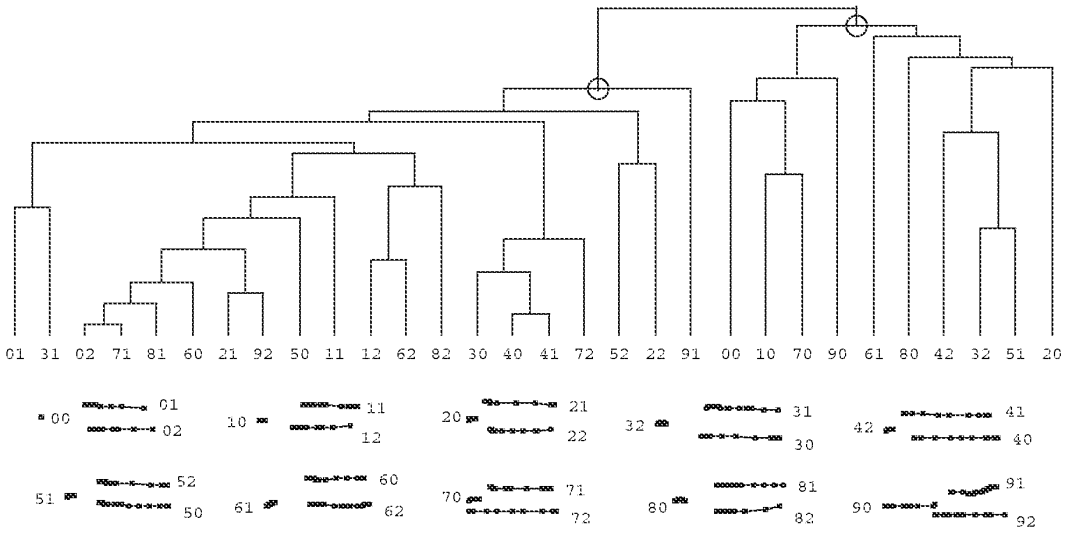


Figure 5.4: Path Clusters

This shows the result of clustering applied to the thirty paths of the ten three-path clasp *gestures shown. Each* clasp *gesture has a short, rightward moving path and two similar, long leftward moving paths. The hierarchical clustering algorithm groups similar paths (or groups of paths) together. The height of an interior node indicates the similarity of its groups, lower nodes being more similar. Note that the right subtree of the root contains 10 paths, one from each multi-path gesture. It is thus termed a good cluster (indicated by a circle on the graph), and its constituent paths correspond. The left subtree containing 20 paths, two from each gesture, is also a good cluster. Had one of its descendants been another good cluster (containing approximately 10 paths, one from each gesture), it would have been concluded that all three paths of the* clasp *gesture are different, with the corresponding paths given by the good clusters. As it happened, no descendant of the left subtree was a good cluster, so it is concluded that two of the paths within the* clasp *gesture are similar, and will thus be treated as examples of one single-path class.*

common for Sensor Frame gestures that are performed by moving the elbow and shoulder while keeping the wrist and fingers rigid. For these paths, it is just as likely that the two paths of the same example be grouped together as it is that corresponding paths of different examples be grouped together. Thus, instead of a histogram that shows one path from each example, ideally there will be a node with a histogram containing two paths per example. This is the case in figure 5.4.

Call a node which has a histogram indicating an equal (or almost equal) number of paths from each example a *good cluster*. The search for good clusters proceeds top down. The root node is surely a good cluster; *e.g.* given examples from a three path gesture class, the root node histogram will indicate three paths from each example gesture. If no descendants of the root node are good clusters, that indicates that all the paths of the gesture are similar. However, if there are good clusters below the root (with fewer examples per path than the root), that indicates that not all the paths of the gesture are similar to each other. In the three path example, if, say, one subnode of the root node was a good cluster with one path per example, those paths form a distinct path class, different than the other path classes in the gesture class. The other subnode of the root will also be a good cluster, with two paths per example. If there does not exist a descendant of that node which is a good cluster with one path per example, that indicates that the two gesture paths classes are similar. Otherwise, good clusters below that node (there will likely be two) indicate that each path class in the gesture class is different. The cluster analysis, somewhat like the path sorting, indicates which paths in each example of a given gesture class correspond. (Good clusters are indicated by circles in figure 5.4.)

Occasionally, there are *stragglers*, paths which are not in any of the good clusters identified by the analysis. An attempt is made to put the stragglers in an appropriate group. If an example contains a single straggler it can easily be placed in the group which is lacking an example from this class. If an example contains more than one straggler, they are currently ignored. If desired, a path classifier to discriminate between the good clusters could be created and then used to classify the stragglers. This was not done in the current implementation since there was never a significant number of stragglers.

Once the path classes in each gesture class have been identified using the clustering technique, a path classifier is trained which can distinguish between every path class of every gesture class. Note that it is possible for a path class to be formed from two or more paths from each example of a single gesture class, if the cluster analysis indicated the two paths were similar. If analogy techniques were used to separate such a class into multiple "one path per example" classes, the resulting classifier would ambiguously classify such paths. In any case, ambiguities are still possible since different gesture classes may have similar gesture paths. As in Section 5.4, the ambiguities are removed from the classifier by combining ambiguous classes into a single class. Each (now unambiguous) class which is recognized by the path classifier is numbered so as to establish a canonical order for sorting path class sequences during training and recognition.

5.7.4 Creating the decision tree

After the single-path and global classifiers have been trained, the decision tree must be constructed. As before, in the class phase, for each multi-path gesture class, the (now unambiguous) classes of each constituent path are enumerated. Since two paths in a single gesture class may be similar, this enumeration of classes may list a single class more than once, and thus may be considered a

multiset. The list of classes is sequenced into canonical order, the global feature class appended, and the resulting sequence is used to add the multi-path class to the decision tree. As before, a conflict, due to the fact that two different gesture classes have the same multiset of path classes, is fatal.

Next comes the example phase. The paths of each example gesture are classified by the single path classifier, and the resulting sequence (in canonical order with the global feature class appended) is used to add the class of the example to the decision tree. Usually no work needs to be done, as the same sequence has already been used to add this class (usually in the class phase). However, if one of the paths in the sequence has been misclassified, adding it to the decision tree can improve recognition, since this misclassification may occur again in the future. Conflicts here are not fatal, but are simply ignored on the assumption that the sequences added in the class phase are more important than those added in the example phase.

5.8 Discussion

Two multi-path gesture recognition algorithms have been described, which are referred to as the "path sorting" and the "path clustering" methods. In situations where there is no uncertainty as to the path index information (*e.g.* a DataGlove, since the sensors are attached to the hand) then the path-sorting method is certainly superior. However, with input devices such as the Sensor Frame, the path sorting has to be done heuristically, which increases the likelihood of recognition error.

The path-clustering method avoids path sorting and its associated errors. However, other sources of misclassification are introduced. One single-path classifier is used to discriminate between all the path classes in the system, so will have to recognize a large number of classes. Since the error rate of a classifier increases with the number of classes, the path classifier in a path-clustering algorithm will never perform as well as those in a path-sorting algorithm. A second source of error is in the clustering itself; errors there cause errors in the classifier training data, which cause the performance of the path classifier to degrade. One way around this is to cluster the paths by hand rather than by having a computer perform it automatically. This needed to be done with some gesture classes from the Sensor Frame, which, because of glitches in the tracking hardware, could not be clustered reliably.

In practice, the path-sorting method always performed better. The poor performance of the path-clustering method was generally due to the noisy Sensor Frame data. It is however difficult to reach a general conclusion, as all the gesture sets upon which the methods were tested were designed with the path sorting algorithm in mind. It it easy to design a set of gestures that would perform poorly using sorted paths. One possibility for future work is to have a parameterizable algorithm for sorting paths, and choose the parameters based on the gesture set.

The Sensor Frame itself was a significant source of classification errors. Sometimes, the knuckles of fingers curled so as not to be sensed would inadvertently break the sensing plane, causing extra paths in the gesture (which would typically then be rejected). Also, three fingers in the sensing plane can easily occlude each other with respect to the sensors, making it difficult for the Sensor Frame to determine each finger's location. The Sensor Frame hardware usually knew from recent history that there were indeed three fingers present, and did its best to estimate the positions of each. However, the resulting data often had glitches that degraded classification, sometimes by confusing

the tracking algorithm. It is likely that additional preprocessing of the paths before recognition would improve accuracy. Also, the Sensor Frame itself is still under development, and it is possible that such glitches will be eliminated by the hardware in the future.

Another area for future work is to apply the single-path eager recognition work described in Chapter 4 to the eager recognition of multi-path gestures. Presumably this is simply a matter of eagerly recognizing each path, and combining the results using the decision tree. How well this works remains to be seen.

It would also be possible to apply the multi-path algorithm to the recognition of multi-stroke gestures. The path sorting in this case would simply be the order that the strokes arrive. To date, this has not been tried.

5.9 Conclusion

In this chapter, two methods for multi-path gesture recognition were discussed and compared. Each classifies the paths of the gesture individually, uses a decision tree to combine the results, and uses global features to resolve any lingering ambiguities. The first method, path sorting, builds a separate classifier for each path in a multi-path gesture. In order to determine which path to submit to which classifier, either the physical input device needs to be able to tell which finger corresponds to which path, or a path sorting algorithm numbers the paths. The second method, path clustering, avoids path sorting (which has an arbitrary component) by using one classifier to classify all the paths in a gesture.

In general, the path sorting method proved superior. However, when the details of the path sorting algorithm are known it is possible to design a set of gestures which will be poorly recognized due to errors in the path sorting. That same knowledge can also be used to design gesture sets that will not run into path sorting problems.

Chapter 6

An Architecture for Direct Manipulation

This chapter describes the GRANDMA system. GRANDMA stands for "Gesture Recognizers Automated in a Novel Direct Manipulation Architecture." This chapter concentrates solely on the architecture of the system, without reference to gesture recognition. The design and implementation of gesture recognizers in GRANDMA is the subject of the next chapter.

GRANDMA is an object-oriented toolkit similar to those discussed in Section 2.4.1. Like those toolkits, is it based on the model-view-controller (MVC) paradigm. GRANDMA also borrows ideas from event-based user-interface systems such as Squeak [23], ALGAE [36], and Sassafras [54].

GRANDMA is implemented in Objective C [28] on a DEC MicroVax-II running UNIX and the X10 window system.

6.1 Motivation

Building an object-oriented user interface toolkit is a rather large task, not to be undertaken lightly. Furthermore, such toolkits are only peripherally related to the topic at hand, namely gesture-based systems. Thus, the decision to create GRANDMA requires some justification.

A single idea motivated the author to use object-oriented toolkits to construct gesture-based systems: gestures should be associated with objects on the screen. Just as an object's class determines the messages it understands, the author believed the class could and should be used to determine which gestures an object understands. The ideas of inheritance and overriding then naturally apply to gestures. The analogy of gestures and messages is the central idea of the "systems" portion of the current work.

It would have been desirable to integrate gestures into an existing object oriented toolkit, rather than build one from scratch. However, at the time the work began, the only such toolkits available were Smalltalk-80's MVC [70] and the Pascal-based MacApp [115], neither of which ran on the UNIX/C environment available to (and preferred by) the author. Thus, the author created GRANDMA.

The existing object-oriented user interface systems tend to have very low-level input models, with device dependencies spread throughout the system. For example, some systems require views to respond to messages such as `middleButtonDown` [28]; others use event structures that can

only represent input from a fixed small set of devices [102]. In general, the output models of existing systems seem to have received much more attention than the input models. One goal of GRANDMA was to investigate new architectures for input processing.

6.2 Architectural Overview

Figure 6.1 shows a general overview of the architecture of the GRANDMA system. In order to introduce the architecture to the reader, the response to a typical input event is traced. But first, a brief description of the system components is in order.

GRANDMA is based on the Model-View-Controller (MVC) paradigm. Models are application objects. They are concerned only with the semantics of the application, and not with the user interface. Views are concerned with displaying the state of models. When a model changes, it is the responsibility of the model's view(s) to relay that change to the user. Controllers are objects which handle input. In GRANDMA, controllers take the form of *event handlers*.[1] A single passive event handler may be associated with many view objects; when input is first initiated toward a view, one of the view's passive event handlers may activate (a copy of) itself to handle further input.

6.2.1 An example: pressing a switch

Consider a display consisting of several toggle switches. Each toggle switch has a model, which is likely to be an object containing a boolean variable. The model has messages to set and retrieve the value of the variable, which are used by the view to display the state of the toggle switch, and by the event handler to change the state of the toggle.

When the mouse cursor is moved over one of the switches and, say, the left mouse button is pressed, the window manager informs GRANDMA, which raises an input `Pick` event. The event is an object which groups together all the information about the event: the fact that it was a mouse event, which button was pressed, and, most significantly, the coordinates of the mouse cursor.

Raising an event causes the *active event handler* list to be searched for a handler for this event. In turn, each event handler on the list is asked if it wishes to handle the event. Assuming none of the other handlers will be interested in the event, the last handler in the list, called the `XYEventHandler`, handles the event. This is what happens in the case of pressing the toggle switch.

The `XYEventHandler` is able to process any event at a location (*i.e.* events with X-Y coordinates). The handler first searches the *view database* and constructs a list of views which are "under" the event, in other words, views that are at the given event location. The search is simple: each view has a rectangular region in which it is included; if the event location is in the rectangle, the view is added to the list. In the switch example, the list of views consists of the indicated toggle switch view followed by the view representing the window in which the toggle switch is drawn.

[1] The distinction between controllers and event handlers is in the way each interacts with the underlying layer that generates input events. Once activated, controllers loop, continually calling the input layer for all input events until the interaction completes. In other words, controllers take control, forcing the user to complete one interaction before initiating the next. In contrast, event handlers are essentially called by the input layer whenever input occurs. It is thus possible to interact simultaneously with multiple event handlers, for example via multiple devices.

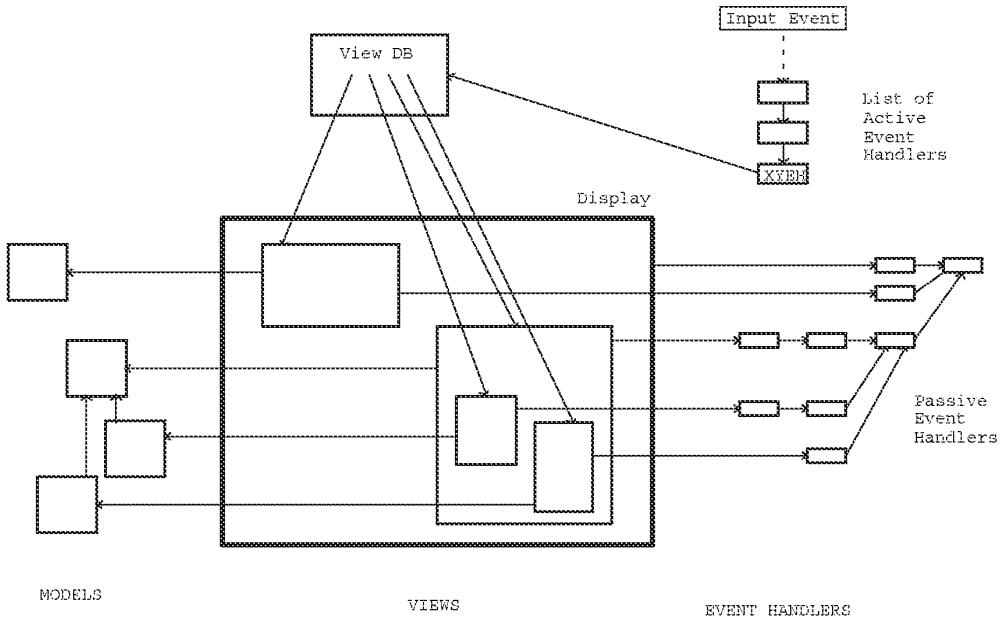


Figure 6.1: GRANDMA's Architecture

In GRANDMA, user actions cause events to be raised (i.e. pressing a mouse button raises a `Pick` *event). Each handler on the active event handler list is asked, in order, if it wishes to handle the event. The* `XYEventHandler`*, last on the list, is asked only if none of the previous active handlers have consumed the event. For an event with a screen location (i.e. a mouse event), the* `XYEventHandler` *uses the view database to determine the views at the given screen location, and asks each view (from front to back) if it wishes to handle the event. To answer, a view consults its list of passive event handlers, some associated with the view itself, others associated with the view's class and superclasses, to see if one of those is interested in the event. If so, that passive handler may activate itself, typically by placing a copy of itself at the front of the active event handler list. This enables subsequent events to be handled efficiently, short-circuiting the elaborate search for a handler initiated by the* `XYEventHandler`*. An event handler only consumes events in which it is interested, allowing other events to propagate to other event handlers.*

The views are then queried starting with the foreground view. First, a view is asked if the event location is indeed over the view; this gives an opportunity for a non-rectangular view to respond only to events directly over it. If the event is indeed over the view, the view is then asked if it wishes to handle the input event. The search proceeds until a view wishes to handle the event, or all the views under the event have declined. In the example, the toggle switch view handles the event, which would then not be propagated to the window view.

A view does not respond directly to a query as to whether it will handle an input event. Instead, that request is passed to the view's *passive event handlers*. Associated with each view is a list of event handlers that handle input for the view; a single passive event handler is often shared among many views in the system. The passive event handlers are each asked about the input in turn; the search stops when one decides to handle the input. In the example, the toggle switch has a toggle switch event handler first on its list of passive handlers that would handle the `Pick` event.

A passive event handler that has decided to handle an event may *activate* a copy or instance of itself, *i.e.* place the copy or instance in the active event handler list. Or, it may not, choosing to do all the work associated with the event when it gets the event. For example, a toggle switch may either change state immediately when the mouse button is pressed over the switch, or it may simply highlight itself, changing state only if the button is released over the switch. In the former case, there is no need to activate an event handler; the passive handler itself can change the state of the switch.

In the latter case, the passive handler activates a copy of itself which first highlights the switch, and then monitors subsequent input to watch if the cursor remains over the view. If the cursor moves away from the view, the active event handler will turn off the highlighting of the switch, and may (depending on the kind of interaction wanted) deactivate itself. Finally, if the mouse button is released over the switch, the active event handler will, through the view, toggle the state of the switch (and associated model), and then deactivate itself.

As noted above, active handlers are asked about events before the view database is searched and any passive handlers queried. Thus, in the switch example, subsequent mouse movements made while the button is held down, or the release of the mouse button, will be handled very efficiently since the active handler is at the head of the active event handler list.

6.2.2 Tools

The *tool* is one component of GRANDMA's architecture not mentioned in the above example. A tool is an object that raises events, and it is through such events that tools operate on views (and thus models) in the system. An event handler may be considered the mechanism through which a tool operates upon a view. The interaction is by no means unidirectional: some event handlers cause views to operate upon tools as well. In addition to operating on views directly, event handlers may themselves raise events, as will be seen.

Every event has an associated tool which typically refers to the device that generated the event. For example, a system with two mice would have two `MouseTool` objects, and the appropriate one would be used to identify which mouse caused a given `Pick` event. When asked to handle an event, an active handler typically checks that the event's tool is the same one that caused the handler

to be activated in the first place. In this manner, the active event handler ignores events not intended for it.

Tools are also involved when one device emulates another. For example, a Sensor Frame may emulate a mouse by having an active handler that consumes events whose tool is a `SensorFrame` object, raising events whose tool is a `MouseTool` in response. That `MouseTool` does not correspond to a real mouse; rather, it allows the Sensor Frame to masquerade as a mouse.

Tools do not necessarily refer to hardware devices. *Virtual tools* are software objects (typically views) that act like input hardware in that they may generate events. For example, file views (icons) would be virtual tools when implementing a Macintosh-like Finder in GRANDMA. Dragging a file view would cause events to be raised in which the tool was the file view. A passive handler associated with folder (directory) views would be programmed to activate whenever an event whose tool is a file view is dragged over a folder. Thus, in GRANDMA the same mechanism is used when the mouse cursor is dragged over views as when the mouse is used to drag one view over other views.

The typical case, in which a tool has a semantic action which operates upon views that the tool is dropped upon, is handled gracefully in GRANDMA. Associated with every view is the passive `GenericToolOnViewEventHandler`. When a tool is dragged over a view which responds to the tool's action, the `GenericToolOnViewEventHandler` associated with the view activates itself, highlighting the view. Dropping the tool on the view causes the action to occur.[2] Thus, semantic feedback is easy to achieve using virtual tools (see section 6.7.7).

This concludes the brief overview of the GRANDMA architecture. A discussion of the details of the GRANDMA system now follows. A reader wishing to avoid the details may proceed directly to section 6.8, which summarizes the main points while comparing GRANDMA to some existing systems.

6.3 Objective-C Notation

As mentioned, GRANDMA is written in Objective C [28], a language which augments C with object-oriented programming constructs. In this part of the dissertation, program fragments will be written in Objective C.

In Objective C, variables and functions whose values are objects are all declared type `id`, as in

```
id aSet;
```

Variables of type `id` are really pointers, and can refer to any Objective C object, or have the value `nil`. Like all pointers, such variables need to be initialized before they refer to any object:

```
aSet = [Set new]; /* create a Set object */
```

The expression `[o messagename]` is used to send the message `messagename` to the object referred to by `o`. This object is termed the *receiver*, and `messagename` the *selector*. A message send is similar to a function call, and returns a value whose type depends on the selector.

Objective C comes supplied with a number of *factory* objects, also known as *classes*. `Set` is an example of a factory object, and like most factory objects, responds to the message `new` with a

[2] The related case, in which a tool is dragged over a view that acts upon the tool (*e.g.* the trash can), is handled by the `BucketEventHandler`.

newly allocated instance of itself.

Messages may also have parameters, as in

```
id aRect = [Rectangle origin:10:10 corner:20:30];
[aSet add:aRect];
```

The message selector is the concatenation of all the parameter labels (`origin::corner::` in the first case, `add:` in the second). In all cases, there is one parameter after each colon.

A factory's fields and methods are declared as in the following example:

```
= Rect:Object { int x1,y1,x2,y2; }
+ origin:(int)_x1 :(int):_y1 corner:(int)_x2 :(int)_y2 {
    self = [self new];
    x1 = _x1, y1 = _y1, x2 = _x2; y2 = _y2;
    return self;
}
- shiftby:(int)x :(int)y
    { x1 += x; y1 += y; x2 += x; y2 += y; return self; }
```

This declares the factory `Rect` to be a subclass of the factory `Object`, the root of the class hierarchy. Note that the factory declaration begins with the "=" token. A method declared with "+" defines a message which is sent directly to a factory object; such methods often allocate and return an instance of the factory. A method declared with "-" defines a message that is sent directly to instances of the class. The variable `self` is accessible in all method declarations; it refers to the object to which the message was sent (the receiver). When `self` is set to an instance of the object class being defined, the fields in the object can be referenced directly. Thus, as in the `origin::corner::` method, the first step of a factory method is often to reassign `self` to be an instance of the factory, then to initialize the fields of the instance. The usage `[self new]` rather than `[Rect new]` allows the method to work even when applied to a subclass of `Rect` (since in that case `self` would refer to the factory object of the subclass). When the types of methods and arguments are left unspecified, they are assumed to be `id`, and typically methods return `self` when they have nothing better to return (rather than `void`, *i.e.* not returning anything).

When describing a method of a class, the fields and other methods are often omitted, as in

```
= Rect ...
- (int)area { return abs( (x2-x1)*(y2-y1) ); }
```

In Objective C, messages selectors are first class objects, which can be assigned and passed as parameters and then later sent to objects. The construct `@selector(`*message-selector*`)` returns an object of type `SEL`, which is a runtime representation of the message selector:

```
id aRect = [Rect origin:10:5 corner:40:35];
SEL op = flag ?    @selector(area) :  @selector(height);
printf("%d\n", [aRect perform:op]);
```

The rectangle `aRect` will be sent the `area` or `height` message depending on the state of `flag`. The `perform:` message sends the message indicated by the passed `SEL` to an object. Variants of the form `perform:with:with:` allow additional parameters to be sent as well.

The first class nature of message selectors distinguishes Objective C from more static object-oriented languages, notably C++. As they are analogous to pointers to functions in C, `SEL` values

may be considered "pointers" to messages. Objective C includes functions for converting between `SEL` values and strings, and a method for inquiring at runtime whether an object responds to an arbitrary message selector. As will be seen, these Objective C features are often used in the GRANDMA implementation.

In the interest of simplicity, debugging code and memory management code have been removed from most of the code fragments shown below, though they are of course needed in practice. Also, as the code is explained in the text, many of the comments have been removed for brevity.

6.4 The Two Hierarchies

Thus far, two important hierarchies in object-oriented user interface toolkits have been hinted at, and it seems prudent to forestall confusion by further discussing them here. The first one is known as the *class hierarchy*. The class hierarchy is the tree of subclass/superclass relationships that one has in a single-inheritance system such as Objective C. In Objective C, the class `Object` is at the root of the class hierarchy; in GRANDMA classes like `Model`, `View`, and `EventHandler` are subclasses of `Object`; each of these has subclasses of its own (*e.g.* `ButtonView` is a subclass of `View`), and so on. The entire tree is referred to as the class hierarchy, and particular subtrees are referred to by qualifying this term with a class name. In particular, the `View` class hierarchy is the tree with the class `View` at the root, with the subclasses of `View` subnodes of the root, and so on.

The second hierarchy is referred to as the *view hierarchy* or *view tree*. A `View` object typically controls a rectangular region of the display window. The view may have *subviews* which control subareas of the parent view's rectangle. For example, a dialogue box view may have as subviews some radio buttons. Subviews are usually more to the foreground than their parent views; in other words, a subview usually obscures part of its parent's view. Of course, subviews themselves might have subviews, and so on, the entire structure being known as the view tree. In GRANDMA, the root of the view tree is a view corresponding to a particular window on the display; a program with multiple windows will have a view tree for each. It is important not to confuse the view hierarchy with the `View` class hierarchy; the former refers to the superview/subview relation, the latter to the superclass/subclass relation.

6.5 Models

Being a Model/View/Controller-based system, naturally the three most important classes in GRANDMA are `Model`, `View`, and `EventHandler` (the latter being GRANDMA's term for controller). The discussion of GRANDMA is divided into three sections, one for each of these classes. Class `Model` is considered first.

Models are objects which contain application-specific data. Model objects encapsulate the data and computation of the task domain. The MVC paradigm specifies that the methods of models should not contain any user-interface specific code. However, a model will typically respond to messages inquiring about its state. In this manner, a view object may gain information about the model in order to display a representation of the model.

In a number of MVC-like systems, there is no specific class named "`Model`" [28]. Instead, any object may act as a model. However, in GRANDMA, as in Smalltalk-80[70], there is a single class named `Model`, which is subclassed to implement application objects. This has the obvious disadvantage that already existing classes cannot directly serve as models. The advantage is the ease of implementation, and the ability to easily distinguish models from other objects.

One of the tenets of the MVC paradigm is that `Model` objects are independent of their views. The intent is that the user interface of the application should be able to be changed without modifying the application semantics. The effect of this desire for modularity is that a `Model` subclass is written without reference to its views.

However, when the state of a model changes, a mechanism is needed to inform the views of the model to update the display accordingly. The way this is accomplished is for each model to have a list of dependents. Objects, such as views, that wish to be informed when a model changes state register themselves as dependents of the model. By convention, a `Model` object sends itself the `modified` message when it changes; this results in all its dependents getting sent the `modelModified` message, at which time they can act accordingly.

The heart of the implementation of the `Model` class in GRANDMA is simple and instructive:

```
= Model :  Object { id dependents; }
- addDependent:d {
    if(dependents == nil) dependents = [OrdCltn new];
    [dependents add:d];
    return self;
}
- removeDependent:d {
    if(dependents != nil) [dependents remove:d];
    return self;
}
- modified {
    if(dependents != nil) /* send all dependents modelModified */
       [dependents elementsPerform:@selector(modelModified)];
    return self;
}
```

Thus, a `Model` is a subclass of `Object` with one additional field, `dependents`. When a `Model` is first created, its `dependents` field is automatically set to `nil`. The first time a dependent is added (by sending the message `addDependent:`), the `dependents` field is set to a new instance of `OrdCltn`, a class for representing lists of objects. The dependent is then added to the list; it can later be removed by the `removeDependent` message.

`Model` is an *abstract* class; it is not intended to be instantiated directly, but instead only be subclassed. A simple example of a `Model` might be boolean variable (whose view might be a toggle switch):

```
= Boolean :  Model { BOOL state; }
- (BOOL)getState { return state; }
```

```
- setState:(BOOL)_state
    { state = _state; return [self modified]; }
- toggle { state = !state; return [self modified]; }
```

Notice that whenever a Boolean object's state changes, it sends itself the modified message, which results in all of its dependents getting sent the modelModified message.

6.6 Views

The abstract class View, as mentioned, handles the display of Models. It is easily the most complex class in the GRANDMA system; it is over 800 lines of code, and it currently implements 10 factory methods and 67 instance methods (not including those inherited from Object). For brevity, most of the methods will not be mentioned, or are only mentioned in passing.

Views have a number of instance variables (fields):

```
= View : Object {
    id    model;
    id    parent, children;
    id    picture, highlight;
    short xloc, yloc;
    id    box;
    int   state;
}
```

The model variable is the view's connection with its model. Some views have no model; in this case model will be nil. The fields parent and children implement the view tree, parent being the superview of the view, children being a list (OrdCltn) of the subviews of this view. The fields picture and highlight refer to the graphics used to draw and highlight the view, respectively. The graphics are drawn with respect to the origin specified by (xloc, yloc), and are constrained to be within the Rectangle object box. The state field is a set of bits indicating both the current state of the view (set by the GRANDMA system) and the desired state of the view (controllable by the view user).

To illustrate some of View's methods, here is a toggle switch view whose model is the class Boolean described above.

```
= SwitchView: View { }
```

To create a toggle switch view:

```
id aBoolean = [Boolean new];
id aSwitchView = [SwitchView createViewOf:aBoolean];
```

The createViewOf: method of class View allocates a new view object (in this case an instance of SwitchView), sets the model instance variable, and, to add itself to the model's dependents, does [model addDependent:self].

The graphics for the switch are implemented as:

```
= SwitchView ...
- updatePicture {
    id p = [self VbeginPicture];
```

```
    [p rectangle 0:0 :10:10];
    if([model getState])
        [p rectangle 2:2 :8:8];
    [self VendPicture];
    return self;
}
```

The intention is to draw an empty rectangle 10 by 10 pixels in size for a switch whose model's state is `FALSE`, but put a smaller rectangle within the switch when the model's state is `TRUE`.

`View`'s `VbeginPicture` and `VendPicture` methods deal with the `picture` instance variable. (The `V` prefix in the method names is a convention indicating that these messages are intended only to be sent by subclasses of `View`.) `VbeginPicture` creates or initializes the `HangingPicture` object which it returns. The graphics are then directed at the picture, which is in essence a display list of graphics commands. Note how the model's state is queried using the `model` instance variable inherited from class `View`. This is done for efficiency purposes; a more modular way to accomplish the same thing would be `if([[self model] getState])`.

The method `updatePicture` gets called indirectly from View's `modelModified` method:

```
= View ...
- modelModified {
    [self update];
    if(state & V_NOTIFY_CHILDREN)  /* propagate modelModified to kids */
        [children elementsPerform:@selector(modelModified)];
    return self;
}
- update { return [self updatePicture]; }
```

The state bit `V_NOTIFY_CHILDREN` is settable by the creator of a view; it determines whether `modelModified` messages will be propagated to subviews. Often when this bit is turned off, the subclass of `View` overrides the `update` method in order to propagate `modelModified` only to certain of its subviews. (For example, a view whose model is a list might have a subview for each element in the list displayed left to right, and when one element is deleted from the list the view could arrange that only the subviews to the right of the deleted one be redrawn.) In the more typical case, the subclass only implements the `updatePicture` method which redraws the view to reflect the state of the model.

For the switch to be displayed, it needs to be a subview (or a descendant) of a `WallView`. Class `WallView` is the abstraction of a window on the display. An instance of `WallView` is created for each window a program requires, as in:

```
id aWallView = [WallView name:"gdp"];
[aWallView addSubView:[aSwitchView at:50:30]];
```

This fragment creates a window named "gdp." The string "gdp" is looked up in a database (in this case, the .Xdefaults file as administered by the X window system) to determine the initial size and location of the window. The switch is added as a subview to the wall view, and displayed at coordinates (50,30) in the newly created window.

This ends the discussion of the major methods of class `View`. As the need arises, additional methods will be discussed. It is ironic how in this dissertation, largely concerned with input, so much effort was expended on output. The initial intention was to keep the output code as simple as possible while still being usable. Unfortunately, thousands of lines of code were required to get to that point.

6.7 Event Handlers

In GRANDMA, the analogue of MVC controllers are event handlers. When input occurs, it is represented as an *event* which is *raised*. Raising an event results in a search for an active event handler that will handle the event. For many events, the last handler in the active list is a catch-all handler whose function is to search for any views at the event's location. Each such view is asked if it wishes to handle the event; the view then asks each of its passive event handlers if it wants to handle the event. As mentioned, a single passive event handler may be associated with many different views. A passive event handler may activate a copy or instance of itself in response to input.

Warning to readers: due to this dissertation's focus on input, this is necessarily a very long section.

6.7.1 Events

Before event handlers can be discussed in detail, it is helpful to make concrete exactly what is meant by "event." All events are instances of some subclass of `Event`:

```
= Event : Object { id instigator; }
- instigator { return instigator; }
- instigator:_instigator
    { instigator = _instigator; return self; }
```

The instigator of an event is the object posting the event. All window manager events are instigated by an instance of class `Wall`.[3]

Figure 6.2 shows the `Event` class hierarchy. (Like `instigator` in class `Event`, each instance variable shown has a method to set and a method to retrieve its value.) The most important subclass is `WallEvent`, which is an event associated with a window, and thus usually raised by (the GRANDMA interface to) the window manager. A `KeyEvent` is generated when a character is typed by the user. A `RefreshEvent` is generated when the window manager requests that a particular window be redrawn.

The subclasses of the abstract class `DragEvent`, when raised by the window manager, indicate a mouse event. In these cases, the `tool` field is an instance of `GenericMouseTool` or one of its subclasses. When a mouse button is pressed, a `PickEvent` is generated. The field `loc` is a

[3]The instigator is mostly used for tracing and debugging. Occasionally, it is used for a quick check by an active event handler that wishes to insure it is only handling events raised by the same object that raised the event which activated the handler in the first place. Most active handlers do not bother with this check, being content to simply check that the tool (rather than the instigator) is the same.

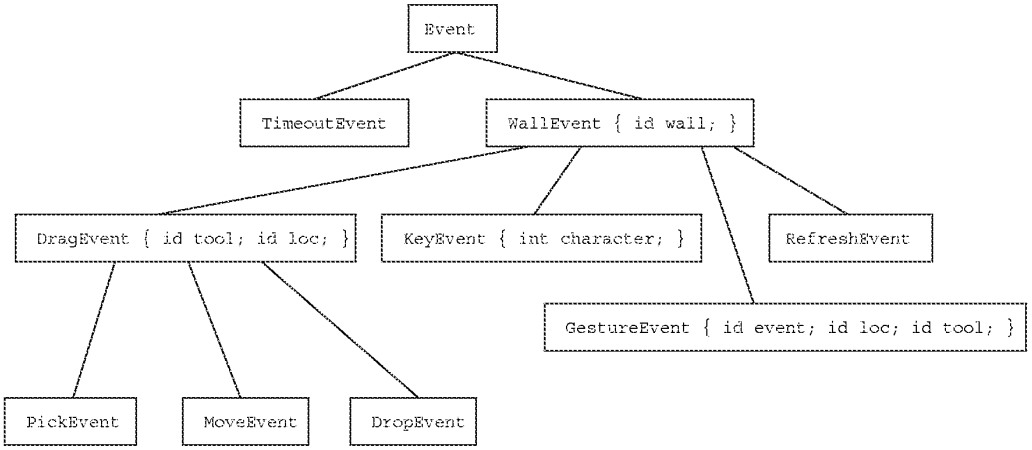


Figure 6.2: The Event Hierarchy

Point object, indicating the location of the mouse cursor.[4] The mouse object referred to by the tool field indicates which button has been pressed. When the mouse is moved (currently only when a mouse button is pressed), a MoveEvent is generated. When the mouse button is released, a DropEvent is generated.

The classes GestureEvent and TimeoutEvent will be discussed in Chapter 7.

6.7.2 Raising an Event

A WallView object represents the root of the view tree of a given window. Associated with each WallView object is a Wall object which actually implements the interface between GRANDMA and the window manager. Also associated with each WallView object (*i.e.* each window) is an EventHandlerList object.

```
= WallView : View { id handlers; id viewdatabase; id wall; }
+ name:(STR)name {
    self = [self createViewOf:nil];
    wall = [Wall create:name wallview:self];
    handlers = [EventHandlerList new];
    viewdatabase = [Xydb new];
    [handlers add:[XyEventHandler wallview:self]];
    return self;
}
- raise:event { return [handlers raise:event]; }
- viewdatabase { return viewdatabase; }
```

[4] In retrospect it probably would have been wiser either to always represent points and rectangles as C structures, or as separate coordinates, instead of using Point and Rectangle objects and their associated overhead.

```
= Wall :  Object (GRANDMA,Geometry)
    { Win win; id pictures; id wallview; }
- raise:event {
    if([event isKindOf:RefreshEvent])
        { [self redraw]; return; }
    return [wallview raise:event];
}
```

Events are raised within a particular window using the `raise` message. `Redraw` events are handled within the wall; since each wall maintains a list of `Picture` objects currently hung on it, redraw is easily accomplished. The `Redraw` special case is really just old code; it would be simple to replace this code with a redraw event handler. All other events are passed from the `Wall` to the `WallView` to the `EventHandlerList`:

```
= EventHandlerList :  OrdCltn { }
- raise:e { int i;
    for(i = [self lastOffset]; i >= 0; i--)
        if( [[self at:i] event:e] )
            break;
    return self;
}
```

An `EventHandlerList` is just an `OrdCltn`, thus `add:` and `remove:` messages can be sent to it to add or remove active event handlers. The `add:` message adds handlers to the end of the list; `raise` iterates through the list backwards, asking each element of the list in order if it wishes to handle the event. Thus, handlers activated most recently are asked about events before those activated earlier. (It is possible to install an active event handler at an arbitrary position in the `EventHandlerList` by using some of `OrdCltn`'s other methods, but this has never been needed in GRANDMA.) Note that the first thing a `WallView` object does when created is activate an `XyEventHandler`; this handler, since it is first in the list, will be tried only after the other handlers have declined to process the event.

6.7.3 Active Event Handlers

Every active event handler must respond to the `event:` message, returning a boolean value indicating whether it has handled the event.

```
= EventHandler :  Object { }
- (BOOL)event:e
    { return (BOOL)[self subclassResponsibility]; }
```

The `event:` method here is a placeholder for the actual method, which would be implemented differently in each subclass of `EventHandler`. The `subclassResponsibility` method is inherited from `Object`. The method simply prints an error message stating that the subclass of the receiver should have implemented the method.

Note that the `event:` message sent to the active event handlers has no reference to any views. When the event handler is first activated, it generally stores the view and tool which caused its

activation[5]; it can then refer to these to decide whether to handle an event. When handling an event, the active event handler typically sends the view messages, if only to find out the model to which the view refers.

As previously mentioned, the last active event handler tried is the `XyEventHandler`. This event handler is rather atypical in that it never exists in a passive state.

```
= XyEventHandler :  EventHandler { id wallview; }
+ wallview:_wallview { self = [self new];
        wallview = _wallview; return self; }
- (BOOL)event:e { id views, seq, v, tool;
    if(!  [e respondsTo:@selector(loc) ]) return NO;
    views = [[wallview viewdatabase] at:[e loc]];
    tool = [e tool];
    for(seq = [views eachElement]; v = [seq next]; )
        if(v != tool && [v event:e]) return YES;
    return NO;
}
```

An `XyEventHandler` is instantiated and activated when a `WallView` is created (see Section 6.7.2). The `WallView` is recorded in the handler so that it can access the current database of views (those views in the `View` subtree of the `WallView`). (In retrospect, it would have been more efficient for the `XyEventHandler` to store a handle to the database directly, rather than always asking the `WallView` for it.)

When an `XyEventHandler` is asked to handle an event (via the `event:` message) it first checks to see if that event responds to the message `loc`. Currently, only (subclasses of) `DragEvents` respond to `loc`, but that could conceivably change in the future so the handler is written as generally as possible. This points to one of the major benefits of Objective C; one can inquire as to whether an object responds to a message before attempting to send it the message. Another example of this will be seen in Section 6.7.7. Since the `XyEventHandler` is going to look up views at the location of an event, it obviously cannot deal with events without locations, so returns `NO` (the Objective C term for `FALSE` or `0`) in this case.

The view database is then consulted, returning all the views whose bounding box contains the given point. The views returned are sorted from foremost to most background, *i.e.* according to their depth in the view tree, deepest first. In this order, each view is queried as to whether it wishes to handle the event, stopping when a view says `YES`. (The enigmatic test `v != tool` will be explained in section 6.7.7; suffice it to say here that in the typical case, `tool` is a kind of `GenericMouseTool` and thus can never be equal to a `View`.)

If no view is found that wishes to handle the event, the `XyEventHandler` returns `NO`. Since this handler is the last active event handler to be tried, when it says `NO`, the event is ignored. If desired, it is a simple matter to activate a catchall handler (to be tried after the `XyEventHandler`), the purpose of which is to handle all events, printing a message to the effect that events are being ignored.

[5] As shown in section 6.7.5, passive event handlers are asked to handle events via the `event:view:` message, one parameter being the event (from which the handler gets the tool), and the other is the view.

Another example event handler is given in Section 6.7.6; more will be said about active event handlers then.

6.7.4 The View Database

The function of the view database is to determine the set of views at a given location in a window. In many object-oriented UI toolkits, this function has been combined with event propagation, in that events propagate down the view tree [105] (or a corresponding controller tree [70, 63]) directly. The idea for a separate view database comes from GWUIMS[118]. By separating out the view database into its own data structure, efficient algorithms for looking up views at a given point, such as Bentley's dual range trees [7], may be applied. Unfortunately, this optimization was never completed, and in retrospect having to keep the view database synchronized with the view hierarchy was more effort than it was worth.

```
= Xydb :  Set { }
- enter:object at:rectangle {
    return [self replace:  [Xydbe object:object
                                    at:rectangle
                depth:[object depth]]];
}

depthcmp(o1, o2) id *o1, *o2;
    { return [*o2 depth] - [*o1 depth]; }

- at:aPoint {
        id seq, e, array[MAXAT], result = [OrdCltn new]; int n;
        for(n = 0, seq = [self eachElement];
        (e = [seq next]) != nil; )
                if([e contains:aPoint]) array[n++] = e;
        qsort(array, n, sizeof(id), depthcmp);
        for(i = 0; i < n; i++) [result add:[array[i] object]];
        return result;
}

= Xydbe :  Rectangle { id object; unsigned depth; }
+ object:o at:rect depth:(unsigned)d {
        self = [self new] object = o; depth = d;
        return [[self origin:[rect origin]] corner:[rect corner]];
}
- object { return object; }
- (unsigned)depth { return depth; }
- (unsigned) hash { return [object hash]; }
- (BOOL)isEqual:o { object == o->object; }
```

An Xydb is a set of Xydbe objects ("e" for "element"), each of which is a rectangle, an associated object (always a kind of View in GRANDMA), and a depth. View objects which move or grow must be sure to register their new locations in the view database for the wall on which they lie. This is currently done automatically in the _sync method of class View which is responsible for updating the display when a View changes. The hash and isEqual: methods are used by Set; here they define two Xydbe objects to be equal when their respective object fields are equal.

6.7.5 The Passive Event Handler Search Continues

Each View object has a list of passive handlers associated with it. The association is often implicit: passive handlers can be associated with the view directly, or with the class of the view, or any of the superclasses of the view's class. For example, the GenericToolOnViewEventHandler is directly associated with class View; it thus appears on every view's list of passive event handlers.

```
= View ...
- (BOOL)event:e { id seq, h;
        if(!  [self isOver:[e loc]]) return NO;
        for(seq = [self eachHandler]; h = [seq next];)
                if([h event:e view:self]) return YES;
    return NO;
}
- eachHandler { id r = [OrdCltn new]; id class;
    [r addContentsOf:[self passivehandlers]];
    for(class = [self class];
        class != Object; class = [class superClass])
        [r addContentsOf:[class passivehandlers]];
    return [r eachElement];
}
+ passivehandlers
    { return [UIprop getvalue:self propstr:"handlers"]; }
- passivehandlers
    { return [UIprop getvalue:self propstr:"handlers"]; }
```

When a view is asked if it wishes to handle an event, it firsts asks if the event's location is indeed over the view. The implementation of the isOver: method in class View simply returns YES. Non-rectangular subclasses of view (*e.g.* LineDrawingView, see Section 8.1) override this method.

Assuming the event location is over the view, each passive event handler associated with the view is sent the event:view: message, which asks if the passive handler wishes to handle the event. The search stops as soon as one of the handlers says YES or all the handlers have been tried.

The method eachHandler returns an ordered sequence of handlers associated with a view. The sequence is the concatenation of the handlers directly associated with the view object, those directly associated with the view's class, those associated with the view's superclass, and so on, up to and including those associated with class View. The associations themselves are stored in a

global property list. The passive event handler is associated with a view object or class under the `"handlers"` property.

Herein lies another advantage of Objective C. An object's superclasses may be traversed at runtime, in this case enabling the simulation of inheritance of passive event handlers. This effect would be difficult to achieve had it not been possible to access the class hierarchy at runtime.

6.7.6 Passive Event Handlers

A passive event handler returns `YES` to the `event:view:` message if it wishes to handle the event directed at the given view. As a side effect, the passive event handler may activate (a copy or instance of) itself to handle additional input without incurring the cost of the search for a passive handler again.

In Objective C, classes are themselves first class objects in the system, known as factory objects. A factory object that is a subclass of `EventHandler` may play the role of a passive event handler.[6] To activate such a handler, the factory would instantiate itself and place the new instance on the active event list.

```
= EventHandler ...
+ (BOOL)event:e view:v
    { return (BOOL)[self subclassResponsibility]; }
```

As an example, consider the following handler for the toggle switch discussed earlier:

```
= ToggleSwitchEventHandler : EventHandler { id view, tool; }
+ (BOOL) event:e view:v {
    if( ! [e isKindOf:PickEvent] ) return NO;
    if( ! [[e tool] isKindOf:MouseTool] ) return NO;
    self = [self new]; view = v; tool = [e tool];
    [[view wallview] activate:self];
    [view highlight];
    return YES;
}
- (BOOL)event:e {
    BOOL isOver;
    if( ![e isKindOf:DragEvent] || [e tool] != tool )
        return NO;
    isOver = [view pointInIboxAndOver:[e loc]];
    if(!isOver || [e isKindOf:DropEvent]) {
        [view unhighlight];
        [[view wallview] deactivate:self];
        if(!isOver) [[view model] toggle];
    }
```

[6] However, using factory objects for passive event handlers is restrictive, as there is only one instance of the factory object for a given class. This makes customization of a factory passive event handler difficult. Section 6.7.8 explains how regular (non-factory) objects may be used as passive event handlers.

```
    return YES;
}
```

Assuming this event handler is associated with a `SwitchView`, when the mouse is pressed over such a view the handler's `event:view:` method is called, which instantiates and then activates this handler, and then highlights the view. Other events, such as typing a character or moving a mouse (with the button already pressed) over the view, will be ignored by this passive handler. Most handlers for mouse events, including this one, only respond to tools of kind `MouseTool`, where `MouseTool` is a subclass of `GenericMouseTool`. The reason for this is explained in Section 6.7.7.

Once the handler is activated, it gets first priority at all incoming events. The beginning of the `event:` method insures that it only responds to mouse events generated by the same mouse tool that initially caused the handler to be activated. For valid events, the handler checks if the location of the event (*i.e.* the mouse cursor) is over the view using `View`'s `pointInIboxAndOver:` method. Note that during passive event dispatch, the more efficient `isOver:` method was used, since by that point, the event location was already known to be in the bounding box of the view. The `pointInIboxAndOver` does both the bounding box check and the `isOver:` method, since active event handlers see events before it is determined which views they are over.

If the mouse is no longer over the switch, or the mouse button has been released, the highlighting of the view is turned off, and the handler deactivated. In the case where the mouse is over the view when the button was released, `[[view model] toggle]` is executed. The clause `[view model]` returns the model associated with the switch, presumably of class `Boolean`, which gets sent the `toggle` message. This will of course result in the switch's picture getting changed to reflect the model's new state.

In any case, by returning `YES` the active event handler indicates it has handled the event, so there will be no attempt to propagate it further.

Typically, the `ToggleSwitchEventHandler` would get associated with the `SwitchView` as follows:

```
= SwitchView ...
+ initialize
    { return [self sethandler:ToggleSwitchEventHandler]; }
```

The `initialize` factory method is invoked for every class in the program (which has such a method) by the Objective C runtime system when the program is first started. In this case, the `sethandler` factory method would create a list (`OrdCltn`) containing the single element `ToggleSwitchEventHandler` and associate it with the class `SwitchView` under the `"handlers"` property.

Note that some simple changes to the `ToggleSwitchEventHandler` could radically alter the behavior of the switch. For example, if `[[view model] toggle]` is also executed when the switch is first pressed (*i.e.* in the `event:view:` method), the switch becomes a momentary pushbutton rather than a toggle switch. Similarly, by changing the initial check to `[e isKindOf:DragEvent]`, once the mouse moves off the switch (thus deactivating the handler), moving the mouse back on the switch with the button still pressed (or onto another instance of the switch) would (re)activate the handler. If the handler is changed only to deactivate when a

`DropEvent` is raised, the button now grabs the mouse, meaning no other objects would receive mouse events as long as the button is pressed. It is clear that many different behaviors are possible simply by changing the event handler.

While GRANDMA easily allows much flexibility in programming the behavior of individual widgets, interaction techniques that control multiple widgets in tandem are more difficult to program. For example, radio buttons (in which clicking one of a set of buttons causes it to be turned on and the rest of the set to be turned off) might be implemented by having the individual buttons to be subviews of a new parent view, and a new handler for the parent view could take care of the mutual exclusion. (Alternatively, the parent view could handle the mutual exclusion by providing a method for the individual buttons to call when pressed; in this case the parent necessarily provides the radio button interface to the rest of the program.)

6.7.7 Semantic Feedback

Semantic feedback is a response to a user's input which requires specialized information about the application objects [96]. For example, in the Macintosh Finder [2], dragging a file icon over a folder icon causes the folder icon to highlight, since dropping the file icon in the folder icon will cause the file to be moved to the folder. Dragging a file icon over another file icon causes no such highlighting, since dropping a file on another file has no effect. The highlighting is thus semantic feedback.

GRANDMA has a general mechanism for implementing (views of) objects which react when (views of) other objects are dropped on them, highlighting themselves whenever such objects are dragged over them. Such views are called *buckets* in GRANDMA. Any view may be made into a bucket simply by associating it with a passive `BucketEventHandler` (which expects the view to respond to the `actsUpon:` and `actUpon::` messages discussed below). Once a view has a `BucketEventHandler`, the semantic feedback described above will happen automatically.

Whereas a bucket is a view which causes an action when another view is dropped in it (*e.g.* the Macintosh trash can is a bucket), a `Tool` is an object which causes an action when it is dropped on a view (a "delete cursor" is thus a tool). As mentioned above, a tool corresponds to a physical input device (*e.g.* `GenericMouseTool`), but it is also possible for a view to be a tool. In the latter case, the view is referred to as a *virtual tool*.

Buckets and tools are quite similar, the main difference being that in buckets the action is associated with stationary views, while in tools the action is associated with the view being dragged. The implementation of tools is considered next. The similar implementation of buckets will not be described.

```
= Tool : Object { }
- (SEL)action { return (SEL) 0; }
- actionParameter { return nil; }
- (BOOL)actsUpon:v { return [v respondsTo:[self action]]; }
- actUpon:v event:e {
    [v perform:[self action]
          with:[self actionParameter]
          with:e
          with:self];
```

```
    return self;
}
```

Every tool responds to the `actsUpon:` and `actUpon::` messages. In the default implementation above, a tool has an action (which is the runtime encoding of a message selector) and an action parameter (an arbitrary object). For example, one way to create a tool for deleting objects is

```
= DeleteTool : Tool { }
- (SEL)action { return @selector(delete); }
```

The `actsUpon:` method checks to see if the view passed as a parameter responds to the action of the tool, in this case `delete`. The `actUpon::` method actually performs the action, passing the action parameter, the event, and the tool itself as additional parameters (which are ignored in the `delete` case).

The `GenericToolOnViewEventHandler` is associated with every view via the `View` class:

```
= View ...
+ initialize
    { [self sethandler:GenericToolOnViewEventHandler]; }

= GenericToolOnViewEventHandler : EventHandler
    { id tool, view; }
+ (BOOL) event:e view:v {
    if( ! [e isKindOf:DragEvent]) return NO;
    if( ! [[e tool] actsUpon:v]) return NO;
    self = [self new];
    tool = [e tool]; view = v; [view highlight];
    [[view wallview] activate:self];
    return YES;
}
- (BOOL)event:e {
    if( ! [e isKindOf:DragEvent]) return NO;
    if( [e tool] != tool) return NO;
    if( [view pointInIboxAndOver:[e loc]] ) {
        if([e isKindOf:DropEvent]) {
            [view unhighlight];
            [[view wallview] deactivate:self];
            [tool actUpon:view event:e];
        }
        return YES;
    }
    [view unhighlight]; [[view wallview] deactivate:self];
    return NO;
}
```

Passively, GenericToolOnViewEventHandler operates by simply checking if the tool over the view acts upon the view. If so, the view is highlighted (the semantic feedback) and the handler activates an instantiation of itself. Subsequent events will be checked by the activated handler to see if they are made by the same tool. If so, and if the tool is still over the view, the event is handled, and if it is a DropEvent then the tool will act upon the view. If the tool has moved off the view, the highlighting is turned off, and the handler deactivates itself and returns NO so that other handlers may handle this event.

The test v != tool in the XYEventHandler (see Section 6.7.3) prevents a view that is a virtual tool from ever attempting to operate upon itself.

6.7.8 Generic Event Handlers

If you have been following the story so far, you know that all the event handlers shown have the passive handler implemented by a factory (class) object which responds to event:view: messages. When necessary, such a passive handler activates an instantiation of itself. The drawback of having factory objects as passive event handlers is that they cannot be changed at runtime. For example, the ToggleSwitchEventHandler only passively responds to PickEvents. If one wanted to make a ToggleSwitchEventHandler that passively responded to any DragEvent, one could either change the implementation of ToggleSwitchEventHandler (thus affecting the behavior of every toggle switch view), or one could subclass ToggleSwitchEventHandler. Doing the latter, it would be necessary to duplicate much of the event:view: method, or change ToggleSwitchEventHandler by putting the event:view: method in another method, so that it can be used by subclasses. In any case, changing a simple item (the kind of event a handler passively responds to) is more difficult than it need be.

In order to make event handlers more parameterizable, the passive event handlers should be regular objects (*i.e.* not factory objects). In response to this problem, most event handlers are subclasses of GenericEventHandler.

```
= GenericEventHandler : EventHandler {
    BOOL shouldActivate;
    id startp, handlep, stopp;
    id view, wall, tool, env;
}
+ passive { return [self new]; }

- shouldActivate { shouldActivate = YES; return self; }

- startp:_startp { startp = _startp; return self; }
- startp { return startp; }
- (BOOL)evalstart:env { return [[startp eval:env] asBOOL]; }

- stopp:_stopp { stopp = _stopp; return self; }
- stopp { return stopp; }
- (BOOL)evalstop:env { return [[stopp eval:env] asBOOL]; }
```

```
- handlep:_handlep { handlep = _handlep; return self; }
- handlep { return handlep; }
- (BOOL)evalhandle:env
    { return [[handlep eval:env] asBOOL]; }

- (BOOL) event:e view:v {
    env = [[[Env new] str:"event" value:e]
              str:"view" value:v];
    if([self evalstart:env])
        { [self startOnView:v]; return YES; }
    return NO;
}
- startOnView:v event:e {
    if(shouldActivate)
        self = [self copy], [[view wallview] activate:self];
    view = v; wall = [view wallview]; tool = [e tool];
    [self passiveHandler:e];
    return self;
}
- (BOOL)event:e {
    if(tool != nil  &&  [e tool] != tool) return NO;
    env = [[[Env new] str:"event" value:e]
              str:"view" value:view];
    if([self evalstop:env])
        [self activeTerminator:e], [wall deactivate:self];
    else if([self evalhandle:env])
        [self activeHandler:e];
    else return NO;
    return YES;
}
- passiveHandler:e { return self; }
- activeHandler:e { return self; }
- activeTerminator:e { return self; }
```

A new passive handler is created by sending a kind of `GenericEventHandler` the `passive` message. A generic event handler object has settable predicates `startp`, `handlep`, and `stopp`. These predicates are expression objects, essentially runtime representations of almost arbitrary Objective C expressions. (The Objective C interpreter built into GRANDMA is discussed in section 7.7.3.) By convention, these predicates are evaluated in an environment where `event` is bound to the event under consideration and `view` is bound to a view at the location of the event. Of course, the result of evaluating a predicate is a boolean value.

The `passive` method is typically overridden by subclasses of `GenericEventHandler` in order to provide default values for `startp`, `handlep`, and `stopp`. The predicate `startp` controls what events the passive handler reacts to. The class `EventExpr` allows easy specification of simple predicates, *e.g.* the call

```
[self startp:[[[EventExpr new] eventkind:PickEvent]
                               toolkind:MouseTool]];
```

sets the start predicate to check that the event is a kind of `PickEvent` and that the tool is a `MouseTool`. This results in the same passive event check that was hard-coded into the factory `ToggleSwitchEventHandler`, but now such a check may be easily modified at runtime.

The message `shouldActivate` tells the passive event handler to activate itself whenever its `startp` predicate is satisfied. Note that it is a clone of the handler that is activated, due to the statement `self = [self copy];` it is thus possible for a single passive event handler to activate multiple instances of itself simultaneously. The active handler responds to any message which satisfies its `handlep` or `donep` predicates. In the latter case, the active event handler is deactivated.

When the `startp`, `handlep`, or `donep` predicates are satisfied, the generic event handler sends itself the `passiveHandler:`, `activeHandler:` or `activeTerminator:` message, respectively. The main work of subclasses of `GenericEventHandler` are done in these methods.

The `startOnView:event:` allows a passive handler to be activated externally (*i.e.* instead of the typical way of having its `startp` satisfied in the `event:view:` method). In this case, the `event` parameter is usually `nil`. For example, an application that wishes to force the user to type some text into a dialogue box before proceeding might activate a text handler in this manner.

The purpose of generic event handlers in GRANDMA is similar to that of *interactors* in Garnet [95, 91] and *pluggable views* in Smalltalk-80 [70]. Since GRANDMA comes with a number of generally useful generic event handlers, application programmers often need not write their own. Instead, they may customize one of the generic handlers by setting up the parameters to suit their purposes. The only parameters every generic event handler has in common are the predicates, and indeed these are the ones most often modified. GRANDMA has a subsystem which allows these parameters to be modified at runtime by the user.[7]

6.7.9 The Drag Handler

As an example of a generic event handler, consider the `DragHandler`. When associated with a view, the `DragHandler` allows the view to be moved (dragged) with the mouse. If desired, moving the view will result in new events being raised. This allows the view to be used as tool, as discussed in section 6.7.7. Also parameterizable are whether the view is moved using absolute or relative coordinates, whether the view is copied and then the copy is moved, and the messages that are sent to actually move the view. Reasonable defaults are supplied for all parameters.

```
= DragHandler : GenericEventHandler {
    BOOL    copyview, genevents, relative;
    SEL     whenmoved, whendone;
```

[7] Typically, it would be the interface designer, rather than the end user, who would use this facility.

```
    BOOL    deactivate;
    int     savedx, savedy;
}

+ passive {
   self = [super passive];
   [self shouldActivate];
   [self startp:[[[EventExpr new] eventkind:DragEvent]
                                  toolkind:MouseTool]];
   [self handlep:[[EventExpr new] eventkind:DragEvent]];
    [self stopp:[[EventExpr new] eventkind:DropEvent]];
    copyview = NO; genevents = YES; relative = NO;
    whenmoved = @selector(at::);  whendone = (SEL) 0;
    return self;
}

/* changing default parameters: */

/* copyviewON causes the view to be copied and then the copy to be dragged */
- copyviewON { copyview = YES; return self; }

/* genEventsOFF makes the handler not raise any events */
- genEventsOFF { genevents = NO; return self; }

/* relativeON makes the handler send the move:: message, passing
  relative coordinates (deltas from the current position) */
- relativeON { relative = YES;
           whenmoved = @selector(move::); }

/* whendone: sets the message sent on the event that terminates the drag */
- whendone:(SEL)sel { whendone = sel; return self; }

/* whenmoved: sets the message sent for every point in the drag */
- whenmoved:(SEL)sel { whenmoved = sel; return self; }

- passiveHandler:e {
    id l = [e loc];
    if(relative) savedx = [l x], savedy = [l y];
    else savedx = [view xloc]-[l x],
         savedy = [view yloc]-[l y];
    if(copyview) view = [view viewcopy];
    [view flash];
```

```
    return self;
}

- activeHandler:e {
    int x, y;
    if(relative) {
        x = [[e loc] x], y = [[e loc] y];
        [view perform:whenmoved
              with:(x - savedx) with:(y - savedy)];
        savedx = x, savedy = y;
    }
    else {
        x = [[e loc] x] + savedx, y = [[e loc] y] + savedy;
        [view perform:whenmoved with:x with:y];
    }
    if(genevents)
         [wall raise:[[e class] tool:view loc:newloc
            wall:wall instigator:self time:[e time]]];
    return self;
}
- activeTerminator:e {
    if(genevents)
         [wall raise:[[e class] tool:view loc:[e loc]
            wall:wall instigator:self time:[e time]]];
    if(whendone) [view perform:whendone];
    return self;
}
```

The `passive` factory method creates a `DragHandler` with instance variables set to the default parameters. Those parameters can be changed with the `startp:`, `handlep`, `stopp:`, `copyviewON`, `genEventsOFF`, `relativeON`, `whendone:`, and `whenmoved:` messages. (Please refer to the comments in the above code for a description of the function of these parameters.)

For example, a `DragHandler` might be associated with class `LabelView` as follows:

```
= LabelView ...
+ initialize {
    [self sethandler:
      [[[DragHandler passive]
        startp:[[[EventExpr new]
              eventkind:PickEvent] toolkind:MouseTool]]
       genEventsOFF]];
}
```

Any `LabelView` can thus be dragged around with the mouse by clicking directly on it (since the start predicate was changed to `PickEvent`). A `LabelView` will not generate events as it is

dragged since `genEventsOFF` was sent to the handler; thus `LabelViews` in general would not be used as tools or items that can be deposited in buckets. Of course, subclasses and instances of `LabelView` may have their own passive event handlers to override this behavior.

When a passive `DragHandler` gets an event that satisfies its start predicate, the `passiveHandler:` method is invoked. For a `DragHandler`, some location information is saved, the view is copied if need be, and the view is flashed (rapidly highlighted and unhighlighted) as user feedback.

Any subsequent event that satisfies the stop predicate will cause the `activeTerminator:` method to be invoked. Other events that satisfy the handle predicate will cause `activeHandler:` to be invoked. In `DragHandler`, `activeHandler:` first moves the view (typically by sending it the `at::` message with the new coordinates as arguments) then possibly raises a new event with the view playing the role of tool in the event. If the view is indeed a tool, raising this event might result in the `GenericToolOnView` handler being activated, as previously discussed.

Note that the event to be raised is created by first determining the class object (factory) of the passed event (given the default predicates, in this case the class will either be `MoveEvent` or `DropEvent`), and then asking the class to create a new event, which will thus be the same class as the passed event. Most of the new event attributes are copied verbatim from the old attributes; only the `tool` and `instigator` are changed. A more sophisticated `DragHandler` might also change the event location to be at some designated hot spot of the view being moved, rather than simply use the location of the passed event. For simplicity, this was not shown here.

The `activeTerminator:` method also possibly raises a new event, and possibly sends the view the message stored in the `whendone` variable. As an example, `whendone` might be set to `@selector(delete)` when `copyview` is set. When the mouse button is pressed over a view, a copy of the view is created. Moving the mouse drags the copy, and when the mouse button is finally released, the copy is deleted.

Creating a new drag handler and associating it with a view or view class is all that is required to make that view "draggable" (since every view inherits the `at::` message). As shown in the next chapter, GRANDMA has a facility for creating handlers and making the association at runtime.

6.8 Summary of GRANDMA

This concludes the detailed discussion of GRANDMA. As the discussion has concentrated on the features which distinguish GRANDMA from other MVC-like systems, much of the system has not been discussed. It should be mentioned that the facilities described are sufficiently powerful to build a number of useful view and controller classes. In particular, standard items such as popup views, menus, sliders, buttons, switches, text fields, and list views have all been implemented. Chapter 8 shows how some of these are used in applications.

GRANDMA's innovations come from its input model. Here is a summary of the main points of the input architecture:

1. Input events are full-blown objects. The `Event` hierarchy imposes structure on events without imposing device dependencies.

2. Raised events are propagated down an active event list.

3. Otherwise unhandled events with screen locations are automatically routed to views at those locations.

4. A view object may have any number of passive event handlers associated with itself, its class, or its superclass, *etc.* Events are automatically routed to the appropriate handler.

5. A passive event handler may be shared by many views, and can activate a copy of itself to deal with events aimed at any particular view.

6. Event handlers have predicates that describe the events to which they respond.

7. The generic event handler simplifies the creation of dynamically parameterizable event handlers.

Because of the input architecture, GRANDMA has a number of novel features. They are listed here, and compared to other systems when appropriate.

GRANDMA can support many different input devices simultaneously. Due to item 1 above, GRANDMA can support many different input devices in addition to just a single keyboard and mouse. Each device needs to integrate the set of event classes which it raises into GRANDMA's `Event` hierarchy. Much flexibility is possible; for example, a Sensor Frame device might raise a single `SensorFrameEvent` describing the current set of fingers in the plane of the frame, or separate `DragEvents` for each finger, the tool in this case being a `SensorFrameFingerTool`. Because of item 6, it is possible to write event handlers for any new device which comes along.

By contrast, most of the existing user interface toolkits have hard-wired limitations in the kinds of devices they support. For example, most systems (the NeXT AppKit [102], the Macintosh Toolbox [1], the X library [41]) have a fixed structure which describes input events, and cannot be easily altered. Some systems go so far as to advocate building device dependencies into the views themselves; for example, Hypertalk event handlers [45] are labeled with event descriptors such as `mouseUp` and Cox's system [28] has views that respond to messages like `rightButtonDown`. Similarly, systems with a single controller per view [70] cannot deal with input events from different devices. On the other hand, GWUIMS [118] seems to have a general object classification scheme for describing input events.

GRANDMA supports the emulation of one device with another. In GRANDMA, to get the most out of each device it is necessary to have event handlers which can respond to events from that device associated with every view that needs them. If those event handlers are not available, it is still possible to write an event handler that emulates one device by another. For example, an active handler might catch all `SensorFrameEvents` and raise `DragEvents` whose tool is a `Mousetool` in response. The rest of the program cannot tell that it is not getting real mouse data; it responds as if it is getting actual mouse input.

GRANDMA can handle multiple input threads simultaneously. Because passive handlers activate copies of themselves, even views that refer to the same handler can get input simultaneously. The input events are simply propagated down the active event handler list, and each active handler only handles the events it expects. In GRANDMA, a system that had two mice [19] would simply have two `MouseTool` objects, which could easily interleave events. Normally, a passive handler would only activate itself to receive input from a single tool (mouse, in this case), allowing input from the two mice to be handled independently (even when directed at the same view). It would also be possible to write an event handler that explicitly dealt with events from both mice, if that was desired.

Event-based systems, such as Sassafras [54] and Squeak [23], are also able to deal with multi-threaded dialogues. Indeed, it is GRANDMA's similarity to those systems which gives it a similar power. This is in contrast to systems such as Smalltalk [70] where, once a controller is activated it loops polling for events, and thus does not allow other controllers to receive events until it is deactivated.

GRANDMA provides virtual tools. Given the general structure of input events, there is no requirement for them only to be generated by the window manager. Event handlers can themselves raise other events. Many events have `tools` associated with them; for example, mouse events are associated with `MouseTools`. The tools may themselves be views or other objects. By responding to messages such as `action`, a tool makes known its effect on objects which it is dragged over. The `GenericToolOnView` handler, which is associated with the `View` class (and thus every view in the system) will handle the interaction when a tool which has a certain action is dragged over an object which accepts that action. The tools are virtual, in the sense that they do not correspond directly to any input hardware, and they may send arbitrary messages to views with which they interact.

GRANDMA supports semantic feedback. Handlers like `GenericToolOnView` can test at runtime if an arbitrary tool is able to operate upon an arbitrary view which it is dragged over, and if so highlight the view and/or tool. No special code is required in either the tool or the view to make this work. A tool and the views upon which it operates often make no reference to each other. The sole connection between the two is that one is able to send a message that the other is able to receive.

Of course, the default behavior may be easily overridden. A tool can make arbitrary enquiries into the view and its model in order to decide if it does indeed wish to operate upon the view.

Event handling in GRANDMA is both general and efficient. The generality comes from the event dispatch, where, if no other active handler handles an event, the `XYEventHandler` can query the views at the location of the event. The views consult their own list of passive event handlers, which potentially may handle many different kinds of events. There is space efficiency in that a single passive event handler may be shared by many views, eliminating the overhead of a controller object per view. There is time efficiency, in that once a passive handler handles an event, it may activate itself, after which it receives events immediately, without going through the elaborate dispatch of the `XYEventHandler`.

Artkit [52] has a priority list of dispatch agents that is similar to GRANDMA's active event handler list. Such agents receive low-level events (*e.g.* from the window manager), and attempt to translate them into higher level events to be received by interactor objects (which seem to be views). Interactor agents register the high-level events in which they are interested.

Artkit's architecture is so similar to GRANDMA's that it is difficult to precisely characterize the difference. The high-level events in Artkit play a role similar to both that of messages that a view may receive and events that a view's passive event handlers expect. In GRANDMA, the registering is implicit; because of the Objective-C runtime implementation, the messages understood by a given object need not be specified explicitly or limited to a small set. Instead, one object may ask another if it recognizes a given message before sending it.

Because of the translation from low-level to high-level events, it does not seem that Artkit can, for example, emulate one device with another. In particular, it does not seem possible to translate low-level events from one device into those of another. GRANDMA does not make a distinction between low-level and high-level events. Instead, GRANDMA distinguishes between events and messages; events are propagated down the active event handler list; when accepted by an event handler, the handler may raise new events and/or send messages to views or their models.

GRANDMA supports gestures. GRANDMA's general input mechanism had the major design goal of being able to support gestural input. As will be seen in the next chapter, the gestures are recognized by `GestureEventHandlers`; these collect mouse (or other) events, determine a set of gestures which they recognize depending on the views at the initial point of the gesture, and once recognized, can translate the gesture into messages to models or views, or into new events.

Artkit also handles gestural input, and, somewhat like GRANDMA, has gesture event handlers which capture low-level events and produce high-level events. The designers claim that Artkit, because of its object-oriented structure, can use a number of different gesture recognition algorithms, and thus tailor the recognizer to the application, or even bits of the application. The same is true for GRANDMA, of course, though the intention was that the algorithms described in the first half of the thesis are of sufficient generality and accuracy that other recognition algorithms are not typically required. Artkit's claim that *many* recognizers can be used seems like an excuse not to provide *any*. One of the driving forces behind the present work is the belief that gesture recognizers are sufficiently difficult to build that requiring application programmers to hand code such recognizers for each gesture set is a major reason that hardly any applications use gestures. Thus, it is necessary to provide a general, trainable recognizer in order for gesture-based interfaces to be explored. How such a recognizer is integrated into an object-oriented toolkit is the subject of the next chapter.

Of course, GRANDMA does have its disadvantages. Like other MVC systems, GRANDMA provides a multitude of classes, and the programmer needs to be familiar with most of them before he can decide how to best implement his particular task. The elaborate input architecture exacerbates the problem: a large number of possible combinations of views, event handlers, and tools must be

considered by the programmer of a new interaction technique. Also, GRANDMA does nothing toward solving a common problem faced when using any MVC system: deciding what functionality goes into a view and what goes into a model. Another problem is that even though the protocol between event handlers and views is meant to be very general (the event handlers are initialized with arbitrary message selectors to use when communicating with the view), in practice the views are written with the intention that they will communicate with particular event handlers, so that it is not really right to claim that specifics of input have truly been factored out of views.

Chapter 7

Gesture Recognizers in GRANDMA

This chapter discusses how gesture recognition may be incorporated into systems for building direct manipulation interfaces. In particular, the design and implementation of gesture handlers in GRANDMA is shown. Even though the emphasis is on the GRANDMA system, the methods are intended to be generally applicable to any object-oriented user interface construction tool.

7.1 A Note on Terms

Before beginning the discussion, some explanation is needed to help avoid confusion between terms. As discussed in Section 6.4, it is important not to confuse the view hierarchy, which is the tree determined by the subview relationship, and the view class hierarchy, which is the tree determined by the subclass relationship. In GRANDMA, the view hierarchy has a `WallView` object (corresponding to an X window) at its root, while the view class hierarchy has the class `View` at its root.

Another potentially ambiguous term is "class." Usually, the term is used in the object-oriented sense, and refers to the type (loosely speaking) of the object. However, the term "gesture class" refers to the result of the gesture recognition process. In other words, a gesture recognizer (also known as a gesture classifier) discriminates between gesture classes. For example, consider a handwriting recognizer able to discriminate between the written digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. In this example, each digit represents a class; presumably, the recognizer was trained using a number of examples of each class.

To make matters more confusing, in GRANDMA there is a class (in the object-oriented sense) named `Gesture`; an object of this class represents a particular gesture instance, *i.e.* the list of points which make up a single gesture. There is also a class named `GestureClass`; objects of this class refer to individual gesture classes; for example, a digit recognizer would reference 10 different `GestureClass` objects.

Sometimes the term "gesture" is used to refer to an entire gesture class; other times it refers to a single instance of gesture. For example, when it is said a recognizer discriminates between a set of gestures, what is meant is that the recognizer discriminates between a set of gesture classes. Conversely, "the user enters a gesture" refers to a particular instance. In all cases which follow, the

intent should be obvious from the context.

7.2 Gestures in MVC systems

As discussed in Chapters 2 and 6, object-oriented user interface systems typically consist of models (application objects), views (responsible for displaying the state of models on the screen), and controllers (responsible for responding to input by sending messages to views and models). Typical Model/View/Controller systems, such as that in Smalltalk[70], have a view object and controller object for each model object to displayed on the screen.

This section describes how gestures are integrated into GRANDMA, providing an example of how gestures might be integrated into other MVC-based systems.

7.2.1 Gestures and the View Class Hierarchy

Central to all the variations of object-oriented user interface tools is the `View` class. In all such systems, view objects handle the display of models. Since the notion of views is central to all object-oriented user interface tools, views provide a focal point for adding gestures to such tools.

Simply stated, the idea for integrating gestures into direct manipulation interfaces is this: *each view responds to a particular set of gestures.* Intuitively, it seems obvious that, for example, a switch should be controlled by a different set of gestures than a dial. The ability to simply and easily specify a set of gestures and their associated semantics, and to easily associate the set of gestures with particular views, was the primary design goal in adding gestures to GRANDMA.

Of course, it is unlikely that every view will respond to a distinct set of gestures. In general, the user will expect similar views to respond to similar sets of gestures. Fortunately, object-oriented user interfaces already have the concept of similarity built into the view class hierarchy. In particular, it usually makes the most sense for all view objects of the same class to respond to the same set of gestures. Similarly, it is intuitively appealing for a view subclass to respond to all the gestures of its parent class, while possibly responding to some new gestures specific to the subclass.

The above intuitions essentially apply the notions of class identity and inheritance[121] (in the object-oriented sense) to gestures. It is seen that gestures are analogous to messages. All objects of a given class respond to the same set of messages, just as they respond to the same set of gestures. An object in a subclass inherits methods from its superclass; similarly such an object should respond to all gestures to which its superclass responds. Continuing the analogy, a subclass may override existing methods or add new methods not understood by its superclass; similarly, a subclass may override (the interpretation of) existing gestures, or recognize additional gestures. Some object-oriented languages allow a subclass to disable certain messages understood by its superclass (though it is not common), and analogously, it is possible that a subclass may wish to disable a gesture class recognized by its superclass.

Given the close parallel between gesture classes and messages, one possible way to implement gesture semantics would be for each kind of view to implement a method for each gesture class it expects. Classifying an input gesture would result in its class's particular message to be sent to the view, which implements it as it sees fit. A subclass inherits the methods of its superclass, and may

override some of these methods. Thus, in this scheme a subclass understands all the gestures that its superclass understands, but may change the interpretation of some of these gestures.

This close association of gestures and messages was not done in GRANDMA since it was felt to be too constricting. Since in Objective C all methods have to be specified at compile time, adding new gesture classes would require program recompilations. Since it is quite easy to add new gesture classes at runtime, it would be unfortunate if such additions required recompilations. One of the goals of GRANDMA is to permit the rapid exploration of different gestures sets and their semantics; forcing recompilations would make the whole system much more tedious to use for experimentation.

Instead, the solution adopted was to have a small interpreter built into GRANDMA. A piece of interpreted code is associated with each gesture class; this code is executed when the gesture is recognized. Since the code is interpreted, it is straightforward to add new code at the time a new class is specified, as well as to modify existing code, all at runtime. While at first glance building an interpreter into GRANDMA seems quite difficult and expensive, Objective C makes the task simple, as explained in Section 7.7.3.

7.2.2 Gestures and the View Tree

Consider a number of views being displayed in a window. In GRANDMA, as in many other systems, pressing a mouse button while pointing at a particular view (usually) directs input at that view. In other words, the view that gets input is usually determined at the time of the initial button press. Due to the view tree, views may overlap on the screen, and thus the initial mouse location may point at a number of views simultaneously. Typically the views are queried in order, from foremost to background, to determine which one gets to handle the input.

A similar approach may be taken for gestures. The first point of the gesture determines the views at which the gesture might be directed. However, determining which of the overlapping views is the target of the gesture is usually impossible when just the first point has been seen. What is usually desirable is that the entire gesture be collected before the determination is made.

Consider a simplification of GDP. The wall view, behind all other views, has a set of gestures for creating graphic objects. A straight stroke "-" gesture creates a line, and an "L" gesture creates a rectangle. The graphic object views respond to a different set of gestures; an "X" deletes a graphic object, while a "C" copies a graphic object. When a gesture is made over, say, an existing rectangle, it is not immediately clear whether it is directed at the rectangle itself or at the background. It depends on the gesture: an "X" is directed at the existing rectangle, an "L" at the wall view. Clearly the determination cannot be made when just the first point of the gesture has been seen.

Actually, this is not quite true. It is conceivable that the graphic object views could handle gestures themselves that normally would be directed at the wall view. There is some practical value in this. For example, creating a new graphic object over an existing one might include lining up the vertices of the two objects. However, while it is nice to have the option, in general it seems a bad idea to force each view to explicitly handle any gestures that might be directed at any views it covers.

Chapter 3 addressed the problem of classifying a gesture as one of a given set of gesture classes. It is seen here that this set of gestures is not necessarily the set associated with a single view, but instead is the union of gesture sets recognized by all views under the initial point. There are some

technical difficulties involved in doing this. It would in general be quite inefficient to have to construct a classifier for every possible union of view gestures sets. However, it is necessary that classifiers be constructed for the unions which do occur. The current implementation dynamically constructs a classifier for a given set of gesture classes the first time the set appears; this classifier is then cached for future use.

It is possible that more than one view under the initial point responds to a given gesture class. In these cases, preference is given to the topmost view. The result is a kind of dynamic scoping. Similarly, the way a subclass can override a gesture class recognized by its superclass may be considered a kind of static scoping.

7.3 The GRANDMA Gesture Subsystem

In GRANDMA, gestural input is handled by objects of class `GestureEventHandler`. Class `GestureEventHandler`, a subclass of `GenericEventHandler`, is easily the most complex event handler in the GRANDMA system. In addition to the five hundred lines of code which directly implement its various methods, `GestureEventHandler` is the sole user of many other GRANDMA subsystems. These include the gesture classification subsystem, the interface which allows the user to modify gesture handlers (by, for example, adding new gesture classes) at runtime, the Objective C interpreter used for gesture semantics and its user interface, as well as some classes (*e.g.* `GestureEvent`, `TimeoutEvent`) used solely by the gesture handler.

Before getting into details, an overview of GRANDMA's various gesture-related components is presented. Figure 7.1 shows the relations between objects and classes associated with gestures in GRANDMA. The main focus is the `GestureEventHandler`. Like all event handlers, when activated it has a `view` object, which itself has a model and a wall view.[1] A `GestureEventHandler` uses the wall view to activate itself, raise `GestureEvents`, set up timeouts and their handlers, and draw the gesture as it is being made.

Associated with a gesture event handler is a set of `SemClass` objects. A `SemClass` object groups together a gesture class object (class `GestureClass`) with three expressions (subclasses of `Expr`). The `GestureClass` objects represent the particular gesture classes recognized directly by this event handler. The three expressions comprise the semantics associated with the gesture class by this event handler. The first expression is evaluated when the gesture is recognized, the second on each subsequent input event handled by the gesture handler after recognition (the manipulation phase, see Section 1.1), and the third when the manipulation phase ends.

Associated with each `GestureClass` object is a set of `Gesture` objects. These are the examples of gestures in the class and are used in the training of classifiers that recognize the class. A `GestureClass` object contains aggregate information about its examples, such as the estimated mean vector and covariance matrix of the examples' features, both of which are used in the construction of classifiers.

When a `GestureEventHandler` determines which gesture classes it must discriminate among (according to the rules described in the previous section), it asks the `Classifier` class

[1] Recall that a wall view is the root of the view tree and represents a window on the screen.

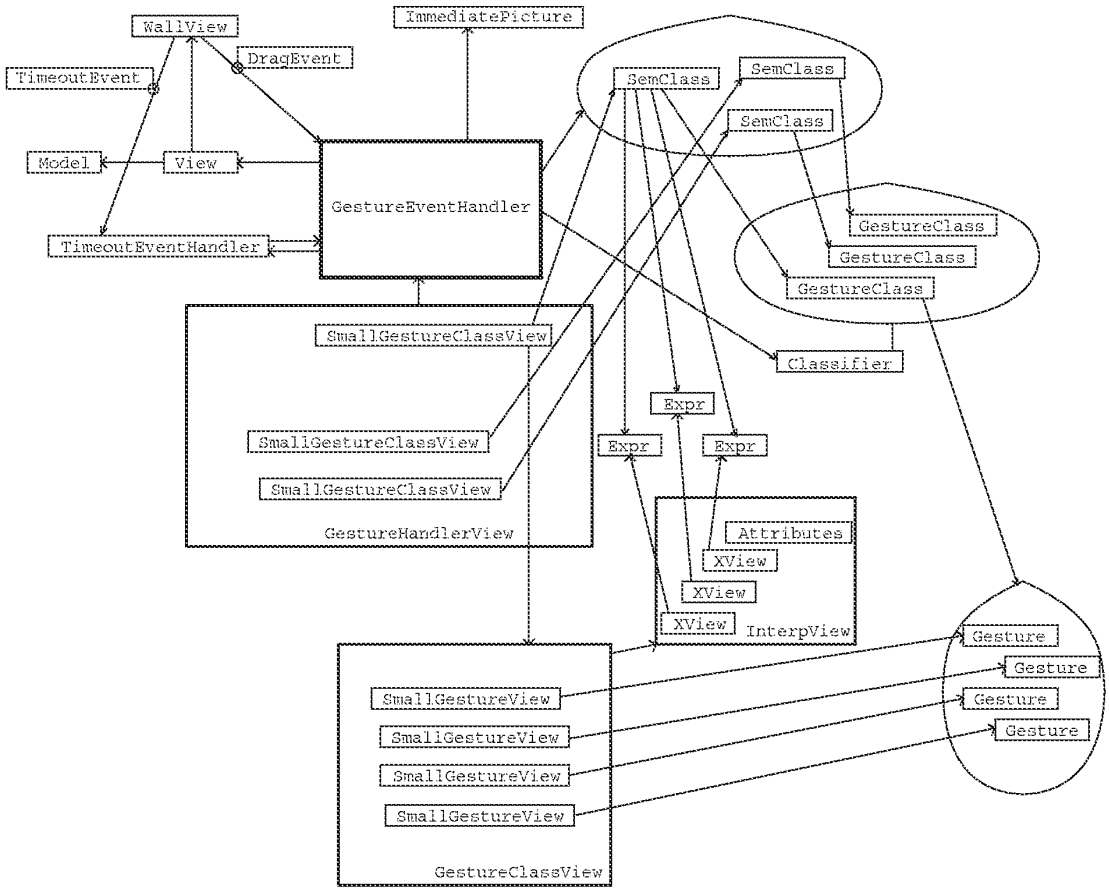


Figure 7.1: GRANDMA's gesture subsystem

A passive `GestureEventHandler` *is associated with a view or view class that expects gestural input. Once gestural input begins, the handler is activated and refers directly to the view at which the gesture was directed, as shown in the figure. The* `ImmediatePicture` *object is used for the inking of the gesture. The handler uses a timeout mechanism to indicate when to change from the collection to manipulation state. A* `SemClass` *object exists for each gesture expected by the handler, with each* `SemClass` *object associating a gesture class with its semantics. Each* `GestureClass` *object is described by a set of example* `Gestures`*, and there are view objects for each of the examples (*`SmallGestureView`*) as well as for the class as a whole (*`GestureClassView`*,* `SmallGestureClassView`*) which allow these to be displayed and edited. The gesture semantics are represented by* `Expr` *objects, and may be edited in the* `InterpView` *window.*

for a classifier object capable of doing this discrimination. Normally such a classifier will already exist; in this case, the existing classifier is simply returned. It is possible that one of the gesture classes in the set has changed; in this case the existing classifier has to be retrained (*i.e.* recalculated). Occasionally, this set of gesture classes has never been seen before; in this case a new classifier is created for this set, returned, and cached for future use.

The components related to the gesture event handler through `GestureHandlerView` are all concerned with enabling the user to see and alter various facets of the event handler. The predicates for starting, handling, and stopping the collection of gesture input may be altered by the user. In addition, gesture classes may be created, deleted, or copied from other gesture event handlers. The examples of a given class may be examined, and individual examples may be added or deleted. Finally, the semantics associated with a given gesture class may be altered through the interface to the Objective C interpreter.

7.4 Gesture Event Handlers

The details of the class `GestureEventHandler` are now described, beginning with its instance variables.

```
static BOOL masterSwitch = YES;
= GestureEventHandler : GenericEventHandler {
    STR     name;
    id      gesture;
    id      picture;
    id      classes;
    id      env;
    int     timeval;
    id      timeouteh;
    short   lastx, lasty;
    id      sclass;
    struct gassoc { id sclass, view; } *gassoc;
    int     ngassocs;
    id      class_set;
    BOOL    manip_phase;
    BOOL    classify;
    BOOL    ignoring;
    id      mousetool;
}
```

The `masterSwitch`, settable via the `masterSwitch:` factory method, enables and disables all gesture handlers in an application. This provides a simple method for an application to provide two interfaces, one gesture-based, the other not. Every gesture handler will ignore all events when `masterSwitch` is NO. It will be as if the application had no gesture event handlers. Typically, the remaining event handlers would provide a more traditional click and drag interface to the application.

A particular handler can be turned off by setting its `ignoring` instance variable via the `ignore:` message. GRANDMA can thus be used to compare, say, two completely different gestural interfaces to a given application, switching between them at runtime by turning the appropriate handlers on and off.

The instance variable `name` is the name of the gesture handler. A handler is named so that it can be saved, along with its gesture classes, their semantics and examples, in a file. This is obviously necessary to avoid having the user enter examples of each gesture class each time an application is started. The name is passed to the `passive:` method which creates a passive gesture handler:

```
= GestureEventHandler ...
+ passive:(STR)_name {
    FILE *f;

    self = [super passive];
    classes = [OrdCltn new];
    [self instantiateON];
    [self startp:[[[EventExpr new] eventkind:PickEvent]
                                   toolkind:MouseTool]];
    [self handlep:[[EventExpr new] eventkind:DragEvent]];
    [self stopp:[[EventExpr new] eventkind:DropEvent]];
    [self name:_name];
    timeval = DefaultTimeval;
    classify = YES;
    if((f = [self openfile:"r"]) != NULL) [self read:f];
    return self;
}
```

The typical gesture handler activates itself in response to mouse `PickEvents`, handles all subsequent mouse events, and deactivates itself when the mouse button is released. Of course, being a kind of generic event handler, this default behavior can be easily overridden, as was done to the `DragEventHandler` discussed in Section 6.7.9.

By default, the gesture event handler plans to classify any gestures directed at it (`classify = YES`). This is changed in those gesture event handlers that collect gestures for training other gesture event handlers.

The default `timeval` is 200, meaning 200 milliseconds, or two tenths of a second. This is the duration that mouse input must cease (the mouse must remain still) for the end of a gesture to be recognized. The user may change the default, thus affecting every gesture event handler. The timeout interval may also be changed on a per handler basis, a feature useful mainly for comparing the feel of different intervals.

When an event satisfies the handler's start predicate, the handler activates itself, and its `passiveHandler` is called.

```
= GestureEventHandler ...
- passiveHandler:e {
    gesture = [[Gesture new] newevent:e];
```

```
        picture = [ImmediatePicture create];
        [view _hang:picture at:0:0];
        lastx = [[e loc] x]; lasty = [[e loc] y];
        env = [Env new];
        [env str:"gesture" value:gesture];
        [env str:"startEvent" value:[e copy]];
        [env str:"currentEvent" value:[e copy]];
        [env str:"handler" value:self];
        manip_phase = NO;
        timeouteh = [[TimeoutEventHandler active]
                rec:self sel:@selector(timedout:)];
        [wall activate:timeouteh];
        [wall timeout:timeval];

        if(classify) {
            class_set = [Set new];
            gassoc = (struct gassoc *)
                    malloc(MAXCLASSES * sizeof(struct gassoc));
            ngassocs = 0;
            [[wall handlers]
               raise:[GestureEvent instigator:self event:e
                        env:[[Env new] str:"event" value:e]]];
        }
        return self;
    }
```

The passive handler allocates a new `Gesture` object which will be sent the input events as they arrive. The initial event is sent immediately.

The `picture` allows the gesture handler to ink the gesture on the display as it is being made. Class `ImmediatePicture` is used for pictures which are displayed as they are drawn, rather than the normal `HangingPicture` class which requires pictures to be completed before they can be drawn.

The `env` variable holds the environment in which the gesture semantics will be executed. Within this environment, the interpreter variables `gesture`, `startEvent`, `currentEvent`, and `handler` are bound appropriately (see Section 7.7.1).

The boolean `manip_phase` is true if and only if the entire gesture has been collected and the handler is now in the manipulation phase (see Section 1.1).

A `TimeoutEventHandler` is created and activated. When a `TimeoutEvent` is received by the handler, the handler will send an arbitrary message (with the timeout event as a parameter) to an arbitrary object. In the current case, the `timedout:` message is sent to the active `GestureEventHandler`. In retrospect, the general functionality of the `TimeoutEventHandler` is not needed here; the `GestureEventHandler` could itself easily receive and process `TimeoutEvents` directly, without the overhead of a `TimeoutEventHandler`.

The code `[wall timeout:timeval]` causes the wall to raise a `TimeoutEvent` if there has been no input to the wall in `timeval` milliseconds. A `timeval` of zero disables the raising of `TimeoutEvents`. As previously mentioned, a gesture is considered complete even if the mouse button is held down, as long as the mouse has not been moved in `timeval` milliseconds. The `TimeoutEvent` is used to implement this behavior.

If the gesture being collected is intended to be classified, the set of possible gesture classes must be constructed, and a `Set` object is allocated for this purpose. Recall from Section 7.2.2 that there may be multiple views at the location of the start gesture each of which accepts certain gestures. An array of `gassoc` structures is allocated to associate each of the possible gesture classes expected with its corresponding view. A `GestureEvent` is then raised, with the instigator being the current gesture handler, and having the current event as an additional field.

Raising the `GestureEvent` initiates the search for the possible gesture classes given the initial event. Recall from Sections 7.2.1 and 7.2.2 that each view under the initial point is considered from top to bottom, and for each view, the gestures associated directly with the view itself, and with its class and superclasses, are added in order. Note that this is exactly the same search sequence as that used to find passive event handlers for events that no active handler wants (see Section 6.7). The `GestureEvent`, handled by the same passive event handler mechanism, will thus be propagated to other `GestureEventHandlers` in the correct order. Each passive gesture handler that would have handled the initial event sends a message to the gesture handler which raised the `GestureEvent` indicating the set of gesture classes it recognizes and the view with which it is associated.

Note that only views under the first point of the gesture are queried. The case where a gesture is more naturally expressed by not beginning on the view at which it is targeted is not handled by GRANDMA. For example, it would be desirable for a knob turning gesture to go around the knob, rather than directly over it. In GRANMDA either the knob view area would have to be larger than the actual knob graphic to insure that the starting point of the gesture is over the knob view, or a background view that includes the knob as a subview must handle the knob-turning gesture. In the latter case, the gesture semantics are complicated because the background view needs to explicitly determine at which knob, if any, the gesture is directed. Henry et. al. [52] also notes the problem, and suggests that one gesture handler might hand off a gesture in progress to another handler if it determines that the initial point of the gesture was misleading, but exactly how such a determination would be made is unclear.

```
= GestureEventHandler ...
- (BOOL)event:e view:v {
    if((classify && masterSwitch==NO) || ignoring==YES)
        return NO;
    if( [e isKindOf:GestureEvent] ) {
        if(classify
        && [self evalstart:[[e env] str:"view" value:v] )
                [ [e instigator] classes:classes view:v ];
        return NO;
    }
```

```
    return [super event:e view:v];
}
```

The `GestureEventHandler` overrides `GenericEventHandler`'s `event:view:` method to check directly for `GestureEvents`. (A check for `GestureEvents` could have been included in the default start predicate, but this would require programs which modify the start predicate to always include such a check, an unnecessary complication.) First the state of the `masterSwitch` and `ignoring` switches is checked, so that this handler will not operate if explicitly turned off. (The reason `classify` is checked is to allow gesture handlers which do not classify gestures, *i.e.* those used to collect gesture examples for training purposes, to operate even though gestures are disabled throughout the system.)

When a `GestureEvent` is seen, the handler checks that it indeed classifies gestures and that it would itself have handled the start event (see Section 6.7.8). The environment used for evaluating the start predicate is constructed so that `"event"` and `"view"` are bound to what they would have been had the handler actually been asked to handle the initial event. If the handler would have handled the event, the set of gesture classes associated with the handler, as well as the view, are passed to the handler which instigated the `GestureEvent`.

Note that no special case is needed for the handler which actually raised the `GestureEvent`. This handler will be the first to receive and respond to the `GestureEvent`, which it will then propagate to any other handlers. The propagation occurs simply because the `event:view:` method returns `NO`, as if it did not handle the event at all.

```
= GestureEventHandler ...
- classes:gesture_classes view:v {
    id c, seq = [gesture_classes eachElement];
    while( c = [seq next] ) {
        if([class_set addNTest:c]) {   /* added new element? */
            gassoc[ngassocs].sclass = c;
            gassoc[ngassocs].view = v;
            ngassocs++;
        }
    }
    return self;
}
```

Each gesture handler that could have handled the initial event sends the gesture handler that did handle the initial event the `classes:view:` message. The latter handler then adds each gesture class to its `class_set`. If the gesture class was not previously there, it is associated with the passed view via the `gassoc` array. This membership test assures that when a given gesture class is expected by more than one view (at the initial point), the topmost view will be associated with the gesture class.

By the time the `GestureEvent` has finished propagating, the `class_set` variable of the instigator will have as elements the gesture classes (`SemClass` objects, actually) that are valid given the initial event. The `gassoc` variable of the instigator will associate each such gesture class with the view that will be affected if the gesture being entered turns out to be that class.

The search for the set of valid gesture classes may be relatively expensive, especially if there are a significant number of views under the initial event and each view has a number of event handlers associated with it. The substantial fraction of a second consumed by the search had an unfortunate interaction with the lower level window manager interface that resulted in an increase in recognition errors. When queried, the low-level window manager software returns only the latest mouse event, discarding any intermediate mouse events that occurred since it was last queried. The time interval between the first and second point of the gesture was often many times larger than the interval between subsequent pairs of points. More importantly, it was much larger than that of the first and second points of the gesture examples used to train the classifier. Details at the beginning of gestures would be lost, and some features, such as the initial angle, would be significantly different. The substantial delay in sampling the second point of the gesture thus caused the classifier performance to degrade.

There are a number of possible solutions to this problem. The window manager software could be set to not discard intermediate mouse events, thus resulting in similar data in the actual and training gestures. This would result in a large additional number of mouse events, and a corresponding increase in processing costs, making the system appear sluggish to the user if events could not be processed as fast as they arrived. Or, the search for gesture classes could be postponed until after the gesture was collected. This would result in a substantial delay after the gesture was collected, again making the system appear sluggish to the user. The solution finally adopted was to poll the window manager during the raising of `GestureEvents`. (In the interest of clarity, the code in `XyEventHandler` and `EventHandlerList` which did the polling was not shown.) After this modification, running `GestureEventHandlers` received input events at the same rate as the `GestureEventHandlers` used for training, improving recognition performance considerably.

The polling resulted in new mouse events being raised before the `GestureEvent` was finished being propagated. The result was a kind of pseudo-multi-threaded operation, with many of the typical problems which arise when concurrency is a possibility. `GestureEventHandlers` were complicated somewhat, since, for example, they had to explicitly deal with the possibility that the end of the gesture might be seen before the set of possible gesture classes was calculated. Also, the event handling methods for `GestureEventHandlers` had to be made reentrant. The complications have been omitted from the code shown here, since they tend to make the program much more difficult to understand.

The end of a gesture is indicated either by a timeout event (resulting in a `timedout:` message being sent to the `GestureEventHandler`), or by the stop predicate being satisfied (resulting in the `activeTerminator:` message being sent to the handler). The third alternative, eager recognition (Chapter 4), has not yet been integrated into the GRANDMA gesture handler, though it has been tested in non-GRANDMA applications (see Section 9.2).

```
= GestureEventHandler ...
- timedout:e { if( !  [self gesture:gesture] )
                    [self deactivate]; return nil; }

- activeTerminator:e {
    [env str:"currentEvent" value:[e copy]];
```

```
    if(!  manip_phase) [self gesture:gesture];
    return self;
}
```

Both methods result in the `gesture:` message being sent when the gesture has been completely collected. The `gesture:` message returns `nil` if the gesture has no semantics to be evaluated during the manipulation phase. This is checked by the `timedout:` method, and in this case the handler simply deactivates itself immediately. This is typically used by gesture classes whose recognition semantics change the mouse tool (*e.g.* a delete gesture that changes the mouse cursor to a delete tool); a timeout deactivates the gesture handler immediately, allowing the mouse to function as a tool as long as the mouse button is held.

The `GenericEventHandler` code arranges for the `deactivate` message to be sent immediately after the `activeTerminator:` message, so there is no need for the `activeTerminator:` method to explicitly send `deactivate`. The environment is changed so that the semantic expression evaluated in the `deactivate` method executes in the correct environment. The `gesture:` method is called if the handler is still in the gesture collection phase, *e.g.* if the gesture end was indicated by releasing the mouse button rather than a timeout.

```
= GestureEventHandler ...
- deactivate {
    id r;
    if(manip_phase && sclass)
        eval([sclass done_expr], env, TypeId, &r);
    return [super deactivate];
}
```

The `gesture:` method sets the `sclass` field to the `SemClass` object of the recognized gesture. The *done expression*, the last of three semantic expressions, is evaluated immediately before the gesture handler is deactivated.

```
= GestureEventHandler ...
- (BOOL)event:e { return ignoring ?  NO : [super event:e]; }

- activeHandler:e {          /* new mouse point */
    [env str:"currentEvent" value:[e copy]];
    if( manip_phase) { id r;    /* in manipulation phase */
        if(sclass) eval([sclass manip_expr], env, TypeId, &r);
    }
    else {                      /* still in collection phase */
        int x = [e [loc x]], y = [e [loc y]];
        [gesture newevent:e];       /* update feature vector */
        [view updatePicture:
               [picture line:lastx :lasty :x :y]]; /* ink */
        lastx = x; lasty = y;
    }
    return self;
```

```
}
```

Once activated, the `GestureEventHandler` functions just like any other `GenericEventHandler` except that it will not handle any events if its `ignoring` flag is set. The active event handler does different things depending on whether the gesture handler is in the collection phase or the manipulation phase. In the former case, the current event location is added to the gesture, and a line connecting the previous location to the current one is drawn on the display. In the latter case, the *manipulation expression* associated with the gesture (the second of the three semantic expressions) is evaluated.

```
= GestureEventHandler ...
- gesture:g { /* called when gesture collection phase in complete */
    double a, d;
    id r;
    id classifier;
    register struct gassoc *ga;
    id c, class;
    id curevent;

    manip_phase = YES;
    [wall timeout:0]; [wall deactivate:timeouteh];
    [view _unhang:picture];    /* erase inking */
    [picture discard]; picture = nil;

    /* inform interested views (only used in a training session) */
    if([view respondsTo:@selector(gesture:)])
        [view gesture:g];

    if(classify) {
        /* find a classifier for the set; create it if necessary */
        classifier = [Classifier lookupOrCreate:class_set];
        /* run the classifier on the feature vector of the collected gesture */
        class = [classifier classify:[g fv]
                          ambigprob:&a distance:&d];
        sclass = nil;
        if(class == nil || a < AmbigProb || d > MaxDist)
            return [self reject];          /* rejected */

        /* find the class of the gesture in the gassoc array */
        for(ga = gassoc; ga < &gassoc[ngassocs]; ga++)
            if([ga->sclass gclass] == class)
                break;
        if(ga == &gassoc[ngassocs])
            return [self error:"gassocs?"];
```

```
            /* the gassoc entry gives the both the view at which the gesture */
            /* is directed and the semantic expressions of the gesture */
            sclass = ga->sclass;
            [env str:"view" value:ga->view];
            [env str:"endEvent"
                 value:curevent=[env atStr:"currentEvent"]];
            eval([sclass recog_expr], env, TypeId, &r);
            if((c = [sclass manip_expr]) != nil &&
               [c val] != nil)
                  eval(c, env, TypeId, &r);
            else {   /* raise event */
                 if(curevent) {
                      ignoring = YES;
                      if(mousetool) [curevent tool:mousetool];
                      [wall raise:curevent];
                 }
                 if( (c = [sclass done_expr]) == nil
                    || [c val] == nil)
                        return nil;
            }
       }
       return self;
  }
```

The `gesture:` method is called when the entire gesture has been collected. It sets the variable `manip_phase` to indicate the handler is now in the manipulation phase of the gestural input cycle, deactivates the timeout event handler, and erases the gesture from the display. If the view associated with the handler responds to `gesture:` it is sent that message, with the collected gesture as argument. This is the mechanism by which example gestures are collected during training: one handler collects the gesture, sends its view (typically a kind of `WallView` devoted to training) the example gesture, which adds it to the `GestureClass` being trained.

In the typical case, the gesture is to be classified. The `Classifier` factory method named `lookupOrCreate:` is called to find a gesture classifier which discriminated between elements of the `class_set`. If no such classifier is found, this method calculates one and caches it for future use. (This lookup and creation could possibly have been done in the pseudo-thread that was spawned during the first point of the gesture, but was not, since most of the time the lookup finds the classifier in the cache, and it was not worth the additional complication and loss of modularity to add polling to the classifier creation code.) The returned classifier is then used to classify the gesture. In addition to the class, the probability that the classification was ambiguous and the distance of the example gesture to the mean of the calculated class are returned. These are compared against thresholds to check for possible rejection of the gesture (see Section 3.6).

The elements of the `gassoc` array are searched to find the one whose gesture class is the class returned by the classifier. This determines both the semantics of the recognized gesture and the view at which the gesture was directed. The `sclass` field is set to the `SemClass` object associated with the recognized gesture, and then the *recognition expression*, the first of the three semantic expressions, is evaluated in an environment in which `"startEvent"`, `"currentEvent"`, `"endEvent"` and `"view"` are all appropriately bound.

If it exists, the manipulation expression is evaluated immediately after evaluating the recognition expression. If there is no manipulation expression, the current event is reraised on the assumption that its tool may wish to operate on a view. The `ignoring` flag is set so that the active handler does not attempt to handle the event it is about to raise. Furthermore, the semantics of the gesture may have changed the current mouse tool. If so, the tool field of the current event would be incorrect, and is changed to the new tool before the event is raised. In order for this to work, any gesture semantics that wish to change the current mouse tool must do so by sending the `mousetool:` message to the gesture handler instead of directly to the wallview.

```
= GestureEventHandler ...
- mousetool:_mousetool {
        mousetool = _mousetool;
        return [super mousetool:_mousetool];
}
```

The `gesture:` method returns `nil` if there are no manipulation or done semantics associated with the recognized gesture class. As seen, this is a signal for the handler to be deactivated immediately after the gesture is recognized.

7.5 Gesture Classification and Training

In this section the implementation of classes which support the gesture classification and training algorithms of Chapter 3 is discussed.

At the lowest level is the class `Gesture`. A `Gesture` object represents a single example of a gesture. These objects are created and manipulated by `GestureEventHandlers`, both during the normal gesture recognition that occurs when an application is being used, and during the specification of gesture classes when training classifiers.

7.5.1 Class `Gesture`

Internally, a gesture object is an array of points, each consisting of an x, y, and time coordinate. Another instance variable is the `GestureClass` object of this example gesture, which is non-`nil` if this example was specified during training. Intermediate values used in the calculation of the example's feature vector, as well as the feature vector itself, are also stored. Also, an arbitrary string of text may be associated with a `Gesture` object.

For brevity, detailed listing of the code for the `Gesture` class is avoided. The interesting part, namely the feature vector calculation, has already been specified in detail in Chapter 3 and C code is

shown in Appendix A. Instead of listing more code here, an explanation of each message Gesture objects respond to is given.

A new gesture is allocated and initialized via `g = [Gesture new]`. Adding a point to a Gesture objects is done by sending it the newevent message: `[g newevent:e]`, which simply results in the call: `[g x:[[e loc] x] y:[[e loc] y] t:[e time]]`. The `x:y:t:` method adds the new point to the list of points, and incrementally calculates the various components of the feature vector (see Section 3.3). The call `[g fv]` returns the calculated feature vector. The methods `class:`, `class`, `text:`, and `text` respectively set and get the class and text instance variables.

A Gesture object can dump itself to a file via `[g save:f]` (given a file stream pointer `FILE *f`) and can also initialize itself from a file dump using `[g read:f]`. Using `save:`, a number of gesture objects may dump themselves sequentially into a single file, and could then be read back one at a time using `read:`. All examples of a given gesture class are stored in a single file via these methods.

The call `[g contains:x:y]` returns a boolean value indicating if the gesture g, when closed by connecting its last point to its first point, contains the point `(x,y)`. This is useful for testing, for example, if a given view has been encircled by the gesture, enabling the gesture to indicate the scope of a command. (The algorithm for testing if a point is within a given gesture is described at the end of section 7.7.3.)

7.5.2 Class GestureClass

The class GestureClass represents a gesture class. A gesture class is simply a set of example gestures, presumably alike, that are to be considered the same for the purposes of classification. The input to the gesture classifier training method is a set of GestureClass objects; the result of classifying a gesture is a GestureClass object.

```
= GestureClass:  NamedModel {
    id      examples;
    Vector  sum, average;
    Matrix  sumcov;
    int     state;
    STR     text;
}
```

GestureClass is a subclass of NamedModel, itself a subclass of Model. GestureClass is a model so that it can have views, enabling new gesture classes to be created and manipulated at runtime. Please do not confuse GestureClass with GestureEventHandler objects; a GestureClass serves only to represent a class of gestures, and itself handles no input. A NamedModel augments the capabilities of a Model by adding functions that facilitate reading and writing the model to a file. Also, models read this way are cached, so that a model asked to be input more than once is only read once. This is important for gesture class objects, since a single GestureClass object may be a constituent of many different classifiers, and it is necessary that every classifier recognizing a particular class refer to the same GestureClass object.

The GestureClass instance variable examples is a Set of examples which make up the class. The field sum is the vector that the sum of all feature vectors of every example in the class; average is sum divided by the number of examples. The covariance matrix for this class may be found by dividing the matrix sumcov by one less than the number of examples. The calculation of classifiers is slightly more efficient given sumcov matrices, rather than covariance matrices, as input (see Chapter 3). C code to calculate the sumcov matrices incrementally is shown in Appendix A.

The state instance variable is a set of bit fields indicating whether the average and sumcov variables are up to date. The text field allows an arbitrary text string to be associated with a gesture class.

The addExample: method adds a Gesture to the set of examples in the gesture class, incrementally updating the sum field. The removeExample: method deletes the passed Gesture from the class, updating sum accordingly. The examples method returns the set of examples of this class, average returns the estimated mean of the feature vector of all the examples in this class, nexamples returns the number of examples, and sumcov returns the unnormalized estimated covariance matrix.

7.5.3 Class GestureSemClass

```
= GestureSemClass:  NamedModel {
        id          gclass;
        id          recog, manip, done;
}
```

GestureSemClass objects are named models, enabling them to be referred to by name for reading or writing to disk, and for being automatically cached when read. The purpose of GestureSemClass objects is to associate a given gesture class with a set of semantics. It is necessary to have a separate class for this because a given GestureClass may have more than one set of semantics associated with it.

In addition to methods for setting and getting each field, there are methods for reading and writing GestureSemClass objects to disk. GestureSemClass uses Objective C's Filer class to read and write each of the three semantic expressions (recog, manip, and done). The availability of the Filer is another advantage of using Objective C [28]. In a typical interpreter, a substantial amount of coding would be required to read and write the intermediate tree form of the program to and from disk files. The Filer, which allows the writing to and from disk of any object (at least those having no C pointers besides strings and ids as instance variables), made it trivial to save interpreter expressions to disk.

Along with the semantics, the disk file of a GestureSemClass contains only the name of gestureClass object referred to by gclass. When reading in a GestureSemClass, the name is used to read in the associated GestureClass. Since GestureClass is a NamedModel, there will be only one GestureClass object for each distinct gesture class.

7.5.4 Class `Classifier`

The `Classifier` class encapsulates the basic gesture recognition capabilities in GRANDMA. Each `Classifier` object has a set (actually an `OrdCltn`) of gesture classes between which it discriminates. Each `Classifier` object contains the linear evaluation function for each class (as described in Chapter 3), and the inverse of the average covariance matrix, which is used to calculate the discrimination functions, as well as to calculate the Mahalanobis distance between two of the component gesture classes, or a given gesture example and one of the gesture classes.

```
= Classifier :  Object {
        id              gestureclasses;
        int             nclasses, nfeatures;
        Vector          cnst, *w;          /* discrimination functions */
        Matrix          invavgcov;
        int             hashvalue;
}
```

`[Classifier lookupOrCreate:classes]` returns a classifier which discriminates between the gesture classes in the passed collection `classes`. The method for `lookupOrCreate:` caches all classifier objects which it creates; thus, if it is subsequently passed a set of gesture classes which it has seen before, it returns the classifier for that set without having to recompute it. The search for an existing classifier for a given set of gestures is facilitated by the `hashvalue` instance variable, which is calculated by "XORing" together the object ids of the particular `GestureClass` objects in the set.

When necessary, the `lookupOrCreate:` method creates a new classifier object, initializes its `gestureclasses` instance variable and then sends itself the `train` message. The `train` method implements the training algorithm of chapter 3.

```
- train {
    register int i, j;
    int denom = 0;
    id c, seq;
    register Matrix s, avgcov;
    Vector avg;
    double det;

    /* eliminate any gesture classes with no examples */
    [self eliminateEmptyClasses];

    /* calculate the average covariance matrix from the (unnormalized)
       covariance matrices of the gesture classes. */
    avgcov = NewMatrix(nfeatures, nfeatures);
    ZeroMatrix(avgcov);
    for(seq = [gestureclasses eachElement];
        c = [[seq next] gclass]; ) {
            denom += [c nexamples] - 1;
```

```
            s = [c sumcov];
            for(i = 0; i < nfeatures; i++)
                for(j = i; j < nfeatures; j++)
                    avgcov[i][j] += s[i][j];
        }

        if(denom == 0) [self error:"no examples"];
        for(i = 0; i < nfeatures; i++)
            for(j = i; j < nfeatures; j++)
                avgcov[j][i] = (avgcov[i][j] /= denom);

        /* invert the average covariance matrix */
        invavgcov = NewMatrix(nfeatures, nfeatures);
        det = InvertMatrix(avgcov, invavgcov);
        if(det == 0.0)
            [self fixClassifier:avgcov];

        /* calculate the discrimination functions:
           w[i][j] is the weight on the jth feature of the ith class.
           cnst[i] is the constant term for the ith class. */

        w = allocate(nclasses, Vector);
        cnst = NewVector(nclasses);
        for(i = 0; i < nclasses; i++) {
            avg = [[[gestureclasses at:i] gclass] average];
                                        /* w[i] = avg*invavgcov */
            w[i] = NewVector(nfeatures);
            VectorTimesMatrix(avg, invavgcov, w[i]);
            cnst[i] = -0.5 * InnerProduct(w[i], avg);
        }
    }
```

The `eliminateEmptyClasses` method removes any gesture classes from the set which have no examples. The (estimated) average covariance matrix is then computed, and an attempt is made to invert it. If it is singular, the `fixClassifier:` method is called, which creates a usable inverse covariance matrix as described in Section 3.5.2. (C code for fixing the classifier is shown in Appendix A.)

Given the inverse covariance matrix, the discrimination functions for each class are calculated as specified in Section 3.5.2. The weights on the features for a given class are computed by multiplying the inverse average covariance matrix by the average feature vector of the class, while the constant term is computed as negative one-half of the weights applied to the class average. This constant computation gives optimal classifiers under the assumptions of that all classes are equally likely and the misclassifications between classes have equal cost (also assumed is multivariate normality and a

common covariance matrix). The `Classifier` class provides a `class:incrconst:` method which allows the constant terms for a given class to be adjusted if the application so desires.

The call `[Classifier trainall:classes]` causes all `Classifier` objects whose set of gestures includes all the gestures in the set `classes` to be retrained (by sending them the `train:` message). This is useful whenever training examples are added or deleted, since all the classifiers depending on this class can then be recalculated at once. Generally a classifier may be retrained in less than a quarter second; Section 9.1.7 presents training times in detail.

Classifying a given example gesture is done by the `classify:ambigprob:distance:` method. This method is passed the feature vector of the example gesture, and evaluates the discrimination function for each class, choosing the maximum. If desired, the probability that the gesture is unambiguous, as well as the Mahalanobis distance of the example gesture from the its calculated class are also computed; this allow the callers of the classification method to implement rejection options if they so choose.

```
- classify:(Vector)fv
  ambigprob:(double *)ap distance:(double *)dp
{
    double maxdisc, disc[MAXCLASSES];
    register int i, maxclass;
    double denom, exp();
    id class;

    for(i = 0; i < nclasses; i++)
        disc[i] = InnerProduct(w[i], fv) + cnst[i];

    maxclass = 0;
    for(i = 1; i < nclasses; i++)
        if(disc[i] > disc[maxclass])
            maxclass = i;
    class = [[gestureclasses at:maxclass] gclass];

    if(ap) {      /* calculate probability of non-ambiguity */
        for(denom = 0, i = 0; i < nclasses; i++)
            denom += exp(disc[i] - disc[maxclass]);
        *ap = 1.0 / denom;
    }

    if(dp)        /* calculate distance to mean of chosen class */
        *dp = [class d2fv:fv sigmainv:invavgcov];

    return class;
}
```

`Classifier` objects respond to numerous messages not yet mentioned. The `evaluate` message causes the example gestures of each class to be classified, so that the recognition rate of the classifier may be estimated. Of course, the procedure of testing the classifier on the very examples it was trained upon results in an overoptimistic evaluation, but it nonetheless is useful. By sending the particular gesture classes and examples `text:` messages, the result of the evaluation is fed back to the user, who can then see which examples of each class were classified incorrectly. A high rate of misclassification usually points to an ambiguity, indicating a poor design of the set of gestures to be recognized. The ambiguity is typically fixed by modifying the gesture examples of one or more of the gesture classes. The incorrectly classified examples indicate to the gesture designer which gesture classes need to be revised.

`Classifier` objects also respond to messages which save and restore classifiers to files, as well as messages which cause the internal state of a classifier to be printed on the terminal for debugging purposes, and a matrix of the Mahalanobis distances between class pairs to be printed (so that the gesture designer can get a measure of how confusable the set of gestures is).

7.6 Manipulating Gesture Event Handlers at Runtime

One goal of this work was to provide a platform that allows experimentation with different gestural interfaces to a given application. To this end, GRANDMA was designed to allow gesture recognizers to be manipulated at runtime. Gesture classes may be added or deleted, training examples for each class may also be added or deleted, and the semantics of a gesture class (with respect to a particular handler) may all be specified at runtime. In addition, gestures as a whole, or particular gesture event handlers, may be turned on and off at runtime, allowing, for example, easy comparison between gesture-based and click-drag interfaces to the same application program. This section discusses the interface GRANDMA presents to the user that facilitates the manipulation of gesture handlers at runtime.

The `View` class implements the `editHandlers` method. When sent `editHandlers`, a view creates a new window (if one does not already exist) as shown in figure 7.2. The top row is a set of pull down menus. Each subsequent row lists the passive event handlers for the view, its class, its superclass, and so on up the class hierarchy until the `View` class. The event handlers are listed in the order that they are queried for events, from top to bottom, and within a row, from left to right.

The "Mouse mode" menu item controls which mouse cursor is currently active in the window. With the normal mouse (indicated by an arrow), the user is able to drag the individual event handler boxes so as to rearrange the order. (The other mode, "edit handler," will be discussed shortly.) A handler may also be dragged into the trash box, in which case it is removed from the list of handler associated with a view or view class. A handler may be dragged into the dock; anything in the dock will remain visible when the handler lists for a different view are accessed. A handler dragged into the dock reappears on its original list as well; thus the dock allows the same event handlers to be shared between different objects and between different classes.

The "create handler" menu item results in a pull-down menu of all classes which respond to the `passive` message. Thus, at runtime new handlers may be created and associated with any view object or class. For example, a drag handler may be created and attached to an object, which can

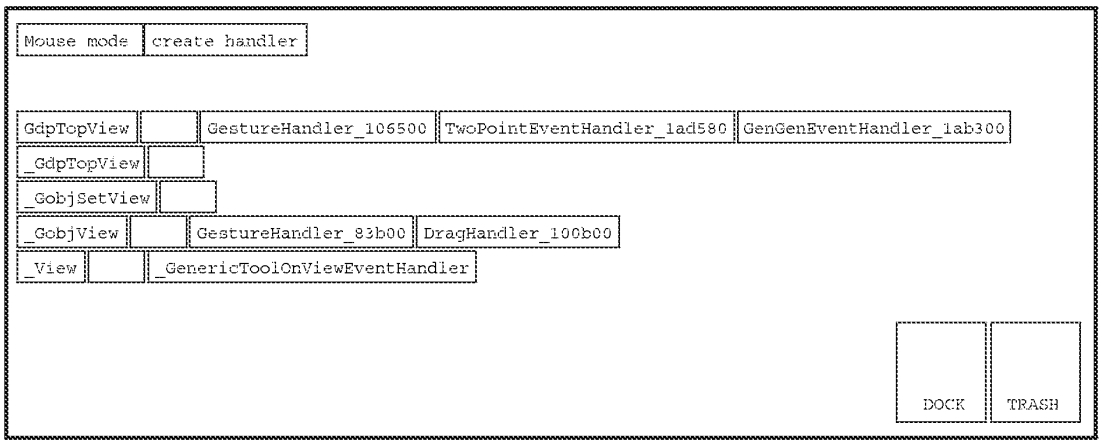


Figure 7.2: Passive Event Handler Lists

then be dragged around with the mouse. New gesture handlers may also be created this way.

The other mouse cursor, "edit handler", may be clicked upon any passive event handler. It results in a new window being created which shows the details of a particular edit handler. Figure 7.3 shows the window for a typical gesture handler.

At the top left of the window is the "Mouse mode" pull down menu, used in the unlikely event that one wishes to examine the handlers of any of the views in this window. To the right is the name of this event handler, constructed by concatenating the class of the handler with its internal address.

The next three rows show three `EventExpr` objects; these are the starting predicate, handling predicate and stopping predicate of the gesture handler. Each item in the predicate display is a button that shows a pop-up menu; it is thus a simple matter to change the predicates at runtime. For example, the start predicate may be changed from matching only `PickEvents` to matching all `DragEvents`. The kind of tool expected may also be changed at runtime, as well as attributes of the tool (*e.g.* a particular mouse button may be specified). If desired, the entire predicate expression may be replaced by a completely new expression. In all cases, the changes take effect immediately.

The window contents thus far discussed are common to all `GenericEventHandlers`. The following ones are particular to `GestureEventHandlers`. First there are a set of buttons ("new class", "train", "evaluate", "save"). Below this are some squares, each representing a gesture class recognized by this handler. In each square is a miniaturized example gesture, some text associated with the class, and a small rectangle which names the class. The text typically shows the result of the evaluation of the particular gesture recognizer for this set of classes when run on the examples used to train it. The small rectangles may be dragged (copied) into the dock. Each such rectangle represents a particular gesture class. Any rectangles in the dock will remain there when another gesture handler is edited. Each then may be dragged into any gesture class square, where it replaces the existing class. Typically, a rectangle from the dock is dragged into empty class square (created by the "new class" button); this is the way multiple gesture handlers can recognize the same class.

Clicking on one of the gesture class squares (but not in the class name rectangle) brings up the

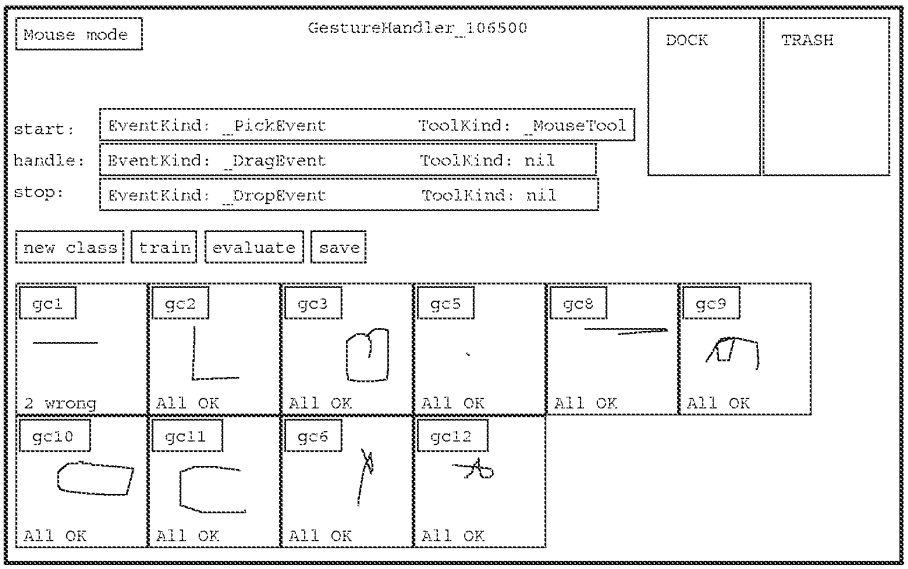


Figure 7.3: A Gesture Event Handler

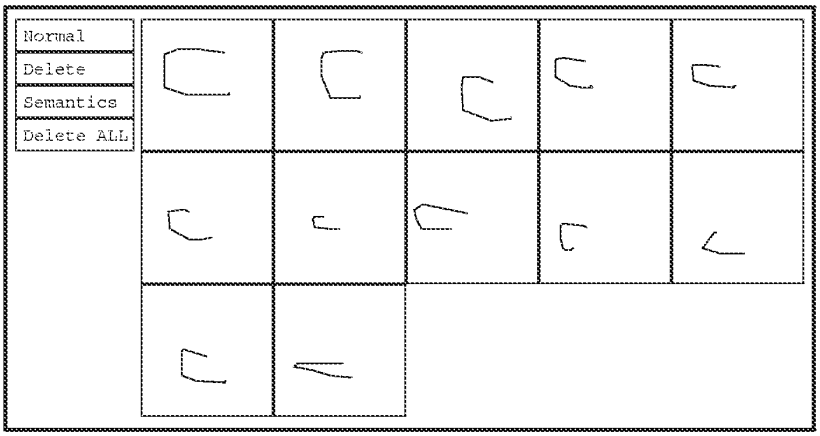


Figure 7.4: Window of examples of a gesture class

window of example gestures, as shown in Figure 7.4. Each square in this window contains a single, miniaturized example of a gesture in this class. These examples are used for training the classifier. A new example may be added simply by gesturing in this window. An example may be deleted by clicking the delete button on the left (which changes the mouse cursor to a delete cursor) and then clicking on the example. A user wishing to change a gesture to something more to his liking simply has to delete all the examples of the class (easily done using the "Delete ALL" button) and then enter new example gestures. The "train" button will cause a new classifier to be built, and the "evaluate" button will cause the examples to be run through the newly built classifier. Any incorrectly classified examples will be indicated by displaying the mistaken class name in the example square; the user can then examine the example to see if it was malformed or otherwise ambiguous.

The "semantics" button in the window of examples causes the semantics of the gesture class to be displayed. This is the subject of the next section.

7.7 Gesture Semantics

GRANDMA contains a simple Objective-C interpreter that allows the semantics of gestures to be specified at runtime. In GRANDMA, the semantics of a gesture are determined by three program fragments per gesture class (per handler). The first program fragment, labeled `recog`, is executed when the gesture is first recognized to be in a particular class. The second fragment, `manip`, is executed on every input event handled by the activated gesture handler after the gesture has been recognized. The third fragment, `done`, is executed just before the handler deactivates itself. The exact sequence of executions was described in detail in section 7.4; this section is concerned with the contents and specification of the program fragments themselves.

7.7.1 Gesture Semantics Code

As mentioned, the semantics of a gesture are defined by three expressions, `recog`, `manip`, and `done`. The kinds of expressions found in practice may be loosely grouped according to the level of the GRANDMA system that they access.

Some semantic expressions deal directly with models, *i.e.* directly with application objects. These are typically the easiest to code and understand. An example from the GSCORE application discussed in section 8.2 is the sharp gesture. GSCORE is an editor for musical scores. In GSCORE, making an "S" gesture over a note in the score causes the note to be "sharped", which is indicated in musical notation by placing the sharp sign "#" before the note. The class `Note` is a model in the GSCORE application, and one of its methods is `acc:` which sets the accidental of a note to one of DOUBLEFLAT, FLAT, NATURAL, SHARP, DOUBLESHARP, or NOACCIDENTAL.

The sharp gesture, performed by making an "S" over a `NoteView`, has the semantics:

```
recog = [ [view model] acc:SHARP ];
manip = nil;
done = nil;
```

In these semantics, the `Note` object (the model of the `NoteView` object) is directly sent the `acc:` message when the sharp gesture is recognized. The model then changes its internal state to

reflect the new accidental, and then calls `[self modified]` which will eventually result in the display updated to add a sharp on the note.

Note that the semantic expressions are evaluated in a context in which certain names are assumed to be bound. In the above example, obviously `view` and `SHARP` must be bound to their correct values for the code to work. Section 7.4 described how the `GestureEventHandler` creates an environment where `view` is bound to the view at which the gesture is directed, `startEvent` is bound to the initial event of the gesture, `endEvent` is bound to the last event of the gesture (*i.e.* the event just before the gesture was classified), and `currentEvent` is bound to the most recent event, typically a `MoveEvent` during the manipulation phase. A particular application may globally bind application-specific symbols (such as `SHARP` in the above example) in order to facilitate the writing of semantic expressions.

Instead of dealing directly with the model, the semantics of a gesture may send messages directly to the view object. In the score editor, for example, the delete gesture (in the handler associated with a `ScoreEvent`) might have the semantics

```
recog = [view delete];
manip = nil;
done = nil;
```

(The actual semantics are slightly more complicated since they also change the mouse cursor; see Section 8.2 for details.) The `delete` method for the typical view just sends `delete` to its model, perhaps after doing some housekeeping.

The semantic expressions of a gesture are invoked from a `GestureEventHandler`, and the sending of messages to models and views seen so far is typical of many different kinds of event handlers. Another thing that event handlers often do (see in particular section 6.7.9 for a discussion of the `DragHandler`) is raise events of their own. There are many reasons a handler might wish to do this. A `DragHandler` raises events in order to make the view being dragged be considered a virtual tool. As mentioned previously, a handler might also raise events in order to simulate one input device with another. (For example, imagine a `SensorFrameMouseEmulator` which responds to `SensorFrameEvents`, raising `DragEvents` whose tool is the current `GenericMouseTool` so as to simulate a mouse with a Sensor Frame.) One of the main purposes of having an active event handler list and a list of passive events handlers associated with each view is to allow this kind of flexibility. In the Smalltalk MVC system, the pairing of a single controller with a view really constrains the view to deal only with a single kind of input, namely mouse input. In GRANDMA, a view can have a number of different event handlers, and thus may be able to deal with many different input devices and methods.

In GRANDMA, gesture-based applications are typically first written and debugged with a more traditional menu driven, click-and-drag, direct manipulation interface. Given that gestures are added on top of this existing structure, there is another level at which gesture semantics may be written. At this level, the gesture semantics emulate, for example, the mouse input that would give the appropriate behavior. In other words, the gesture is translated into a click-and-drag interaction which gives the desired result.

An example of this from the score editor is the placement of a note into a score. In the click-and-drag interface, adding a note to the score involves dragging a note of appropriate duration from

a palette of notes to its desired location in a musical staff. This is implemented by having the `NoteView` be a virtual tool which sends a message to which `StaffView` objects respond. While the note is being dragged, a `DragHandler` raises an event whose tool is a `NoteView` which will be processed by the `GenericToolOnView` handler when the note is over the `StaffView`.

In the gesture-based interface, there is a gesture class for each possible note duration recognized by handler associated with the `StaffView` class. The semantics for the gesture which gives rise to an eighth note are

```
recog = [[[noteview8up viewcopy] at:startLoc]
                reraise:currentEvent];
manip = nil;
done = nil;
```

The symbol `noteview8up` is bound to the view of one of the notes in the palette; it is copied and moved to the starting location of the gesture. The `currentEvent` (either a `MoveEvent` or `DropEvent` which ended the gesture) is copied, its tool field is set to the copy of the note view, and the resulting event is raised. The moving of the note and the raising of a new event is exactly what a `DragHandler` does; the effect is to simulate the dragging of a note to a particular location. Note that the note is moved to `startLoc`, the starting point of the gesture, which necessarily is over a `StaffView` (otherwise this gesture handler would never have been invoked). Thus, the handlers for `StaffView` will handle the event, and use the location of the note view to determine the new note's pitch and location in the score.

It would have been possible in the semantics to simulate the mouse being clicked on the appropriate note in the palette and then being dragged onto the appropriate place in the staff. In this case, that was not done as it would be needlessly complex. The point is that, due to the flexibility of GRANDMA's input architecture, the writer of gesture semantics can address the system at many levels of abstraction, from simulated input to directly dealing with application objects.

The example semantics seen thus far have only had `recog` expressions, which are evaluated at recognition time. The following example, which implements the semantics of a gesture which creates a line and then allows the line to be rubberbanded, illustrates the use of `manip`:

```
recog = [[view createLine] endpoint0at:startLoc];
manip = [recog endpoint1at:currentLoc];
done = nil;
```

In this example, `view` is assumed to be a background view, typically a `WallView` of a drawing editor program (Section 8.1 discusses GDP, a gesture-based drawing editor). Sending it the `createLine` message results in a new line being created in the window, whose first endpoint is the start of the gesture. The other endpoint of the line moves with the mouse after the gesture has been recognized; this is the effect of the `manip` expression. Note the use of `recog` as a variable to hold the newly created line object. If desired, the semantics programmer may create other local variables to communicate between different (or even the same) semantic expressions.

7.7.2 The User Interface

GRANDMA allows the specification of gesture semantics to be done at runtime. In the current implementation, the semantics must be specified at runtime; there is no facility for hardwiring the

semantic expressions of a given gesture into an application. Currently, the semantics of a gesture class are read in from a file (as are examples of the gesture class) each time an application is started. The semantics of a gesture may only be created or modified using the user interface facilities discussed in this section.

Gesture semantics are currently specified using a limited set of expressions. An expression may be a constant expression (integer or string), a variable reference, an assignment, or a message send. Each expression has its obvious effect: a constant evaluates to itself, a variable evaluates to its value in the current environment, an assignment evaluates to the evaluation of its right hand side (with the side effect of setting the variable on the left hand side), and a message send first evaluates the receiver expression and each argument expression, and then sends the specified message and resulting arguments to the receiver. The value of a message expression is the value that the receiver's method returns. For programming convenience, integer, string, and objects are converted as needed so that the types of the arguments and receiver of a message send match what is expected by the message selector.

Figure 7.5 shows the window activated when the "Semantics" button of a gesture class is pressed. At the top of the window are a row of buttons used in the creation of various kinds of expressions. They work as follows:

new message The new message button creates a template of a message send, with a slot for the receiver and the message selector. Any expression may then be dragged into the receiver ("REC?") slot. Clicking on the "SELECTOR?" box causes a dialogue box to be displayed (figure 7.6). Users can then browse through the class hierarchy until they find the message selector they desire, which can then be selected. The "+" and "-" buttons may be used to switch between factory and instance methods. The starting point in the browsing is set to the class of the receiver, when it can be determined. Once the selector has be okayed, the template changes to have a slot for each argument expected by the selector, as shown into figure 7.7. Any expression may then be dragged into the argument slots. In particular, gesture attributes (see below) are often used.

new int This button creates a box into which an integer may be typed.

new string This button creates a box into which a string may be typed.

new variable This button creates a template (1 = VALUE?) for assigning a variable into which the name of a variable may be typed. Any expression may then be dragged into the "VALUE?" slot. The entire assignment expression may be dragged around by the "=" sign. Attempting to drag the variable name on the left hand side actually copies the variable name before allowing it to be dragged; this resulting expression (simply the name of the variable) may be used anywhere the value of the variable is needed.

factory This button generates a constant expression which is the object identifier of an Objective C class (also known as a "factory"). Pressing the button pops up a browser which allows the user to walk through the class hierarchy to select the desired class.

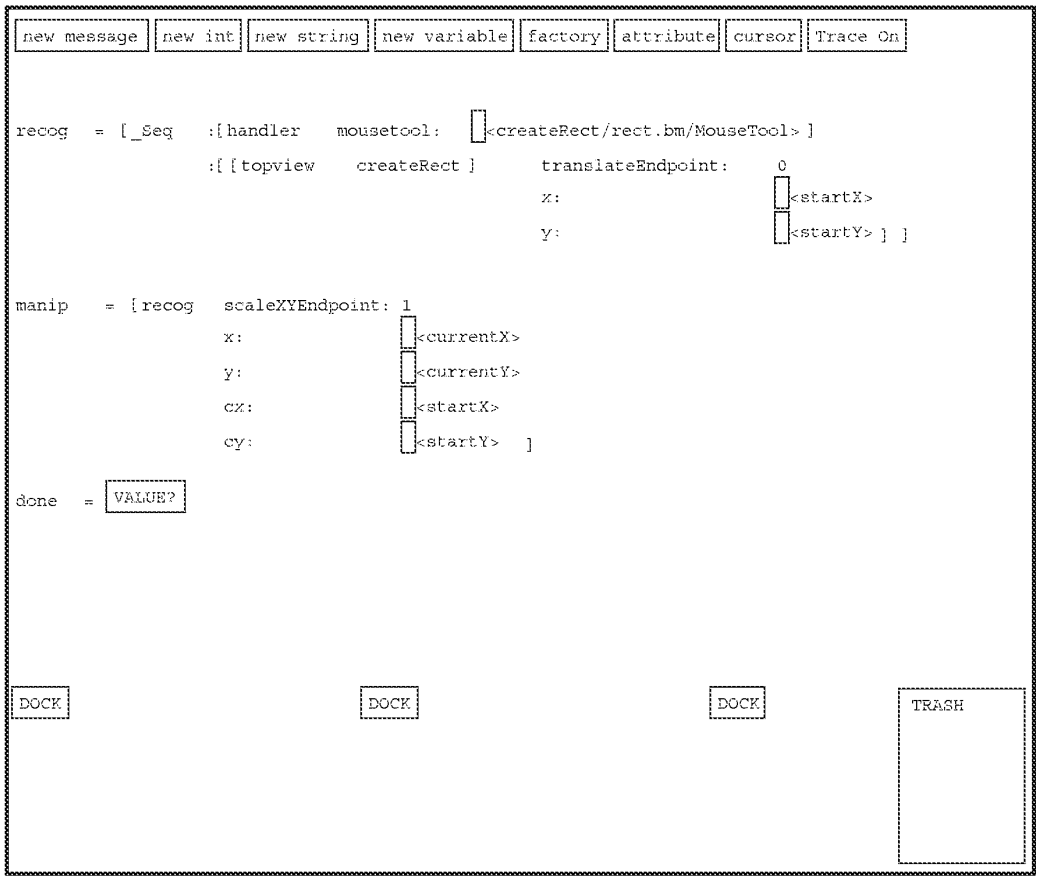


Figure 7.5: The interpreter window for editing gesture semantics

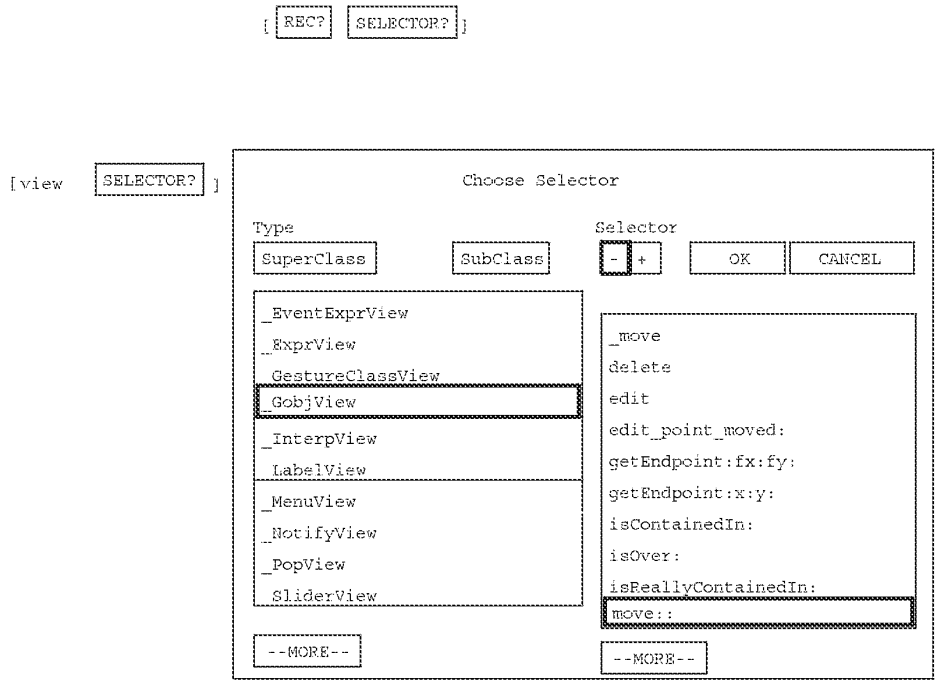


Figure 7.6: An empty message and a selector browser

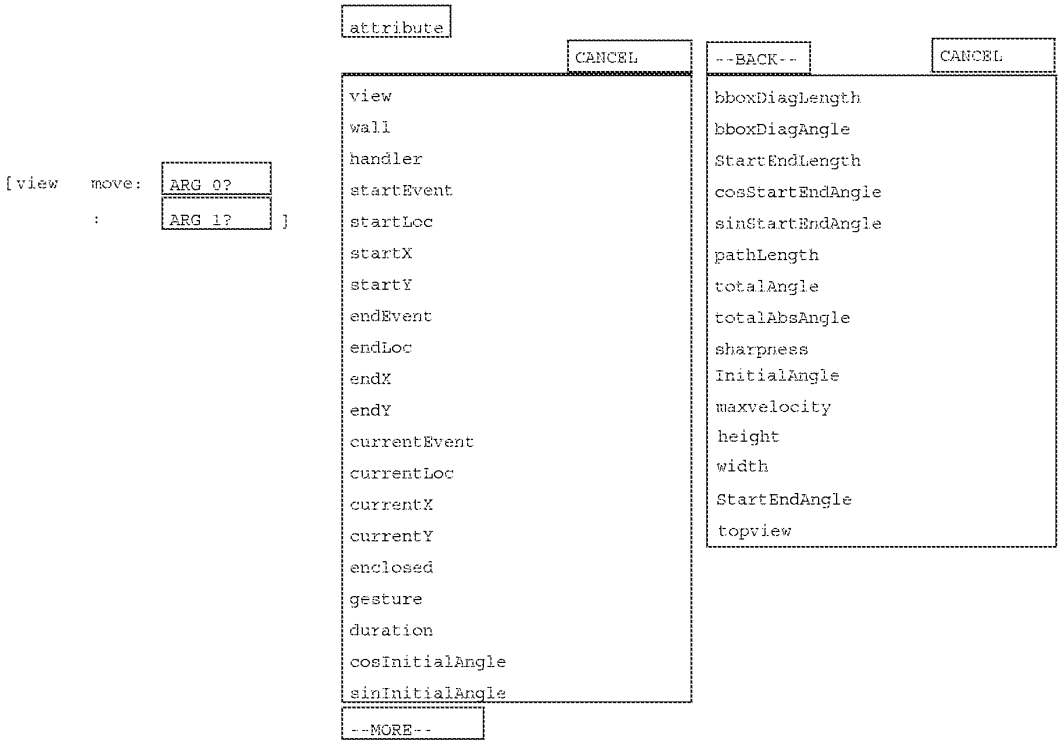


Figure 7.7: Attributes to use in gesture semantics

.

attribute Clicking this button generates a menu of useful subexpressions that are often used in gesture semantics. (Figure 7.7 shows both pages of attributes). The expressions are either variable names, or named messages. As expressions, named messages are distinguishable from variable names by the angle brackets and the small box before the name. Clicking in the box reveals the underlying expression to which the name refers. (Note the angle brackets and box are not shown in the list of attributes but appear once an attribute is selected. Figure 7.5 contains some examples of such attributes.)

Most attributes in the list refer to characteristics of the current gesture (*i.e.* the gesture which causes the semantics to be evaluated). Other attributes refer to the current view, wall, event handler, events, and set of objects enclosed by the gesture. Many examples of using attributes in gesture semantics are covered in the next chapter.

Having the attributes of a gesture available when writing the semantics of the gesture is the embodiment of one central idea of idea of this thesis. The idea is that the meaning of a gesture may depend not only upon its classification, but also on the features of the particular instance of the gesture. For example, in the drawing program it is a simple matter to tie the length of the line gesture to the thickness of the resulting line. This is in addition to using the starting point of the gesture as one endpoint of the line, another example of how gesture attributes are useful in gesture semantics.

cursor This button displays a menu of the available cursors. The cursors are almost always a kind of `GenericMouseTool`, and consists of an icon that has been read in from a file, and the message that the tool sends. The cursors are useful, for example, in semantic expressions that wish to provide some feedback to the user by changing the cursor after the gesture has been recognized.

Trace On This button turns on tracing of the interpreter evaluation loop, which prints the values of all expressions and subexpressions as they are evaluated. This helps the writer of gesture semantics to debug his code.

The middle mouse button brings up a menu of useful operations. "Normal" restores the cursor to the default cursor which drags expressions. "Copy" changes the cursor to the copy cursor, which when used to drag expressions causes them to be copied first. "Hide" hides the semantics window, which is so large that it typically obscures the application window. The various remaining editing commands are useful for examining the event handlers associated with various objects in the user interface, and are not really of general interest to the writer of gesture semantics. They would be of interest if one attempted to add a gestural interface to the interpreter itself.

An expression dragged into a "DOCK" slot remains there even when the gesture class is changed. The dock provides a useful mechanism for sharing code between different gesture classes, or between the same gesture class in different handlers. Any expression dragged into the trash is, of course, deleted.

The above-described interface to the semantics is usually slower to use than a more straightforward textual interface. A straightforward textual interface would require a parser but would still be simpler and better that the current click-and-drag interface. On the other hand, with the

click-and-drag interface it is not possible to make a syntax error. The main reason such an interface was built was to exercise the facilities of the GRANDMA system. Before the project began the author suspected that a click-and-drag interface to a programming language would be awkward, and he was not surprised. He did, however, consider the possibility of building a gesture-based interface to the interpreter, one which might have been significantly more efficient to use than the current click-and drag interface. It should be possible at the present time to add a gesture-based interface to the interpreter without even recompiling, though to date the author has not made the attempt.

7.7.3 Interpreter Implementation

The interpreter internals are implemented in a most straightforward manner. The class `Expression` is a subclass of `Model` and has a subclass for each type of expression: `VarExpr`, `AssignExpr`, `MessageExpr`, and `ConstantExpr` (and some not discussed: `CharEventExpr`, `EventExpr`, and `FunctionExpr`). `AssignExpr` and `MessageExpr` objects have fields which hold their respective subexpressions, while `ConstantExpr` and `VarExpr` objects have fields which hold the constant object and name of the variable, respectively.

Expression Evaluation

All expressions are evaluated in an environment, which is simply an association of names with values (which are objects). Evaluating `VarExpr` objects is done by looking up the variable in an environment and returning its value; `AssignExpr` objects are evaluated by adding or modifying an environment so as to associate the named variable with its value. In addition to the environment that is passed whenever an expression is evaluated, there is a global environment. If a name is not found in the passed environment, it is then looked up in the global environment.

The interpreter has a number of types with which it can deal. Each type is represented by a subclass of class `Type`. An instance of one of these subclasses is a value of that type. The commonly used type classes are `TypeChar`, `TypeId`, `TypeInt`, `TypeShort`, `TypeSTR`, `TypeUnsigned`, and `TypeVoid`. The `TypeId` represents an arbitrary Objective-C object; the others represent their corresponding C type.

Consider the implementation of `TypeInt`:

```
= TypeInt :  Type { int _int; }
+ initialize { [super register:"int"];
               [super register:"long"]; }
+ set_int:(int)v { return [[super new] set_int:v]; }
+ (void *)fromObject:o result:(void *)r
        { *(int *)r = [o asInt]; return r; }
+ toObject:(void *)r { return [self set_int:*(int *)r]; }
- set_int:(int)v { _int = v; return self; }
- (int)asInt { return _int; }
- (short)asShort { return (short)_int; }
- (char)asChar { return (char)_int; }
- (unsigned)asUnsigned { return (unsigned)_int; }
```

```
- (STR)asString:(STR)s { sprintf(s, "%d", _int); return s; }
- (int)Plus:(int)b { return _int + b; }
- (int)Minus:(int)b { return _int - b; }
- (int)Times:(int)b { return _int * b; }
- (int)DividedBy:(int)b { return b == 0 ?
        [self error:"division by zero"], 0 :  _int / b; }
- (int)Mod:(int)b { return b == 0 ?
        [self error:"mod by zero"], 0 :  _int % b; }
- (int)Clip:(int)b :(int)c
        { return _int < b ?  b :  _int > c ?  c :  _int; }
- (int)Times:(int)b Plus:(int)c { return _int * b + c; }
```

The initialize method declares that this type represents the C types "int" and "long." This information is used when reading in the files that the Objective-C compiler writes to describe the arguments and return types of message selectors. A sample line from one of these files is:

```
(id)at::,int,int;
```

This line says that the `at::` method (as implemented by `View`, for example) takes two integers as arguments, and returns an `id`, *i.e.* an object. (In Objective C, the type or signature of a selector such as `at::` must be the same in all classes that provide corresponding methods.) The interpreter reads this line and creates a `Selector` object which records the fact that `at::` expects its first argument to be `TypeInt`, its second argument to be `TypeInt`, and returns a `TypeId`. This `Selector` object is used when a `MessageExpr` whose selector is `at::` is evaluated; it assures that the arguments are converted to machine integers before the `at::` method is invoked.

The knowledge of how to do conversions is embodied in the `fromObject:result:` and `toObject:` methods. The intent is to freely convert between the values represented as machine integers, or characters, etc., and the values represented as objects. Given `int r; id anInt = TypeInt set_int:3];`, the call `[TypeInt fromObject:anInt result:&r]` sets `r` to 3. Conversely, `r = 4; anInt = [TypeInt toObject:&r];` sets `anInt` to a newly created object of class `TypeInt` whose `_int` field is 4.

Note that the ability to do arithmetic is embodied in `TypeInt`, as is the ability to convert between `TypeInts` and the other integer types (and string type).

Evaluating an expression node in a given environment is done by calling eval:

```
eval(expr, env, type, resultp)
id expr, env, type; void *resultp;
```

The `eval` function takes as argument an expression object, an environment object, a type object, and a pointer to a place to put the result. The eval function takes care of printing out tracing information, if necessary, and then simply sends `expr` the `eval:resultType:result:` message. Each expression class is responsible for knowing how to evaluate itself, and is able to convert its return value into the appropriate type.

The most interesting case is the evaluation of a `MessageExpr`:

```
= MessageExpr:  Expression {
        id      sel;            /* Selector object */
        id      rec;            /* (unevaluated) receiver object */
```

```
        id      arg[MAXARGS];       /* unevaluated arguments */
}

- (void*)eval:env resultType:rt result:(void *)r {
    id v;
    id _rec, _arg[5];
    int i;
    int nargs = [sel nargs];
    SEL _sel = [sel sel];
    id rettype = [sel rettype];

    eval(rec, env, TypeId, &_rec);
    for(i = 0; i < nargs; i++)
        eval(arg[i], env, [sel argtype:i], &_arg[i]);
    v = _msg(_rec, _sel, _arg[0], _arg[1],
             _arg[2], _arg[3], _arg[4]);
    if(rt == rettype) {  /* no need to convert */
        *(id *)r = v;           /* hack, assumes id or equal size */
        return r;
    }
    return [rt fromObject:[rettype toObject:&v] result:r];
}
```

There is some pointer cheating going on here, as the arguments which are to be sent to the receiver object are stored in an array of `id`s, even though they are not necessarily objects. This relies on the fact that, at least on the hardware this code runs upon (a MicroVax II), pointers, long integers, short integers, and characters are all represented as four-byte values when passed to functions.

The `sel` variable is the `Selector` object, and is used to get the number and types of the arguments and the return value of this selector. First `eval` is called recursively to evaluate the receiver of the message; the result type is necessarily `TypeId` since a receiver of a message must be an Objective C object. Each of the argument expressions is evaluated, the result being stored in the `_arg` array. The type of the returned result is that which is expected for this argument in the message about to be sent. The function `_msg` is the low-level message sending function that lies at the heart of Objective C; it is passed a receiver, a selector, and any arguments, and returns the result of sending the message specified by the selector and the arguments to the specified receiver. This result is then converted to the correct type. If this message selector is already known to return the same type as desired, then no conversion is necessary, and the value is simply copied into the correct place. Otherwise, the returned value is first converted to an object (by invoking the `toObject:` method of the known return type) and then converted from an object to the desired return type (via the `fromObject:result:` method). In the typical case, either `rt` or `rettype` is `TypeId`, so one of the conversions to or from an object does no significant work.

The reason for passing the return type to `eval`, rather than having `eval` always return an object, and then converting returned objects to machine integers, characters, and strings when needed, is

efficiency. In the current scheme, nested message expressions, where the inner expression returns, say, an integer which is the expected argument type of the outer expression, there is no overhead converting the intermediate result to an object and then immediately back to an integer.

Note that the automatic conversion to objects allows arithmetic to be done relatively painlessly. For example, to add 10 to the x coordinate of a view, use:

```
[[view xloc] Plus:10]
```

The `[view xloc]` returns a machine integer; since this is the intended receiver of the `Plus:` message it must be converted to a `TypeId`, *i.e.* an object, which in this case will be an instance of `TypeInt`. The `Plus:` method expects its argument to be a machine integer; since the interpreter will represent the constant 10 by a `TypeInt` object, it is converted to a machine integer (by calling `eval` with a result type argument of `TypeInt`). The `Plus:` method is then invoked, and it returns a machine integer, which may or may not be converted to a `TypeInt` object depending on the context in which the above program fragment is used.

The above example could be specified more efficiently in the gesture semantics as `[10 Plus:[view xloc]]`. In this case, all the conversions are avoided, since `10` is already represented as an object of `TypeInt`, and `Plus:` expects a machine integer as argument, which is exactly what is returned by `[view xloc]`.

One thing not shown in the above implementation is garbage collection. During expression evaluation, objects are freely being created and discarded, and it is important that the memory associated with them be released when they are discarded. The current implementation of the interpreter does not do this very well, since there is not much point given the lax attitude toward memory management throughout GRANDMA.

Interface Implementation

All the expression nodes are subclasses of `Model`, and each one has a corresponding subclass of `View` to display it on the screen. The expression views act as virtual tools; these tools act on empty argument and receiver slots, as well as the docks and the trash. Implementing the interpreter interface in GRANDMA was a good exercise of the GRANDMA facilities, but is not especially interesting so will not be covered in detail here.

Control Constructs

The only control construct currently implemented is `Seq`, which allows a list of expressions to be evaluated in order. `Seq`, it turns out, was implemented without any extra mechanism in the interpreter; all that was required was the creation of a `Seq` class, whose class methods simply returned their last argument:

```
= Seq:  Object (GRANDMA, Primitive) { }
+ :a1 { return a1; }
+ :a1:a2 { return a2; }
+ :a1:a2:a3 { return a3; }
+ :a1:a2:a3:a4 { return a4; }
+ :a1:a2:a3:a4:a5 { return a5; }
```

Since arguments are evaluated in order, this has the desired effect.

Other control constructs, such as `While` and `If`, have not been implemented, but could easily be implemented if the need arose. One simple implementation technique would to make `WhileExpr` and `IfExpr` both subclasses of `MessageExpr`, and then make `While` and `If` classes which have methods that have the right number of arguments. For simplicity, the normal message expression display code could be used to display `If` and `While` expressions; the only new code to be added would be new `eval:resultType:result:` methods in `WhileExpr` and `IfExpr` which have the desired effect.

Attributes and Cursors

An important consideration in allowing gesture semantics to be specified at runtime is exactly what the application programmer makes visible to the gesture semantics programmer. There are a number of means by which the application programmer can make a feature available to the semantics programmer; all of these hinge on making visible objects which can be the receivers of relevant messages.

The "Attributes" lists provides a way of giving the semantics writer easy access to application objects and features. This is done by creating expressions for each attribute. GRANDMA already supplies entries for all accessible gesture attributes and features.

As an illustrative example of how attributes are specified and implemented, consider the two attributes `handler` and `enclosed`. The `handler` attribute simply refers to the gesture handler that is currently executing. The `enclosed` attribute refers to the list of `View` objects enclosed by the current gesture. Selecting `enclosed` from the attribute list results in a named message; clicking on its box reveals that the message is `[handler enclosed]`.

Internally,

```
handlerVar = [[VarExpr str:"handler"]
                    vclass:GestureEventHandler];
/* The above statement adds "handler" to the list of attributes to be displayed
   in the interpreter window, and declared that its value is of type GestureEventHandler.
   Its value is actually set by the GestureEventHandler before any gesture
   semantics are evaluated. */

enclosedExpr = [[[[MessageExpr sel:@selector(enclosed)]
                      rec:handlerVar]
                    str:"enclosed"]
                  vclass:OrdCltn];
/* The above statement adds "enclosed" to the attribute list. When evaluated
   in gesture semantics, the "enclosed" attribute will result in
   [handler enclosed] being executed. */
```

Both `handlerVar` and `enclosedExpr` are added to the list of interpreter attributes, and show up in the list as "handler" and "enclosed" respectively. Each of these expressions evaluates to an Objective C object; the `vclass:` message records the expected class of the object. The

recorded class is used by the selector browser as a starting point when choosing a message to send to an attribute.

The "handler" attribute, being a `VarExpr`, is evaluated by looking up the string "handler" in the current environment. Section 7.4 described how the environment in which semantic expressions are evaluated is initialized so as the bind `handler` to the current event handler. Evaluating `enclosed` thus results in the `enclosed` message being sent to the current handler:

```
= GestureEventHandler ...
- enclosed { id o, e, seq; int xmin, ymin, xmax, ymax;
    [gesture xmin:&xmin ymin:&ymin xmax:&xmax ymax:&ymax];
    o = [[wall viewdatabase]
       partiallyInRect:xmin:ymin:xmax:ymax];
    for(seq = [o eachElement]; e = [seq next]; )
        if(!  [e isContainedIn:gesture] ) [o remove:e];
    return o;
}
```

The interpreter's evaluation of the `enclosed` attribute thus results in a call to the above method. This method determines the bounding box of the current gesture, and consults the view database for a list of views contained within this bound. Each object is polled to see if it is enclosed by the gesture, and is removed from the list if it is not. The list is then returned.

The default implementation of `isContainedIn:`, in the `View` class, simply tests if each corner of the bounding box is enclosed within the gesture. This test may be overridden by non-rectangular views, or rectangular views that wish to ensure its each edge is entirely contained within the gesture.

```
= View ...
- (BOOL)isContainedIn:g {
    int x1, y1, x2, y2; [self _calc_new_box];
    x1 = [box left]; y1 = [box top];
    x2 = [box right]; y2 = [box bottom];
    return [g contains:x1:y1] && [g contains:x1:y2] &&
           [g contains:x2:y1] && [g contains:x2:y2];
}
```

The `Gesture` class implements the `contains::` message, which tests if a point is enclosed within the gesture. The current implementation first closes the gesture by conceptually connecting the ending point to the starting point, and then counts the number of times a line from the point to a known point outside the gesture crosses the gesture. An odd number of crossings indicates that the point is indeed enclosed by the gesture.

Other attributes work similarly, although their code tends to be much simpler than that of `enclosed`. In particular, there are attributes for each feature discussed in Section 3.3; the attributes are named messages implemented as `[[handler gesture] ifvi:N]`, where `N` is the corresponding index into the feature vector.

Cursors are added to the list of cursors available for use in semantic expressions simply by sending them the `public` message. The application programmer should create and make available

any cursor that might prove useful to the semantics writer.

7.8 Conclusion

The gesture subsystem of GRANDMA consists of the gesture event handler, the low level gesture recognition modules, the user interface which allows the modification of gesture handlers, gesture examples, and gesture classes, and the interpreter for evaluating the semantics of gestures. Each of these parts has been discussed in detail. The next chapter demonstrates how GRANDMA is used to build gesture-based applications.

Chapter 8

Applications

This chapter discusses three gesture-based applications built by the author. The first, GDP, is a simple drawing editor based on the drawing program DP [42]. The second, GSCORE, is an editor for musical scores. The third, MDP, is an implementation of the GDP drawing editor that uses multi-finger gestures.

GDP and GSCORE are both written in Objective C, and run on a DEC MicroVAX II. They are both gesture-based applications built using the GRANDMA system, discussed in Chapters 6 and 7. As such, the gestures used are all single-path gestures drawn with a mouse. GRANDMA interfaces to the X10 window system [113] through the GDEV interface written by the author. GDEV runs on several different processors (MicroVAX II, SUN-2, IBM PC-RT), and several different window managers (X10, X11, Andrew). GRANDMA, however, only runs on the MicroVax, which for years was the only system available to the author that ran Objective C. It should be relatively straightforward to port GRANDMA to any UNIX-based environment that ran Objective-C, though to date this has not been done.

MDP is written in C (not Objective C), and runs on a Silicon Graphics IRIS 4D Personal Workstation. MDP responds to multiple-finger gestures input via the Sensor Frame. Unlike GDP and GSCORE, MDP is not built on top of GRANDMA. The reason for this is that the only functioning Sensor Frame is attached to the above-mentioned IRIS, for which no Objective C compiler exists. It would be desirable and interesting to integrate Sensor Frame input and multi-path gesture recognition into GRANDMA (see Section 10.2).

8.1 GDP

GDP, a gesture-based drawing program, is based on DP [42]. In DP there is always a *current mode*, which determines the meaning of mouse clicks in the drawing window. Single letter keyboard commands or a popup menu may be used to change the current mode. The current mode is displayed at the bottom of the drawing window, as are the actions of the three mouse buttons. For example, when the current mode is "line", the left mouse button is used for drawing horizontal and vertical lines, the middle button for arbitrary lines, and the right button for lines which have no gravity. Some DP commands cause dialogue boxes to be displayed; this is useful for changing

parameters such as the current thickness to use for lines, the current font to use for text, and so on.

With the gesture handlers turned off, GDP (loosely) emulates DP. The current mode is indicated by the cursor. For example, when the "line" cursor is displayed, clicking a mouse button in the drawing window causes a new line to be created and one endpoint to be fixed at the position of the mouse. As long as the mouse button is held down, the other end of the line follows any subsequent motion of the mouse, in a "rubberband" fashion. The user releases the mouse button when the second endpoint of the line is at the desired location.

Both DP and GDP support *sets*, whereby multiple graphic objects may be grouped together and subsequently function as a single object. Once created, a set is translated, rotated, copied, and deleted as a unit. A set may include one or more sets as components, allowing the hierarchical construction of drawings. In DP, there is the "pack" command, which creates a new set from a group of objects selected by the user, and the "unpack" command, whereby a selected set object is transformed back into its components. GDP functions similarly, though the selection method differs from DP.

GDP makes no attempt to emulate every aspect of DP. In particular, the various treatments of the different mouse buttons are not supported. These and other features were not implemented since doing so would be tangential to the purpose of the author, which was to demonstrate the use of gestures. As the unimplemented features present no conceptual problems for implementation in GRANDMA, the author chose not to expend the effort.

8.1.1 GDP's gestural interface

GDP's gesture-based operation has already been briefly described in Section 1.1. That description will be expanded upon, but not repeated, here.

Figures 1.2a, b, c, and d show the rectangle, ellipse, line, and pack gestures, all of which are directed at the GDP window, rather than at graphic objects. Also in this class is the text gesture, a cursive "t", and the dot gesture, entered by pressing the mouse button with no subsequent mouse motion. The text gesture causes a text cursor to be displayed at the initial point of the gesture. The user may then enter text via the keyboard. The dot gesture causes the last command (as indicated by the current mode) to be repeated. For example, after a delete gesture, a dot gesture over an existing object will cause that object to be deleted.

Figures 1.2e, f, and g show the copy, rotate, and delete gestures, all of which act directly on graphic objects. The move gesture, a simple arrow (figure 8.1), is similar. All of these gestures act upon the graphic object at the initial point of the gesture. These gestures are also recognized by the GDP window when not begun over a graphic object. In this case, the cursor is changed to indicate the corresponding mode, and the underlying DP interface takes over. In particular, dragging one of these cursors over a graphic object causes the corresponding operation to occur.

8.1.2 GDP Implementation

Since GDP was built on top of GRANDMA, the implementation followed the MVC paradigm. Figure 8.2 shows the position in the class hierarchy for the new classes defined in GDP.

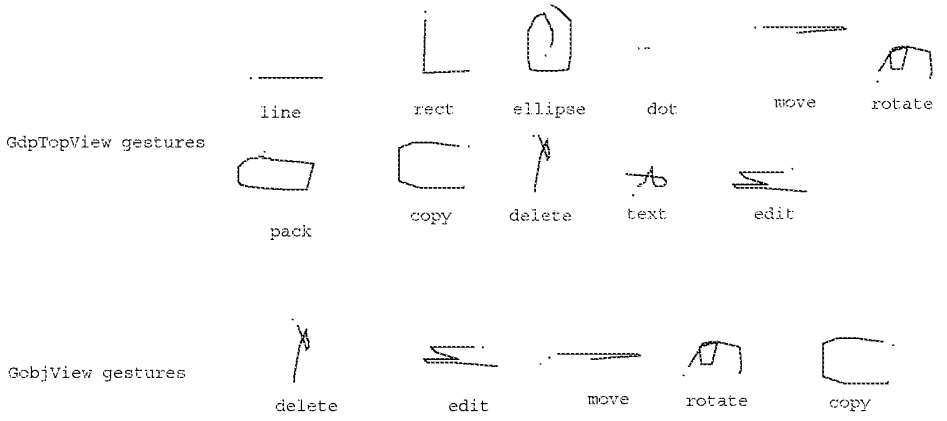


Figure 8.1: GDP gestures

As always, the period indicates the start of the gesture.

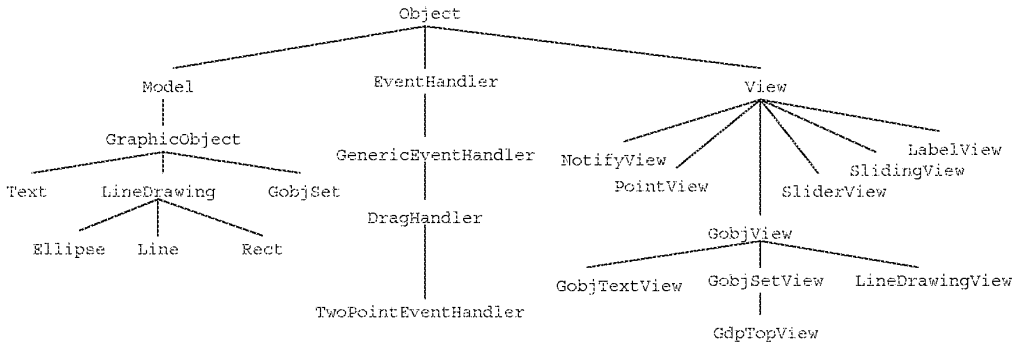


Figure 8.2: GDP's class hierarchy

8.1.3 Models

The implementation of GDP centers on the class `GraphicObject`, a subclass of `Model`. Each component of the drawing is a `GraphicObject`. The entire drawing is also implemented as a graphic object. `GraphicObjects` are either `Text` objects, `LineDrawing` objects (lines, rectangles, and ellipses), or `GobjSet` objects, which implement the set concept.

A `GraphicObject` has two instance variables: `parent`, the `GobjSet` object of which this object is a member, and `trans`, a transformation matrix [101] for mapping the object into the drawing. Every `GraphicObject` is a member of exactly one set, be it the set which represents the entire drawing (these are top level objects), or a member of a set which is itself part of the drawing.

`LineDrawing` objects have a single instance variable, `thickness`, that controls the thickness of the lines used in the line drawing. The three subclasses of `LineDrawing`, namely `Line`, `Rectangle`, and `Ellipse`, represent all graphics in the drawing. Associated with each `LineDrawing` subclass is a list of points which specify a sequence of line segments for drawing the object. The points in the list are normalized so that one significant point of the object lies on the origin and another significant point is at point (1,1). For `Lines`, one endpoint is at (0,0) and the other at (1,1). The point list for `Rectangles` specifies a square with corners at (0,0), (0,1), (1,1), and (1,0). The `Ellipse` is represented by 16 line segments that approximate a circle with center (0,0) and that passes through the point (1,1). The transformation matrix in each `LineDrawing` object is used to map the list of points in each `LineDrawing` object into drawing (window) coordinates.

A `GobjSet` object contains a `Set` of objects that make up the set. In order to display a set, the transformation matrix of the set is composed with (multiplied by) that of each of the constituent objects. This composition happens recursively, so that deeply nested objects are displayed correctly.

`Text` objects contain a font reference and text string to be displayed.

8.1.4 `Views`

Each of the immediate subclasses of `GraphicObject` has a corresponding subclass of `GobjView` associated with it. Each `LineDrawingView` object is responsible for displaying the `LineDrawing` object which is its model on the screen. Similarly, `GobjTextViews` display `Text` objects, and `GobjSetViews` display `GobjSets`.

All `GobjViews` respond to the `updatePicture` message in order to redraw their picture appropriately. A `LineDrawingView` simply asks its model for the lists of points (suitably transformed) which it proceeds to connect via lines. The model also provides the appropriate thickness of the lines as well. (Note that it is not necessary to provide view classes for the three subclasses of `LineDrawing` since all three classes are taken care of by `LineDrawingView`.)

`GobjTextViews` draw their models one character at a time in order to accommodate the transformation of the model. Transformations which have a unit scale factor (no shrinking or dilation) and no rotation component cause the text to be drawn horizontally, with the characters spacing determined by their widths in the current font. In the current implementation, scaling or rotation does not effect the character size or orientations (as X10 will not rotate or scale characters), but does effect the character positions.

`GobjSetViews` have the views of their model's component objects as subviews. Since

the update method for View will automatically propagate update messages to subviews, no updatePicture method is required for GobjSetView.

The GobjView class overrides the move:: method (of View). Recall from Section 6.6 that this method simply changes the location of the view, thus translating the view in two dimensions. This method is used, for example, by the drag handler (section 6.7.9) to cause views to move with the mouse cursor. The purpose of overriding the default method is so that dragging any GobjView causes its model to be changed so as to reflect the new coordinates of the object in the drawing. The model is changed by first sending it the message getLocalTrans, which returns the model's transformation matrix, then calling a function which modifies the matrix to reflect the additional translation, and then sending the model a setLocalTrans: message, which causes the new transformation matrix to be recorded in the model. Of course the model then sends itself the modified message which causes the model's view to redraw the model at its new location.

GobjView also implements the delete message, by first sending itself the free message (which, among other things, removes it from its parent's subview list), and then sending its model the delete message. GobjView also overrides the default isOver: and isContainedIn: methods (Sections 6.7.5 and 7.7.3) so that they always return NO for objects not at the top-level of the drawing. Each subclass of GobjView implements isReallyOver: and isReallyContainedIn:, which are invoked when the object is indeed top-level.

The outermost window is itself a view. It is an instance of GdpTopView, which is a subclass of GdpSetView. The GdpTopView representing the entire drawing.

8.1.5 Event Handlers

GDP required the addition of one new event handler, TwoPointEventHandler, which is of sufficient utility and generality to be incorporated into the standard set of GRANDMA event handlers. The purpose of the TwoPointEventHandler is to implement the typical "rubberbanding" interaction. For example, clicking the "line" cursor in the drawing window causes a new line to be created, one endpoint of which is constrained to be at the location of the click, the other endpoint of which stays attached to the cursor until the mouse button is released. A TwoPointEventHandler can be used to produce this behavior.

As a GenericEventHandler, a TwoPointEventHandler has a parameterizable starting predicate, handling predicate, and stopping predicate (Section 6.7.8). In order for a passive TwoPointEventHandler to be activated, the tool of the activating event must operate on the view to which the handler is attached (like a GenericToolOnViewHandler, section 6.7.7). If the tool operates on the view and the event satisfies the starting predicate, the handler is activated. When activated, the tool is allowed to operate on the view, and the operation is expected to return an object which is to be the receiver of subsequent messages. In the above example, the "line" tool operates upon the drawing window view (a GdpTopView) the result of which is a newly created Line object. The handler then sends the new object a message whose parameters are the starting event location coordinates. The actual message sent is a parameter to the passive event handler; in the example the message is setEndpoint0::. Each subsequent event handled results in the new object being sent another message containing the coordinates of the event (setEndpoint1:: in the example).

8.1.6 Gestures in GDP

This section describes the addition of gestures to the implementation described above. The gesture handlers, gesture classes, example gestures, and gesture semantics were all added at runtime, allowing them to be tested immediately. I should admit that in several cases it was necessary to add some features directly to the existing C code and recompile. This was partly due to the fact that GRANDMA's gesture subsystem was being developed at the same time as this application, and partly due to the gesture semantics wanting to access models and views through methods other than ones already provided, for reasons such as readability and efficiency.

Figure 8.1 shows the gesture classes recognized by each of the two GDP gesture handlers. Note that the gestures expected by a `GobjView` are a subset of those expected by a `GdpTopView`. Allowing one gesture class to be recognized by multiple handlers allows the semantics of the gesture to depend upon the view at which it is directed.

Several gestures (line, rect, ellipse, and text) cause graphic objects to be created. These gestures are only recognized by the top level view, which covers the entire window, a `GdpTopView`. When, for example, a line gesture (a straight stroke) is made, a line is created, the first endpoint of which is at the gesture start, while the second endpoint tracks the mouse in a rubberband fashion.

The semantics for the line gesture are:

```
recog = [Seq :[handler mousetool:createLine_MouseTool]
             :[[topview createLine] translateEndpoint:0
                                    x:<startX> y:<startY>] ];
manip = [recog scaleXYEndpoint:1 x:<currentX> y:<currentY>
                                 cx:<startX>  cy:<startY>];
```

(The `done` expression is assumed to be `nil`.) When the line gesture is recognized, the gesture handler is sent the `mousetool:` message, passing the `createLine_MouseTool` as a parameter. The handler sends a message to its view's wall, and the cursor shape changes. (Internally, the handler changes its `tool` instance variable to the new tool, as well.) Then, a line is created (via the `createLine` message sent to the top view), and the new line is sent a message which translates one endpoint to the starting point of the gesture. (The identifiers enclosed in angle brackets are gestural attributes, as discussed in Section 7.7.3.) The `::` message to `Seq`, which is used evaluate two expressions sequentially, returns its last parameter, in this case the newly created line, which is assigned to `recog`.

Upon each subsequent mouse input the `manip` expression is evaluated. It sends the new line (referred to through `recog`) a message to scale itself, keeping the "center" point (`startX`, `startY`) in the same location, mapping the other endpoint to (`currentX`, `currentY`).

The semantics for the rect and ellipse gestures are similar to those of line, the only difference being the resultant cursor shape and the creation message sent to `topview`. The start of the rectangle gesture controls one corner of the rectangle and subsequent mouse events control the other corner. The start of the ellipse gesture determines the center of the ellipse, and the scaling guarantees that the mouse manipulates a point on the ellipse. The rectangle is created so that its sides are parallel to the window. Similarly, the ellipse is created so that its axes are horizontal and vertical. Manipulations after any of the creation gestures is recognized never effect the orientation of the created object. With only a single mouse position for continuous control (two degrees of

freedom) it is impossible to independently alter the orientation angle, size, and aspect ratio of the graphic object. The design choice was made to modify only the size and aspect ratio in the creation gesture; a rotate gesture may subsequently be used to modify the orientation angle.

It is still possible, however, to use other features of the gesture to control additional attributes of the graphic object. Changing the `recog` semantics of a line gesture to

```
recog = [Seq :[handler mousetool:<createLine>]
             :[[[topview createLine] translateEndpoint:0
                                        x:<startX> y:<startY>]
               thickness:[[pathLength DividedBy:40]
                           Clip:1 :9] ]  ];
```

causes the thickness of the line to be the length of the gesture divided by 40 and constrained to be between 1 and 9 (pixels) inclusive. The length of the gesture determines the thickness of the newly created line, which can subsequently be continuously manipulated into any length.

The dot gesture (where the user simply presses the mouse without moving it) has the null semantics. When it is recognized, the gesture handler turns itself off immediately, enabling events to propagate past it, and thus allowing whatever cursor is being displayed to be used as a tool. Thus GDP, like DP, has the notion of a current mode, accessible via the dot gesture.

The pack gesture has semantics:

```
recog = [Seq :[handler mousetool:pack_MouseTool]
             :[topview pack_list:<enclosed>]];
```

The attribute `<enclosed>` is an alias for `[handler enclosed]`. Recall from Section 7.7.3 that this message returns a list of objects enclosed by the gesture. This list is passed to the `topview`, which creates the set. As long as the mouse button is held down, the pack tool will cause the `pack` message to be sent to any object it touches; those objects will execute `[parent pack:self]` (the implementation of the `pack` method) to add themselves to the current set.

The copy, move, rotate, edit, and delete gestures simply bring up their corresponding cursors when aimed at the background (`GdpTopView`) view. They have more interesting semantics when associated with a `GobjView`. The copy gesture, for example, causes:

```
recog = [Seq :[handler mousetool:viewcopy_MouseTool]
             :copy = [[view viewcopy]
                        move:<endX> :<endY>]
             :lastX = <endX>
             :lastY = <endY>]
manip = [Seq :[copy move:[<currentX> Minus:lastX]
                      :[<currentY> Minus:lastY]]
             :lastX = <endX>
             :lastY = <endY>]
```

This illustrates that the gesture semantics can mimic the essential features of the `DragHandler` (Section 6.7.9). The semantics of the move gesture are almost identical, except that no copy is made. A simpler way to do this kind of thing (by reraising events) is shown when the semantics of the GSCORE program are discussed.

The delete gesture has semantics

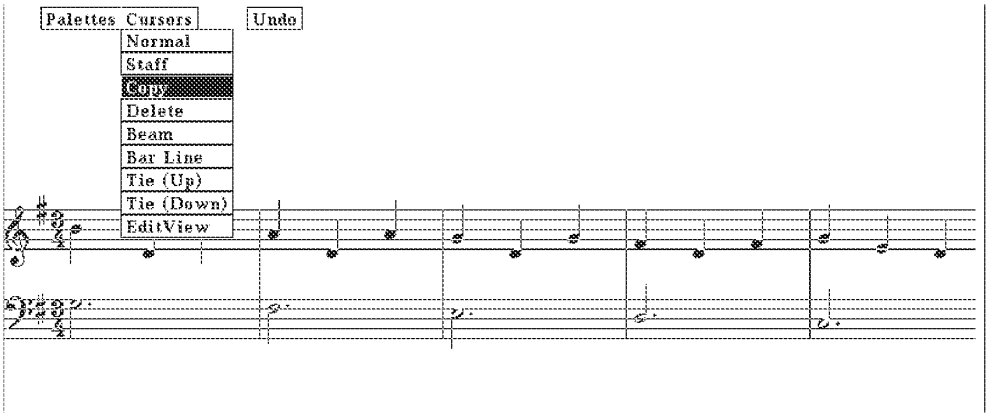


Figure 8.3: GSCORE's cursor menu

```
recog = [Seq :[handler mousetool:delete_Mousetool]
             :[view delete]];
```

The edit gesture semantics are similar.

The rotate gesture has semantics:

```
recog = nil;
manip = [Seq :[handler mousetool:rotate_MouseTool]
             :[view rotateAndScaleEndpoint:0
                    x:<currentX>
                    y:<currentY>
                    cx:<startX>
                    cy:<startY>]];
```

The `rotateAndScaleEndpoint:` message causes one point of the view to be mapped to the coordinate indicated by `x:` and `y:` which keeping the point indicated by `cx:` and `cy:` constant. This gesture always drags endpoint 0 of a graphic object. It would be better to be able to drag an arbitrary point, as is done by MDP, discussed later.

8.2 GSCORE

GSCORE is a gesture-based musical score editor. Its design is not based on any particular program, but its gesture set was influenced by the SSSP score-editing tools [18] and the Notewriter II score editor.

8.2.1 A brief description of the interface

GSCORE has two interfaces, one gesture-based, the other not. Figure 8.3 shows the non-gesture-based interface in action. Initially, a staff (the five lines) is presented to the user. The user may call

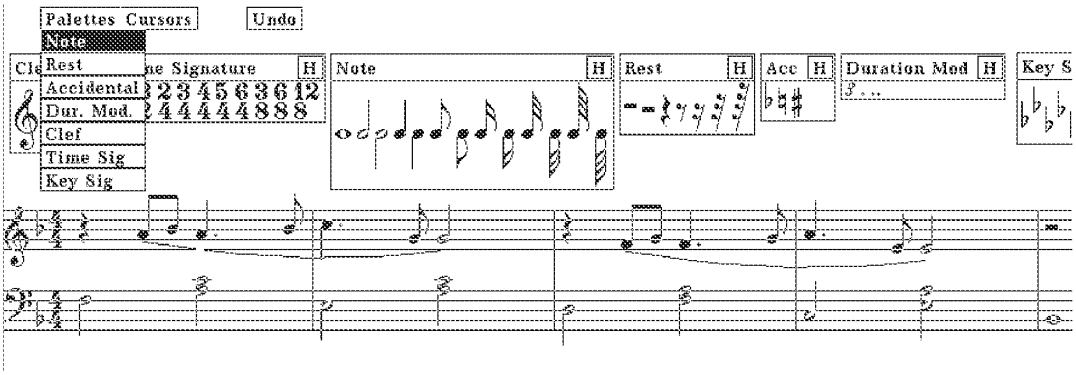


Figure 8.4: GSCORE's palette menu

up additional staves by accessing the staff tool in the "Cursors" menu (which is shown in the figure). In figure 8.4, the user has displayed a number of palettes from which he can drag musical symbols onto the staff. As can be seen, the user has already placed a number of symbols on the staff. The user has also used the down-tie tool to indicate two phrases and the beam tool to add beams so as to connect some notes.[1] Both tools work by clicking the mouse on a starting note, then touching other notes. The tie tool adds a tie between the initial note and the last one touched, while the beam tool beams together all the notes touched during the interaction.

Dragging a note onto the staff determines its starting time as follows: If a note is dragged to approximately the same x location as another note, the two are made to start at the same time (and are made into a chord). Otherwise, the note begins at the ending time of the note (or rest or barline) just before it. Other score objects are positioned like notes.

The palettes are accessed via the palette menu, shown in figure 8.4. The palettes themselves may be dragged around so as to be convenient for the user. The "H" button hides the palette; once hidden it must be retrieved from the menu.

The delete cursor deletes score events. When the mouse button is pressed, dragging the delete button over objects which may be deleted causes them to be highlighted. Releasing the button over such a highlighted object causes it to be deleted. Individual chord notes may be deleted by clicking on their note heads; an entire chord by clicking on its stem. When a beam is deleted, the notes revert to their unbeamed state.

The gestural interface provides an alternative to the palette interface. Figure 8.5 shows the three sets of gestures recognized by GSCORE objects. The largest set, associated with the staff, all result

[1]Note to readers unfamiliar with common music notation: A tie is a curved line connecting two adjacent notes of the same pitch. A tie indicates that the two connected notes are to be performed as a single note whose duration equals the sum of those of the connected notes. A curved line between adjacent differently pitched notes is a slur, performed by connecting the second note to the first with no intermediate breath or break. Between nonadjacent notes, the curved line is a phrase mark, which indicates a group of notes that makes up a musical phrase, as shown in figure 8.4. In GSCORE, the tie tool can be used to enter ties, slurs, and phrase marks. A beam is a thick line that connects the stems of adjacent notes (again see figure 8.4). By grouping multiple short notes together, beams serve to emphasize the metrical (rhythmic) structure of the music.

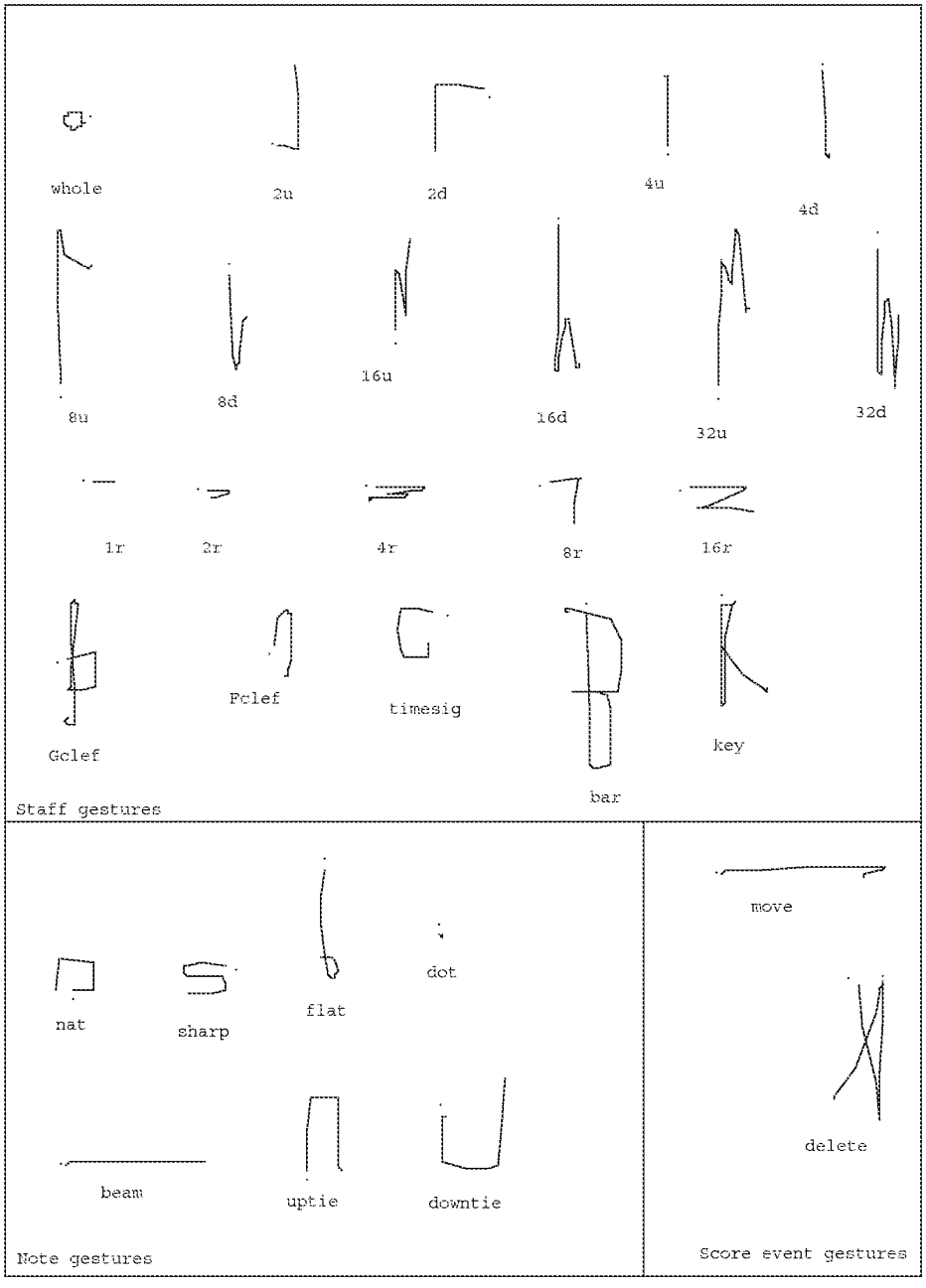


Figure 8.5: GSCORE gestures

in staff events being created. There are two gestures, move and delete, that operate upon existing score events. Seven additional gestures are for manipulating notes.

A gesture at a staff creates either a note, rest, clef, bar line, time signature, or key signature object. The object created will be placed on the staff at (or near) the initial point of the gesture. For notes, the x coordinate determines the starting time while the y coordinate determines the pitch class. The gesture class determines the actual note duration (whole note, half note, quarter note, eight note, sixteenth note, or thirtysecond note) and the direction of the stem.

Like note gestures, the remaining staff gestures use the initial x coordinate to determine the staff position of the created object. The five rest gestures generate rests of various durations. The two clef gestures generate the F and G clefs (C clefs may only be dragged from the palette). The timesig gesture generates a time signature. After the gesture is recognized, the user controls the numerator of the time signature by changes in the x coordinate of the mouse, and the denominator by changes in y. Similarly, after the key gesture is recognized, the user controls the number of sharps or flats by moving the mouse up or down. When a bar gesture is recognized, a bar line is placed in the staff, and the cursor changes to the bar cursor. While the mouse button is held, the newly created bar line extends to any staff touched by the mouse cursor.

The note-specific gestures all manipulate notes. Accidentals are placed on the note using the sharp, flat, and natural gestures. The beam gesture causes the notes to be beamed together. The note on which the beam gesture begins is one of the beamed notes; the beam is extended to other notes as they are touched after the gesture is recognized. The uptie and downtie gestures operate similarly. The dot gesture causes the duration of the note to be multipled by $\frac{3}{2}$, typically resulting in a dot being added to a note.

Since a note is a score event, and always exists on a staff, a gesture which begins on a note may either be note specific (*e.g.* sharp), score-event specific (*e.g.* delete), or directed at the staff (*e.g.* one of the note gestures). The first time a gesture is made at a note, the three gesture sets are unioned and a classifier created that can discriminate between each of them, as described in Section 7.2.

Figure 8.6 shows an example session with GSCORE.

8.2.2 Design and implementation

Figure 8.7 shows where the classes defined by GSCORE fit into GRANDMA's class hierarchy. In general, each model class created has a corresponding view class for displaying it. No new event handlers needed to be created for GSCORE; GRANDMA's existing ones proved adequate.

Generally useful views

Two new views of general utility, `PullDownRowView` and `PaletteView`, were implemented during the development of GSCORE. A `PullDownRowView` is a row of buttons, each of which activates a popup menu. It provides functionality similar to the Macintosh menu bar. A `PaletteView` implements a palette of objects, each of which is copied when dragged. `PaletteView` instantiates a single `DragHandler` (Section 6.7.9) that it associates with every object on a palette. The drag handler has been sent the message `copyviewON`, which gives the palette its functionality.

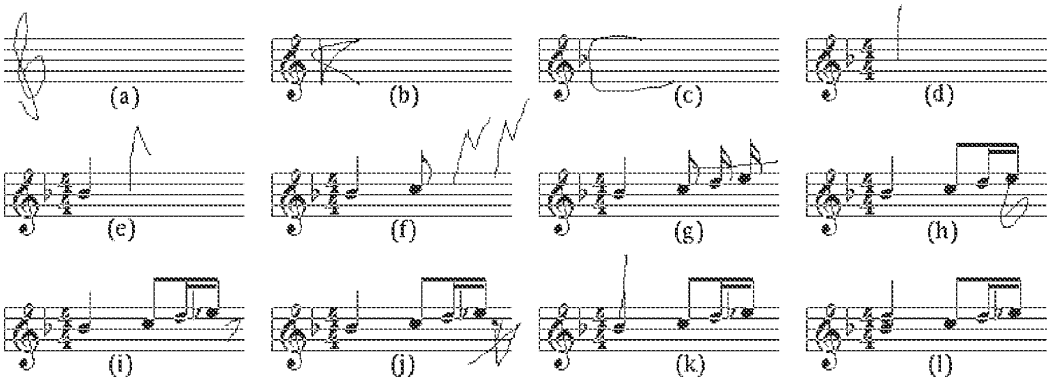


Figure 8.6: A GSCORE session

Panel (a) shows a blank staff upon which the Gclef *gesture has been entered. Panel (b) show the created treble clef, and a* key *(key signature) gesture. After recognition, the number of flats or sharps can be manipulated by the distance the mouse moves above the staff or below the staff, respectively. Panel (c) shows the created key signature (one flat), and a* timesig *(time signature) gesture. After recognition, the horizontal distance from the recognition point determines the numerator of the time signature, and the vertical distance determines the denominator. Panel (d) shows the resulting time signature, and the* 4u *(quarter note) gesture, a single vertical stroke. Since this is an upstroke, the note will have an upward stem. The initial point of the gesture determines both the pitch of the note (via vertical position) and the starting time of the note (via horizontal position). Panel (e) shows the created note, and the* 8u *(eighth note) gesture. Like the quarter note gesture, the gesture class determines the note's duration, and gestural attributes determines the note's stem direction, start time and pitch. Panel (f) shows two* 16u *(sixteenth note) gestures (combining two steps into one). Panel (g) shows a* beam *gesture. This gesture begins on a note, rather than the gestures mentioned thus far, which begin on a staff. After the gesture is recognized, the user touches other notes in order to beam them together. Panel (h) shows the beamed notes, and a* flat *gesture drawn on a note. Panel (i) shows the resulting flat sign added before the note, and an* 8r *(eighth rest) gesture drawn on the staff. Panel (j) shows the resulting rest, and a* delete *gesture beginning on the rest. Panel (k) shows a* 4u *(quarter note) gesture drawn over an existing quarter note (all symbols in GSCORE have rectangular input regions), the result being a chord, as shown in panel (l).*

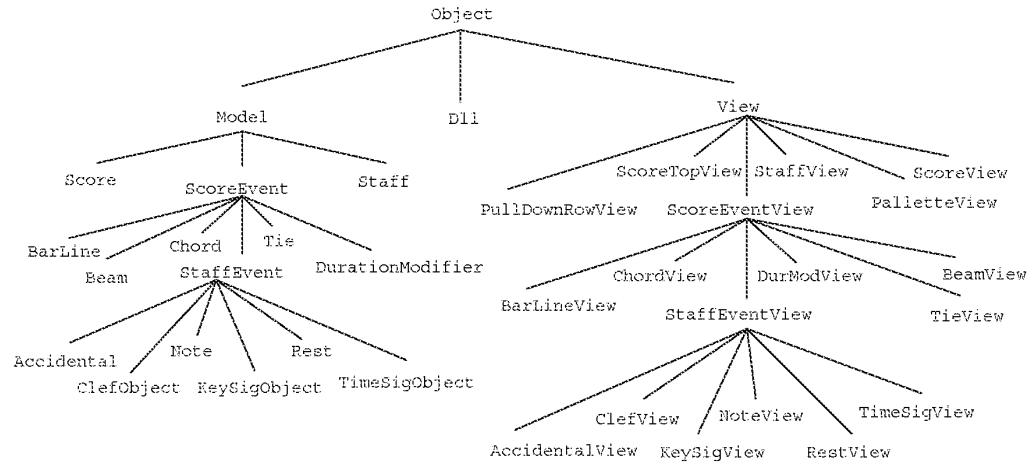


Figure 8.7: GSCORE's class hierarchy

Each palette can implement an arbitrary action when one of the dragged objects is dropped. For most palettes of score events (notes, rests, clefs, and so on), no special action is taken. The copied view becomes a subview of a `StaffView` when dragged onto a staff. However, accidentals and duration modifiers (dots and triplets) are tools which send messages to `NoteView` objects when dragged over them; the `NoteView` takes care of updating its state and creating any accidentals or duration modifiers it needs. The copies that are dragged from the palette thus never become part of the score, and so are automatically deleted when dropped.

GSCORE Models

With the exception of `PullDownRowView` and `PaletteView`, the new classes created during the implementation of GSCORE are specific to score editing. A `Score` object represents a musical score. It contains a list of `Staff` objects and a doubly-linked list (class `Dll`) of `ScoreEvent` objects. Each `ScoreEvent` has a `time` field indicating where in the score it begins; the doubly-linked list is maintained in time order.

The subclass `StaffEvent` includes all classes that can only be associated with a single staff. A `BarLine` is not a `StaffEvent` since it may connect more than one staff, and thus maintains a `Set` of staves in an instance variable. Similarly, a `Chord` may contain notes from different staves, as may a `Tie` and `Beam`. A `DurationModifier` is not attached directly to a `Staff`, but instead with a `Note` or `Beam`, so it is not a `StaffEvent` either.

The responsibility of mapping time to x coordinate in a staff rests mainly with the `Score` object. It has two methods `timeOf:` and `xposOf:` which map x coordinates to times, and times to x coordinates, respectively. `Score` has the method `addEvent:` for adding events to the list and `delete:` and `erase:` for deleting and erasing events. Erase is a kind of "soft" delete; the object is removed from the list of score events, but it is not deallocated or in any other way disturbed. A

typical use would be to erase an object, change its `time` field, and then add it to the score, thus moving it in time.

Each `ScoreEvent` subclass implements the `tiebreaker` message; this orders score events that occur simultaneously. This is important for determining the position of score events; bar lines must come before clefs, which must come before key signatures, and so on. Besides determining the order events will appear on the staff, tiebreakers are important because they maintain a canonical ordering of score events which can be relied upon throughout the code.

Particular `ScoreEvent` classes have straightforward implementations. `Note` has instance variables that contain its pitch, raw duration (excluding duration modifiers), actual duration, stem direction, back pointers to any `Chord` or `Beam` that contain it, and pointers to `Accidental` and `DurationModifier` objects that apply to it. It has messages for setting most of those, and maintains consistency between dependent variables. Notes are able to delete themselves gracefully, first by removing themselves from any beams or chords in which they participate, and deleting any accidentals or duration modifiers attached to them, then finally deleting themselves from the score. Other score events behave similarly.

Sending a `ScoreEvent` the `time:` message, which changes its start time, results in its `Score` being informed. The score takes care to move the `ScoreEvent` to the correct place in its list of events. This is accomplished by first erasing the event from the score, and then adding it again.

While the internal representation of scores for use in editing is quite an interesting topic in its own right [20, 83, 88, 29] it is tangential to the main topic, gesture-based systems. The representation has now been described in enough detail so that the implementation of the user interface, as well as the gesture semantics, can be appreciated. These are now described.

GSCORE Views

As expected from the MVC paradigm, there is a `View` subclass corresponding to each of the `Models` discussed above. `ScoreView` provides a backdrop. Not surprisingly, instances of `StaffView` are subviews of `ScoreView`. Perhaps more surprisingly, all `ScoreEventView` objects are also subviews of `ScoreView`. For simplicity, the various `StaffEventView` classes are not subviews of the `StaffView` upon which they are drawn. This simplifies screen update, since the `ScoreView` need not traverse a nested structure to search for objects that need updating.

It is often necessary for a view to access related views; for example a `BeamView` needs to communicate with the `NoteView` or `ChordView` objects being beamed together. One alternative is for the views to keeps pointers to the related views in instance variables. This is very common in MVC-based systems: pointers between views explicitly mimic relations between the corresponding models. It is the task of the programmer to keep these pointers consistent as the model objects are added, deleted, or modified.

In one sense, this is one of the costs associated with the MVC paradigm. For reasons of modularity, MVC dictates that views and models be separate, and that models make no reference to their views (except indirectly, through a model's list of dependents). The benefit is that models may be written cleanly, and each may have multiple views. Unfortunately, the separation results in redundancy at best (since the structure is maintained as both pointers between models and pointers between views), and inconsistency at worse (since the two structures can get "out of sync"). Also,

any changes to a model's relationship to other models requires parallel changes in the corresponding views. This duplication, noticed during the initial construction of GSCORE, seemed to be contrary to the ideals of object-oriented programming, where techniques such as inheritance are utilized to avoid duplication of effort.

GRANDMA attempts to address this problem of MVC in a general way. The problem is caused by the taboo which prevents a model from explicitly referencing its view(s). GRANDMA maintains this taboo, but provides a mechanism for inquiring as to the view of a given model. In order to retain the possibility of multiple views of a single model, the query is sent to a *context object*; within the context, a model has at most one view. The implementation requires that a context be a kind of `View` object:

```
View ...
- setModelOfView:v { /* associates v with [v model] */ }
- getViewOfModel:m { /* returns view associated with m */ }
```

The implementation is done using an association list per context: given a context, the message `setModelOfView:` associates a view with its model in the context. Objective C's association list object uses hashing internally, so `getViewOfModel:` typically operates in constant time independent of the number of associations. The result is a kind of inverted index, mapping models to views.

In GSCORE, only a single context is used (since there is only one view per model), which, for convenience, is the parent of all `ScoreEventView` objects, a `ScoreView`. The various subclasses of `ScoreEventView` no longer have to keep consistent a set of pointers to related objects. For example, a `BeamView` needs only to query its model for the list of `Note` and/or `Chord` models that it is to beam together; it can then ask each of those models `m` for its view via `[parent getViewOfModel:m]`. The instance variable `parent` here refers to the `ScoreView` of which the `BeamView` is a subview. Thus, the problem of keeping parallel structures consistent is eliminated. One drawback, however, is that it is now necessary to maintain the inverted index as views are created and deleted.

Now that the problem of how views access their related views has been solved, redisplaying a view is straightforward. Recall (Section 6.5) that when a model is modified, it sends itself the `modified` message, which results in all its dependents (in particular its view) getting the message `modelModified`. The default implementation of `modelModified` results in `updatePicture` being sent to the view and all of its subviews (Section 6.6). Normally, `updatePicture` is the method that is directly responsible for querying the model and updating the graphics. `ScoreEventView` overrides `updatePicture`, and the task of actually producing the graphics for a score event is relegated to a new method, `createPicture`, implemented by each of `ScoreEventView`'s subclasses. `ScoreEventView`'s `updatePicture` sends itself `createPicture`, but also does some additional work to be discussed shortly.

As an example, consider what happens when the pitch of a note is changed. When a `Note` is sent the `abspitch:` message, which changes its pitch, it updates its internal state and sends itself the `modified` message. (Changing the pitch might result in `Accidental` objects being added or deleted from the score, a possibility ignored for now.) This `Note`'s `NoteView` will get sent `createPicture`, and query its model (and the `Score` and `Staff` objects of the model) to

determine the kind and position of the note head, as well as the stem direction, if needed. The proper note head is selected from the music font, and drawn on the staff (with ledger lines if necessary) at the determined location.

One reason for `ScoreEvent`'s `updatePicture` sending `createPicture` is to test in a single place the possibility that the view may have moved since the last time it was drawn. In particular, if the *x* coordinate of the right edge of the view's bounding box has changed, this is an indication that the score events after the this might have to be repositioned. If so, the `Score` object is sent a message to this effect, and takes care of changing the *x* position of any affected models. Another reason for the extra step in creating pictures is to stop a recursive message that attempts to create a picture currently being created, a possibility in certain cases.

Adding or deleting a `ScoreEvent` causes the `Score` object to send itself the `modified` message. Before doing so, it creates a record indicating exactly what was changed. When notified, its `ScoreView` object will request that record, creating or deleting `ScoreEventViews` as required. `ScoreView` uses an association list to associate view classes with model classes; it can thus send the `createViewOf:` message to the appropriate factory.

`ScoreEventViews` function as virtual tools, performing the action `scoreeventview:`. (This default is overridden by `AccView`, `DurModView`, `BarLineView`, and `TieView`, as these do not operate on `StaffViews`.) The only class that handles `scoreeventview:` messages is `StaffView`. A version of `GenericToolOnViewEventHandler` different than the one discussed in Section 6.7.7 is associated with class `ScoreEventView`. This version is a kind of `GenericEventHandler`, and thus more parameterizable than the one discussed earlier. The instance associated with `StaffViews` has its parameters set so that it performs its operation immediately (as soon as a tool is dragged over a view which accepts its action), rather than the normal behavior of providing immediate semantic feedback and performing the action when the tool is dropped on the view.

Thus, when a `ScoreEventView` whose action is `scoreeventview:` is dragged over a `StaffView`, the `StaffView` immediately gets sent the message `scoreeventview:`, with the tool (*i.e.* the `ScoreEventView`) as a parameter. The first step is to `erase:` the model of the `ScoreEventView` from the score, if possible. The `StaffView` then sends its model's `Score` the `timeOf:` message, with parameter the *x* coordinate of the `StaffEventView` being dragged. The time returned is made the time of the `ScoreEventView`'s model, which is then added to the score. When a subsequent drag event of the `ScoreEventView` results in the `scoreeventview:` message to be sent to the `StaffView`, the process is repeated again. Thus as the user drags around the `ScoreEventView`, the score is continuously updated, and the effect of the drag immediately reflected on the display.

Though they have different actions, `AccView`, `TieView`, `DurModView`, and `BarLineView` tools operate similarly to the other `ScoreEventViews`. Rather than explain their functionality in the non-gesture-based interface, the next section discusses the semantics of the gestural interface to GSCORE.

GSCORE's gesture semantics

The gesture semantics rely heavily on the palette interface described above. When the palettes are first created, every view placed in the palette is named and made accessible via the "Attributes" button in the gesture semantics window (see Sections 7.7.2 and 7.7.3). It is then a simple matter in the gesture semantics to simulate dragging a copy of the view onto the staff (see Section 7.7.1). For example, consider the semantics of the 8u gesture, which creates an eighth note with an up stem:

```
recog = [[[noteview8up viewcopy] at:<startLoc>]
              reRaise:<currentEvent>];
```

The name `noteview8up` refers to the view of the eighth note with the up stem placed in the palette during program initialization. That view is copied (which results in the model being copied as well), moved to the starting location of the gesture (another "Attribute"), and the `currentEvent` (another "Attribute") is reraised using this view as the tool and its location as the event location. This simulates the actions of the `DragHandler`, and since `startLoc` is guaranteed to be over the staff (otherwise these semantics would never have been executed) the effect is to place an eighth note into the score. Similar semantics (the only difference is the view being copied) are used for all other note gestures, as well as all rest gestures and clef gestures.

The semantics of the bar gesture is similar to that of the note gestures, the difference being that a mouse tool is used rather than a virtual (view) tool.

```
recog = [handler mousetool:
              [barlineEvent_MouseTool
                    reRaise:<currentEvent>
                    at:<startLoc>]];
```

The timesig gesture for creating time signatures is more interesting. After it is recognized, x and y of the mouse control the numerator and the denominator of the time signature, respectively:

```
recog = [Seq :sx = <currentX>
             :sy = <currentY>
             :[[[timesigview4_4 viewcopy] at:<startLoc>]
                    reRaise:<currentEvent>]]
manip = [[recog model]
             timesig:[[[<currentX> Minus:sx]
                           DividedBy:10] Clip :1 :100]
                    :[[[<currentY> Minus:sy]
                         DividedBy:10] Clip :1 :100]]
```

Note that the `recog` expression is similar to the others; a view from the palette is copied, moved to the staff, and used as a tool in the reraising of an event. The `manip` expression, in contrast, does not operate on the level of simulated drags. Instead, it accesses the model of the newly created `TimeSigView` directly, sending it the `timesig::` message which sets its numerator and denominator. The division by 10 means that the mouse has to move 10 pixels in order to change one unit. The `Clip::` message ensures the result will be between 1 and 100, inclusive. For musical purposes, it is probably better to only use powers of two for the denominator, but unfortunately no `toThe:` message has been implemented in `TypeInt` (though it would be simple to do).

The key signature gesture (key) works similarly, except that only the *y* coordinate of the mouse is used (to control the number of accidentals in the key signature):

```
recog = [Seq :sy = <currentY>
             :[[[keysigview1sharps viewcopy]
                   at:<startLoc>]
                     reRaise:<currentEvent>]]
manip = [[recog model]
             keysig:[[[sy Minus:<currentY>]
                           DividedBy:10] Clip:[0 Minus:6] :6]]
```

A positive value for key signature indicates the number of sharps, a negative one the (negation of the) number of flats. The awkward `[0 Minus:6]` is used because the author failed to allow the creation of negative numbers with the "new int" button.

The above gestures are recognized when made on the staff. The delete and move gestures are only recognized when they begin on `ScoreEventViews`. The semantics of the delete gesture are:

```
recog = [Seq :[handler mousetool:delete_MouseTool]
             :[view delete]];
```

This changes the cursor, and deletes the view that the gesture began on. The latter effect could also have been achieved using `reRaise:`, but the above code is simpler.

The move gesture simply restores the normal cursor and reraises it at the starting location of the gesture, relying on the fact that in the non-gesture-based interface, score events may be dragged with the mouse:

```
recog = [[handler mousetool:normal_MouseTool]
            reRaise:startEvent];
```

In addition to the gestures that apply to any `ScoreEventView`, `NoteView` recognizes a few of its own. The three gestures for adding accidentals to notes (sharp, flat, and natural) access the `Note` object directly. For example, the semantics of the sharp gesture are:

```
recog = [[view model] acc:SHARP];
```

The beam gesture changes the cursor to the beam cursor and simulates clicking the beam cursor on the `NoteView` at the initial point:

```
recog = [[handler mousetool:beamtool_MouseTool]
           reRaise:startEvent];
```

The tie gestures (uptie and downtie) could have been implemented similarly. Instead, a variation of the above semantics causes the mouse cursor to revert to the normal cursor when the mouse button is released after the gesture is over:

```
recog = [Seq :[handler mousetool:tieUpEvent_MouseTool]
             :[tieUpEvent_MouseTool reRaise:startEvent]]];

manip = :[tieUpEvent_MouseTool reRaise:currentEvent]]];

done = [Seq :[tieUpEvent_MouseTool reRaise:currentEvent]
            :[handler mousetool:normal_MouseTool]];
```

The dot gesture accesses the `Note`'s raw duration, multiplies it by $\frac{3}{2}$ and changes the duration to the result. The note will add the appropriate dot in the score when it receives its new duration

```
recog = [Seq :m = [view model]
              :[m dur:[[[m rawdur] Times:3] DividedBy:2]]];
manip = recog;
```

The `manip = recog` statement itself does nothing of itself, but by virtue of it being non-nil, the gesture handler does not relinquish control until the mouse button is released. Without this statement, the mouse cursor tool (whatever it happens to be) would operate on any view it was dragged across after the dot gesture was recognized.

8.3 MDP

MDP is gesture-based drawing program that takes multi-finger Sensor Frame gestures as input. Though primarily a demonstration of multi-path gesture recognition, MDP also shows how gestures can be incorporated cheaply and quickly into a non-object-oriented system. This is in contrast to GRANDMA, which, whatever its merits, requires a great deal of mechanism (an object=oriented user interface toolkit with appropriate hooks) before gestures can be incorporated.

The user interface to MDP is similar to that of GDP. The user makes gestures, which results in various geometric objects being created and manipulated. The main differences are due to the different input devices. In addition to classifying multiple finger gestures, MDP uses multiple fingers in the manipulation phase. This allows, for example, a graphic object to be rotated, translated, and scaled simultaneously.

Figure 8.8 shows an example MDP session. Note that how, once a gesture has been recognized, additional fingers may be brought in and out of the picture to manipulate various parameters. Multiple finger tracking imbues the two-phase interaction with even more power than the single-path two-phase interaction.

8.3.1 Internals

Figure 8.9 shows the internal architecture of MDP. The lines indicate the main data flow paths through the various modules.

Like the gesture-based systems built using GRANDMA, when MDP is first started, a set of gesture training examples is read from a file. These are used to train the multi-path classifier as described in Chapter 5. MDP itself provides no facility for creating or modifying the training examples. Instead, a separate program is used for this purpose.

The Sensor Frame is not integrated with the window manager on the IRIS, making the handling of its input more difficult than the handling of mouse input. In particular, coordinates returned by the Sensor Frame are absolute screen coordinates in an arbitrary scale, while the window manager generally expects window-relative coordinates to be used. Fortunately, the IRIS windowing system supports general coordinate transformations on a per-window basis, which MDP uses as follows.

When started, MDP creates a window on the screen, and reads an *alignment file* to determine the coordinate transformation for mapping window coordinates to screen coordinates that makes the

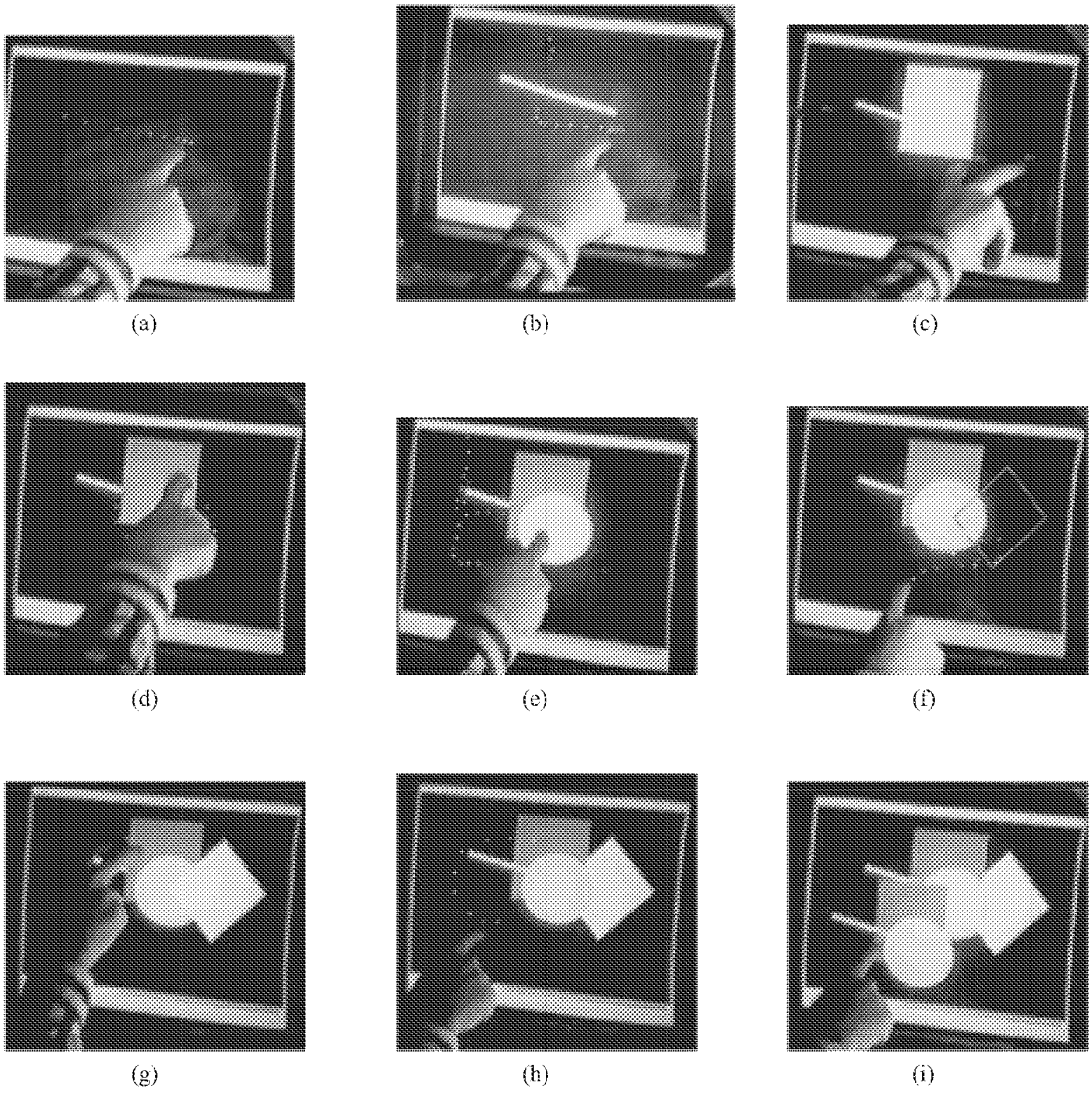


Figure 8.8: An example MDP session

This figure consists of snapshots of a video of an MDP session. Some panels have been retouched to make the inking more apparent. Panel (a) shows the single finger line *gesture, which is essentially the same as GDP's* line *gesture. As in GDP, the start of the gesture gives one endpoint of the line, while the other endpoint is dragged by the gesturing finger after the gesture is recognized. Additional fingers may be used to control the line's color and thickness. Panel (b) shows the created line, and the* rectangle *gesture, again the same as GDP's. After the gesture is recognized, additional fingers may be brought into the sensing plane to control the rectangle's color, thickness, and filled property, as shown in panel (c). Panel (d) shows the* circle *gesture, which works analogously. Panel (e) shows the two finger* parallelogram *gesture. After the gesture is recognized, the two gesturing fingers control two corners of the parallelogram. An additional finger*

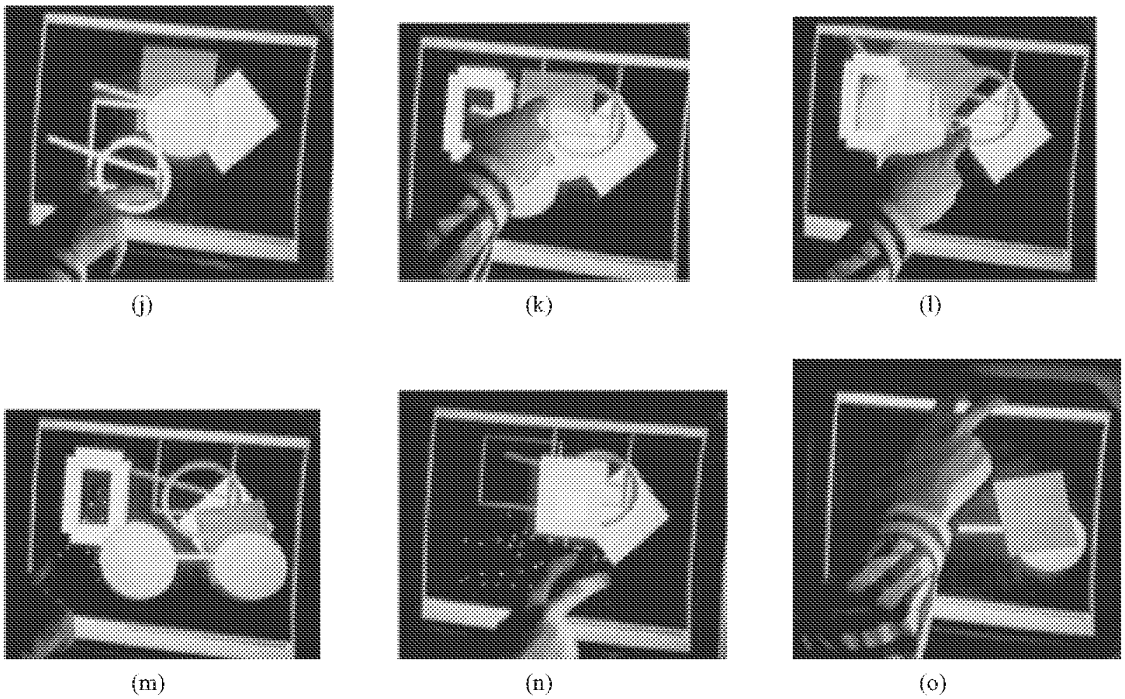


Fig 8.8 (continued)

in the sensing plane will then control a third corner, allowing an arbitrary parallelogram to be entered. Panel (f) shows the edit color *gesture being made at the newly created parallelogram. After this gesture is recognized, the parallelogram's color and filled property may be dynamically manipulated. Panel (g) shows the three finger* pack *(group) gesture. During the* pack *interaction, all object touched by any of the fingers are grouped into a single set. Here, the line, rectangle, and circle are grouped together to make a cart. Panel (h) shows the* copy *gesture. After the gesture is recognized, the object indicated by the first point of the gesture (in this case, the cart) is dragged by the gesturing finger, as shown in panel (i). Additional fingers allow the color, edge thicknesses, and filled property of the copy to be manipulated, as shown in panel (j).* Circle *and* rectangle *gestures (both not shown) were then used to create some additional shapes. Panel (k) shows the two finger* rotate *gesture. After it is recognized, each of the two fingers become attached to their respective points where they first touched the designated object. By moving the fingers apart or together, rotating the hand, and moving the hand, the object may be simultaneously scaled, rotated, and translated as shown in panel (l). (The fingers are not touching the object due to the delay in getting the input data and refreshing the screen.) Panel (m) shows the* delete *gesture being used to delete a rectangle. Not shown are more deletion and creation gestures, leaving the drawing in the state shown in panel (n). Panel (n) shows the three finger* undo *gesture. Upon recognition, the most recent creation or deletion is undone. Moving the fingers up causes more and more operations to be undone, while moving the fingers down allows undone operations to be redone, interactively. Panel (o) shows a state during the interaction where many operations have been undone. In this implementation, creations and deletions are undoable, but position changes are not. This explains why, in panel (o), only the cart items remain (undo back to panel (e)), but those items are in the position they assumed in panel (n).*

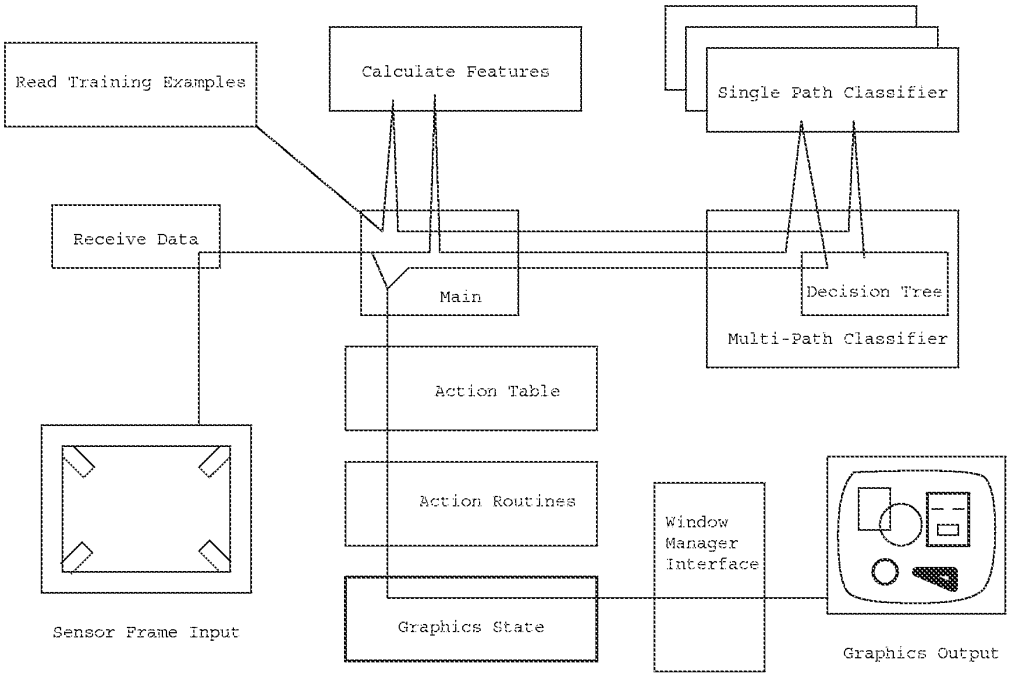


Figure 8.9: MDP internal structure

window coordinate system identical with the Sensor Frame coordinate system. If the given window size and position has not been seen before (as indicated by the alignment file) the user is forced to go through an alignment dialogue before proceeding (this also occurs when the window occurs moved or resized). Two dots are displayed, one in each corner of the window, and the user is asked to touch each dot. The data read are used to make window coordinates exactly match Sensor Frame coordinates. The transformation for window coordinates to screen coordinates is done by the IRIS software, and does not have to be considered by the rest of the program. The parameters are saved in the alignment file to avoid having to repeat the procedure each time MDP is started.[2]

Once initialized, the MDP begins to read data from the Sensor Frame. The current Sensor Frame software works by polling, and typically returns data at the rate of approximately 30 snapshots per second. The "Receive Data" module performs the path tracking (see section 5.1) and returns snapshot records consisting of the current time, number of fingers seen by the frame, and tuples (x, y, i) for each finger, (x, y) being the finger's location in the frame. The i is the path identifier, as determined by the path tracker. The intent is that a given value of i represents the same finger in successive snapshots.

Normally, MDP is in its WAIT state, where the polling indicates that there are no fingers in the plane of the frame. Once one or more fingers enter the field of view of the frame, the COLLECT state is entered. Each successive snapshot is passed to the "calculate features" module, which performs the incremental feature calculation. The COLLECT state ends when the user removes all fingers from the frame viewfield or stops moving for 150 milliseconds. (The timeout interval is settable by the user, but 150 milliseconds has been found to work well.) Unlike a mouse user, it is difficult for Sensor Frame users to hold their fingers perfectly still, so a threshold is used to decide when the user has not moved. In other words, the threshold determines the amount of movement allowable between successive snapshots that is to count as "not moving." This is done by comparing the threshold to the error metric calculated during the path tracking (sum of squared distances between corresponding points in successive snapshots).

Once the gesture has been collected, its feature vectors are passed to the multi-path classifier, which returns the gesture's class. Then the recognition action associated with the class is looked up in the action table and executed. As long as at least one finger remains in the field of view, the manipulation action of the class is executed.

Many of GRANDMA's ideas for specifying gesture semantics are used in MDP. Although MDP does not have a full-blown interpreter, there is a table specifying the recognition action and manipulation action for each class. While it would be possible for the tables to be constructed at runtime, currently the table is compiled into MDP. Each row in the entry for a class consists of a finger specification, the name of a C function to call to execute the row, and a constant argument to pass to the function. The finger specification determines which finger coordinates to pass as additional arguments to the function.

Consider the table entries for the MDP line gesture, similar to the GDP line gesture:

```
ACTION(_LINErecog)
    { ALWAYS,       BltnCreate,     (int)Line, },
```

[2]Moving or resizing the window often requires the alignment procedure to be repeated, a problem that would of course have to be fixed in a production version of the program.

```
    { START(0),      BltnSetPoint,    0, },
END_ACTION

ACTION(_LINEmanip)
    { CURRENT(0),    BltnSetPoint,    1, },
    { CURRENT(1),    BltnThickness,   0, },
    { CURRENT(2),    BltnColorFill,   0, },
END_ACTION
```

When a line gesture is recognized, the `_LINErecog` action is executed. Its first line results in the call `BltnCreate(Line)` being executed. The `ALWAYS` means that this row is not associated with any particular finger, thus no finger coordinates are passed to `BltnCreate`. The next line results in `BltnSetPoint(0,` x_{0s}, y_{0s}`)` being called, where (x_{0s}, y_{0s}) is the initial point of the first finger (finger 0) in the gesture.

For each snapshot after the line gesture has been recognized, the `_LINEmanip` action is executed. The first line causes `BltnSetPoint(1,` x_{0c}, y_{0c}`)` to be called, where (x_{0c}, y_{0c}) is the current location of the first finger (finger 0). The next line causes `BltnThickness(0,` x_{1c}, y_{1c}`)` to be called, (x_{1c}, y_{1c}) being the current location of the second finger. Similarly, the third line causes `BltnColorFill(0,` x_{2c}, y_{2c}`)` to be called.

If any of the fingers named in a line of the action are not actually in the field of view of the frame, that line is ignored. For example, the line gesture in MDP, as in GDP, is a single straight stroke. Immediately after recognition there will only be one finger seen by the frame, namely finger zero, so the lines beginning `CURRENT(1)` and `CURRENT(2)` will not be executed. If a second finger is now inserted into the viewfield, both the `CURRENT(0)` and `CURRENT(1)` lines will be executed every snapshot. If the initial finger is now removed, the `CURRENT(0)` line will no longer be executed, until another finger is placed in the viewfield.

The assignment of finger numbers is done as follows: when the gesture is first recognized, each finger is assigned its index in the path sorting (see Section 5.2). During the manipulation phase, when a finger is removed, its number is *freed*, but the numbers of the remaining fingers stay the same. When a finger enters, it is assigned the smallest free number.

The semantic routines (*e.g.* `BltnColorFill`) communicate with each other (and successive calls to themselves) via shared variables. All these functions are defined in a single file with the shared variables declared at the top. When there are no fingers in the viewfield, the call `BltnReset()` is made; its function is to initialize the shared variables. In MDP, all shared variables are initialized by `BltnReset()`; from this it follows that the interface is *modeless*. Another system might have some state retained across calls to `BltnReset()`; for example, the current selection might be maintained this way.

The `Bltn...` functions manipulate the drawing elements through a package of routines. The actual implementation of those routines is similar to the implementation of the GDP objects. Rather than go into detail, the underlying routines are summarized. MDP declares the following types:

```
typedef enum { Nothing, Line, Rect,
               Circle, SetOfObjects } Type;
```

```
typedef struct { /* ... */ } *Element;
typedef struct { /* ... */ } *Trans;
```

Assume the following declarations for expositional purposes:

```
Element e;          /* a graphic object */
Type    type;       /* the type of a graphic object */
int     x, y;       /* coordinates */
int     p;          /* a point number: 0, 1, or 2, 3 */
int     thickness, color;
BOOL    b;
Trans   tr;         /* a transformation matrix */
```

The `Element` is a pointer to a structure representing an element of a drawing, which is either a `Line`, `Rect`, `Circle` or `SetOfObjects`. The `Element` structure includes an array of points for those element types which need them. A `Line` has two points (the endpoints), a `Rect` has three points (representing three corners, thus a `Rect` is actually a parallelogram), and a `Circle` has two points (the center and a point on the circle). A `SetOfObjects` contains a list of component elements which make up a single composite element.

`Element StNewObj(type)` adds a new element of the passed `type` to the drawing, and returns a handle. Initially, all the points in the element are marked *uninitialized*. Any element with uninitialized points will not be drawn, with the exception of `Rect` objects, which will be drawn parallel to the axes if point 1 is uninitialized.

`StUpdatePoint(e, p, x, y)` changes point `p` of element `e` to be `(x,y)`. Returns FALSE iff `e` has no point `p`.

`StGetPoint(e, p, &x, &y)` sets `x` and `y` to point `p` of element `e`. Returns FALSE iff `e` has no point `p` or point `p` is uninitialized.

`StDelete(e)` deletes object `e` from the drawing.

`StFill(e, b)` makes object `e` filled if `b` is `TRUE`, otherwise makes `e` unfilled. This only applies to circles and rectangles, which will be only have their borders drawn if unfilled, otherwise will be "colored in."

`StThickness(e, t)` sets the thickness of `e`'s borders to `t`. Only applies to circles, rectangles, and lines.

`StColor(e, color)` changes the color of `e` to `color`, which is an index into a standard color map. If `e` is a set, all members of `e` are changed.

`StTransform(e, tr)` applies the transformation `tr` to `e`. In general, `tr` can cause translations, rotations, and scalings in any combination.

`void StMove(e, x, y)` is a special case of `StTransform` which translates `e` by the vector `(x,y)`.

`StCopyElement(e)` adds an identical copy of `e` to the drawing, which is also returned. If `e` is a set, its elements will be recursively copied.

`StPick(x, y)` returns the element in the drawing at point `(x,y)`, or `NULL` if there is no element there. The topmost element at `(x,y)` is returned, where elements created later are considered to be on top of elements created earlier. The thickness and "filled-ness" of an element are considered when determining if an element is at `(x,y)`.

`StHighlight(e, b)` turns on highlighting of `e` if `b` is `TRUE`, off otherwise. Highlighting is currently implemented by blinking the object.

`StUnHighlightAll()` turns off highlighting on all objects in the drawing.

`void StRedraw()` draws the entire picture on the display. Double buffering is used to ensure smooth changes.

`StCheckpoint()` saves the current state of the drawing, which can be later restored via `StUndoMore`.

`StUndoMore(b)` changes the drawing to its previously checkpointed state (if `b` is `TRUE`). Each successive call to `StUndoMore(TRUE)` returns to a previous state of the picture until the state of the picture when the program was started in reached. `StUndoMore(FALSE)` performs a redo, undoing the effect of the last `StUndoMore(TRUE)`. Successive calls to `StUndoMore(FALSE)` will eventually restore a drawing to its latest checkpointed state.

`Trans AllocTran()` allocates a transformation, which is initialized to the identity transformation.

`SegmentTran(tr, x0,y0, x1,y1, X0,Y0, X1,Y1)` sets `tr` to a transformation consisting of a rotation, followed by a scaling, followed by translation, the net effect of which would be to map a line segment with endpoints `(x0,y0)` and `(x1,y1)` to one with endpoints `(X0,Y0)` and `(X1,Y1)`. Other transformation creation functions exist, but this is the only one used directly by the gesture semantics.

`JotC(color, x, y, text)` draws the passed text string on the screen in the passed color, at the point `(x,y)`. The text will be erased at the next call to `StRedraw`.

8.3.2 MDP gestures and their semantics

Now that the basic primitives used by MDP have been described, the actual gestures used, and their effect and implementation are discussed. Figure 8.10 shows typical examples of the MDP gestures used. Each is described in turn.

Line The line gesture creates a line with one endpoint being the start of the gesture, the other tracking finger 0 after the gesture has been recognized. Finger 1 (which must be brought in after the gesture has been recognized) controls the thickness of the line as follows: the point

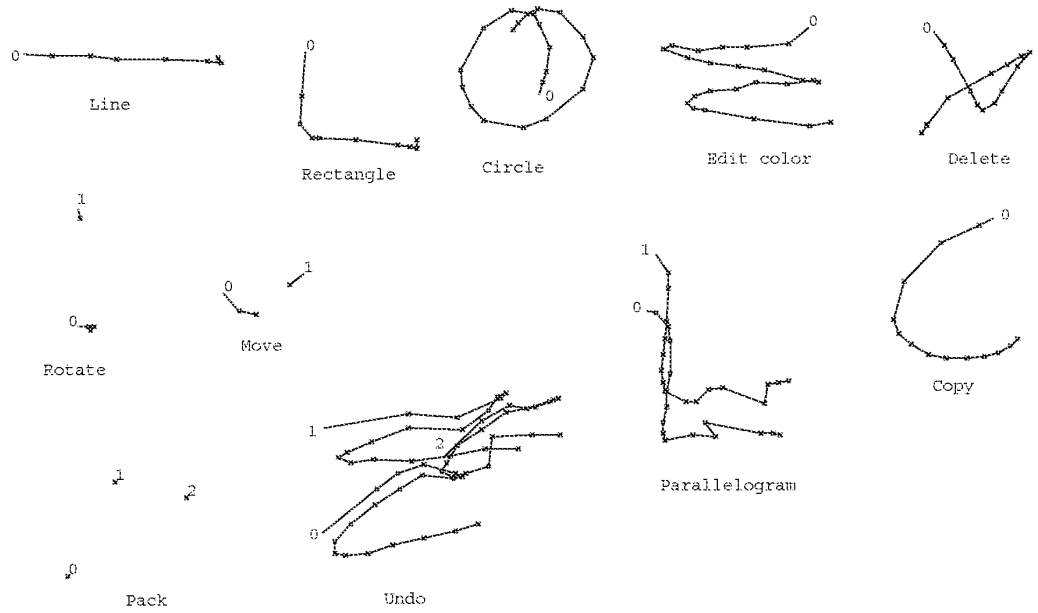


Figure 8.10: MDP gestures

where finger 1 first enters is displayed on the screen; the thickness of the line is proportional to the difference in y coordinate of finger 1's current point and initial point. Finger 2 controls the color of the line in a similar manner. (Here a color is represented simply by an index into a color map.)

The action table entry for line has already been listed in the previous section. The C routines called are listed here:

```
BltnCreate(arg) {
    E = StNewObj(arg);
    shouldCheckpoint = TRUE;
}
BltnSetPoint(arg, gx, gy) {
    if(E) StUpdatePoint(E, arg, gx, gy);
}
BltnThickness(arg, gx, gy) { int x, t;
    if(tx == -1) tx = gx, ty = gy;
    if(!E) return;
    x = arg==0 ?  abs(tx-gx) :  abs(ty-gy);
    t = Scale(x, 1, 2, 1, 100);
    StThickness(E, t);
    JotC(RED, tx, ty, arg==0 ?  "TX%d" :  "TY%d", t);
```

```
    JotC(RED, gx, gy+10, "t");
}
BltnColorFill(arg, gx, gy) { int color, fill;
    if(!E) return;
    if(cfx == -1) cfx = gx, cfy = gy;
    fill = Scale(cfx - gx, 1, 10, -1, 1);
    StFill(E, fill >= 0);
    color = Scale(cfy - gy, 1, 25, -15, 15);
    if(color < 0) color = -color;
    else if(color == 0) color = 1;
    StColor(E, color);
    JotC(GREEN, cfx, cfy, "CF%d/%d", color, fill);
    JotC(GREEN, gx, gy+10, "cf");
}
Scale(i, num, den, low, high) {
    int j = i * num;
    int k = j >= 0 ? j/den : -((-j)/den);
    return k < low ? low : k > high ? high : k;
}
```

The `BltnReset()` function sets `E` to `NULL`, and sets `tx`, `ty`, `cfx`, and `cfy` all to `-1`. `BltnReset()` calls `StCheckpoint()` if `shouldCheckpoint` is `TRUE` and then sets `shouldCheckpoint` to `FALSE`.

The functions `BltnThickness` and `BltnColorFill` provide feedback to the user by jotting some text ("TX" and "CF", respectively) that indicates the location that the finger first entered the viewfield. Lower case text ("t" and "cf") is drawn at the appropriate fingers, indicating to the user which finger is controlling which parameter.

Rectangle The rectangle gesture works similarly to the line gesture. After the gesture is recognized, a rectangle is created, one corner at the starting point of the gesture, the opposite corner tracking finger 0. Fingers 1 and 2 control the thickness and color as with the line gesture. Finger 2 also controls whether or not the rectangle is filled; if it is to the left of where it initially entered, the rectangle is filled, otherwise not.

```
ACTION(_RECTrecog)
    { ALWAYS,        BltnCreate,    (int)Rect, },
    { START(0),      BltnSetPoint,  0, },
END_ACTION

ACTION(_RECTmanip)
    { CURRENT(0), BltnSetPoint,  2, },
    { CURRENT(1), BltnThickness, 0, },
    { CURRENT(2), BltnColorFill, 0, },
END_ACTION
```

Circle The circle gesture causes a circle to be created, the starting point of the gesture being the center, and a point on the circle controlled by finger 0. Fingers 1 and 2 operate as they do in the rectangle gesture. Its semantics of the circle gesture are almost identical that of the line gesture, and are thus not shown here.

Edit color This gesture lets the user edit the color and "filled-ness" of an existing object. Beginning the gesture on an object edits that object. Otherwise, the user moves finger 0 until he touches an object to edit. Once selected, finger 0 determines the color and fill properties of the object as finger 2 did in the previous gestures.

```
ACTION(_COLORrecog)
    { START(0),     BltnPick,     0, },
END_ACTION

ACTION(_COLORmanip)
    { CURRENT(0),     BltnPickIfNull,     0, },
    { CURRENT(0),     BltnColorFill,      0, },
END_ACTION

BltnPick(arg, gx, gy) {
    E = StPick(gx, gy);
    if(E) px = gx, py = gy;
}
BltnPickIfNull(arg, gx, gy) {
    if(!E) BltnPick(arg, gx, gy);
}
```

Copy The copy gesture picks an element to be copied in the same manner as the edit-color gesture above. Once copied, finger 0 drags the new copy around, while finger 1 can be used to adjust the color and thickness of the copy.

```
ACTION(_COPYrecog)
    { START(0),         BltnPick,         0, },
END_ACTION

ACTION(_COPYmanip)
    { CURRENT(0),       BltnPickIfNull,     0, },
    { CURRENT(0),       BltnCopy,           0, },
    { CURRENT(0),       BltnMove,           0, },
    { CURRENT(1),       BltnColorFill,      0, },
END_ACTION
```

In the interest of brevity the C routines will no longer be listed, since they are very similar to those already seen.

Move Move is a two-finger "pinching" gesture. An object is picked as in the previous gestures, and then tracks finger 0.

```
ACTION(_MOVErecog)
    { START(0),         BltnPick,         0, },
END_ACTION

ACTION(_MOVEmanip)
    { CURRENT(0),       BltnPickIfNull,     0, },
    { CURRENT(0),       BltnMove,           0, },
END_ACTION
```

Delete The delete gesture picks an object just like the previous gestures, and then deletes it.

```
ACTION(_DELETErecog)
    { START(0),         BltnPick,         0, },
END_ACTION

ACTION(_DELETEmanip)
    { CURRENT(0),       BltnPickIfNull,   0, },
    { CURRENT(0),       BltnDelete,       0, },
END_ACTION
```

Parallelogram The parallelogram gesture is a two-finger gesture. One corner of the parallelogram is determined by the initial location of fingers 0; an adjacent corner tracks finger 0, and the opposite corner tracks finger 1. Adding a third finger (finger 2) moves the initial point of the parallelogram.

```
ACTION(_PARArecog)
    { ALWAYS,         BltnCreate,     (int)Rect, },
    { START(0),       BltnSetPoint,    0, },
END_ACTION

ACTION(_PARAmanip)
    { CURRENT(0),         BltnSetPoint,    1, },
    { CURRENT(1),         BltnSetPoint,    2, },
    { CURRENT(2),         BltnSetPoint,    0, },
END_ACTION
```

Rotate Rotate is a two-finger gesture. An object is picked with either finger. At the time of the pick, each finger becomes attached to a point on the picked object. Each finger then drags its respective point; the object can thus be rotated by rotating the fingers, scaled by moving the fingers apart or together, or translated by moving the fingers in parallel.

```
ACTION(_ROTATErecog)
    { START(0),     BltnPick,         0, },
    { START(1),     BltnPickIfNull,   0, },
END_ACTION
```

```
ACTION(_ROTATEmanip)
    { CURRENT(0),   BltnPickIfNull,   0, },
    { CURRENT(1),   BltnPickIfNull,   0, },
    { CURRENT(0),   BltnRotate,       0, },
    { CURRENT(1),   BltnRotate,       1, },
END_ACTION
```

Pack The pack gesture is a three-finger gesture. Any objects touched by the any of the fingers are added to a newly created `SetOfObjects`.

```
ACTION(_PACKrecog)
END_ACTION

ACTION(_PACKmanip)
    { CURRENT(0),         BltnPick,         0, },
    { ALWAYS,         BltnAddToSet,         0, },
    { CURRENT(1),         BltnPick,         0, },
    { ALWAYS,         BltnAddToSet,         0, },
    { CURRENT(2),         BltnPick,         0, },
    { ALWAYS,         BltnAddToSet,         0, },
END_ACTION
```

Undo The undo gesture is also a three-finger gesture, basically a "Z" made with three fingers moving in parallel. After it is recognized, moving finger 0 up causes more and more of the edits to be undone, and moving finger 0 down causes those edits to be redone.

```
ACTION(_UNDOrecog)
    { CURRENT(0),         BltnUndo,         0, },
END_ACTION

ACTION(_UNDOmanip)
    { CURRENT(0),         BltnUndo,         0, },
END_ACTION
```

8.3.3 Discussion

MDP is the only system known to the author which uses non-DataGlove multiple finger gestures. Thus, a brief discussion of the gestures themselves is warranted.

MDP's single finger gestures are taken directly from GDP. After recognition, additional fingers may be brought into the sensing plane to control additional parameters. Wherever an additional finger is first brought into the sensing plane becomes the position that gives the current value of the parameter which that finger controls; the position of the finger relative to this initial position determines the new value of the parameter. This relative control was felt by the author to be less awkward than other possible schemes, though this of course needs to be studied more thoroughly.

The multiple finger gestures are designed to be intuitive. The parallelogram gesture is, for example, two fingers making the rectangle gesture in parallel. The move gesture is meant to be a pinch, whereby the object touched is grabbed and then dragged around. The two finger rotate gesture allows two distinct points on an object to be selected carefully. During the manipulation phase, each of these points tracks a finger, allowing for very intuitive translation, rotation, and scaling of the object. The three finger undo gesture is intended to simulate the use of an eraser on a blackboard.

The Sensor Frame is not a perfect device for gestural input. One problem with the Sensor Frame is that the sensing plane is slightly above the surface of the screen. It is difficult to precisely pull a finger out without changing its position. This often results in parameters that were carefully adjusted during the manipulation phase of the interaction being changed accidentally as the interaction ends. This problem happens more often in multiple finger gestures, where, due to problems with the Sensor Frame, removing one finger may change the reported position of other fingers even though those fingers have not moved. Also, it is more difficult to pull out one finger carefully when other fingers must be kept still in the sensing plane. Finally, it does not take very long for a gesturer's arm to get tired when using a Sensor Frame attached to a vertically mounted display.

In MDP, the two-phase interaction technique is applied in the context of multiple fingers. As each finger's position represents two degrees of freedom, multi-path interactions allow many more parameters to be manipulated than do single-path interactions. Also, since people are used to gesturing with more than one finger, multiple fingers potentially allows for more natural gestures. Even though sometimes only one or two fingers are used to enter the recognized part of the gesture, additional fingers can then be utilized in the manipulation phase. The result is a new interaction technique that needs to be studied further.

8.4 Conclusion

This chapter described the major applications which were built to demonstrate the ideas of this thesis. Two, GDP and GSCORE, were built on top of GRANDMA, and show how single-path gestures may be integrated into MVC-based applications. The third, MDP, demonstrates the use of multi-path gestures, and shows how gestures may be integrated in a quick and dirty fashion in a non-objected-oriented context.

Chapter 9

Evaluation

The previous chapters report on some algorithms and systems used in the construction of gesture-based applications. This chapter attempts to evaluate how well those algorithms and systems work. When possible, quantitative evaluations are made. When not, subjective or anecdotal evidence is presented.

9.1 Basic single-path recognition

Chapter 3 presents an algorithm for classifying single-path gestures. In this section the performance of the algorithm is measured in a variety of ways. First, the recognition rate of the classifier is measured, as a function of the number of classes and the number of training examples. By examining the gestures that were misclassified, various sources of errors are uncovered. Next, the effect of the rejection parameters on classifier performance is studied. Then, the classifier is tested on a number of different gesture sets. Finally, tests are made to determine how well a classifier trained by one person recognizes the gestures of another.

9.1.1 Recognition Rate

The *recognition rate* of a classifier is the fraction of example inputs that it correctly classifies. In this section, the recognition rates of a number of classifiers trained using the algorithm of Chapter 3 are measured. The gesture classes used are drawn from those used in GSCORE (Section 8.2). There are two reasons for testing on this set of gestures rather than others discussed in this dissertation. First, it consists of a fairly large set of gestures (30) used in a real application. Second, the GSCORE set was not used in the development or the debugging of the classification software, and so is unbiased in this respect.

GRANDMA provides a facility through which the examples used to train a classifier are classified by the classifier. While running the training examples through the classifier is useful for discovering ambiguous gestures and determining approximately how well the classifier can be expected to perform, it is not a good way to measure recognition rates. Any trainable classifier will be biased toward recognizing its training examples correctly. Thus in all the tests described below, one set of

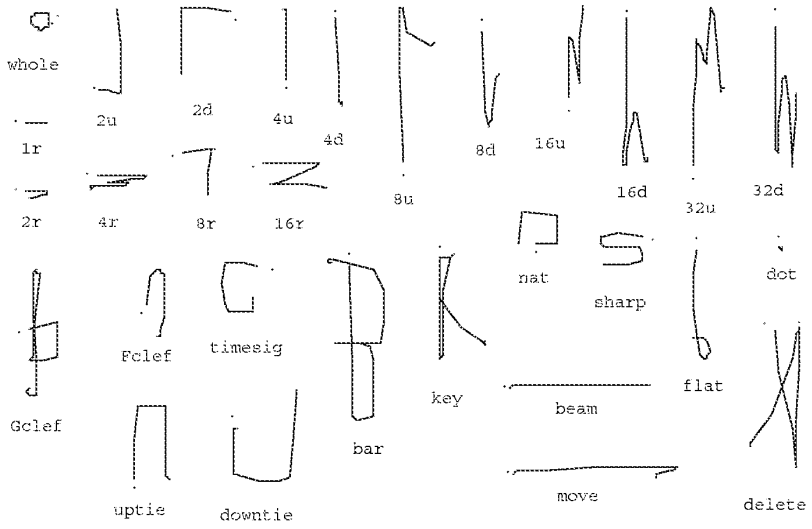


Figure 9.1: GSCORE gesture classes used for evaluation

example gestures is used to train the classifier, while another, entirely distinct, set of examples is used to evaluate its performance.

Figure 9.1 shows examples of the gesture classes used in the first test. All were entered by the author, using the mouse and computer system described in Chapter 3. First, 100 examples of each class were entered; these formed the *training set.* Then, the author entered 100 more examples of each class; these formed the *testing set.* For both sets, no special attempt to was made to gesture carefully, and obviously poor examples were not eliminated.

There was no classification of the test examples as they were entered; in other words, no feedback was provided as to the correctness of each example immediately after it was entered. Given such feedback, a user would tend to adapt to the system and improve the recognition of future input. The test was designed to eliminate the effect of this adaptation on the recognition rate.

The performance of the statistical gesture recognizer depends on a number of factors. Chief among these are the number of classes to be discriminated between, and the number of training examples per class. The effect of the number of classes is studied by building recognizers that use only a subset of classes. In the experiment, a class size of C refers to a classifier that attempts to discriminate between the first C classes in figure 9.1. Similarly, the effect of the training set size is studied by varying E, the number of examples per class. A given value of E means the classifier was trained on examples 1 through E of the training data for each of C classes.

Figure 9.2 plots the recognition rate against the number of classes C for various training set sizes E. Each point is the result of classifying 100 examples of each of the first C classes in the testing set. The number of correct classifications is divided by the total number of classifications attempted ($100C$) to give the recognition rate. (Rejection has been turned off for this experiment.) Figure 9.3 shows the results of the same experiment plotted as recognition rate versus E for various values of

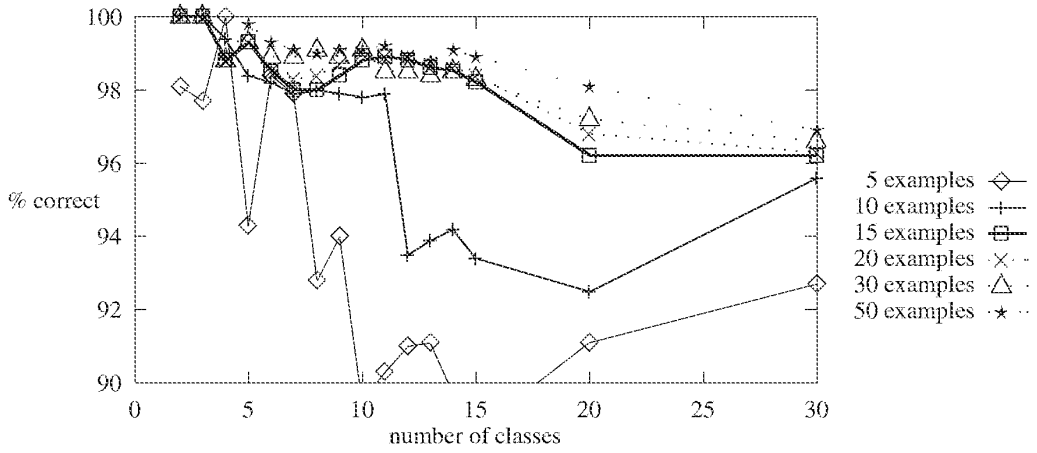


Figure 9.2: Recognition rate vs. number of classes

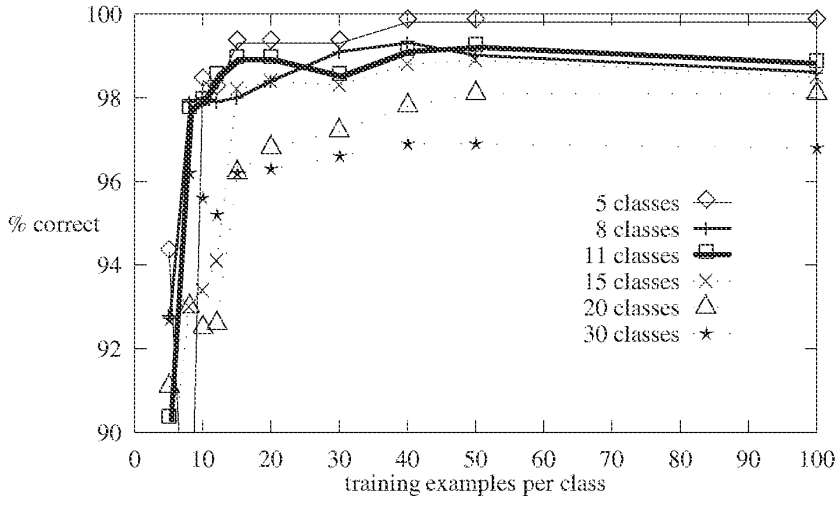


Figure 9.3: Recognition rate vs. training set size

C.

In general, the data are not too surprising. As expected, recognition rate increases as the training set size increases, and decreases as the number of classes increases. For $C = 30$ classes, and $E = 40$ examples per class, the recognition rate is 96.9%. For $C = 30$ and $E = 10$ the rate is 95.6%. $C = 10$ and $E = 40$ gives a rate of 99.3%, while for $C = 10$ and $E = 10$ the rate is 97.8%.

Of practical significance for GRANDMA users is the number of training examples needed to give good results. Using $E = 15$ examples per class gives good results, even for a large number of classes. Recognition rate can be marginally improved by using $E = 40$ examples per class, above which no significant improvement occurs. $E = 10$ results in poor performance for more than $C = 10$ classes. It is comforting to know that GRANDMA, a system designed to allow experimentation with gesture-based interfaces, performs well given only 15 examples per class. This is in marked contrast to many trainable classifiers, which often require hundreds or thousands of examples per class, precluding their use for casual experimentation [125, 47].

Analysis of errors

It is enlightening to examine the test examples that were misclassified in the above experiments. Figure 9.4 shows examples of all the kinds of misclassifications by the $C = 30$, $E = 40$ classifier. Not every misclassification is shown in the figure, but there is a representative of every A classified as B, for all $A \neq B$. The label "A as B (x n)" indicates that the example was labeled as class A in the test set, but classified as B by the classifier. The n indicates the number of times an A was classified as a B, when it is more than once.

The following types of errors can be observed in the figure. Many of the misclassifications are the result of a combination of two of the types.

Poorly drawn gestures. Some of the mistakes are simply the result of bad drawing on the part of the user. This may be due to carelessness, or to the awkwardness of using a mouse to draw. Examples include "8u as uptie," "2r as sharp," "8r as 2r," and "delete as 16d." "Felef as dot" was due to an accidental mouse click, and in "delete as 8d" the mouse button was released prematurely. The example "key as delete" was likely an error caused by the mouse ball not rolling properly on the table. "4u as 8u" and "16d as delete" each have extraneous points at the end of the gesture that are outside the range normally eliminated by the preprocessing. "4r as 16r" is drawn so that the first corner in the stroke is looped (figure 9.5); this causes the accumulated-angle features f_9, f_{10}, and f_{11} to be far from their expected value (see Section 3.3).

Poor mouse tracking. Many of the errors are due to poor tracking of the mouse. Typically, the problem is a long time between the first mouse point of a gesture and the second. This occurs when the first mouse point causes the system to page in the process collecting the gesture; this may take a substantial amount of time. The underlying window manager interface queues up every mouse event involving the press or release of a button, but does not queue successive mouse-movement events, choosing instead to keep only the most recent. Because of this, mouse movements are missed while the process is paged in.

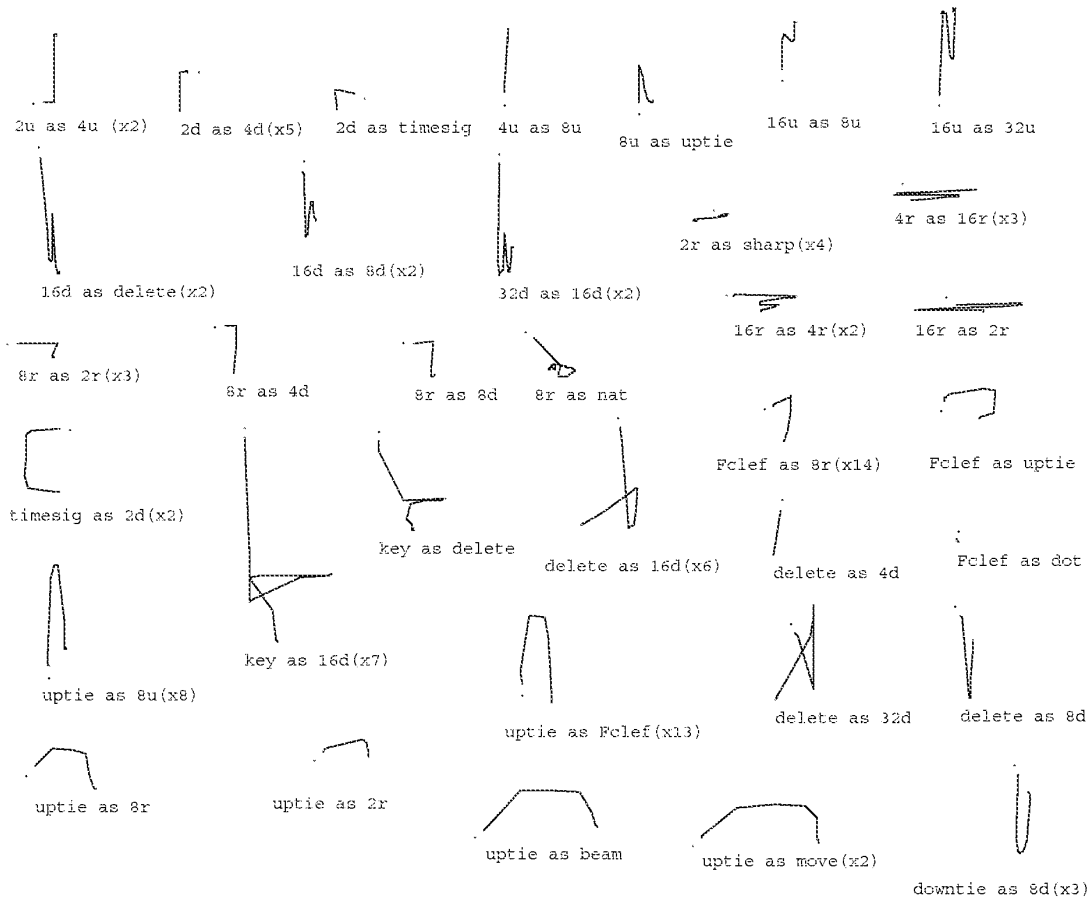


Figure 9.4: Misclassified GSCORE gestures

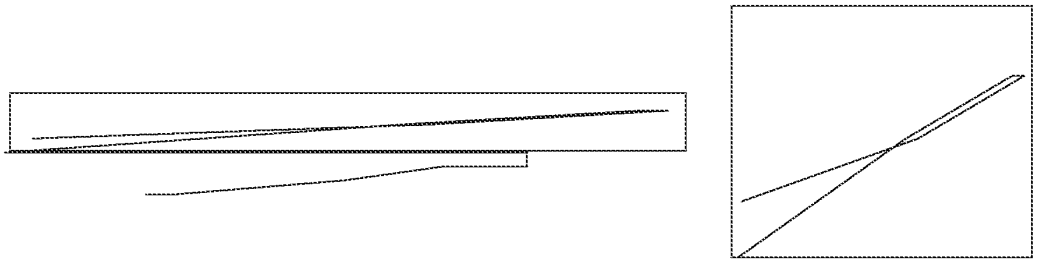


Figure 9.5: A looped corner

The left figure is a magnification of a misclassified "4r as 16r" shown in the previous figure. The portion of the gesture enclosed in the rectangle has been copied and its aspect ratio changed, resulting in the figure on the right. As can been seen, the corner, which should be a simple angle, is looped. This resulted in the angle-based features having values significantly different from the average 4r *gesture, thus the misclassification.*

In "2u as 4u," "2d as 4d," "8r as 4d," "8r as 8d," and "timesig as 2d" there is no point between the initial point and the first corner, probably due to the paging. This interacts badly with the features f_1 and f_2, the cosine and sine of initial angle. Features f_1 and f_2 are computed from the first and *third* point; this usually results in a better measurement than using the first and second point. In these cases, however, this results in a poor measurement, since the third point is after the corner.

"8r as nat" was the result of a very long page in, during which the author got impatient and jiggled the mouse.

Ambiguous classes. Some classes are very similar to each other, and are thus likely to be mistaken for each other. The 14 misclassifications of "Fclef as 8r" are an example. Actually, these may also be considered examples of poor mouse tracking, since points lost from the normally rounded top of the Fclef gesture caused the confusion. The mistakes "uptie as 8u," and "uptie as Fclef" are also examples of ambiguity.

Ideally, the gesture classes of an application should be designed so as to be as unambiguous as possible. Given nearly ambiguous classes, it is essential that the input device be as reliable and as ergonomically sound as possible, that the features be able to express the differences, and that the decider be able to discriminate between them. Without all of these properties, it is inevitable that there will be substantial confusion between the classes.

Inadequacy of the feature set. The examples where the second mouse point is the first corner show one way in which the features inadequately represent the classes. For example, the "2r as sharp" examples appear to the system as simple left strokes. Sometimes, a small error in the drawing results in a large error in a feature. This occurs most often when a stroke doubles back on itself; a small change results in a large difference in the angle features f_9, f_{10}, and f_{11} (see figure 9.5). The mistakes "4r as 16r" and "16d as delete" are in this category. "16u as 8u" and "16u as 32u" point to other places where the features may be improved.

The mapping from gestures to features is certainly not invertible; many different gestures might have the same feature vector in the current scheme. This results in ambiguities not due entirely to similarities between classes, but due to a feature set unable to represent the difference. Example "key as 16d" is an illustration of this, albeit not a great one.

Inadequacy of linear, statistical classification. Given that the differences between classes can be expressed in the feature vector, it still may be possible that the classes cannot be separated well by linear discrimination functions. This typically comes about when a class has a feature vector with a severely non-multivariate-normal distribution. In the current feature set, this most often happens in a class where the gesture folds back on itself (as discussed earlier), causing f_9, and thus the entire feature vector, to have a bimodal distribution.

The averaging of the covariance matrix in essence implies that a given feature is equally important in all classes. In the above class, the initial angle features are deemed important by the classifier. When compounded with errors in the tracking, this leads to bad performance on examples such as "uptie as beam" and "uptie as move." It is possible for a linear classifier to express the per-class importance of features in a linear classifier; in essence this is what is done by the neural-network-like training procedures (*a.k.a* back propagation, stochastic gradient, proportional increment, or perceptron training).

Inadequate training data. Drawing and tracking errors occur in the training set as well as the testing set. Given enough good examples, the effect of bad examples on the estimates of the average covariance matrix and the mean feature vectors is negligible. This is not the case when the number of examples per class is very small. Bad or insufficient training data causes bad estimates for the classifier parameters, which in turn causes classification errors. The gestures classified correctly by the $C = 30$, $E = 40$ classifier, but incorrectly by the $C = 30$, $E = 10$ classifier are examples of this.

Analyzing errors in this fashion leads to a number of suggestions for easy improvements to the classifier. Timing or distance information can be used to decide whether to compute f_1 and f_2 using the first two points or the first and third points of the gesture. Mouse events could be queued up to improve performance in the presence of paging. Some new features can be added to improve recognition even in the face of other errors; in particular, the cosine and sine of the final angle of the gesture stroke would help avoid a number of errors. These modification are left for future work, as the author, at the present time, has no desire to redo the above evaluation using 6000 examples from a different gesture set.

One error not revealed in these tests, but seen in practice, is misclassification due to a premature timeout in the two-phase interaction. This results in a gesture being classified before it is completely entered.

9.1.2 Rejection parameters

Section 3.6 considered the possibility of rejecting a gesture, *i.e.* choosing not to classify it. Two parameters potentially useful for rejection were developed. An estimate of the probability that a gesture is classified unambiguously, $\tilde{P}$, is derived from the values of the per-class evaluation

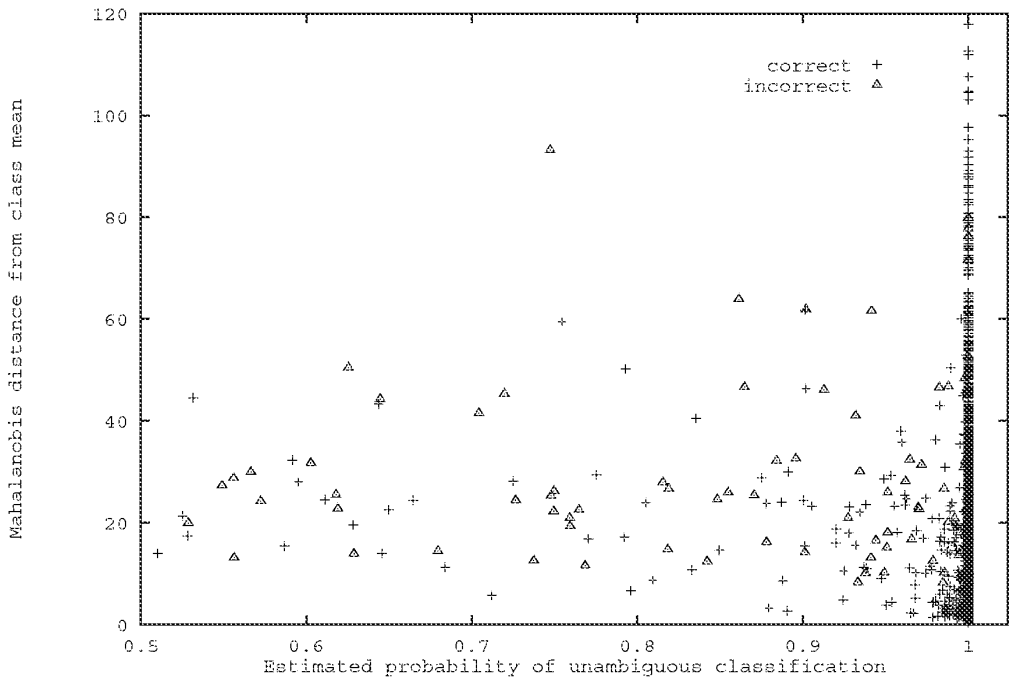


Figure 9.6: Rejection parameters

functions. An estimate of the Mahalanobis distance, d^2, is used to determine how close a gesture is to the norm of its chosen class.

It would be nice if thresholds on the rejection parameters could be used to neatly separate correctly classified example from incorrectly classified examples. It is clear that it would be impossible to do a perfect job; as "delete as 8d" illustrates, the system would need to read the user's mind. The hope is that most of the incorrectly classified gestures can be rejected, without rejecting too many correctly classified gestures.

A little thought shows that any rejection rule based solely on the ambiguity metric $\tilde{P}$ will on the average reject at least as many correctly classified gestures as incorrectly classified gestures. This follows from the reasonable conjecture that the average ambiguous gesture is at least as likely to be classified correctly as not. (This assumes that the gesture is not equally close to three or more classes. In practice, this assumption is almost always true.)

Figure 9.6 is a scatter plot that shows the value for both rejection parameters for all the gestures in the GSCORE test set. A plus sign indicates a gesture classified correctly; a triangle indicates each gesture classified incorrectly, *i.e.* those represented in figure 9.4. Most of the correctly classified examples have an estimated unambiguity probability of very close to one, thus accounting for the dark mass of points at the right of the graph. 96.3% of the correctly classified examples had

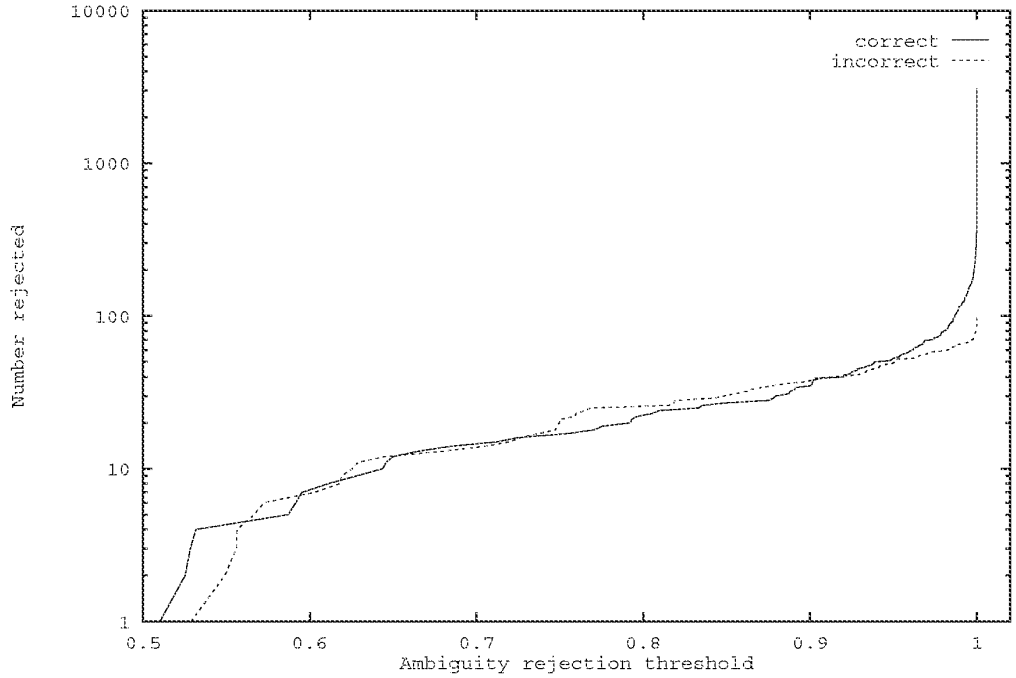


Figure 9.7: Counting correct and incorrect rejections

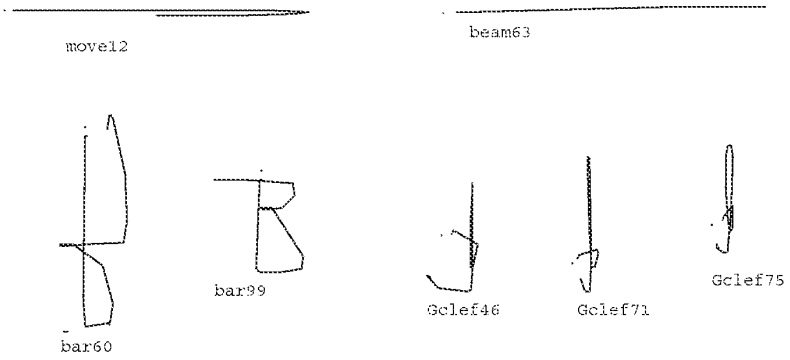


Figure 9.8: Correctly classified gestures with $d^2 \geq 90$

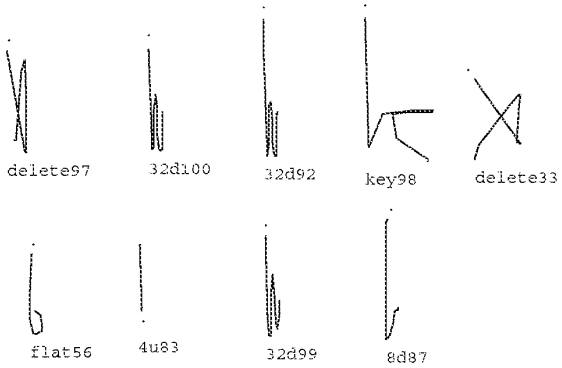


Figure 9.9: Correctly classified gestures with $\tilde{P} \leq .95$

$\tilde{P} \geq 0.99$. However, the same interval contained 33.7% of the incorrectly classified examples. Figure 9.7 shows how many correctly classified and how many incorrectly classified gestures would be rejected as a function of the threshold on $\tilde{P}$.

Examining exactly which of the incorrect examples have $\tilde{P} \geq 0.99$ is interesting. The garbled "8r as nat" and the left stroke "2r as sharp" have $\tilde{P} = 1.0$ within six decimal places. In retrospect this is not surprising; those gestures are far from *every* class, but happen to be unambiguously closest to a single class. This is borne out in the d^2 for those gestures, which is 380 (off the graph) for "8r as nat" and at least 70 for each "2r as sharp" gesture. Other mistakes have $\tilde{P} > .999$ but $d^2 < 20$. In this category are "Fclef as 8r," "uptie as Fclef," "delete as 8d," and "4u as 8u"; these gestures go beyond ambiguity to look like their chosen classes so could not be expected to be rejected.

Also interesting are those correctly classified test examples that are candidates for rejection based on their $\tilde{P}$ and d^2 values. Figure 9.8 shows some GSCORE gestures whose $\tilde{P} = 1$ and $d^2 \geq 90$. Examples "move12" and "beam63" are abnormal only by virtue of the fact that they are larger than

normal. The two bar examples have their endpoints in funny places, among other things, while the three Gclef examples are fairly unrecognizable. The algorithm does however classify all of these correctly; and it would be too bad to reject them. Figure 9.9 shows gestures whose ambiguity probability is less that .95. In many of the examples this is caused by at least one corner being made by two mouse points rather than one. In "delete33" one corner is looped. These gestures look so much like their prototypes it would be too bad to reject them.

The Mahalanobis estimate is mainly useful for rejecting gestures that were deliberately entered poorly. This is not as silly as it sounds; a user may decide during the course of a gesture not to go through with the operation, and at that time extend the gesture into gibberish so that it will be rejected.

One possible improvement would be to use the per-class covariance matrix of the chosen class in the Mahalanobis distance calculation. Compared to using the average covariance matrix, this would presumably result in a more accurate measurement of how much the input gesture differs for the norm of its chosen class.

9.1.3 Coverage

Figure 9.10 shows the performance of the single-path gesture recognition algorithm on five different gesture sets. The classifier for each set was trained on fifteen examples per class and tested on an additional fifteen examples per class. The first set, based on Coleman's editor [25], had a substantial amount of variation within each class, both in the training and the testing examples. The remaining sets had much less variation with each class. As the rates demonstrate, the single-path gesture recognition algorithm performs quite satisfactorily on a number of different gesture sets.

9.1.4 Varying orientation and size

One feature that distinguishes gesture from handwriting is that the orientation or size of a gesture in a given class may be used as an application parameter. For this to work, gestures of such classes must be recognized as such independent of their orientation or size. However, the recognition algorithm should not be made completely orientation and size independent, as some other classes may depend on orientation and size to distinguish themselves.

It is straightforward to indicate those classes whose gestures will vary in size or orientation: simply vary the size or orientation of the training examples. The goal of the gesture recognizer is to make irrelevant those features in classes for which they do not matter, while using those feature in classes for which they do.

Theoretically, having some classes that vary in size and orientation, while other that depend on size or orientation for correct classification should be a problem for any statistical classifier based on the assumptions of a multivariate normal distribution of features per class, with the classes having a common covariance matrix. A class whose size is variable is sure to have a different covariance matrix than one whose size remains relatively constant; the same may be said of orientation. Thus, we would suspect the classifier of Chapter 3 to perform poorly in this situation. Surprisingly, this does not seem to be the case.

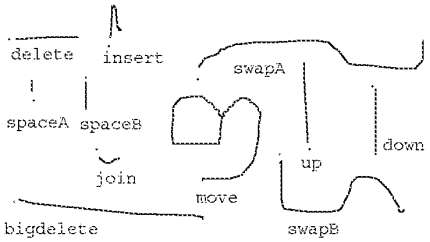
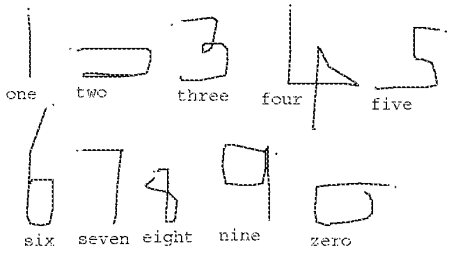
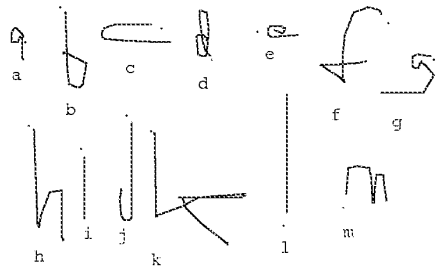
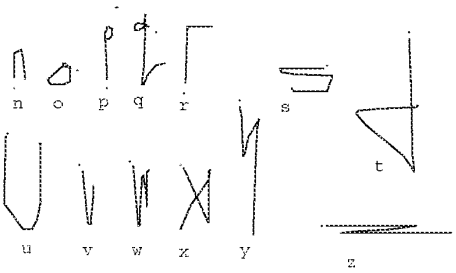
Set Name	Gesture Classes	Number of Classes	Recognition Rate
Coleman		11	100.0%
Digits		10	98.5%
Let:a-m		13	99.2%
Let:n-z		13	98.4%
Letters	Union of Let:a-m and Let:n-z	26	97.1%

Figure 9.10: Recognition rates for various gesture sets

Each set was trained with 15 examples per class and tested on an additional 15 examples per class.

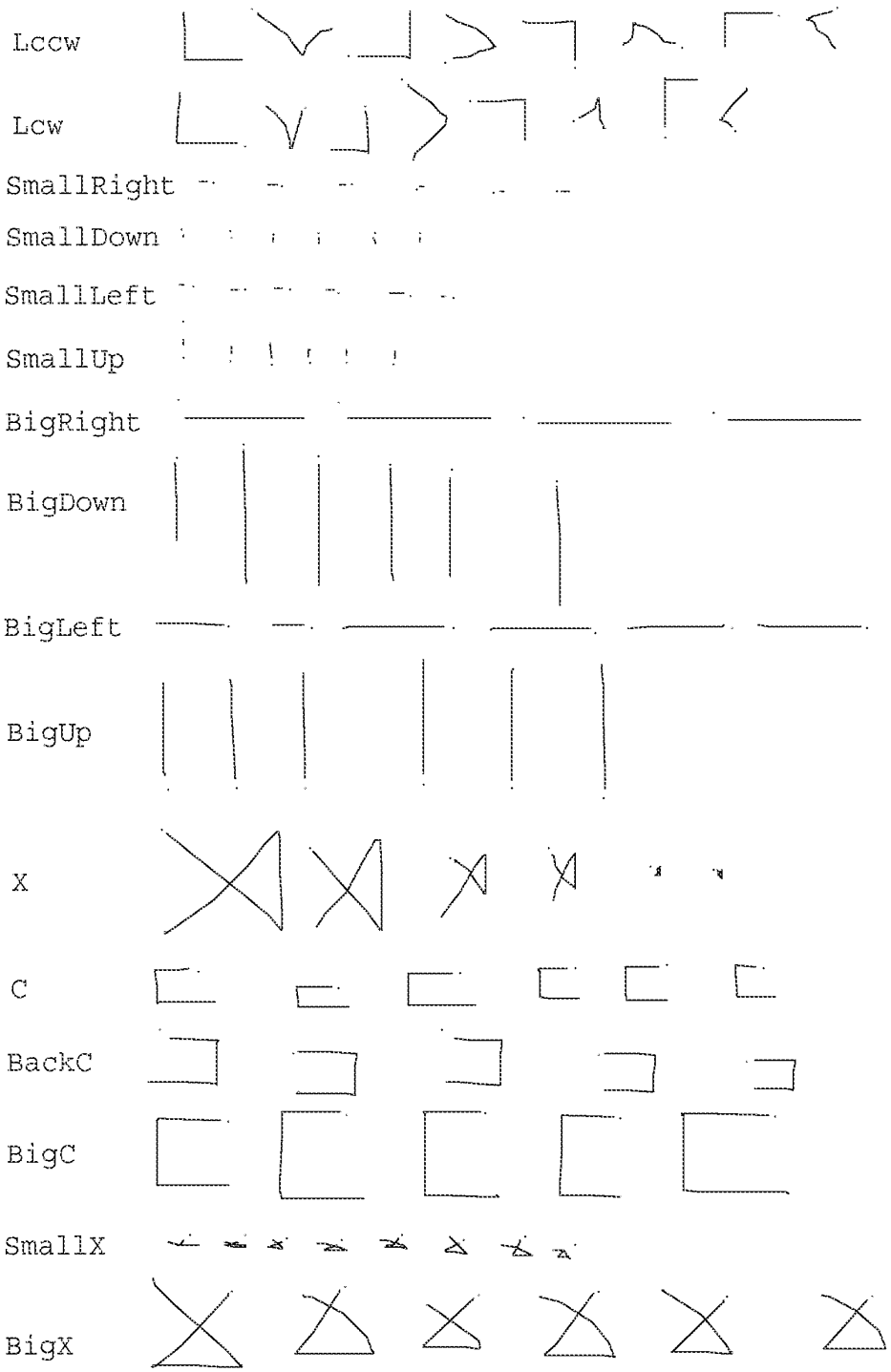


Figure 9.11: Classes used to study variable size and orientation

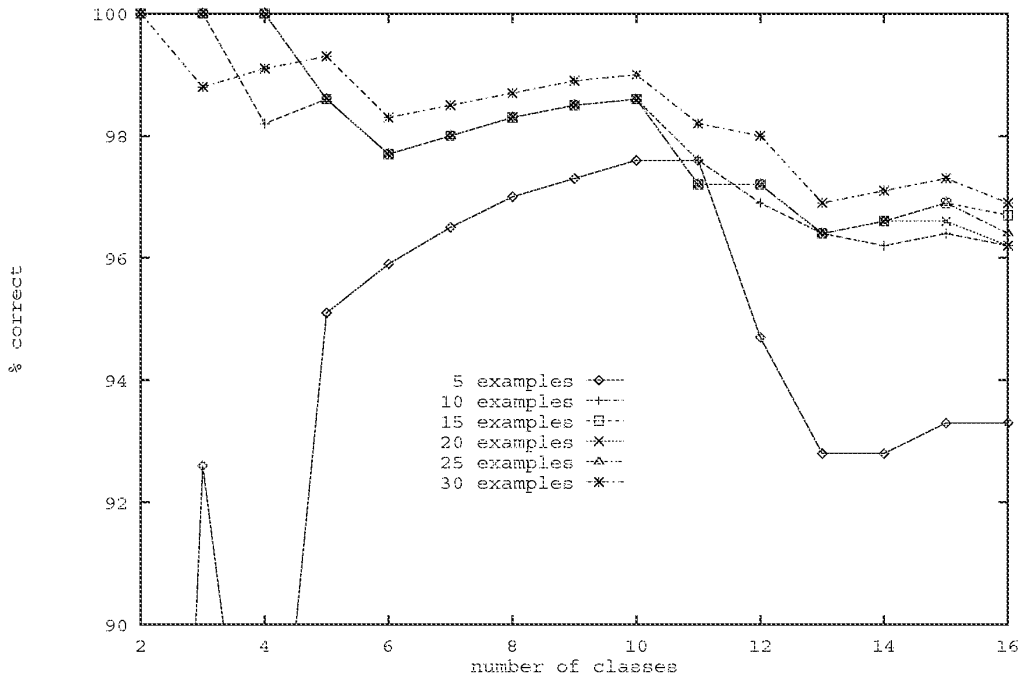


Figure 9.12: Recognition rate for set containing classes that vary

Figure 9.11 shows 16 classes, some of which vary in size, some of which vary in orientation, others of which depend on size or orientation to be distinguishable. The training set consists of thirty examples of each class; variations in size or orientation were reflected in the training examples, as shown in the figure. A testing set with thirty or so examples per class was similarly prepared.

Figure 9.12 shows the recognition rate plotted against the number of classes for various numbers of examples per class in the training set. As can be seen, the performance is good; 96.9% correct on 16 classes trained with 30 examples per class. Using only 15 examples per class results in a recognition rate of 96.7%.

Figure 9.13 shows all the mistakes made by the classifier. None of the mistakes appear to be a result of the size or orientation of a gesture being confused. Rather, the mistakes are quite similar to those seen previously. The conclusion in that the gesture classifier does surprisingly well on gesture sets in which some classes have variable size or orientation, while others are discriminated on the basis of their size or orientation.

9.1.5 Interuser variability

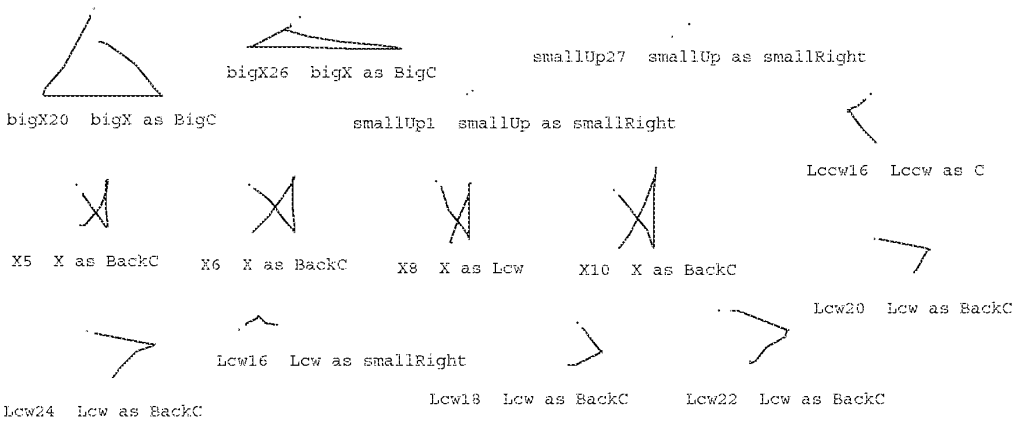


Figure 9.13: Mistakes in the variable class test

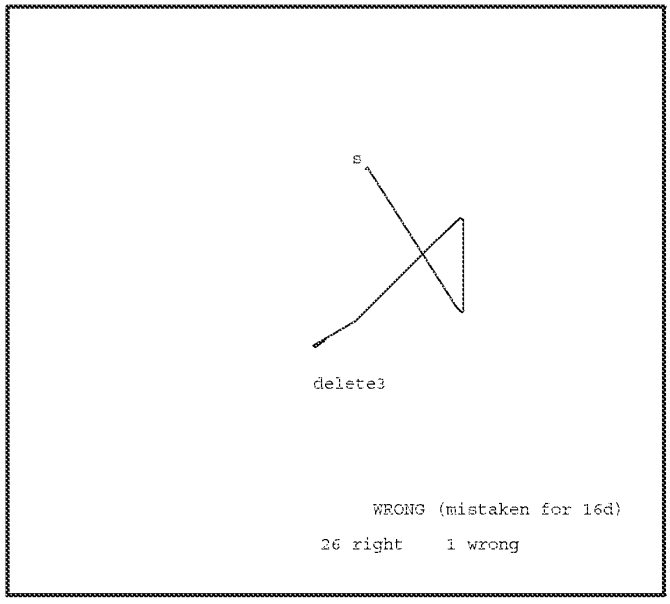


Figure 9.14: Testing program (user's gesture not shown)

All the gestures shown thus far have been those of the author. It was deemed necessary to show validity of the current work by demonstrating that the gestures of at least one other person could be recognized. Two questions come to mind: what recognition rate can be achieved when a person other than the author gestures at a classifier trained with the author's gestures, and can this rate be improved by allowing the person to train the classifier using his or her own gestures?

Setup

As preparation for someone besides the author actually using the GSCORE application (see Section 9.4.2 below), the GSCORE gesture set (figure 9.1) was used in the evaluation. The hardware used was the same hardware used in the majority of this work, a DEC MicroVAX II running UNIX and X10.

A simple testing program was prepared for training and evaluation (Figure 9.14). In a trial, a prototype gesture of a given class is randomly chosen and displayed on the screen, with the start point indicated. The user attempts to enter a gesture of the same class. That gesture is then classified, and the results fed back to the user. In training mode, if the system makes an error, the trial is repeated. In evaluation mode, each trial is independent.

Subject PV is a music professor, a professional musician, and an experienced music copyist. He is also an experienced computer user, familiar with Macintosh and NeXT computers, among others.

Procedure

The subject was given one half hour of practice with the testing program in training mode. He was also given a copy of figure 9.1 and instructed to take notes at his own discretion. After the half hour, the tester was put in evaluation mode, and two hundred trials run. The test was repeated one week later, without any warmup. The subject was then instructed to create his own gesture set, borrowing from the set he knew as much as he liked. Thirty examples of each gesture class were recorded, and two hundred evaluation trials run on the new set.

Results

During the initial training there was some confusion on the subject's part regarding which hand to use. The subject normally uses his right hand for mousing, but, being left handed, always writes music with his left. After about ten minutes, the subject opted to use his left hand for gesturing.

In the initial evaluation trial the system classified correctly 188 out of 200 gestures. The subject felt he could do better and was allowed a second run, during which 179 out of 200 gestures were correctly classified. By his own admission, he was more "cocky" during the second run, generally making the gestures faster than during the first. The average recognition rate is 91.8%.

After the test, the subject commented that he felt much of his difficulty was due to the fact that he was not used to using the mouse with his left hand, and that the particular mouse felt very different than the one he was used to (NeXT's). He felt his performance would further improve with additional practice.

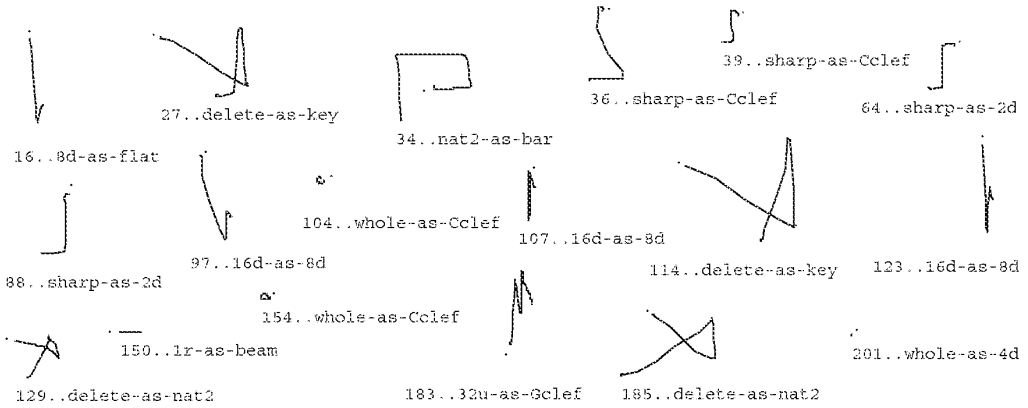


Figure 9.15: PV's misclassified gestures (author's set)

His notes are interesting. Although the subject had no particular knowledge of the recognition algorithm being used, in many cases his notes refer to the particular features used in the algorithm. For gestures whole and sharp he wrote "start up" and "don't begin too vertically" respectively, noting the importance of the correct initial angle. For 1r he wrote "make short," for bar he wrote "make large," and for delete he wrote "make quickly." For 2u and 2d he wrote "sharp angle."

The subject commented on places where the gesture classes used did not conform to standard copyist strokes. For example, he stated the loop in flat goes the wrong way. He explained that many music symbols are written with two strokes, and said that he might prefer a system that could recognize multiple-stroke symbols.

When the test was repeated a week later, the subject, without any warmup, achieved a score of 183 out of 200, 91.5%. Figure 9.15 shows the misclassified gestures. The subject was again unsure of which hand to use, but used his left hand at the urging of the author.

The subject then created his own gesture set, examples of which are shown in figure 9.16. A training set consisting of 30 examples of each class was entered. Running the training set through the resulting classifier resulted in the rather low recognition rate of 94.7% (by comparison, running the author's training set through the classifier it was used to train yielded 97.7%.) The low rate was due to the some ambiguity in the classes (*e.g.* "flat" and "16d" were frequently confused) as well as many classes where the corners were looped (as seen before in section 9.1.1), causing a bimodal distributions for f_9, f_{10}, and f_{11}

The problems in the new gesture set nonwithstanding, PV ran two hundred trials of the tester on the new set. He was able to get a score of 186 out of 200, 93%.

At the time of this writing, PV has not yet made the attempt to remove the ambiguities from the new gesture set and to be more careful on the sharp corners.

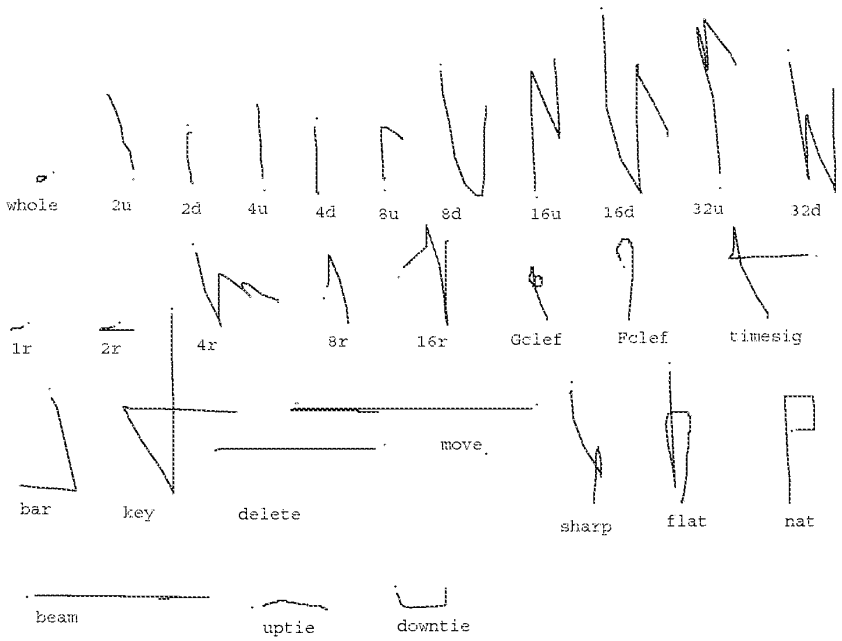


Figure 9.16: PV's gesture set

Conclusion

It is difficult to draw a conclusion given data from only one subject. The author expected the recognition rate to be higher when PV trained the system on his own gestures than when he used the author's set. The actual rate *was* slightly higher, but not enough to make a convincing argument that people do better on their own gestures (some slightly more convincing evidence is presented in section 9.4.2 below). In retrospect, PV should have created a training set that copied the author's gestures before attempting to significantly modify that gesture set. The author's gesture set turned out to be better designed than PV's, in the sense of having less inherent ambiguities; this tended to compensate for any advantage PV gained from using his own gestures.

However, PV's new gesture set is not without merit; on the contrary, it has a number of interesting gestures. The new delete gesture, a quick, long, leftward stroke, gives the user the impression of throwing objects off the side of the screen. The new move gesture is like a delete followed by a last minute change of mind. The flat gesture is much closer to the way PV writes the symbol, as are the leftward whole and half rests gestures 1r and 2r. The stylized "4" for timesig is clever, as is the way it relates to key. PV's bar gesture is much more economical that the author's.

The experiments indicate that a person can use a classifier trained on another person's gesture with moderately good results. Also indicated is that people can create interesting gesture sets on their own. Some modification to the feature set also seems desirable, mainly to make the features

less sensitive to "looped" corners. It would be useful to give more feedback to the gesture design as to which classes are confusable. This should be simple to do simply by examining the Mahalanobis distance between every pair of classes.

9.1.6 Recognition Speed

It is well known that a user interface must respond quickly in order to satisfy users; thus for gesture-based systems the speed of recognition is an important factor in the usability of the system. This section reports on measurements of the speed of the components of the recognition process.

The statistical gesture recognizer described in Chapter 3 was designed with speed in mind. Each feature is incrementally calculated in constant time; thus $O(F)$ work must be done per mouse point, where F is the number of features. Given a gesture of P mouse points, it thus takes $O(PF)$ time to compute its feature vector. The classification computes a linear evaluation function over the features for each of C classes; thus classification take $O(CF)$ time.

Feature calculation

The abstract datatype `FV` is used to encapsulate the feature calculation as follows:

`FV FvAlloc()` allocates an object of type `FV`. A classifier will generally call `FvAlloc()` only once, during program initialization.

`FvInit(fv)` initializes `fv`, an object of type `FV`. `FvInit(fv)` is called once per gesture, before any points are added.

`FvAddPoint(fv, x, y, t)` adds the point (x, y) which occurs at time t to the gesture. `FvAddPoint` performs the incremental feature calculation. It is called for every mouse point the program receives. There are thirteen features calculated ($F = 13$).

`Vector FvCalc(fv)` returns the feature vector as an array of double precision floating point numbers. It performs any necessary calculations needed to transform the incrementally calculated auxiliary features into the feature set used for classification. It is called once per gesture.

The function `CalcFeatures(g)` represents the entire work of computing the feature vector for a gesture that is in memory:

```
Fv fv;      /* allocated via FvAlloc() elsewhere */
Vector
CalcFeatures(g)
register Gesture g;
{
    register Point p;
    FvInit(fv);
    for(p = g->point; p < &g->point[g->npoints]; p++)
        FvAddPoint(fv, p->x, p->y, p->t);
```

Processor	Time(sec)	Relative Speed
MicroVAX II	227.95	0.76
VAX 11/780	172.20	1.0
MicroVAX III	60.97	2.8
PMAX-3100	11.30	15

Table 9.1: Speed of various computers used for testing

Processor	milliseconds per call		
	`FvAddPoint`	`FvCalc`	`CalcFeatures`
MicroVAX II	0.22	0.34	3.9
MicroVAX III	0.074	0.13	1.3
PMAX-3100	0.029	0.040	0.44

Table 9.2: Speed of feature calculation

```
    return FvCalc(fv);
}
```

To obtain the timings, the testing set of Section 9.1.1 was read into memory, and then each gesture was passed to `CalcFeatures`. Three processors were used: the DEC MicroVAX II that was used for the majority of the work reported in this dissertation, a DEC MicroVAX III, and a DEC PMAX-3100 (to get an idea of the performance on a more modern system). The UNIX profiling tool was used to obtain the times. In all cases, the times are virtual times, *i.e.* the time spent executing the program by the processor. All tests were run on unloaded systems, and the real times were never more than 10% more than the virtual times.

Before timing any code related to gesture recognition, the following code fragment (compiled with "cc -O") was timed on a number of processors, MicroVAX II, VAX 11/780, MicroVAX III, and PMAX-3100, in order to compare the speed of the processors used in the following tests to that of a VAX 11/780:

```
register int  i, n = 1000000;
double s, a[15], b[15];
for(i = 0; i < 15; i++) a[i] = i, b[i] = i*i;
do {
    s = 0.0;
    for(i = 0; i < 15; i++) s += a[i] * b[i];
} while(--n);
```

The times for the above fragment shown in table 9.1.

Note that on this code fragment the PMAX-3100 runs about 20 times faster than the MicroVAX II. On more typical code, it usually runs only 10-15 times faster.

The testing set averaged 13.4 points per gesture. The timings for the routines that calculate features are shown in table 9.2.

The cost per mouse point to incrementally process a mouse point is a small fraction of a millisecond, even on the slowest processor. Since mouse points typically come no faster than 40

Processor	Computation time (milliseconds)				
	$v^{\hat{c}}$ (one class)	max $v^{\hat{c}}$ (30 classes)	$\tilde{P}$	d^2	total
MicroVAX II	0.27	8.0	0.8	3.7	12.6
MicroVAX III	0.074	2.2	0.3	1.1	3.6
PMAX-3100	0.022	0.66	0.01	0.26	0.99

Table 9.3: Speed of Classification

per second, only a small fraction of the processor is consumed incrementally calculating the feature vector. Indeed, substantially more of the processor is consumed communicating with the window manager to receive the mouse point and perform the inking.

Classification

Once the feature vector is calculated it must be classified. This involves computing a linear evaluation function $v^{\hat{c}}$ on F features ($F = 13$) for each of C classes. If the rejection parameters are desired, it takes an additional $O(C)$ work to estimate the ambiguity $\tilde{P}$ and $O(F^2)$ work to estimate the Mahalanobis distance d^2. The computation times for each of these is shown in table 9.3.

To get these times, four runs were made. Every gesture in the testing set was classified in every run. The first run did not calculate either rejection parameter. The average time to classify a gesture as one of thirty classes is reported the *max*$v^{\hat{c}}$ column; the $v^{\hat{c}}$ column is computed as $\frac{1}{30}$ of that time. (The $v^{\hat{c}}$ column thus gives the time to compute the evaluation function for a single class; multiply this by the number of classes to estimate the classification time of a particular classifier.) The second run computed $\tilde{P}$ after each classification; the difference between that time and the max $v^{\hat{c}}$ time is reported in the $\tilde{P}$ column. The third run computed d^2 and is reported similarly. The fourth run computed both $\tilde{P}$ and d^2; the average time per gesture is reported in the "total" column.

For a 30-class discrimination with both rejection parameters being used, after the last mouse point of a gesture is entered it takes a MicroVAX II 13 milliseconds to finish calculating the feature vector (`FvCalc`) and then classify it. This is acceptable, albeit not fantastic, performance. If the end of the gesture is indicated by no mouse motion for a timeout interval, the classification can begin before the timeout interval expires, and the result be ignored if the user moves the mouse before the interval is up.

Currently, all arithmetic is done using double precision floating point numbers. There is no conceptual reason that the evaluation functions could not be computed using integer arithmetic, after suitably rescaling the features so as not too lose much precision. The resulting classifier would then run much faster (on most processors). This has not been tried in the present work.

If eager recognition is running, classification must occur at every mouse point, and the number of classes is $2C$. This puts a ceiling on the number of the classes that the eager recognizer can discriminate between in real-time. On a MicroVAX II, the cost per mouse point includes `FvAddPoint` (0.22 msec) plus `FvCalc` (0.34 msec) plus the per class evaluation of $2C$ classes, $0.54C$. If mouse points come at a maximum rate of one every 25 milliseconds, $C = 45$ classes would consume the entire processor. Practically, since there is other work to do (*e.g.* inking), $C = 20$ is

probably the maximum that can be reasonably expected from an eager recognizer on a MicroVAX II. On today's processors, instead of computation time, the limiting factor will be the lower recognition rate when given a large number of classes.

One approach tried to increase the number of classes in eager recognizers was to use only a subset of the features. While this improved the response time of the system, the performance degraded significantly, so the idea was abandoned. There is no point getting the wrong answer quickly.

9.1.7 Training Time

The stated goal of the thesis work is to provide tools that allow user interface designers to experiment with gesture-based systems. One factor impacting on the usability of such tools is the amount of time it takes for gesture recognizers to retrain themselves after changes have been made to the training examples. In almost all trainable character recognizers, deleting even a single training example requires that the training be redone from scratch. For some technologies, notably neural networks, this retraining may take minutes or even hours. Such a system would not be conducive to experimenting with different gesture sets.

By contrast, statistical classifiers of the kind described in Chapter 3 can be trained very rapidly. Training the classifier from scratch requires $O(EF)$ to compute the mean feature vectors, $O(EF^2)$ time to calculate the per-class covariance matrices, $O(CF^2)$ to average them, $O(F^3)$ to invert the average, and $O(CF^2)$ to compute the weights used in the evaluation functions. If the average covariance matrix is singular, an $O(F^4)$ algorithm is run to deal with the problem.

Often, a fair amount of work can be reused in retraining after a change to some training examples. Adding or deleting an example of a class requires $O(F)$ work to incrementally update its per-class class mean vector, and $O(F^2)$ work to incrementally update its per-class covariance matrix [137]. Retraining then involves repeating the steps starting from computing the average covariance matrix. Thus, for retraining, the dependency on E, the total number of examples, is eliminated. The retraining time is instead a function of the number of examples added or deleted.

The Objective C implementation does not attempt to incrementally update the per-class covariance matrix when an example is added. Instead, only the averages are kept incrementally, and the per-class covariance matrix is recomputed from scratch. This involves $O(E^c F^2)$ work for each class c changed, where E^c is the number of training examples for class c. This results in worse performance when a small number of examples are changed, but better performance when all the examples of a class are deleted and a new set entered. The latter operation is common when experimenting with gesture-based systems.

The author has implemented both C and Objective C versions of the single path classifier. Besides maintaining the per-class covariance matrices incrementally, the C version differs in that it does not store the list of examples that have been used to train it. (It is not necessary to store the list to add and remove examples, since the mean vector and covariance matrix are updated incrementally.) It is thus more efficient since it does not need to maintain the lists of examples. (Objective C's `Set` class, implemented via hashing, is used to maintain the lists in that version.) It also does not have the overhead of separate objects for examples, classes and classifiers that the Objective C version has (see Section 7.5).

Processor	Time (milliseconds per call)			
	`sAddExample`	`sRemoveExample`	`sDoneAdding` (10 classes)	`sDoneAdding` (30 classes)
MicroVAX II	3.7	3.8	130	234
MicroVAX III	0.90	0.90	43	78
PMAX-3100	0.024	0.026	14	22

Table 9.4: Speed of classifier training

Since only the C version could be ported to the PMAX-3100, it was used for the timings. (C versions of the feature computation and gesture recognition were used for the timings above; however in these cases the Objective C methods are straightforward translations of their corresponding C functions. In some cases, the methods merely call the corresponding C function.) The following C functions encapsulate the process of training a classifier:

`sClassifier sNewClassifier()` allocates and returns a handle to a new classifier. Initially it has no classes and no examples. The "s" at the beginning of the type and function names refers to "single-path"; there are corresponding types and functions for the multi-path classifiers.

`sAddExample(sClassifier sc, char *classname, Vector e)` adds the training example (feature vector `e`) to the named class `classname` in the passed classifier. The class is created if it has not been seen before. Linear search is used to find the class name; however, it is optimized for successive calls with the same name. The `sAddExample` function incrementally maintains the per-class mean vectors and covariance matrices.

`sRemoveExample(sClassifier sc, char *classname, Vector e)` removes example `e`, assumed to have been added earlier, from the named class. The per-class mean vector and covariance matrix are incrementally updated.

`sDoneAdding(sClassifier sc)` trains the classifier on its current set of examples. It computes the average covariance matrix, inverts it (fixing it if singular), and computes the weights.

`sClass sClassify(sClassifier sc, Vector e, double *p, *d2)` actually performs the classification of `e`. If `p` is non-`NULL` the probability of ambiguity is estimated; if `d2` is non-`NULL` the estimated Mahalanobis distance of `e` to its computed class is returned. This is the function timed in the previous section.

The functions were exercised first by adding every example in the training set, training the classifier, and then looping, removing and then re-adding 10 consecutive examples before retraining. No singular covariance matrix was encounted, due to the large number of examples. Table 9.4 shows the performance of the various routines.

Even on a MicroVAX II, training a 30 class classifier once all the examples have been entered takes less than a quarter second. Thus GRANDMA is able to produce a classifier immediately the

first time a gesture is made over a set of views whose combined gesture set has not been encountered before (see Sections 7.2.2 and 7.4). The user has to wait, but does not have to wait long.

9.2 Eager recognition

This section evaluates the effectiveness of the eager recognition algorithm on several single-stroke gesture sets. Recall that eager recognition is the recognition of a gesture while it is being made, without any explicit indication of the end of the gesture. Ideally, the eager recognizer classifies a gesture as soon as enough of it has been seen to do so unambiguously (see Chapter 4).

In order to determine how well the eager recognition algorithm works, an eager recognizer was created to classify the eight gestures classes shown in 9.17. Each class named for the direction of its two segments, e.g. ur means "up, right." Each of these gestures is ambiguous along its initial segment, and becomes unambiguous once the corner is turned and the second segment begun.

The eager recognizer was trained with ten examples of each of the eight classes, and tested on thirty examples of each class. The figure shows ten of the thirty test examples for each class, and includes all the examples that were misclassified.

Two comparisons are of interest for the gesture set: the eager recognition rate versus the recognition rate of the full classifier, and the eagerness of the recognizer versus the maximum possible eagerness. The eager recognizer classified 97.0% of the gestures correctly, compared to 99.2% correct for the full classifier. Most of the eager recognizer's errors were due to a corner looping 270 degrees rather than being a sharp 90 degrees, so it appeared to the eager recognizer the second stroke was going in the opposite direction than intended. In the figure, "E" indicates a gesture misclassified by the eager recognizer, and "F" indicates a misclassification by the full classifier.

On the average, the eager recognizer examined 67.9% of the mouse points of each gesture before deciding the gesture was unambiguous. By hand, the author determined for each gesture the number of mouse points from the start through the corner turn, and concluded that on the average 59.4% of the mouse points of each gesture needed to be seen before the gesture could be unambiguously classified. The parts of each gesture at which unambiguous classification could have occurred but did not are indicated in the figure by thick lines.

Figure 9.18 shows the performance of the eager recognizer on GDP gestures. The eager recognizer was trained with 10 examples of each of 11 gesture classes, and tested on 30 examples of each class, five of which are shown in the figure. The GDP gesture set was slightly altered to increase eagerness: the group gesture was trained clockwise because when it was counterclockwise it prevented the copy gesture from ever being eagerly recognized. For the GDP gestures, the full classifier had a 99.7% correct recognition rate as compared with 93.5% for the eager recognizer. On the average 60.5% of each gesture was examined by the eager recognizer before classification occurred. For this set no attempt was made to determine the minimum average gesture percentage that needed to be seen for unambiguous classification.

From these tests we can conclude that the trainable eager recognition algorithm performs acceptably but there is plenty of room for improvement, both in the recognition rate and the amount of eagerness.

Computationally, eager recognition is quite tractable on modest hardware. A fixed amount of

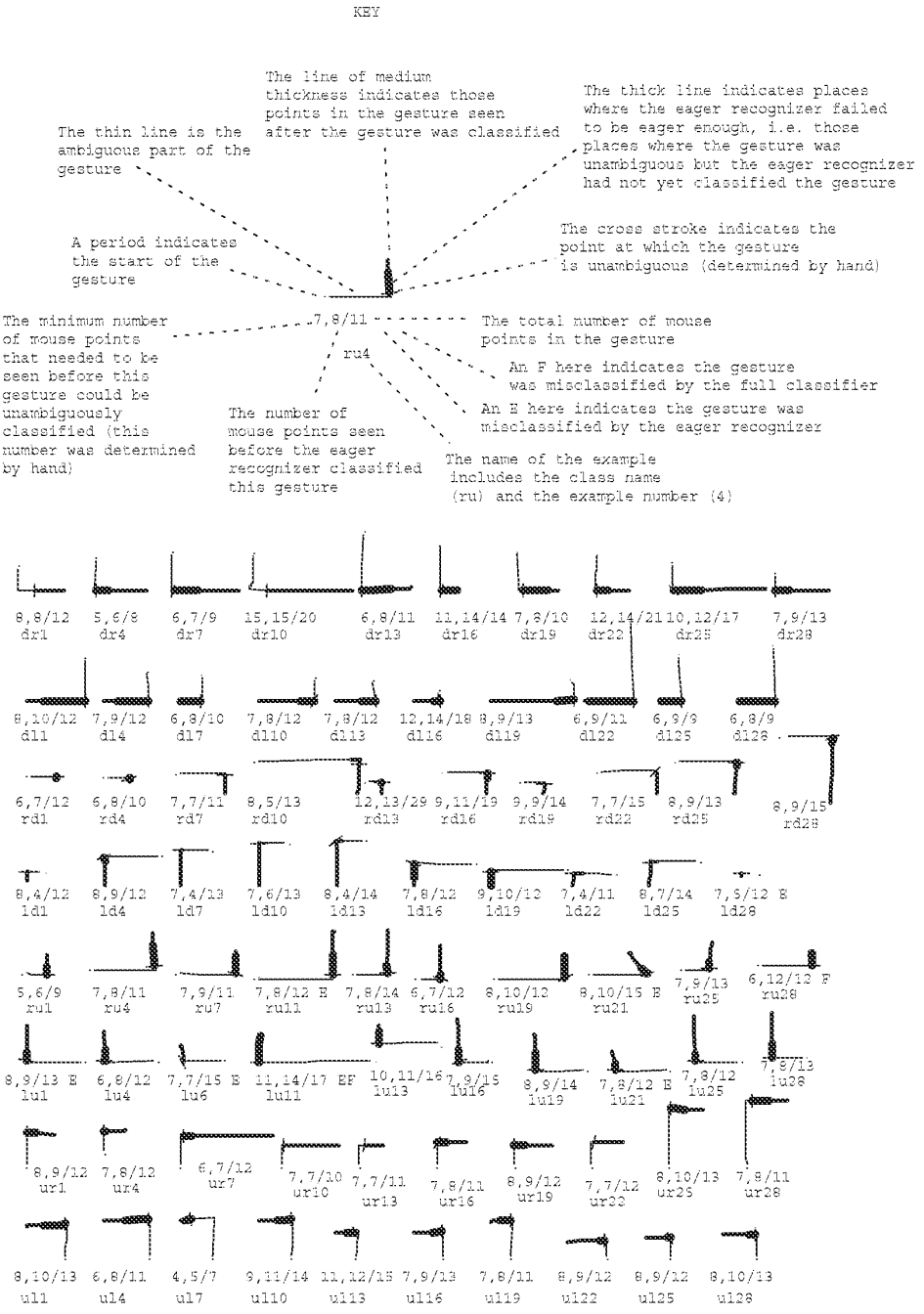


Figure 9.17: The performance of the eager recognizer on easily understood data

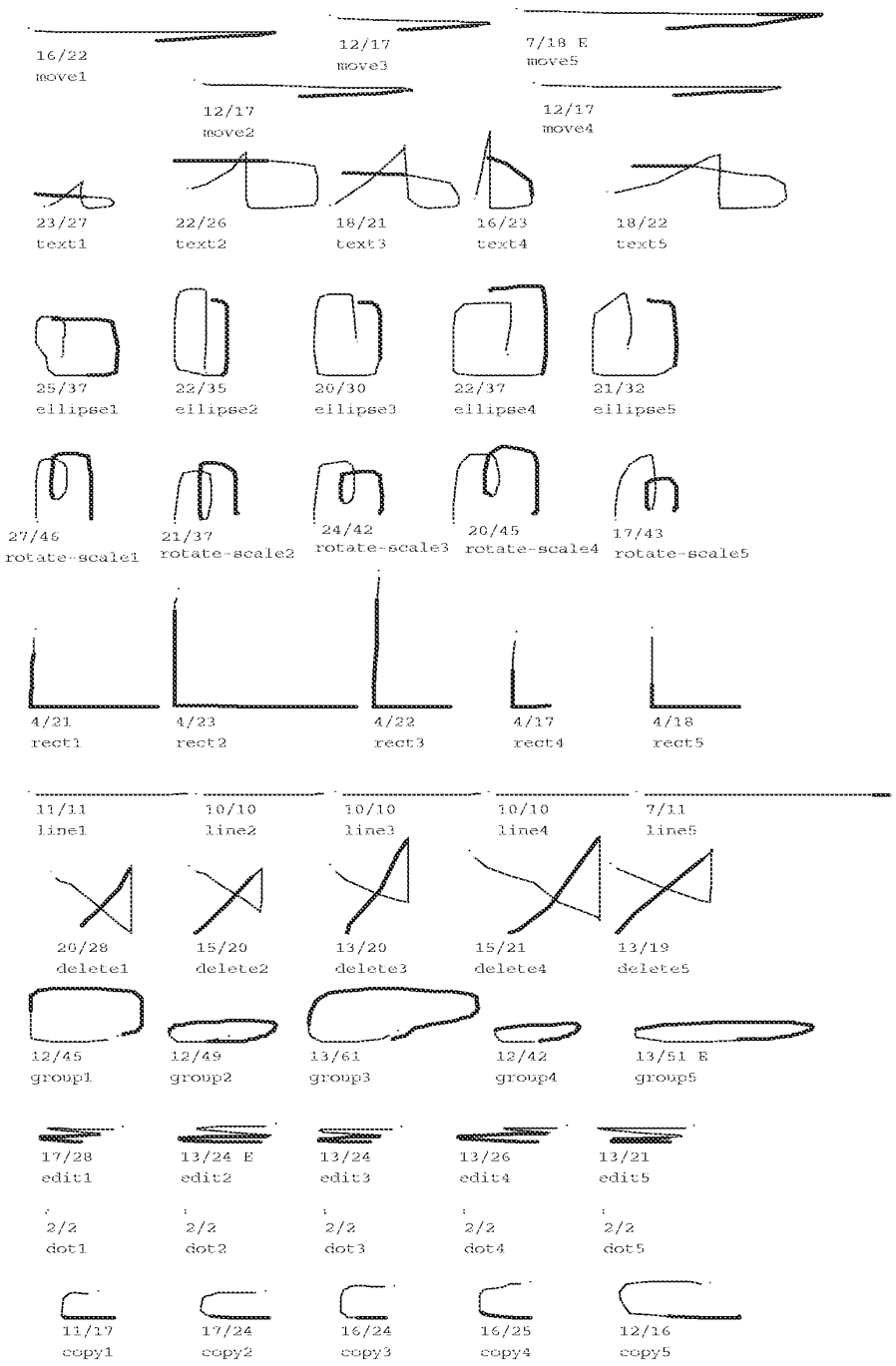


Figure 9.18: The performance of the eager recognizer on GDP gestures

The transitions from thin to thick lines indicate where eager recognition occurred.

computation needs to occur on each mouse point: first the feature vector must be updated (taking 0.5 msec on a DEC MicroVAX II), and then the vector must be classified by the AUC (taking 0.27 msec per class, or 6 msec in the case of GDP).

9.3 Multi-finger recognition

Multi-finger gestural input is a significant innovation of this work. Unfortunately, circumstances have conspired to make the evaluation of multi-finger recognition both impossible and irrelevant. The Sensor Frame is the only input device upon which the multi-finger recognition algorithm was tested. Unfortunately, there is only one functioning Sensor Frame in existence, and that was damaged sometime after the multi-finger recognition was running, but before formal testing could begin. (Fortunately, a videotape of MDP in action was made while the Sensor Frame was working.) No progress was made in repairing the Sensor Frame for over a year; testing was thus impossible. Eventually the Sensor Frame was repaired, but Sensor Frame, Inc. went out of business shortly afterward, making any detailed evaluation irrelevant. The owner of the Sensor Frame has left the country, taking the device with him.

An informal estimate of the multi-finger recognition accuracy may be estimated from ten minutes of videotape of the author using MDP. This version of MDP uses the path sorting multi-finger recognition algorithm (Section 5.2). As shown in figure 8.10, MDP recognizes 11 gestures (6 one finger gestures, 3 two finger gestures, and 2 three finger gestures). In the videotape, the author made 30 gestures, 2 of which appear to have been misclassified, and one of which was rejected, resulting in a correct recognition rate of 90%. The processing time appears to be negligible.

All three misclassifications are the result of the Sensor Frame seeing more fingers in the gesture than were intended. This was due to knuckles of fingers curled up (so as not to be used in the gesture) accidentally penetrating the sensing plane and being counted as additional fingers. As there are distinct classifiers for single finger, two finger, and three finger gestures, an incorrect number of fingers inevitably leads to a misclassification. While it is possible to imagine methods for dealing with such errors during recognition, the main cause of this problem is the ergonomics of the Sensor Frame.

For the small gesture set examined, the recognition rate is 100% once the errors due to "extra fingers" are eliminated. This is to be expected, given the small number of gestures for each number of fingers. It is expected that the multi-path classifier operating on one finger gestures would perform about as well as the single-path classifier, as the algorithms are essentially identical. The single-path classifier, when given only six classes to discriminate among, has been shown (on mouse data) to perform at 100% in almost all cases. When operating on two finger gestures, it is expected that the performance of the recognition algorithm would be similar to that of the single-path classifier on *twice* the number of classes. Actually, it is possible that some of the paths in the two-finger gestures will be similar to other paths in the set, and be merged into a single class by the training algorithm (Section 5.4). Thus, when the number of unique paths will be less than twice the number of two-finger gesture classes, performance may be expected to improve accordingly. Similarly, the three finger gesture classifier may be expected to perform as well as a single-path classifier the recognizes between one and three times the number of three finger gesture classes, depending on

the number of unique paths in the class set.

One more factor to consider is that mouse data tends to be much less noisy than Sensor Frame data. The triangulation by the Sensor Frame is erratic, especially when multiple fingers are being tracked. For example, both horizontal segments of the Parallelogram gesture of figure 8.10 should be straight lines. Until this problem can be solved, it is expected that recognition rates for Sensor Frame gesture sets will suffer.

9.4 GRANDMA

Evaluating GRANDMA is much more subjective than evaluating the low-level recognition rates. GRANDMA may be evaluated on several levels: the effort required to build new interaction techniques, to build new applications, to add gestures to an application, to change an application's gestures, or to use an application to perform a task.

No attempt was made to formally evaluate any of these. In order to get statistically valid results, it would have been necessary to run carefully designed experiments on a number of users, something the author had neither the time, space, inclination, or qualifications to do. Furthermore, the author does not wish to claim that GRANDMA is superior to existing object-oriented toolkits for any particular task. GRANDMA is simply the platform through which some ideas for input processing in object-oriented toolkits were explored. GRANDMA's significance, if any, will be its influence on future toolkits, rather than any more direct results.

Nonetheless, this section informally reports on the author's experience building gesture-based systems with GRANDMA. (No one besides the author tried to program with GRANMDA.) Also, in order to confirm that GRANDMA can be used by someone other than the author, this section also reports on observations of a subject trying to use GSCORE and GRANDMA to due some tasks.

9.4.1 The author's experience with GRANDMA

GRANDMA took approximately seven months to design and develop. It consists of approximately 12000 lines of Objective C code. There are an additional 5000 lines of C code which implement a graphics layer as well as the feature vector calculation. GDP took an additional 2000 lines of Objective C code to implement. GDP was developed at the same time as GRANDMA, as it was the primary application used to test GRANDMA.

Initially, only two GDP gestures were used to test GRANDMA's gesture handler and associated utilities. Once these were working well, it took four days to add the remaining gestures to GDP. Most of this time was spent writing Objective C methods to use in semantic expressions. These were methods that were not needed for the existing direct manipulation interface.

GSCORE consists of 6000 lines of Objective C code. It took six weeks to design and implement GSCORE, including its palette-based interface. Much of this time was spent on the details of representing common music notation, mechanisms for displaying music notation, and producing usable music fonts. The palette, an interaction technique that did not as yet exist in GRANDMA, took about eight hours to implement. It took two weeks to add the gestural interface to GSCORE,

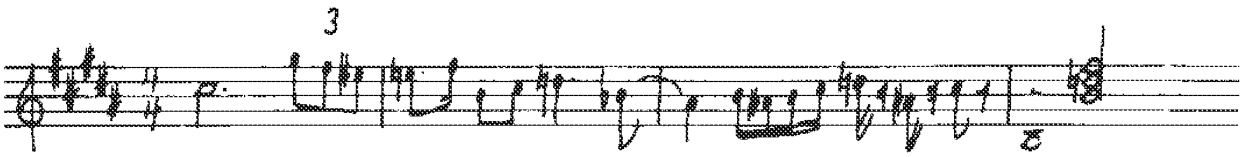


Figure 9.19: PV's task

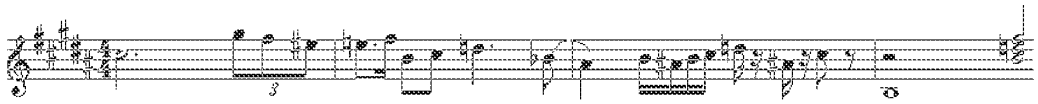


Figure 9.20: PV's result

including writing some additional methods. Much of this time was spent experimenting with different semantics for the gestures.

Section 10.1.3 lists features of GRANDMA that will be important to incorporate into future toolkits that support gestures.

9.4.2 A user uses GSCORE and GRANDMA

This sections informally reports on subject PV's (see Section 9.1.5) attempts to use the GSCORE program.

The task was to enter the music shown in figure 9.19. The music was chosen to exercise many of the GSCORE gestures. PV is an experienced music copyist, and it took him 100 seconds to write out the music as shown, copying it from an earlier attempt.

Using gestures, the author was able to enter the above score in 280 seconds (almost five minutes). He made a total of 53 gestures, four of which did not give the desired results and were immediately undone. Only two of those were misclassifications; the other two were notes gestures where the note was created having the wrong pitch, due to misplacement of the cursor at the start of the gesture. Turning off gestures and using only the palette interface, it took the author 670 seconds (eleven minutes). No mistakes needed to be undone in the latter trial.

PV's first attempt was at using the GSCORE program trained with the author's gestures. PV had already gained experienced with this set of gestures during the study of interuser variation. PV practiced for one half hour with the GSCORE program before attempting the task. The author coached PV during this time, as no other documentation or help was available.

PV took 600 seconds (10 minutes) to complete the task. He made a total of 73 gestures, 16 of which were immediately undone. It appeared to the author, who was silently observing, that each undo was used to recover from a misclassification. Figure 9.20 shows the product of his labor. PV then turned off gestures, and used the palette interface to enter the example. He completed the task in 680 seconds (11 minutes).

PV then entered his own gestures in place of some of the author's. In particular, he substituted his own gestures for the nine classes: delete, move, beam, 1r, 2r, 8r, 16r, keysig, and bar, entering 15 or more examples of each. The total time to do this, including incremental testing of the new gestures and periodic saves to disk, was 25 minutes. PV did not attempt to emulate the author's gestures; instead, he used the forms he had created earlier (see Section 9.1.5).

Once done, PV repeated the experiment. It took him 310 seconds (5 minutes) to enter the music. He made 58 gestures, 4 of which were undone.

PV was interviewed after the tests, and made the following comments: The biggest problem, he stated, was that mouse tracking in the GSCORE program was much more sluggish than in the recorder and tester. (This is accurate, as the time required for GSCORE events to be created and consumed adds significant overhead to the mouse tracking. Much of this is overhead due to GRANDMA.) PV characterized the system as "sluggish." Bad tracking, especially at the start of the gesture, contributed significantly to the number of misclassifications.

PV stated that he thought the system "intuitive" for entering notes. He described the gesture-based interface as "excellent" compared to the palette-based system, but when asked how the gesture-based interface compared to writing on paper, he replied "it sucks." He did not like using the mouse for gesturing, and believed that a stylus and tablet would be much better.

It is again difficult to draw conclusions from an informal study of one user. Did PV's performance improve because he tailored the gestures to his liking, or because he had been practicing? This is unknown. Some things are clear: GRANDMA makes it easy to experiment with new gesture sets, and, in GSCORE, with moderate practice the gesture-based interface improved task performance by a factor of two over the palette-based interface. Whether gesture-based interfaces generally improve task performance over non-gesture-based interfaces is a question that requires *much* further study.

Chapter 10

Conclusion and Future Directions

This chapter summarizes the contributions of this thesis and indicates some directions for future work.

10.1 Contributions

This thesis makes contributions in four areas:

- New interaction techniques
- New recognition-related algorithms
- Integrating gestures into interfaces
- Input in object-oriented toolkits

Each of these will be discussed in turn.

10.1.1 New interactions techniques

A major contributions of this work has been the invention and exploration of three new interaction techniques.

The two-phase single-stroke interaction The two-phase interaction enables gesture and direct manipulation to be integrated in a single interaction that combines the power of each. The first phase is collection, during which the points that make up the gesture are collected. In the simplest case, the end of the collection phase is indicated by a motion timeout, classification occurs, and the second phase, manipulation, is entered. In the manipulation phase, the user moves the mouse to manipulate some parameters in the application. The particular parameters manipulated depend on the classification of the collected gesture. The collection phase is like character entry in handwriting interfaces; the manipulation phase is like a drag interaction in direct-manipulation interfaces. Generally, the operation, operands, and some parameters are

determined at the phase transition (when the gesture is recognized), and then the manipulation phase allows additional parameters to be set in the presence of application feedback.

Eager recognition Eager recognition is a modification of the two-phase single-stroke interaction in which the phase transition from gesturing to manipulation occurs as soon as enough of the gesture has been seen so that it may be classified unambiguously. The result is an interaction that combines gesturing and direct manipulation in a single, smooth interaction.

The two-phase multiple-finger interaction Gesture and direct manipulation may be combined for multiple path inputs in a way similar to the two-phase single-stroke interaction. With multiple finger input, opportunities exist for expanding the power of each phase of the interaction. By allowing multiple fingers in the collection phase, the repertoire of possible gestures is greatly increased, and a multiple finger gesture allows many parameters to be specified simultaneously when the gesture is recognized. Similarly, even when only one finger is used for the gesture, additional fingers may be brought in during the manipulation phase. Thus, the two-phase multiple-finger interaction allows a large number of parameters to be specified and interactively manipulated.

10.1.2 Recognition Technology

This thesis discloses five new algorithms of general utility in the construction and use of gesture recognizers.

Automatic generation of single-stroke gesture recognizers from training examples A practical and efficient algorithm for generating gesture recognizers has been developed and tested. In it, a gesture is represented as a vector of real-valued features, and a standard pattern recognition technique is used to generate a linear classifier that discriminates between the vectors of different gesture classes. The training algorithm depends on aggregate statistics of each gesture class, and empirically it has been shown that usually only fifteen examples of each class are needed to produce accurate recognizers. It is simple to incorporate dynamic attributes, such as the average speed of the gesture, into the feature set. The algorithm has been shown to work even when some classes vary in size and orientation while others depend on size or orientation to be recognized. The recognizer size is independent of the number of training examples, and both the recognition and training times have been shown to be small. A features set that is both meaningful and extensible potentially allows the algorithm to be adapted to future input devices and requirements.

Incremental feature calculation The calculation used to generate features from the input points of a gesture is incremental, meaning that it takes constant time to update the features given a new input point. This allows arbitrarily large gestures to be processed with no delay in processing.

Rejection algorithms Two algorithms for rejecting ill-formed gestures have been developed and tested. One estimates the probability of correct classification, enabling input gestures that are ambiguous with respect to a set of gesture classes to be rejected. The other uses a normalized

distance metric to determine how close an input gesture is to the typical gesture of the its computed class, allowing outliers to be rejected.

Automatic generation of eager recognizers from training examples An eager recognizer classifies a gesture as soon as it is unambiguous, alleviating the need for the end of the gesture to be explicitly indicated. An algorithm for generating eager recognizers from training examples has been developed and tested. The algorithm produces a two-class classifier which is run on every input point and used to determine if the gesture being entered is unambiguous.

Automatic generation of multi-path gesture recognizers The single-stroke recognition work has been extended so that a number of single-stroke recognizers may be combined into a multi-finger gesture recognizer. The described algorithm produces a multi-path recognizer given training examples. Relative path timing information is considered during the recognition, and global classification is attempted when the individual path classifications do not uniquely determine the class of the multi-path gesture. For dealing with the problems that arise from multi-path input devices that do not *a priori* determine "which path is which," two approaches, path sorting and path clustering, have been explored. The resulting algorithm has been demonstrated using the Sensor Frame as a multi-finger input device.

10.1.3 Integrating gestures into interfaces

A paradigm for integrating gestures into object-oriented interfaces has been developed and demonstrated. The key points are:

A gesture set is associated with a view or view class. Each class of object in the user interface potentially responds to a different set of gestures. Thus, for example, notes respond to a different set of gestures than staves in the GSCORE music editor.

The gesture set is dynamically determined. From the first point of a gesture, the system dynamically determines the set of gestures possible. The first point determines the possible views at which the gesture is directed. For each of these views, inheritance up the class hierarchy determines the set of gestures it handles. These sets are combined, and if need be, a classifier for the resulting union is dynamically created.

The gesture class and attributes map to an application operation, operands, and parameters. Gestures are powerful because they contain additional information beyond the class of the gesture. The attributes of a gesture, such as size, orientation, length, speed, first point, and enclosed area, can all be mapped to parameters (including operands) of application routines. In the two-phase interaction, after the gesture is recognized there is an opportunity to map subsequent input to application parameters in the presence of application feedback.

Gesture handlers may be manipulated at runtime. In order to encourage exploration of gesture-based systems, all aspects of the gestural interface can be specified while the application is running. A new gesture handler may be created at runtime and associated with one or more views or view classes. Gesture classes may be added, deleted, or copied from other handlers.

Examples of each gesture class can be entered and modified at runtime. Finally, the semantics of the gesture class can be entered and modified at runtime. Three semantic expressions are specifiable: one evaluated when the gesture is first recognized, one evaluated on each subsequent mouse point, and one evaluated when the interaction completes.

10.1.4 Input in Object-Oriented User Interface Toolkits

A number of new ideas in the area of input in object-oriented user interface toolkits arose in the course of this work.

Passive and active event handlers A single passive event handler may be associated with multiple views. When input occurs on one such view, the handler usually activates a copy of itself. Thus, the active/passive dichotomy eliminates the need to have a controller object instantiated for each view that expects input, a major expense in many MVC systems.

Event handlers may be associated with view classes Instead of having to associate a handler with every instance of a view, the handler may be associated with one or more view classes. A view may have multiple handlers associated with it, and handlers are queried in a specific order to determine which handler will handle particular input.

Unified mouse input and virtual tools All input devices are tools, but when desired a single input device may at times be different tools, one way to implement modes in the interface. Tools may also be software objects, and some views are indeed such virtual tools. Tools often have an action, which allows them to operate on any views that respond to that action. The test of whether a given view responds to a given tool is made by an event handler associated with every view; this allows semantic feedback to occur automatically without any explicit action on the part of the view or the tool.

Automatic semantic feedback As just mentioned, the feedback as to whether a given tool operates upon a view over which it is has been dragged happens automatically. For example, objects that respond to the delete message will automatically highlight when a delete tool is dragged over them. If desired, an object can do more elaborate processing to determine if it truly responds to a given tool, e.g. an object may check that the user has permission to delete it before indicating it responds to the delete tool.

Runtime creation and manipulation of event handlers Event handlers may be created and associated with views or view classes at runtime. For example, a drag handler may be associated with an object, allowing that object to be dragged (i.e. have its position changed). In addition, such handlers may be modified at runtime, for example, to change the predicate that activates the handler.

10.2 Future Directions

In this section, directions for future work are discussed. These directions include remedies for deficiencies of the current work as well as extensions.

The single-stroke training and recognition algorithm is the most robust and well-tested part of the current work, and even in its current form it is probably suitable for commercial applications. However, a number of simple modifications should improve performance. Sections 9.1.1 and 9.1.5 contain suggestions for additional features as well as modifications to existing features; these should be implemented. Tracking the mouse in the presence of paging has proved to be a problem, and a significant improvement in recognition rate would be achieved if real-time response to mouse events could be guaranteed.

It should be simple to extend the algorithm to three dimensional gestures. All that would be required would be to add several more features to capture motion in the extra dimension. The training algorithm and linear classifier would be untouched by this extension.

Alternatives for rejection should be explored further. The estimated probability of ambiguity is useful, though using it will always result in rejections of about as many gestures that would have been correctly classified as not. The estimated Mahalanobis distance based on the common covariance matrix is really only useful for rejecting deliberately garbled gestures. The Mahalanobis distance based on the per-class covariance matrix fares somewhat better, but requires significantly more training examples to work well.

Given the obvious false assumption of equal per-class covariance matrices, it seems that the statistical classifier should not perform well on gesture sets, some classes of which vary in size and orientation, others of which do not. In practice, when the gesture classes are unambiguous, the classifiers have tended to perform admirably. Presumably this would not be the case for all such gesture sets. One area for exploration is a method for calculating the common covariance matrix differently, in particular, by not weighing the per class contributions by the number of examples of that class.

Another challenge would be to handle such gesture sets without giving up linear classification with a closed form training formula. There seems to be only one candidate, which relies on the multiclass minimum squared error and the pseudoinverse of the matrix of examples [30]. It should be explored as a potential alternative to classifiers that rely on estimates of a common covariance matrix.

It would be interesting to explore the possibility of allowing the user to indicate declaratively that a given gesture classes will vary in size and/or orientation. This might be handled simply by generating additional training examples by varying the user-supplied examples accordingly. Alternatively, it may be possible to augment the training algorithm so that the evaluation functions for certain classes are constrained to ignore certain features.

Relaxing the requirement that a closed form exist for the per-class feature weights allows iterative training methods to be considered. They have been ignored in this dissertation since they are expensive in training time and tend to require many training examples. However, as processor speed increases, iterative methods become more practical for use in a tool for experimenting with gesture-based interfaces.

Similarly, relaxing the requirement that the classifier be a linear discriminator opens the door for many other possibilities. Quadratic discrimination, and various non-parametric discrimination algorithms are but a few. These too are expensive and require many training examples.

Perhaps recognition technologies that require expensive training may be used in a production

system while the cheaper technology developed here used for prototyping. This is analogous to using a fast compiler for development and an optimizing compiler for production. At the time of this writing it seems likely that neural networks will soon be in common use, and gesture recognition is but one application.

Additional attention should be given to the problem of detecting ambiguous sets of gesture classes and useless features. The triangular matrix of Mahalanobis distance between each pair of gesture classes is a useful starting point for determining similar gesture classes. Multivariate analysis of variance techniques [74] can determine which features contribute to the classification and which features are irrelevant. These techniques can be used to support the design of new features.

Eager recognition needs to be explored further. The classifiers generated by the algorithm of Chapter 4 are less eager than they could possibly be, due to the conservative choices being made. Hand labeling of ambiguous and unambiguous subgestures should be explored more fully; it is not difficult to imagine an interface that makes such labeling relatively painless, and it is likely to give better results than the current automatic labeling. Another possible improvement comes from the observation that, during eager recognition, the full classifier is being used to classify subgestures, upon which it was not trained. It might be worth trying to retrain the full classifier on the complete subgestures. Even better, perhaps a new classifier, trained on the *newly* complete subgestures (*i.e.* those made complete by their last point), should be substituted for the full classifier. Also, eager recognition needs to be extended to multi-path gestures.

Algorithms for automatically determining the start of a gesture would also be useful, especially for devices without any discrete signaling capability (most notably the DataGlove). In the current work, gestures are considered atomic, essentially having no discernible structure. It is easy to imagine separate gestures such as select, copy, move, and delete that are concatenated to make single interactions: select and move, select and delete. This raises the segmentation question: when does one gesture end and the next begin? Specifying allowable combinations of gestures opens up the possibility of gesture grammars, an interesting area for future study.

This dissertation has concentrated on single-path gestures that are restricted to be single strokes, for reasons explained previously. The utility of multiple-stroke gestures needs to be examined more thoroughly. In a multiple-stroke gesture, does the relaxation between strokes ruin the correspondence between mental and physical tension that makes for good interaction? Does the need for segmentation make the system less responsive than it otherwise might be? Can a manipulation phase and eager recognition be incorporated into a system based on multiple-stroke gestures? These questions require further research.

Due to the interest in multiple stroke recognition, the question arises as to whether the single-stroke algorithm can be extended to handle multiple stroke gestures. First, the segmentation problem (grouping strokes into gestures) needs to be addressed. One way this might be done is to add a large timeout to determine the end of a gesture. The distance of a stroke from the previous stroke might also be used. A sequence of strokes determined to be a single gesture might then be treated as a single stroke, with the exception of an additional feature which records the number of strokes in the gesture. The single-stroke recognition algorithm may then be applied.

Multi-path recognition is really still in its infancy. While the recognition algorithms of Chapter 5 seem to work well, there is not much to compare them against. Many others methods for multi-path

recognition need to be explored. That said, the author is somewhat wary that multiple finger input devices are so seductive that gesture research will concentrate on such devices to the exclusion of single-path devices. This would be unfortunate, as it seems likely that single-path devices will be much more prevalent for the foreseeable future, and thus more users will potentially benefit from the availability of single-path gesturing. Also, a thorough understanding of the issues involved in single-path gesturing will likely be of use in solving the more difficult problems encountered in the multi-path case.

The advent of pen-based computers leads to the question of how the single-stroke recognition described here may be combined with handwriting recognition. One approach is to pass input to the gesture recognizer after it has been rejected by the handwriting recognizer. The context in which the stroke has been made (e.g. drawing window or text window) can also be used to determine whether to invoke handwriting recognition or stroke recognition first.

The start of a single-stroke gesture is used to determine the set of possible gestures by looking at possible objects at which the gesture is directed. It may be desirable to explore the possibility that the gesture is directed at an object other than one indicated by the first point, e.g. an object may be indicated by a hot point of the gesture (e.g. the intersection point of the delete gesture). A similar ambiguity occurs when the input is a multiple-finger gesture; which of the fingers should be used to determine the object(s) at which the gesture is directed? In this case, a union of the gestures recognized by objects indicated by each finger could be used, but the possibility of conflict remains.

One problem with gesture-based systems is that there is usually no indication of the possible gestures accepted by the system.[1] This is a difficulty that will potentially prevent novices from using the system. One approach would be to use animation[6] to indicate the possible gestures and their effects, although how the user asks to see the animation remains an open question.

Also daunting to beginners is the timeout interval, where "stillness" is used to indicate that collection is over and manipulation is to begin. Typically, a beginner presses a mouse button and then thinks about what to do next; by that time the system has already classified the gesture as a dot. The timeout cannot be totally disabled, since it is the only way to enter the manipulation phase for some gestures. Perhaps some scheme where the timeouts are long (0.75 seconds) for novices and decrease with use is desirable. Another possibility is eliminating the timeout totally at the beginning of the gestures, thus disallowing dot gestures.

The current work suffers from a lack of formal user evaluation. Additional studies are needed to determine classifier performance as a function of training examples, and whether one user can use a classifier trained by another. In general, the costs and the benefits of fixed verses trainable recognition strategies need to be studied. The usability of eager recognizers is also of interest.

Recognizers that gradually adapt to users need to be studied as well. Such a recognizer requires the user to somehow indicate when a gesture is misclassified by the system. Lerner [78] demonstrated a potentially applicable scheme in which the system monitored subsequent actions to see if the user was satisfied with the result of an applied heuristic. There are dangers inherent in doubly-adaptive systems–if the system adapts to the user and the user to the system, both are aiming at moving targets, and thrashing is possible. The current approach requires the user explicitly to replace the

[1] Kurtenbach et. al. [75] say that gesture-based interfaces are "non-revealing," and present an interesting solution that unifies gesturing and pie-menu selection.

existing training examples with his own–a workable, if not glamorous, solution.

The low-level recognition work in this thesis is quite usable in its current state, and may be directly incorporated into systems as warranted. GRANDMA, however, is not useful as a base for future development. It is purely a research system, built as a platform for experimenting with input in user interface toolkits. Its output facilities are totally inadequate for real applications. GRANDMA was built solely by and for the author, who has no plans to maintain it. Nonetheless, GRANDMA embodies some important concepts of how gestures are to be integrated into object-oriented user interface tools.

The obvious next step is to integrate gestures into some existing user interface construction tools. Issues of technical suitability are important, but not paramount, in deciding which system to work on. Any chosen system must be well supported and maintained, so that there is a reasonable assurance that the system will survive. Furthermore, any chosen system must be widely distributed, in order to make the technology of gesture recognition available to as many experimenters as possible.

A number of existing systems are candidates for the incorporation of gestures. The NeXT Application Kit is technically the ideal platform–it is even programmed in Objective C. The appropriate hooks seem to be there to capture input at the right level in order to associate gestures with view classes. It is probably not worth the effort to implement an entire interpreter for entering gesture semantics at runtime, as this is not something a user will typically manipulate. A graphical interface to control semantics, based on constraints, would be an interesting addition. In general, a simpler way for mapping gestural attributes to application parameters needs to be determined.

The Andrew Toolkit (ATK) is another system into which gestures may be incorporated. ATK uses its own object-oriented programming language on top of C, so runtime representation of the class hierarchy, if not already present, should be straightforward to add. ATK has implemented dynamic loading of objects into running programs–this should make it possible to compile gesture semantics and load them into a running program without restarting the program. Unfortunately, due to their overhead, views tend to be large objects in ATK (e.g. individual notes in a score editor would not be separate views in ATK) making it difficult to associate different gestures with the smaller objects of interest in the interface. Scott Hassan, in a different approach, has added the author's gesture recognizer to the ATK text object, creating an interface that allows text editing via proofreader's marks.

Integrating gestures into Garnet is another possibility. What would be required is a gesture interactor, analogous to the gesture event handler in GRANDMA. Garnet interactors routinely specify their semantics via constraints, with an escape into Lisp available for unusual cases. Specifying gesture semantics should therefore be no problem in Garnet. James Landay has begun work integrating the author's recognizer into Garnet.

Gestures could also be added to MacApp. Besides being widely used, MacApp has the advantage that it runs on a Macintosh, which historically has run only one process at a time and has no virtual memory (this has changed with a recent system software release). While these points sound like disadvantages, the real-time operation needed to track the mouse reliably should be easy to achieve because of them. Because MacApp is implemented in Object Pascal, minimal meta-information about objects is available at runtime. In particular, message selectors are not first class objects in Object Pascal, it is not possible to ask if a given object responds to a message at runtime, and there

is no runtime representation of the class hierarchy. Many things that happen automatically because GRANDMA is written in Objective C will need to be explicitly coded in MacApp.

It would be desirable to have additional attributes of the gesture available for use in gesture semantics. Notably missing from the current set are locations where the path intersects itself and locations of sharp corners of the stroke. Both kinds of attributes can be used for pointing with a gesture, and allow for multiple points to be indicated with one single-path gesture. Also, having the numerical attributes also available in a scaled form (*e.g.* between zero and one) would simplify their use as parameters to application functions.

10.3 Final Remarks

The utility of gesture-based interfaces derives from the ability to communicate an entire primitive application transaction with a single gesture. For this to be possible, the gesture needs to be classified to determine the operation to be performed, and attributes of the gesture must be mapped to the parameters of the operation. Some parameters may be culled at the time the gesture is recognized, while others are best manipulated in the presence of feedback from the application. This is the justification for the two-phase approach, where gesture recognition is followed by a manipulation phase, which allows for the continuous adjustment of parameters in the presence of application feedback.

From the outset, the goal of this work was to provide tools to allow the easy creation of gesture-based applications. This research has led to prototypes of such tools, and has thus laid much of the groundwork for building such tools in the future. However, the goal will not have been achieved until gestures are integrated into existing user interface construction tools that are both well maintained and highly available. This involves more development and marketing than it does research, but it is vitally important to the future of gesture-based systems.

Appendix A

Code for Single-Stroke Gesture Recognition and Training

This appendix contains the actual C code used to recognize single-stroke gestures. The feature vector calculation, classifier training algorithm, and the linear classifier are all presented. The code may be obtained free of charge via anonymous ftp to emsworth.andrew.cmu.edu (subdirectory gestures) and is also available as part of the Andrew contribution to the X11R5 distribution.

A.1 Feature Calculation

The lowest level of the code deals with computing a feature vector from a sequence of mouse points that make up a gesture. Type `FV` is a pointer to a structure that holds a feature vector as well as intermediate results used in the calculation of the features. The function `FvAlloc` allocates an `FV`, which is initialized before processing the points of a gesture via `FvInit`. `FvAddPoint` is called for each input point of the gesture, and `FvCalc` returns the feature vector for the gesture once all the points have been entered.

The following is a sample code fragment demonstrating the use of these functions:

```
#include "matrix.h"
#include "fv.h"

Vector
InputAGesture()
{
    static FV fv;
    int x, y; long t; Vector v;

    /* FvAlloc() is typically called only once per program invocation. */
    if(fv == NULL) fv = FvAlloc();
```

```
    /* A prototypical loop to compute a feature vector from a gesture
       being read from a window manager. */
    FvInit(fv);
    while(GetNextPoint(&x, &y, &t) != END_OF_GESTURE)
        FvAddPoint(fv, x, y, t);
    v = FvCalc(fv);
    return v;

}
```

The returned vector `v` might now be passed to `sClassify` to classify the gesture.

The remainder of this section shows the header file, fv.h, which defines the `FV` type and the feature vector interface. This interface is implemented in fv.c, shown next.

```
/*****************************************************************************
fv.h -- Create a feature vector, useful for gesture classification,
   from a sequence of points (e.g. mouse points).
*****************************************************************************/

/* ------------------- compile time settable parameters ------------------- */
/* some of these can also be set at runtime, see fv.c */

#undef  USE_TIME
                    /* Define USE_TIME to enable the duration and maximum */
                    /* velocity features.  When not defined, 0 may be passed */
                    /* as the time to FvAddPoint. */

#define DIST_SQ_THRESHOLD  (3*3)
                    /* points within sqrt(DIST_SQ_THRESHOLD) */
                    /* will be ignored to eliminate mouse jitter */

#define SE_TH_ROLLOFF     (4*4)
                    /* The SE_THETA features (cos and sin of */
                    /* angle between first and last point) will */
                    /* be scaled down if the distance between the */
                    /* points is less than sqrt(SE_TH_ROLLOFF) */

/* ------------------------ Interface ------------------------ */

typedef struct fv *FV;
                    /* During gesture collection, an FV holds */
                    /* all intermediate results used in the */
                    /* calculation of a single feature vector */
```

```
FV      FvAlloc();      /* */
void    FvFree();       /* FV fv */
void    FvInit();       /* FV fv */
void    FvAddPoint();   /* FV fv, int x, y; long time; */
Vector  FvCalc();       /* FV fv; */

/*---------- internal data structure ----------*/
#define MAXFEATURES 32
        /* maximum number of features, occasionally useful as an array bound */

/* indices into the feature Vector returned by FvCalc */

#define PF_INIT_COS    0 /* initial angle (cos) */
#define PF_INIT_SIN    1 /* initial angle (sin) */
#define PF_BB_LEN      2 /* length of bounding box diagonal */
#define PF_BB_TH       3 /* angle of bounding box diagonal */
#define PF_SE_LEN      4 /* length between start and end points */
#define PF_SE_COS      5 /* cos of angle between start and end points */
#define PF_SE_SIN      6 /* sin of angle between start and end points */
#define PF_LEN         7 /* arc length of path */
#define PF_TH          8 /* total angle traversed */
#define PF_ATH         9 /* sum of abs vals of angles traversed */
#define PF_SQTH       10 /* sum of squares of angles traversed */

#ifndef USE_TIME
#   define NFEATURES      11
#else
#   define PF_DUR         11      /* duration of path */
#   define PF_MAXV        12      /* maximum speed */
#   define NFEATURES      13
#endif

/* structure which holds intermediate results during feature vector calculation */

struct fv {

    /* the following are used in calculating the features */
    double      startx, starty; /* starting point */
    long        starttime;      /* starting time */

    /* these are set after a few points and then left alone */
```

```
    double      initial_sin, initial_cos;  /* initial angle to x axis */

    /* these are updated incrementally upon every point */
    int     npoints;            /* number of points in path */

    double      dx2, dy2;       /* differences: endx-prevx, endy-prevy */
    double      magsq2;         /* dx2*dx2 + dy2*dy2 */

    double      endx, endy;     /* last point added */
    long        endtime;

    double      minx, maxx, miny, maxy;    /* bounding box */

    double      path_r, path_th;  /* total length and rotation (in radians) */
    double      abs_th;           /* sum of absolute values of path angles */
    double      sharpness;        /* sum of squares of path angles */
    double      maxv;             /* maximum velocity */


    Vector      y;                /* Actual feature vector */
};


/*****************************************************************************
fv.c -- Creates a feature vector, useful for gesture classification,
   from a sequence of points (e.g. mouse points).
*****************************************************************************/

#include <stdio.h>
#include <math.h>
#include "matrix.h"  /* contains Vector and associated functions */
#include "fv.h"

/* runtime settable parameters */
double dist_sq_threshold = DIST_SQ_THRESHOLD;
double se_th_rolloff = SE_TH_ROLLOFF;

#define EPS (1.0e-4)

/* allocate an FV struct including feature vector */

FV
```

```
FvAlloc()
{
    register FV fv = (FV) mallocOrDie(sizeof(struct fv));
    fv->y = NewVector(NFEATURES);
    FvInit(fv);
    return fv;
}

/* free memory associated with an FV struct */
void
FvFree(fv)
FV fv;
{
    FreeVector(fv->y);
    free((char *) fv);
}

/* initialize an FV struct to prepare for incoming gesture points */
void
FvInit(fv)
register FV fv;
{
    register int i;

    fv->npoints = 0;
    fv->initial_sin = fv->initial_cos = 0.0;
    fv->maxv = 0;
    fv->path_r = 0;
    fv->path_th = 0;
    fv->abs_th = 0;
    fv->sharpness = 0;
    fv->maxv = 0;
    for(i = 0; i < NFEATURES; i++)
        fv->y[i] = 0.0;
}

/* update an FV struct to reflect a new input point */
void
FvAddPoint(fv, x, y, t)
register FV fv; int x, y; long t;
{
    double dx1, dy1, magsq1;
```

```
    double th, absth, d;
#ifdef PF_MAXV
    long lasttime;
#endif

    ++fv->npoints;
    if(fv->npoints == 1) {      /* first point, initialize some vars */
        fv->starttime = fv->endtime = t;
        fv->startx = fv->endx = fv->minx = fv->maxx = x;
        fv->starty = fv->endy = fv->miny = fv->maxy = y;
        fv->endx = x; fv->endy = y;
        return;
    }

    dx1 = x - fv->endx; dy1 = y - fv->endy;
    magsq1 = dx1 * dx1 + dy1 * dy1;

    if(magsq1 <= dist_sq_threshold) {
        fv->npoints--;
        return;       /* ignore a point close to the last point */
    }

    if(x < fv->minx) fv->minx = x;
    if(x > fv->maxx) fv->maxx = x;
    if(y < fv->miny) fv->miny = y;
    if(y > fv->maxy) fv->maxy = y;

#ifdef PF_MAXV
    lasttime = fv->endtime;
#endif
    fv->endtime = t;

    d = sqrt(magsq1);
    fv->path_r += d;      /* update path length feature */

    /* calculate initial theta when the third point is seen */
    if(fv->npoints == 3) {
        double magsq, dx, dy, recip;
        dx = x - fv->startx; dy = y - fv->starty;
        magsq = dx * dx + dy * dy;
        if(magsq > dist_sq_threshold)  {
            /* find angle w.r.t. positive x axis e.g. (1,0) */
```

```
            recip = 1 / sqrt(magsq);
            fv->initial_cos = dx * recip;
            fv->initial_sin = dy * recip;
        }
    }

    if(fv->npoints >= 3) {   /* update angle-based features */
        th = absth = atan2(dx1 * fv->dy2 - fv->dx2 * dy1,
                           dx1 * fv->dx2 + dy1 * fv->dy2);
        if(absth < 0) absth = -absth;
        fv->path_th += th;
        fv->abs_th += absth;
        fv->sharpness += th*th;

#ifdef PF_MAXV   /* compute max velocity */
        if(fv->endtime > lasttime &&
           (v = d / (fv->endtime - lasttime)) > fv->maxv)
            fv->maxv = v;
#endif
    }

    /* prepare for next iteration */
    fv->endx = x; fv->endy = y;
    fv->dx2 = dx1; fv->dy2 = dy1;
    fv->magsq2 = magsq1;

    return;
}

/* calculate and return a feature vector */
Vector
FvCalc(fv)
register FV fv;
{
    double bblen, selen, factor;

    if(fv->npoints <= 1)
        return fv->y;          /* a feature vector of all zeros */

    fv->y[PF_INIT_COS] = fv->initial_cos;
    fv->y[PF_INIT_SIN] = fv->initial_sin;
```

```
    /* compute the length of the bounding box diagonal */
    bblen = hypot(fv->maxx - fv->minx, fv->maxy - fv->miny);

    fv->y[PF_BB_LEN] = bblen;

    /* the bounding box angle defaults to 0 for small gestures */
    if(bblen * bblen > dist_sq_threshold)
        fv->y[PF_BB_TH] = atan2(fv->maxy - fv->miny,
                                fv->maxx - fv->minx);

    /* compute the length and angle between the first and last points */
    selen = hypot(fv->endx - fv->startx,
                  fv->endy - fv->starty);
    fv->y[PF_SE_LEN] = selen;

    /* when the first and last points are very close, the angle features
       are muted so that they satisfy the stability criterion */
    factor = selen * selen / se_th_rolloff;
    if(factor > 1.0) factor = 1.0;
    factor = selen > EPS ? factor/selen :  0.0;
    fv->y[PF_SE_COS] = (fv->endx - fv->startx) * factor;
    fv->y[PF_SE_SIN] = (fv->endy - fv->starty) * factor;

    /* the remaining features have already been computed */
    fv->y[PF_LEN] =  fv->path_r;
    fv->y[PF_TH] = fv->path_th;
    fv->y[PF_ATH] = fv->abs_th;
    fv->y[PF_SQTH] = fv->sharpness;

#ifdef PF_DUR
    fv->y[PF_DUR] = (fv->endtime - fv->starttime)*.01;
#endif

#ifdef PF_MAXV
    fv->y[PF_MAXV] = fv->maxv * 10000;
#endif

    return fv->y;
}
```

A.2 Deriving and Using the Linear Classifier

Type `sClassifier` points at an object that represents a classifier able to discriminate between a set of gesture classes. Each gesture class is represented by an `sClassDope` type. The functions `sRead` and `sWrite` read and write a classifier to a file. The function `sNewClassifer` creates a new (empty) classifier. A training example is added using `sAddExample`. There is no function to explicitly add a new class to a classifier. When an example of a new class is added, the new class is created automatically. To train the classifier based on the added examples, call `sDoneAdding`. Once trained, `sClassify` and `sClassifyAD` are used to classify a feature vector as one of the classes; `sClassifyAD` optionally computes the rejection information.

Here is an example fragment for creating a new classifier, entering new training examples, and writing the resulting classifier out to a file. Some of these functions are timed (and further described) in section 9.1.7.

```
#include <stdio.h>
#include <math.h>
#include "bitvector.h"
#include "matrix.h"
#include "sc.h"

#define NEXAMPLES   15

sClassifier
MakeAClassifier()
{
    sClassifier sc = sNewClassifier();
    Vector InputAGesture();
    char name[100];
    int i;

    for(;;) {
        printf("Enter class name, newline to exit: ");
        if(gets(name) == NULL || name[0] == '\0')
            break;
        for(i = 1; i <= NEXAMPLES; i++) {
            printf("Enter %s example %d\n", name, i);
            sAddExample(sc, name, InputAGesture());
        }
    }
    sDoneAdding(sc);
    sWrite(fopen("classifier.out", "w"), sc);
    return sc;
```

```
}
```

Once a classifier has been created it can be used to classifier gestures as follows:

```
TestAClassifier(sc)
sClassifier sc;
{
    Vector v;
    sClassDope scd;
    double punambig, distance;

    for(;;) {
        printf("Enter a gesture\n");
        v = InputAGesture();
        scd = sClassifyAD(sc, v, &punambig, &distance);
        printf("Gesture classified as %s ", scd->name);
        printf("Probability of unambiguous classification: %g\n",
                punambig);
        printf("Distance from class mean: %g\n", distance);
    }
}
```

What follows is the header file and code to implement the statistical classifier.

```
/*************************************************************************
sc.h -- create single path classifiers from feature vectors of examples,
  as well as classifying example feature vectors.
*************************************************************************/

#define MAXSCLASSES 100 /* maximum number of classes */

typedef struct sclassifier *sClassifier;   /* classifier */
typedef int sClassIndex;                   /* per-class index */
typedef struct sclassdope  *sClassDope;    /* per-class information */

struct sclassdope { /* per gesture class information within a classifier */
    char        *name;        /* name of a class */
    sClassIndex number;       /* unique index (small integer) of a class */
    int         nexamples;    /* number of training examples */
    Vector      average;      /* average of training examples */
    Matrix      sumcov;       /* covariance matrix of examples */
};
```

```
struct sclassifier {  /* a classifier */
    int      nfeatures;      /* number of features in feature vector */
    int      nclasses;       /* number of classes known by this classifier */
    sClassDope  *classdope;  /* array of pointers to per class data */

    Vector      cnst;        /* constant term of discrimination function */
    Vector      *w;          /* array of coefficient weights */
    Matrix      invavgcov;   /* inverse covariance matrix */
};

sClassifier sNewClassifier();       /* */
sClassifier sRead();                /* FILE *f */
void        sWrite();               /* FILE *f; sClassifier sc; */
void        sFreeClassifier();      /* sc */
void        sAddExample();          /* sc, char *classname; Vector y */
void        sDoneAdding();          /* sc */
sClassDope  sClassify();            /* sc, y */
sClassDope  sClassifyAD();          /* sc, y, double *ap; double *dp */
sClassDope  sClassNameLookup();     /* sc, classname */
double      MahalanobisDistance(); /* Vector v, u; Matrix sigma */

/*************************************************************************
sc.c -- creates classifiers from feature vectors of examples, as well as
  classifying example feature vectors.
*************************************************************************/


#include <stdio.h>
#include <math.h>
#include "bitvector.h"
#include "matrix.h"
#include "sc.h"

#define EPS (1.0e-6)      /* for singular matrix check */

/* allocate memory associated with a new classifier */

sClassifier
sNewClassifier()
{
    register sClassifier sc =
```

```
        (sClassifier) mallocOrDie(sizeof(struct sclassifier));
    sc->nfeatures = -1;
    sc->nclasses = 0;
    sc->classdope = (sClassDope *)
        mallocOrDie(MAXSCLASSES * sizeof(sClassDope));
    sc->w = NULL;
    return sc;
}

/* free memory associated with a new classifier */
void
sFreeClassifier(sc)
register sClassifier sc;
{
    register int i;
    register sClassDope scd;

    for(i = 0; i < sc->nclasses; i++) {
        scd = sc->classdope[i];
        if(scd->name) free(scd->name);
        free(scd);
        if(sc->w && sc->w[i]) FreeVector(sc->w[i]);
        if(scd->sumcov) FreeMatrix(scd->sumcov);
        if(scd->average) FreeVector(scd->average);
    }
    free(sc->classdope);
    if(sc->w) free(sc->w);
    if(sc->cnst) FreeVector(sc->cnst);
    if(sc->invavgcov) FreeMatrix(sc->invavgcov);
    free(sc);
}

/* given a string name of a class, return its per-class information */
sClassDope
sClassNameLookup(sc, classname)
register sClassifier sc;
register char *classname;
{
    register int i;
    register sClassDope scd;
    static sClassifier lastsc;
    static sClassDope lastscd;
```

```
    /* quick check for last class name */
    if(lastsc == sc && STREQ(lastscd->name, classname))
        return lastscd;

    /* linear search through all classes for name */
    for(i = 0; i < sc->nclasses; i++) {
        scd = sc->classdope[i];
        if(STREQ(scd->name, classname))
            return lastsc = sc, lastscd = scd;
    }
    return NULL;
}

/* add a new gesture class to a classifier */
static sClassDope
sAddClass(sc, classname)
register sClassifier sc;
char *classname;
{
    register sClassDope scd;

    sc->classdope[sc->nclasses] = scd = (sClassDope)
                mallocOrDie(sizeof(struct sclassdope));
    scd->name = scopy(classname);
    scd->number = sc->nclasses;
    scd->nexamples = 0;
    scd->sumcov = NULL;
    ++sc->nclasses;
    return scd;
}

/* add a new training example to a classifier */
void
sAddExample(sc, classname, y)
register sClassifier sc;
char *classname;
Vector y;
{
    register sClassDope scd;
    register int i, j;
    double nfv[50];
```

```
    double nm1on, recipn;

    scd = sClassNameLookup(sc, classname);
    if(scd == NULL)
        scd = sAddClass(sc, classname);

    if(sc->nfeatures == -1)
        sc->nfeatures = NROWS(y);

    if(scd->nexamples == 0) {
        scd->average = NewVector(sc->nfeatures);
        ZeroVector(scd->average);
        scd->sumcov = NewMatrix(sc->nfeatures,sc->nfeatures);
        ZeroMatrix(scd->sumcov);

    }

    if(sc->nfeatures != NROWS(y)) {
        PrintVector(y, "sAddExample: funny vector nrows!=%d",
            sc->nfeatures);
        return;
    }

    scd->nexamples++;
    nm1on = ((double) scd->nexamples-1)/scd->nexamples;
    recipn = 1.0/scd->nexamples;

    /* incrementally update covariance matrix */
        for(i = 0; i < sc->nfeatures; i++)
        nfv[i] = y[i] - scd->average[i];

    /* only upper triangular part computed */
        for(i = 0; i < sc->nfeatures; i++)
       for(j = i; j < sc->nfeatures; j++)
        scd->sumcov[i][j] += nm1on * nfv[i] * nfv[j];

    /* incrementally update mean vector */
    for(i = 0; i < sc->nfeatures; i++)
        scd->average[i] =
             nm1on * scd->average[i] + recipn * y[i];

}
```

```
/* run the training algorithm on the classifier */
void
sDoneAdding(sc)
register sClassifier sc;
{
    register int i, j;
    int c;
    int ne, denom;
    double oneoverdenom;
    register Matrix s;
    register Matrix avgcov;
    double det;
    register sClassDope scd;

    if(sc->nclasses == 0)
        error("sDoneAdding: No classes\n");

    /* Given covariance matrices for each class (* number of examples - 1)
       compute the average (common) covariance matrix */

    avgcov = NewMatrix(sc->nfeatures, sc->nfeatures);
    ZeroMatrix(avgcov);
    ne = 0;
    for(c = 0; c < sc->nclasses; c++) {
        scd = sc->classdope[c];
        ne += scd->nexamples;
        s = scd->sumcov;
        for(i = 0; i < sc->nfeatures; i++)
            for(j = i; j < sc->nfeatures; j++)
                avgcov[i][j] += s[i][j];
    }

    denom = ne - sc->nclasses;
    if(denom <= 0) {
        printf("no examples, denom=%d\n", denom);
        return;
    }

    oneoverdenom = 1.0 / denom;
    for(i = 0; i < sc->nfeatures; i++)
        for(j = i; j < sc->nfeatures; j++)
```

```
            avgcov[j][i] = avgcov[i][j] *= oneoverdenom;

    /* invert the avg covariance matrix */

    sc->invavgcov = NewMatrix(sc->nfeatures, sc->nfeatures);
    det = InvertMatrix(avgcov, sc->invavgcov);
    if(fabs(det) <= EPS)
        FixClassifier(sc, avgcov);

    /* now compute discrimination functions */
    sc->w = (Vector *)
              mallocOrDie(sc->nclasses * sizeof(Vector));
    sc->cnst = NewVector(sc->nclasses);
    for(c = 0; c < sc->nclasses; c++) {
        scd = sc->classdope[c];
        sc->w[c] = NewVector(sc->nfeatures);
        VectorTimesMatrix(scd->average, sc->invavgcov,
                          /* product = */ sc->w[c]);
        sc->cnst[c] = -0.5 *
                  InnerProduct(sc->w[c], scd->average);
        /* could add log(priorprob class c) to cnst[c] */
    }

    FreeMatrix(avgcov);
    return;
}

/* classify a feature vector */
sClassDope
sClassify(sc, fv) {
    return sClassifyAD(sc, fv, NULL, NULL);
}

/* classify a feature vector, possibly computing rejection metrics */
sClassDope
sClassifyAD(sc, fv, ap, dp)
sClassifier sc;
Vector fv;
double *ap;
double *dp;
{
    double disc[MAXSCLASSES];
```

```
    register int i, maxclass;
    double denom, exp();
    register sClassDope scd;
    double d;

    if(sc->w == NULL)
        error("sClassifyAD: %x no trained classifier", sc);

    for(i = 0; i < sc->nclasses; i++)
        disc[i] = InnerProduct(sc->w[i], fv) + sc->cnst[i];

    maxclass = 0;
    for(i = 1; i < sc->nclasses; i++)
        if(disc[i] > disc[maxclass])
            maxclass = i;

    scd = sc->classdope[maxclass];

    if(ap) {      /* calculate probability of non-ambiguity */
        for(denom = 0, i = 0; i < sc->nclasses; i++)
            /* quick check to avoid computing negligible term */
            if((d = disc[i] - disc[maxclass]) > -7.0)
                denom += exp(d);
        *ap = 1.0 / denom;
    }

    if(dp)    /* calculate distance to mean of chosen class */
        *dp = MahalanobisDistance(fv, scd->average,
                                  sc->invavgcov);

    return scd;
}

/* Compute the Mahalanobis distance between two vectors v and u */
double
MahalanobisDistance(v, u, sigma)
register Vector v, u;
register Matrix sigma;
{
    register i;
    static Vector space;
    double result;
```

```
    if(space == NULL || NROWS(space) != NROWS(v)) {
        if(space) FreeVector(space);
        space = NewVector(NROWS(v));
    }
    for(i = 0; i < NROWS(v); i++)
        space[i] = v[i] - u[i];
    result = QuadraticForm(space, sigma);
    return result;
}

/* handle the case of a singular average covariance matrix by removing features */
FixClassifier(sc, avgcov)
register sClassifier sc;
Matrix avgcov;
{
    int i;
    double det;
    BitVector bv;
    Matrix m, r;

    /* just add the features one by one, discarding any that cause
       the matrix to be non-invertible */

    CLEAR_BIT_VECTOR(bv);
    for(i = 0; i < sc->nfeatures; i++) {
        BIT_SET(i, bv);
        m = SliceMatrix(avgcov, bv, bv);
        r = NewMatrix(NROWS(m), NCOLS(m));
        det = InvertMatrix(m, r);
        if(fabs(det) <= EPS)
            BIT_CLEAR(i, bv);
        FreeMatrix(m);
        FreeMatrix(r);
    }

    m = SliceMatrix(avgcov, bv, bv);
    r = NewMatrix(NROWS(m), NCOLS(m));
    det = InvertMatrix(m, r);
    if(fabs(det) <= EPS)
        error("Can't fix classifier!");
    DeSliceMatrix(r, 0.0, bv, bv, sc->invavgcov);
```

```
    FreeMatrix(m);
    FreeMatrix(r);

}

/* write a classifier to a file */
void
sWrite(outfile, sc)
FILE *outfile;
sClassifier sc;
{
    int i;
    register sClassDope scd;

    fprintf(outfile, "%d classes\n", sc->nclasses);
    for(i = 0; i < sc->nclasses; i++) {
        scd = sc->classdope[i];
        fprintf(outfile, "%s\n", scd->name);
    }
    for(i = 0; i < sc->nclasses; i++) {
        scd = sc->classdope[i];
        OutputVector(outfile, scd->average);
        OutputVector(outfile, sc->w[i]);

    }
    OutputVector(outfile, sc->cnst);
    OutputMatrix(outfile, sc->invavgcov);
}

/* read a classifier from a file */
sClassifier
sRead(infile)
FILE *infile;
{
    int i, n;
    register sClassifier sc;
    register sClassDope scd;
    char buf[100];


    printf("Reading classifier "), fflush(stdout);
```

```
    sc = sNewClassifier();
    fgets(buf, 100, infile);
    if(sscanf(buf, "%d", &n) != 1) error("sRead 1");
    printf("%d classes ", n), fflush(stdout);
    for(i = 0; i < n; i++) {
        fscanf(infile, "%s", buf);
        scd = sAddClass(sc, buf);
        scd->name = scopy(buf);
        printf("%s ", scd->name), fflush(stdout);
    }
    sc->w = allocate(sc->nclasses, Vector);
    for(i = 0; i < sc->nclasses; i++) {
        scd = sc->classdope[i];
        scd->average = InputVector(infile);
        sc->w[i] = InputVector(infile);
    }
    sc->cnst = InputVector(infile);
    sc->invavgcov = InputMatrix(infile);
    printf("\n");
    return sc;
}

/* compute pairwise distances between classes, and print the closest ones,
   as a clue as to which gesture classes are confusable */

sDistances(sc, nclosest)
register sClassifier sc;
{
    register Matrix d = NewMatrix(sc->nclasses, sc->nclasses);
    register int i, j;
    double min, max = 0;
    int n, mi, mj;

    printf("--------------------\n");
    printf("%d closest pairs of classes\n", nclosest);
    for(i = 0; i < NROWS(d); i++) {
        for(j = i+1; j < NCOLS(d); j++) {
            d[i][j] = MahalanobisDistance(
                        sc->classdope[i]->average,
                        sc->classdope[j]->average,
                        sc->invavgcov);
            if(d[i][j] > max) max = d[i][j];
```

```
            }
        }

        for(n = 1; n <= nclosest; n++) {
            min = max;
            mi = mj = -1;
            for(i = 0; i < NROWS(d); i++) {
                for(j = i+1; j < NCOLS(d); j++) {
                    if(d[i][j] < min)
                        min = d[mi=i][mj=j];
                }
            }
            if(mi == -1)
                break;

            printf("%2d) %10.10s to %10.10s d=%g nstd=%g\n",
                n,
                sc->classdope[mi]->name,
                sc->classdope[mj]->name,
                d[mi][mj],
                sqrt(d[mi][mj]));

            d[mi][mj] = max+1;
        }
        printf("----------\n");
        FreeMatrix(d);
    }
```

A.3 Undefined functions

The above code uses some functions whose definitions are not included in this appendix. These fall into four classes: standard library functions (including the math library), utility functions, bitvector functions, and vector/matrix functions, The standard library calls will not be discussed.

The utility functions used are

`STREQ(s1, s2)` returns FALSE iff strings `s1` and `s2` are equal.

`scopy(s)` returns a copy of the string `s`.

`error(format, arg1...)` prints a message and causes the program to exit.

`mallocOrDie(nbytes)` calls malloc, dying with an error message if the memory cannot be obtained.

The bit vector operations are an efficient set of functions for accessing an array of bits.

`CLEAR_BIT_VECTOR(bv)` resets an entire bit vector `bv` to all zeros,

`BIT_SET(i, bv)` sets the i^{th} bit of `bv` to one, and

`BIT_CLEAR(i, bv)` sets the i^{th} bit of `bv` to zero.

The vector/matrix functions are declared in matrix.h. Objects of type `Vector` and `Matrix` may be accessed like one and two dimensional arrays, respectively, but also contain additional information as to the size and dimensionality of the object (accessible via macros `NROWS`, `NCOLS`, and `NDIM`. It should be obvious from the names and the use of most of the functions (`NewVector`, `NewMatrix`, `FreeVector`, `FreeMatrix`, `ZeroVector`, `ZeroMatrix`, `PrintVector`, `PrintMatrix`, `InvertMatrix`, `InputVector`, `InputMatrix`, `OutputVector`, `OutputMatrix`, `VectorTimesMatrix`, and `InnerProduct`) what they do. As for the remaining functions,

`double QuadraticForm(Vector V, Matrix M)` computes the quantity $V'MV$, where the prime denotes the transpose operation.

`Matrix SliceMatrix(Matrix m, BitVector rowmask, BitVector colmask)` creates a new matrix, consisting only of those rows and columns in `m` whose corresponding bits are set it `rowmask` and `colmask`, respectively.

`Matrix DeSliceMatrix(Matrix m, double fill, BitVector rowmask; BitVector colmask; Matrix result)` first sets every element in `result` to `fill`, and then, every element in `result` whose row number is on in `rowmask` and whose column number is on in `colmask`, is set from the corresponding element in the input matrix `m`, which is smaller than `r`. The result of `SliceMatrix(DeSliceMatrix(m, fill, rowmask, colmask, result), rowmask, colmask)` is a copy of `m`, given legal values for all parameters.

These auxiliary functions, as well as a C-based X11R5 version of GDP, are all available as part of the ftp distribution mentioned above.

Bibliography

[1] Apple. *Inside Macintosh.* Addison-Wesley, 1985.

[2] Apple. *Macintosh System Software User's Guide, Version 6.0.* Apple Computer, 1988.

[3] H. Arakawa, K. Okada, and J. Masuda. On-line recognition of handwritten characters: Alphanumerics, Hiragana, Katakana, Kanji. In *Proceedings of the 4th International Joint Conference on Pattern Recognition*, pages 810–812, 1978.

[4] R. Baecker. Towards a characterization of graphical interaction. In Guedj, R. A., et. al., editor, *Methodology of Interaction*, pages 127–147. North Holland, 1980.

[5] R. Baecker and W. A. S. Buxton. *Readings in Human-Computer Interaction - A Multidisciplinary Approach.* Morgan Kaugmann Readings Series. Morgan Kaufmann, Los Altos, California, 1987.

[6] Ronald Baecker, Ian Small, and Richard Mander. Bringing icons to life. In *CHI'91 Conference Proceedings*, pages 1–6. ACM, April 1991.

[7] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM* 18(9), 1975.

[8] M. Berthod and J. P. Maroy. Learning in syntactic recognition of symbols drawn on a graphic tablet. *Computer Graphics Image Processing*, 9:166–182, 1979.

[9] J. Block and R. Dannenberg. Polyphonic accompaniment in real time. In *International Computer Music Conference*, Cambridge, Mass., 1985. Computer Music Association.

[10] R. Boie. Personnel communication. 1987.

[11] R. Boie, M. Mathews, and A. Schloss. The Radio Drum as a synthesizer controller. In *1989 International Computer Music Proceedings*, pages 42–45. Computer Music Association, November 1989.

[12] R. A. Bolt. *The Human Interface: where people and computers meet.* Lifetime Learning Publications, 1984.

[13] Radmilo M. Bozinovic. *Recognition of Off-line Cursive Handwriting: A Case of Multi-level Machine Perception.* PhD thesis, State University of New York at Buffalo, March 1985.

[14] W. A. S. Buxton. Chunking and phrasing and the design of human-computer dialogues. In *Information Processing 86*, North Holland, 1986. Elsevier Science Publishers B.V.

[15] W. A. S. Buxton. There's more to interaction than meets the eye: Some issues in manual input. In D. A. Norman and S. W. Draper, editors, *User Centered Systems Design: New Perspectives on Human-Computer Interaction*, pages 319–337. Lawrence Erlbaum Associates, Hillsdale, N.J., 1986.

[16] W. A. S. Buxton. Smoke and mirrors. *Byte*, 15(7):205–210, July 1990.

[17] W. A. S. Buxton. A three-state model of graphical input. *Proceedings of Interact '90*, August 1990.

[18] W. A. S. Buxton, R. Hill, and P. Rowley. Issues and techniques in touch-sensitive tablet input. *Computer Graphics*, 19(3):215–224, 1985.

[19] W. A. S. Buxton and B. Myers. A study in two-handed input. In *Proceedings of CHI '86*, pages 321–326. ACM, 1986.

[20] W. A. S. Buxton, W. Reeves, R. Baecker, and L. Mezei. The user of hierarchy and instance in a data structure for computer music. In Curtis Roads and John Strawn, editors, *Foundations of Computer Music*, chapter 24, pages 443–466. MIT Press, Cambridge, Massachusetts, 1985.

[21] W. A. S. Buxton, R. Sniderman, W. Reeves, S. Patel, and R. Baecker. The evolution of the SSSP score-editing tools. In Curtis Roads and John Strawn, editors, *Foundations of Computer Music*, chapter 22, pages 387–392. MIT Press, Cambridge, Massachusetts, 1985.

[22] S. K. Card, Moran, T. P., and A. Newell. The keystroke-level model for user performance time with interactive systems. *Communications of the ACM*, 23(7):601–613, 1980.

[23] L. Cardelli and R. Pike. Squeak: A language for communicating with mice. *SIGGRAPH '85 Proceedings*, 19(3), April 1985.

[24] R. M. Carr. The point of the pen. *BYTE*, 16(2):211–221, February 1991.

[25] Michael L. Coleman. Text editing on a graphic display device using hand-drawn proofreader's symbols. In M. Faiman and J. Nievergelt, editors, *Pertinent Concepts in Computer Graphics, Proceedings of the Second University of Illinois Conference on Computer Graphics*, pages 283–290. University of Illinois Press, Urbana, Chicago, London, 1969.

[26] P. W. Cooper. Hyperplanes, hyperspheres, and hyperquadrics as decision boundaries. In J. T. Tou and R. H. Wilcox, editors, *Computer and Information Sciences*, pages 111–138. Spartan, Washington, D.C., 1964.

[27] Brad J. Cox. Message/object programming: An evolutionary change in programming technology. *IEEE Software*, 1(1):50–61, January 1984.

[28] Brad J. Cox. *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 1986.

[29] R. B. Dannenberg. A structure for representing, displaying, and editing music. In *Proceedings of the 1986International Computer Music Conference*, pages 153–160, San Francisco, 1986. Computer Music Association.

[30] R. Duda and P. Hart. *Pattern Classification and Scene Analysis.* Wiley Interscience, 1973.

[31] The Economist. Digital quill. *The Economist*, 316(7672):88, September 15 1990.

[32] H. Eglowstein. Reach out and touch your data. *Byte*, 15(7):283–290, July 1990.

[33] W. English, D. Engelbart, and M. L. Berman. Display-selection techniques for text manipulation. *IEEE Transactions on Human Factors in Electronics*, HFE-8(1):21–31, 1967.

[34] S. S. Fels and Geoffrey E. Hinton. Building adaptive interfaces with neural networks: The glove-talk pilot study. Technical Report CRG-TR-90-1, University of Toronto, Toronto, Canada, 1990.

[35] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7:179–188, 1936.

[36] Flecchia and Nergeron. Specifying complex dialogs in ALGAE. *SIGCHI+GI 87Proceedings*, April 1987.

[37] K. S. Fu. *Syntactic Recognition in Character Recognition*, volume 112 of *Mathematics in Science and Engineering.* Academic Press, 1974.

[38] K. S. Fu. Hybrid aproaches to pattern recognition. In K. S. Fu J. Kittler and L. F. Pau, editors, *Pattern Recognition Theory and Applications*, NATO Advanced Study Institute, pages 139–155. D. Reidel, 1981.

[39] K. S. Fu. *Syntactic Pattern Recognition and Applications.* Prentice Hall, 1981.

[40] K. S. Fu and T. S. Yu. *Statistical Pattern Classification using Contextual Information.* Pattern Recognition and Image Processing Series. Research Studies Press, New York, 1980.

[41] J. Gettys, R. Newman, and R. W. Schiefler. *Xlib - C Language Interface X11R2* Massachusetts Institute of Technology, 1988.

[42] Dario Giuse. DP command set. Technical Report CMU-RI-TR-82-11, Carnegie Mellon University Robotics Institute, October 1982.

[43] R. Glitman. Startup readies 4-pound stylus pc. *PC Week*, 7(34):17–18, August 27 1990.

[44] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley series in Computer Science. Addison-Wesley, 1983.

[45] D. Goodman. *The complete HyperCard handbook.* Bantam Books, 1988.

[46] G. H. Granlund. Fourier preprocessing for hand print character recognition. *IEEE Transactions on Computers*, 21:195–201, February 1972.

[47] I. Guyon, P. Albrecht, Y. Le Cun, J. Denker, and W. Hubbard. Design of a neural network character recognizer for a touch terminal. *Pattern Recognition*, 24(2):105–119, 1991.

[48] D. J. Hand. *Kernel Discriminant Analysis.* Pattern Recognition and Image Processing Research Studies Series. Research Studies Press (A Division of John Wiley and Sons, Ltd.), New York, 1982.

[49] A. G. Hauptmann. Speech and gestures for graphic image manipulation. In *CHI '89 Proceedings*, pages 241–245. ACM, May 1989.

[50] Frank Hayes. True notebook computing arrives. *Byte*, pages 94–95, December 1989.

[51] P. J. Hayes, P. A. Szekely, and R. A. Lerner. Design alternatives for user interface management systems based on experience with COUSIN. In *CHI '85 Proceedings*, pages 169–175. ACM, April 1985.

[52] T. R. Henry, S. E. Hudson, and G. L. Newell. Integrating gesture and snapping into a user interface toolkit. In *UIST '90* pages 112–122. ACM, 1990.

[53] C. A. Higgins and R. Whitrow. On-line cursive script recognition. In B. Shackel, editor, *Human-Computer Interaction - Interact '84, IFIP*, pages 139–143, North-Holland, 1985. Elsevier Science Publishers B.V.

[54] R. Hill. Supporting concurrency, communication, and synchronization in human-computer interaction. *ACM Transactions on Graphics*, 5(3):179–210, July 1986.

[55] J. Hollan, E. Rich, W. Hill, D. Wroblewski, W. Wilner, K. Wittenberg, J. Grudin, and Members of the Human Interface Laboratory. An introduction to HITS: Human interface tool suite. Technical Report ACA-HI-406-88, Microelectronics and Computer Technology Corporation, Austin, Texas, 1988.

[56] Bruce L. Horn. An introduction to object oriented programming, inheritance and method combination. Technical Report CMU-CS-87-127, Carnegie Mellon University Computer Science Department, 1988.

[57] A. B. S. Hussain, G. T. Toussaint, and R. W. Donaldson. Results obtained using a simple character recognition procedure on Munson's handprinted data. *IEEE Transactions on Computers*, 21:201–205, February 1972.

[58] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct manipulation interfaces. In D. A. Norman and S. W. Draper, editors, *User Centered System Design*, pages 118–123. Lawrence Erlbaum Associates, Hillsdale, N.J., 1986.

[59] Pencept Inc. Software control at the stroke of a pen. In *SIGGRAPH Video Review*, volume Issue 18: Edited Compilations from CHI '85. ACM, 1985.

[60] F. Itakura. Minimum prediction residual principle applied to speech recognition. *IEEE Trans. Acoustics, Speech, Signal Processing*, ASSP-23(67), 1975.

[61] J. C. Jackson and Renate J. Roske-Hofstrand. Circling: A method of mouse-based selection without button presses. In *CHI '89 Proceedings*, pages 161–166. ACM, May 1989.

[62] Mike James. *Classification Algorithms*. Wiley-Interscience. John Wiley and Sons, Inc., New York, 1985.

[63] R. E. Johnson. Model/View/Controller. November 1987 (unpublished manuscript).

[64] Dan R. Olsen Jr. Syngraph: A graphical user interface generator. *Computer Graphics*, 17(3):43–50, July 1983.

[65] K. G. Morse Jr. In an upscale world. *Byte*, 14(8), August 1989.

[66] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, New Jersey, 1978.

[67] Joonki Kim. Gesture recognition by feature analysis. Technical Report RC12472, IBM Research, December 1986.

[68] Nancy T. Knolle. Variations of model-view-controller. *Journal of Object-Oriented Programming*, 2:42–46, September/October 1989.

[69] D. Kolzay. Feature extraction in an optical character recognition machine. *IEEE Transactions on Computers*, 20:1063–1067, 1971.

[70] Glenn E. Krasner and Stephen T. Pope. A description of the Model-View-Controller user interface paradigm in the Smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, August 1988.

[71] M. W. Kreuger. *Artificial Reality*. Addison-Wesley, Reading, MA., 1983.

[72] M. W. Kreuger, T. Gionfriddo, and K. Hinrichsen. Videoplace: An artificial reality. In *Proceedings of CHI'85*, pages 35–40. ACM, 1985.

[73] E. Kreyszig. *Advanced Engineering Mathematics*. Wiley, 1979. Fourth Edition.

[74] W. J. Krzanowski. *Principles of Multivariate Analysis: A User's Perspective*. Oxford Statistical Science Series. Clarendon Press, Oxford, 1988.

[75] G. Kurtenbach and W. A. S. Buxton. GEdit: A test bed for editing by contiguous gestures. *SIGCHI Bulletin*, pages 22–26, 1991.

[76] Martin Lamb and Veronica Buckley. New techniques for gesture-based dialog. In B. Shackel, editor, *Human-Computer Interaction - Interact '84, IFIP*, pages 135–138, North-Holland, 1985. Elsevier Science Publishers B.V.

[77] C. G. Leedham, A. C. Downton, C. P. Brooks, and A. F. Newell. On-line acquisition of pitman's handwritten shorthand as a means of rapid data entry. In B. Shackel, editor, *Human-Computer Interaction - Interact '84, IFIP*, pages 145–150, North-Holland, 1985. Elsevier Science Publishers B.V.

[78] Barbara Staudt Lerner. *Automated Customization of User Interfaces.* PhD thesis, Carnegie Mellon University, 1989.

[79] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, 22(2):8–22, February 1989.

[80] James S. Lipscomb. A trainable gesture recognizer. *Pattern Recognition*, 1991. Also available as IBM Tech Report RC 16448 (#73078).

[81] D. J. Lyons. Go Corp. gains ground in pen-software race. *PC Week*, 7(29):135, July 23 1990.

[82] Gale Martin, James Pittman, Kent Wittenburg, Richard Cohen, and Tome Parish. Sign here, please. *Byte*, 15(7):243–251, July 1990.

[83] J. T. Maxwell. Mockingbird: An interactive composer's aid. Master's thesis, MIT, 1981.

[84] P. McAvinney. The Sensor Frame–a gesture-based device for the manipulation of graphic objects. Available from Sensor Frame, Inc., Pittsburgh, Pa., December 1986.

[85] P. McAvinney. Telltale gestures. *Byte*, 15(7):237–240, July 1990.

[86] Margaret R. Minsky. Manipulating simulated objects with real-world gestures using a force and position sensitive screen. *Computer Graphics*, 18(3):195–203, July 1984.

[87] P. Morrel-Samuels. Clarifying the distinction between lexical and gestural commands. *International Journal of Man-Machine Studies*, 32:581–590, 1990.

[88] G. Muller and R. Giulietti. High quadlity music notation: Interactive editing and input by piano keyboard. In *Proceedings of the 1987International Computer Music Conference*, pages 333–340, San Francisco, 1987. Computer Music Association.

[89] Hiroshi Murase and Toru Wakahara. Online hand-sketched figure recognition. *Pattern Recognition*, 19(2):147–160, 1988.

[90] B. Myers and W. A. S. Buxton. Creating highly-interactive and graphical user interfaces by demonstration. *Computer Graphics*, 20(3):249–258, 1986.

[91] B. A. Myers. A new model for handling input. *ACM Transactions on Information Systems*, 1990.

[92] B. A. Myers, D. Giuse, R. B. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, Andrew Mickish, and Phillippe Marchal. Garnet: comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11):71–85, Nov 1990.

[93] B. A. Myers, B. Vander Zanden, and R. B. Dannenberg. Creating graphical interactive application objects by demonstration. In *UIST'89 Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 95–104. ACM, November 1989.

[94] Brad A. Myers. A taxonomy of user interfaces for window managers. *IEEE Computer Graphics and Applications*, 8(5):65–84, 1988.

[95] Brad A. Myers. Encapsulating interactive behaviors. In *Human Factors in Computing Systems*, pages 319–324, Austin, TX, April 1989. Proceedings SIGCHI'89.

[96] Brad A. Myers. User interface tools: Introduction and survey. *IEEE Software*, 6(1):15–23, January 1989.

[97] Brad A. Myers. Demonstration interfaces: A step beyond direct manipulation. Technical Report CMU-CS-90-162, Carnegie Mellon School of Computer Science, Pittsburgh, PA, August 1990.

[98] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, March 1990.

[99] L. Nakatani. Personal communication, Bell Laboratories, Murray Hill, N.J. January 1987.

[100] T. Neuendorffer. *Adew Reference Manual*. Information Technology Center, Pittsburgh, PA, 1989.

[101] W. M. Newman and R. F. Sproull. *Principles of Interactive Computer Graphics*. McGraw-Hill, 1979.

[102] NeXT. *The NeXT System Reference Manual*. NeXT, Inc., 1989.

[103] L. Norton-Wayne. A coding approach to pattern recognition. In J. Kittler, K. S. Fu, and L. F. Pau, editors, *Pattern Recognition Theory and Applications*, NATO Advanced Study, pages 93–102. D. Reidel, 1981.

[104] K. K. Obermeier and J. J. Barron. Time to get fired up. *Byte*, 14(8), August 1989.

[105] A. J. Palay, W. J. Hansen, M. L. Kazar, M. Sherman, M. G. Wadlow, T. P. Neuendorffer, Z. Stern, M. Bader, and T. Peters. The Andrew toolkit: An overview. In *Proceedings of the USENIX Technical Conference*, pages 11–23, Dallas, February 1988.

[106] PC Computing. Ten other contendors in the featherweight division. *PC Computing*, 2(12):89–90, December 1989.

[107] J. A. Pickering. Touch-sensitive screens: the tecnologies and their application. *International Journal of Man-Machine Studies*, 25:249–269, 1986.

[108] R. Probst. Blueprints for building user interfaces: Open Look toolkits. Technical report, Sun Technology, August 1988.

[109] James R. Rhyne and Catherine G. Wolf. Gestural interfaces for information processing applications. Technical Report RC12179, IBM T.J. Watson Research Center, IBM Corporation, P.O. Box 218, Yorktown Heights, NY 10598, September 1986.

[110] J. Rosenberg, R. Hill, J. Miller, A. Schulert, and D. Shewmake. UIMSs: Threat or menace? In *CHI '88* pages 197–212. ACM, 1988.

[111] D. Rubine and P. McAvinney. The Videoharp. In *1988 International Computer Music Proceedings*. Computer Music Association, September 1988.

[112] D. Rubine and P. McAvinney. Programmable finger-tracking instrument controllers. *Computer Music Journal*, 14(1):26–41, 1990.

[113] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[114] K. J. Schmucker. MacApp: An application framework. *Byte*, 11(8):189–193, August 1986.

[115] Kurt J. Schmucker. *Object-Oriented Programming for the Macintosh*. Hayden Book Company, 1986.

[116] A. C. Shaw. Parsing of graph-representable pictures. *JACM*, 17(3):453, 1970.

[117] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, pages 57–62, August 1983.

[118] John L. Sibert, William D. Hurley, and Teresa W. Bleser. An object-oriented user interface management system. In *SIGGRAPH '86* pages 259–268. ACM, August 1986.

[119] Jack Sklanksy and Gustav N. Wassel. *Pattern Classifiers and Trainable Machines*. Springer-Verlag, New York, 1981.

[120] W. W. Stallings. Recognition of printed chinese characters by automatic pattern analysis. *Computer Graphics and Image Processing*, 1:47–65, 1972.

[121] Mark Stefik and Daniel G. Bobrow. Object-oriented programming: Themes and variations. *AI Magazine*, 6(4):40–62, Winter 1986.

[122] Jess Stein, editor. *The Random House Dictionary of the English Language*. Random House, Cambridge, Mass., 1969.

[123] Martin L. A. Sternberg. *American Sign Language: A Comprehensive Dictionary*. Harper and Row, New York, 1981.

[124] M. D. Stone. Touch-screens for intuitive input. *PC Magazine*, pages 183–192, August 1987.

[125] C. Y. Suen, M. Berthod, and S. Mori. Automatic recognition of handprinted characters: The state of the art. *Proceedings of the IEEE*, 68(4):469–487, April 1980.

[126] Sun. *SunWindows Programmers' Guide*. Sun Miscrosystems, Inc., Mountain View, Ca., 1984.

[127] Sun. *NeWS Preliminary Technical Overview*. Sun Miscrosystems, Inc., Mountain View, Ca., 1986.

[128] Shinichi Tamura and Shingo Kawasaki. Recognition of sign language motion images. *Pattern Recognition*, 21(4):343–353, 1988.

[129] C. C. Tappert, C. Y. Suen, and Toru Wakaha. The state of the art in on-line handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(8):787–808, August 1990.

[130] E. R. Tello. Between man and machine. *Byte*, 13(9):288–293, September 1988.

[131] A. Tevanian. MACH: A basis for future UNIX development. Technical Report CMU-CS-87-139, Carnegie Mellon University Computer Science Dept., Pittsburgh, PA, 1987.

[132] D. S. Touretzky and D. A. Pomerleau. What's hidden in the hidden layers? *Byte*, 14(8), August 1989.

[133] V. M. Velichko and N. G. Zagoruyko. Automatic recognition of 200 words. *Int. J. Man-Machine Studies*, 2(2):223, 1970.

[134] A. Waibel and J. Hampshire. Building blocks for speech. *Byte*, 14(8):235–242, August 1989.

[135] A. I. Wasserman. Extending state transition diagrams for the specification of human-computer interaction. *IEEE Transactions on Software Engineering*, SE-11(8):699–713, August 1985.

[136] D. Weimer and S. K. Ganapathy. A synthetic visual environment with hand gesturing and voice input. In *CHI '89Proceedings*, pages 235–240. ACM, 1989.

[137] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, August 1962.

[138] A. P. Witkin. Scale-space filtering. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1019–1022, 1983.

[139] C. G. Wolf. A comparative study of gestural and keyboard interfaces. *Proceedings of the Humans Factors Society*, 32nd Annual Meeting:273–277, 1988.

[140] C. G. Wolf and J. R. Rhyne. A taxonomic approach to understanding direct manipulation. *Proceedings of the Human Factors Society*, 31st Annual Meeting:576–580, 1987.

[141] Catherine G. Wolf. Can people use gesture commands? Technical Report RC11867, IBM Research, April 1986.

[142] Xerox Corporation. JUNO. In *SIGGRAPH Video Review Issue 19 CHI '85 Compilation.* ACM, 1985.

THE DESIGN AND EVALUATION OF MARKING MENUS

by

Gordon Paul Kurtenbach

A thesis submitted in conformity with the requirements
of the Degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

This research focuses on the use of hand drawn marks as a human-computer input technique. Drawing a mark is an efficient command input technique in many situations. However, marks are not intrinsically self-explanatory as are other interactive techniques such as buttons and menus. This research develops and evaluates an interaction technique called marking menus which integrates menus and marks such that both self-explanation and efficient interaction can be provided.

A marking menu allows a user to perform a menu selection by either popping up a radial menu and then selecting an item, or by drawing a straight mark in the direction of the desired menu item. Drawing a mark avoids popping up the menu. Marking menus can also be hierarchic. In this case, hierarchic radial menus and "zig-zag" marks are used. Marking menus are based on three design principles: self-revelation, guidance and rehearsal. Self-revelation means a marking menu reveals to a user what functions or items are available. Guidance means a marking menu guides a user in selecting an item. Rehearsal means that the guidance provided by the marking menu is a rehearsal of making the mark needed to select an item. Self-revelation helps a novice determine what functions are available, while guidance and rehearsal train a novice to use the marks like an expert. The intention is to allow a user to make a smooth and efficient transition from novice to expert behavior.

This research evaluates marking menus through empirical experiments, a case study, and a design study. Results shows that (1) 4, 8 and 12 item menus are advantageous when selecting using marks, (2) marks can be used to reliably select from four-item menus that are up to four levels deep or from eight-item menus that are up to two levels deep, (3) marks can be performed more accurately with a pen than a mouse, but the difference is not large, (4) in a practical application, users tended towards using the marks 100% of the time, (5) using a mark, in this application, was 3.5 times faster than selection using the menu, (6) the design principles of marking menus can be generalized to other types of marks.

THE DESIGN AND EVALUATION OF MARKING MENUS

Gordon Paul Kurtenbach
Degree of Doctor of Philosophy
Graduate Department of Computer Science, University of Toronto, 1993

abstract

This research focuses on the use of hand drawn marks as a human-computer input technique. Drawing a mark is an efficient command input technique in many situations. However, marks are not intrinsically self-explanatory as are other interactive techniques such as buttons and menus. This research develops and evaluates an interaction technique called marking menus which integrates menus and marks such that both self-explanation and efficient interaction can be provided.

A marking menu allows a user to perform a menu selection by either popping up a radial menu and then selecting an item, or by drawing a straight mark in the direction of the desired menu item. Drawing a mark avoids popping up the menu. Marking menus can also be hierarchic. In this case, hierarchic radial menus and "zig-zag" marks are used. Marking menus are based on three design principles: self-revelation, guidance and rehearsal. Self-revelation means a marking menu reveals to a user what functions or items are available. Guidance means a marking menu guides a user in selecting an item. Rehearsal means that the guidance provided by the marking menu is a rehearsal of making the mark needed to select an item. Self-revelation helps a novice determine what functions are available, while guidance and rehearsal train a novice to use the marks like an expert. The intention is to allow a user to make a smooth and efficient transition from novice to expert behavior.

This research evaluates marking menus through empirical experiments, a case study, and a design study. Results shows that (1) 4, 8 and 12 item menus are advantageous when selecting using marks, (2) marks can be used to reliably select from four-item menus that are up to four levels deep or from eight-item menus that are up to two levels deep, (3) marks can be performed more accurately with a pen than a mouse, but the difference is not large, (4) in a practical application, users tended towards using the marks 100% of the time, (5) using a mark, in this application, was 3.5 times faster than selection using the menu, (6) the design principles of marking menus can be generalized to other types of marks.

Acknowledgments

Many years ago when I was in high school my classmates and I spent three days writing occupation aptitude tests. Months later the computer graded tests were returned to us. I remember my friends' and my own excitement as we ripped open the envelopes to see what the computer had recommended. My friends cheered as they read out their long list of possibilities: doctor! lawyer! pilot! writer! scientist! With great anticipation I opened my computer recommendation. There, before my eyes, was one lonely recommendation: *pre-cast concrete worker.*

Although I have failed to fulfill my destiny as pre-cast concrete worker, I have created this thesis with the support of many people. In particular, I would like to thank:

- My supervisor and friend, Bill Buxton. Bill's creativity, intellect, and humor inspired me to pursue research and make bad jokes.

- The members of my committee: Ron Baecker, Mark Chignell, Marilyn Mantei, Ken Sevcik, and Cathy Wolf. Each contributed in helping me polish my research into a doctoral thesis.

- Great researchers and friends. Abigail Sellen greatly helped by designing experiments, writing, and putting on excellent parties; Tom Moran, Stuart Card, and Ken Pier provided creative insights and guidance; George Fitzmaurice and Beverly Harrison waded through treacherous drafts of my thesis, helped me make it a better document, and listened to my concerns over many a cappuccino; Gary Hardock utilized my work in his research and put up with my kidding; George Drettakis and Dimitri Nastos kept the lab systems running, humored me, and organized the most delicious Greek barbecues; Tim Brecht advised me, made me laugh way too loud and long, yet still managed to keep me sane.

I don’t think I’ll thank the computer that graded the aptitude tests...

To my parents, Helen and Leo,

and my brother and sisters,

Robert, Beverly, Donna, Carole, and Tammy:

"My thesis is done, you can probably reach me at home now"

Table of Contents

Chapter 1: Introduction

Research in the last forty years has brought great improvements in the quality of human-computer interactions. In the past, human-computer dialogs were optimized for the computer; humans communicated with computers using protocols that were easy for the computer to understand but were hard for a human to understand and use, for example, machine languages. Advances in human-computer interaction have changed this situation. Controlling a computer no longer requires memorizing obtuse, cryptic codes or an intimate understanding of the internal workings of the computer. In well-designed systems, human-computer interactions are optimized for the human. Interfaces now make use of sophisticated graphics, sound, and pointing devices to make the human's job easier.

The major advances in human-computer interaction have been in making computers easier to use. Specifically, research on methods to reduce the amount of training a person needs before being able to operate a computer has come a long way. For example, the *Apple Macintosh* has set standards for the minimal amount of instruction that a person needs before operating a computer. Because of these advances, the world of computers opened up for people who otherwise would not have invested the time in training to operate a computer system.

Given these advances in human-computer interaction, we can think of the interface as currently being optimized for the human, specifically, the novice computer user. Clearly, this is of great value, but we can consider another important class of user—the expert. Human capacity for the development of skills is great. Virtuoso pianists are proof of this. Virtuosos invest a great deal of time in practicing their skills—eight hours of practice a day is not uncommon. Now consider expert computer users. It is not uncommon for an expert computer user to spend eight hours a day working on the computer. Therefore, there is untapped potential for human skill

development in human-computer interactions. A good interface should take advantage of this potential and not limit the efficiency of a skilled user.

In order for this skill potential to be tapped, an interface must have certain properties. First, the interface must provide interaction methods that are suitable for an expert. Experts require efficient interactions. As a result, interactions may be terse and unprompted. Second, and most critically, the interface must also provide support for a novice to become expert. We look at the interface design not so much as making the interface easier to use but rather as *accelerating the rate at which novices begin to perform like experts.* This goal demands three components: support for the novice, support for the expert, and an efficient mechanism to support the transition from novice to expert (see Figure 1.1).

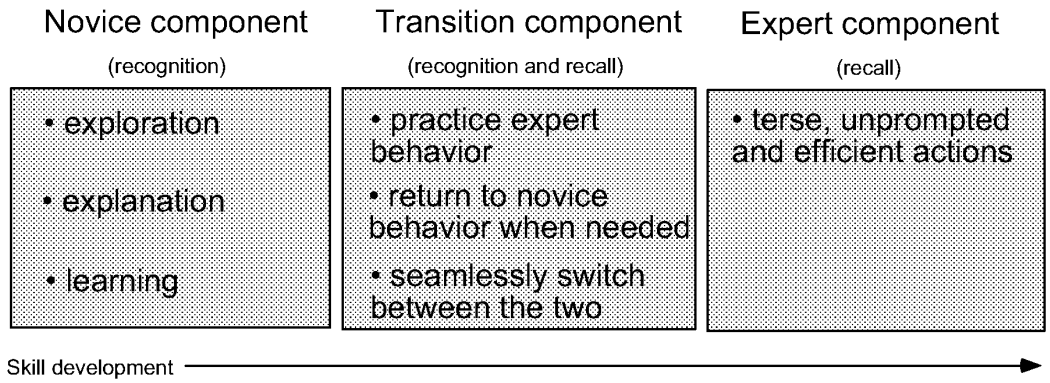


Figure 1.1: The components required to accelerate the rate at which users begin to perform like experts. The novice component allows a user to issue commands by searching for them and recognizing them. The expert component allows a user to efficiently issue commands by recalling the action associated with the command. The transition component allows a user to efficiently switch between these two methods to learn and practice command action associations.

In this dissertation, we focus on an interaction technique that is intended to take advantage of this skill potential and support the development of skill. We propose an interaction technique which has a two modes. In the first mode, the style of interaction is intended to facilitate novice use. In the second mode, the style of interaction is intended for skilled expert behavior. The first mode is also designed to allow a novice to practice the skills required in the second mode. A user can switch to the second mode by operating the technique quickly. One can think of this in metaphorical terms. When you are learning to drive a car, its suitable to have a car

that is designed for a student driver. However, as your driving skills improve, the car incrementally transforms into a Ferrari.

1.1. GENERAL AREA AND DEFINITIONS

To support the expert component described in the previous section, we focus on a style of human computer interaction in which a user "writes" on the display surface. This style of interaction is similar to writing or drawing with a pen on ordinary paper. Writing on a display, however, is accomplished with a special pen and the computer simulates the appearance of ink.[1]

We define a *mark* as the series of pixels that are changed to a special "ink" color when the pen is pressed and then moved across the display. The pixels that are changed to an ink color are those which lay directly under the tip of the pen as it is moved across the display. Free hand drawings, ranging from meaningless scribbles to meaningful line drawings and symbols, including handwriting, are examples of marks. The act of drawing a mark is referred to as *marking*.

Marks can be created not only with a pen but also with other types of input devices. For example, a mouse can leave a trail of ink (commonly referred to as an *ink-trail*) behind the tracking symbol when the mouse button is pressed and the mouse is dragged. Some systems use a pen and tablet. In this case, marks are made on the display by writing on the tablet instead of the display.

From a user's point of view, these interfaces allow one to make marks and then have the system interpret those marks. There are, however, systems in which marks can be made but not recognized by the system. They are interpreted strictly as annotations, for example, *Freestyle* (Perkins, Blatt, Workman, & Ehrlich, 1989). The focus of this dissertation, however, is on systems in which marks are interpreted as commands and parameters.

Much of the literature refers to marks as *gestures*. However, the term gesture is inappropriate in this context. Indeed creating a mark does involve a physical

---

[1] The pen, in these types of systems, is sometimes referred to as a stylus.

gesture but the real object of interpretation is the mark itself.[2] For example, the "X" mark requires a completely different physical gesture if performed with a pen instead of a mouse. Gesture is an important aspect of mark because some marks may require awkward physical gestures with the input device. However, the two terms should be distinguished. The term gesture is more appropriate for systems in which the gestures leave no marks, for example, *VideoPlace* (Krueger, Giofriddo & Hinrichsen, 1985). The term mark is more appropriate for pen-based computer systems or applications that emulate paper and pen.

1.2. WHY USE MARKS?

Current human-computer interfaces are asymmetric in terms of input and output capabilities. There a number of computer output modes: visual, audio and tactile. Most computers extensively utilize the visual mode; high resolution images which use thousands of colors of can be displayed quickly and in meaningful ways to a user. In contrast, a computer's ability to sense user input is limited. Humans have a wide range of communication skills such as speech and touch, but most computers sense only a small subset of these. For example, keyboards only sense finger presses (but not pressure) and mice only sense very simple arm or wrist movements. Therefore, we believe the advent of the pen as a computer input device provides the opportunity to increase input bandwidth through the use of marks.[3]

There are two major motivations for using marks. The first addresses the problem of efficiently accessing the increasing number of functions in applications. The second motivation is that there are some intrinsic qualities that marks have which can provide a more "natural" way to articulate otherwise difficult or awkward concepts (such as spatial or temporal information). Both of these motivations will now be examined in more detail.

---

[2] There are systems where interpretation depends not only on what is drawn but also how it is drawn. For example, an "X" drawn quickly may have a different interpretation from a "X" that is drawn slowly. By this dissertation's terminology, these systems would contain a combination of marking and gesture recognition.

[3] It is ironic that one of the first input devices for graphics was a light pen which wrote directly on the display surface (Sutherland, 1963).

1.2.1. Symbolic nature

The inadequacy of mouse and keyboard interfaces is exemplified by applications that are controlled through button presses and position information.[4] Buttons must be accessible and thus require physical space. Problems occur when an application has more functions than can be mapped to buttons or reasonably managed on the display. Other problems also exist: arbitrary mappings between functions and buttons can be confusing, and user management of the display and removal of graphical buttons can be tedious.

Expert users of these types of systems find the interface inadequate because button interfaces are inefficient. The existence of interaction techniques that override buttons for the sake of efficiency is evidence of this. Experts, having great familiarity with the interface, are aware of the set of available commands. Menus are no longer needed to remind them of available commands and invoking commands through menu display becomes very tedious.

Designers have addressed this problem in several ways. One solution is *accelerators keys* which allow experts direct access to commands. An accelerator key is a key on the keyboard which, when pressed, immediately executes a function associated with a menu item or button. The intention is that using an accelerator key saves the user the time required to display and select a menu item or button. Many systems display the names of accelerator keys next to menu items or buttons to help users learn and recall the associations between accelerator keys and functions.

Another way of supporting an expert is by supplying a command line interface in addition to a direct manipulation interface. Commodore's command line interface, *CLI*, and graphical user interface, *Intuition*, are an example of this approach.

Both these approaches have their problems. In the case of accelerator keys, arbitrary mappings between functions and keys can be hard to learn and remember. Sometimes mnemonics can be established between accelerator key and function (e.g., control-o for "open"), but mnemonics quickly run out as the number of accelerator keys increases. Further confusion can be caused by different applications

---

[4] The term buttons is used as a generic way of describing menus items, dialog box items, icons, keys on a keyboard, etc., which are typical of direct manipulation interfaces.

using a common key for different functions or by different applications using different keys for a common function. Experts must then remember arbitrary or complex mappings between keys and functions depending on application. Command line interfaces are problematic because they are radically different from direct manipulation interfaces. To become an expert, a novice must learn another entirely different interface.

Marks, because of their symbolic nature, can make functions more immediately accessible. Rather than triggering a function by a button press, a mark can signal a command. For example, a symbolic mark can be associated with a function and a user can invoke the function by drawing the symbol. In theory, because marks can be used to draw any symbol or series of symbols, marks can provide a quicker method of choosing a command than searching for a physical or graphical button and pressing it. In practice, the number of marks is limited by the system's ability to recognize symbols and a human's ability to remember the set of symbols. Nevertheless, even if only a small set of marks are used, a user can invoke the associated functions immediately.

Marks can also be used to hide functions because they are user generated symbols. For example, researchers at Xerox PARC made use of this property when faced with a dilemma during the design of a pen-based application. This application runs on a wall sized display where a user can write on the display using an electric pen (Elrod et. al., 1992). There were two major design requirements. First, the designers wanted the application to look and operate like a whiteboard and maximize the size of the area where drawing could take place. Second, they wanted to provide additional functions commonly found in computer drawing programs. This second requirement meant that many graphical buttons would need to appear on the screen. This, however, violated the first design requirement because the numerous graphical buttons would consume too much of the drawing area and make the interface look complicated.

The design solution was to assign many of the drawing functions to marks. Marks provided a way to hide additional functionality from novices while expert users could use the marks to access additional functions. This design also avoided using buttons for these functions and, in many cases, marks were a much more effective way of articulating a function.

1.2.2. Intrinsic advantages

The advantages of pen input and marks have been expressed in the literature (Bush, 1945; Licklider 1960; Ellis & Sibley, 1967; Hornbuckle, 1967; Coleman, 1969; Ward & Blesser, 1985; Rhyne & Wolf, 1986; Wolf, 1986; Buxton, 1986; Welbourn & Whitrow, 1988; Wolf, Rhyne, & Ellozy, 1989; Morrel-Samuels, 1990; Kurtenbach & Hulteen, 1990). Specifically, marks provide the ability to:

• embed many command attributes into a single mark;

• reduce learning time due to the mnemonic nature of marks and users' existing knowledge of pen and paper marks;

• capture and recognize handwriting and drawing;

• enter different types of data without switching input device. For example, text, menu selections, button presses, and screen locations can be entered without changing input device;[5]

• replace the computer keyboard, thus making computers smaller and more portable;

• maintain a visible audit trail of operations;

• maintain a clear figure/ground relationship (Hardock, 1991). For example, marks written over formatted text can be distinguished from the text.

1.3. SELF-REVELATION, GUIDANCE AND REHEARSAL

Despite all of these advantages, pen input and marks have not been widely used. Pen-based interfaces have many difficult technological requirements. Historically, hardware for pen-based systems was too expensive and recognition was not reliable (Sibert, Buffa, Crane, Doster, Rhyne, & Ward, 1987). Given these limitations pen-based applications presented no advantage (in reality, more of a disadvantage) over a mouse-based version of the application.

---

[5] This eliminates homing time between physical input devices but it does not eliminate homing time between graphical devices such as graphical buttons, sliders, etc.

This situation is changing and this change is clearly evident in the marketplace (Normile & Johnson, 1990; Rebello, 1990). Several companies such as Go, Grid, IBM, Apple, Microsoft, and NCR are introducing pen-based systems. Hardware and recognition has improved to the point where pen-based systems are technically possible. Applications such as portable notebook computers and large whiteboard size computer screens make the pen an attractive input device (Goldberg & Goodisman, 1991; Weiser, 1991).

On the surface, it appears that once the recognition and hardware problems are solved, pen-based systems will be successful. However, there is still a serious interface problem when using marks.

1.3.1. The problem: learning and using marks

An intrinsic problem with marks is that they are not *self-revealing.* In contrast, menus and buttons are self-revealing; the set of available commands and how to invoke a command is readily visible as a byproduct of the way commands are invoked. An interface which uses only marks as a means of command entry cannot support *walk-up-and-use* situations. A first time user has no way of finding out interactively from the system what marks/commands are available. This situation is reminiscent of command line interfaces such as the *UNIX* shell or *MS-DOS* where the only information presented by the system is a command line prompt. Some source of information distinct from the process of making a mark must be consulted before commands can be generated.

The problem is even more acute. Not only do users need to know what marks can be made but also when or where these marks can be made. In menu and button interfaces, one can find out when and where a command can be invoked by which buttons or menu items appear active when an interface object is selected. Marks do not have this property.

Is there a problem? Aren't the existing pen-driven systems easy to use and self-revealing? Hybrid interfaces which use both direct manipulation and marks (e.g., the *PenPoint* or *Momenta* interfaces (Go, 1991; Momenta, 1991)) may be somewhat capable of walk-up-and-use. However, only the direct manipulation components of

the interface can be used without external instruction.[6] Manuals must still be used to find out about marks. Hence these system do not solve the self-revealing/marks problem.

The motivation for creating walk-up-and-use interfaces is strong. Successful computer interfaces such as the Macintosh are based on the notion that "nobody reads manuals". These types of interfaces are designed to help a user learn and remember how to operate the interface without explicit external help such as on-line help or manuals (Sellen, & Nicol, 1990). This situation can be viewed practically: a user wants to get a certain task done; this task can be accomplished using a computer tool; the shortest path between the user and task completion is using the tool; a manual will be consulted only if the tool cannot be used directly.

If we expect a worker in the information age to utilize many different applications, a huge amount of training for each application is an unrealistic demand. Users expect interfaces that are consistent and permit transfer of skills from other applications. They also expect interfaces to be self-explanatory and to guide a user in the operation of the application. Thus, the motivation for walk-up-and-use self-revealing interfaces is paramount.

An argument can be made that walk-up-and-use interfaces are not efficient, but this argument misses the point. The reason to make marks self-revealing is so a user can graduate from using the walk-up-and-use techniques to the more efficient marks. Once this graduation has taken place, the user can benefit from the advantages of marks such as efficient articulation and conservation of screen space. The key to the success of this scheme is in how easily a novice can acquire expert skills.

It can be argued that if marks are mnemonic, then no self-revealing mechanism is needed. However, this argument is analogous to using mnemonic names for commands in command line interfaces. This technique relies on the user "being a good guesser" and it has been shown that they are not; command naming behavior of individuals is extremely variable (Furnas, et al., 1982; Carroll, 1985; Jorgensen et al. 1983; Wixon et al., 1983). The more fail-safe approach is to provide an explicit mechanism which explains the command set (Barnard & Grudin, 1988). On the

---

[6] Of course, even some of the direct manipulation components may require instruction.

other hand, other researchers have shown or argued that users commonly agree on certain marks for certain operations (Wolf, 1986; Wolf & Morrel-Samuels; Gould, & Salaun, 1987; Buxton, 1990). Nevertheless, if we wish to use marks for operations which do not have commonly agreed upon marks, a mechanism must be provided for learning about these marks.

We define three design principles to support learning and using marks. We do not claim that these principles are unique. Other researchers have described similar general principles, and many systems have interactions which obey these general principles. However, we define specific design principles for two reasons. First, our application of the general design principles to marks is novel, and second, our own specific definitions help us to explain and discuss the details of the application.

The three design principles to support learning and using marks are self-revelation, guidance, and rehearsal.

Self-revelation

> *The system should interactively provide information about what commands are available and how to invoke those commands.*

When an interface provides information to a user about what commands are available and how to invoke those commands, we refer to this as self-revelation or the system being self-revealing. Menus and buttons, for example, are self-revealing. The available commands and how to invoke those commands can be inferred from the display of menus or buttons. Marks, on the other hand, are not self-revealing because they must be generated by the user.

To ensure that every aspect of a system is self-revealing is a difficult task. For example, displaying menu items may help a user understand what functions are available but does not guarantee that the user will understand, from the display, the mechanics of selecting a menu item.

A common approach to interface design, and the approach that we adopt in this dissertation, is to rely on a user receiving a small amount instruction before starting to use the system. These instructions explain the basic mechanics and semantics of operating the interface. For example, pointing, dragging, double clicking, and the meaning of these actions may be explained. The Macintosh computer uses this

technique. The intention is that with this small set of skills a user can start interactively exploring and learning about the remainder of the system.

The interaction technique developed in this dissertation uses this type of design. A user must be informed, *a priori*, that in order to display a menu the pen must be pressed against the display and held still for a fraction of second. We call this "press and wait for more information". Once users have this bit of information, however, they receive further instructions interactively from the system. In our model of the interface, users can interactively learn about what functions can be applied to various displayed objects by "pressing and waiting" on the objects for menus.

The principle of self-revelation is based on interface design principles and psychological mechanisms proposed by others. Norman and Draper (1986) propose a design principle to "bridge the gulfs of execution and evaluation". Specifically, a designer should make interface objects visible so users can see what actions are possible, how actions can be done, and the effects of their actions. Shneiderman (1987) proposes a similar principle: "offer informative feedback". The principle states that objects and actions of interest should be made visible to the user. Shneiderman claims that this design principle is the basis of direct manipulation interfaces.

The principle of self-revelation is distinct from affordance theory (Gibson, 1979; Gibson 1982). Self-revelation is concerned with absence/presence of information about what functions are available and how to invoke those functions. Affordance theory, in human computer interaction, is concerned with an interface object's appearance suggesting its function (Gaver, 1991). These two notions, however, are related. For example, consider the display of a pop-up menu. The principle of self-revelation dictates, first, that function names or icons must be displayed, and, second, that they are displayed in a menu so that a user knows by convention how to invoke them. Affordance theory, on the other hand, dictates that the name or icon for an item accurately suggests its function, and that the appearance of the menu suggest items are selectable. Correct use of affordances may help reduce the amount of *a priori* instruction a user requires. For example, items in a menu may "look" selectable (they "afford" selection) and therefore the user does not have to be explicitly taught these mechanics.

Guidance

The way in which self-revelation occurs should guide a user through invoking a command.

If an interface actually assists a user in the articulation of commands we refer to this as guidance. For example, in the editor *emacs*, by hitting a "command completion key" while typing a command, emacs will display all the command names that match the partially completed command. In effect, emacs "guides" a user in completion of the command, as opposed to waiting for the command to be completely typed before examining its validity. Another example is selection from a hierarchic menu. In this case, selection of an item guides a user to the next menu.

Guidance does not necessarily have to be triggered by the user. Some on-line help systems prompt the user with information to guide them through a command. The critical point is that in these systems getting or receiving helpful information on how to invoke a command (guidance) does not interrupt the articulation of a command. On the other hand, a system like the on-line manual pages in UNIX violates the principle of guidance. In this case, in order to receive information about what commands are available and how to invoke those commands, a user must terminate or at least suspend the act of invoking a command.

Rehearsal

Guidance should be a physical rehearsal of the way an expert would issue the command.

Rehearsal is the notion of designing interactions such that the physical actions made by a novice in articulating a command are a rehearsal of the actions an expert would make invoking the same command. The goal of rehearsal is to develop skills in a novice that transfer to expert behavior. It is hoped that this leads to an efficient transition from novice to expert.

Many interaction techniques support rehearsal. When the basic action of the novice and the expert are the same for a particular function we can say that rehearsal takes place. For example, novices may draw lines, move icons, or select from menus using the same actions as an expert when there is one and only one way of issuing the command. In many cases, the single way of issuing the command may be suitable for both the novice and expert.

There are also many situations, however, where a single method for invoking a command is not sufficient. The popularity of accelerator techniques is proof of this. Typically, good interfaces provide two modes of operation. The first mode, designed for novices, is self-revealing. Conventional menu-driven interactions are an example of this. The self-revealing component of this mode is emphasized over efficiency of interaction because novices are more concerned with how to do things rather than how quickly things can be done. The second mode, designed for experts, typically allows terse, non-prompted interactions. Command line interfaces and accelerator keys are examples of this mode. However, usually there is a dramatic difference between novice and expert behavior at the level of physical action. For example, a novice uses the mouse to select from a menu whereas an expert presses an accelerator key.

The intention of the three design principles is to reduce this discrepancy in action without reducing the efficiency of the expert and ease of learning for the novice. The basic actions of the novice and expert should be the same. It is hoped that as novice performance develops the skills that lead to expert performance will develop in a smooth and direct manner.

1.3.2. Unfolding interfaces

The principles of self-revelation, guidance and rehearsal support the notion of an *unfolding interface*. An unfolding interface works as follows. Initially, a novice is provided with a small amount of information about how to get information on parts of the interface. For example, double clicking on an object may open it up or "unfold" it to reveal additional functions. Thus, given this key to unfolding objects, a user can explore the interface, learning and using new functions. The intention is that, with experience, exploration and use leads to expert knowledge of the system.

There are other schemes which control the number and types of functions available to a user, for example, *Training Wheels* (Carroll & Carrithers, 1984). These types of systems provide explicit novice/expert modes in which the novice mode has fewer functions than the expert mode. The intention is to avoid confusing a novice with a large set of complex functions. Once the reduced set of functions is mastered, the novice can switch to the larger "expert" set of functions. The major difference between this approach and the notion of an unfolding interface is that an unfolding

interface has no explicit novice and expert modes. An unfolding interface allows users to incrementally add functions to their repertoire.

Marks, self-revelation, guidance and rehearsal can play important roles in an unfolding interface. Unfolding is essentially an inefficient operation. As suggested earlier, by associating marks with "hidden" functions, unfolding can be avoided. For example, rather than double clicking on an object to unfold it and then clicking on a function button, a mark can be made on the object to invoke the function. To help users learn the marks associated with functions, it would be beneficial if unfolding a function also revealed its mark. This is an application of the principle of self-revelation. Ideally, we want the principles of guidance and rehearsal to hold as well; we want to design an interface such that exploration is equivalent to invoking commands, and exploration allows a novice to practice skills that lead to expert behavior.

1.3.3. Solution: ways of learning and using marks

The concerns of this research are interfaces that use marks but are also self-revealing. Therefore, solutions for making marks self-revealing can be classified by how tightly coupled the act of marking is with the act of getting information about command/mark associations.

Interfaces that use marks and only supply information about those marks through off-line manuals are considered to be at one end of a self-revelation continuum. These interfaces are not interactively self-revealing. Interfaces which supply information about marks as a command is actually being articulated can be considered the other end of the self-revelation continuum. These would be considered interactively self-revealing interfaces.

In the following sections we classify solutions based on this criterion. Since interfaces that use marks are still in their infancy there are few pre-existing examples.

Off-line documentation

Off-line documentation consists of manuals which provide information about how marks are used in an interface. Examples of the marks are displayed and text or graphics provides information on their usage. Although this type of scheme is not self-revealing it is of interest because, first, it is the status quo for pen-based

products and, second, it demonstrates the type of information needed for a user to understand marks.

Figure 1.2 shows a section from a pen-based system's manual. Clearly this type of scheme is not interactively self-revealing. However, if the mark set is small, the documentation could be placed directly on the computer in the form of a "cheat sheet". This scheme would be partially self-revealing.

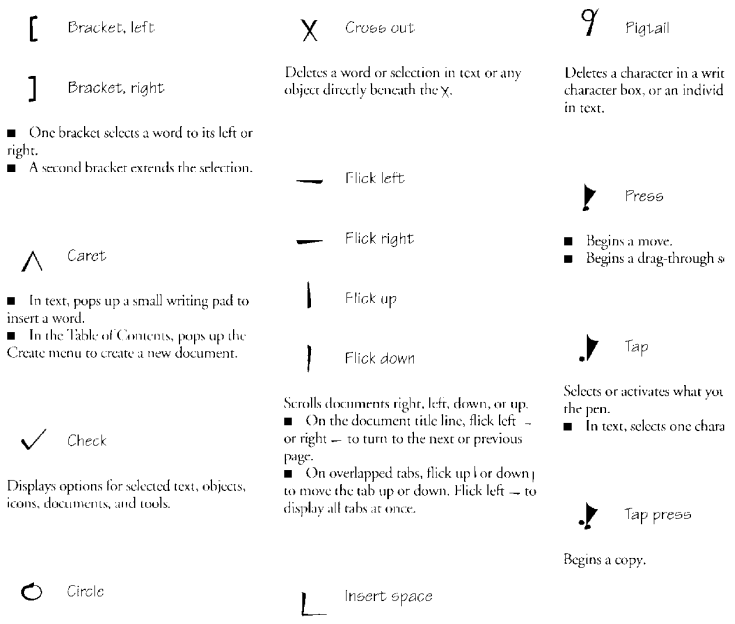


Figure 1.2: Typical off-line documentation for mark commands (PenPoint system, Go, 1991)

On-line documentation

This class is essentially the "on-screen" version of off-line documentation. A user can display manual pages on-screen while the application in question is running. Note that this does constrain the user into suspending the real task of issuing a command while obtaining command information.

Sometimes command information can be found in the application used to train the software module that recognizes marks. Figure 1.3 shows one such example.

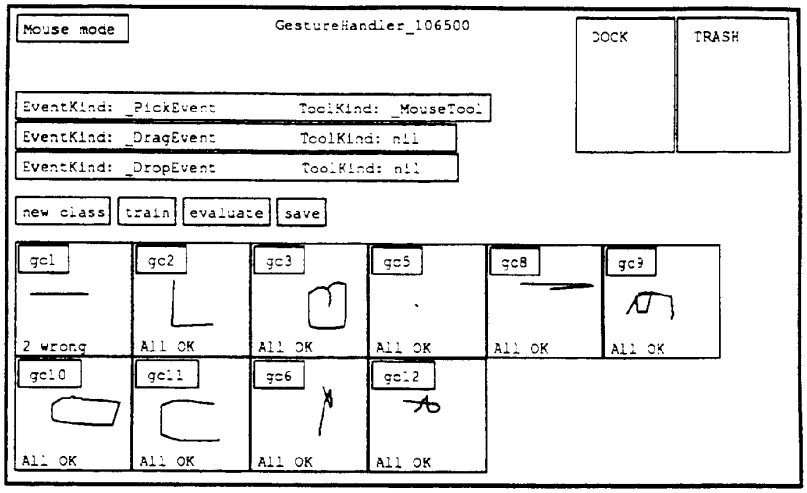


Figure 1.3: Gesture handler window allows inspection of marks associated with a view in Rubine's system. This window is, however, intended for the system programmer. The window shows ten classes of marks but does not shows the semantics associated with each mark. (from Rubine, 1990).

Unfortunately, training interfaces are not designed specifically to deliver this type of information, and the information can be very minimal and confusing to the user.

Microsoft's *Windows for Pen Computing* uses on-line documentation. A special application provides a tutorial which features animations demonstrating marks and editing operations. A user can also practice using the marks on sample text. While the tutorial is effective, a user still has to change context (i.e., switch from the working application to the tutorial application) in order to get information on marks.

On-line interactive methods

On-line interactive methods supply information about marks as one issues a command. Figure 1.4 shows an example where sample marks are displayed beside menu items. Windows for Pen Computing using this technique to a limited degree. This technique relies initially on another interaction method such as menus or buttons to invoke commands. In Figure 1.4, the interaction technique initially relied on is a menu. As the menu is used, it reveals the marks that can be used. Once a user remembers the mark associated with a command, the revealing technique (the

menu) can be bypassed and a more efficient mark can be used. Figure 1.5 shows a system called *XButtons* which also uses this method. In contrast to on-line documentation, an on-line interactive method does not constrain the user into suspending the real task of issuing a command, while obtaining command information.

This method is similar to accelerator keys. Every time a user uses a menu item or button, the mark is seen. Like accelerator keys, the mark can be memorized and used as a shortcut in calling the command. Note that "accelerator marks" are more powerful that accelerator keys because they are not limited to characters on the keyboard, they indicate the object of the requested action by the location of the mark, and they can contain command attributes, such as destinations or modifiers.

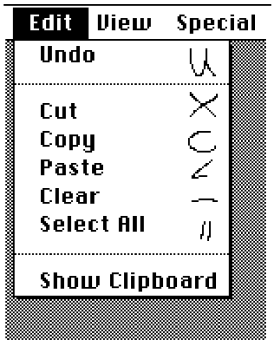


Figure 1.4: An example of "accelerator marks" which allow quick access to menu items similar to accelerator keys.

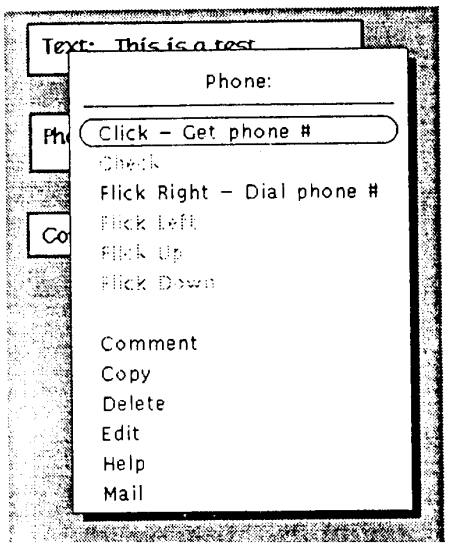


Figure 1.5: XButtons provides a menu which shows what commands are available from a button and the associated marks. A command can be invoked by either a menu selection or by making the mark on the button (Robertson, et al, 1991).

On-line interactive rehearsal methods

This category is similar to on-line interactive methods except invoking a command using the self-revealing technique (i.e., a menu) makes the user physically rehearse making the corresponding mark. In contrast, when using on-line interactive methods, the user does not physically rehearse making the mark (e.g., selecting "copy" from the menu in Figure 1.4 requires a vertical movement, not a hand drawn "C" movement).

Marking menus, the technique focused on in this dissertation, is an example of this class (Kurtenbach & Buxton, 1991). The complete definition of this technique is given in Chapter 2. Figure 1.6 illustrates this technique in the context of creating three simple objects. An expert uses simple shorthand marks to create and place circles, square, or triangles.

If a user is unsure of what marks can be made, the user presses the pen against the display and waits for approximately 1/3 of a second. This signals to the system that no mark is being made and it then prompts the user with a radial menu of the available commands, which appears directly under the cursor. The user may then select a command from the radial menu by keeping the pen tip pressed and making a stroke towards the desired menu item. This results in the item being highlighted (see Figure 1.7). The selection is confirmed when the pen is lifted from the display.

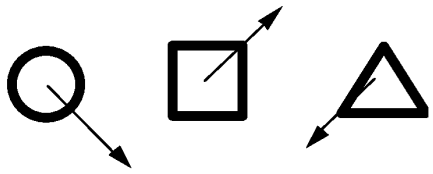


Figure 1.6: An example of the technique using three simple shorthand marks. Three objects can be defined: a circle, square and triangle. A mark which is a simple straight line (shown here with an arrowhead to indicate drawing direction) defines the type of object created, and its placement.

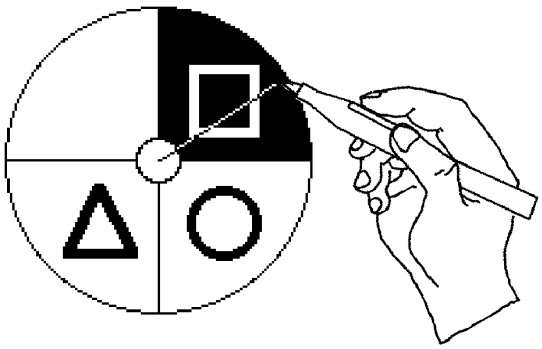


Figure 1.7: A radial (or "pie") menu can also be popped up if the user does not know what commands or marks are available. Rather than drawing a mark as in Figure 1.6 a novice keeps the pen pressed and a menu appears. An object can then be selected from the menu.

The important point is that the physical movement involved in selecting a command is identical to the physical movement required to make the mark corresponding to that command. For example, a command that requires an up-and-to-the-right movement for selection from the pie menu, requires an up-and-to-the-right mark in order to invoke that command. The intention is that selection from the menu is a rehearsal of making a mark.

Other menu layouts can be used for interactive rehearsal methods besides radial menus. Another possibility is a "bull's eye menu" which is a menu that is divided into concentric circles rather than sectors, where each concentric circle corresponds to a different command (Figure 1.8).[7] The corresponding marks are therefore discriminated by length rather than angle. Many more exotic schemes have been proposed and are as of yet unexplored.[8] Chapter 2 presents the motivation for choosing radial menus, and describes in detail the design of marking menus.

---

[7] We thank Professor John W. Senders for this suggestion originally called "donut menus". Professor William Buxton later took great exception to the use of the word "donut" and suggested the more dramatic name of "bull's eye menu".

[8] Dr. Tom Moran has proposed a combination of donut and pie menus. Dr. Stuart Card has proposed a continuous version of hierarchical marking menus.

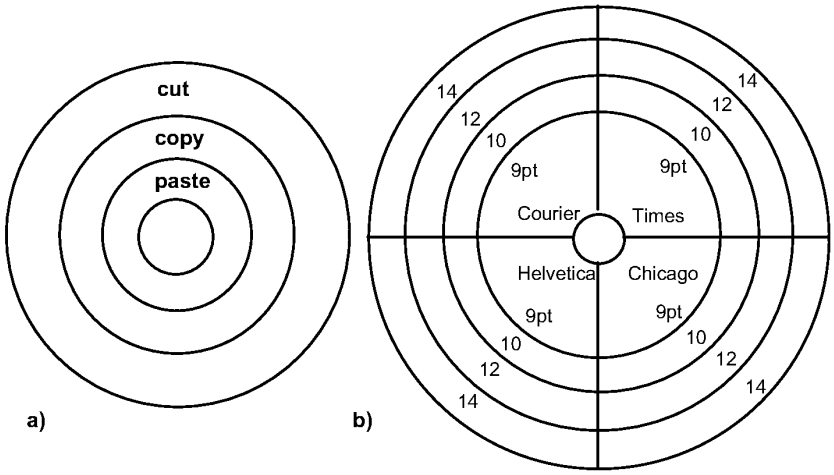


Figure 1.8 Examples of alternate menu styles in which selection will result in a unique marks. a) is a "bull's eye" menu which discriminates by mark length rather than angle. b) is a "dart board" menu which discriminates by length and angle.

1.4. THESIS STATEMENT

This dissertation is an in-depth investigation of marking menus. We present the thesis that marking menus are a valuable interaction technique. When used in the proper situation, marking menus are easy and efficient to use, can be used with different input devices, and integrate well with existing interface techniques. Furthermore, marking menus allow a user to take advantage of writing skills with a pen and attain levels of performance not possible with other interaction techniques. To support this thesis, we present a design for marking menus, evaluate marking menus by means of user behavior experiments, and provide a case study of marking menus in practice. We conclude our investigation by showing how the design concepts of marking menus, self-revelation, guidance, and rehearsal, can be generalized to other situations.

The intention of this investigation is to provide practical guidelines for interface designers interested in using marking menus. With this in mind, we describe when and where marking menus would be an effective technique, and the limitations and properties that must be observed and maintained for marking menus to work well in an interface. We also describe the design principles behind marking menus and give examples of how these principles can be applied to other contexts.

1.5 SUMMARY

This chapter has provided motivation for marks as an interaction technique, described a basic interface problem with marks, set out design principles to solve this problem and introduced an approach, marking menus, which observes these design principles. In Chapter 2 we expand on our motivation for using marking menus and explain in detail the design and design rationale behind marking menus. Chapter 3 reports on an empirical study of the non-hierarchic marking menus. Chapter 3 is a condensed version of a paper that appears in *Human Computer Interaction* (Kurtenbach, Sellen, & Buxton, 1993). Chapter 4 is a case study which reports on how marking menus can be designed into an application and investigates user behavior with marking menus in an "everyday work" situation. Chapter 5 presents an empirical study on the limits of user performance with hierarchic marking menus. Chapter 5 is an expanded version of a paper published in *The Proceedings of InterCHI '93* (Kurtenbach & Buxton, 1993). Chapter 6 describes how we integrated marking menus into a pen-based application and applied the notions of self-revelation, guidance and rehearsal to this application. Chapter 7 summarizes this dissertation and its contributions, and proposes future research.

Chapter 2: Marking menus

In this chapter we expand on our description of marking menus. First, we present a definition of marking menus and the motives for investigation. Next, we describe previous research that is related to marking menus and we identify open research questions and the issues pursued in this dissertation. Finally, we complete our description of marking menus by providing the complete rationale behind our design.

2.1. DEFINITION

A *marking menu* is an interaction technique that allows a user to select from a menu of items. There are two basic ways (or modes) in which a selection can be performed:

menu mode In this mode a user makes a selection by displaying a menu. A user enters this mode by pressing the pen against the display and waiting for approximately 1/3 of a second. We refer to this action as *press-and-wait*. A *radial menu* of items is then displayed centered around the pen tip. A radial menu is a menu where the menu items are positioned in a circle surrounding the cursor and each item is associated with a certain sector of the circle. A user can select a menu item by moving the pen tip into the sector of the desired item. The selected item is highlighted and the selection is confirmed when the pen is lifted from the display. (See Figure 2.1)

mark mode In this mode, a user makes a selection by drawing a mark. A user enters this mode by pressing the pen against the display and immediately moving in the direction of the desired menu item. Rather than displaying a menu, the system

draws an ink-trail following the pen tip. When the pen is lifted, the item that corresponds to the direction of movement is selected. (See Figure 2.1)

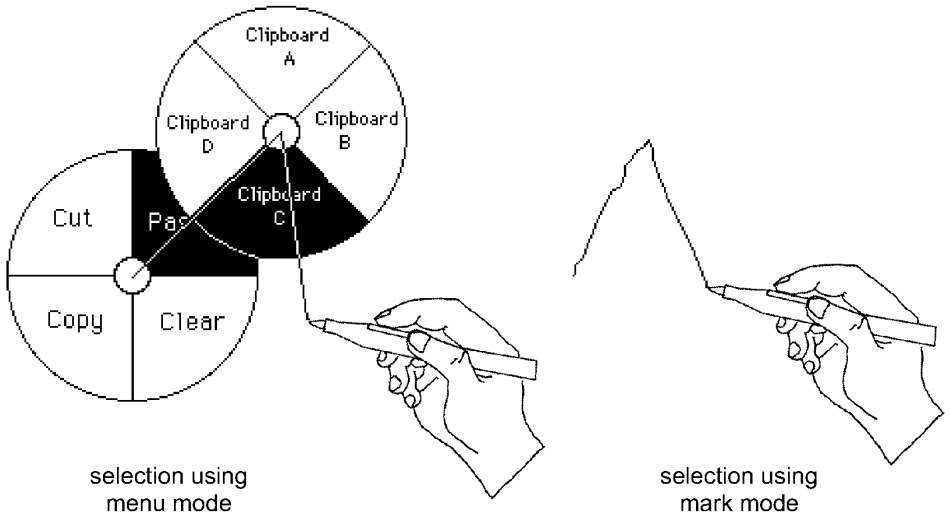


Figure 2.1: The two basic ways of selecting from a marking menu.

The key concept of marking menus is that the physical movement involved in selecting an item in menu mode mimics the physical movement required to select an item using a mark.

Marking menus may also be hierarchic. In menu mode, if a menu item has a subitem associated with it, rather than lifting the pen to select the item, the user waits with the pen pressed to trigger the display of the submenu. The submenu is also a radial menu. The user can then select an item from the submenu in the manner previously described. In mark mode, a user makes a selection by drawing a mark where changes in direction correspond to selections from submenus. Figure 2.1 show an example of selecting from hierarchic menus using menu mode and mark mode.

Using radial menus in this way produces a set of mark which consist of a series of line segments at various angles ("zig-zag" marks). Marking menus which have no hierarchic items produce strictly straight line segments. Figure 2.2 shows an example of a menu hierarchy and the associated marks.

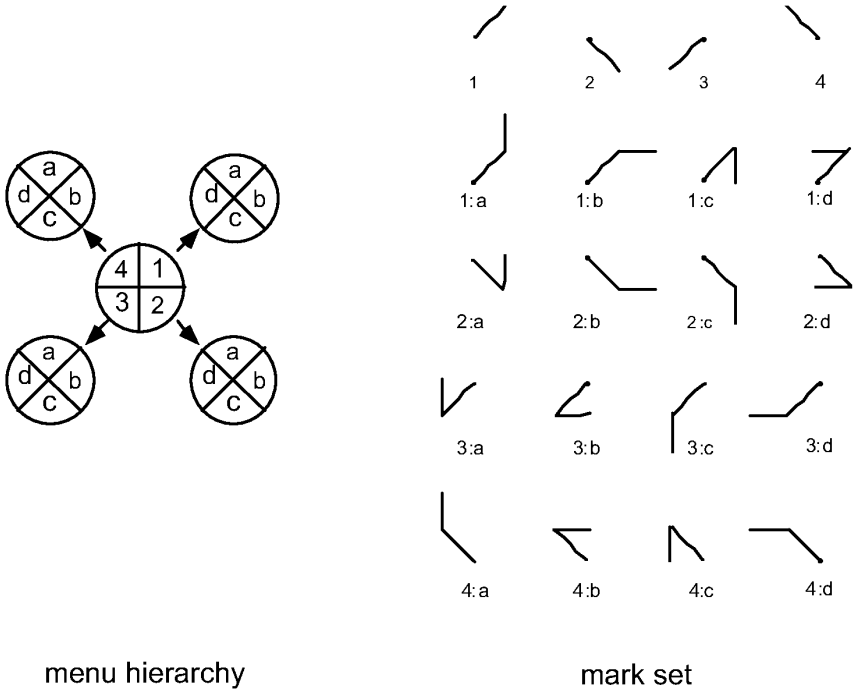


Figure 2.2: An example of a radial menu hierarchy and the marks that select from it. Each item in the numeric menu has a submenu consisting of the items a, b, c and d. A mark's label indicates the menu items it selects. A dot indicates the starting point of a mark.

It is also possible to verify the items associated with a mark or a portion of a mark. We refer to this as *mark confirmation*. In this case a user draws a mark but presses-and-waits at the end of drawing the mark. The system then displays radial menus along the mark "as if" the selection were being performed in menu mode. Figure 2.3 shows an example of this.

Other types of behavior can occur when selecting from a marking menu such as backing-up in a menu hierarchy or reselecting an item in menu mode. Details of the behavior are discussed in Section 2.5.

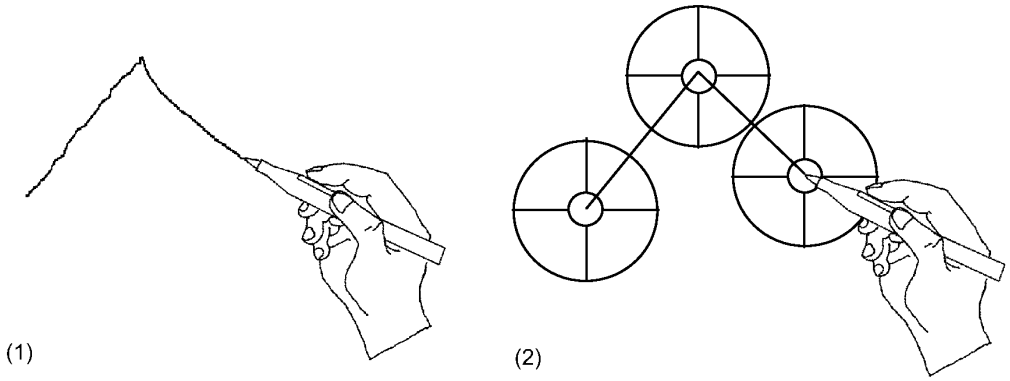


Figure 2.3: An example of mark-confirmation in a menu with three levels of hierarchy. In (1), the user draws the first part of the mark then waits with the pen pressed for the system to recognize the selection so far. In (2), the system then displays its interpretation of the mark and goes into menu mode for completion of the selection.

2.2. MOTIVATION FOR STUDY

We have many motives for studying marking menus; they have advantages over traditional menus; they use marks that are easy to draw and that are easy for computer to recognize; they can be used for functions that have no intuitive mark; they are compatible with different interface styles; and they exploit human motor skills. In this section, we expand on these motivations.

2.2.1. Advantages over traditional menus

One motivation for studying marking menus is that they have many differences and potential advantages over the traditional menus used in current practice. Examples of the current practice in menu design are the pop-up menus or pull-down menus on the Macintosh. With these types of menus, selection is performed by popping up the menu and selecting items by pointing with the mouse. Menu items can also be selected by pressing an accelerator key associated with a menu item. There are several specific advantages marking menus have over these traditional menus:

Keyboardless acceleration

Marking menus allow menu selection acceleration without a keyboard. With traditional linear menus, keypresses must be used to accelerate selection. Marking

menus provide a method of accelerating menu selections when no keyboard is available. This is extremely important for portable, keyboardless, pen-based computers.

Acceleration on all items

Marking menus, if configured accordingly, can permit acceleration on all menu items. With traditional menus, it is common for the application developer to assign accelerator keys to the most frequently used menu items. This assumes that the application designer is able to predict the most frequently used menu items. In many cases, however, it is not possible to accurately predict which menu items will be frequently used, if there is a large variance in the way an application may be used. In contrast, with marking menus, the selection of all items can be accelerated by the user making a mark. The designer does not have to predict, *a priori*, which items will be the most frequently used.

Menu selection mimics acceleration

Marking menus minimize the difference between the menu selection and accelerated selection. Selecting a menu item from a marking menu physically mimics the act of making the accelerating mark. The design intention is to help users become skilled at the movements required for accelerated menu selection. This is dramatically different from traditional menus and accelerator keys where menu selection is performed with the mouse and accelerated selection is performed with the keyboard. In this case selection from the menu in no way physically mimics selection using an accelerator key.

Combining pointing and selecting

Marking menus permit pointing and menu selection acceleration with the same input device. This is an intrinsic property of marks and has been utilized by other researchers (e.g., Coleman, 1969; Rhyne 1987; Wolf & Morrel-Samuels, 1987). In mouse-based direct manipulation interfaces it is very common to point to an object and then select a menu item. If accelerator keys are used, this operation requires coordinating pointing with the mouse and pressing on the keyboard. With a marking menu, not requiring a hand to be on the keyboard frees the hand to control other input devices or perform auxiliary tasks such as controlling a VCR transport or turning the pages of a book.

Spatial mnemonics

Marking menus use a spatial method for learning and remembering the association between menu items and marks. In contrast, traditional menus and accelerator keys, rely on symbolic mnemonics to help users remember the associations between menu items and keys. Due the limited number of symbols on a keyboard, mnemonics often cannot be established between all menu items and their accelerators keys. This results in menu item/key associations that may be arbitrary or inconsistent. Marking menus avoid this problem by relying on a consistent method to establish mnemonics: the shape of a mark corresponds to the spatial layout of a menu item in the menu hierarchy.

2.2.2. Ease of drawing and recognition

Marking menus use a very simple set of marks consisting of straight and zig-zag marks. This simple set of marks has three advantages. First, these types of marks are easy and fast to draw and are therefore suitable for accelerated performance. Ease of drawing is especially important when drawing precision is hampered by imperfect pen/display technology. Second, computer recognition of these types of marks can be reliable, fast and user independent. The recognizer requires little processing power and no training. Third, any interface designer, by using marking menus, can make use of some of the advantages of marks without having to design their own mark symbols. Of course, it is still necessary to design the layout of the menus.

The single contiguous marks in marking menus have several advantages. Other types of marks which require multiple non-contiguous pen strokes create many problems. Recognizer design is more complicated when groups of strokes must be recognized. This is referred to as the *segmentation problem*. Sometimes groups of strokes are distinguished by constraining the user to put all the strokes associated with a mark in a certain region. Alternatively, strokes may be grouped by time. This constrains the user to momentarily pause between making different marks. With a marking menu mark, a user is not constrained by timing, size of mark, or location. Recognition takes places the moment the pen is lifted.

The marking menu mark set does have disadvantages. First, a designer has no choice in the shape of the marks (besides what can be controlled through the layout of the menus). Fortunately, marking menus do not prohibit the use of other mark

sets and mark recognition techniques (see Chapter 6 for a detailed discussion of this issue). Second, the size of the mark set is limited by a user's accuracy at drawing lines at various angles. Third, the mark set is not particularly expressive. The angle at which the stroke is drawn is used to define the type of mark. The line must also be somewhat straight. This leaves starting point, ending point and temporal information about how the line was drawn to be used as additional information encoding parameters. In contrast, other mark vocabularies permit many more parameters to be controlled by the shape of the mark (Makuni, 1986). Nevertheless, we have discovered that the limited set of parameters of a marking menu mark can be quite useful (see Chapter 4).

2.2.3. Marks when no obvious marks exists

Researchers have shown or argued that users commonly agree on certain marks for certain functions (Wolf, 1986; Gould, & Salaun, 1987; Morrel-Samuels, 1990; Buxton, 1990). However, we believe that there are many situations where invoking a function with a mark could be beneficial but no commonly agreed upon mark exists for the function. This is similar to icon design where some functions have no intuitive icon. For example, there is no "natural mark" for "change pen width to thin". Marking menus might work well in these types of situations because the menu can provide textual or pictorial explanations of functions while the mark for the menu item provides a quick way to invoke the function.

2.2.4. Compatibility with unfolding interfaces

Marking menus are compatible with unfolding interfaces (described in Section 1.3.2). The intention is that menus pop up to self-reveal or unfold functions and the marks provide way to efficiently invoke the functionality. Guidance and rehearsal are intended to help a novice learn the efficient way of invoking a function.

2.2.5. Compatibility with existing interfaces

Marking menus are compatible with popular input devices and interface paradigms. First, the type of marks used can be reasonably drawn with a mouse (Chapters 3 and 5 explore this issue in detail). Second, since traditional menus are created by the application calling library routines, by replacing the library routines, marking menus could be used in place of pop-up menus without changing a single line of application code or changing application functionality. Finally, marking menus can

extend existing dialogue styles without major changes to an interface paradigm. An example of this is *HyperMarks*, developed by the author (Kurtenbach & Baudel, 1992), which is a *Hypercard xcommand* that supports marking menus in Hypercard (Apple Computer, 1992). When a marking menu is used from a Hypercard button, the Hypercard button still retains its single function when pressed. However, if the button is kept pressed, a marking menu pops up with more commands. A user can select from the marking menu using menu mode or marks. In this way, the function of a button can be extended.

Marking menus can be effective because they are a pop-up interaction technique. When displays become small or very large, marking menus can be effective. On large displays, a mark or a menu selection can be made at a user's current location without a long trip to a menu bar or tool pallet. On small screens, since both the menu and mark "go-away" once performed, no valuable screen space is consumed.

2.2.6. Novices, experts, and rehearsal

Marking menus are intended to support both the novice and expert user. The intention is that a novice uses menu mode and an expert uses the marks. Menu mode can provide the self-revelation and guidance needed for a novice to invoke a command. The marks can provide efficient interactions for experts.

Marking menus are also intended to support the transition between novice and expert. Selection in menu mode provides the user with rehearsal for making a mark. In essence, using the menu trains a novice to use marks. We believe that rehearsal helps in learning the association between mark and command.

There are other menuing schemes which support the novice and expert and the transition between the two. For example, the Macintosh supports novices by providing menus and supports experts by providing menu accelerator keys. The transition between novice and user is supported by the user being reminded of the keystrokes associated with particular menu items every time a menu is displayed. This is done by having the names of the accelerator keys appear next to menu items in the menu. However, actually using an accelerator key is avoidable. The user can always just select from the menu. Furthermore, this is easiest because the user is already displaying the menu. The end result is that accelerator keys are sometimes not used even after extensive exposure to the menu. With marking menus the user is not only reminded, but rehearses the physical movement involved in making the

mark every time a selection from the menu is made. What makes marking menus unique from the accelerator key scheme is that rehearsal is unavoidable. We believe this helps in learning the association between mark and command.

2.2.7. Utilizing motor skills

The idea of using physical rehearsal to train novices to become experts is a unique concept and is worth investigating for pedagogical reasons. Marking menus purport to reduce the cognitive load of memorizing mark/command association by relying on muscle memory (since each mark/command is a distinct physical movement). This technique is similar to the approach used in the Information Visualizer Project (Card et al, 1991). The Information Visualizer relies on low level sensory input processing such as depth or motion perception to reduce the burden on higher cognitive processes in visualizing information. Marking menus can be thought of in a similar manner. It is believed that low level sensory output processes (muscle memory) are used to reduce the load on higher level cognitive processes. We explore this issue in this dissertation.

2.2.8. "Eyes-free" selection

Selection by a distinct physical movement with a marking menu lends itself to "eyes-free" selection. For example, most of us can draw the eight directions of a compass without looking. Eyes-free selection is useful in situations where a user's visual attention must be on something other than the selection process, for example, selecting commands while watching a video tape. An eyes-free selection technique is also extremely valuable to the visually impaired.

2.3. RELATED WORK AND OPEN PROBLEMS

This dissertation develops and explores the use of marking menus. There is no previous research on this technique, *per se*, however, marking menus are based on radial menus (see Section 2.1 for the definition of radial menus). Therefore, research on radial menus is relevant. The most widely used instance of a radial menu is the pie menu (Hopkins, 1991). A pie menu is a radial menu where the visual representation of the menu resembles a sliced pie. Other types of visual representations are possible, for example, we have developed an alternative representation for a radial menu which does not look like a pie (see Figure 2.12).

Two instances of radial menus are pie menus and command compasses. We now describe these two techniques, contrast them with marking menus, and report on the current state of research on their design and usage.

2.3.1. Pie menus

To date, there is little research on pie menus. The origin of pie menus can be traced back to radial menus proposed by Wiseman, Lemke, & Hiles (1969). Since then, research on pie menus has mainly been concerned with menu layout and suitable applications (Hopkins, 1991; Hopkins, 1987). The only empirical study of pie menus investigated menu item selection time and error rates for 8-item menus but concentrated on comparing them to linear menus (Callahan, Hopkins, Weiser, & Shneiderman, 1988). It was found that selection from pie menus was significantly faster (15%) and produced marginally significant fewer errors (42%) than linear menus. The experiment also investigated the effect of using menu items with a natural linear ordering (i.e., "First", "Second", "Third", etc.), with a natural radial ordering (i.e., "North", "North-east", "East", etc.), and with an unclassifiable ordering (i.e., "Center", "Bold", "Italic", etc.). Callahan et al. hypothesized that certain types of menus (pie or linear) would perform better with items that have a certain type of natural ordering (radial, linear, or unclassified). A marginally significant correlation was found between menu types and types of orderings. The weak correlation occurred because selection time means for the pie menus were lower even on items with natural linear orderings. Results also showed that unclassified menu items produced significantly slower selections than ordered menu items regardless of menu type.

What has not been extensively studied is the claim that muscle memory for different gestures plays a helpful role in menu selection. Anecdotal evidence from designers of pie menu systems suggest that item selection from a menu hierarchy is possible without displaying the menus after practice (Hopkins, 1987). Not only was unprompted selection possible but it was also desirable for efficiency reasons.

Unprompted selection is supported in pie menus by a technique called *mousing-ahead*. Mousing-ahead means the user does not have to wait for the system to display the menu before moving the cursor to make a selection. As the user moves the cursor, the input system buffers cursor location data. When the menu is finally displayed, the system reads the buffered data and analyzes it as if it were generated

with the menu displayed. The system then immediately selects a menu item and removes the menu. In this way a user can make a selection without waiting for the menu to display (in effect, the mouse is being operated "ahead" of the display, hence the term mousing-ahead). Hopkins' implementation is slightly more sophisticated than just described. Menu display is suppressed until the user stops moving the cursor.

On the surface, it appears as if a marking menu is a pie menu with an ink-trail added to cursor. However, there is a major difference in the way the two techniques behave. Marking menus, depending on the context, may use sophisticated recognition. Marking menus analyze the path of a cursor as a mark, looking for certain features. If the interface recognizes other types of marks, a mark has to "look like" a marking menu mark before it can select from the menu. For example, suppose an interface recognizes a "C" mark (e.g., "C" triggers the copy command) and also marking menu marks (i.e., zig-zag marks). If mousing-ahead was used, the "C" would select the bottom item of a menu (assuming the user started drawing from the top of the "C"). With marking menus, the recognizer identifies the mark as a "C" and not as a zig-zag mark. Chapter 6 discusses in more detail, issues of integrating marking menu marks with other types of marks.

As a consequence of mark recognition, marking menu marks can be performed more casually than mousing-ahead movements with pie menus, especially with hierarchic menus. Mousing-ahead on pie menus must be an exact imitation of cursor movement used when selecting with the menu displayed. Marking menus, on the other hand, recognize the shape of the mark, independent of size and therefore the user can be more casual when drawing marks as opposed to mousing-ahead. There are designs where mousing-ahead can be made independent of movement size but, in general, this is not possible. See Section 2.5.6 for a detailed discussion of these issues.

The visual difference between marking and mousing-ahead is that marking leaves an ink-trail after the cursor, whereas mousing-ahead does not. We believe that, without an ink-trail during selection, a user must visualize selection from the menu. With an ink-trail, the user does not have to visualize selection, but rather remember the mark associated with a menu item and then correctly draw the mark. We believe the ink trial provides feedback which helps the user to correctly draw the mark.

2.3.2. Command compass

An interface mechanism very similar to a marking menu is the command compass used in the Momenta pen-based computer. Figure 2.3 shows how the command compass is used to move text.

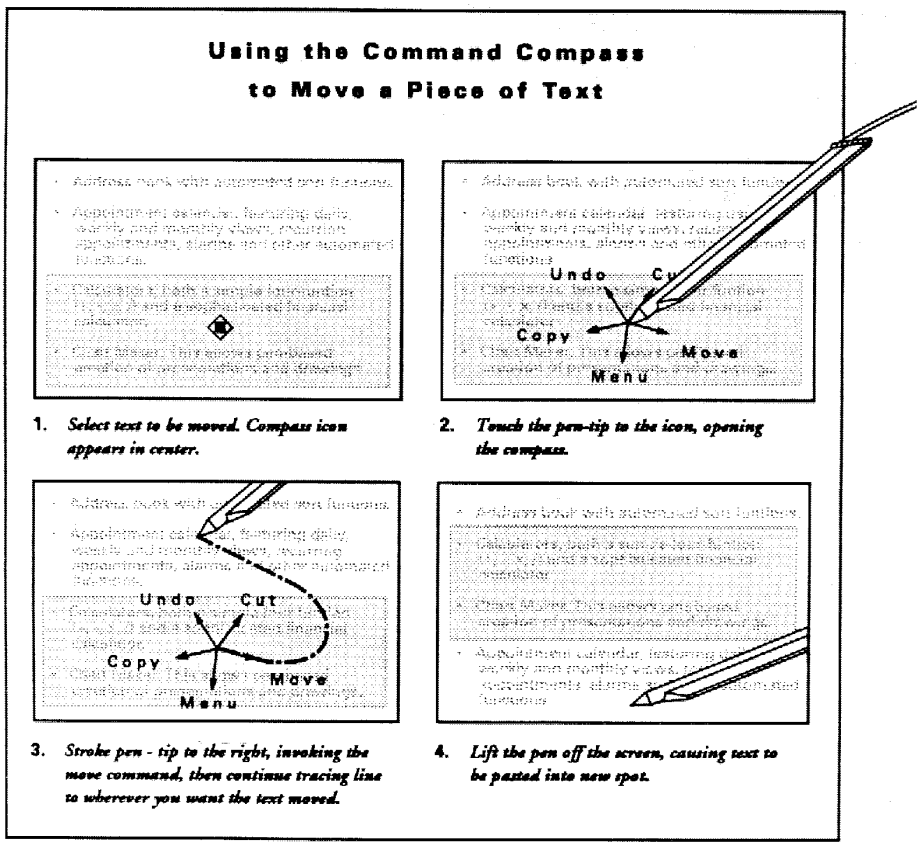


Figure 2.3: The Momenta Command Compass (Momenta, 1991).

There are several differences between the command compass and marking menus. First, the command compass does not permit reselection. Once the pen is moved in the direction of a command, that command is immediately selected. Second, an explicit unprompted selection mode is not provided. No ink-trail is provided and unprompted selection relies on mousing-ahead (or "penning-ahead", since Momenta is a pen-based computer). While the Momenta interface uses marks, the

command compass does not utilize marks. Finally, only one type and size of command compass is used. No hierarchic command compasses are supported.

The subtle difference in the way selection is done with a command compass versus selection with a marking menu affects the type of interactions each technique can support. With marking menus command selection occurs after a sector has been moved into and the pen lifted. With the command compass, command selection is done the moment a sector is moved into. Thus when selection occurs, the user is still in a physical mode (keeping the pen pressed). This physical mode can be used to express more parameters for the command, hence, physically pairing a command verb and its parameters. This is, of course, at the expense of not permitting reselection.

2.4. RESEARCH ISSUES

The ultimate goal of this research is to create a useful interaction technique. To attain this goal, several things must be accomplished. First, we must create a design for marking menus. Next, this design must be evaluated to determine its limitations and possible applications. From these evaluations, we can refine our design and develop recommendations for interface designers about when, where, and how marking menus can be beneficial. Given these goals, research issues surround the following question: what characteristics of marking menus do we need to understand to effectively incorporate this mechanism into the interface?

The most immediate question about marking menus is: how many items can be placed in the menus before it becomes too difficult to make selections using marks? Common sense tell us that parameters governing this aspect are articulation accuracy (i.e., how precisely can a human draw directional strokes), and human memory limitations (i.e., how quickly can a human learn and remember associations between menu items and marks). Other issues concern how hierarchic structure affects selection performance, how command parameters can be attached to marks, and how the design can be varied to accommodate the constraints of an interface. The following sections expand on these issues.

2.4.1. Articulation

Accuracy in selecting menu items and in marking is limited by the human motor system and the input device being used. This constrains the number of items that can be placed in a marking menu. *Articulation* refers to the motor system activities associated with selecting from a menu or making a mark, not memory activities like recalling the mark associated with a menu item. For example, suppose a user remembers the mark for a desired menu item. Can the user draw the mark accurately enough to select the menu item? In other words, can the user successfully articulate the mark once it is remembered?

Many factors may affect the success of articulation:

The type and characteristics of the input device. While the pen appears to be a natural input device for marks, operating marking menus with other types of input devices is also desirable. Thus, it is of interest to study users' performance not only with a pen but also with other popular types of input devices.

The number of items in a menu. As the number of items in a menu increases, the size of the menu items decreases and therefore pointing to them will become more error-prone and slower. Using a mark for selection should behave in a similar fashion. Precision of marking must increase as the number of items increases.

The type of articulation feedback provided. Feedback helps a user verify that a selection is being successfully articulated. For example, highlighting a menu item provides feedback. Supplying an ink-trail is another form of feedback, but is perhaps less salient. Finally no ink-trail (i.e., just the pen's or cursor's movement) provides even less feedback.

Chapters 3 and 5 investigate the effect of these factors through empirical experiments which measure speed and accuracy of selection when using marking menus. The results from these experiments are then interpreted to produce design guidelines.

2.4.2. Memory

Another aspect of marking menus concerns human memory. Using a mark to select from a marking menu involves, first, learning the association between menu item and mark, and then, recalling the association from memory before articulating the

mark. There are several ways in which learning and recall can occur. For example, a user can memorize the association by rote memory ("this mark invokes this command"), or a user can reconstruct a mental image of the spatial layout of the menu or process of selection.

There are other factors affecting learning and recall. Differences in the angles between items must be memorable enough so the angle can be reproduced in drawing the mark. For example, a user may remember an item was the third from the top in a very densely packed menu, but the angular difference between items may be so small that it cannot be remembered precisely enough.

Whatever technique is used to remember the mark/item association, the exact limitations of marking menus relative to the limitations of human memory is a very complex question. Human memory in some situations can be considered almost infinite. For example, humans are capable of memorizing many complex symbol systems such as languages. With enough practice, the paths through extremely complex hierarchies of menus could be memorized and recalled. The question of how quickly one "learns the marks" depends on many variables: frequency of use, presence or absence of mnemonics or metaphors, menu layout, intelligence, motivation, application, etc.

Determining hard figures for "learning time" or "maximum number of items" relative to human memory is not possible. These measures depend largely on the user and the application. The intent of this research is to come up with guidelines that help designers exploit aspects such as frequency of use, metaphors, and menu layout to help make marking menus easier to learn.

In the case of marking menus note that training time is not as critical as with other interface techniques because a user "trains on the job". A user of marking menus does not have to spend time training before the selections can be performed. A novice can use the menus while a forgetful expert may occasionally have to use the menu. In either case, the user will still be performing "training on the job".

Do users learn and use marking menus the way the design suggests? The three modes of interacting with a marking menu (menu, mark-confirmation and mark modes) are intended to support the transition from problem solving to skilled behavior in a user. Card, Moran and Newell (1983) suggest that novices exhibit problem solving behavior ("how do I do this?") and experts exhibit skilled behavior

(an expert knows how to solve the problem and does it efficiently). Rasmussen (1984) further refines this notion to include a middle step called rule-based behavior. Informally, rule-based behavior can be thought of as the user explicitly thinking "in order to do this I must do this". As Figure 2.3 shows, these stages of behavior can be mapped to the three modes of marking menus. The intention is that these modes are designed such that use of one mode builds the skills for the next mode and this assists in making the transition between modes. Do users actually behave this way with marking menus? If not, what sort of behavior is occurring and why?

We examine these issues of learning and remembering through empirical experiments (Chapter 3 and 5), and user behavior case studies (Chapter 4). The empirical experiments reveal learning curves and insights into the sort of menu structures that assist in learning and remembering menu layout and marks. A case study of user behavior using marking menus in a real application investigates learning and behavior patterns when marking menus are used in "everyday work" situations.

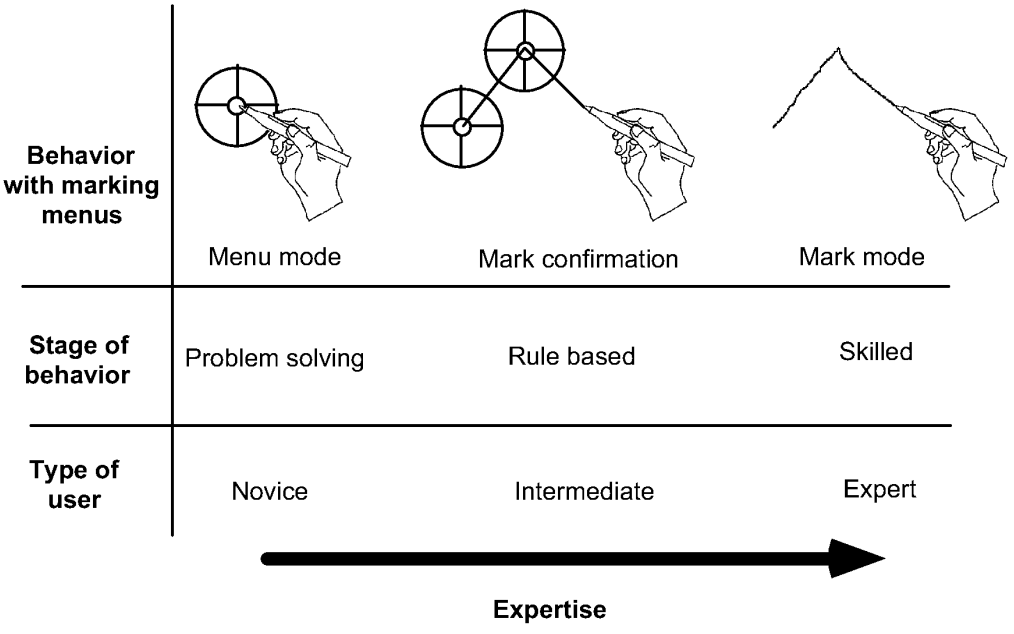


Figure 2.3: The relationship between stages of behavior, type of user, a user's behavior with a marking menu and expertise.

2.4.3. Hierarchic structuring

Another question concerns the effect that the structure of the menu hierarchy has on user performance. Specifically, how is user performance affected when breadth or depth is increased? (Depth is the number of levels in a hierarchy of menus; breadth is the number of items in a menu.)

Most of the research on hierarchic structuring of traditional menu systems focuses on depth versus breadth. This research can be divided into two types of studies: (1) theoretical models describing menu structure and user performance, and (2) empirical studies of menu usage. The theoretical studies concern models that describe menu search performance based on structure. From these models, structures that optimize search-time can be produced. The empirical studies attempt to verify the theoretical models, and estimate search time and error rates. These research efforts have addressed some basic issues concerning depth versus breadth.

The *navigation problem* (getting lost or using an inefficient path to find a menu item) becomes more likely as depth increases. Snowberry, Parkinson and Sission (1983) showed that error rates increased from 4% to 34% as menu depth increases from one to six levels.

Despite the problem of errors, there are several reasons to increase menu depth: *crowding*, *insulation* and *funneling*. Crowding refers to the problem of not having enough space on the screen to simultaneously display all the menu items. Insulation refers to the hiding information in deeper menus to protect a user from information overload. Funneling refers to the structuring of menus such that the hierarchy helps a user "narrow down" the choice and access items more quickly than using a flat menu structure.

Lee and MacGregor (1985) examine the tradeoff between funneling and response-execution time. Assuming all items were viewed before a selection is made, they found that optimal breadth was between 3 to 8 items per menu level depending on user response time and computer processing response time. Depth was effective when user response times were fast and computer processing time per option was slow. If it is assumed that the search terminates on average halfway through the items, then the optimal breadth is between 3 to 13 items at each level. These results

should be tempered by the fact that they are based on a theoretical model and not on empirical user tests.

If meaningful groupings of items are used, Paap and Roske-Hofstrand (1988) show that optimal breadth at any level tends to be in the range of 16 to 36 and sometimes as high as 78 for traditional menu systems depending on human and computer response time. In terms of marking menus, these ranges are well outside the maximum number of items that can be selected with a mark. This raises the issue that reduction of breadth in a marking menu may increase the performance of marking but degrade the efficiency of menu selection in the menu mode.

Menu search time increases monotonically with depth (Landauer & Nachbar, 1985). This produces a log-linear relationship between search time and number of menu items. Kiger (1984) also found that performance (time and accuracy) decreased as depth increased further confirming that depth presents navigation problems to users.

Kiger also included error recovery in his analysis. This increased the variance in search time from 6 seconds to 20 seconds. Since error recovery occurs in the real world, this study more realistically characterizes the costs associated with hierarchical structuring. Kiger tested five types of hierarchical structures varying the depth from two to six levels and the breadth from two to eight items.

Performance can vary at different levels of the hierarchy. Snowberry, Parkinson and Sission (1983) report on error rate versus hierarchy level in a six level hierarchy. A higher proportion of errors occurred at the top two levels of the hierarchy than at the bottom two despite the fact that every level was a binary choice. The explanation for this is that higher level items are more abstract and therefore more subject to misinterpretation. Kiger also found that search times gradually become faster as a user came closer to the goal item. Other studies have revealed opposite results—better performance occurred at top levels (Allen, 1983). The explanation offered for the differences is that users were much more familiar with the top level items than the lower level items. This lends support to the notion that performance, structure, and item semantics in menus are intimately related.

Paap and Roske-Hofstrand (1986) point out that users restrict navigation because the menu structure has semantics or because they have experience with the menu. Both Card (1982), and McDonald, Stone, & Liebelt (1983) report that effects of

organization disappear with practice. In other words, with practice, users navigate directly to the desired menu item. With experience, users move from a state of great uncertainty to one of total certainty. This lends support to the hypothesis that marking menu users will use marks with practice.

The previous research on depth versus breadth in menus indicates two important points relative to marking menus. First, users need to explore to make selections from menus with which they are not familiar, and the semantics associated with the structure has an effect on human performance. Marking menus behave somewhat like traditional menu systems when used in the menu mode (i.e., users can see item names and navigate through the hierarchy). Therefore, we can assume that the research findings mentioned above are applicable in menu mode. Second, once familiar with the menu structure, users of traditional menu systems want to directly select an item. In other words, users no longer require a menu. This behavior bodes well with using a mark to select from a marking menu.

Since the previous research in this area is somewhat applicable to the menuing mode of marking menus, the open research issues concern using mark mode to access hierarchic marking menus. The main issue is the effect of breadth and depth on user performance when using marks. Specifically, how deep and how wide can menus be made before marking becomes too slow or error prone? What sort of structuring makes mark articulation easier? For example, selection using marks from a menu with 16 items seems difficult. Selection from a menu with two levels of four item menus (16 items in total) seems more reasonable. In Chapter 5, we examine the effect of breadth and depth on marking by means of an empirical experiment on human performance using marks to select items from hierarchic marking menus.

2.4.4. Command parameters and design rationale

Besides the angle of a mark specifying the command verb, other aspects of a mark can express command parameters. For example, a mark's starting point, ending point and size can all contribute to command semantics. The question is how can these aspects of a mark be exploited in an interface? Issues of this type are examined in a case study which involved implementing marking menus in a real application (Chapter 4).

Subtle differences in design may have a profound effect on the way in which marking menus can be used. For example, a design that uses selection upon sector entry (e.g., the Momenta command compass) must be used differently than a design that uses selection on pen release (e.g., marking menus). These small design details can have a large impact on a design's ability to support hierarchic menus, command/parameter pairing, and reselection. In section 2.5, we describe this design space and present a design rationale for marking menus.

2.4.5. Generalizing self-revelation, guidance and rehearsal

Marking menus provide self-revelation, guidance, and rehearsal for the particular class of mark. Specifically, this is the type of mark that is created as a byproduct in selecting from directional menus. We referred to this class of marks as "zig-zag" marks. A pen-based application may also use other types of marks (e.g., editing symbols). There are two issues concerning the relationship of marking menus and other types of marks. First, can marking menu marks be integrated with other types of marks? Second, can a mechanism be developed to provide self-revelation, guidance and rehearsal for other types of marks?

A major advantage of marks is the ability to use features of a mark as additional command parameters. For example, a copy mark not only specifies that a copy command should be executed but also specifies what should be copied and to where it should be copied. How self-revelation, guidance and rehearsal can be provided for this type of information is an open question. Chapter 6 addresses this question.

2.5. DESIGN RATIONALE

This section presents the design rationale behind marking menus. First, the fundamental goals and the space of the design are defined. Next, an explanation and taxonomy of design options is presented. Finally, the rationale for choosing a particular set of options for the design of marking menus is given.

2.5.1. Fundamental design goals

The fundamental design goals of marking menus are:

• in the mark mode, speed of selection is emphasized over the self-revealing features.

• in the menu mode, self-revelation and guidance are emphasized over speed of selection

• in menu mode movement must be as close as possible to a rehearsal of marking. Ultimately, using the menu must facilitate learning the marks.

The last goal dictates that marking must mimic selecting in menu mode. Furthermore, marks must be distinguishable from one another. This provides a further goal for the design:

• selection in menu mode must create a unique path which can be reliably recognized by a computer.

We next examine the types of designs that address these goals.

2.5.2. The design space

In the most general sense, the design space can be described as: "discriminating selections from menus by cursor movements". Linear menus, array menus, and radial menus all fall into this design space. Linear menus are menus where the items are laid out in sequential linear fashion (top to bottom, or left to right). Array menus are menus where the items are laid out in both a top to bottom and left to right fashion. Radial menus are menus where the items are laid out in a circle. In these types of menus, the position of the cursor ultimately determines the item selected. A design that does not fit in this class would be menu selection based on time. In this case, the computer cyclically displays each menu item and the user presses a button when the desired item appears. This type of menu selection is often used in interfaces for handicapped users.

In this space, selection is performed relative to a starting point and the amount and direction of movement determines the selection being made. For example, in a linear menu, when the cursor is initially placed on the first item in the list, selection is determined by how far the cursor is moved down the menu.

Within this design space we are only considering designs in which menu selection is a physical rehearsal of marking. We want each movement path traced by a menu selection to be unique relative to the other movement paths involved in selecting

from the menu. This will result in an unambiguous language of movements (or marks when the cursor leaves an ink-trail).

Within this design space we can identify several important design issues. These issues are discrimination, control, selection, display, backing up, and aborting.

2.5.3. Discrimination method

Discrimination method is defined as the type of movement used to discriminate selections. This can be either angle, length, or a combination of the two. Figure 1.8 shows a menu that uses length, and another menu that uses the combination of length and angle. Whether humans are better at discrimination by length or by angle is an open question.[9] In our context, discrimination by angle is preferable to discrimination by length for two reasons: efficiency, and scaling and rotation issues.

Under certain conditions, discrimination by angle (radial menus and angular marks)

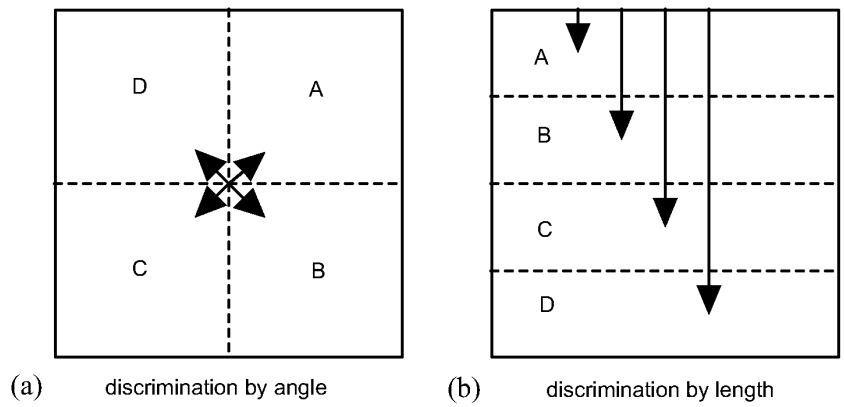


Figure 2.4: An example where discrimination by angle makes selection faster than discrimination by length. The lines with arrow heads show the movement needed to select an item. In the discrimination by angle case, selection of any item requires a movement of distance d. In the discrimination by length case, assuming all items are accessed with the same frequency and distance is equivalent to movement time, the average selection time will be 2L, where L is the height of a menu item. Assuming d is 0.5L, selection is four times faster with discrimination by angle.

allow faster selection than discrimination by length (linear menus and linear marks). First, because all the menu items are equidistant from the center of the menu in a radial menu, selection time is approximately the same for any item in the menu. In contrast, with linear menus, the first item can be selected more quickly than the last item in the menu. Figure 2.4 shows an example which compares a four-item radial menu and a four item linear menu. As described in Section 2.3.1, Callahan, Hopkins, Wieser, & Shneiderman (1988) have empirical evidence that eight-item radial menus are 15% faster and produce 42% fewer errors than eight-item linear menus. Treating selection from a radial menus as a one dimensional pointing task, and assuming that the amount of area used by a radial menu and a linear menu are the same, it can be shown that target size in a radial menu will always be larger than target size in a linear menu. For example, in Figure 2.4, the target size in the radial menu is the diagonal of an item. In contrast, target size in the linear menu is the height of an item. However, as the number of items increase in a radial menu, pointing to the narrow slices will become more difficult. To compensate for this, users will have to move farther away from the center, thus slowing their selection time. Determining the point where performance with a radial menu will degrade to the performance level of a linear menu is an open problem. Current research on two dimensional pointing (Mackenzie & Buxton, 1992) only deals with rectangular targets and therefore cannot be directly applied to radial menu slices.

There are also issues related to mark-based interfaces that make discrimination by angle preferable. Angular marks are preferred over linear marks because an angular mark can be scaled without changing its meaning (or, rather, changing the item the mark selects). In terms of a mark-based interface this means that a user is not restricted to draw the marks at a prescribed size. For example, a small "L" shaped mark would have the same meaning as a large "L" shaped mark. This is not the case with marks that are discriminated by length.

However, the meaning of angular marks changes if the mark is rotated. Rotating a horizontal to the right mark 45 degrees will cause it to be interpreted as a down to-

---

[9] It should be noted that discrimination can be performed at the reading or at the writing level (i.e., perception versus production of marks). These are significantly different problems. This dissertation examines production of angular marks. See Westheimer & McKee (1977) for a discussion of the perception of angle and length.

the-right mark by the system. In contrast, linear marks are not affected by rotation (i.e., a bull's eye menu. See Figure 1.8).

Discrimination by angle better reflects the way marks are interpreted in everyday life. Marks are generally insensitive to scaling but sensitive to rotation. For example, a small "l" has the same meaning as a large "l" but if it is rotated 90 degrees it perhaps takes on the meaning of "dash".

There is also the issue of *C:D ratio*. C:D ratio is defined as the ratio between the amount of movement of the input device (Control) and the amount of movement this imparts to the cursor (Display). On a pen-based system, the C:D ratio is constant and one to one because the cursor follows directly under the pen tip. For example, a one inch movement of the pen corresponds to a 1 inch mark. Therefore, with pen-based systems, C:D ratio is not an issue. However, with input devices that do not write directly on the display, (i.e., the mouse), C:D ratio is an issue. A one inch movement of the mouse may result in different lengths of marks on different computers if they have different C:D ratios. C:D ratios that vary depending on the speed of the movement (referred to as *cursor acceleration*) complicate this situation even further. A one inch movement made quickly can generate a much longer mark than the same movement made slowly, for example. Therefore, under these conditions, discrimination by length may be unreliable. However, discrimination by angle is not affected by varying C:D ratios. For example, a 45 degree mark is a 45 degree mark whether or not it is one or two inches long. Since it is desirable that our technique be usable with other input devices besides the pen, discrimination by angle is a better choice.

2.5.4. Control methods

Selection from a menu with a pointing device is generally accomplished by *dragging*, by *tapping*, or a combination of the two. We refer to these as the control methods. When dragging is the control method, pressing the pen down on the screen ("pen-down") displays the menu; moving the pen while it pressed against the screen ("dragging") selects different items; lifting the pen from the screen ("pen-up") confirms the selection. When menus are hierarchic, dragging into certain areas may cause submenus to be displayed for selection. When tapping is the control method, a pen-down followed quickly by a pen-up (a "tap") causes the menu to be

displayed; a "tap" over an item confirms its selection. If the menu is hierarchic, the selection will result in another menu being displayed.

Dragging is preferred because selection in menu mode must be a rehearsal of the movement needed to make the mark. Marks are created by dragging the pen across the display surface and therefore dragging is a more accurate rehearsal of marking than tapping.

Marking menus use an action called press-and-wait to allow the user to switch into menu mode. We elected to use this action for several reasons. First, it deviates very slightly from the act of marking (the wait is only 1/3 of second). Thus the principle of rehearsal is not dramatically violated. For example, an action such as holding down a special key or making a special movement to invoke the menu would be a much more dramatic violation of rehearsal. Second, when a user wants to avoid menu mode, it usually means one wants to articulate the command quickly. Press-and-wait is easily avoided by quick articulation and avoiding it also makes selection faster. Third, according to our design goals, we assume that novices are not concerned with fast selection and therefore a slight delay in selection is a minor inconvenience. However, as users become more experienced with the menus and desires faster selection, the delay may also provide incentive to use marks.

There are other reasons why delaying the pop-up of the menu is valuable: it can be distracting; it can obliterate part of the screen; and it takes time. For a novice user these may not be problem since displaying the menu is desirable. For expert users, however, a delayed menu pop-up allows the creation of marks and avoids the negative side effects of the menu's display.

2.5.5. Selection events: preview, confirm and terminate

There are several events that occur when making a selection. Selection from a menu generally involves some sort of feedback indicating which item is about to be selected, for example, an item highlights. We refer to this capability as *selection preview*. Selection also involves an action which indicates to the system that it should actually carry out the selection. We refer to this as *selection confirmation*.

In the non-hierarchic case, selection confirmation results in the termination of the entire selection process. In the hierarchic case, selection confirmation will not necessarily terminate the selection process if the item selected has a sub-menu. We

use the term *selection termination* to indicate the action that ends the entire menu selection process. In non-hierarchic case, selection confirmation and selection termination are combined in the same action.

There are many different types of input events that could be used to signal selection confirmation:

• *Pen-up/pen-down*

• *Item entry:* Item entry means selection confirmation occurs the moment the pen enters an item.

• *Boundary crossing:* Boundary crossing means that selection confirmation occurs when the pen crosses the outside border of a menu item.

• *Dwelling:* Dwelling is the act of keeping the pen pressed and not moving for a fraction of a second. A user can avoid issuing dwelling events by keeping the pen moving. Press-and-wait is an example of a dwelling event. However, we distinguish between these two events because press-and-wait signals the entry into menu mode while dwelling signals selection confirmation.

• *Events distinct from pen movement:* This includes things like a button press or an increase in pressure with a pressure sensing pen.

The type of selection confirmation event used affects other design features:

• *mimicking drawing a mark:* Since selection from a hierarchy of menu items involves a series of selection confirmations and we wish to mimic that act of making a mark, an event for selection confirmation that does not interrupt dragging must be used.

• *reselection:* In some cases, a user may desire to change the previewed selection. For example, a user may accidentally move into the wrong item then want to move to the correct item. We refer to this process as reselection. Most menu systems support reselection.

• *pairing command and parameters:* The command compass allows dragging to continue after the final selection confirmation. Dragging is then used to indicate additional parameters to the menu command just selected.

Figure 2.5 shows which selection confirmation methods support these features. Item entry is not feasible because it does not allow reselection. Boundary crossing, dwelling and events distinct from pen movements support both reselection and pairing. We discount "events distinct from pen movement" because it requires additional input sensors like pen buttons or a pressure sensing pen.

Figure 2.5 indicates that boundary crossing and dwelling are the only applicable choices. Boundary crossing is preferable because a visible boundary (i.e., the edge of a menu) gives precise information as to when selection will occur. This information is not visible if dwelling is used. Furthermore, waiting for a dwelling to occur slows interaction. It is also possible to use pen release as a confirmation method if pairing is not required and the item being selected is the last in a series of selections.

We implemented boundary crossing by having selection confirmation occur when the user crossed over the outer edge of a menu item. Specifically, selection previewing occurred as long as the user stayed within the circle of the menu. Selection confirmation occurred when the user moved outside the circle. We discovered, in practice, that boundary crossing created a problem. As a user moves away from the center of the menu to confirm an item, the item's sub-menu pops up when the outer boundary is crossed. Unless a user moves very slowly, one is still moving when the sub-menu appears. This results in one of the items in the sub-menu being selected immediately. If the user is moving fast, the boundary point for the sub-menu may have already been crossed and this results in an erroneous selection confirmation. Even if the boundary point was not crossed, this overshooting in the sub-menu causes reselection to be the first action to occur each time a sub-menu is popped up. This means that users are not rehearsing the movement of drawing a mark, but are rather making a movement which involves reselection. This approach was therefore unacceptable.

To solve this problem, we used a hybrid approach which combines boundary crossing and dwelling. The approach works as follows. As long as the pointer is within some distance from the center of menu, a dwelling event is ignored. Selection preview and reselection are therefore possible without the threat of an accidental dwelling occurring. Once the boundary is crossed, selection preview and reselection are still possible but, if the user dwells, the selected item is confirmed and its sub-menu appears. This allowed users to use coarser movements to make selections without fear of overshooting and selecting from sub-menus.

Dwelling is also consistent with press-and-wait. In both these activities, keeping the pen pressed against the display and holding it triggers the display of a menu.

A selection can also be confirmed without dwelling by releasing the pen at any point in the hierarchy of a menu. This allows any item in the hierarchy to be selected and also signals selection termination.

2.5.6. Mark ambiguities

The current design presents a dilemma if we consider using marks to make selections from hierarchies of menus. The idea behind using marks for selection is

Selection confirmation event	**allows mimicking marking?**	**allows reselection?**	**allows pairing?**
pen release	no*	yes	no
item entry	yes	no	yes
boundary crossing	yes	yes	yes
dwelling	yes	yes	yes
events distinct from pen movement	yes	yes	yes

(* yes in the non-hierarchic case)

(as long as the pointer is kept moving)

Figure 2.5: Different selection confirmation methods characteristics.

that selection will be fast and fluid. This implies that we do not desire or expect a user to "include" dwellings when making selections using marks. This would be unnatural and slow the marking process.

A problem can occur if dwellings are not included when making marks. Consider a selection from a hierarchy that is two levels deep. Suppose the user makes a straight line mark. Does the mark correspond to a selection from the parent menu or the child menu? Figure 2.6 shows the problem. If dwellings no longer occur we cannot disambiguate the selection. If we base the interpretation on boundary crossing, then the mark is unambiguous. Unfortunately, this makes the size of a mark affect its interpretation (i.e., the marks cannot be scaled).

One solution to this problem is called *no category selections.* It is based on the observation that items which have subitems are generally categories of commands, not commands themselves, and selecting a category is not a meaningful operation. For example, when using linear hierarchic menus on the Macintosh, selecting the "font" category leads to a menu of commands that change the font. Selecting "font" by itself (i.e., releasing the mouse button when "font" is selected) performs no operation. Therefore we assume that there is no need to select a category. Thus, we can consider any straight line to be a selection into a submenu (case (b) in Figure 2.6). Note that this permits selection of certain menu items that are embedded in submenus by drawing a short straight mark. We recommend designers put the most popular item in a category in this position to promote efficiency.

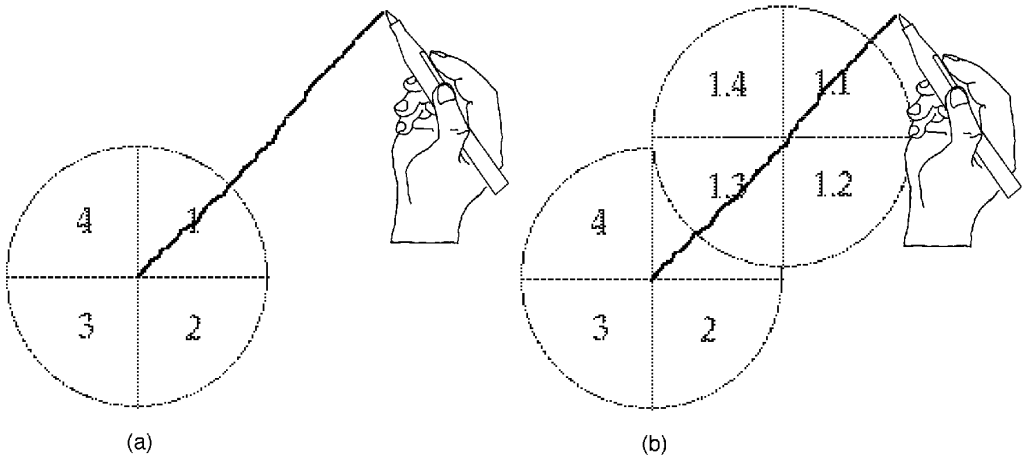


Figure 2.6: Ambiguity in selecting from a hierarchy of menu items two levels deep using a mark. Overlaid grayed menu show possible interpretations. In (a), the interpretation is the selection of item 1. However, (b) is another interpretation according to boundary crossing rules (the selection of item 1.1). Interpretation by boundary crossing is sensitive to the size of marks.

No category selections breaks down when the depth of the hierarchy is greater than two. Suppose a user makes a "^" mark as shown in Figure 2.7 (a). The start of the mark and the change in direction within the mark indicate two points of menu selection. However, what indicates selection from the third level of menu? Figure 2.7 shows this problem. Once again, boundary crossing can be applied to derive an unambiguous set of menu selections but this results in unscalable marks.

There are several solutions to this problem which preserve scaling. The first solution, referred to as the *no-oping* (from the phrase "no operation"), is to simply not permit a series of menu selections that result in a straight line. One way of doing this involves making the item in the child menu that "lines up" with the selection angle of the parent a null operation. This ensures that the beginning of a selection of a non-null item from a child menu is indicated by a change in angle. Unfortunately, this "wastes" a useful sector in a menu.

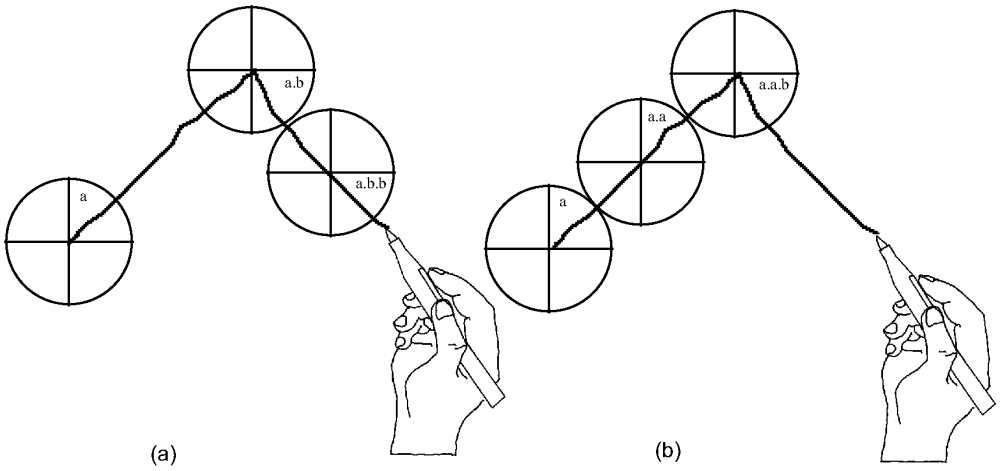


Figure 2.7: Possible interpretations of mark when selecting from hierarchies greater that two levels deep. The straight line sections of the mark have no artifacts to indicate whether the selection at that point is being made from the parent or from the child.

A second solution is *axis-shifting*. This involves rotating child menus such that no item appears at the same angle as an item in the parent menu. Figure 2.8 shows an example of this technique. Axis-shifting involves aligning the boundary between two items in the child menu with the selection angle of the parent item. This ensures that the beginning of a selection from child menu is indicated by a change in angle. Axis-shifting avoids the wasted sectors that occur with no-oping.

This discussion has presented four solutions to hierarchic menu design which are intended to produce an unambiguous vocabulary of marks. The four solutions are: boundary crossing, no category selections, no-oping, and axis-shifting. The aspects of the design that are affected by these solutions are: the ability to select any item within the hierarchy, the ability to have mark interpretation independent of the size of a mark, the ability to select leaf items with a single straight line, and the ability to have all items in a menu active. These aspects may also vary relative to the depth of the menu. Figure 2.9 summarizes this design space.

A solution can be chosen based on the demands of the menu. If menus are only one or two levels deep and menu categories do not need to be selected, then no category selections will work. Boundary crossing and axis-shifting are suitable when hierarchies are more than two levels deep and category menu items need to be

selected. Boundary crossing is also an acceptable solution if category items need to be selected and mark scaling is not an issue.

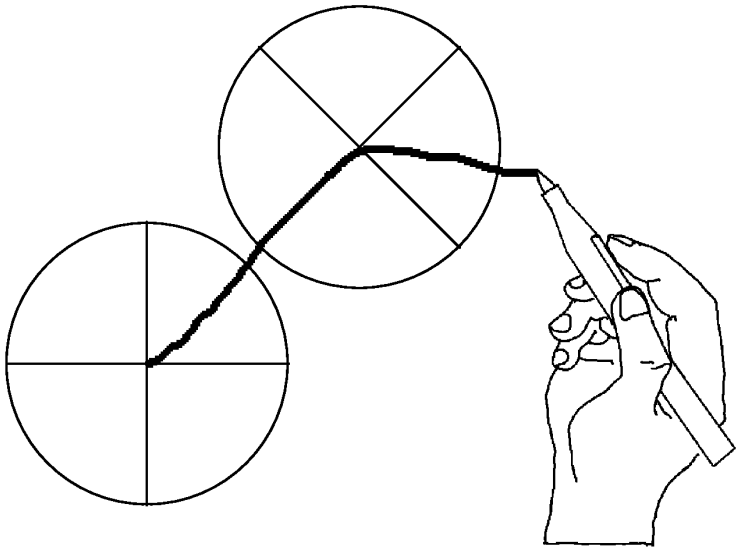


Figure 2.8: Axis shifting rotates a child menu such that child menu items do not appear on the same angle as the parent menu item. This results in a mark language where selection confirmations are indicated by changes in angle. With this scheme marks can be drawn at any size.

Policy	no depth limit?	select any item?	marks scalable?	allows "straight lining"	all items active?
boundary crossing	Yes	Yes	No	Yes	Yes
no-oping	Yes	Yes	Yes	No	No
no category selections	No (2)	No (except in 1 deep case)	Yes	Yes	Yes
axis-shifting	Yes	Yes	Yes	No	Yes

Figure 2.9: Policies that avoid ambiguous interpretation of marking menu marks.

2.5.7. Display methods

There are several design options which concern how menus are displayed:

• *Menu trail* refers to leaving parent menus displayed as a user descends a hierarchy of menu items.

• *Menu overlap* refers to displaying child menus over the top of parent menus.

These methods become important when backing up in a hierarchy of menus.

2.5.8. Backing-up the hierarchy

The ability to back-up in a hierarchy of menus is useful for browsing menu items and correcting mistakes. Backing-up can be one of three types: back-up only to the parent menu, back-up to any ancestor menu, back-up to any ancestor menu item. Backing-up can be accomplished in several ways. Pointing to an item can trigger a back-up to the item, or an explicit action can trigger a back-up (i.e., tapping the pen triggers a back-up to the parent menu). A combination of these two methods can be used (i.e., tapping on an item to back-up to it). Lifting the pen is already used to indicate selection termination, so the back-up technique is restricted to pointing while the pen is being dragged.

Backing-up brings the roles of menu trail and menu overlap into play. Pointing to the item in order to back-up to it requires that item be displayed on the screen. Therefore a menu trail must be provided. However, child menu items may cover up parent items making it impossible to point to "covered" items. The design must ensure that parent items are not covered up.

Design requirements dictate that backing-up in marking menus operates like backing-up in traditional drag-through hierarchical menus: to back-up to a parent menu item, a user points to it; the system then closes the currently displayed child menu and displays the child menu of the parent item. We can adopt this scheme for marking menus but it reduces the advantage of radial menu selection. Figure 2.10 shows the problem that occurs. A selection from a child menu may result in pointing to a parent menu item and this causes an unintended back-up. A prototype implementation of marking menus revealed this to be a real problem. The problem could be avoided if a user is "careful", but this tends to slow users down.

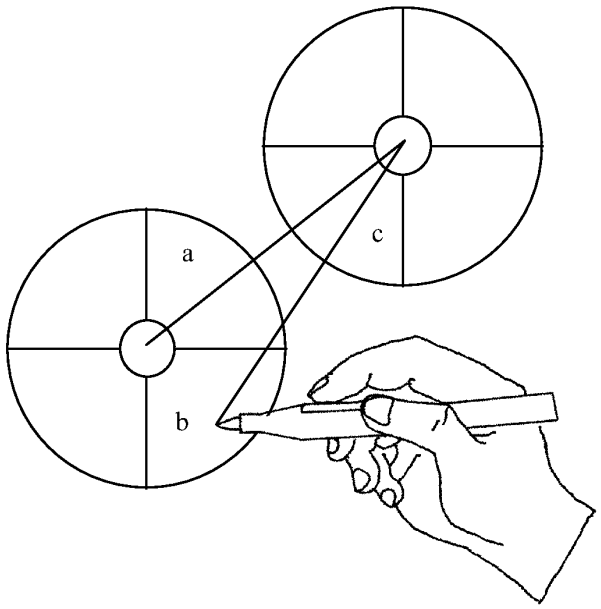


Figure 2.10: A problem with the backing up by pointing to a parent item. Is the user selecting item a.c or backing up to item b?

To solve this problem, we could restrict marking menus to operate like linear menus where selection occurs only if the user is pointing inside a menu item. This has two

major disadvantages. First, it selection sensitive to the length of strokes, and second, it massively reduces item size from a sector of the entire screen to the small sector of the menu.

The solution is to reduce the size of the back-up targets. This is done by restricting the back-up targets to the center hole of the parent menus. This drastically reduces the probability of accidentally pointing to a back-up target. Furthermore, we constrain the user to dwell on a center before back-up takes place. This allows the user to "pass through" centers without backup occurring. Figure 2.11 shows this back-up scheme.

This approach has the restriction of only allowing back-up to parent menus. Backing up to a parent menu and displaying another one of the child menus cannot be combined in the same operation. Some hierarchic linear menus allow this. However, this restriction permits fast and unconstrained selection when moving forward in the hierarchy, while still allowing back-up.

This back-up scheme has several more advantages. First, one can back-up to any parent menu, grandparent menu, etc. Second, menu overlap can occur just as long as menu centers do not get covered. Finally, because backing-up actually returns the cursor to parent menus, rather than redisplaying parent menu at the cursor location, this reduces the chances of menus "walking off" the screen (this problem is further discussed in Section 6.2.3).

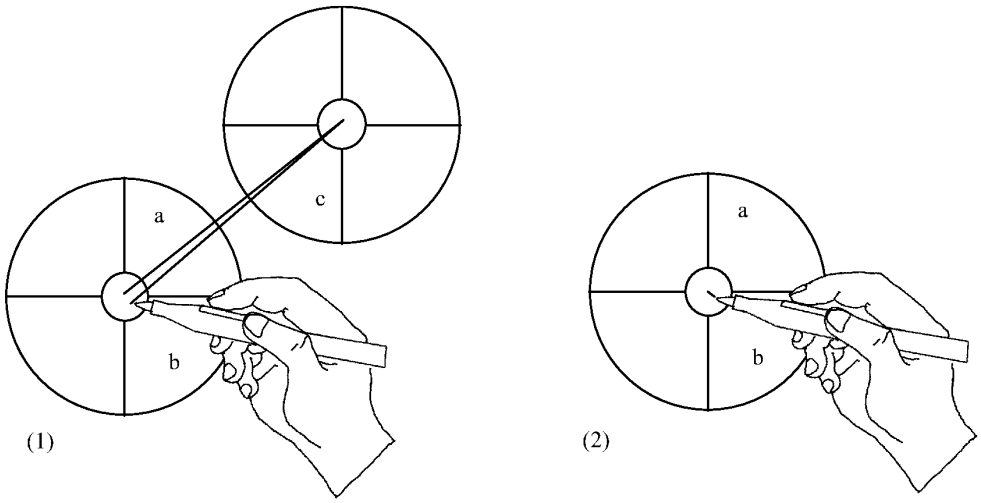


Figure 2.11: Backing-up in hierarchic marking menus. In (1) the user moves into the center of a parent menu and dwells momentarily. In (2) the system senses the dwelling and backs-up to the parent menu by removing the child of item a. Selection may then continue from the parent.

2.5.9. Aborting selection

Most menu systems have a way of specifying a null selection. Generally this is accomplished by selecting outside a menu item. As explained previously, marking menus allow selection to occur outside the item to make selection easier. To circumvent this problem, the center hole of a menu is used to indicate no selection. Lifting the pen within the center hole results in the menu selection being aborted.

A mark may be also be aborted. This involves either lifting the pen before the mark is complete or turning the mark into an uninterpretable scrawl while drawing it.

2.5.10. Graphic designs and layout

During everyday use of marking menus we observed some problems with a "pie" graphical representation. First, as the number of items in the menu increases and the length of labels increases, the size of the pie grows rapidly. This creates several problems. First, having large areas of the screen display and undisplay is visually annoying. Second, a large menu occludes too much of the screen. In many situations, a menu associated with a graphical object must be popped up over the

object. The problem is that displaying the menu completely hides the object. This results in the context of the selection being lost. Third, large menus take time to display and undisplay. In most systems, the image "underneath " a menu is saved before a menu is displayed, and restored when a menu is undisplayed. When a menu is very large, these operations take considerable amounts of time because large sections of memory are being copied to and from the display. Also, algorithms for sizing and laying out labels within the pie of the menu can be quite complex. This makes the implementation of menu layout procedures complex. Complex computations may also delay the display of menus.

To solve these problems we designed an alternate graphic layout for marking menus called "label"[10]. Figure 2.12 shows an example. This alternate design has several advantages over a pie representation. First, it reduces the amount of screen that changes when a marking menu is displayed and undisplayed, and therefore, it reduces visual annoyance. Second, it occludes less of the screen than a pie representation because only the menu center and labels are opaque. Thus more of the context underneath a menu can be seen. This design also reduces the amount of memory that must be copied to and from the display, and hence it reduces the amount of time needed to display a menu.

Another issue of graphical layout is the problem of displaying menus near an edge or corner of the screen. Pie menu systems deal with this issue by using a technique called "cursor warping". Unfortunately, cursor warping is not suitable for pen-based systems. In Chapter 6, we further discuss this issue and describe an alternative to cursor warping.

Although not shown in Figure 2.12, marking menus have many standard features found in traditional menus. For example, marking menus allow grayed-out and checked items. Also, if an item has a submenu, a small circle appears to the right of the label. The intention is that this circle represents the center hole of the submenu. We also found it valuable to hide the labels of parent menus, thus reducing screen clutter. The only portion of a parent menu that is displayed is the center hole (so a user can point to it to back-up). We have also experimented with transparent menus

---

[10] We acknowledge Mark Tapia for his assistance in designing and implementing the alternate graphical layout for marking menus

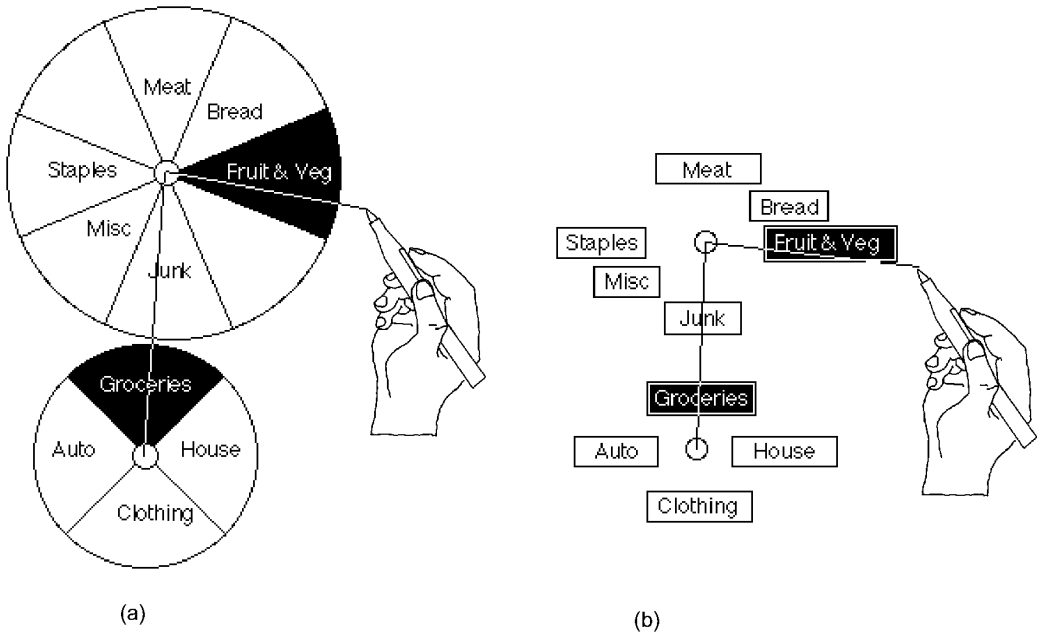


Figure 2.12: An alternate graphic representation for a radial menu "label". Rather than displaying "pie" shapes (a), only the labels and center are displayed (b). The menu then occludes less of display and can be displayed faster.

and graying out parent menus but a full discussion of these experiments is beyond the scope of this dissertation.

2.5.11. Summary of design

The previous sections described and discussed various design features and options of marking menus. We now summarize the features and indicate which design options we elected to use.

Marking menus use discrimination by angle. Selection previewing in menu mode is supported by dragging the pen into an item, and the item being highlighted. Selection confirmation is indicated by a combination of boundary crossing and dwelling. Selection termination is indicated by pen up.

To avoid mark ambiguities, we recommend three possible strategies: no-oping, no category selections and axis-shifting. If menus require only a few items, no-oping may be a suitable solution. If menus are only two levels deep and category selection is not required, no category selection is a suitable solution. If menus require many

menu items, and are more than two levels deep, axis-shifting must be used. In practice, we used no category selection in many situations.

Making a selection in menu mode leaves a menu trail but only the center of parent menu is displayed. We found in practice this reduces the visual clutter the would be caused by the display of inactive parent menu items. Menus are allowed to overlap, but because only the center of parent menu is displayed, this generally does not cause visual confusion.

In menu mode, selection can be aborted by terminating the selection while pointing to the center hole of a menu. In mark mode, selection can be aborted by turning the mark into a "scribble".

If a user dwells while drawing a mark, the system indicates the menu items that would be selected by the mark by displaying the menus "along" the mark. The system then goes into menu mode. This process, called mark confirmation, can be used to verify the items that are about to be selected by a mark or a portion of a mark.

Marking menus can be displayed in either a "pie" representation or a "label" representation. A "label" representation is suitable when there is a need to minimize the amount of screen occluded by the display of the menu.

2.6. SUMMARY

The success of an interaction technique depends not only on its acceptance by users but also on its acceptance by interface designers and implementors. An "industrial strength" interaction technique must not only be effective for a user, but also have the ability to co-exist with other interaction techniques, other paradigms, and differing features of the software and hardware. Because of these demands, as in many other interaction techniques, our motivation and design behind marking menus is complex. What appears on the surface as a simple interaction technique is actually based on many different motivations and has many design subtleties and details.

In this chapter we defined marking menus and described the various motivations for developing and evaluating them. These included providing marks for functions which have no intuitive mark, supporting unfolding interface paradigms,

simplifying mark recognition, maintaining compatibility with existing interfaces, and supporting both novice and expert users. We are also motivated to study marking menus as a way to evaluate the design principles they are based on.

We then outlined the issues involved in evaluating marking menus and proposed an initial design. The major parameters to be evaluated concern the question of how much functionality can be loaded on a marking menu. Essentially our research focus is on establishing the limitations of marking menus so interface designers who are utilizing marking menus can design accordingly. The remaining chapters explore the limitations and characteristics of the design.

Chapter 3: An empirical evaluation of non-hierarchic marking menus

This chapter addresses basic questions about marking menu design variables: how many items can marking menus contain; what kinds of input devices can be used in conjunction with marking menus; how quickly can users learn the associations between items and marks; how much is performance degraded by not using the menu; and whether there is any advantage in using an ink-trail. This chapter describes an experiment which addresses these questions. The approach is to pose specific hypotheses about the relationship between important design variables and performance, and then to test these hypotheses in the context of a controlled experiment. The results of the experiment are then interpreted to provide answers to the basic questions posed above.

In this experiment we limit our investigation to non-hierarchic marking menus. We do this for several reasons. First, this experiment serves as a feasibility test of non-hierarchic marking menus. If non-hierarchic marking menus prove feasible, then an investigation of hierarchic marking menus is warranted. Second, we feel that the characteristics of non-hierarchic marking menus must be understood before we can begin to investigate hierarchic marking menus. Our findings on non-hierarchic marking menus can then be used to refine our design and evaluation of hierarchic marking menus. Third, this experiment addresses many factors. To include the additional factor of hierarchic structuring would make the experiment too large and impractical.

To date there is little research applicable to our investigation. Callahan, Hopkins, Weiser, and Shneiderman (1988) investigated target seek time and error rates for 8-item pie menus, but concentrated on comparing them to linear menus. In particular

they were interested in what kind of information is best represented in pie menu format. Section 2.3.1 described their results.

Our experiment focuses on selecting from marking menus using marks. To address the questions posed at the start of this chapter, the experiment examines the effect that the number of items in a menu, choice of input device, amount of practice, and presence or absence of an ink-trail or menu, has on response time and error rate.

3.1. THE EXPERIMENT

3.1.1. Design

In this experiment, we varied the number of items per menu and input device for three groups of subjects and asked them to select target items as quickly as possible from a series of simple pie menus. One group selected target items from fully visible or "exposed" menus (Exposed group). Since there is little cognitive load involved in finding the target item from menus which are always present, we felt that this group would reveal differences in articulation performance due to input device and number of items in a menu.

Two other groups selected items from menus which were not visible ("hidden" menus). In one group, the cursor left an ink-trail during selection (Marking group), and in the other, it did not (Hidden group). The two hidden menu groups were intended to uncover cognitive aspects of performance. Hiding the menus would require the added cognitive load of either remembering the location of the target item by remembering or mentally constructing the menu, or by remembering the association between marks and the commands they invoke through repeated practice. Comparing use of an ink-trail with no ink-trail was intended to reveal the extent to which supporting the metaphor of marking and providing additional visual feedback affects performance. The Exposed group provided a baseline to measure the amount that performance degraded when selecting from hidden menus.

3.1.2. Hypotheses

We formed the following specific hypotheses to address the questions posed at the start of this chapter:

How much is performance degraded by not using the menu?

Hypothesis 1. Exposed menus will yield faster response times and lower error rates than the two hidden menu groups. However, performance for the two hidden groups will be similar to the Exposed group when the number of items per menu is small. When the number of items is large, there will be greater differences in performance for hidden versus exposed menus. This prediction is based on the assumption that the association between marks and items is acquired quickly when there are very few items. As the number of menu items increases, the association between marks and items takes longer to acquire, and mentally reconstructing menus in order to infer the correct mark becomes more difficult.

How many items can marking menus contain?

Hypothesis 2. For exposed menus, response time and number of errors will monotonically increase as the number of items per menu increases. This is because we assume that performance on exposed menus is mainly limited by the ease of articulation of menu selection, as opposed to ease of remembering or inferring the menu layout. We know that performance time and errors monotonically increase as target size decreases, all else being equal (Fitts, 1954).

Hypothesis 3. For hidden menus (Marking and Hidden groups), response time will not solely be a function of number of items per menu. Instead, menu layouts that are easily inferred or that are familiar will tend to facilitate the cognitive processes involved. We predict that menus containing eight items can be more easily mentally represented than those containing seven items, for example. Similarly, a menu containing twelve items is familiar since it is similar to a clock face, and thus we predict it is more easily mentally represented than a menu containing eleven items.

What kinds of input devices can be used in conjunction with marking menus?

Hypothesis 4. The stylus will outperform the mouse both in terms of response time and errors. The mouse will outperform the trackball. This prediction is based on previous work (Mackenzie, Sellen, & Buxton, 1991) comparing these devices in a Fitts' law task (i.e., a task involving fast, repeated movement between two targets in one dimension).

Hypothesis 5. Device differences will not interact with hidden or exposed menus, or the presence or absence of marks. Differences in performance due to device will

not depend on whether the menus are hidden or exposed, or whether or not marks are used. The rationale for this is that we assume performance differences stemming from different devices are mostly a function of articulation rather than cognition. We also assume that the articulatory requirements of the task are relatively constant across groups.

Is there any advantage in using an ink-trail?

Hypothesis 6. Users will make straighter strokes in the Marking group. We based this prediction on the assumption that visual feedback is provided in the Marking group and also that hidden menus support the "marking" metaphor as opposed to the "menu selection" metaphor.

How quickly can users learn the associations between items and marks?

Hypothesis 7. Performance on hidden menus (Marking and Hidden groups) will improve steadily across trials. Performance with exposed menus will remain fairly constant across trials. This prediction is based on belief that articulation of selection (or simply executing the response) will not dramatically increase with practice since it is a very simple action. Performance on hidden menus, however, involves the additional cognitive process of recalling the location of menu items. We believe this process will be subject to more dramatic learning effects over time.

3.1.3. Method

Subjects. Thirty-six right-handed subjects were randomly assigned to one of three groups (Exposed, Hidden, and Marking groups). All but one had considerable experience using a mouse. Only one subject had experience using a trackball. None of the subjects had experience with a stylus.

Equipment. The task was performed on a Macintosh IIX computer. The standard Macintosh mouse was used and set to the smallest C:D ratio. The trackball used was a Kensington *TurboMouse*, also set to the smallest C:D ratio. The stylus was a Wacom tablet and pressure-sensitive stylus (an absolute device). The C:D ratio used was approximately one-to-one.

Task. Subjects used each of three input devices to select target "slices" from a series of pie menus as quickly and as accurately as possible. The pies contained either 4, 5, 7, 8, 11, or 12 slices. All pie menus contained numbered segments, always beginning

with a "1" immediately adjacent and to the right of the top segment. The other slices were labeled in clockwise order with the maximum number at the top (see Figure 3.1 (a)). The diameter of all pie menus was 6.5 cm., and Geneva 14 point bold font was used to label the slices.

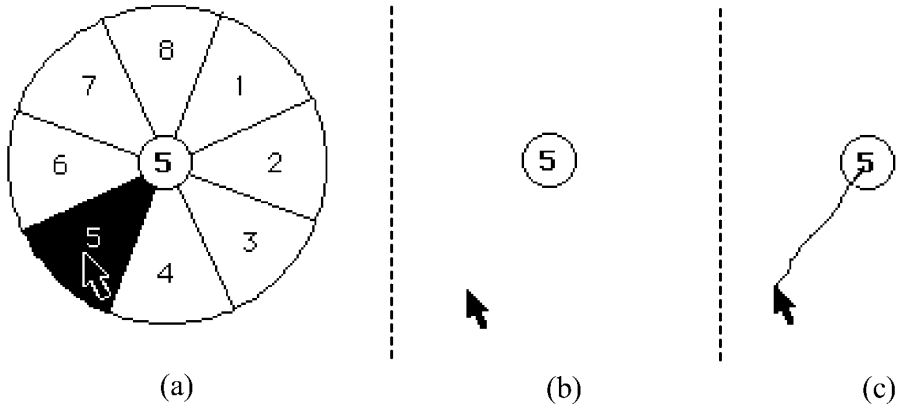


Figure 3.1: Selecting item 5 from an eight-item pie menu (a) in the Exposed group, (b) in the Hidden group, and (c) in the Marking group.

In designing this experiment, a great deal of time was spent discussing what kind of items should be displayed in the pie menus. Menus in real computer applications usually contain meaningful items, but the order in which they appear is not easily inferred. The numbered menus we used, on the other hand, used ordered, meaningless labels. We wanted to approximate the case of an expert user who is familiar with the menu layout. We decided to reduce as much as possible the learning time associated with memorizing the items. Our focus was on the articulation of actions, and the cognitive processes involved in mentally representing or mentally constructing menu layout. Since Callahan et al. (1988) have shown that performance varies depending on the kinds of items represented, using the same kind of items for all menus (numbered items) was an attempt to eliminate this effect. Thus our comparisons between menus with different numbers of items would be more accurate. We acknowledge that both the choice of menu items and their mapping within a menu may have a significant effect on performance. These factors are outside the scope of this investigation.

In the Exposed menu group, the entire menu was presented on each trial (Figure 3.1 (a)). The target number corresponding to the slice to be selected was presented when the subject located the cursor within the center circle of the pie menu and either pressed down and held the mouse or trackball button, or pressed down and maintained pressure on the stylus. The subject's task was then to maintain pressure and move in the direction of the target slice. Menu slices would highlight as the cursor moved over them, indicating to the subject a potential selection. A slice would remain highlighted even if the cursor went outside the outer perimeter of the pie. Releasing the button, or pressure, signaled to the system that the highlighted slice was selected. After the selection was made, the menu would "gray out" displaying the menu with the slice selected for a period of 1 second. If an incorrect slice was selected, the Macintosh would beep on release. This marked the end of a trial.

In the Hidden menu group, the task was essentially the same, except that during selection, only the central circle of the pie menu would be visible (Figure 3.1 (b)). After confirming the selection, subjects would receive the same grayed-out feedback as in the Exposed group, indicating which response had been made, and whether or not it had been correct. The Marking group was almost identical to the Hidden group, except that the movement of the cursor with the button depressed left an ink-trail (Figure 3.1 (c)) .

After each trial, subjects received a running score, presented in the lower right-hand corner of the screen. A minimum of 10 points could be obtained for each correct response, with more points scored as response time became shorter. However, subjects were penalized 20 points for errors.[11] At the end of each block of trials, each subject's current performance was shown in relation to the best score obtained by other subjects in the same conditions. The scoring criterion was the same for all groups.

Design and Procedure. One third (twelve) of the subjects were randomly assigned to the Exposed group, one third to the Hidden group, and one third to the Marking

---

[11] This scoring scheme was arrived at by experimenting with different scoring schemes on pilot subjects. We found that the choosen scheme emphasized both accuracy and speed. On average, subject scores were positive and they found this encouraging and fair.

group. Every subject used each of the three input devices (mouse, trackball and stylus). Trials were blocked by device and order of device was counterbalanced.

For each device, *all groups* began by practicing on exposed menus for a total of six trials for each of six different menus, containing either 4, 5, 7, 8, 11 or 12 items. During practice, number of items per menu was blocked and presented in random order. This practice period was intended to acquaint subjects with the feel of the particular input device they were about to use. It also provided an opportunity for subjects to familiarize themselves with the layout of the menus before beginning the timed trials.

Subjects in the Exposed group then moved on to the timed trials, while subjects in the Hidden and Marking groups received a further set of practice trials designed to acquaint them with the "feel" of hidden menus. For this practice session, menus containing both three and six items were used (six trials each) since 3-item or 6-item menus were never used in the actual timed trials. This was a deliberate attempt to equalize exposure to the menus of interest in the three groups.

For the timed portion of the experiment, trials were again blocked by number of items (4, 5, 7, 8, 11, or 12). The order in which the number of items appeared was randomly permuted for each subject. Each subject began a particular block by first studying the menu layout for 6 seconds. They then received a total of 40 trials for each different menu with a short break at intervals of ten trials. Targets were drawn randomly from a uniform distribution with replacement, with the added constraint that no target could be repeated on consecutive trials.

In summary, each subject performed 40 trials on each of the six menus (menus consisting of 4, 5, 7, 8, 11, and 12 items) and using all three devices, resulting in a total of 720 scores per subject. Each group consisted of twelve subjects which resulted in 8640 scores per group. The three different groups provided a total of 25920 scores for the experiment.

3.2. RESULTS AND DISCUSSION

The main dependent variables of interest were response time and number of errors. Response time was defined as the total time from presentation of the target number

to confirmation of the selection for error-free trials. An error was defined as an incorrect selection. The means for each group are shown in Figure 3.2.

3.2.1. Effects due to number of items per menu

As expected, increasing the number of items per menu significantly increased both response time ($F(5,55) = 388.4$, $p < .001$) and errors ($F(5,55) = 382.8$, $p < .001$).[12] There were overall performance differences among the groups in terms of errors ($F(2,22) = 21.97$, $p < .001$) but not in terms of response time. However, these main effects are not particularly meaningful because differences among groups depend on the number of items per menu (see Figure 3.3). That is, there was a significant interaction between group and number of items per menu both in terms of response time ($F(10,110) = 3.5$, $p < .001$) and errors ($F(10,110) = 64.7$, $p < .001$).

These results address the first three hypotheses:

(1) As predicted by Hypothesis 1, mean response time was consistently lower in the Exposed group versus the Hidden and Marking groups as the number of items increased. This is supported by the significant interaction between group and number of items per menu (reported above), and by specific comparison tests. No difference was found between the two hidden groups and the Exposed group for menus containing four items. However, for menus containing five items, response times were significantly slower for hidden menus compared to Exposed ($F(1,110) = 6.5$, $p < .001$). The two hidden groups were no different from each other in terms of errors (post hoc comparison of error means, Tukey HSD, $\alpha = .05$), but both produced significantly more errors than the Exposed menu group.

(2) Our second hypothesis predicted that in the Exposed group, response time and errors would monotonically increase as a function of number of items per menu. In the case of errors, this relationship seems to hold. However, this must be qualified by the fact that errors were infrequent and thus floor effects may obscure the true shape of the function.

---

[12] Throughout this disseration we use the F-statistic to evaluate the equality of population means. See Appendix A for an explanation.

Group	Mean RT in sec. (SD)	Mean Number of Errors in 40 Trials (SD)	Mean Percentage Errors
Exposed	*0.98 (0.23)*	*0.64 (1.00)*	*1.6%*
Hidden	*1.10 (0.31)*	*3.27 (3.57)*	*8.2%*
Marking	*1.10 (0.31)*	*3.76 (3.67)*	*9.4%*

Figure 3.2: Mean response time and number of errors for each experimental group.

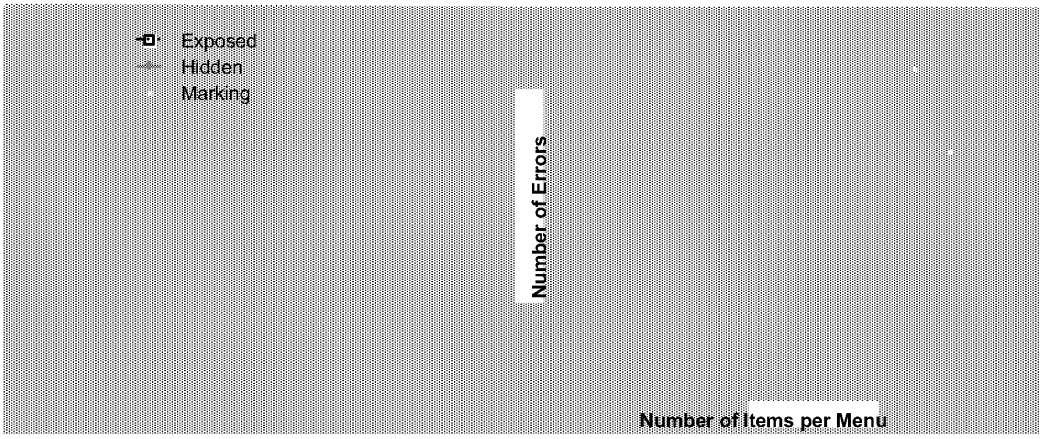


Figure 3.3: Response time and average number of errors (of a total of 40 trials) as a function of number of items per menu and group.

Response time also increased monotonically except for menus containing twelve items. Specific comparisons at the .05 level confirm significant increases in response time from four to five items per menu ($F(1,55) = 16.8$, $p < .001$), and from seven to eight items per menu ($F(1,55) = 7.4$, $p < .01$), but no differences between eleven and twelve items per menu. One possibility is that familiarity with the "clock face" layout may have reduced the time for visual search, thereby reducing overall response time. Another possibility is that this could be a case of diminishing effects. Adding an extra item to a menu containing four items represents a 20% increase in

the number of items, whereas, adding an extra menu item to one which contains eleven represents only an 8% increase in number of items.

(3) The pattern of results predicted by Hypothesis 3 is also supported: when menus were hidden, some kinds of menus were easier to evoke or reconstruct from memory than others. This was not purely a function of number of items per menu. The characteristic curve that emerges (Figure 3.3) shows that performance in general does tend to degrade as the number of items per menu increases, but that certain numbers of items do not follow this pattern (i.e., eight and twelve items).

This hypothesis is also confirmed by a series of specific comparisons showing no differences in either hidden menu group for seven versus eight items per menu. Further, performance on menus of twelve items was faster than on menus of eleven items for the Hidden group ($F(1,55) = 11.25$, $p < .001$) and was more accurate than on menus of eleven items in both groups (Hidden, $F(1,55) = 50.96$, $p < .001$; Marking $F(1,55) = 13.51$, $p < .001$). By contrast, for both groups, tests show menus of four items yielded faster response times than menus of five items (Hidden, $F(1,55) = 4.05$, $p < .05$; Marking $F(1,55) = 9.00$, $p < .05$).

The results show that menus containing twelve items in particular may have facilitated performance. Many subjects mentioned that the metaphor of a clock face helped them to select the target item because it could be brought readily to mind. Thus it seems reasonable to suggest that it is the cognitive bottleneck, or the difficulty of evoking the mapping between target and action, that limits performance.

3.2.2. Device effects

As predicted by Hypothesis 4, subjects performed better with a stylus and a mouse than they did with a trackball. Response time ($F(2,22) = 9.64$, $p < .001$) and errors ($F(2,22) = 11.29$, $p < .001$) were both affected by the type of input device subjects used. Pairwise comparisons (Tukey HSD test, $\alpha = .05$) showed the trackball was both significantly slower and gave rise to more errors than the stylus or mouse. However, contrary to our expectations, there was no difference in mean response time or errors between the stylus and mouse.

Initial analyses supported Hypothesis 5 where we predicted that the effect of input device would not depend on whether or not the menus were exposed, or whether or

not there was an ink-trail. Input device did not interact with group, either in terms of response time or errors.[13] However, on closer examination, a more interesting result emerged.

We discovered that in the Marking group, the stylus was significantly faster than both the trackball and mouse with no difference between the trackball and mouse (Figure 3.4). In the Exposed group, the mouse and stylus were faster than the trackball, with no difference between the mouse and stylus. These discoveries were based on separate analyses of variance for each of the three groups on the response time data. There were significant differences among devices in the Exposed ($F(2,22) = 10.44$, $p < .001$) and Marking groups ($F(2,22) = 8.32$, $p < .002$), but not in the Hidden group. Tukey tests revealed the superiority of the stylus in the Marking group and the inferiority of the trackball in the Exposed group. No significant interactions between device and number of items were found in any of the three groups. Given these results we cautiously reject Hypothesis 5.

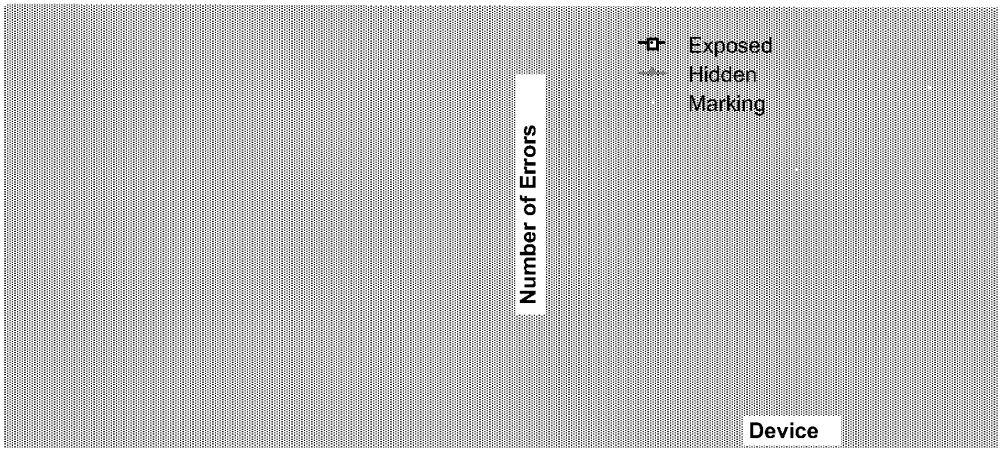


Figure 3.4: Response time and average number of errors (of a total of 40 trials) as a function of device and group.

There may be two reasons for the superiority of the stylus when marks are added to selection from hidden menus. First, it is often difficult to perceive when enough

[13] There were also no significant interactions between number of items per menu and device, nor significant three-way interactions (group by number of items by device).

pressure is being applied to the stylus to make a selection. Thus, providing visual feedback when this state is maintained may be important to realize the full potential of this device. Second, providing an ink-trail is consistent with the metaphor of marking with a pen, which may improve performance. Alternatively, failing to support the pen metaphor by not providing the ink trial (Hidden group) may violate users' expectations and thus negatively affect performance.

Separate analyses of the error data within each group further supported the inferiority of the trackball. The trackball was found to be the source of significant device differences in the Exposed ($F(2,22) = 9.92$, $p < .001$)[14] and Marking groups ($F(2,22) = 9.92$, $p < .001$). Pairwise comparisons in the Exposed and Marking groups showed differences between the trackball and the other two devices, and no difference between mouse and stylus.

The finding that the trackball was no more slower or error prone than the mouse and stylus in the Hidden group may be due to the fact that in both the Exposed and Marking groups, visual feedback emphasized the difficulty of articulating the actions of the trackball thereby causing performance to be worse. In the Exposed case, sectors were highlighted as they were selected and it is possible that the trackball caused a great deal of reselection. In the Marking case, users reported that the ink-trail was disturbing in conjunction with the trackball because the paths looked erratic and inaccurate.

3.2.3. Mark analysis

We were interested in seeing if subjects used straight marks when making selections. This was important to discover because, if menu selection tended to be done in some manner other than a straight mark, we could not claim that users rehearse this physical movement when selecting from menus. Thus we would not expect as much transfer of skill between making menu selections and making marks. Another reason we were interested in seeing if subjects used straight marks was related to using marking menus in applications that recognize other marks beside those used in menu selection. Unlike conventional menu selection which is based

---

[14] Both a significant device by menu size interaction ($F(10,110) = 2.47$, $p < .011$) and floor effects should make us cautious in interpreting the main effect of device in the Exposed group. However, the fact that the trackball produces consistently more errors on average across menu size, supports the claim that the trackball is outperformed by stylus and mouse.

only on the last location of the cursor, mark recognition systems take the entire shape of the stroke into account. For example, suppose the system also recognizes the symbol "C". A very crooked mark intended to make a selection from a hidden menu might be interpreted as an "C". The success of recognition depends to some extent on knowing the shapes of the strokes that users tend to create. To address these issues we recorded and displayed the path data for users' individual marks. Figure 3.5 shows a typical example.

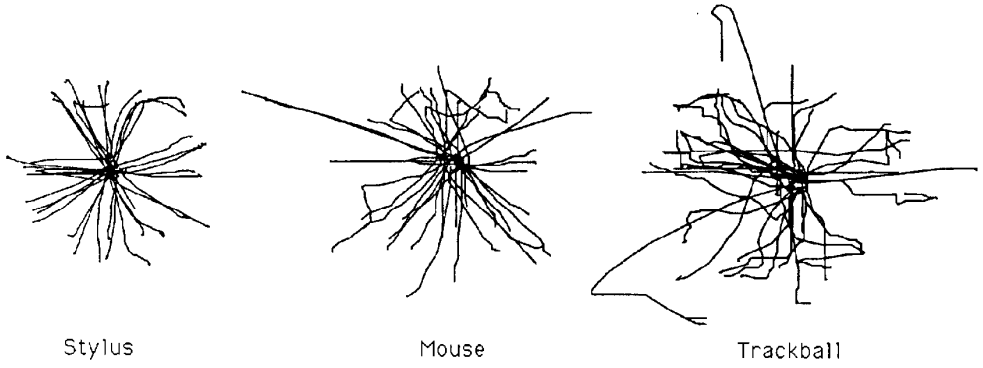


Figure 3.5: The marks a subject used in selecting from a hidden twelve-item menu.

Subjects made approximately straight marks. No alternate strategies such as starting at the top item and then moving to the correct item were observed. However, there was evidence of reselection from time to time, where subjects would begin a straight mark and then change direction in order to select something different.

Surprisingly, we observed reselection even in the Hidden and Marking groups. This was especially unexpected in the Marking group since we felt the idea of drawing a mark does not naturally suggest the possibility of reselection. Hence, we reject Hypothesis 6. It was clear though, that training the subjects in the hidden groups on exposed menus first made this option apparent. Clearly many of the subjects in the Marking group were not thinking of the task as making marks *per se*, but of making selections from menus that they had to imagine. This brings into question our *a priori* assumption that the Marking group was using a marking metaphor, while the Hidden group was using a menu selection metaphor. It may explain why very few behavioral differences were found between the two groups.

Reselection in the hidden groups most likely occurred when subjects began a selection in error but detected and corrected the error before confirming the selection. This was even observed in the "easy" four-slice menu, which supports the assumption that many of these reselections are due to detected mental slips as opposed to problems in articulation. There was also evidence of "fine tuning" in the hidden cases, where subjects first moved directly to an approximate area of the screen, and then appeared to adjust between two adjacent sectors.

Strokes produced with the trackball appeared more jagged and less controlled than those made with the mouse or stylus. This is consistent with the statistical results showing that the trackball tends to be slower and less accurate than the stylus or mouse. For four-item menus, most subjects made straighter marks with the stylus than the mouse. The presence or absence of an ink-trail did not appear to make any discernible difference to stroke shape.

3.2.4. Learning effects

The forty trials for each different menu were divided into eight consecutive blocks. Response time and mean errors were calculated for each five-trial block in order to look more closely at learning effects. Overall, there was a small but steady decrease in response time over trials which was statistically significant ($F(7,77) = 5.79$, $p < .001$). Error rate also showed signs of improving with number of trials ($F(7,77) = 10.52$, $p < .001$).

We have claimed that the major factor limiting performance on exposed menus is the physical accuracy required for the action of selection. The results support this claim. In the case of hidden menus, results support the claim that the factor limiting performance is cognitive. In other words, the time it takes to remember or infer the correct mental representation becomes the overriding factor determining performance. Thus, performance in the Exposed group can serve as a baseline measure that users should approach as they become expert.

Hypothesis 7 states that the cognitive component is the component most affected by learning, as opposed to the articulatory component. Thus, we expect a steady improvement in performance in the two hidden groups, as opposed to fairly constant performance in the Exposed group over time.

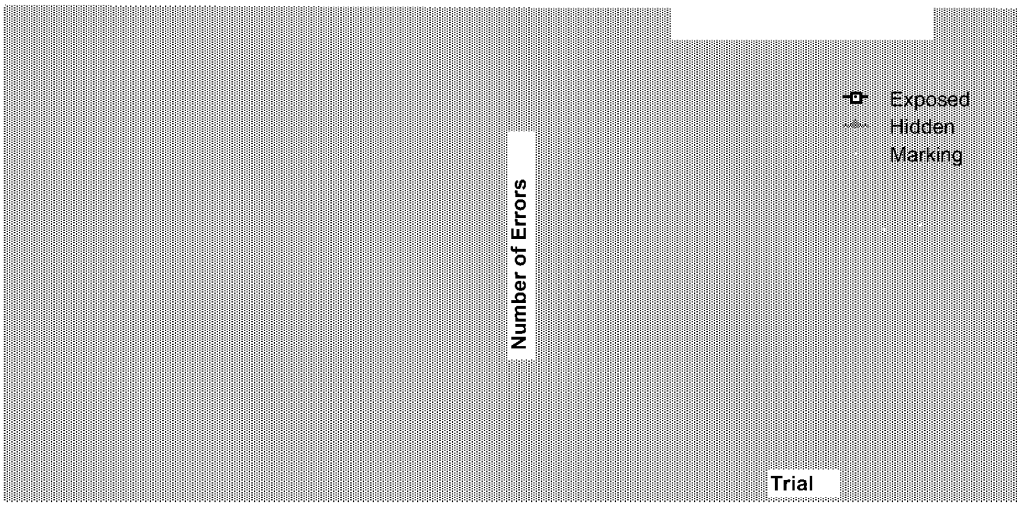


Figure 3.6: Group effects in terms of response time and number of errors in five trial intervals.

As is shown in Figure 3.6, response time in the hidden groups appears to improve across trials while the curve for the Exposed group is fairly flat. Errors also remain relatively constant for the Exposed group over trials, while decreasing on average for the two hidden groups. Support for Hypothesis 7 is found in a significant group by trial interaction for response time ($F(14,154) = 2.90$, $p < .001$) and errors ($F(14,154) = 3.15$, $p < .001$).

As a final point, it follows from the above reasoning that we would expect no significant interaction of input device by trial, since type of input device would presumably have the greatest impact on the articulation as opposed to the cognitive component of performance. The fact that no significant interaction of device by trial was found is consistent with expectation.

3.3. CONCLUSIONS

Relative to our seven hypotheses, the results and their implications for design can be summarized as follows:

Hypothesis 1. As predicted, when menus have many items, hiding menus from users both slows their performance and increases their error rate. As number of items per menu increases, added to the problems of articulation is the difficulty of successfully mentally reconstructing the menu layout or remembering the necessary

strokes to make menu selections. However, when the number of items is small (only four), there is little or no performance difference, even early in practice.

Design Implications. For ordered sets of commands, users should be as fast and error-free in making marks as in selecting from a visible pie menu of up to four slices. If the commands are not ordered, then it may take more time to acquire the skill. However, command semantics can be exploited. For example, "Open" and "Close" can be positioned opposite to each other, as can "Cut" and Paste". This may speed the learning process and allow users to mark ahead faster. In addition, the most frequently used commands form a very small set, and thus we can be optimistic that these can be invoked successfully with marking menus.

Hypothesis 2 and 3. For exposed menus, the results showed performance declines steadily as the number of items increases. This is probably due to two factors: (1) the increasing reaction time to visually search and choose among alternatives, and (2), the increasing difficulty of articulating the action as targets become smaller.

These results agree with other results concerning the effect of the number of items on performance. Perlman conducted an experiment in which subjects made selections from exposed linear menus (Perlman, 1984). Menus containing 5, 10, 15 and 20 items were used. The menus contained ordered numbers from 1 to 20. Beside each item was a randomly chosen left or right arrow character. The task was to find a target item in the menu and indicate it by pressing the corresponding left or right arrow-key. It was found that the number of items in a menu had a linear effect on the time it takes to find an item. These results agree with our results for exposed menus.

Performance on hidden menus in this experiment was different, however. Instead of a result showing monotonically increasing response times and error rates as a function of number of items, even numbers of items (four, eight, or twelve) appeared to facilitate performance. Not surprisingly perhaps, four-item menus yielded significantly faster and more accurate performance than five-item menus. However, performance on eight-item menus was no worse than performance on menus with one less item. Subjects also reported that the eight-item menu was easy to learn because they could easily mentally subdivide the pie and infer the position of the target slice. Most dramatic was the finding that a twelve-item menu actually yielded faster and more accurate performance than a menu containing only eleven

items. We speculate that this difference may be enhanced by familiarity with circles subdivided into twelve sectors, such as in clock faces.

Design Implications. When menus are hidden, overcoming the difficulty of learning and using mental representations of menus can be facilitated by using layouts which exploit known metaphors, or which are easily subdivided. Using an even number of items or laying out items at the points of a compass or hour positions of clock can be used to counteract the increased difficulty of having many items in a menu. The ease with which subjects learned and performed with the twelve-item menu is testimony to the strength of a good metaphor. One could imagine a user remembering a command location or mark by mapping it to an hour/hand position: "undo is at three o'clock".

Hypothesis 4 and 5. The stylus and mouse outperformed the trackball both in terms of response time and errors. Analysis of the paths showed that paths made with the trackball were more jagged and less controlled than those made with the mouse or stylus. The stylus and mouse yielded similar performance, with the exception that the stylus was significantly faster than the mouse when an ink-trail was present.

Design Implications. The results speak strongly against using a trackball for marking menus. Further, subjects' comments suggest that the combination of trackball and ink trial was especially bad. One subject complained of being disturbed by the messy ink-trail left when using a trackball. It seems that the visual feedback provided by the ink-trail only served to emphasize the inadequacy of the paths made by this device.

The performance similarity of the mouse and stylus suggests that either may be appropriate devices for this kind of mechanism. Two cautionary notes should be made, however. First, it is likely that the ink-trail added important feedback to tell the user when the appropriate amount of pressure was being applied to the stylus. This suggests that another kind of stylus (i.e. one with audio or tactile feedback to indicate a "button-click") might have fared better against the mouse in all groups. It also reveals a design deficiency of the stylus that could easily be overcome. Second, while the mouse and stylus yielded similar performance, observation of people using the mouse to make marks other than straight strokes suggests that the mouse may be inferior to the stylus in other situations.

Hypothesis 6. Subjects made essentially straight strokes. However, there was evidence of reselection (where subjects would begin a straight stroke and then change stroke direction in order to select something different) even in the hidden groups. This casts doubt on our initial assumption that subjects in the Marking group would begin to think of the task as making marks, instead of making menu selections. Instead, it suggests that they thought of the task in terms of making selections from the exposed menus they were trained on, which now happened to be hidden. Marks themselves do not afford reselection, whereas pie menus do.

The fact that the marking metaphor was not supported as strongly as we hypothesized may account for the fact that no major differences were found between the Hidden and Marking groups. For example, the presence or absence of an ink-trail did not appear to make any discernible difference to stroke shape.

Design Implications. Since users tended to make straight strokes we are optimistic that users are rehearsing the physical movement required to make marks as they perform menu selection. This bodes well for learning. There was some evidence of non-straight strokes which appeared to be reselection in the Marking group but it was not overwhelming. Perhaps in the context of a mark recognition system a user will learn that reselection results in a mark that cannot be recognized and that reselection is not possible when using a mark.

Hypothesis 7. Performance across trials was uniform for exposed menus but underwent steady and significant improvement across trials for hidden menus (both groups). We argue that the performance limiting factor for exposed menus is the difficulty of articulating selection actions, whereas in the hidden groups the limiting factor is the time it takes to evoke or construct the correct mental representation. Articulation skills were acquired fairly rapidly and reached stable performance. Thus performance in the Exposed group provides a baseline measure that users of hidden menus approach.

Design Implications. The substantial improvement for hidden menus over only 40 trials suggests that if the menus contain meaningful and frequently used commands, users will acquire the necessary skills quickly and easily. Both response time and error rates can be expected to rapidly improve with time. The question of *how much* practice is necessary for hidden menu performance to equal exposed menu performance, and how that varies with number of items per menu is an issue for

further research and analysis. Meanwhile, we can be confident that small numbers of items will enable users to quickly begin marking ahead.

3.4. SUMMARY

This chapter investigated basic questions concerning design variables of marking menus: how many items can marking menus contain; what kinds of input devices can be used in conjunction with marking menus; how quickly do users learn the associations between items and marks; how much is performance degraded by not using the menu; is there any advantage in using an ink-trail. An experiment addressed these questions by varying the number of items per menu and input device for three groups of subjects, and asking them to select target items as quickly as possible from a series of simple pie menus. One group selected from menus that were visible at all times, another group selected from menus that were hidden, and the final group selected from menus that were also hidden, but had the additional visual feedback of a cursor ink-trail. The differences in group conditions were intended to separate articulation and cognitive aspects. The experiment compared selection times and error rates. In addition, learning effects were analyzed.

The results of the experiment indicate that non-hierarchic marking menus, or specifically the action of using a mark to select from a menu, is a useful idea. Our results indicate that: (1) four, eight and twelve items menus are suitable for marks; (2) if that number of items is kept low (e.g., four, eight and twelve), users will be able to use marks very early in practice; (3) higher numbers of items are possible but require more practice; (4) for non-hierarchic menus, users will perform as well with the mouse as they would with the stylus/tablet. Using a trackball, however, will be slower and more error-prone than using a mouse or stylus/tablet.

In terms of using marking menus in an application, the results indicate that a designer should attempt to use four, eight or twelve item menus. For example, if seven commands are to be placed in a menu, the designer should use an eight-item menu and leave one item blank or duplicate one of the more popular commands in the extra item. Although this experiment did not address this issue, it may also be also be advantageous to maintain consistent subdivisions for menu items. For example, use four and eight item menus (items on 45 angles) but not twelve item menus (items on 30 angles).

The results are encouraging because there are many applications where menus which have a small number of items could be effective. For example, *Microsoft Word* has seven groups of function icons that appear in the "ribbon" and "ruler" display area. These icons could be grouped into seven marking menus containing four or less items. Each group of icons could be replaced by a single icon which when pressed displays a four-item marking menu. The elimination of icons would allow space to display more text, or other or larger function icons (larger icons make pointing to them easier). The graphics editor in Microsoft Word already has tool pallet icons that work this way but uses pop-up linear menus. The popular Macintosh drawing program called *Canvas* also uses a similar scheme. Many of the menus that pop up from tool pallets icons in Canvas have twelve or fewer items.

While there are many situations where menus with twelve items or less may be sufficient, there are also many situations where menus contain more that twelve items. For example, font menus, large color pallets and paragraph style menus commonly contain more than twelve items. Chapter 5 shows that hierarchic marking menus make it possible to use a mark to select from a large number of items.

Given the results of this experiment, we can now apply them to the design of hierarchic marking menus. We recommend that hierarchic marks contain only menus with even numbers of items and the number of items be less than twelve. Because the poor performance of the trackball in this experiment, it would not be suitable for hierarchic marking menus. Also it would be worthwhile to see if the mouse performs as well as the stylus on "zig-zag" marks. Chapter 5 applies these design recommendations and evaluates hierarchic marking menus.

Despite the value of such controlled studies, there are a number of questions which can only be answered by careful design and implementation of marking menus in real applications. How long will it take for users to start using marks? How intensely will users use marks? What are the issues involved in integrating such a mechanism into a larger, more complex interface? Chapter 4 addresses these types of questions by means of a case study of user behavior using a marking menu for a real task.

Chapter 4: A case study of marking menus

The previous chapter has developed an empirical understanding of non-hierarchic marking menus. From this understanding, guidelines for designing marking menus and interfaces that use marking were generated. In this chapter we report on a study which applies those guidelines to the design of marking menus in an application and we evaluate user behavior while operating this application. The application was designed to solve a real world task and was used in accomplishing real work for a project not related to this thesis. The intention was to gain insight on integrating marking menus with other interface components and to find out how well marking menus perform in everyday practical work situations.

4.1. DESCRIPTION OF THE TEST APPLICATION

A conversation analysis/editor program, named *ConEd*, developed at University of Toronto, was used as a test application for marking menus (Sellen, 1992). By digitizing audio from a conversation among four people, data were collected concerning who is speaking and when. The conversation analysis/editor program is then used to display this data in a "piano roll" like representation. The program runs on a Macintosh computer. Figure 4.1 shows a typical display of the data window. The y-axis represents the four participants in the conversation, and the x-axis represents time. A black rectangle indicates that a particular person is speaking for a duration of time (this is referred to as an event). The window can be scrolled to reveal different moments in the conversation. Besides displaying the data, the application can be synchronized to a video recording of the conversation. As the video plays, the application moves a horizontal bar across the window to indicate

the current location in the conversation. If the bar moves past the right side of the display, the application automatically scrolls to the next section of conversation.

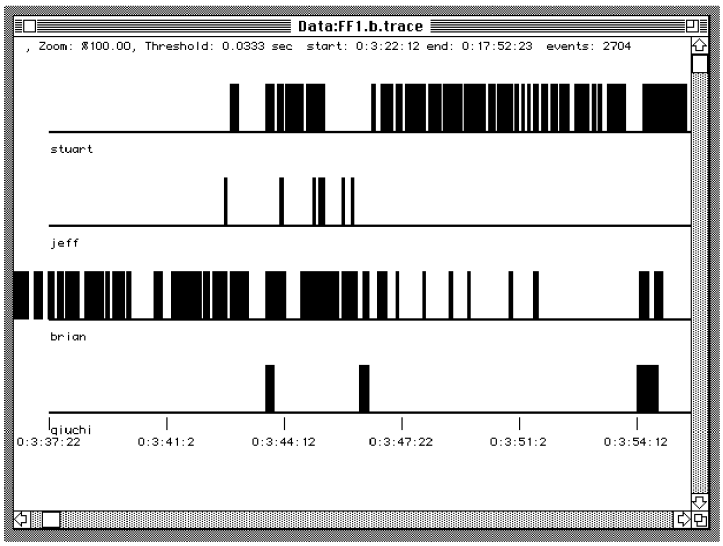


Figure 4.1: The "piano roll" representation of speaker versus time in ConEd.

Data can be edited as well as viewed with this application. Such things as coughs and extraneous noises need to be deleted. Other pieces of conversation, such as laughter, must be tagged for later analysis. Very often events must be added or extended because the automated speaker tracking system was not accurate enough.

Typically, a user sits in front of the Macintosh and video monitor, watching the video and editing events in real-time. Most of the time, a user operated the video transport with the left hand and the mouse with the right hand.

A marking menu triggers the six most frequently used commands, which consisted of commands that coded and edited the blocks of speech. The amount of coding and editing required was extremely high. Over 18 hours of operation, the two users performed 5,237 selections.

4.2. HOW MARKING MENUS WERE USED

4.2.1. The design

Figure 4.2 shows the marking menu used in ConEd. This menu can be popped up by pressing-and-waiting with the mouse in the "piano roll" window. Alternatively, a mark can be made to select the command. A user can issue six commands using this menu: laugh, delete, add, fill-in, ignore, and extend.

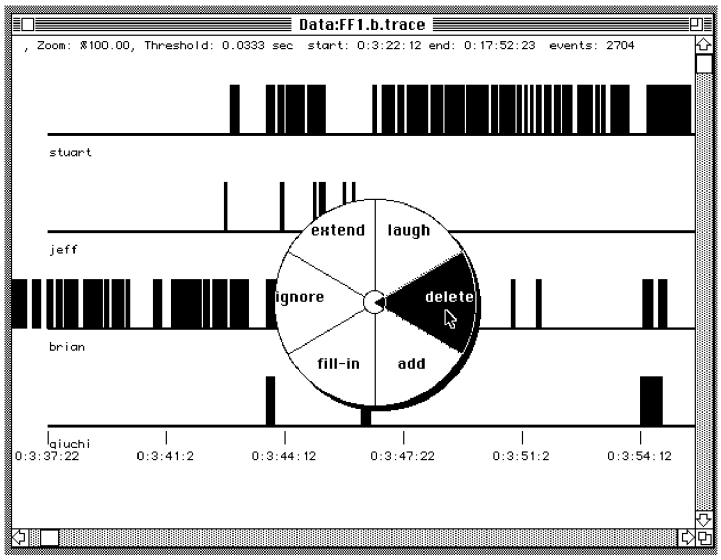


Figure 4.2: The six most frequently used editing commands are placed in a marking menu in ConEd.

Delete: The "delete" command deletes events. If the starting point of the delete selection/mark is made over an event, then that event is deleted. If the starting point is not over an event, then the events lying between the starting and ending points of the selection/mark are deleted. See Figure 4.3.

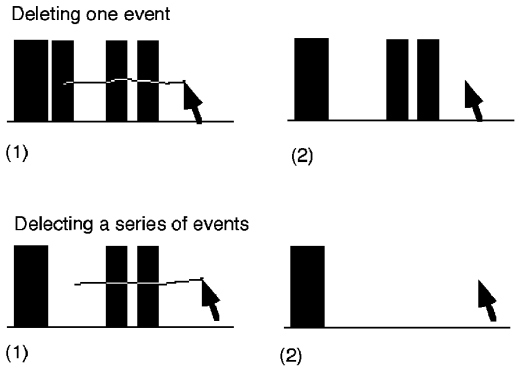


Figure 4.3: Events can be deleted one at time, by pointing to the event, or in a series by drawing over a series events.

Add: "Add" allows new events to be added. The starting point of the add selection/mark defines the beginning of a new event. The starting point of the following add selection/mark defines the end point of the new event and causes it to be displayed. If add is performed over an existing event, it is disregarded. See Figure 4.4.

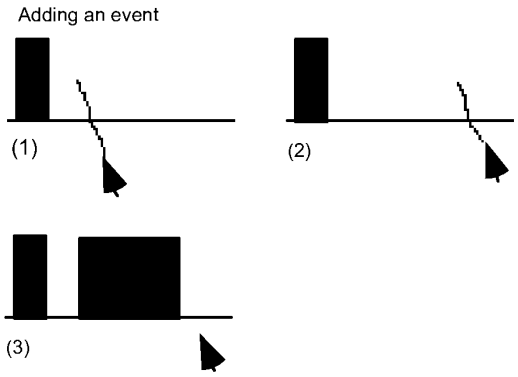


Figure 4.4: Events are added by specifying a starting point followed by and endpoint.

Extend: "Extend" elongates an event. The starting point of the extend selection/mark defines the length of the elongation. Either the start or the end of an event can be extended. If the selection/mark is made between two events, the event whose starting or ending point is closest to the starting point of the selection/mark is elongated. If extend is started over an event, it is ignored. See Figure 4.5.

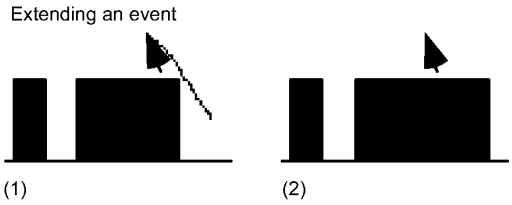


Figure 4.5: Events can be extended by pointing to the location of the extension.

Fill-in: "Fill-in" allows a gap of silence between two events to be filled. The two events are replaced by one long event. The starting point of the selection/mark indicates the gap to be filled. If Fill-in is ignored if started over an event. See Figure 4.6.

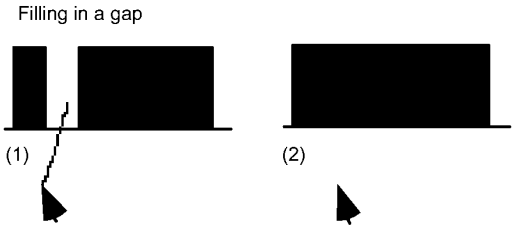


Figure 4.6: Gaps between events can be filled in by pointing to the gaps.

Ignore and Laugh: "Ignore" and "Laugh" allow events to be coded as special types. For example, speaking events generated by laughter must be tagged so they can be excluded from analysis of the conversation. Back-channel events (i.e., someone saying "uh huh" or "yes" but not trying to interrupt while another person is talking) must also be tagged. The starting point of the ignore or laugh selection/mark defines the event being coded. Either command is disregarded if not started over an event. See Figure 4.7 and 4.8.

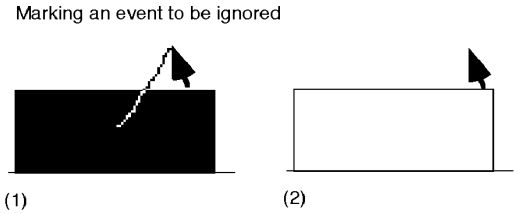


Figure 4.7: An event can be marked to be ignored by pointing to it.

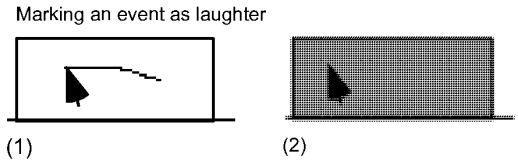


Figure 4.8: An event can be marked as laughter by pointing to it.

4.2.2. Discussion of design

Menu item choice

ConEd has more commands than the six contained in the marking menu. There are several reasons for placing this particular set of commands in a marking menu. First, the experiment in Chapter 3 showed that even numbers of items, up to twelve, enhance marking performance. Hence, six is within this range. Second, a requirements analysis told us that these six commands are the most frequently used. This implied several things. First, it would be advantageous if these commands could be invoked quickly. Therefore, marks would be suitable for these commands since marks can be issued very quickly. Second, these commands would be good candidates for marking menus because using the commands frequently would help a user memorize the associations between marks and commands. This, in turn, would lead to users using the marks.

Spatial aspects

Use of end points: While the marks used in marking menus are very simple, other features of a mark besides its angle can be used. The starting and ending points of a mark are obvious candidates. Features of a mark have been used in a similar manner by previous researchers (Coleman, 1969; Rhyne, 1987).

A requirements analysis revealed that the most frequent operations would involve selecting an event and applying an operation to it. Thus, marking menus were used in an object oriented manner—the starting point of the selection/mark indicates the object of the command. Note that this is not always the case. For example, the extend command does not point to an event to be extended but to the location of the extension. The particular event to be extended is inferred by the system. However, we found that this inconsistency caused no problems for the user.

The combination of pointing and marking produces the feeling of directness one gets when pointing and moving in objects in direct manipulation interfaces. When using marks in ConEd, there is no sensation of explicitly making a selection before applying an operation.

Use of horizontal/vertical dimension: Spatial commonalties between the representation being edited and the direction of menu items can be used to determine the assignment of directions to commands. For example, horizontal and vertical aspects of the marks can be exploited. Specifically, the direction of a mark means the objects along that direction can be selected using the mark. The delete command is an example of this. Preliminary design testing indicated that deleting a series of horizontal events was a very frequent operation. This meant putting the delete command at a horizontal menu position would allow deletion of several events in a row. This "trick" was found to be very useful.

Spatial commonalties can also be used to provide mnemonics to help recall the mark associated with a command. The add and extend commands are examples of this. Both these commands require a vertical time location value. A common way to indicate location along the horizontal is by a vertical "tick". This serves as a mnemonic for the marks associated with these commands.

Temporal aspects

Time versus space pointing: There are many temporal aspects of a mark that can be used. For example, the speed of drawing (i.e., fast or slow, fast at the start then slow at the end) or the time when a drawing occurred can be used. The aspect we exploited is time-based drawing. Specifically, the add command has two modes of operation. The first mode has been described already—the starting location of the mark is used to define the start or end of the event being added. However, if ConEd is synchronized to the playback of a video tape of the conversation, the start or end

point of an event is defined by the current playback location of the video, not by the spatial position of the mark. This is analogous to indicating a point in time by saying "... now" instead of pointing "here". However, users did find that adding events while the tape was playing was difficult.

Inverting semantics of menu items

ConEd's marking menu permits a unique method for undoing. Commands can be undone in ConEd in the standard Macintosh manner (i.e., by pressing the "undo" key or selecting "undo" from the Edit menu). The limitation of this approach is that only the most recent edit can be undone. However, the laugh and ignore commands can also be undone by repeating the laugh or ignore command on the same event. The first laugh mark turns an event into a laugh event. A second laugh mark toggles the event back to a normal event. Therefore, even if these types of edits are not the most recent, they still can be undone.

Toggling the way the laugh and ignore commands work is an example of inverting a menu item semantics. In this case, once a function in a menu is invoked, it is replaced by the corresponding inverse function. Hence, the semantics are "inverted". For example, selecting "open" will invoke the open function and replace the "open" menu item with "close". There are several reasons why inverting semantics are important to marking menus. First, inverting semantics allows extra functions to be associated with a menu without increasing the number of items in a menu. This helps keep the number of items in a marking menu small, which in turn makes marking easier. Second, inverting semantics provides a mnemonic to help recall the association between mark and function. For example, if one remembers the mark associated with "open" then one can recall the mark associated with "close", because the two functions are the inverse of each other.

The role of command feedback

There are several ways that a user receives command feedback using marking menus in ConEd. When using the menu, the user knows which command is about to be executed because the name of the command appears highlighted in the menu. When marking, a user can either recall the mark/command correspondence or watch the results of drawing the mark. We have observed that, as users gain more experience with marking menus, they graduate from watching the menus and

marks, to watching the results of their actions to determine if they have selected the correct command.

Context also plays an important role in determining the command a mark triggers when semantic inversion is being used. For example, events that were marked as "laugh" events appeared in a gray color. This feedback provides essential information to the user that a "laugh" mark on this event was not actually a laugh command but a command to "unlaugh" the event.

In ConEd, a marking menu interaction combines object selection and command application. Typically, in mouse-based direct manipulation systems, these two actions are distinct. For example, a user selects an object by pointing to it; the object then appears "selected"; next, a command is applied to the object by selecting from a menu. When using the marking menu in ConEd objects never appear "selected". It is interesting to note that none of the users ever reported missing it. We can speculate the reason for this is that the combination of selection and marking is intuitive (i.e., emulates our experiences with pen and paper), and the result of a command appeared quickly enough that the starting point of the mark was still in visual image storage.

4.3. ANALYSIS OF USE

The behavior of two users using ConEd over an extended period of time was studied. Both users were employed to edit conversation data. The edited data was used in a research project which was independent of this research thesis. Therefore, a user's main motivation was not to use marking menus, but to complete the task of editing and coding the data. The amount of data to be edited was extremely large and therefore the users were mainly interested in performing the edits as quickly as possible.

The first user (user A) was an experienced Macintosh user and was also familiar with video technology. User A was also familiar with the intentions of the conversation analysis experiment. Given this profile, user A could be considered an expert, although unfamiliar at the start of the study with marking menus. The second user (user B) could be considered a novice. While user B did have some computer experience, it was mainly with the MS-DOS environment, not the Macintosh. Therefore, user B not only had to learn how marking menus worked,

she also had to learn the many details of the Macintosh interface, and the correct way to edit the conversation data.

It was explained to both users how the conversation data was to be edited. The goal of editing was to ensure that the data matched the conversation patterns on the video tape. Users edited the conversation patterns using ConEd and then checked their work by playing back the video tape and comparing the audio of the conversation with the data in ConEd. This process was very interactive. The user played the video and watched the conversation data "playback" on ConEd. When the user saw a piece of data that did not match the audio on the video tape, the user edited the data, then rewound and replayed the video tape and data to ensure the edit was correct.

Each user had the interface to ConEd explained to them and some example edits were performed for their benefit. In particular, the commands in the marking menu were carefully explained and demonstrated. The menu and mark mode was explained and demonstrated, as well as the ability to reselect menu items or confirm a mark. We then verified that the user understood the marking menu by having them perform a few edits using the menu and marks.

Data on user behavior was gathered by recording information about a marking menu selection every time a selection was performed. The information included the time the selection was made, the user's name, the item selected, the mode used to select the item (menu or mark), the length of time the selection took, and the path of the mark or the series of reselections from the menu. A user only needed to register his or her name at the start of an editing session. The rest of the trace data was accumulated transparently.

User A edited for a total of 8.55 hours over approximately six days. User B edited for 10.1 hours over a 29 day period. Most editing sessions lasted one to two hours.

After completing the task, the users were asked to fill out a questionnaire on their experiences using marking menus. The intention of the survey was to reveal users' perception of marking menus and gauge their level of satisfaction.

4.3.1. Issues of use and hypotheses

The main goal for tracing menu usage was to understand how users behave when using marking menus. Specifically, we wanted to find out whether or not in a real

work situation users would evolve from using the menus to using marks and the characteristics of this evolution. In Chapter 2, we described the design of marking menus and how it embodied several assumptions concerning user behavior. The assumptions are that, first, a user will begin by using the menu but with experience the user will evolve to using marks, and second, as part of this evolution, users will make use of intermediate modes of selection (i.e., mark-confirmation and reselection). We wanted to discover whether or not user behavior reflected this in order to prove our assumptions about the novice to expert transition, and to verify that these intermediate modes are actually needed in the marking menu design.

With these goals in mind, we formed the following hypotheses about user behavior with the marking menu in ConEd:

(1) Menu mode will dominate a user's behavior at first. However, with experience, mark mode will dominate.

(2) The more frequently a command is executed the more likely it is to be invoked by a mark.

(3) Users will make use of mark-confirmation and reselection but with experience this behavior will disappear.

The following hypotheses test our assumptions concerning the differences between novice and expert behavior. Specifically, expert behavior will demonstrate faster selection times and more efficient movement than novice behavior.

(4) Time to select from the menu, even with the wait delay subtracted, will be greater than time to make a mark.

(5) With experience, the average length of a mark and time required to make a mark will become smaller.

4.3.2. Results

We analyzed the data from the two users separately for several reasons. First, we were concerned with individual differences. Combining the data would have masked these differences. Second, this study was not a controlled experiment. The data being edited varied, as did the amount of time and number of sessions the users worked. Thus, there was no logical way to merge the users' trace data.

Finally, our two users were very different in attitude and expertise, and therefore combining the trace data would have been inappropriate.

Menu versus mark usage

Hypothesis (1) was shown to be true. Figure 4.9 shows the percentage of times a mark was used to make a selection (as opposed to using the menu to make a selection) versus the total number of selections performed. Over time, marking dominated as the preferred mode of selection. For user A, out of a total of 3,013 selections 6.6% used the menu. For user B, out of a total of 1,945 selections, 45% used the menu.

There are several interesting observations concerning the usage of marks over time. First, when users returned to using ConEd after a lay-off period, the percentage of marking dropped. Figure 4.10 shows that several long lay-offs from ConEd occurred during the study. Note the correspondence between periods of inactivity and dips in mark usage. This indicates that mark/command associations were forgotten when not practiced. However, the amount of fading reduced with the amount of experience (i.e., the dips in Figure 4.9 become less pronounced with experience). Second, note how user B's mark usage rises dramatically at approximately 650 selections. We believe the reason this happened was because user B was a very cautious and inexperienced user. For user B, every command was a new experience. For example, user B needed help opening, saving, and closing files. User B commented that it took her several hours to get comfortable with the video machine and the Macintosh interface before she could begin to think about using marks.

Hypothesis (2) claims that the more frequently a command is used, the more likely it will be invoked by marking. This is based on the assumption that frequent use demands fast interaction and this motivates a user to learn the association between mark and command. Some commands were used more frequently than others. The horizontal axes in the graphs in Figure 4.11 shows this. Hypothesis (2) is shown to be true by a strong correlation between the frequency at which a command was used, and the frequency at which that command was invoked by a mark. Figure

4.11 shows a linear relationship between frequency of command and frequency of marking (for user A, $r^2 = 0.81$, $p<.05$; for user B, $r^2 = 0.88$, $p<.05$)[15].

[15] Note that the add command was not used in this analysis because it appeared to be an outlier point. Its frequency of marking was much lower than the rest of the commands. Our users reported that the add command didn't work correctly all the time. Therefore we assume that users were not as confident about using a marking for the add command as they were for the other commands and hence the outlying mark frequency.

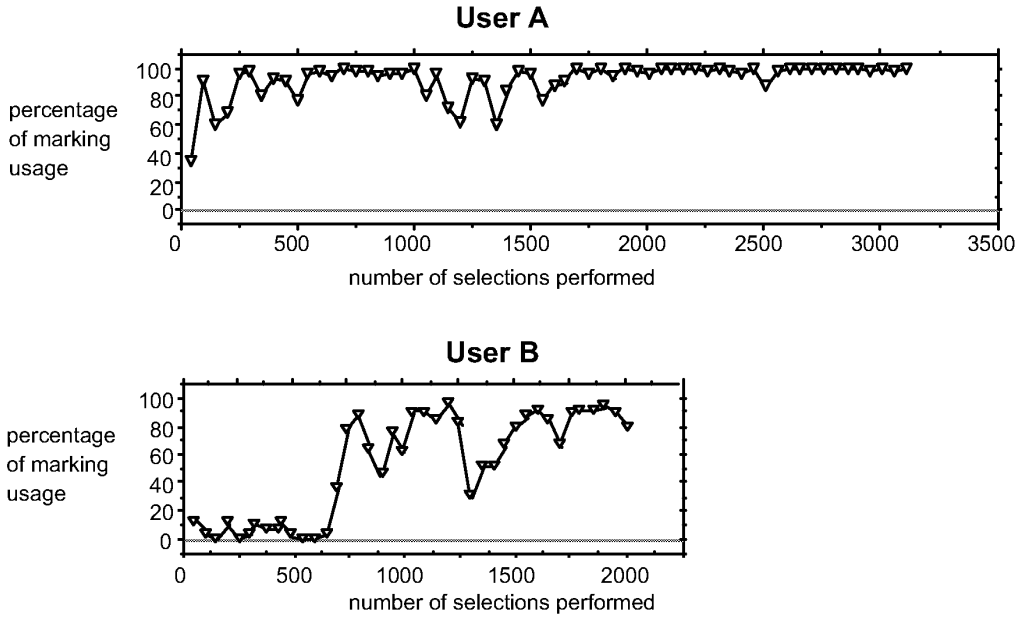


Figure 4.9: With experience, marking becomes the dominate method for selecting a command. Each data point is the percentage of times a mark was used in 50 selections.

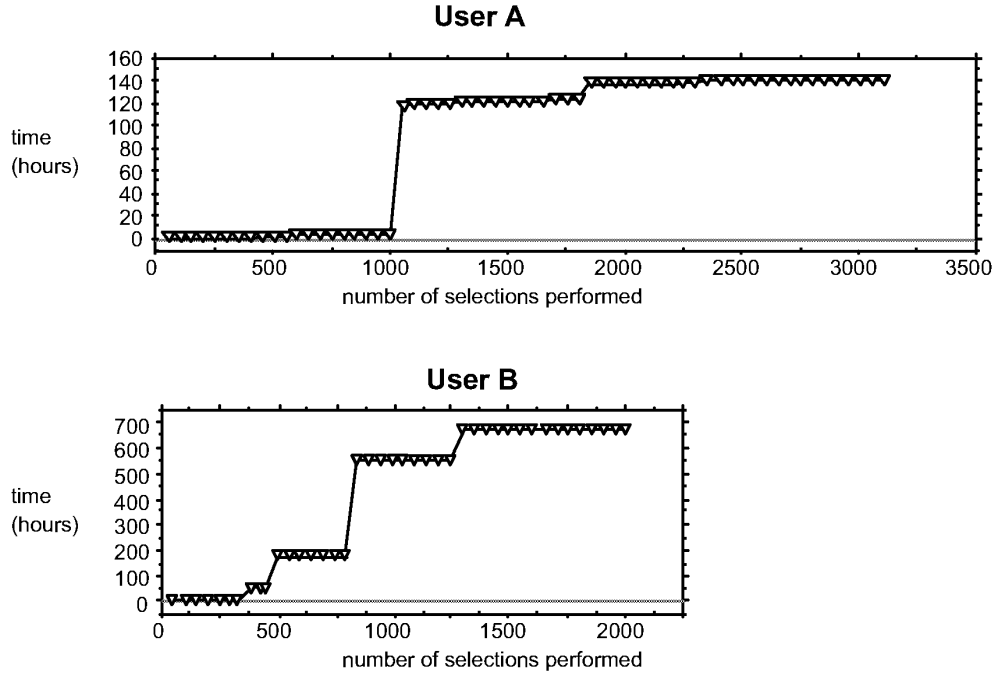


Figure 4.10: Usage of ConEd spanned many days with "lay-offs" between sessions. Steps in the graph represent layoff periods. Dips in the graphs in Figure 4.9 correspond to lay-offs. After a layoff, a user had to resort to the menu to reacquaint oneself with the marks (especially user B).

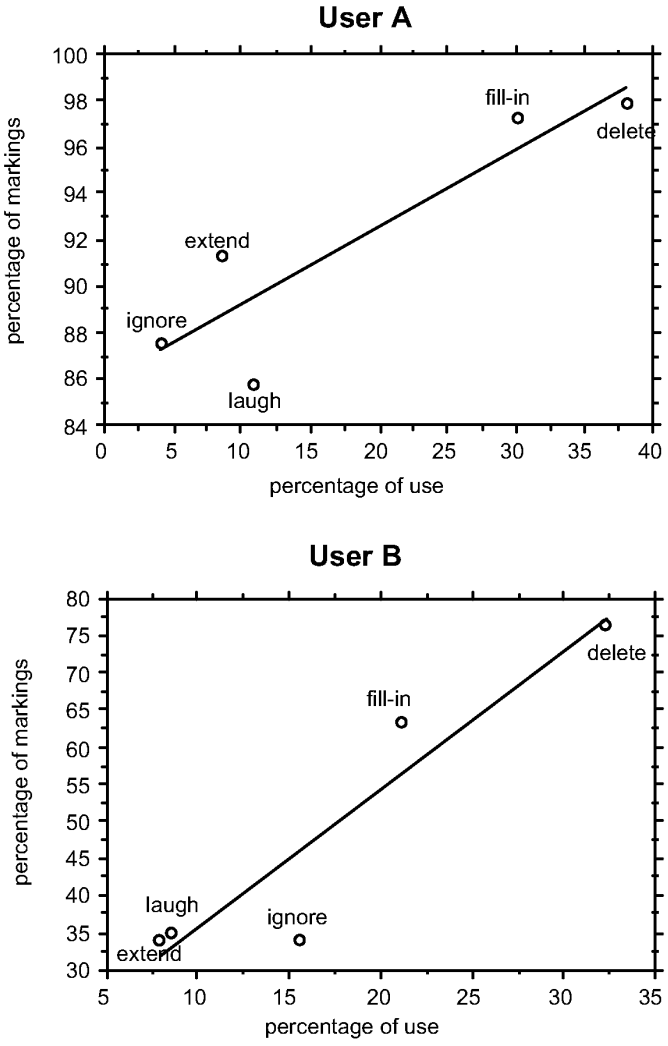


Figure 4.11: The more frequently a command is invoked the more likely it is to be invoked by a mark. The vertical axes show the percentage of times a mark was used to invoke a particular command. The horizontal axes show the percentage of times a particular command was invoked using either a mark or the menu.

Mark confirmation and reselection

As predicted by hypothesis (3), users did make use of the ability to confirm a mark and reselect from a menu but with experience this behavior disappeared. We draw evidence for this from Figure 4.12 as follows. Figure 4.12 (a) plots three types of behavior when using the marking menu:

• *mark*: a selection is made by making a mark;

• *mark-confirm*: a selection is made by making a mark but waiting at the end of the mark, thus popping up the menu to confirm the mark selects the correct item;

• *mark-corrected*: a selection is made in the same manner as "mark-confirm" but after popping up the menu another item is reselected.

We conjecture that these three behaviors are indicative of a user's skill in making accurate marks. Mark is the most skilled behavior. In this case, a user is so skilled at making a mark that no feedback is needed before confirming the selection. Mark-confirm is the next level of skilled behavior. In this case, a user has enough skill to make the correct mark but not the confidence to invoke it without checking it against the menu. Mark-corrected is a third level of skilled behavior. In this case, a user has made a mark, checked it against the menu and has corrected the mark using reselection.

Figure 4.12 shows several things. First, mark-confirm and mark-corrected behavior did occur and therefore this functionality is used and needed. Second, this behavior occurs during the transition from using the menu to drawing marks. Third, when used, this type of behavior occurred less than ten percent of the time.

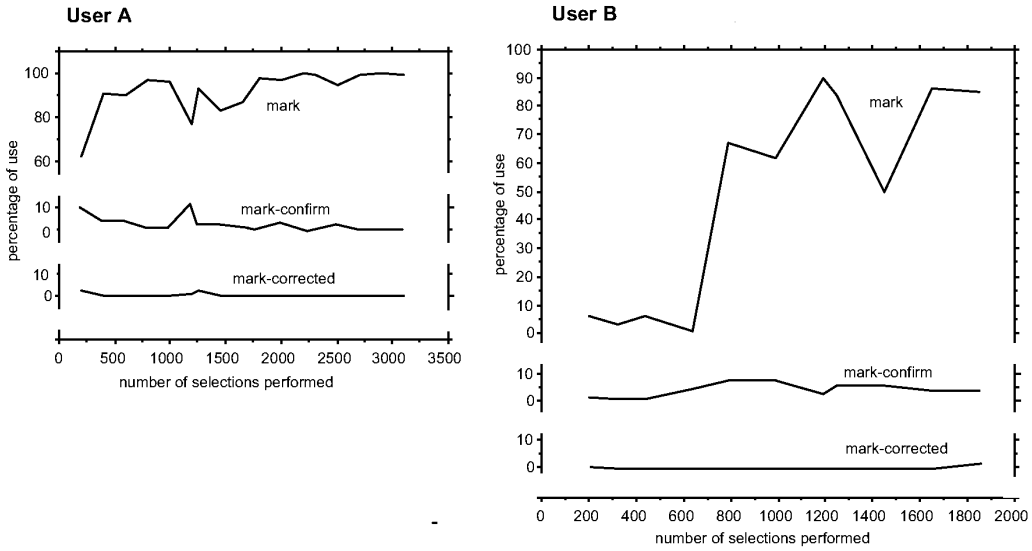


Figure 4.12: Users made use of the ability to confirm the selection a mark would make before committing to it. However, with experience this behavior disappears. Measures were averaged every 200 selections.

Reselection

Another topic of interest was whether or not users reselected when using menu mode. Figure 4.13 shows that reselection occurred less than ten percent of the time. User A demonstrated that with experience reselection disappears. However, user B did not exhibit this behavior. This is evidence that the reselection function in a radial menu is needed.

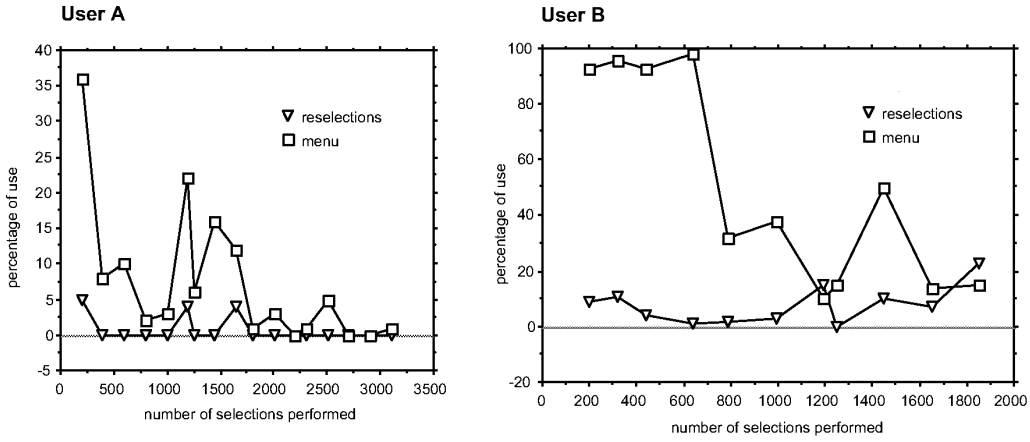


Figure 4.13: Both users utilized reselection in menu mode. While user A's use of reselection diminished with time, user B utilized reselection even after substantial experience. Measures were averaged every 200 selections.

Selection time and length of mark

Selection time is defined as the time elapsed from the moment the mouse button is pressed down to invoke a marking menu, to the moment the button is released, completing the selection from the menu. This measurement applies to either a menu or mark mode. The selection time, for both users, was substantially faster in mark mode than in menu mode. Figure 4.14 shows these differences. For user A, a mark was seven times faster than using the menu. For user B, a mark was four times faster.

Even though menu and mark mode require the same type of movement, using the menu is slower than making the mark. There are several reasons why. First, a user must press-and-wait to pop up the menu. This delay was set to 0.33 seconds. However, as the fourth column in Figure 4.14 shows, even with this delay subtracted from the menu selection time, a mark is still much faster (i.e., user A is 4.2 times faster, user B is 3.0 times faster). The user most likely waits for the menu to appear on the screen. Displaying the menu takes the system about 0.15 seconds. The user must then react to the display of the menu (simple reactions of this type take no more than 0.4 seconds, according to Card, Moran, & Newell, 1983). However, when making a mark, the user does not have to wait for a menu to display and react to its display. Thus, a mark will always be faster than menu selection, even if press-and-wait was not required to trigger the menu. Figure 4.15 graphically shows this. The

	average time to perform a selection (seconds)		
	mark	menu	menu - delay
User A	0.18 ± 0.004	1.097 ± 0.042	0.763
User B	0.404 ± 0.01	1.543 ± 0.052	1.209

Figure 4.14: On average, marks were much faster than using the menu. For user A, a mark was seven times faster than using the menu. For user B, a mark was four times faster. Confidence intervals are at 95%.

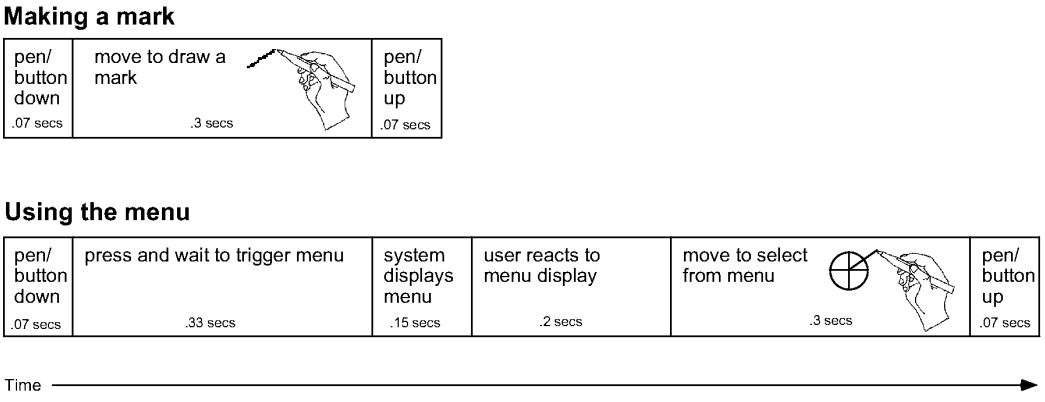


Figure 4.15: Why a mark is faster than using a menu. The typical durations of various events that take place when making a selection are depicted. Even if press-and-wait was eliminated from menu selection it would still take longer than making a mark because of the additional events.

fourth column of Figure 4.14 provides evidence of this. This supports hypothesis (4).

Selection time, using a mark, decreased with practice, however the decrease was very small. In view of the very fast times for marking performance, this is good news, since this means that, even early in practice, novice performance was very similar to expert performance. The decrease in selection time was less than 0.1 seconds. For this analysis we used the Power Law of Practice (performance time declines linearly with practice if plotted in log-log coordinates (Snoddy, 1926)). Linear relationships were found for both users (an analysis of variance of linear

regression used; for user A, $F(1, 1654) = 166.5$, $p<0.0001$; for user B, $F(1, 541) = 23.03$, $p <0.0001$).[16]

The average length of a mark decreased slightly with practice for user B, but not for user A (an analysis of variance of linear regression used; $F(1, 2813) = 10.82$, $p<0.01$). The average length of a mark was approximately one inch. The delete mark was excluded from this analysis because its length was used to indicate a range of events.

Given these results for mark time and length we accept hypothesis (5)–mark time decreases with practice, but only in the case of user B is there support for the hypothesis that mark length also decreases with practice.

Users' perceptions

Both users were given a questionnaire after performing the editing task. The intention of the questionnaire was to discover if a user's perception of marking menus matched their behavior and also to allow us to obtain information not captured in the trace data.

An important parameter not captured in the trace data was selection errors. The reason for this is that prior to a selection we did not know what item a user intended to select. Therefore, when a selection was made, we could not tell whether or not the user had successfully invoked the desired selection. Since users should be the judges of what acceptable error rates are, we simply asked them how many errors they made with the marking menu: no errors, few but acceptable, or too many? Both users reported "few errors but acceptable".

Users perceived marking menus as a tool that helped them get the task completed quickly. Both users reported that their performance with the marking menu was "fast". User B, the less confident user, however, reported she didn't have enough regular experience with the marking menu to be completely comfortable drawing marks.

---

16 Linear relationships were determined by estimating a regression line using an analysis of variance approach (see Appendix A further explanation).

Both users confirmed the differences we found in performance between menu and mark mode. The trace data indicates that using a mark was substantially faster than using the menu. Both users reported a mark was "much faster" than using a menu.

We were also interested in how users recalled the relationship between command and mark. We suggested to both users three methods they could have used to recall mark/command associations. The first is by recollecting the spatial layout of the menu. The second is by rote–"this mark produces this command". The third method is the situation where one is so skilled at performing the mark/command that one is not aware of performing an explicit association—one just "does" the correct mark.[17] User A reported using the second technique, except in the case of "delete" for which he used the third method. User B reported using the first technique. If we assume that the three methods represent various stages of increasing practice, we can conjecture that user A was farther along in expertise and practice than user B. Our data shows this to be true (i.e., user A performed 1,068 more selections than user B).

Marking menus versus linear menus

The results from this study allow us to build on the comparison between marking menus and linear menus discussed in Chapter 2. When a user is familiar with the layout of a menu, selection from a radial menu will be faster than selection from a linear menu. Callahan et. al., (1989) provide empirical evidence of this for eight-item menus. It is possible that a linear menu may be more suitable when there an natural linear ordering to the menu items and a user must search the menu for an item before making a selection. Alternatively, a radial menu may be more suitable when there is a natural radial ordering of menu items. However, as shown by both Card (1982), and McDonald, Stone, & Liebelt (1983), the effects of organization disappear with practice. Callahan et. al., (1989) provide evidence that, for eight-item menus, even when menu items have a natural linear ordering, selection using a radial menu is still faster and less error-prone than selection using a linear menu.

Drawing from data in an experiment by Nilsen (1991), we can directly compare six-item marks and six-item pop-up linear menus. In Nilsen's experiment, a selection

---

[17] Recall may also be by rote in this case, but, since recall is so quick, users may perceive it differently.

from a six-item linear menu required on average 0.79 seconds. In our study, user A and user B required, on average, 0.18 and 0.40 seconds respectively to perform a selection using marks. Furthermore, in Nilsen's experiment the subjects' only task was to select from a linear menu. Therefore, one would expect selection speed to be artificially fast. In our study, in contrast, the users were performing selections in the context of other real world tasks.

The fact that radial menus are faster to select from than linear menus is not the complete story. Selection using a mark is faster than selection via a radial menu. This case study has shown marks to be substantially faster than selection from a radial menu, even if press-and-wait time is factored out. The reason for this is, when selecting using a menu, a user must react to the display of the menu before selecting. However, making a mark involves no reaction time. Hence selection with the mark is faster by design. Obviously faster selection with a mark comes at the price of higher error rates, especially when menus become dense. But the results from this chapter, Chapter 3, and Chapter 5 indicate that menus of breadths four, six and eight have acceptable error rates.

Thus, we can conclude that if menus contain an even numbers of items and less than ten of them, and users frequently use the menus, marking menus will have a distinct advantage over linear menus. Data from this chapter tells us that using the marks will be approximately 3.5 times faster than selecting from a radial menu. We conjecture this speed-up figure would be greater if compared to linear menus.

As a practical example of the impact of this speed-up, we can consider the performance of another real user using ConEd.[18] This user performed 16,026 selections during 36 hours of work. Her average time to select using a mark was 0.23 seconds. Her average time to select using the menu was 1.48 seconds. If the task had been done exclusively with a radial menu that did not use a press-and-wait delay of 0.33 seconds, the average time to select from a menu would have been 1.07 seconds, and 16,026 selections would have required 17,099 seconds in total. However, when using the marking menu, the menu was used for 185 selections and marks were used for 15,841 selections. Thus, menu selections required 185 × 1.48 =

---

[18] A third user used ConEd extensively over a long period of time but she was not included in this study because she assisted in the design of the marking menu used in ConEd and ConEd itself. Therefore, we felt she would not be an unbiased user of marking menus.

274 seconds. Selections made with a mark required 15,841 × 0.23 = 3627 seconds. This results in the 16,026 selections requiring 3901 seconds in total. Thus using a marking menu, as opposed to a radial menu that popped up immediately, saved the user 17,099 - 3,901 = 13,198 seconds or 3.66 hours.

4.4. SUMMARY

This chapter has described a case study which served two purposes. First, the case study involved designing an application that used a marking menu. From this exercise we gained insights on design. Second, data on two users' behaviors using this application to perform a real task was collected and analyzed. Information was collected on a user's performance using a marking menu every time a selection was performed. This information consisted of selection time, selection method, item selected, time of selection, and cursor movement. Analysis of this information allowed us to verify whether or not our assumptions about user behavior, which are embodied in the design of marking menus, are true.

This study demonstrated several things:

- A marking menu was a very effective interaction technique in this setting. Its effectiveness was contingent on applying the technique to an appropriate setting—specifically, using a marking menu to invoke a few commands that are used frequently, and require a screen location as a command parameter. Also, despite the simplicity of the mark, features of the mark, such as the start and end points, and the orientation of the mark, can be used to make interactions more efficient and easier to learn.

- A user's skill with marking menus definitely increases with use. A user begins by using the menu, but, with practice, graduates to making marks. Users reported that marking was relatively error free and empirical data showed marking was substantially faster than using the menu.

- The various modes of a marking menu (menu, mark, mark-confirmation, and reselection) are utilized by users and reflect levels of skills. In addition, when a user's skill depreciates during a long lay-off period, the user utilizes these modes to reacquire skills. We conclude that these features are a necessary part of the design,

and furthermore, interfaces which supply mutually exclusive novice and expert modes are inappropriate when a user's level of skill depreciates.

- In this setting a mark is a very fast way to invoke a command, and users, very early in practice, become skilled at making marks. Evidence of this is that selection time was much faster in mark mode than in menu mode, and did not decrease substantially with practice. This same data indicates that even if the delay time is removed from a menu selection time, menu selection is still slower than marking. This may be due to a user simply moving slower when using the menu. In theory, however, even if there was not press-and-wait delay, and the user moved as quickly in menu mode as they do in mark mode, the user would still be delayed by, first, having to wait for the system to display the menu, and, second, by their own reaction time to its display. Hence, within the limitations on the number of items in a menu described in Chapter 3, we conclude that a mark will always be faster than a menu that immediately pops up. This, of course, is dependent on the user recalling the menu layout.

We can expect hierarchic marking menus to exhibit the same performance properties as non-hierarchic marking menus, since selection from a hierarchic marking menu consists of a series of selections from non-hierarchic menus. Chapter 5 establishes the breadths and depths of hierarchy at which we can expect these properties to hold true.

Chapter 5: An empirical evaluation of hierarchic marking menus

This chapter reports on an experiment to investigate the characteristics of human performance with hierarchic marking menus. Performance using a hierarchic marking menu is affected by the number of items in each level of the hierarchy and the depth of hierarchy. This chapter reports on an experiment which systematically varied these parameters to determine the conditions under which using a mark to select an item becomes too slow or prone to errors. Increasing depth and breadth tends to degrade performance. Thus the intention of this experiment was to find an practical upper bound for these parameters. Understanding of the role of depth and breadth helps us address the types of questions one asks when designing hierarchic marking menus for an interface:

Q1: Can users use hierarchic marks? Chapters 3 and 4 have shown non-hierarchic marking menus to be useful. (Hopkins, 1991) describes how hierarchic pie-menus can be useful. Thus we can expect hierarchic marking menus, even without using marks, to also be useful. However, the question remains: Is it possible to use a mark to quickly and reliably select from a hierarchic radial menu?

Q2: How deep can one go using a mark? Just how "expert" can users become? Can an experienced user use a mark to select from a menu which has three levels of hierarchy and twelve items at each level? By discovering the limitations of the technique we are able to predict which menu configurations, with enough practice, will lead to reliable selections using marks, and which menu configurations, regardless of the amount of practice, will never permit reliable selections using marks. Also, will some items be easier to select regardless of depth? For example, it

seems easier to select items that are on the up, down, left and right axes even if the menus are cluttered and deep.

Q3: Is breadth better than depth? Will wide and shallow menu structures be easier to access with marks than thin and deep ones? Traditional menu designs have breadth/depth tradeoffs (Kiger, 1984). What sort of tradeoff exists for marking menus?

Q4: Will mixing menu breadths result in poorer performance? The experiment on non-hierarchic marking menus described in Chapter 3 has shown that the number of items in a menu and the layout of those items in the menu affects subjects' performance when using marks. Specifically, menus with 2, 4, 6, 8 and 12 items work well for marks. What will be the effect of selecting from menu configurations where number of items in a menu varies from sub-menu to sub-menu?

Q5: Will the pen be better than the mouse for hierarchic marking menu marks? The experiment in Chapter 3 compared making selections from non-hierarchic marking menus using a stylus/tablet, a trackball and a mouse. Subjects' performance was poorest with the trackball while performance with the stylus/tablet and mouse was approximately equal. However, hierarchic marking menus require more complex marks. Will the mouse prove inadequate?

We are also concerned with some pedagogical issues which help us design human-computer interactions. Buxton has described the notion of *chunking* in human-computer dialogs (Buxton, 1986). For example, when using a mark to specify a "move" command, one can issue the command verb, source and destination all in one mark or "chunk". This notion is related to the concept of a "motor program" in motor control studies. A motor program is "a set of muscle movements structured before a movement begins, which allows the entire sequence to be carried out uninfluenced by peripheral feedback" (Keele, 1968).

Some systems or interaction techniques allow chunking to take place while others don't. In some systems a user can articulate a series of operations without having to wait for the system to finish each operation. This allows these commands to be chunked. For example, a user quickly clicks on three graphical buttons without having to wait for each button to complete its operation. In this case, the user may perceive the three clicks as a chunk. If the user was restricted to wait for each button to complete its operation before clicking on the next button, the user may not

perceive the three operations as chunk. Hence, this indicates that something as low-level as input event handling policies can affect user perception and behavior.

Relative to marking menus, the phenomenon of chunking occurs when a user, rather than articulating a selection from a hierarchic menu as a series of directional strokes separated by pauses in movement, performs the entire series of selections in one fluid movement or "chunk". We speculate that chunking is related to expertise. The more expert a user becomes with an interface the more the user chunks. This experiment provides the opportunity to investigate this phenomenon.

5.1. THE EXPERIMENT

5.1.1. Design

In order to determine the limits of performance, we needed to simulate expert behavior. We defined expert behavior as the situation where the user is completely familiar with the contents and layout of the menu and can easily recall the mark needed to select a menu item. To make subjects "completely familiar" with the menu layouts we chose menu items whose layout could be easy memorized. We tested menus with four, eight and twelve items. For menus of four items, the labels were laid out like the four points of a compass: "N", "E", "S" and "W". This type of menu we referred to as a "compass4". Similarly, a "compass8" menu had these four directions plus "NE", "SE", "SW" and "NW". Menus with twelve items, referred to as a "clock" menus, were labeled like the hours on a clock.

Will users of real applications ever be as familiar with menus as they are with a clock or compass? We believe the answer is yes, and base this on three pieces of evidence. First, our own behavioral study of users using a marking menu in a real application (Chapter 4) shows, with practice, they used marks without the aid of menus over ninety percent of the time. Other researchers have reported this type of familiarity with pie menus (Hopkins, 1991). Second, Card (1982), and McDonald, Stone, & Liebelt (1983) report that effects of menu organization disappear with practice. In other words, with practice, users memorize menu layouts and navigate directly to the desired menu item. Finally, it must be remembered that a user does not have to memorize the layout of an entire menu. For example, a hierarchic

marking menu could contain 64 items but the user might only memorize the marks needed to select the two most frequently used menu items.

The design of a trial in our experiment was as follows. A subject was completely familiar with the menu layout and the marks needed to select an item. The system would ask the subject to select a certain item using a mark (the menu could not be popped up by the subject). The subject would input the mark and the system would then record the time taken to draw the mark and whether or not the mark successfully selected the requested item. After a series of trials, we would then vary the menu configuration and input device in order to see what effect these variables had on selection performance.

The rationale for choosing menus of four, eight and twelve items was based on the results from the experiment in Chapter 3. This experiment showed that menus with even numbers of items and less that twelve items were suitable for marking. Using four, eight and twelve item menus is a deliberate attempt to explore a reasonable range of menu breadth. We would expect that performance on a menu of four items to be quite acceptable even at extreme depths. Whereas selection from a menu structure consisting of twelve-item menus which are two levels deep, seems quite treacherous.

Using a similar rationale, we chose to evaluate menu depths from one to four. A depth of one is a non-hierarchic menu which we know from the experiment in Chapter 3 produces acceptable performance. A maximum depth of four was chosen since it is in the range where we believe performance will become unacceptable.

For the sake of brevity we adopt a simple notation in the experiment. A menu structure can be described by a tuple B,D. B is the breadth of each menu in the structure and D is depth of menu structure. For example, 8,2 menu is a menu hierarchy where every menu contain eight items and the hierarchy is two level deep. An 8,2 menu contain 64 leaf menu items. When menu structures consist of different breadth at different levels we use the notation B:B:B, where B is the breadth of a menu at a certain depth. For example, a 12:8:12 menu is menu structure consisting of a top level menu of twelve items, second level menus of eight items and a third level menus of twelve items. A 8,2 menu is represented in this notation as 8:8.

In menu structures of even moderate depth and breadth the number of selectable items becomes very large. For example, in an 8,2 menu there are 64 selectable items.

As stated earlier, we wanted to simulate the case where the user was familiar with the marks being drawn. Given the practical time constrains of the experiment we could not expect subjects to become familiar all marks. Instead we decided to use only three target selections for each menu structure. A subject could then quickly become familiar with the mark needed to make the target selection with a reasonable amount of practice. In this way, the experiment addressed the question: given that the user knows the mark and is practiced at making it, will selection be quick and reliable?

The next issue concerning targets was "which three targets"? For menus of small breadth and depth this was not a major issue as one type of selection is approximately as easy to draw as another. However, in the case of menus which consist of combinations of larger breadths and depths, some selections are definitely harder that others. For example, we observed that making the selection "12-6-3-9" from a 12,4 menu was much easier that "10-11-10-11".

Our approach was to pick three targets for each menu configuration such that one was easy, one was moderately difficult, and one was difficult. Easy targets were those that had items along the vertical and horizontal axes (on-axis items). Difficult targets were those with items not on the vertical and horizontal axes (off-axis items), and little angle change between items. Finally, targets of moderate difficulty were those with a 50% mix of on-axis and off-axis items, and a 50% mix of little and large angle changes between items. It was hoped this mixture of targets would result in behavior that would be representative of an "average selection".

In the case of menus that contain only on-axis items and large angle changes, we observed, prior to the experiment, that up and to the left selections seemed to be hardest, and down to the right selections seemed to be easiest. Thus we chose hard targets and easy targets accordingly. For moderately hard targets we chose either down-and-to-the-left, or up-and-to-the-right targets.

We also had subjects perform selections from a 12:8:12 menu. This was done so we could observe the effects of combining menus of different breadths in a menu configuration.

5.1.2. Hypotheses

Nine hypotheses are proposed:

(1) Pen outperforms mouse: The subjects will perform better with the pen than with the mouse in terms of response time and errors. Once again, the experiment in Chapter 3 showed that subjects performed marginally better with the stylus/tablet than with the mouse on non-hierarchic marking menus. However as depth increases, marks become more complex to draw and therefore the pen should be a more suitable device.

(2) Increasing breadth increases response time and errors: As breadth increases, response time and error rate will increase. The experiment in Chapter 3 demonstrates this effect for non-hierarchic marking menus. Therefore, we believe this effect will apply to hierarchic marking menus as well.

(3) Increasing depth linearly increases response time: As depth increases response time will increase. We base this on the belief that marks to access deep menu configurations will require more time to draw because they will be longer.

A study by Fischman is the most relevant work to this hypothesis (Fischman, 1984). In the study, subjects used a stylus to tap on a series of metal disks (ranging from one to five disks) that were either arranged in a straight line or in a staircase pattern that required changes in direction of 90û between disks. Changes in direction and different numbers of disks in the series roughly correspond to directional movements and different depths in hierarchical marking menus. Fischman found that response time linearly increased with the number of disks, but changes in direction did not affect response time.

(4) Increasing depth increases errors: As depth increases error rate will increase. As depth increases so does the number of times a subject has to estimate at the angle of mark needed to select an item. Therefore the error rate will increase as the probability of error increases.

(5) Inaccuracies propagate: We hypothesize that the depth at which errors take place will be on average greater than half the depth of a menu structure. Informally, we claim that inaccuracies in one portion of a mark will affect the accuracy of the remaining portion of a mark. Our reasoning is as follows: a subject uses the angle of a partially drawn mark to estimate the angle for the next portion of the mark. Inaccuracy in the first portion of the mark may then propagate into the rest of the mark, eventually resulting in an error. The effect of this is that the probability of an error increases with depth. If the probability of an error was the same at every level,

an error would occur on average at half the depth of the menu structure. However, if the probability of an error increases with depth we should see errors take place on average at a depth greater than half the depth of the menu structure.

(6) Mixing menus degrades performance: Combining menus of different breadths in a menu configuration will degrade response time and increase the error rate relative to menu configurations where all menu breadths are the same. Specifically, we hypothesize that subjects will perform better on a 12:12:12 menu than on a 12:8:12 menu, even though an eight-item menu is easier to select from than a twelve-item menu. We believe it is easier for users to select items when the difference of stroke angle needed to select different items is consistent. For example, in a menu structure consisting exclusively of eight-item menus, all items are at 45 degree angles. If a twelve-item was introduced into the menu structure, some items would be at 45 degrees while others, the ones from the twelve-item menu, would be at 30 degree angles. We believe inconsistency in "item angle" will degrade performance.

(7) On-axis items enhance performance: Marks that consist of on-axis items will be faster to draw and produce fewer errors than marks that consist of off-axis marks. This hypothesis is based on prior practical experiences using hierarchic marking menus.

(8) Drawing direction affects performance: The direction of drawing will affect performance. Specifically, marks that require drawing left to right will be performed faster than marks that require drawing right to left. Other researchers have found a similar bias in directional movements (Boritz, Booth, & Cowan, 1991; Malfara & Jones, 1981; Guiard, Diaz, & Beaubaton, 1983).

(9) Subjects will chunk: The number of pauses when drawing a selection will approach zero with practice. Once a subject starts to think of selection not as a series of strokes at certain angles, but as a mark of a certain shape, the subject will draw the mark without pauses between strokes. This hypothesis was based on our own experiences using marking menus in the laboratory.

5.1.3. Method

Subjects: Twelve right handed subjects were recruited from University of Toronto. All subjects were skilled in using a mouse but had little or no experience using the pen on a pen-based computer.

Equipment: A Momenta pen-based computer development system was used. The input devices consisted of a Microsoft mouse for IBM personal computers, and a Momenta pen and digitizer. The digitizer was transparent and placed over the screen. This allowed subject to "write on the screen" with the pen. The screen was placed in front of the subject at approximately a 45 degree angle. When using the pen the hand could be rested on the screen. The mouse was placed to the right of the screen on a mouse pad. No mouse acceleration was used and the sensitivity of the mouse was set to a value of 50 in the control panel. A setting of 50 corresponds to a one to one C:D ratio.[19]

Task: A trial occurred as follows. The type of menu structure being tested appears in the top left corner of the screen. A small circle appears in the center of the screen. A subject then presses and holds the pen or mouse button over the circle. The system then displays instructions describing the target at the top center of the screen. A subject then responds by drawing a mark that is hoped to be the correct response. The system responds by displaying the selection produced by the mark. If the selection did not match the target, the system beeps to indicate an error. The system then displays each menu in the current menu structure at its appropriate location along the mark and indicates the selection from each menu. The subject's score would be shown in the lower left of the screen. Figure 5.1 shows the experimental screen at this point. If the selection is incorrect, a subject loses 100 points and the trial is recorded as an error. If the selection is correct, the subject earns points based on how quickly the response was executed. Response time is defined as the time that elapsed between the display of the target and the completion of the mark.

A subject's score (accumulated points) is displayed in the lower left corner of the screen plotted against current trial number. The graph also shows the best score for that particular pairing of menu structure and input device. This gives subjects a performance level to compete against. This helped to ensure that subjects performed the task both quickly and accurately.

A subject's progress through the trials was self paced. Subjects could pause between trials for as long as they liked. Subjects used this pause to check their score and rest.

---

[19] See Section 2.5.3 for the definitions of C:D ratio and mouse acceleration.

Most subjects paused just a few seconds. All subjects required approximately one hour and fifteen minutes to complete the experiment.

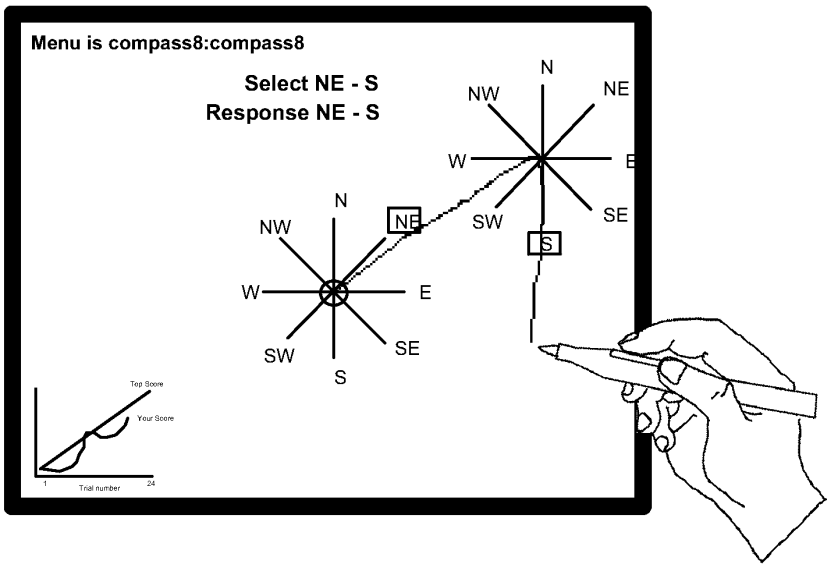


Figure 5.1: The experiment screen at the end of a trial where the target was "NE-S". After the mark is completed, the system displays the menus along the mark to indicate to the subject the accuracy of their marking.

Design: All three factors, device, breadth and depth were within-subject. Trials were blocked by input device with every subject using both the pen and the mouse. One half of the subjects began with the pen first while the other half began with the mouse. For each device, a subject was tested on the thirteen menu structures (breadths 4, 8 and 12 crossed with depths 1 to 4, plus the mixed menu structure of 12:8:12). Menu structures were presented in random order. For each menu structure a subject performed 24 trials. For the 24 trials, subjects were repeatedly asked to select one of three different targets. Each target appeared eight times in the 24 trials but the order of appearance was random.

Given this design, for each data point (a particular combination of input device and menu structure) 288 selections were collected (24 selections times 12 subjects). For the experiment, 7,488 selections were performed in total.

Before starting a block of trials for a particular menu configuration, subjects were allowed eight seconds to study the menu configuration. Before starting trials with a particular input device, a subject was given ten practice trials using the device on a 4,3 menu. This was intended to acquaint a subject with the "feel" of the input device.

It can be argued that the practice session on the 4,3 menu gave subjects an unfair advantage on this particular menu. We believe the effect was small for several reasons. First, a different set of targets was used for practice than those used in the timed trials so subjects did not become practiced at drawing the targets for the timed trials. Second, because of our choice of obvious menu labels and structure for all menus, a subject was already familiar with all of the menu structures even before practice.

5.2. RESULTS AND DISCUSSION

The main dependent variables of interest were response time and percentage of errors. Response time was defined as the time that elapsed between the display of the target and the completion of the mark. Percentage of errors was the percentage of incorrect selections out of 24 trials on a particular combination of device, breadth and depth. Figures 5.2 and 5.3 show the means tables.

Response time averaged across all subjects, breadths and depths for the pen was 1.69 seconds, while the mouse was significantly slower at 2.07 seconds ($F(1,11)=19.7$, $p < .001$). The subjects produced significantly more errors with the mouse than with the pen ($F(1, 11)=6.41$, $p < .05$). Subjects' performance with the pen was better than with the mouse both in terms of response time and percentage of errors, and therefore we accept hypothesis 1.

Breadth significantly affected both response time ($F(2,22)=91.7$, $p < .001$) and errors ($F(2,22)=130.5$, $p < .001$). Figure 5.3 shows, in general, that increasing breadth increases response time and percentage of errors. Based on these results we accept hypothesis 2.

Depth significantly affected both response time ($F(3,33)=195.4$, $p < .001$) and errors ($F(3,33)=51.5$, $p < .001$). Figure 5.3 (a) shows a linear increase in response time as depth increases. Linear regression on each device, menu breadth pair verifies this

claim (for the pen: breadth four, $r^2 = 0.79$, breadth eight, $r^2 = 0.88$, breadth twelve, $r^2 = 0.82$; for the mouse: breadth four, $r^2 = 0.73$, breadth eight, $r^2 = 0.77$, breadth twelve, $r^2 = 0.67$; $p < .001$ for all values). Figure 5.3 (b) shows that as depth increased so did percentage of errors. Given these results we accept hypotheses 3 and 4.

		Depth					
Breadth	**Device**	**1**	**2**	**3**	**4**	**Total**	**mouse & pen**
4	**mouse**	.752 (.146)	1.189 (.188)	1.797 (.407)	2.102 (.445)	1.460 (.616)	1.367 (.554)
	pen	.710 (.108)	1.098 (.142)	1.451 (.275)	1.835 (.298)	1.279 (.473)	
8	**mouse**	.932 (.211)	1.665 (.543)	2.938 (.829)	3.309 (.845)	2.211 (1.159)	2.021 (1.047)
	pen	.810 (.142)	1.411 (.298)	2.258 (.420)	2.843 (.698)	1.831 (.895)	
12	**mouse**	1.170 (.289)	1.842 (.407)	3.011 (.763)	4.181 (1.363)	2.551 (1.406)	2.250 (1.208)
	pen	.915 (.236)	1.531 (.266)	2.331 (.519)	3.022 (.443)	1.950 (.888)	
Total	**mouse**	.951 (.278)	1.565 (.484)	2.582 (.877)	3.197 (1.272)	2.074 (1.194)	
	pen	.812 (.186)	1.347 (1.272)	2.013 (.572)	2.567 (.724)	1.685 (.826)	
	mouse & pen	.881 (.245)	1.456 (.415)	2.298 (.789)	2.882 (1.075)		

Figure 5.2: Means table for response time. Each entry is average response time in seconds. Standard deviation is shown in parentheses.

		Depth					
Breadth	**Device**	**1**	**2**	**3**	**4**	**Total**	**mouse & pen**
4	**mouse**	1.43 (2.81)	4.18 (4.35)	4.24 (3.59)	5.10 (4.20)	3.74 (3.92)	3.94 (4.65)
	pen	2.19 (5.09)	4.59 (4.63)	4.91 (5.97)	4.89 (5.69)	4.15 (5.32)	
8	**mouse**	5.90 (4.85)	8.82 (4.62)	20.44 (8.60)	22.98 (9.64)	14.51 (10.18)	12.42 (9.93)
	pen	5.21 (7.13)	6.64 (6.56)	16.71 (9.84)	12.77 (9.26)	10.33 (9.34)	
12	**mouse**	13.19 (6.37)	21.26 (8.79)	38.56 (14.98)	38.58 (12.06)	27.90 (15.45)	24.50 (16.26)
	pen	8.33 (8.33)	14.61 (14.91)	31.87 (14.13)	31.87 (14.13)	21.09 (16.48)	
Total	**mouse**	6.84 (6.84)	11.42 (9.51)	21.08 (17.32)	22.19 (16.52)	15.38 (14.69)	
	pen	5.24 (7.24)	8.62 (10.46)	17.83 (15.5)	15.74 (14.90)	11.86 (13.29)	
	mouse & pen	6.04 (7.04)	10.12 (10.02)	19.46 (16.24)	18.96 (15.95)		

Figure 5.3: Means table for percentage of errors. Standard deviation is shown in parentheses.

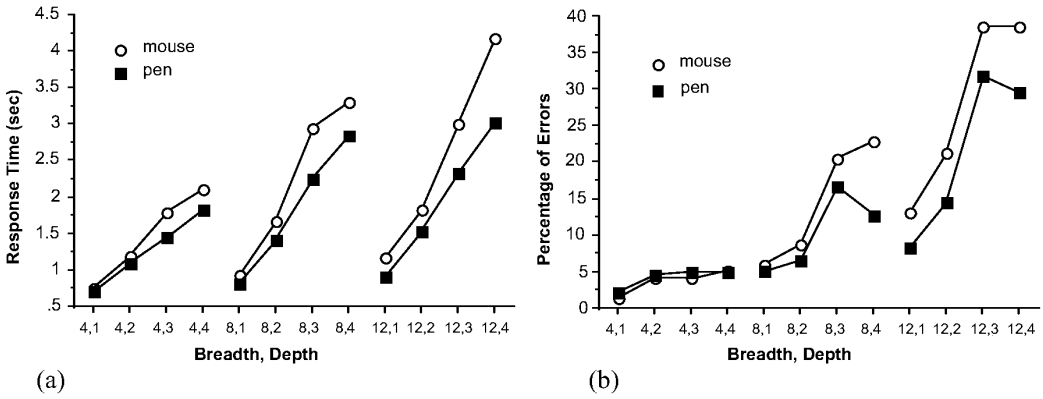


Figure 5.3: Response time and percentage of errors as a function of menu breadth, depth and input device. Each data point is the average of 288 trials.

All three factors, input device, breadth and depth affected response time. Analysis of variance revealed a three way interaction between input device, breadth, and depth ($F(6,66)=3.32$, $p < .05$) affecting response time. Figure 5.3 (a) shows these relationships. As one would expect, increasing breadth and depth increases

response time, however subjects' performance degraded more quickly with the mouse than with the pen.

Both depth and breadth interacted to affect error rate ($F(6,66)=12.28$, $p < .0001$). Variance in the error data is large, so the curves in Figure 5.3 (b) must be interpreted carefully. Individual comparisons of error means revealed no significant differences for breadth four at any depth. For breadth eight and twelve, the only significant change in error rate occurred between depth two and three ($F(1, 11) = 23.85$, $p < .001$; $F(1, 11) = 60.52$, $p < .0001$). This indicates that the "rolling off" of the errors curves for breadths eight and twelve between depths two and three is not statistically significant but the increase between depths two and three is significant.

It is important to compare these errors against what we believe would be reliable menu configurations. It seems reasonable that selection from 4,2 menus would be reliable since these marks can be recognized even if drawn very inaccurately. A comparison between 4,2 and 8,2 menus reveals no significant difference. Hence, we have no evidence to claim that eight-item menus, up to two levels deep are more unreliable than 4,2 menus.

A similar comparison between the 4,2 and 12,1 menu revealed a significant difference ($F(1, 11) = 8.25$, $p < .01$). However, the 12,1 menu was not significantly different from the 8,2 menu. Continuing the comparison, we found that the 12,2 menu was significantly different from the 8,2 menu ($F(1, 11) = 21.11$, $p < .0001$). Hence, we claim that the 12,1 menu borders on being unreliable. Section 5.2 has further interpretations on these results.

Hypothesis 5 (inaccuracies propagate) was shown to be true. As depth increased, the average depth at which errors occurred became significantly greater than half the depth of the hierarchy ($F(3,33)=7.62$, $p < .001$). However, the input device had an effect on this behavior. Figure 5.4 shows the pen consistently demonstrated this effect but the mouse exhibited a more erratic behavior.

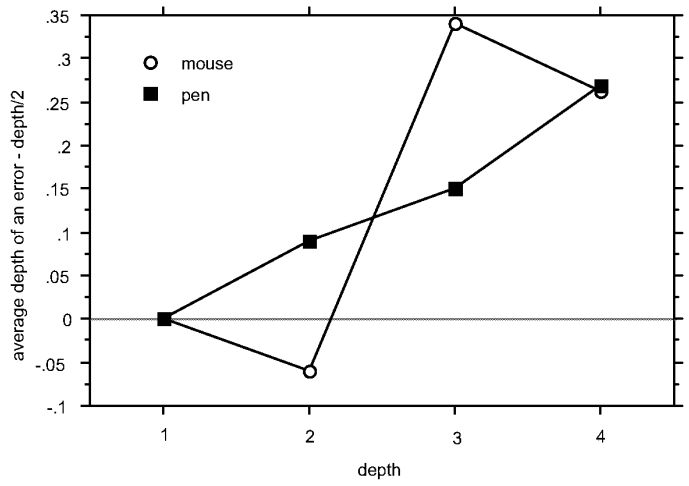


Figure 5.4: Average depth of error - depth/2 versus depth. Depth is the depth of the menu structure being selected from.

We tested the effects of mixing menu breadths in menu configurations by comparing the performance of a 12:12:12 menu with a 12:8:12 menu. We found no significant performance difference between the two menu structures. Therefore, we have no evidence that hypothesis 6 (mixing menus degrades performance) is true.

In order to test the hypothesis 7 (on-axis items enhance performance), targets for the 12,2, 12,3 and 12,4 menus were picked such that the experiment data could be divided into 3 groups. With each group we associated an "off-axis-level": a1, a2 and a3. Experimental data was placed in group a1 if the target consisted only of menu items that were on-axis, such as "12-3-9-3." Group a3 consisted of data on targets that consisted of entirely off-axis targets such as "1-2-1-2". Group a2 consisted of data on targets that were a mixture of on-axis and off-axis menu items, such as 12-7-3-9. Figure 5.5 shows that axis level had a significant effect on response time ($F(2,22)=104.84$, $p < .001$), and on percentage of errors ($F(2,22)=36.2$, $p < .001$). Figure 5.5 (a) shows how the type of device interacted with off-axis level ($F(2,22)=6.93$, $p < .05$). This indicates that subjects response time using the pen did not degrade as much as their response time with the mouse on the worse off-axis targets.

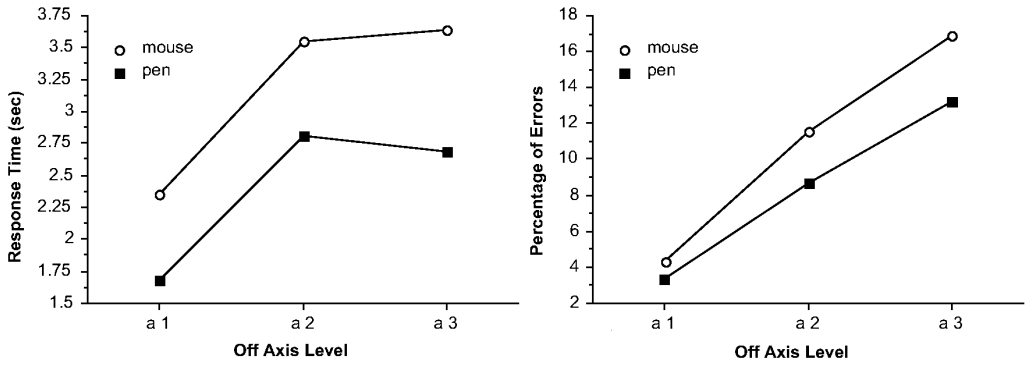


Figure 5.5: Average response time and percentage of errors for targets with an increasing number of "off-axis" items.

In order to evaluate hypothesis 8 (drawing direction affects performance), targets were picked for 4,3, 4,4, 8,3, and 8,4 menus such that mirror image pairs of targets could be compared. For example, the target N-W-N-W was compared with the target N-E-N-E. No significant different in response time was found between "left" and "right" direction targets. Therefore this experiment provides no evidence that hypothesis 8 is true.

The data was analyzed for learning effects by examining performance after every sixth trial. Figure 5.6 shows the results. Response time dropped over 24 trials ($F(3,33)=59.227$, $p<.0001$). Percentage of errors dropped as well ($F(3,33)=8.294$, $p<.0003$). This shows that not only were subjects getting faster but also producing fewer errors. No significant performance differences were found between trial 18 and trial 24. It may be possible that, because subjects were only selecting from three targets, their performance was beginning to asymptote by trial 24.

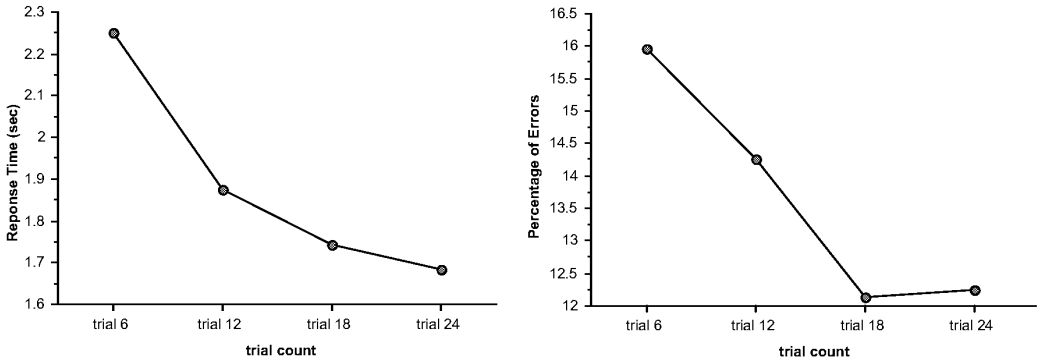


Figure 5.6: Average response time and percentage of errors after every sixth trial.

We analyzed the data for the number of pauses that occurred as a selection was being drawn. A pause was defined as the pen or mouse not moving more than five pixels for more than 1/2 of a second. Figure 5.7 shows that, as users gained experience the number of pauses dropped ($F(9,99)=38.409$, $p < .0001$). This is evidence that subjects, with experience, began to draw a mark not as a series of discrete selections but as a single mark of a certain pattern (assuming that when pauses did occur they occurred between different selections). The number of pauses did not fall all the way to zero because some of the most difficult targets required careful drawing which resulted in pauses. Given these results we accept hypothesis 9 (subjects will chunk).

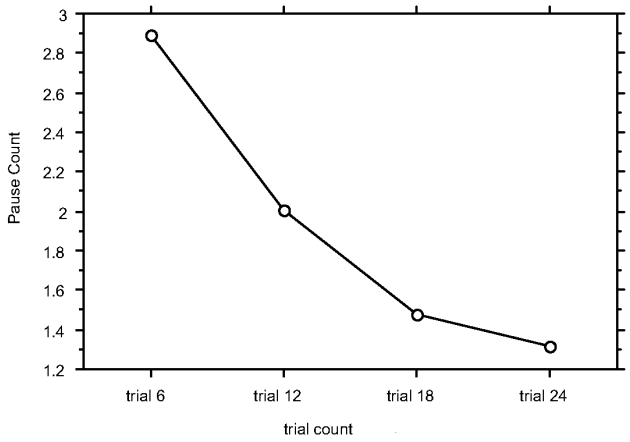


Figure 5.7: Average number of pauses counted after every six trials.

We also gave subjects a questionnaire after the experiment. This was to elucidate subjects' perception of their own performance and compare some of the experimental data with subjects' perceptions.

Eleven out the twelve subjects thought making selections with the pen was faster and more accurate than with the mouse. This agrees with the data from the experiment. Also, when asked to comment on the experiment, four subjects reported that, although their performance with the mouse was fast, they found the mouse required more effort.

We wanted subjects' opinion on the accuracy of our mark recognizer. In some cases, for example, menus which are only one level deep, recognition is simple. In this case, only the start and end points of the mark need to be examined to determine the item picked. However, at depths greater than one, submenu selections must be determined so changes in direction along a mark must be recognized. The algorithm for determining these "kinks" along a mark is complex because it has to handle dense menus and marks that are drawn sloppily. Typical of most recognition systems, occasionally what appears to the subject as a correct mark is misinterpreted by the system. However, on average, subjects claim that this happened only three percent of the time. This is an acceptable recognition rate by mark recognizer standards (Sibert, Buffa, Crane, Doster, Rhyne, & Ward, 1987). Nonetheless, after observing the type of recognition errors that occurred during the experiment, we believe the recognition rate can be further improved by a few refinements to the recognizer algorithm.

Another phenomenon that occurred in the experiment was subjects selecting the wrong direction by accident. For example, the screen would display "select N" and the subject would select south. Errors of this type are referred to as "mental slips" (Norman, 1981). These types of errors were removed from the data set before analysis because they are not caused by drawing inaccuracies. Other errors such as clear cut errors on part of the recognizer were also removed from the data. Subjects reported several causes for mental slips: "I just goofed" or "I started to draw the mark from the previous trial". Subjects, on average, claimed that mental errors occurred two percent of the time. This approximately agrees with the data: we found a one percent error rate for clear cut "mental slips". We do not feel these

errors are particular to drawing marks—mental slips are common in any human activity (Norman, 1981).

We hypothesized before the experiment that drawing marks that were predominately left to right movements would be easier, and hence faster, than right to left marks. However, our analysis of the data showed drawing direction had no significant effect on selection times. This agrees with the results of the questionnaire: six out of the twelve subjects thought left to right marks were easier to draw that right to left marks. This even split among subjects perhaps explains the non-significant effect of direction. A closer examination of the data might reveal individual effects.

5.3. CONCLUSIONS

We can now revisit the questions posed at the start of this chapter and interpret the results of this experiment.

Q1: Can users use hierarchic marks? Even if using a mark to access an item is too hard to draw or cannot be remembered, a user can perform a selection by displaying the menus. Nevertheless, since the subjects could perform some of the marks in the experiment with acceptable response times and error rates, marking is a usable method of selection.

Q2: How deep can one go using a mark? Our data indicates that increasing depth increases response time linearly. The limiting factor appears to be error rate. Error rate was found to rise significantly for menus beyond the 8,2 menu. 8,2 menus were not any more unreliable than 4,2 menus. Common sense tell us that the marks required to select from a 4,2 menus are not difficult to draw. Hence we consider menu configurations which did not significantly differ in error rate from 4,2 menus to be reliable. It seems reasonable to recommend using menus of breadth four, up to depth four, and menus of breadth eight, up to depth two. 12,1 menus border on unreliability.

Off-axis analysis indicates that the source of poor performance at higher breadths and depths is due to selecting off-axis items. Thus, when designing a wide and deep menu, the frequently used items should be placed at on-axis marks. This would

allow some items to be accessed quickly and reliably with marks, despite the breadth and depth of the menu.

What is an acceptable error rate? The answer to this question depends on the consequences of an error, the cost of undoing an error or redoing the command, and the attitude of the user. For example, there is data that indicates, in certain situations, experts produce more errors than novices (Sellen, Kurtenbach, & Buxton, 1990). The experts were skilled at error recovery and thus elected to sacrifice accuracy for fast task performance. Our experiences with marking menus with six items in a real application indicate that experts perceived selection to be error-free. Other research reports that radial menus with up to eight items produce acceptable performance (Hopkins, 1991). Marking menus present a classic time versus accuracy tradeoff. If the marking error rate is too high, a user can always use the slower but more accurate method of popping up the menus to make a selection.

Marking error rates can be compared to linear menu error rates but one must be very cautious when comparing results from different experiments and different interaction techniques. Even within the same experiment, subjects may not consistently perform at the same level of accuracy, or the experimental task may artificially inflate or deflate the error rate. We can, however, make some approximate comparisons. In a study of selection performance using pop-up hierarchic linear six-item menus of depth two, Nilsen (1991) reports error rates of 2.3%. Nilsen also reports that subjects accidentally popped up the wrong submenu on their way to making a correct selection 6.3% of the time. In another study of similar pop-up linear menus, Walker, Smelcer, & Nilsen (1991) report error rates that range from 2.0% to 12.6% for subjects selecting from nine-item menus of depth 2. These error rate figures are in the range of the error rates found in our experiment for menus of up 8,2 menus. Therefore, we can conclude, with caution, that marks, within the limits discussed above, can be as accurate as selection from linear menus. It is also critical to note that this level of accuracy is not the expense of speed. For example, in this experiment selection from 8,2 menus required on average 1.5 seconds. In Nilsen's experiment, selection from six-item linear menus of depth 2 required on average 1.8 seconds (six-item menus should be faster). We found that, comparing the data from Nilsen and Walker experiments with this experiment, for equivalent menu configurations, selection from linear menus is slower than selection using marks.

Q3: Is breadth better than depth? For menu structures that resulted in acceptable performance, breadth and depth seems to be an even tradeoff in terms of response time and errors. For example, accessing 64 items using 4,3 menus, is approximately as fast as using 8,2 menus. Both have approximately equivalent error rates. Thus, within this range of menu configurations, a designer can let the semantics of menu items dictate whether menus should be narrow and deep, or wide and shallow.

Q4: Will mixing menu breadths result in poorer performance? The experiment did not show this to be true. One possible explanation is that our menu labels strongly suggested the correct angle to draw and thus eliminated confusion. A stronger test would use less suggestive labels when mixing breadths. Our results do indicate that, when there is enough familiarity with the menus, mixing breadths is not a significant problem.

Q5: Will the pen be better than the mouse for marking menu marks? Overall, the pen proved to be more suitable. However, for small menu breadths and depths, the mouse produced approximately equivalent performance. We found this extremely encouraging because it implies that marking menus are an interaction technique that not only takes advantage of the pen but also remains compatible with the mouse. Of course, it is worthwhile to note that some subjects thought their performance with the mouse was just as good as with the pen, but that the mouse required more effort to attain this level of performance.

These conclusions should be tempered by reminding the reader that this experiment simulated an expert situation (i.e., subjects were asked only to select from three different targets, thus they quickly became "expert" at those targets). We have argued that this situation is reasonably realistic. Other realistic situations, such as the performance of users on unfamiliar hierarchic marking menus with varying targets, has yet to be explored.

5.4. SUMMARY

The chapter described an experiment to test the limitations of using marks to select from hierarchic marking menus. Subjects were asked to select from marking menus using marks only. Menus were chosen such that the subject would very quickly learn and remember the mark required to perform a given selection. The breadth and depth of these menus and the input device was then systematically varied to

elucidate the effects of these variables. Subjects' time to perform selections and error rates were collected and analyzed. Subject's perceptions were collected using a questionnaire.

The experiment revealed that error rate was the limiting factor. Menus of breadth 4, 8 and 12 were examined. Error rate became a factor when menu breadth was eight or twelve. For these breadths of menu, error rate rose significantly when depth was greater than two. For these menu structures with acceptable error rates, there appeared to be an even depth/breadth tradeoff. When menus structures contained equivalent numbers of items, subjects showed equivalent performance on both narrow, deep menus and wide, shallow menus. It was also discovered that mixing menus of different breadths in a menu structure did not adversely affect performance. Finally, we concluded that the pen is more suitable for drawing marking menu marks than the mouse, but the difference is not large.

This chapter has answered some basic questions about the design variables of hierarchic marking menus. Specifically, how deep and wide can menu structures be yet still allow a user to perform selections using marks? The following chapter takes the answers to these questions and applies them to the design of hierarchic marking menus in a pen-based application.

Chapter 6: Generalizing the concepts of marking menus

6.1. INTRODUCTION

This chapter reports on a design experiment which deals with applying the design principles of self-revelation, guidance, and rehearsal to interface design. Two issues are explored. First, we examine the ramifications of integrating an interaction technique that is based on these principles (marking menus) into a pen-based interface. We found that it is possible to integrate marking menus into an interface but several compromises needed to be made. Although these compromises change the original design of marking menus, we show that the resulting design still obeys our three design principles. Second, we examine how these design principles can be applied to other types of marks besides zig-zag marks. With this goal in mind, we developed an interaction technique that provides self-revelation, guidance, and rehearsal for these other types of marks. These experiences provide a better understanding of the role of marking menus in interface design and demonstrate the value of the design principles.

The test bed for this design experiment was a pen-based electronic whiteboard application called *Tivoli* (Pederson, McCall, Moran, & Halasz, 1993). Tivoli is intended to be used in collaborative meeting situations, much in the same way that a traditional whiteboard is used. Tivoli runs on a large vertical display, called *Liveboard* (see Figure 6.1) (Elrod et. al., 1992), that can be written on with an electronic pen (see Figure 6.2). Much like a whiteboard, several people can stand in front of a Liveboard and write, erase, gesture at, and discuss hand drawn items.

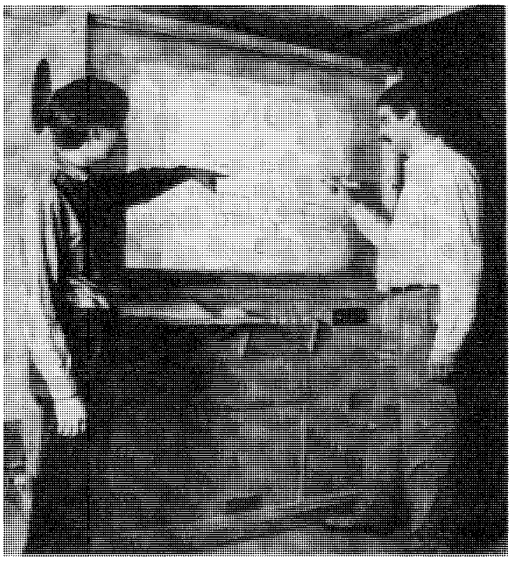


Figure 6.1: The Liveboard in use.

Tivoli, however, does more than just emulate marking on a whiteboard. Marks can be edited, stored, and retrieved. Marks are remembered by Tivoli in terms of *strokes*. A stroke is the path of the pen recorded from the moment the pen is pressed against the screen and moved, until it is released from the screen. A screenfull of strokes can be grouped into a "slide", and saved for retrieval later. Typical operations on strokes include moving or copying groups of strokes, changing the color or thickness of the pen tip, and undoing edit operations. Users draw "edit marks" to perform some of the editing operation described above. Figure 6.3 shows the types of marks used. Other operations are triggered using graphical buttons, dialog boxes, and menus.

One basic goal of our design study was to address the problem of operating extremely large displays. It is envisioned that someday the *Liveboard* display surface would be very large, and therefore, we wanted to address the problem of bringing the commands to the user as opposed to the user moving to the commands. Marking menus seemed suitable for this type of design since the menus can pop up at any location and the marks can be made at any location.[20] Furthermore, since

---

[20] This is not completely true. Depending on the design of the interface a user may have to be over some particular area or object on the display before a menu can be popped-up or a marking intepreted. However, the point is that pop-up menus and marks help reduce the amount of movement a user must make to invoke functions. For example, when a user wants to change pen color, traditionally one has to move from the drawing

Tivoli has many commands, we felt that hierarchical marking menus might allow access to many of these commands from a single location. The issue was whether or not we could integrate marking menus into the existing Tivoli interface design to solve some of these problems.

Another basic design goal was that Tivoli should be based on the unfolding interface paradigm described in Chapter 2. For example, for a novice Tivoli user the interface presents a limited set of functions—the type of functions one gets with an ordinary whiteboard. However, additional functions can be discovered and used with minimal instruction and experience. In effect, once a user has the "key" to unlock the hidden functionality, Tivoli can be unfolded and additional functions invoked. Using edit marks is a way to hide additional functions. The edit marks are not in themselves self-revealing, and therefore, this serves as a way to hide functions from a novice.

---

area to a color pallet and back. With a pop-up menu, this trip is avoided since the menu can be popped-up over some white space in the drawing area.

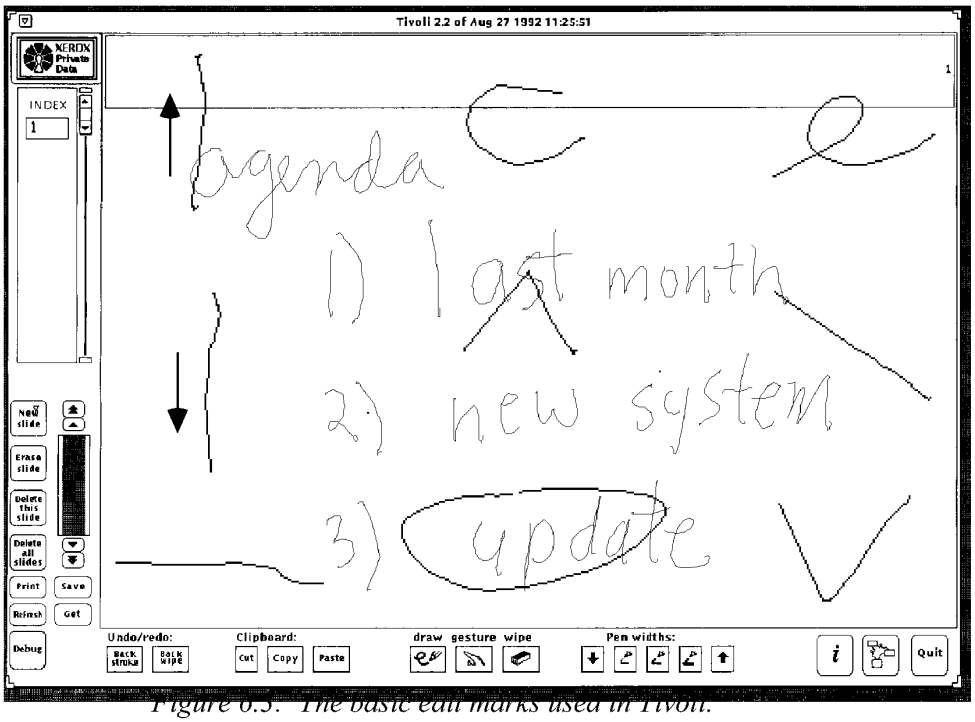


Figure 6.3: The basic edit marks used in Tivoli.

Figure 6.2: An application called Tivoli, running on Liveboard, emulates a whiteboard but also allows drawings to be edited, saved and restored.

Given these basic goals, we explored two problems. The first problem was to determine which Tivoli functions would be suitable for marking menus, and how marking menus could be integrated into the existing interface. The second problem was how to provide self-revelation, guidance, and rehearsal for the edit marks in Tivoli.

6.2. INTEGRATING MARKING MENUS INTO A PEN-BASED INTERFACE

We decided we would explore design issues by using marking menus to control pen settings in Tivoli. In Tivoli, the pen can be set to different colors and thicknesses. Originally, these settings were performed using a pallet of buttons which had an individual button for each pen thickness and color. There were several reasons why it would be advantageous to control these functions using marking menus. First, the original buttons consumed a large amount of screen space. Replacing these buttons with a marking menu would free up this screen space. Second, changing pen

settings was a frequent operation while drawing. Changing settings meant many trips to and from the button pallet. A marking menu could be made to pop up at the drawing location, thus avoiding trips to the button pallet. Third, no intuitive set of marks exist for controlling pen settings. Marking menus could provide a set of marks and a method for learning those marks.

6.2.1. Adapting to drawing and editing modes

Figure 6.4 shows the marking menu we used to control pen thickness and color. The range of items is deliberately small. We felt that, in Tivoli, users need only a few different thicknesses and colors for the pen. This is like real whiteboards, where the number of markers is limited. The menu items "inc" and "dec" allow a user to increment and decrement the pen size to get custom thicknesses. The menu appears when a user presses-and-waits with the pen anywhere in the drawing area. This allows a user to change pen settings without having to move the pen from the current drawing location.

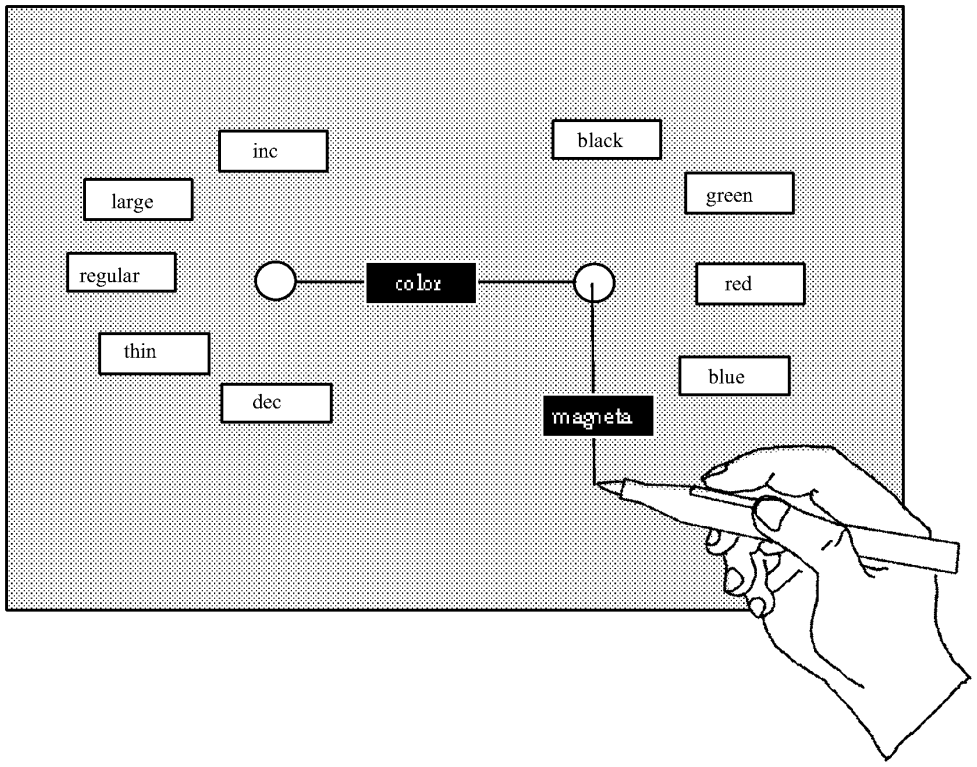


Figure 6.4: The hierarchical marking menu used to control pen settings in Tivoli. The menu can be popped up by pressing-and-waiting instead of drawing.

There is a complication with this design. Normally, a marking menu allows a user the alternative of drawing a mark to select a menu item. However, in the situation just described, Tivoli is in the "drawing mode" (i.e., all marks are interpreted as drawings, not commands). A mark is interpreted as a command in Tivoli when it is drawn while a button on the pen (the command button) is pressed. Thus, the design of the Tivoli's interface requires that menu selection marks (which are actually command marks) be performed with the command button pressed. However, this deviates from the rehearsal principle slightly: the physical action of making a selection mark is the same as selecting from the menu, but the command button must also be pressed. All the directional motions remain the same so we can be hopeful that using the menu still develops skills useful for learning and making the selection marks.

We have no empirical data to verify that, despite this deviation in rehearsal, skills developed in using the menu are still transferred to using the marks. However,

when using Tivoli ourselves, because of our experiences with the menu, we were able to recall the spatial layout of the menu, and issue marks. The role of spatial memory and physical movement memory in the transition from menus to marks is a topic for future research.

6.2.2. Avoiding ambiguity

Typically interfaces that use marks as commands identify marks by the shape of the mark or the context in which the mark is made. This discussion discriminates between marks intended for marking menu selections and other kinds of marks intended for commands. For the sake of brevity in this discussion, we will refer to these other kinds of marks as *iconic marks*, although the meanings of these marks may not be strictly based on iconic shape (see section 6.3.1 for further discussion). Also for the sake of brevity, we refer to marking menu's zig-zag marks as *menu marks*. The important point is that the potential exists for marking menu marks (menu marks) to be confused with iconic marks. Figure 6.5 shows an example of two marks for a menu structure of breadth eight and depth of two which are the same as some of the iconic marks in an early version of Tivoli.

These types of ambiguities are not peculiar to marking menu marks. Many interfaces that use marks exhibit this problem. For example, a classic problem is drawing an "O" for the letter "O" and having it confused with a small circle (where circling performs a selection). We present three strategies for overcoming this problem for marking menus, and the advantages and disadvantages of each one. We then describe how a one of these three strategies was used in Tivoli.

Avoidance

One way to avoid ambiguities between marking menu marks and iconic marks is to eliminate the ambiguous marks from the marking menu set. This can be done by avoiding the placement of menu items at locations in a menu structure that would result in ambiguous marks. These "avoided locations" can be occupied by null menu items. A mark that selects a series of null items is then considered no longer a marking menu mark, and therefore ambiguity is eliminated.

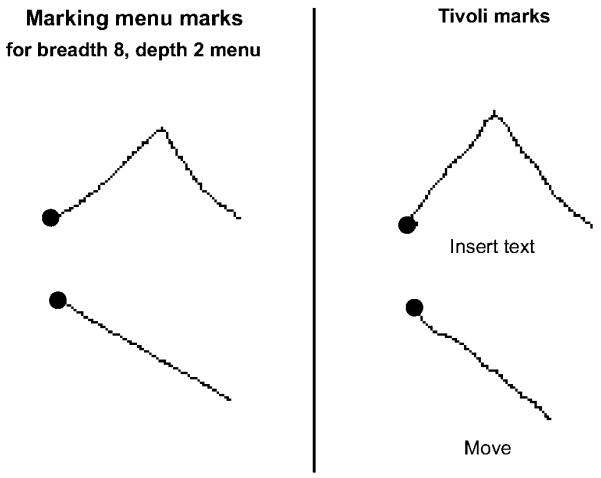


Figure 6.5: The marks used for a marking menu may conflict with other marks. The example shows two marks used for selecting from a marking menu that can be confused with edit marks in an early version of Tivoli. A dot indicates the starting point of the mark.

One drawback to this approach is that the number of items a menu can hold is reduced and "unnatural" gaps may appear in the menus. For example, suppose a menu contains an ordered set of font sizes. If one of the menu items is not used, then a gap appears between two menu items that logically should appear adjacent to one another. This may make learning the layout of the menus more difficult.

Another drawback is that eliminating a menu item from certain location forces that the item to be placed somewhere else. Menu structures can be expanded to hold displaced items either in breadth or in depth. As shown in Chapter 5, expanding in breadth or depth slow menu selection and increases errors. Furthermore, eliminating items may result in losing on-axis items, which have been shown in Chapter 5 to enhance performance. Ultimately, rearranging menus may lead to menus that appear to be oddly structured, and this results in menus that are hard to learn, slow to use and error-prone.

These drawbacks makes avoidance a poor solution. In certain restricted cases, though, it can be a simple and easy solution to implement. For example, suppose the only conflicting mark is a horizontal stroke which is to the right, and the marking menu only needs to contain six items. The simple solution is to use an

eight-item menu and the make the "right stroke" menu item and some other menu item null items, and populate the remaining menu items with the six commands. A variation on this strategy is to change the iconic marks. This, of course, avoids the problems with modifying a marking menu as described earlier, but in certain situations may cause confusion for a user when obvious or common marks are replaced by non-obvious, uncommon iconic marks.

Different context

Another design alternative is to allow iconic marks and marking menu marks of the same shape to coexist but determine their meaning by the context in which they are drawn. Two dimensions in which the context can vary are time (i.e., when the system is in a certain mode a mark has a certain meaning), and by space (i.e., a mark's meaning varies depending on the location at which it is drawn).

Distinguishing the meaning of a mark by the context of time leads to moded interfaces. An interface where a user must enter a "marking menu mode" to issue a marking menu mark seems to defeat the purpose of making a mark—a fast way to invoke a particular command. However, if the cost of switching modes is very low and properly designed (Sellen, Kurtenbach, & Buxton, 1992), this can be an effective technique. An example of low cost mode switching is a dedicated pop-up menu button on the mouse which is found in many windowing systems such as *X11* (X11, 1988) and *Open Look* (Hoeber, 1988). After developing the habit of holding down the button to pop-up and maintain a menu, a user no longer perceives using a menu as a mode. One can imagine such a similar design for marking menus where a user presses down a button on the pen or mouse to indicate to the system that the mark is intended for the marking menu. The obvious disadvantage to this scheme is that a hardware button must be dedicated strictly to a menu. Many pen-based system pens do not have buttons, or the buttons have already been assigned other functions. For example, in Tivoli the two buttons on the pen were already used for other functions. The first button is used to distinguish between drawing mode and command (edit mark) mode. The second button is used to control whether the pen is in drawing mode or erasing mode.

Another type of context that can be used to distinguish the meaning of a mark is location. For example, a stroke through a word may mean "delete the word" while the same stroke starting on a graphic may mean "move the graphic". This type of

scheme works well with object oriented direct manipulation systems, where the combination of an object and a mark can be used to distinguish a mark's meaning. Of course, distinguishing meaning by location will not work if the same location must accept two identically shaped marks.

Marking menus work very well in identification by location situations. For example, on a different project, we found an effective interaction technique can be created by embedding a marking menu in an ordinary graphical button. In effect, this extends the functionality of the button. Along these lines, we developed a system called *HyperMark* which allows marks to be used in Apple's Hypercard (Apple 1992). For example, if HyperMarks are added to a button, not only does a button react to a mouse press, but marks can also be drawn on the button which trigger other actions. This results in the interface having fewer buttons and faster interactions in some cases. In effect, HyperMarks are similar to pop-up menus where additional functions are hidden under a button until popped up. However, with HyperMarks, a user does not have to wait for a menu to pop up, visually search the menu and point to an item. Instead, a mark triggers the item directly. Our intention was to permit ordinary Hypercard users or programmers to incorporate marks into their own Hypercard stacks.

With HyperMark, different buttons accept the same mark but the interpretation of the mark is different. Figure 6.6 shows an example of different locations having different menus but reusing the same set of marks. The meaning of the marks is disambiguated by the location of the mark. We feel this is a reasonable design as long as the common commands (scroll up and scroll down, for example) are kept consistent from button to button.

The disadvantages of discriminating by location are, first, it does not eliminate the problem if the same location accepts two ambiguous commands and second, it consumes screen space. Consumption of screen space results in situations where the desired location is not displayed on the screen and must be acquired by the user. This can slow interactions and defeat the purpose of using marks.

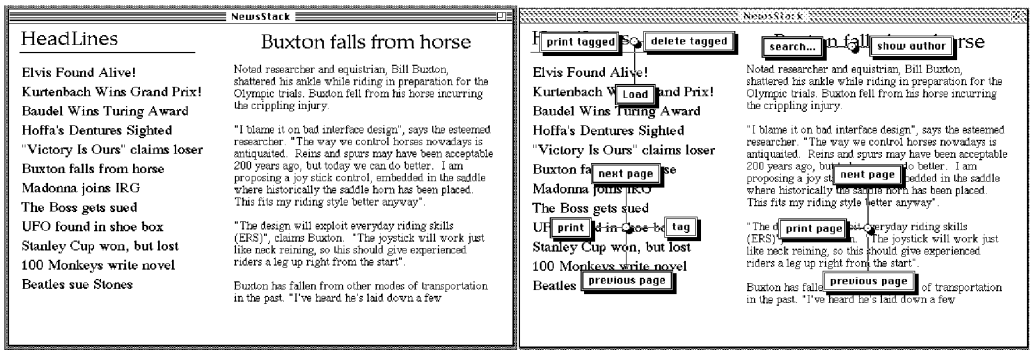


Figure 6.6: A simple news reader program in Hypercard that is controlled by marking menus. (a) shows the four major area of the screen: "Headlines", a list of articles, the title of the current article, and text of the article. (b) shows the marking menus associated with each of these areas. When marks are used to select from the menus the context (the location) of the mark contributes to its meaning.

Distinguishing tokens

Distinguishing marks by tokens involves augmenting a mark with some characteristic that disambiguates it. Augmentation can be of several forms. The shape of marks can be augmented. Alternatively, the dynamics of drawing the mark can be used to augment a mark.

Figure 6.7 shows how an augmenting "dot" at the start of a marking menu mark is used to indicate the mark is intended for a marking menu. An augmenting token, however, does not have to be at the start of the mark. The token could appear as a prefix to the mark, within the mark or as a suffix to the mark. However, if the mark is not distinguished from the start, then mark-confirmation may lead to ambiguities, since the system may identify the partially completed mark as both the start of a marking menu mark and an iconic mark.

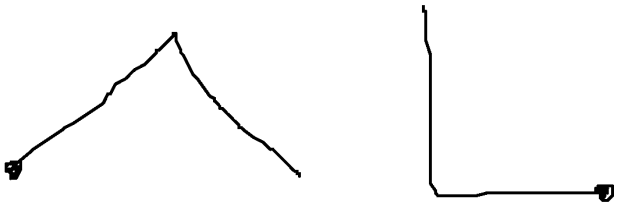


Figure 6.7: Two marking menu marks that are augmented by a "dot" to distinguish them from other types of marks in an interface.

There are many alternatives to "dot". Any sort of token that guarantees distinction could be used. In practice, we found "dot" easy to draw and easily and reliably recognized by the system.[21] We also found that one could make an analogy between it and press-and-wait. In Tivoli, pressing-and-waiting in drawing mode popped up a marking menu to change pen settings. "Dot" could be thought of as a mark in command mode that mimicked press-and-wait, and allowed access to the pen setting menu.

Another way of distinguishing marks is by dynamics. For example, in some systems the speed at which a mark is drawn determines its meaning. For example, a slow up-stroke may mean "next page", while a quick up-stroke (a "flick-up") may mean "go to the end of the document" (Go, 1991). In Tivoli, we experimented with dynamic schemes and found several problems. First, flicks are not consistently recognized because the speed of a flick varied with direction and the user's dexterity. Also, quick movements sometimes caused the pen to skip off the display surface before the speed of a flick could be attained. Flicking was not very reliable because of these problems. We also experimented with prefix flicks and suffix flicks. Prefix flicks made drawing the remaining mark too hard: slowing the pen down after drawing the flick to draw the rest of the mark, was difficult. Alternatively, drawing the entire mark at flick speed was too hard. Suffix flicks were more reliable: we could safely draw the first part of the mark then add a "flick flourish" on the end of the mark to indicate it as a marking menu mark.

---

[21] We occasionally operated Tivoli with a mouse, although it is intended to be operated with a pen. In this case we found a "dot" very difficult to draw. Thus we would not recommend the use of the "dot" for mouse driven system that use markings.

Recognizing flicks was further complicated by limitations in the input event software. On occasion, input events are buffered. Time stamping of input events occurs after events are read from the input buffers and therefore, at times, these buffering delays confuse the flick recognition process. This problem could be overcome by immediately time stamping all events. Nevertheless, this indicates that tracking dynamics place special demands on input software.

Even if flicks could be made reliable they still present a problem: how can flicks be demonstrated to a user? The "dot" is easy to learn because a user can simply be told: "make a dot, about this big". Flicks on the other hand are dynamic in nature and are best learned by demonstration and practice. Section 6.3.2 discusses issues concerning self-revelation of mark dynamics.

To summarize, we have presented three strategies to avoid ambiguity between menu marks and iconic marks: avoidance, different context, and distinguishing tokens. Based on the various advantages and disadvantages each strategy just discussed, we elected to use a distinguishing tokens strategy in Tivoli. Specifically we used the "dot" prefix mark shown in Figure 6.7. Section 6.4 discusses our experiences with this strategy.

6.2.3. Dealing with screen limits

One problem that can occur in a pop-up menu system is that, when a menu is displayed near the edge of a screen, some portion of the menu may be clipped-off. This may make it impossible to see or select some items. We refer to this as the *screen limit* problem. Marking menus suffer from this problem because they use pop-up menus.

One possible solution to the screen limit problem is not to allow menus to be displayed too close to the edge of the screen. This implies placing menu "pop-up spots" some safe distance away from the edge of the screen. While this is a workable solution, it is not practical when menu hierarchies are deep, since pop-up spots may have to be located a large distance from the edge of the screen to keep the submenus from hitting the edge of the screen. Furthermore, it seems to be an unreasonable constraint given popular interface design. For example, most drawing programs have scrollable windows, and a user is allowed to scroll a window till menu pop-up spots are close to the edge of the screen.

Another solution to the screen limit problem is *constraining*., Most pop-up menu systems constrain menus to display entirely on the screen, even if the location from which the menu was invoked would cause some portion of it to be clipped-off. For example, the menus in Open Look use this solution (Hoeber, 1988). Constraining, however, causes problems when hierarchic menus are used. In this case, accessing a series of menus causes each menu to hit the edge of the screen. We refer to this problem as *crowding*. When crowding occurs, users end up making a series of selections from menus that are against the screen edge and this can sometimes make menu selection slow and error-prone.

Hopkins (1991) uses a constraining solution for radial menus. Since marking menus use radial menus, it is worthwhile to consider this solution. With Hopkins' radial menus (or pie menus), normally, a pie menu pops up centered around the cursor location. However, when the cursor is close to the edge of the screen, this results in some portion of the menu being clipped-off, To overcome this problem the menu is displayed not centered around the cursor, but shifted over so it is completely displayed. The cursor is then reset by the system to the center of the menu (this is referred to as "warping" the cursor). At this point, the user can make a selection in the usual way.

Problems occur with Hopkins' solution when the input device is an absolute device like the pen, and this makes it unsuitable for marking menus in Tivoli. The problem is that the system cannot change the location of pen (given the constraint that the cursor always appears under the tip of the pen). An example demonstrates this. Suppose a radial menu is popped up too close to the edge of the screen. If the menu is constrained to display completely on the screen, the pen tip is no longer in the center of the menu. The pen tip generally ends up located in one of the menu items. This immediately highlights the item. If the highlighted menu has a submenu, this menu would then be displayed. Thus, a user inadvertently descends the menu hierarchy. Even if the menu item has no submenu the user would still have to move the pen out of the menu item if the menu item was not the desired one.

We propose the following solution which permits marking menu selections near the edge of the screen when using a pen. When the pen is pressed close to the edge of the screen, the marking menu appears centered around the pen tip cursor with some portion of it clipped-off. If the clipped-off portion is large enough to obscure some menu items, another special menu item (referred to as the "pull-out" menu item)

appears on the screen (see Figure 6.8). At this point a user can select the visible menu items in the normal fashion. However, if the user moves the cursor to the pull-out menu item, the menu is redisplayed centered at the location of the pull-out item. The pull-out menu item is located far enough away from the edge of the screen so that the menu is completely visible when redisplayed. At this point the pen is located in the center of the menu and all items are accessible. This same scheme works with hierarchic menus. Every time a submenu hits the edge of the screen, a pull-out item is displayed.

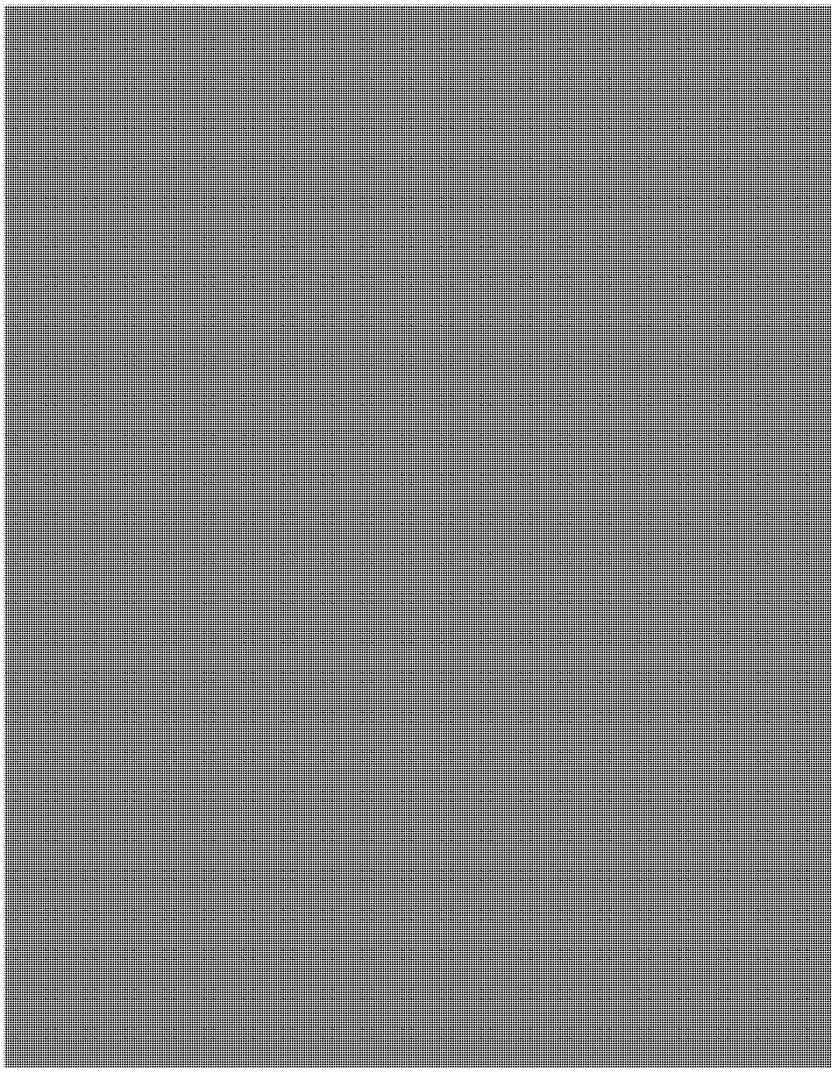


Figure 6.8: A "pull-out" menu item allows a user to access menu items that would be clipped-off by the edge of the screen. In (a) a user has displayed a marking menu but a portion of it is clipped-off by the edge of the screen. Because of this, a pull-out item appears (the gray circle). In (b) when the user drags over to the pull-out item, the menu is redisplayed so all items can be accessed.

Marks also have a screen limit problem. If one starts a mark too close to the edge of the screen one may run into the edge. As with menu mode, the input device used makes an important difference in a solution to the problem.

If a relative input device like the mouse is used, it is possible for users to draw marks "beyond" the edge of the screen. Hopkins (1991) has proposed a solution that

is suitable for marks. Hopkins' pie menus use a technique called mousing-ahead which is similar to marking but the path of the cursor leaves no ink-trail (see Section 2.3.1 for a complete explanation of mousing-ahead). Mousing-ahead is possible even when the cursor hits the edge of the screen. Although the cursor is constrained to the area of the screen, mouse movement after the cursor hits the edge of the screen is still tracked. Thus, a user can mouse-ahead beyond the edge of the screen. Applying this solution to marks, a user could draw marks beyond the edge of the screen, although some portion of the mark would not be visible. This solution is important because it preserves the principle of rehearsal. The movement to select from the menu must be the same as movement to make a mark and this happens even when menus and marks hit the edge of screen.

If the input device is a pen, drawing a mark close to the edge of the screen behaves logically: if the mark does not hit the edge of the screen, it can be performed as usual; if the mark does hit the edge of the screen, a user cannot physically draw it. This behavior mimics the way pen and paper works—if one is too close to the edge of the page one cannot draw certain marks.

We still need to, however, be able to apply marks to objects that are near the edge of screen. To do this we mimic pen and paper traditions. Generally, when something is too close to the edge the page to fit, a line is drawn from the object, out to some clear space and then an annotation is made. We propose a similar design. Suppose an object is too close to the edge of the screen for a certain mark to be made. A user can draw a line, out to some clear space on the screen, then make a "pull-out" mark, followed by the desired mark. Figure 6.9 shows this.

6.3. APPLYING THE PRINCIPLES TO ICONIC MARKINGS

Marking menus provide self-revelation, guidance, and rehearsal for "zig-zag" types of marks, specifically, the type of marks that are the byproducts of selecting from radial menus. Can a similar mechanism be provided for iconic marks? As a design experiment we decided to see if we could design mechanisms similar to marking menus but for the edit marks in Tivoli. Thus we attempted to design ways to self-reveal these marks, guide a user in making them, and have this be a rehearsal which builds skills for expert behavior.

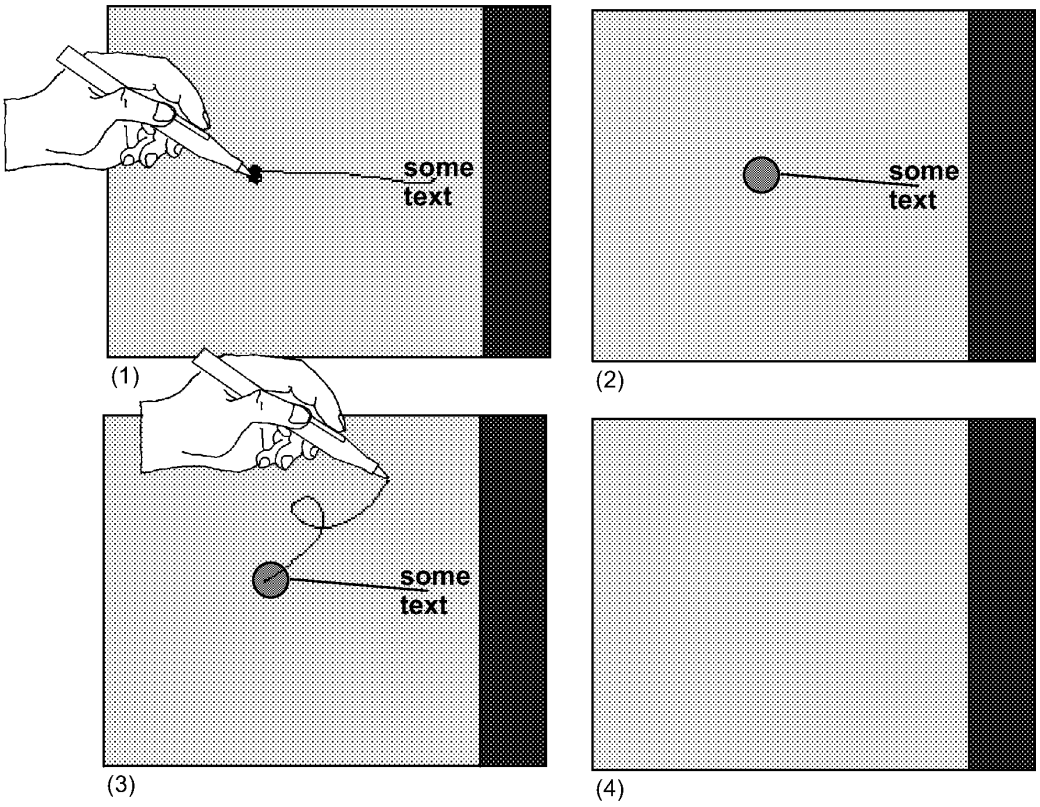


Figure 6.9: Using a "pull-out" mark to apply a mark to an object close to the screen edge. In (1), the pull-out mark is drawn (a line followed by a dot). In (2), the system has turned the mark into a pull-out object. A mark is then drawn in the pull-out object, in (3). In (4), the mark is applied to the object that is "pulled out", and it is deleted.

Another design goal was ease of programming. One of the attractions of marking menus is that an interface programmer can implement interactions which provide self-revelation, guidance, and rehearsal with something as simple as a pop-up menu subroutine call. We wanted a mechanism for iconic marks that was just as convenient to program. The idea was to avoid creating custom code to self-reveal each different type of mark.

6.3.1. Problems with the marking menu approach

Overlap

Suppose we strictly applied the marking menu design to the marks shown in Figure 6.3. In other words, display all the possible marks a user could make starting from a

certain location. Figure 6.10 shows the result of this approach. Marks overlap and can cause confusion. Part of the problem is that iconic marks are not suitable for displaying in this manner. Menu marks, however, are suitable because of their directional nature. Another problem in the example is that each entire mark is displayed. If all the marks of a hierarchical marking menu were displayed, this too, would result in overlap.

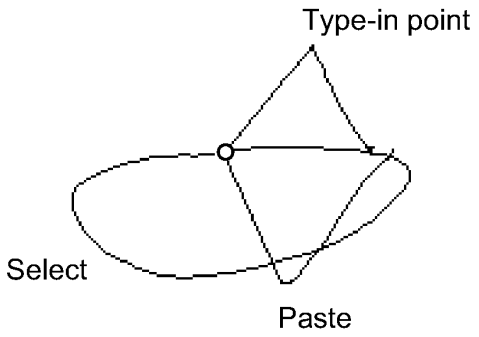


Figure 6.10: Overlap causes confusion when using the marking menu approach to self-reveal other types of marks. Here we display the commands available when starting a mark from a clear spot in the drawing region of Tivoli.

Not enough information

A display like Figure 6.10 gives little contextual information. For example, the important thing about the "Select" mark is that it should encircle objects and the shape of the circle can vary. This type of information is not shown in Figure 6.10.

The meaning of several edit marks in Tivoli is determined not only by the shape of the mark but also by the context in which the mark is made. For example, a straight line over a bullet-point moves an item in a bullet-point list, while a straight line in a margin scrolls the drawing area. These types of inconsistencies can potentially confuse the user. To avoid these problems, we wanted to provide context sensitive information about which edit marks a user can make over what objects. Informally, we wanted a user to be able to answer the question: "what marks can I draw on this object or location?". Since marking menus are sensitive to context (i.e., the contents of a menu may vary depending on where it is popped up), we hoped that some similar mechanism could be designed for iconic marks in Tivoli.

For mark sets in general, besides Tivoli's iconic mark set and the marking menu mark set, the following characteristics may contribute to a mark's meaning and this type of information therefore needs to be self-revealed.

Shape: This is the case where a particular shape is an icon for a certain command. For example the "pigtail" shape is an icon for the delete command.

Direction: Sometimes the direction of a mark affects its meaning. For example a up-stroke means "scroll up" while a down-stroke means "scroll down". The shape of the mark is basically the same but the direction or orientation of the mark has meaning.

Location of features: The location of particular features of a mark can affect its meaning. For example, the summit of the "Type-in" point mark shown in Figure 6.10, determines the exact placement of the text cursor.

Dynamics of drawing: How a mark is drawn can affect its meaning. For example, a flick could mean "scroll to the end of document", while a slow up-stroke could mean "scroll to the next page".

6.3.2. Solutions

Crib-sheets

Interactive crib-sheets self-reveal marks without the overlap problem. When the user requires help, a crib-sheet can be popped up which shows the available marks and what they mean. The user can then dismiss the crib-sheet (or "pin" it down on the side) and make a mark. In Chapter 1, two systems that use mechanisms similar to this were described. Crib-sheets can be as succinct as a simple list of named marks or as elaborate as multi-page explanations of the marks in great detail. Thus a crib-sheet could contain complete information on all the characteristics of a mark. However, since crib-sheets are for reminding and guidance, they are usually succinct.

Figure 6.11 shows the crib-sheet technique we designed for Tivoli. The design works as follows. Similar to a marking menu, if one doesn't know what marks can be applied to a certain object or location on the screen, one presses-and-waits over the object for more information, rather than marking. At this point, rather than a menu popping up as in the marking menu case, a crib-sheet is displayed. The crib-

sheet displays the names of the functions that are applicable to the object or location, and example marks. If this is enough information, a user can draw one of the marks in the crib-sheet (or take any other action) the crib-sheet automatically disappears. If the pen is released without drawing mark, the crib-sheet remains displayed until the next occurrence of a pen press followed by a pen release or a press-and-wait event.

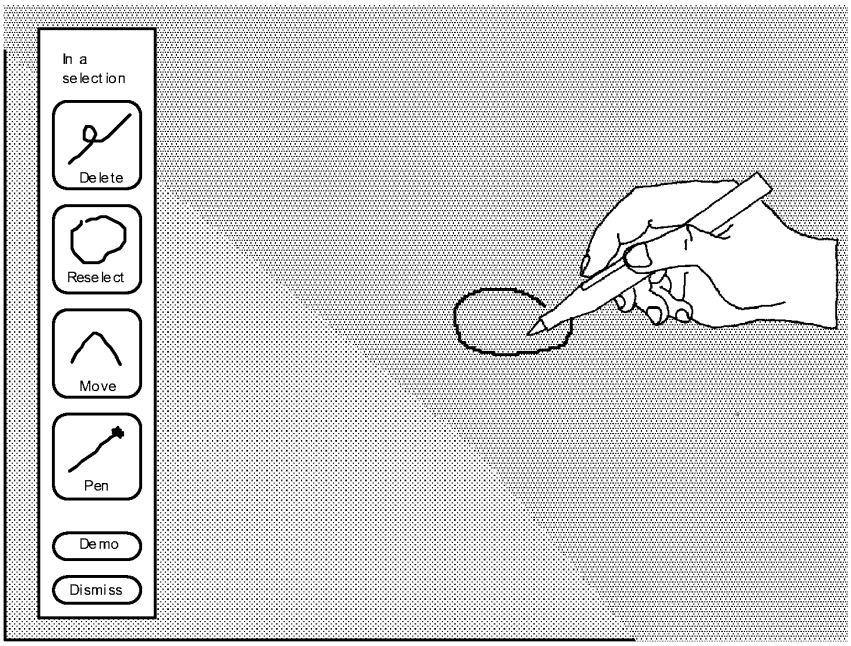


Figure 6.11: Self-revealing iconic marks in Tivoli: The user has selected the word "Tea" by circling it. To reveal what functions can be applied to the selection, the user presses-and-waits within the selection loop. A crib-sheet pops up indicating the context ("In a selection") and the available functions and their associated marks.

This design has several important features. First, the system displays the crib-sheet some distance away from the pen tip so that the crib-sheet does not occlude the context. This leaves room for a user to draw a mark. Second, the significance of the location of the pen tip is displayed at the top of the crib-sheet (i.e., in Figure 6.11 "In a selection" is displayed at the top of the crib-sheet). This is useful for revealing the meaning of different locations and objects on the screen.

This design obeys the principles of self-revelation, guidance, and rehearsal. The crib-sheet provides self-revelation, and a user can use the examples as guidance when drawing a mark. Rehearsal is enforced because a user must draw a mark to

invoke a command. For example, a user cannot press the delete button on the crib-sheet to perform a deletion. The user must draw a delete mark to perform a deletion.

Animated, annotated demonstrations

While the crib-sheet does self-reveal contextual information about marks, it still lacks certain types of information. For example, one static example of a mark relays little information about variations and features of a mark. It has been shown that people need good examples to help visualize procedures (Lieberman, 1987). Ideally a demonstration of the mark in context should be provided, similar to what one receives when an expert user demonstrates a command. The tutorial program in Windows for Pen Computing works like this. In the tutorial, a user is shown how marks are made by animated examples.

Demonstrations can be provided through animation. Baecker and Small have described how animation can assist a user, and how the animation of icons can be effective (Baecker & Small, 1990; Baecker, Small, & Mander, 1991). The idea of animated help is not new. Cullingford (1982) used "precanned" graphical animation coupled to natural language contextual messages to provide help. Feiner (1985) used graphical explanations to illustrate the problem solving process of real world physical actions. Feiner's system, however, was not sensitive to the user's current context. A research system, called *Cartoonist*, which automatically generates context sensitive animated help for direct manipulation interfaces, has been developed (Sukaviriya & Foley, 1990; Sukaviriya, 1988). The major difference between Cartoonist and the system we are about to describe is that Cartoonist is designed for direct manipulation interfaces, not mark-based interfaces. As we shall see, an animation of drawing a mark must have special features to make it meaningful and helpful. Specifically, in our system, the animation of a mark is annotated with text for explanation. Cartoonist does not support annotations. Furthermore, Cartoonist relies on an extensive knowledge base to describe the application and interface. The system we describe has a vastly simpler implementation which is compatible with existing user interface architectures.

Crib-sheets could be animated. A crib-sheet could show how to draw a mark, variations on a mark, and the various features of a mark. This certainly would help a user understand how a mark should be drawn. However, crib-sheets illustrate

marks outside of the context of the material that the user is working on, and this can make it difficult to see how the mark applies to the context. Marking menus, on the other hand, have the advantage of showing the available marks directly on top the object being worked on,.

To solve these problems we extended the function of the crib-sheet by adding animations of marks which take place in context. If the crib-sheet does not provide sufficient information, a demonstration of a mark can be triggered by pressing the "demo" button on the crib-sheet. The demonstration of the mark begins at the location originally pressed. The demonstration is an animation of the drawing of the mark which is accompanied by text describing the special features of the mark (see Figure 6.12).

There are several important aspects to this design:

• Marks are shown in context. The animation of the mark is full size, and emanates from the exact location originally pressed on by the user. A user can trace the animated mark to invoke the command.

• Variations in marks can be demonstrated by multiple animations. There is usually a variety of ways to draw mark. For example, a pigtail, signifying deletion, may be drawn in any direction, clockwise or counterclockwise, big or little. To prevent users taking a single animated example too literally, we show variations by animating multiple examples of mark. Usually, two examples seems to be enough.

• Information about features is provided by annotations. Not only is the drawing of a mark animated but the animation is annotated with text to explain features or semantics of marks (e.g., in Figure 6.12 "A pigtail deletes the *selected objects*."). In addition, features of the application can be displayed. For example, in Tivoli scrolling marks can only be drawn in the margins of the drawing area, but the borders of margins are not visible.[22] In situations like this, the animation can display these features to clarify matters. Annotations appear in sequence during the

[22] This was done to keep the drawing area uncluttered.

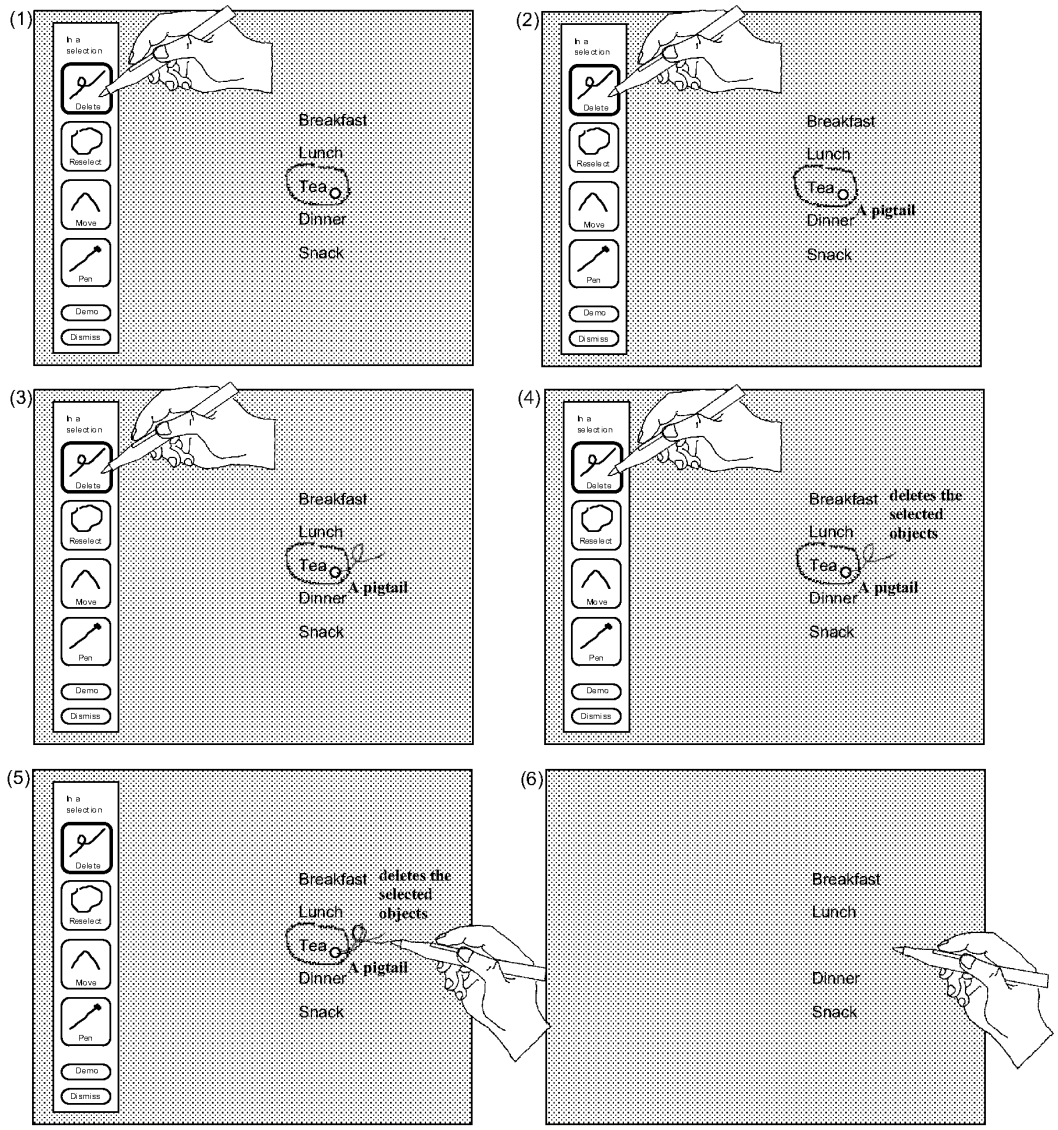


Figure 6.12: A demonstration of a particular function can be attained by pressing its icon. In (1) the user presses on the delete icon for more information. This triggers an animated demonstration of the mark with text annotation to explain its features. This is shown in (2), (3) and (4). In (5), the user traces along the example mark to invoke the function. When the pen is lifted, the action for the mark is carried out, and the crib-sheet and animation disappear (shown in (6)).

mark's animation, and they are timed to remain on the display long enough for the user to be able to read them.

• Animation can be controlled. A long series of animations takes quite a bit of time and this can be tedious for the user. By pressing a button in the crib-sheet, individual animations of the marks can be started or stopped. Pressing "Demo All" causes the system to cycle through all the animations. Pressing the "Dismiss" button stops the animation and removes the crib-sheet. As in the case of the crib-sheet by itself, the moment a user completes a mark, the crib-sheet is removed and the animation terminates.[23]

• The user is not required to make a mark from the crib-sheet. The user is free to perform any mark at any location on the screen while the animation is running. As before, the moment the user completes a mark, the animation and crib-sheet are removed. The user can also choose to not draw a mark by tapping the pen against the screen. This also removes the animation and crib-sheet.

Architecture

A goal for our crib-sheet/animation design was that it be easy for an interface programmer to use. We designed the software architecture with this in mind. To describe the characteristics of this architecture, we will describe an interactive computer system as consisting of two parts, an application module and animator module. The application allows the user to interact with a particular domain of materials by means of marks (i.e., Tivoli is the application and the materials are free-hand drawings). The animator is called by the application to show the marks to the user. The animator is generic—it can be made to work with different applications.

The design of the animator raises many specific design problems. We describe the animator by laying out the problems and describing how they are addressed.

How does the animator get invoked? This is the job of the application. As with a marking menu, the user deliberately presses-and-waits while the command button pressed. The application detects this action and then calls the animator.

---

[23] The animation actually freezes when a user begins drawing a marking so a user can trace the animated mark. The animation is removed from the screen when the user finishes drawing the mark and raises the pen.

How does the animator know which marks to animate? In order to make an application work with the animator, an application-specific Mark Animation Database (MAD) must exist. The MAD contains descriptions of examples of marks grouped by application context. When the user presses-and-waits, the application calls the animator with a description of the current context. The animator can then select the marks to be animated based on context.

How are marks and contextual features animated? In order to understand how marks are animated it is convenient to first understand the structure of MAD. Figure 6.13 shows an example of the structure of MAD. MAD consists of annotated examples of marks which are grouped by context. When the application calls the animator with a context, the examples corresponding to the context are retrieved from MAD. When a user requests a demonstration, the animator animates these examples. A mark is a sequence of x and y coordinates which is animated by incrementally displaying the mark. The marks that appear in MAD were originally drawn by hand. When animating a mark the animator uses the same drawing dynamics as the original hand-drawing (a technique developed by Baecker (1969)). In this way, dynamics of drawing can be revealed and the speed of an animation can be controlled by the constructor of the database. Annotations are labeled by where and when they should occur in the animation cycle (e.g., "start" and "end"). The pacing of the animation of text annotations is determined by length of text: after an annotation is displayed the animator pauses for an amount of time that is proportional to the length of the text before continuing with the rest of the animation. This gives a user time to read the annotation and then watch the rest of the animation.

How are variations shown? Variations are shown by animating another example of a mark. A mark in MAD can have more than one example. If an extra example is tagged as "variation", it is then included in the animation along with the original example.

How is the crib-sheet constructed? When the animator retrieves the examples from MAD, labels for the crib-sheet buttons are extracted, and example marks are shrunk down to be displayed in the buttons. We found it convenient to designate certain example marks for shrinking. Therefore, a function in MAD can contain an extra

example mark that is tagged for use as an "icon" in the crib-sheet. If no "icon" example is found, the animator shrinks the first example mark it finds.

How are application features animated? Like text annotations, application features appear in MAD. If during an animation an application feature needs to be displayed, the animator makes a call-back to the application. For example, the call-back may ask the application to display the margin boundaries of the drawing area. Therefore, a call-back protocol must exist between the application and animator.

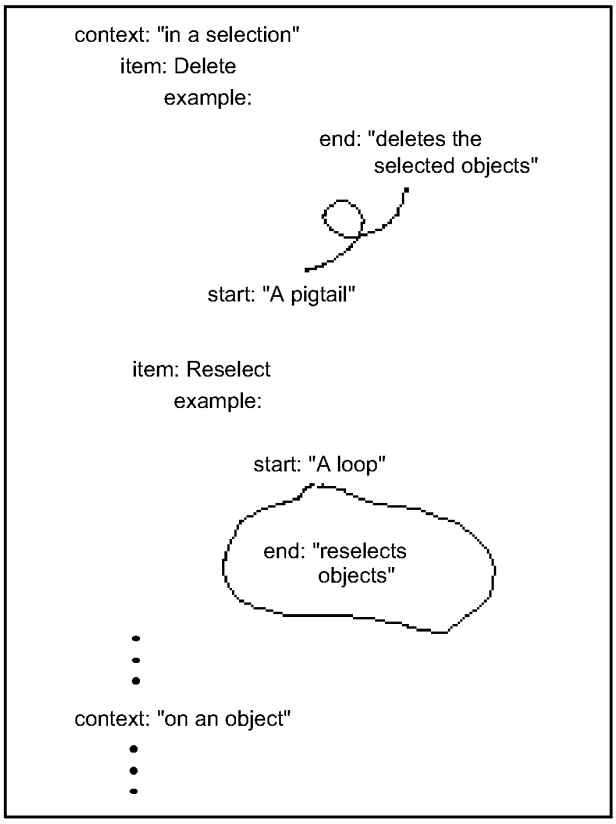


Figure 6.13: An example of the structure of the Mark Animation Database (MAD). Annotated examples of marks used for the crib-sheet and animations are grouped by context and function.

How are marks animated in constrained spaces? Assume that a user invokes the animator near the bottom of the drawing area, and that one of the possible marks at that point is a pigtail. At the bottom of the drawing area, there is no room to draw a

pigtail downwards, but there is room to draw it upwards. Thus, the animator should show only pigtails that fit in this place. The solution to this problem lies in the fact that MAD contains multiple examples of marks. When the animator retrieves examples from MAD it looks for examples that will fit in the space it is working with. Thus, MAD should be set up with many examples of each mark, so that the animator can find an example for any location. We found as little as four different examples were sufficient. In the event that an example which fits cannot be found, the animator generates and displays a "no room message (e.g., "not enough room to demo pigtail here"). This tends to only happen when there is not enough room for a user to actually draw the mark.

How is MAD constructed? MAD is constructed by drawing the examples in the form shown in Figure 6.13 and then copying these examples into MAD. For Tivoli, we constructed the examples by drawing them in Tivoli. Thus we could easily design examples that fit in constrained spaces in Tivoli by drawing them in those spaces. For example, we drew instances of pigtails that fit at the top, bottom, left and right edge of the screen. The animator does not have to be sophisticated at laying out the animations—the layouts are determined by the constructor of the examples. The animator need only check if an example will fit at a certain location. If it does not fit, it merely looks for another example.

More sophisticated features

The design for the crib-sheet/animator and MAD previously described has been implemented. Section 6.4 describes experiences using it. We now discuss future designs which are currently not implemented.

One problem with our current implementation is that, although animations do appear in context, they do not "work with" the context. For example, the animation of a loop being drawn to select objects sometimes doesn't enclose any objects. The problem is the animator has no knowledge about the application objects underlying the animation.

A more advanced version that we have not implemented extends the notion of parameterized marks to allow them to utilize application objects in the current working context. For example, assume we have a mark to move a list item. There would be two typed parameters to this mark: the list item and the location to which it is moved. In Tivoli, the list item would be a set of strokes between two "blue

lines" (like the blue lines on lined paper), and the location would be a blue line between two other list items. When the application calls the animator and tells it to animate a move-list-item mark, it would have to also give the animator some actual items and locations in the current context. The animator would then deform a move-list-item mark to fit the items and locations. Thus, the user would see a real example in the current context.

Having examples that manipulate the objects in the current context requires a much more sophisticated architecture for the animator. The animator must be able to manipulate objects in the application interface, and therefore a protocol that allows this must exist. Essentially, the distinction between the application and the animator becomes blurred in this more sophisticated scheme: the animator needs to know how to manipulate the application in the same way a user does. It must be able to identify objects and locations, construct marks and apply those marks. In addition, it needs to annotate the examples in a meaningful way. All these features require that examples in MAD be parameterizable. The design of this architecture is future research. A good starting point is to build on the work that Sukaviriya and Foley have done on the generation of parameterizable, context sensitive animated help for direct manipulation interfaces (Sukaviriya & Foley, 1990; Sukaviriya, 1988).

Integrating menu marks

As described earlier, menu marks in Tivoli are treated in the same manner as iconic marks. Specifically, menu marks will be interpreted as commands if drawn in command mode (i.e., drawn with the command button pressed down). The crib-sheet/animator provides self-revelation for all marks available in this mode including menu marks.

It would be impractical to include in the crib-sheet and animations all the marks used to access the pen settings menu. The menu is a much better mechanism for revealing this information, but is available only in drawing mode. Therefore, the crib-sheet/animator refers the user to the marking menu that is available in drawing mode. The animation of this is shown in Figure 6.14. The animation shows how to draw the dot required for a menu mark to be distinguished from other marks, and shows one example pen setting. The animation then displays a message for the user to see the marking menu available in drawing mode for more information. In this way an information link exists between the crib-sheet/animator and the marking

menu. Hence the crib-sheet provides self-revelation for the menu marks by referring the user to the marking menu.

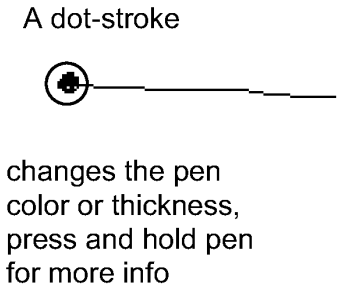


Figure 6.14: To self-reveal menu marks, the animator shows one example then refers the user to pop up the marking menu itself for more information. This avoids the problem of explaining and animating the many marks used for the marking menu.

6.4. USAGE EXPERIENCES

A large portion of the design described in this chapter has been implemented. The crib-sheet, animator, and MAD have been implemented, although the parameterized version of the animator was not implemented. Tivoli currently supports animations with multiple examples for every mark it uses. As Tivoli evolves, we expect the mark set to change. This can be supported by simply modifying MAD. The pen setting menu and marks were completely implemented. The "pull-out" menu item has yet to be implemented.

Future research will include formal user tests of our designs. It would be optimistic of us not to expect users to have problems with our system. First, there are many details that user might trip over: are the menus and buttons labeled meaningfully? Are the press-and-wait time thresholds correct? We believe the next step in user testing is to evaluate some of these details and refine the content of the animations. As Baecker, Small, & Mander (1991) point out, animations require significant development and refinement. Fortunately, our design makes this easier than a frame by frame process.

The design has been used informally by several researchers at Xerox PARC. Users appeared to be quite successful at using the marking menu, once press-and-wait was

understood. Users were also successful at selection using a mark but found recognition unreliable. We traced this unreliability to incorrectly drawn "dots" at the start of marks. We found the source was not that a user failed to draw the "dot", but that the system occasionally did not start tracking the pen till after the "dot" was drawn. This implies that the pen tracking hardware and software needed improvement.

Another problem revealed through informal use was the "right-handedness" of the marking menu. Depending on a user's handedness, some portion of the screen is occluded from view when one's arm is holding the pen against the screen. When using the marking menu, left handed users found some menu items occluded from view (they had to look "under" their arms). This implied that, like most pen-based systems, marking menus must be configurable for handedness.

Users also experimented with using the crib-sheet/animator. Initially, we found that users did not notice the crib-sheet pop up on the left side of the display. This was because users were so close to the large display that the crib-sheet popped up outside their visual focus. We then added an animation of the crib-sheet expanding from the point at which press-and-wait occurred. This helped users notice the display of the crib-sheet.

Users were also able to make use of the crib-sheet/animator after a brief demo. We found that users explored the interface by pressing-and-waiting at different spots to see what functions where available. We also observed users tracing the animated marks. The most common error involved a user pressing-and-waiting with the command button pressed, then releasing the button while watching the animation. The user would then trace the animated mark without the command button being pressed. This would result in the mark being drawn but not interpreted (i.e., the mark as drawn in drawing mode, not in command mode). We feel this type of error may disappear when a user gets into the habit of holding down the command button to issue a command. It is also possible to have the system recognize this error and advise the user to press the command button.

6.5. SUMMARY

In the beginning of this chapter we set out to integrate hierarchical marking menus into a pen-based application, and provide self-revelation, guidance, and rehearsal

for iconic marks. A design was developed and implemented to satisfy these goals. The design gives rise to many issues and conclusions:

The integration of marking menus into Tivoli reflects the situation with many applications today. Tivoli had an interface prior to our design experiment. Thus we were faced with the task of integrating marking menus with other interaction techniques. The main effect of this was that our design of marking menus had to change, not the existing interface components. This was an excellent test of the resiliency of the marking menus paradigm.

Marking menus had to be integrated with a range of interaction techniques. The interface to Tivoli not only contains edit marks but also free-hand drawing, buttons, dialog boxes, pop-up menus, mode buttons and a windowing system. Thus it was a challenge to find a spot where marking menus could fit in and be effective. The exploration also reminded us that interaction techniques cannot be added to an interface design without considering the other interaction techniques that surround it.

There were many other situations where we could have experimented with marking menus. One goal in redesigning the Tivoli interface was to reduce the number of buttons on the screen. Consolidating many buttons into a marking menu, hence, removing them from the screen, would have accomplished this. Also, using marking menus to issue commands to Tivoli objects such as list-items would have been another effective use. Time constrained us to only explore one particular usage. We thought using a marking menu to control pen settings would elucidate many design issues, since the menu marks would have to be used in the same mode as the edit marks. Nevertheless, this simple implementation gave rise to many design issues which one would encounter in a larger scale integration.

This design exploration also revealed issues concerning using marking menus in mark-based interfaces. Figure 6.15 summarizes the major design problems and the solutions we developed. Specifically, ambiguities can develop between menu marks and iconic marks. We proposed three solutions: avoidance, different context, and distinguishing token. We elected to used a distinguishing token strategy, given the way marking menus were being used in Tivoli. The other strategies, however, can be useful in other situations. Also this design exploration allowed us to use marking

Problem	Solution
Ambiguity between iconic and menu marks.	Draw a distinguishing token (a "dot") at the start of an menu mark.
Menu items clipped-off near edge of screen.	Use pull-out menu item.
Object too close to edge of screen to mark.	Use pull-out mark.
Need self-revelation for iconic marks.	Use crib-sheet/animator.
Provide guidance for iconic marks.	Draw a mark based on crib-sheet example or... Trace a mark over an example displayed by the animator.
Ensure rehearsal of iconic marks.	A mark is the only way to issue a command.
Crib-sheet/animator should be easy to program and work at any screen location.	The programmer generates multiple examples in MAD.
Getting information on marking menus marks from the crib-sheet/animator.	A crib-sheet/animator item refers user to the marking menu.

Figure 6.15: Major design problems encountered integrating marking menus into Tivoli and the solutions developed.

menus with a pen. This uncovered issues and led to developments concerning screen limits and drawing dynamics.

The crib-sheet/animator is designed to support the principles of self-revelation, guidance and rehearsal. These mechanisms do not appear and behave exactly like marking menus, and we have shown why this must be so, but we feel that the design supplies the same type of information to the user and promotes the same type of behavior.

Designing a mechanism to self-reveal iconic marks brings to light many issues concerning the self-revelation of marks. First, revelation can occur at various levels of detail. The crib-sheet is the first level: a quick glance at the icon for the mark may be sufficient for the user. An animation is the second level: it requires more time

but provides more information and explanation. Our design essentially supports a hierarchy of information where there is a time versus amount of information tradeoff.

A hierarchic view of information can also be applied to the way in which marks themselves are self-revealed. For some marks, it is sufficient just to show a static picture of the mark. For other marks an annotated animation is needed before each one can be understood. Besides an animation, some marks need to show variations. Finally some marks, like menu marks, are best self-revealed incrementally. Depending on the characteristics of a mark, there are different ways of explaining the mark. This implies our self-revelation schemes must support these different forms of explanation. Marking menus, crib-sheets, and animations are instances of different forms of explanation. A complete taxonomy of forms of explanation is future research.

While user testing is needed to refine our design, we feel that this design supports the desired type of information flow. Users can interactively obtain information on marks and this information is intended to interactively teach them how to use these marks like an expert. No mark-based system that we know of supports this type of paradigm.

Chapter 7: Conclusions

7.1. SUMMARY

This dissertation develops and evaluates an interaction technique called marking menus. Marking menus were developed based on several observations:

1) Marks can be an efficient and expressive way to issue commands, especially for pen-based computers.

2) Marks, by themselves, are not easy to use because unlike buttons, menus, and icons, they do not automatically reveal themselves to a user.

3) Therefore, marks must rely on some other interaction technique to reveal themselves to the user.

Given these observations we designed an interaction technique that combines menus and marks with the intention that using the menu helps a user learn the marks. The design of marking menus was based the design principles of self-revelation, guidance, and rehearsal. The principle of self-revelation states the system should interactively provide information about what commands are available and how to invoke those commands. The principle of guidance states that the way in which this information is provided should guide a user through invoking a command. The principle of rehearsal states that the guidance provided should be a rehearsal of act of drawing the mark associated with a command. The goal of these design principles is to help a user learn and use marks and quickly move from novice to expert.

After proposing a design for the technique based on these principles, we then evaluated the technique. The intention of the evaluation was to determine the limitations of the technique.

The first evaluation was an empirical experiment on non-hierarchic (i.e., one level) marking menus. This experiment showed that certain configurations of menu items make marking faster and less error-prone. Specifically, the experiment showed that four, eight, and twelve item menus enhance performance when marking. Also this experiment showed that subjects, on average, performed marks faster and more accurately with a mouse and stylus/tablet than with a trackball.

The second evaluation was a practical case study of two users' behaviors using a six-item marking menu for a real-life editing task. From this study we observed several things. First, with practice, users learn to use the marks and tend towards using the marks 100% of the time. Second, users utilized the features of the technique that were designed to aid in learning the marks (i.e., reselection and mark-confirmation). Third, using a mark in this situation was on average 3.5 times faster than selection using the menu.

A third evaluation was an empirical experiment examining the effect of menu breadth and depth on users' performance when selecting from hierarchic marking menus using marks. We found as breadth and depth of a menu structure increases, subject performance slows and the number of incorrect selections increases. Error rate appears to the limiting factor when selecting using marks. The experiment examined menus of breadth four, eight, and twelve, and menu depths from one to four. A significant change in error rate occurred when menu depth was greater than two and breadth was eight or twelve. The results suggest that marks can be used to reliably select from four-item menus up to four levels deep, or from eight-item menus up to two levels deep. This experiment also examined the effect of using a pen or a mouse. We found that subjects, on average, performed better with the pen than with the mouse. However, the difference in performance was not large. This indicated that the mouse would be an acceptable input device for hierarchic marking menus.

A final design study examined generalizing the design concepts of marking menus. Marking menus are an interaction technique that provides self-revelation, guidance, and rehearsal for a particular class of marks (i.e., straight lines and zig-zag marks).

We developed an interaction technique that provides self-revelation, guidance, and rehearsal for more general classes of marks. We also showed why the technique must differ from marking menus, and described an efficient means of implementing the technique.

7.2. CONTRIBUTIONS

The contributions of this work can be divided into two categories: contributions concerning marking menus specifically, and contributions concerning larger issues of human computer interaction.

7.2.1. Marking menus

The design of marking menus is a contribution in itself because of several design features. These features were described in detail in Section 2.2. The following is a summary of the design features that make marking menus a valuable and unique interaction technique. Marking menus:

- Allow menu selection acceleration without a keyboard.
- Permit acceleration on all menu items.
- Minimize the difference between the menu selection and accelerated selection.
- Permit pointing and menu selection acceleration with the same input device.
- Utilize marks that are easy and fast to draw.
- Use a spatial method for learning and remembering the association between menu items and marks.
- Are implementable as a "plug-in" software module.

The empirical studies and case studies in this work have contributed in:

Proving that users behave with marking menus as predicted. The design of marking menus features three modes of interaction: menu mode, mark confirmation mode, and mark mode. The case study in Chapter 4 has shown that users utilize all

three modes in the transition from novices, who use menus, to experts, who use marks. The case study also showed that users performed marks as quickly as keypresses. An equivalent interaction implemented with accelerator keys would have required pointing with the mouse *and* pressing an accelerator key. Hence we can conjecture that interaction was faster with marks than with accelerator keys in this setting.

Increasing our understanding of the limitations of marking menus. There is a limit to how accurately one can select items from a marking menu using a mark. The experiment in Chapter 5 has determined that selection using marks from menus with more than eight items per level and more than two levels of hierarchy will be error-prone. However, if two levels of eight item menus are used, marks can be used to quickly select from 64 menu items.

Determining configurations of marking menus that produce the best performance. Certain configurations of menu items make marking faster and less error-prone than other configurations. Specifically, our experiments have shown that 4, 6, 8 and 12 item menus and on-axis items enhance performance.

Demonstrating how command item selection and command parameters can be combined. Our case study demonstrates how both the starting point and end point of a mark can be used to express command parameters. This results in efficient interactions.

7.2.2. Issues of human computer interaction.

This work has several contributions to the study of human computer interaction in that it:

Identifies the fact that markings are not self-revealing. In the past, it has been assumed that mark-based interfaces will be easy to use because marks will be "natural" or mnemonic. This may be true in a some situations but not in all cases. There is a danger of falling into the trap that a system will be easy to use because it uses marks. This research makes the important point that while marks can be a very efficient means of interaction, this efficiency cannot be obtained if the user does not first have knowledge about the mark set. In some situations our experience with everyday pen and paper conventions supplies this knowledge. In other situations it

does not, and a self-revealing mechanism must be provided in conjunction with the marks.

Develops interaction techniques for self-revealing markings. Marking menus are a solution to the self-revealing problem for one particular class of mark. The crib-sheet/animator is a solution for more general classes of marks.

Identifies and develops the design principles of self-revelation, guidance and rehearsal. To solve the problem of marks not being self-revealing, this research develops the design principles of self-revelation, guidance, and rehearsal. Marking menus serves as an example of the application of the design principles and the crib-sheet/animator demonstrates that the principles can be applied to other situations. We feel that these design principles are valuable for interface design in general.

Develops a unique way to support novice/expert differences. The notions of guidance and rehearsal are a unique way of supporting novice/expert differences and transitions in mark-based interfaces. We know of no other systems that use a similar scheme.

Other research has dealt with novice/expert differences by providing explicit novice/expert modes. In these types of systems, novice mode has fewer functions than expert mode. The focus of this research is on supporting novice/expert differences and transitions using mark-based interfaces at the level of interaction, not at the level of available functions. These two approaches differ but they are not mutually exclusive.

Demystifies "the folk legend of gesture" in human computer interaction. It is clear from the literature that the types of gestures performed while operating an interface contribute to the overall sense of satisfaction with an interface. While others have observed that careful design of the body language of interactions results in better interface design, the research here is an explicit attempt to make use of this philosophy in a practical interaction technique.

Identifies the real value of marks as an interaction technique. Finally this research demonstrates that if the real advantages of particular interactions are understood, simple technology, used appropriately, can exploit these advantages. It is not simply the case that marks are desirable because marks are easy to remember. Another desirable property is the ability of a mark to efficiently express a command

and its parameters. The marks created by marking menus demonstrate this property. Furthermore, the technology required to support this property is not overly complex. Recognition methods, and ways of embedding and recognizing command parameters, are easily programmable.

7.3. FUTURE RESEARCH

As we developed marking menus we came across many interesting design variations, extensions and applications worth exploring:

• Adapt marking menus to be used on very small screens. A problem with very small screen computers is that there isn't enough room to draw long marks or display hierarchic menus. A variation on our marking menu design is to use a series of short strokes, all starting from the same location to perform a selection from a hierarchy of menus.

• Investigate other types of combinations of marks and menus. Continuous menu items, and dartboard and donut layouts, which were mentioned in Chapter 1, are examples of other types of combinations of marks and menus.

• Investigate feedback and pairing with command parameters. This research has only scratched the surface of things that can be done while performing a selection or after making a selection. Marking menus need the ability to show system status (e.g., display the current font), to preview the effects of selecting a menu item (e.g., highlighting a particular font in a menu causes an example of the font to be displayed), and to embed command parameters after a selection is confirmed (e.g., after selecting "volume" a user is automatically connected to a graphical slider). Integrating these features while maintaining the design principles is an open problem.

• 3D marking menus. Marking menus are based on selection by direction in two dimensions with two dimensional pointing devices. A natural generalization is to three dimensions.

• While our research has established some upper bounds on the limits of hierarchic marking menus, a natural extension would be a case study of user behavior with hierarchic marking menus in a real application. We know from our first case study on non-hierarchic menus that with enough practice users will use marks. Hierarchic

menus have many more menu items than non-hierarchic menus. For example, a menu hierarchy which is two levels deep, with eight items in each menu, contains 64 items. It would be interesting to see if this potential could be tapped in an application.

• Further development and evaluation of the crib-sheet/animator is another topic for future research. Clearly, user testing of the design is required. Also developing a parameterized version of the animator is an interesting research challenge.

• Investigating the application of self-revelation, guidance and rehearsal to other domains, besides marking is of interest. An example of the use of guidance and rehearsal in another domain is keyboard driven menus. The menus serve to reveal functionality to a novice, and the novice is guided through the menu by hitting keys to select menu items. This guidance provides a rehearsal of an expert type of behavior in which menu items are selected without looking or waiting for the menus to be displayed.

• There are many open questions concerning using marks and motor behavior. Does using a distinct gesture when drawing a mark have an advantage? What is a distinct gesture? Are there ways that we can design the gestures of drawing marks such that learning or performance is improved?

7.4. FINAL REMARKS

The interfaces to many ordinary, non-computerized objects have properties which make human operation of them second nature. For example, gear-shifts and turn-signal levers in automobiles have labels which we initially look at to learn the function mappings but with experience these mappings become automatic. Furthermore, with practice, the gestures of operating these devices become secondary to the task of driving. The fact that the gestures are unique contribute to our ability to perform them with very little attention. This provides the advantage of allowing our attention to be focused on other more important tasks, for example, watching traffic or reading street signs.

In this thesis, we have tried to exploit these types of properties in the realm of the computer interface. As computers become more entrenched as our everyday objects, tools and instruments, it is not unreasonable to expect them to exhibit the

properties that make many non-computerized objects easy and effective to use. This dissertation contributes to the understanding and creation of human computer interactions that have these properties.

References

Allen, R. B. (1983) Cognitive factors in the use of menus and trees: an experiment. *IEEE Journal on Selected Areas in Communications,* SAC 1(2), 333-336.

Apple Computer (1992) *Hypercard User's Guide.* Apple Computer, Cupertino, California.

Baecker, R., & Small, I. (1990) Animation at the interface. In Laurel, B. (Ed.)*The Art of Human-Computer Interface Design,* 251-267, Reading Massachusetts: Addison Wesley.

Baecker, R., Small, I., & Mander, R. (1991) Bringing icons to life. *Proceedings of the CHI '91 Conference on Human Factors in Computing Systems,* 1-6, New York: ACM.

Baecker, R. M. (1969) Picture-driven animation. *Proceedings of the 1969 Spring Joint Computer Conference,* 273-278.

Barnard, B. J., & Grudin, J. (1988) Command Names. In Helander, M. (Ed.) *Handbook of Human Computer Interaction,* 237-255, B. V. North Holland: Elsevier Science.

Boritz, J., Booth, K. S., & Cowan, W. B. (1991) Fitts's law studies of directional mouse movement. *Proceedings of Graphics Interface '91,* 216-223.

Bush, V. (1945) As We May Think. *Atlantic Monthly.* July, 101-108.

Buxton, W. (1986). Chunking and phrasing and the design of human-computer dialogues. In Kugler, H. J. (Ed.) *Information Processing '86, Proceedings of the IFIP 10th World Computer Congress,* 475-480, Amsterdam: North Holland Publishers.

Buxton, W. (1990) The "Natural" Language of Interaction: A Perspective on Nonverbal Dialogues. In Laurel, B. (Ed.)*The Art of Human-Computer Interface Design,* 405-416, Reading Massachusetts: Addison Wesley.

Callahan, J., Hopkins, D., Weiser, M., & Shneiderman, B. (1988) An empirical comparison of pie vs. linear menus. *Proceedings of the CHI '88 Conference on Human Factors in Computing Systems,* 95-100, New York: ACM.

Card S. K. (1982) User perceptual mechanisms in the search of computer command menus. *Proceedings of the CHI '82 Conference on Human Factors in Computing Systems,* 190-196, New York: ACM.

Card, S. K., Moran, T. P., & Newell, A. (1983) *The Psychology of Human-Computer Interaction.* Hillsdale NJ: Lawerence Erlbaum.

Card, S. K., Robertson, G. G., & Mackinlay, J. D., (1991) The information visualizer, an information workspace. *Proceedings of the CHI '91 Conference on Human Factors in Computing Systems,* 181-188, New York: ACM.

Carroll, J. M, (1985) *What's in a name?* New York: Freeman.

Carroll, J. M., & Carrithers, C. (1984) Training wheels in a user interface. *Communications of the ACM,* 27, 800-806.

Coleman, M. L. (1969) Text Editing on a Graphics Display Device Using Hand-drawn Proofreader's Symbols. *Proceedings of the Second University of Illinois Conference on Computer Graphics.* 282-291, Chicago: University of Illinois Press.

Cullingford, R. E., Krueger, M. W., Selfridge, M., & Bienkowski, M. A. (1982) Automated explanations as a component of a computer-aided design system. *IEEE Transactions on System, Man and Cybernetics,* March/April, 168-181

Ellis, T. O., & Sibley, W. L. (1967) On the Development of Equitable Graphic I/O. *IEEE Transactions on the Human Factors in Electronics.* 8(1), 15-17.

Elrod, S., Bruce, R., Gold, R., Goldberg, D., Halasz, F., Janssen, W., Lee, D., McCall, K., Pedersen, E., Pier, K., Tang, J., & Welch, B. (1992) Liveboard: A large interactive display supporting group meetings. presentations and remote collaboration. *Proceedings of the CHI '92 Conference on Human Factors in Computing Systems,* 599-607, New York: ACM.

Fischman, M. G. (1984) Programming time as a function of number of movement parts and changes in movement direction. *Journal of Motor Behavior,* 16(4), 405-423.

Fitts, P. M. (1954). The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology,* 47, 381-391.

Furnas, G., Gomez, L., Landauer, T., & Dumais, S. (1982) Statistical Semantics: How can a computer use what people name things to guess what things people mean when they name things? *Proceedings of the CHI '82 Conference on Human Factors in Computing Systems,* 251-253, New York: ACM.

Gaver, W., W. (1991) Technology affordances, *Proceedings of the CHI '91 Conference on Human Factors in Computing Systems,* 79-84, New York: ACM.

Gibson, J. J. (1979) *The ecological approach to visual perception.* Houghton Mifflin, New York.

Gibson, J. J. (1982) *Reasons for realism: Selected essays of James J. Gibson.* Reed, E. & Jones, R. (Ed.), Hillsdale NJ: Lawerence Erlbaum.

Go (1991) *PenPoint System Manual*, Go Corporation, Foster City, CA.

Goldberg D., & Goodisman A. (1991) Stylus User Interfaces for Manipulating Text. *Proceedings of the ACM Symposium on User Interface Software and Technology,* 127-135, New York: ACM.

Gould, J. D., & Salaun, J. (1987) Behavioral Experiments in Handmarks. *Proceedings of the CHI + GI '91 Conference on Human Factors in Computing Systems and Graphics Interface,* 175-181, New York: ACM.

Guiard, Y., Diaz, G., & Beaubaton, D. (1983) Left-hand advantage in right-handers for spatial constant error: preliminary evidence in a unimanual ballistic aimed movement. *Neuropsychologia,* Vol. 21, No. 1, 111-115.

Hardock, G. (1991). Design issues for line driven text editing/annotation systems. *Proceedings of the Graphics Interface '91 Conference,* 77-84, Toronto: Canadian Information Processing Society.

Hoeber, T. (1988) Face to face with Open Look, Byte, V(13) (Dec. '88), 286-288.

Hopkins, D. (1987) Direction selection is easy as pie menus! *login: The USENIX Association Newsletter,* 12(5), 31-32.

Hopkins, D. (1991) The design and implementation of pie menus. *Dr. Dobb's Journal,* 16(12), 16-26.

Hornbuckle, G. D. (1967) The Computer Graphics User/Machine Interface. *IEEE Transactions on the Human Factors in Electronics.* 8(1), 17-20.

Jorgensen, A. H., Barnard, P., Hammond, N., and Clark, I., (1983) Naming commands: An analysis of designers' naming behavior. *Psychology of computer use,* Green T. R. G., Payne S. J., and van derr Veer, G. C. (Eds.), 69-88, London: Academic Press.

Keele, S. W. (1968) Movement control in skilled motor performance, *Psychological Bulletin,* 70, 387-403.

Kiger, J. L. (1984) The depth/breadth tradeoff in the design of menu-driven user interfaces. *International Journal of Man Machine Studies,* 20, 210-213.

Kirk, R. E. (1982) *Experimental design: Procedures for the Behavioral Sciences.* Belmont California: Wadsworth.

Krueger M. W., Giofriddo T., & Hinrichsen K. (1985) VIDEOPLACE—An Artificial Reality. *Proceedings of the CHI '85 Conference on Human Factors in Computing Systems,* 35-40, New York: ACM.

Kurtenbach, G. & Baudel, T. (1992) HyperMark: Issuing commands by drawing marks in Hypercard. *Proceedings of CHI '92 Conference poster and short talks,* 64, New York: ACM.

Kurtenbach, G. & Buxton W. (1991) Issues in combining marking and direct manipulation techniques. *Proceedings of UIST '91 Conference*, 137-144, New York: ACM.

Kurtenbach, G. & Buxton, W. (1991) GEdit: A testbed for editing by contiguous gesture. *SIGCHI Bulletin,* 22-26, New York: ACM.

Kurtenbach, G. & Buxton, W. (1993) The limits of expert performance using hierarchical marking menus. to appear in *Proceedings of the CHI '93 Conference on Human Factors in Computing Systems,* New York: ACM.

Kurtenbach, G. & Hulteen, E. (1990) Gesture in Human-Computer Communication. In Laurel, B. (Ed.)*The Art of Human-Computer Interface Design,*, 309-317, Reading Massachusetts: Addison Wesley.

Kurtenbach, G., Sellen, A., & Buxton, W. (1993) An empirical evaluation of some articulatory and cognitive aspects of "marking menus". *Human Computer Interaction* , 8(2), 1-23

Landauer, T. K. & Nachbar, D. W. (1985) Selection from alphabetic and numeric trees using a touch screen: breadth, depth and width. *Proceedings of the CHI '85 Conference on Human Factors in Computing Systems,* 73-78, New York: ACM.

Lee, E. & MacGregor, J. (1985) Minimizing user search time in menu driven systems. *Human Factors*, 27(2), 157-162.

Licklider, J. C. R. (1960) Man-Computer Symbiosis. *IRE Transactions on Human Factors in Electronics.* March 1960, 4-11.

Lieberman, H. (1987) An example-based environment for beginning programmers. *AI and Education: Volume One*, Lawler, R. and Yazdani, M., (Ed.), 135-152, Norwood NJ: Ablex Publishing.

Mackenzie, I. S. & Buxton, W. (1992) Extending Fitts' law to two-dimensional tasks. *Proceedings of the CHI '92 Conference on Human Factors in Computing Systems,* 219-226, New York: ACM.

Mackenzie, I. S., Sellen, A. J., and Buxton, W. (1991) A comparison of input devices in elemental pointing and dragging tasks. *Proceedings of the CHI '91 Conference on Human Factors in Computing Systems,* 161-166, New York: ACM.

Makuni, R. (1986) Representing the Process of Composing Chinese Temples. *Design Computing*. Vol. 1, 216-235.

Malfara, A. & Jones, B. (1981) Hemispheric asymmetries in motor control of guided reaching with and without optic displacement. *Neuropsychologia,* Vol. 19, No. 3, 483-486.

McDonald, J. E., Stone, J. D., & Liebelt, L. S. (1983) Searching for items in menus: The effects of organization and type of target. *Proceedings of Human Factors Society 27th Annual Meeting.* 834-837, Santa Monica, CA: Human Factor Society.

Momenta, (1991) *Momenta User's Reference Manual.* Momenta, 295 North Bernardo Avenue, Mountain View, California.

Morrel-Samuels, P. (1990) Clarifying the distinction between lexical and gestural commands. *International Journal of Man-Machine Studies* , 32, 581-590.

Nilsen, E. L. (1991) *Perceptual-motor control in human-computer interaction.* Technical Report No. 37, University of Michigan, Cognitive Science and Machine Intelligence Laboratory.

Norman, D. A. & Draper, S. W. (1986) *User centered system design: New perspectives on human-computer interaction.* Hillsdale, NJ: Erlbaum Associates.

Norman, D. A. (1981) Categorization of action slips. *Psychological Review*, 88, 1-15.

Normile, D. & Johnson, J. T. (1990). Computers without keys. *Popular Science,* August 1990, 66-69.

Paap, K. R. & Roske-Hofstrand, R. J. (1986) The optimal number of menu options per panel. *Human Factors*, 28(4), 1-12.

Paap, K. R. & Roske-Hofstrand, R. J. (1988) Design of Menus. *Handbook of Human Computer Interaction*, Helander, M. (Ed.), 205-235, B. V. North Holland: Elsevier Science.

Pederson, E. R., McCall, K., Moran, T. P., & Halasz, F. G. (1993) Tivoli: An Electronic Whiteboard for Informal Workgroup Meetings. to appear in *Proceedings of the CHI '93 Conference on Human Factors in Computing Systems,* New York: ACM.

Perkins R., Blatt L. A., Workman D., & Ehrlich S. F. (1989) Interactive tutorial design in the product development cycle. *Proceeding of the Human Factors Society 33rd Annual Meeting,* 268-272.

Perlman, G. (1984) Making the right choices with menus. *Proceedings of Interact '84,* 317-320, B. V. North Holland: Elsevier Science.

Rasmussen, J. (1983) Skills, Rules and Knowledge: Signals, Signs and Symbols and other Distinctions in Human Performance Models. *IEEE Transactions on Systems, Man and Cybernetics,* SMC-13, 257-266.

Rebello, K. (1990) New PCs can kiss keyboards good-bye. *USA Today,* Feb. 22., 6B.

Rhyne, J. R. & Wolf, C. G. (1986) *Gestural Interfaces for Information Processing Applications.* IBM Technical Report 12179 (#54544).

Rhyne, J. R. (1987) Dialogue Management for Gestural Interfaces. *ACM Computer Graphics.* 21(2), 137-142.

Robertson, G. G., Henderson, Jr. A. D., & Card S. K., (1991) Buttons as First Class Objects on an X Desktop. *Proceedings of UIST '91 Conference,* 35-44, New York: ACM.

Rubine, D. (1991) Specifying Gestures by Example. *Computer Graphics,* 25(4), 329-337.

Rubine, D. H. (1990) *The Automatic Recognition of Gestures.* Ph.D. Thesis, Dept. of Computer Science, Carnegie Mellon University.

Sellen, A. J., Kurtenbach, G., & Buxton, W. (1992) The prevention of mode errors through sensory feedback. *Human-Computer Interaction.* 7(2), 141-164.

Sellen, A. J. & Nicol, A. (1990) Building user-centered on-line help. In Laurel, B. (Ed.)*The Art of Human-Computer Interface Design,* 143-153, Reading Massachusetts: Addison Wesley.

Sellen, A. J. (1992) Speech patterns in video-mediated conversation, *Proceedings of the CHI '92 Conference on Human Factors in Computing Systems,* 49-59, New York: ACM.

Shneiderman, B. (1987) *Designing the User Interface: Strategies for Effective Human Computer Interaction.* Reading Massachusetts: Addison-Wesley.

Sibert, J., Buffa, M. G., Crane, H. D., Doster, W., Rhyne, J. R., & Ward, J. R. (1987) Issues Limiting the Acceptance of User Interfaces Using Gestures Input and Handwriting Character Recognition. *Proceedings of the CHI + GI '91 Conference on Human Factors in Computing Systems and Graphics Interface,* 155-158, New York: ACM.

Snoddy, G. S. (1926) Learning and stability. *Journal of Applied Psychology* 10, 1-36.

Snowberry, K., Parkinson, S. R., & Sisson, N. (1983) Computer display menus. *Ergonomics,* 26(7), 699-712.

Sukaviriya, P. & Foley, J. D. (1990) Coupling a UI framework with automatic generation of context-sensitive animated help. *Proceedings of the ACM Symposium on User Interface Software and Technology '88,* 152-166, New York: ACM.

Sukaviriya, P. (1988) Dynamic construction of animated help from application context. *Proceedings of the ACM Symposium on User Interface Software and Technology '88,* 190-202, New York: ACM.

Sutherland, I. E. (1963) Sketchpad: A man-machine graphical communication system. *AFIPS Conference Proceedings 23,* 329-346.

Walker, N., Smelcer, J. B., & Nilsen, E. (1991) Optimizing speed and accuracy of menu selection: a comparison of walking and pull-down menus. *International Journal of Man-Machine Studies*, 35, 871-890.

Ward, J. R. & Blesser, B. (1985) Interactive Recognition of Handprinted Characters for Computer Input. *IEEE Computer Graphics & Algorithms* . Sept. 1985, 24-37.

Weiser, M. (1991) The computer for the 21st century. *Scientific American,* 265(3), 94-104.

Welbourn, L. K. & Whitrow, R. J. (1988) A gesture based text editor. *People and Computers IV, Proceedings of the Fourth Conference of the British Computer Society Human-Computer Specialist Group.* 363-371, Cambridge UK: Cambridge University Press.

Westheimer, G. & McKee, S. P. (1977) Spatial configurations for visual hyperacuity. *Vision Research*, 17, 941-947.

Wiseman, N. E., Lemke, H. U., & Hiles, J. O. (1969) PIXIE: A New Approach to Graphical Man-machine Communication. *Proceedings of 1969 CAD Conference Southhampton*, 463, IEEE Conference Publication 51.

Wixon, D., Whiteside, J., Good, M., & Jones, S. (1983) Building a user-defined interface. *Proceedings of the CHI '83 Conference on Human Factors in Computing Systems,* 185-191, New York: ACM.

Wolf, C. G. & Morrel-Samuels, P. (1987) The use of hand-drawn gestures for text editing. *International Journal of Man-Machine Studies* , 27, 91-102.

Wolf, C. G. (1986) Can People Use Gesture Commands? *ACM SIGCHI Bulletin*, 18, 73-74, Also *IBM Research report* RC 11867.

Wolf, C. G., Rhyne, J. R., & Ellozy, H. A., (1989). The paper-like interface. *Designing on Using Human Computer Interface and Knowledge-Based Systems.* 494-501, B. V. North Holland: Elsevier Science.

X11 (1988) *X window system user's guide for version 11.* X window series, v. 3, Sebastopol CA: O'Reilly & Associates._

Appendix A: Statistical Methods

This appendix explains the statistical methods used in this dissertation. Analysis of variance (ANOVA) is used for hypothesis testing. Specifically, the *F*-statistic is used to determine if an *independent variable* has any effect on a *dependent variable.* In an experiment, the dependent variable is a variable being measured. The independent variable is a variable being controlled.

Testing for differences in means: $F(k - 1, k(n -1)) = f, p < \alpha$.

Data is grouped according to different values of the independent variable. Each group is commonly referred to as a *treatment.* Random samples of size n are selected from each of k treatments. It is assumed that the k treatments each have a population that is independent and normally distributed with means $\mu_1, \mu_2, \ldots, \mu_k$ and a common variance σ^2. The null hypothesis can be represented as:

$$\mu_1 = \mu_2 = \ldots = \mu_k$$

The ANOVA procedure separates the total variability of the samples into two component: s_1^2 and $s^{2.}$ The variance s_1^2 is the variability between treatments attributed to changes in the independent variable and chance or random variation. The variance s^2 is the variability within treatments due to chance or random variation.

It can be shown that, assuming the null hypothesis is true, the ratio:

$$f = s_1^2/s^{2.}$$

is a value of the random variable F having the F distribution with k - 1 and $k(n - 1)$ degrees of freedom. Since s_1^2 overestimates the true variance when the null hypothesis is false, a large value for f suggest a large portion of the variance in the

dependent variable is caused by the independent variable. A test can be done by comparing the observed value f with the theoretical value of $F(k - 1, k(n - 1))$ and reporting the probability, p, of such a large value for f occurring simply by chance. If p is very small (e.g., $p < .05$), this suggests that the null hypothesis should be rejected.

Multiple comparison of means: Tukey HSD, $\alpha = p$

After determining a significant f ratio, it is may be necessary to determine which pairs of means are significantly different. Various procedures, which are referred to as post-hoc comparisons, allow this. If means μ_1 and μ_2 are being compared, the null hypothesis is:

$$\mu_1 - \mu_2 = 0.$$

A Tukey HSD post-hoc test reports the significantly differing means with a probability of α of incorrectly rejecting the null hypothesis (i.e., no difference exists between the means). Generally a .05 level of significance is used. This means one can be 95% sure that two means actually differ.

Contrasting means: $F(1) = f, p < \alpha$.

Post-hoc tests are not available for within subjects factors in repeated measures experimental design. An alternative method for determining which pairs of means are significantly different is by contrasting means. ANOVA separates the variance into two components: SSw and s^2. SSw is the variance attributed to the difference between the means. The variance s^2 is the variability due to chance or random variation.

It can be shown that, assuming the null hypothesis is true, the ratio:

$$f = SSw/s^2.$$

is a value of the random variable F having the F distribution with 1 and $n - k$ degrees of freedom. Since SSw overestimates the true variance when the null hypothesis is false, large values of f indicate a large portion of the variance is due to a difference between the means. A test can be done by comparing the observed value f with the theoretical value of $F(1, n - k)$) and reporting the probability, p, of such a large value

for f occurring simply by chance. If p is very small (e.g., $p < .05$), this suggests that the null hypothesis should be rejected.

Testing for linear relationships: F(1, n - 2)

The F-statistic is used to provide a single significance probability of a linear relationship between dependent and independent variables. In this case, the null hypothesis is that the slope of the regression line is zero. If the null hypothesis is true, then

$$f = SSR/s^2.$$

Where SSR is the amount of variation explained by the straight regression line. The variance s^2 is the variability around the regression line due to errors. It can be shown f is the value of the random variable F having the F distribution with 1 and n - 2 degrees of freedom. A test can be done by comparing the observed value f with the theoretical value of $F(1, n - 2)$) and reporting the probability, p, of such a large value for f occurring simply by chance. If p is very small (e.g., $p < .05$), this suggests that the null hypothesis should be rejected.

Testing a linear relationship for goodness of fit: r^2

The sample correlation coefficient r^2 is used to test the quality of the fit of a linear regression line. The amount of variation in the dependent variable which is explained by the independent variable is $r^2 \times 100\%$. A r^2 value greater than .5 is considered to indicate a linear relationship.

For further information on these statistical methods see Kirk (1982).

Navigation

- topics
  - Software
    - Pie Menus
      - Pie Menu Applications
      - Pie Menu Design
      - Pie Menu Implementations
    - The Sims
    - Languages
    - Music
    - OLPC
    - Servers
    - SimCity
    - Speech
    - Live Demos
    - Downloadable Software
  - Politics
  - Funny
- blogs
- Recent posts
- opml

Syndicate

User login

Username: *

Password: *

Log in

- Request new password

ITS.Svensson.org

The Design and Implementation of Pie Menus -- Dr. Dobb's Journal, Dec. 1991

Submitted by dhopkins on Tue, 2005-09-27 00:34. NeWS | Pie Menu Applications | Pie Menu Design

The Design and Implementation of Pie Menus

There're Fast, Easy, and Self-Revealing.

Originally published in Dr. Dobb's Journal, Dec. 1991, lead cover story, user interface issue.

Introduction

Although the computer screen is two-dimensional, today most users of windowing environments control their systems with a one-dimensional list of choices -- the standard pull-down or drop-down menus such as those found on Microsoft Windows, Presentation Manager, or the Macintosh.

This article describes an alternative user-interface technique I call "pie" menus, which is two-dimensional, circular, and in many ways easier to use and faster than conventional linear menus. Pie menus also work well with alternative pointing devices such as those found in stylus or pen-based systems. I developed pie menus at the University of Maryland in 1986 and have been studying and improving them over the last five years.

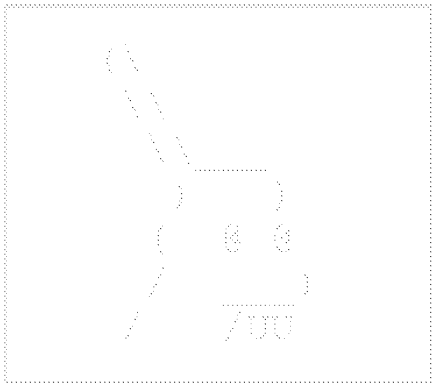
During that time, pie menus have been implemented by myself and my colleagues on four different platforms: X10 with the uwm window manager, SunView, NeWS with the Lite Toolkit, and OpenWindows with the NeWS Toolkit. Fellow researchers have conducted both comparison tests between pie menus and linear menus, and also tests with different kinds of pointing devices, including mice, pens, and trackballs.

Included with this article are relevant code excerpts from the most recent NeWS implementation, written in Sun's object-oriented PostScript dialect.

Pie Menu Properties

In their two-dimensional form, pie menus are round menus containing menu items positioned around the cursor -- as opposed to the rows or columns of traditional linear menus. The menu item target regions are shaped like the slices of a pie, and the cursor starts out in the center, in a small inactive region. The active regions are all adjacent to the cursor, but each in a different direction. You select from a pie menu by clicking the mouse or tapping the stylus, and then pointing in a particular direction.

Bongo



Technorati

www.DonHopkins.com:

www.DonHopkins.com/drupal:

Although there are multiple kinds of pie menus, the most common implementation uses the relative direction of the pointing device to determine the selection -- as compared with the absolute positioning required by linear menus. The wedge-shaped slices of the pie, adjacent to the cursor but in different direction, correspond to the menu selections. Visually, feedback is provided to the user in the form of highlighting the wedge-shaped slices of the pie. In the center of the pie, where the cursor starts out, is an inactive region.

When a pie menu pops up, it is centered at the location of the click that invoked it: where the mouse button was pressed (or the screen was touched, or the pen was tapped). The center of the pie is inactive, so clicking again without moving dismisses the menu and selects nothing. The circular layout minimizxes the motion required to make a selection. As the cursor moves into the wider area of a slice, you gain leverage, and your control of direction improves. To exploit this property, the active target areas can extend out to the edges of the screen, so you can move the cursor as far as required to select precisely the intended item.

You can move into a slice to select it, or move around the menu, reselecting another slice. As you browse around before choosing, the slice in the direction of the cursor is highlighted, to show what will happen if you click (or, if you have the button down, what will happen if you release it). When the cursor is in the center, none of the items are highlighted, because that region is inactive.

Pie menus can work with a variety of pointing devices -- not just mice, but also pens, trackballs, touchscreens, and (if you'll pardon the hand waving) data gloves. The look and feel should, of course, be adapted to fit the qualities and constraints of the particular device. For example, in the case of the data glove, the two-dimensional circle of a pie could become a three-dimensional sphere, and the wedges could become cones in space.

In all cases, a goal of pie menus is to provide a smooth, reliable gestural style of interaction for novices and experts.

Pie Menu Advantages

Pie menus are faster and more reliable than linear menus, because pointing at a slice requires very little cursor motion, and the large area and wedge shape make them easy targets.

For the novice, pie menus are easy because they are a self-revealing gestural interface: They show what you can do and direct you how to do it. By clicking and popping up a pie menu, looking at the labels, moving the cursor in the desired direction, then clicking to make a selection, you learn the menu and practice the gesture to "mark ahead" ("mouse ahead" in the case of a mouse, "wave ahead" in the case of a dataglove). With a little practice, it becomes quite easy to mark ahead even through nested pie menus.

For the expert, they're efficient because -- without even looking -- you can move in any direction, and mark ahead so fast that the menu doesn't even pop up. Only when used more slowly like a

traditional menu, does a pie menu pop up on the screen, to reveal the available selections.

Most importantly, novices soon become experts, because every time you select from a pie menu, you practice the motion to mark ahead, so you naturally learn to do it by feel! As Jaron Lanier of VPL Research has remarked, "The mind may forget, but the body remembers." Pie menus take advantage of the body's ability to remember muscle motion and direction, even when the mind has forgotten the corresponding symbolic labels.

By moving further from the pie menu center, a more accurate selection is assured. This feature facilitates mark ahead. Our experience has been that the expert pie menu user can easily mark ahead on an eight-item menu. Linear menus don't have this property, so it is difficult to mark ahead more than two items.

This property is especially important in mobile computing applications and other situations where the input data stream is noisy because of factors such as hand jitter, pen skipping, mouse slipping, or vehicular motion (not to mention tectonic activity).

There are particular applications, such as entering compass directions, time, angular degrees, and spatially related commands, which work particularly well with pie menus. However, as we'll see further on, pies win over linear menus even for ordinary tasks.

Pie Menu Flavors

There are many flavors or variants of pie menus. One obvious variation is to use the semicircular pie ("fan") menus at the edge of the screen.

Secondly, although the usual form of pie menus is to use only the directional angle in determining a selection, there is a vartiant of pie menus which offers two parameters of choice with a single user action. In this case, both the direction and the distance between the two points are used as parameters to the selection. The ability to specify two input parameters at once can be used in situations where the input space is two-dimensional. Direction and distance may be discrete or continuous, as appropriate.

For example, for a graphics or word processing application, a dual-parameter pie menu allows you to specify both the size and style of a typographic font in one gesture. The direction selects the font style from a set of possible attributes, and the distance selects the point size from the range of sizes. An increased distance from the center corresponds to an increase in the point size. This pie menus provides satisfying visual feedback by dynamcally shrinking and swelling a text sample in the menu center, as the user moves the pointer in and out.

Other variants include scrolling spiral pies, rings, pies within square windows, and continuous circular fields. These variants are discussed in a later section.

A minor variation in the use of pie menus is whether you click-and-

drag as the menu pops up, or whether two clicks are required: one to make the menu appear, another to make the selection. In fact, it's possible to support both.

Pie Menu Implentations

As mentioned earlier, several pie menu implementations exist, including: X10, SunView, and two NeWS implementation (using different toolkits).

I first attempted to implement pie menus in June 1986 on a Sun 3/160 running the X10 window system by adding them to the "uwm" window manager. The user could define nested menus in a ".uwmrc" file and bind them to mouse buttons. The default menu layout was specified by an initial angle and a radius that you could override in any menu whose labels overlapped. The pop-up menu was rectangular, large enough to hold the labels, and had a title at the top.

Then I linked the window manager into Mitch Bradley's Sun Forth, to make a Forth-extensible window manager with pie menus. I used this interactively programmable system to experiment with pie menu tracking and window management techniques, and to administer and collect data for Jack Callahan's experiment comparing pie menus with linear menus.

In January 1987, while snowed in at home, Mark Weiser implemented pie menus for the SunView window system. They are featured in his reknowned "SDI" game, the source code for which is available free of charge.

I implemented pie menus in round windwos for the Lite Toolkit in NeWS 1.0 in May 1987. The Lite Toolkit is implemented in NeWS, Sun's object-oriented PostScript dialect. Pie menus are built on top of the abstract menu class, so they have the same application program interface as linear menus. Therefore, pie menus can transparently replace the default menu class, turning every menu in the system into a pie, without having to modify other parts of the system or applications.

Because of the equivalence in semantics between pie menus and linear menus, pies can replace linear menus in systems in which menu processing can be revectored. Both the Macintosh and Microsoft Windows come to mind as possible candidates for pie menu implementations. Of course, for best results, the application's menus should be arranged with a circular layout in mind.

My most recent implementation of pie menus runs under the NeWS Toolkit, the most modern object-oriented toolkit for NeWS, shipped with Sun Open Windows, Version 3. The pie menu source code, and several special-purpose classes, as well as sample applications using pie menus are all available for no charge.

Usability Testing

Over the years, there have been a number or research projects studying the human factors aspects of pie menus.

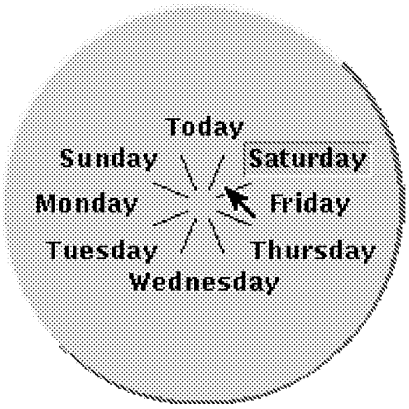
Jack Callahan's study compares the seek time and error rates in pies versus linear menus. There is a hypothesis known as Fitts' law, which states that the "seek time" required to point the cursor at the target depends on the target's area and distance. The wedge-shaped slices of a pie menu are all large and close to the cursor, so Fitts' law predicts good times for pie menus. In comparison, the rectangular target areas of a traditional linear menu are small, and each is placed at a different distance from the starting location.

Callahan's controlled experiment supports the result predicted by Fitt's law. Three types of eight-item menu task groupings were used: Pie tasks (North, NE, East, and so on), linear tasks (First, Second, Third, and so on), and unclassified tasks (Center, Bold, Italic, and so on). Subjects with little or no mouse experience were presented menus in both linear and pie formats, and told to make a certain selection from each. Those subjects uising pie menus were able to make selection significantly faster and with fewer errors for all three task groupings.

The fewer the items, the faster and more reliable pie menus are, because of their bigger slices. But other factors contribute to their efficiency. Pies with an even number of items are symmetric, so the directional angles are convenient to remember and articulate. Certain numbers of items work well with various metaphors, such as a clock, an on/off switch, or a compass. Eight-item pies are optimal for many tasks: They're symmetric, evenly divisible along vertical, horizontal, and diagonal axes, and have distinct, well-known directions.

Gordon Kurtenbach carried out an experiment comparing pie menus with different visual feedback styles, numbers of slices, and input devices. One interesting result was that menus with an even number of items were generally better than those with odd numbers. Also, menus with eight items were especially fast and easy to learn, because of their primary and secondary compass directions. Another result of Kurtenbach's experiment was that, with regard to speed and accuracy, pens were better than mice, and mice were better than trackballs.

The "Eight Days a Week" menu shown in Figure 1 is a contrived example of eight-item symmetry: It has seven items for the days of the week, plus one for today. Monday is on the left, going around counterclockwise to Friday on the right. Wednesday is at the bottom, in the middle of the week, and the weekend floats above on the diagonals. Today is at the top, so it's always an easy choice. The NeWS Toolkit code that creats this pie menu is shown in Listing 1.



Pie Menu Disadvantages

The main disadvantage of pie menus is that when they pop up, they can take a lot of screen space due to their circular layout. Long item labels can make then very large, while short labels or small icons make them more compact and take up less screen space.

The layout algorithm should have three goals: to minimize the menu size, to prevent menu labels from overlapping, and to clearly associate labels with their direction. It's not necessary to confine each label to the interior of its slice -- that could result in enormous menus. In a naive implementation, you might use text labels rotates around the center of the pie. But rotated text turns out not to work well, because it exaggerates "jaggies". This is hard to read without rotating your head, and doesn't even satisfy the goal of minimizing menu size.

One successful layout policy I've implemented justifies each label edge within its slice, at an inner radius big enough that no two adjacent labels overlap. To delimit the target areas, short lines are drawn between the slices, inside the circle of labels, like cuts in a pie crust.

One solution to the problem of pie menus with too many items is to divide up large menus into smaller, logically related submenus. Nested pies work quite well, as you can mark ahead quickly through several levels. You remember the route through the menus in the same way you remember how to drive to a friend's house: by going down familiar roads and making the correct turn at each intersection.

Another alternative is to use a scrolling pie menu that encompasses many items in a spiral but only displays a fixed number of them at once. By winding the cursor around the menu center, you can scroll through all the items, like walking up or down a spiral staircase.

Other Design Considerations

When you mark ahead quickly to select from a familiar pie, it can be annoying if the menu pops up after you've already finished the selection, and then pops down, causing the screen to repaint and slowing down interaction. If you don't need to see the menu, it shouldn't show itself. When you mark ahead, interaction is much quicker if the menu display is preempted while the cursor is in motion, so you never have to stop and wait for the computer to catch up. If you click up a menu when the cursor is at rest, it should pop up immediatly, but if you press and move, the menu should not display until you sit still. If you mark ahead, selecting with a smooth continuous motion, the menu should not display at all. However, it's quite helpful to give some type of feedback, such as displaying the selected label on an overlay near the cursor, or previewing the effect of the selection.

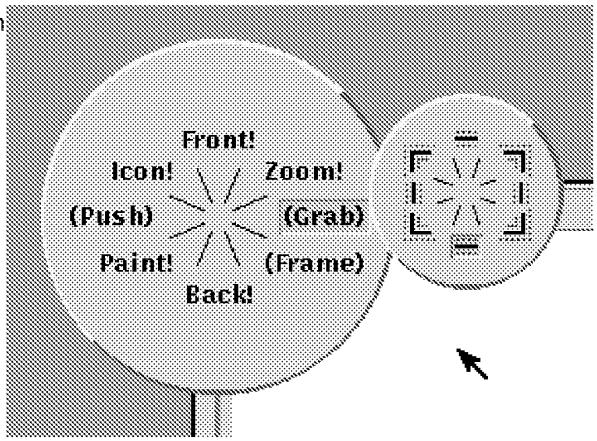
When you pop up a pie menu near the edge of the screen, the menu may have to be moved by a certain offset in order to fit completely on the screen, otherwise you couldn't see or select all the items. But it would be quite unexpected were the menu to slip out from under the click, leaving the cursor pointing at the wrong

slice. So whenever the menu is displayed on the screen, and it must be moved in order to fit, it is important to "warp" the cursor by the same offset, relative to its position at the time the menu is displayed. If you mark ahead so quickly that the menu display is preempted, the cursor shouldn't be warped. Pen- and touchscreen-based pie menus can't warp your pen or finger, so pie menus along the screen edge could pop up as semicircular fans. Note that cursor warping is also an issue that linear menus should address.

Ideally, pie menu designers should arrange the labels and submenus in directions that reflect spatial associations and relationships between them, making it easy to remember the directions. Complementary items can be opposite each other, and orthogonal pairs at right angles.

It's difficult to mark ahead into a pie menu whose items are not always in the same direction, because if the number of items changes, and they move around, you never know in which directions to expect them. Pie menus are better for selecting from a constant set of items, such as a list of commands, and best when the items are thoughtfully arranged to exploit the circular layout.

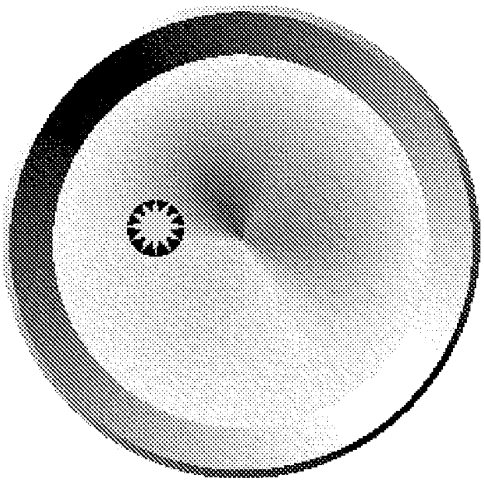
Sample Pie Menu



The pie menu shown in Figure 2 is an example of one that I added to the NeWS environment. Clicking on the window frame pops up this menu of window-management commands, designed to take advantage of mark ahead. Because this menu is so commonly used, you can learn to use it quickly, and save a lot of time. At the left of the figure is the top-level menu with commonly used commands and logically related submenus. The "Grab" item has been selected, popping up a graphical submenu of corners and edges. The icon for the bottom edge is highlighted, but has noot yet been selected. Clicking in that slice allows you to grab and stretch the edge of the window frame.

Figure 3 shows a second example, a color wheel that allows you to set the brightness, and to select a color from a continuous range of hues and saturations. The hue varies smoothly around the color wheel with direction, and the saturation varies smoothly with distance, with pure colors in the center fading to gray around the edge. Outside the pale perimeter is a continuous band of grays from white to black, that looks like the shadow inside a paint can, and functions as a circular brightness dial. Dipping into this gray border sets the brightness of the whole wheel. You may select any shade

of gray around the border, or move back into the paint can, to select a color at the current brightness. As you move around, the cursor shows the true color selected, and because the cursor is displayued even before the menu is popped up, you can mark ahead and select a color without popping up the menu!



Conclusion

Pie menus are easy to learn, fast to use, and provide a gestural style of interaction that suits both novices and experts. The techniques are available for anyone to share, so take a look and feel free!

» **dhopkins's blog** | **Login** to post comments

Bloomberg Businessweek

Innovation & Design

http://www.businessweek.com/stories/2008-01-02/the-long-nose-of-innovationbusinessweek-business-news-stock-market-and-financial-advice

The Long Nose of Innovation

By Bill Buxton January 02, 2008

The bulk of innovation is low-amplitude and takes place over a long period. Companies should focus on refining existing technologies as much as on creation

In October of 2004, Chris Anderson wrote an article in Wired magazine called The Long Tail, a theory he expanded upon in his 2006 book, The Long Tail: Why the Future of Business is Selling Less of More. In it he captures some interesting attributes of online services, using a concept from statistics which describes how it is now possible for the "long tail" of a low-amplitude population to make up the majority of a company's business.

One of his examples came from music: A large quantity of often obscure but nonetheless listened-to music can outperform a much smaller quantity of huge hits. The implications of the phenomenon have been significant for those interested in understanding the meaningful attributes of online vs. brick-and-mortar businesses and the book has apparently had an enormous impact among executives and entrepreneurs.

But those looking to apply the theory to the implementation of innovation within an organization should beware. My belief is there is a mirror-image of the long tail that is equally important to those wanting to understand the process of innovation. It states that the bulk of innovation behind the latest "wow" moment (multi-touch on the iPhone, for example) is also low-amplitude and takes place over a long period—but well before the "new" idea has become generally known, much less reached the tipping point. It is what I call The Long Nose of Innovation.

A Mouse Family Tree

As with the Long Tail, the low-frequency component of the Long Nose may well outweigh the later high-frequency and (more likely) high-visibility section in terms of dollars, time, energy, and imagination. Think of the mouse. First built in around 1965 by William English and Doug Engelbart, by 1968 it was copied (with the originators' cooperation) for use in a music and animation system at the National Research Council of Canada. Around 1973, Xerox PARC adopted a version as the graphical input device for the Alto computer.

In 1980, 3 Rivers Systems of Pittsburgh released their PERQ-1 workstation, which I believe to be the first commercially available computer that used a mouse. A year later came the Xerox Star 8010 workstation, and in January, 1984, the first Macintosh—the latter being the computer that brought the mouse to the attention of the general public. However it was not until 1995, with the release of Windows 95, that the mouse became ubiquitous.

On the surface it might appear that the benefits of the mouse were obvious—and therefore it's surprising it took 30 years to go from first demonstration to mainstream. But this 30-year gestation period turns out to be more

typical than surprising. In 2003 my office mate at Microsoft (MSFT), Butler Lampson, presented a report to the Computer Science and Telecommunications Board of the National Research Council in Washington which traced the history of a number of key technologies driving the telecommunications and information technology sectors.

Understanding Immature Technologies

The report analyzed each technology (time-sharing, client/server computing, LANs, relational databases, VLSI design, etc.) from first inception to the point where it turned into a billion dollar industry. What was consistent among virtually all the results was how long each took to move from inception to ubiquity. Twenty years of jumping around from university labs to corporate labs to products was typical. And 30 years, as with the mouse and RISC processors, was not at all unusual (and remember, this is the "fast-paced world of computers," where it is "almost impossible" to keep up).

Any technology that is going to have significant impact over the next 10 years is already at least 10 years old. That doesn't imply that the 10-year-old technologies we might draw from are mature or that we understand their implications; rather, just the basic concept is known, or knowable to those who care to look.

Here's the message to be heeded: Innovation is not about alchemy. In fact, innovation is not about invention. An idea may well start with an invention, but the bulk of the work and creativity is in that idea's augmentation and refinement. The newer the idea, the coarser the granularity of most analysis, and the more likely people are to say, "oh, that's just like X" or "that's been done before," without any appreciation for how much work and innovation is involved in taking an idea from concept to wide practice.

Rewarding the Art of Refinement

The heart of the innovation process has to do with prospecting, mining, refining, and goldsmithing. Knowing how and where to look and recognizing gold when you find it is just the start. The path from staking a claim to piling up gold bars is a long and arduous one. It is one few are equipped to follow, especially if they actually believe they have struck it rich when the claim is staked. Yet the true value is not realized until after the skilled goldsmith has crafted those bars into something worth much more than its weight in gold. In the meantime, our collective glorification of and fascination with so-called invention—coupled with a lack of focus on the processes of prospecting, mining, refining, and adding value to ideas—says to me that the message is simply not having an effect on how we approach things in our academies, governments, or businesses.

Too often, universities try to contain the results of research in the hope of commercially exploiting the resulting intellectual property. Politicians believe that setting up tech-transfer incubators around universities will bring significant economic gains in the short or mid-term. It could happen. So could winning the lottery. I just wouldn't count on it. Instead, perhaps we might focus on developing a more balanced approach to innovation—one where at least as much investment and prestige is accorded to those who focus on the process of refinement and augmentation as to those who came up with the initial creation.

To my mind, at least, those who can shorten the nose by 10% to 20% make at least as great a contribution as those who had the initial idea. And if nothing else, long noses are great for sniffing out those great ideas sitting there neglected, just waiting to be exploited.

Bloomberg Businessweek

Innovation & Design

http://www.businessweek.com/innovate/content/oct2009/id20091021_629186.htm

The Mad Dash Toward Touch Technology

By Bill Buxton October 21, 2009

Buried within the current mad scramble towards touch and multitouch technologies lies an important lesson in innovation: "God is in the details" (Ludwig Mies van der Rohe).

So while executives and marketers all seem to be saying, "It has to have touch," I am more inclined to say that anyone who describes a product as having a "touch interface" is likely unqualified to comment on the topic. The granularity of the description is just too coarse. Everything—including touch—is best for something and worst for something else. True innovators needs to know as much about when, why, and how not to use an otherwise trendy technology, as they do about when to use it. Let me explain.

The photo above shows four watches in my collection. On three of them (a, b, and c), the entire crystal is a touchscreen. Three of them (a, b, and d) have built-in calculators.

When Fat Fingers Meet Small Targets Watch (a) is the Casio AT-550. Despite its conservative styling, it has some pretty amazing software. To put it into calculator mode, you push a button on the lower left side. To enter numbers or operators into the calculator, you just draw them on the crystal with your finger. So, for example, a downward stroke from 12 to 6 o'clock enters the digit one (1), whereas the same stroke followed by a horizontal stroke from 9 to 3 o'clock enters a plus (+) sign. The numbers appear in the main part of the LCD window, and the current operator as a kind of superscript, above them.

The whole screen is used for entering each character, thereby bringing the scale of the action well within the bounds of normal human finger motor control. Less obvious but just as important, the technique enables "heads up" data entry—the equivalent of touch typing. In other words, I can input numbers without diverting my gaze from you or the document from which I am copying a number.

Watch (b) is the Casio TC-50. To put it into calculator mode, you also push a button on the lower left side of the watch. In this case, however, a graphical representation of the familiar calculator numerical keypad appears on the watch face. To enter a number, you touch the desired digit on the virtual keypad. To enter an operator, you touch the appropriate icon
(Ã·, x, -, +) permanently marked just below the LCD at the bottom of the watch crystal. The design is intended to take advantage of your previous experience with calculators. However, while this all seems clear, it does little to make the calculator usable. The watch is a victim of what happens when fat fingers meet small targets—even when accompanied by high concentration. As for touch typing, forget it.

Important Product Lessons Watch (c) is a Tissot Touch. While the crystal is touch-sensitive, this watch does not have a calculator. To activate the touchscreen you push and hold the watch stem for a couple of seconds.

Different functions are enabled by touching the crystal at particular places. For example, if you touch at the 6 o'clock digit, the hands of the watch align and point north, converting the watch into a compass.

Watch (d) is a third calculator watch, a Casio Data Bank 150. This one has a physical, mechanical keypad rather than a touchscreen. While the physical keys are small, they can be accurately used, but not without looking.

What I like about these watches is their power to teach us, using relatively simple existing products, important lessons about products that we might be dreaming about. Take watches (a), (b), and (c). Even though they are all just watches, and all use a touchscreen to gain access to their functionality, knowing how to use any one of them buys you pretty much nothing in terms of knowing how to use the other two. Even if you know how to use two of them, you still don't know how to use the third.

In fact, isn't it interesting to note that there is a closer affinity between the touch interface of (b) and the non-touch interface of (d) than between the two touch ones? In light of this, what in terms of user experience is conveyed by specifying that a product requires a touch interface? Very little. Yet how many of those insisting on a touch interface know about products such as these, much less the lessons that they have to teach?

Touch Isn't New As with almost any suddenly hot technology, touch and multitouch are decidedly not new. They are a textbook example of my notion of the "Long Nose of Innovation. " For example, multitouch was first discovered by researchers in the very early 1980s, before the first generally available PC using a mouse was commercially released. It has been gradually mined and refined ever since. The companies whose products have initiated the current buzz just happened to recognize the latent value of touch, and believe in it enough to take on the risk and investment required to effectively exploit its potential.

Significantly, these companies neither invented the underlying technology, nor were they the first companies to exploit it commercially. This is not a criticism, by the way, but rather a respectful commentary on the nature of design and innovation—one that counters the myth of the genius inventor, and gives appropriate recognition to those who laid the foundation that enabled this to happen.

Understand the Long Nose Finally, consider the following: Casio released the AT-550 in 1984 for under $100. That's the same year that the first Macintosh was released. Working Moore's Law backward, that means that wonderful "heads up" character recognition was created using only one 131,072th of the computer power that would be found on an equivalently sized chip today.

There is a serious lesson here for those would-be innovators who, on seeing the great success of one company's use of some technology or another, scramble to adopt it in the hope that it will bring them a share of that wealth as well. Such behavior is more appropriate for lemmings than innovators.

Rather than marveling at what someone else is delivering today, and then trying to copy it, the true innovators are the ones who understand the long nose, and who know how to prospect below the surface for the insights and understanding that will enable them to leap ahead of the competition, rather than follow them. God is in the details, and the details are sitting there, waiting to be picked up by anyone who has the wit to look for them.

Bill Buxton is Principal Scientist at Microsoft Research and the author of *Sketching User Experiences: Getting the Design Right and the Right Design*. Previously, he was a researcher at Xerox PARC, a professor at the University of Toronto, and Chief Scientist of Alias Research and SGI Inc.

The Sensor Frame Graphic Manipulator
NASA Phase II Final Report

PROJECT SUMMARY

PURPOSE OF THE RESEARCH:

Most of the useful information in the real world resides in humans, not in computers. Therefore we must find better ways of moving spatial information *from the human to the computer.* Quality 3-D graphics displays are necessary but *not sufficient* for a highly interactive and intuitive human interface. We need to improve input devices that capture human gestures and spatial knowledge.

One problem associated with direct manipulation interfaces in a design environment is that the user may not be skilled or precise enough to achieve the desired result. We can alleviate this problem through the use of *constrained virtual tools.* We define virtual tools as tools, displayed on the computer's video monitor, which are analogous to the tools used in factories, machine shops, or design studios. They include, but are not limited to, tools for cutting, smoothing, shaping, or joining operations. Virtual tools would map multifinger two and three-space gestures into the operations performed by the "business end" of the tool (such as the blade of a cutting tool), with constraints imposed by the model of the tool itself, the material or workpiece being operated upon, and the objectives of the user. The virtual tool would allow us to sculpt a smooth 3-D surface, varying the curvature or even the smoothness of a curve as it is drawn. However, the manipulation of a virtual tool requires *more* than six degrees of freedom. We believe that optical gesture recognition can provide up to twelve degrees of freedom per hand without the necessity for wires or gloves which inhibit casual use. The essential purpose of our research was to implement the enabling technology which makes casual use of virtual tools possible.

RESEARCH ACTIVITIES:

A prototype Sensor Cube was built using a neon-tube light source for contrast enhancement. A UNIX X-Windows interface was developed, and a control-panel builder was designed and implemented using X-Windows. A gesture-analysis package was developed, and is currently being extended for use in a multiple-finger environment.

RESEARCH RESULTS:

During the course of development of the three-dimensional Sensor Cube, we were informed that the sensors intended for use in the cube would no longer be available (see Section 3.1 for a more detailed discussion). This forced us to evaluate different approaches to optical multifinger sensing. Subsequently, we discovered a method of building the Sensor Cube with only one CCD sensor. This development will allow the three-dimensional Sensor Cube device to be less expensive than it's predecessor, the Sensor Frame. Unfortunately, the need to redesign the optical system and controller hardware and software of the cube delayed completion of this part of the project. Interesting and useful algorithms for 3-D finger tracking were developed and will be evaluated in detail as soon as sensor cube construction and interfacing are complete.

POTENTIAL COMMERCIAL APPLICATIONS:

The two-dimensional Sensor Frame technology will soon be supplanted by the three-dimensional capability of the Sensor Cube. However, the technology developed for use in the Sensor Frame has been transferred to a recently-announced commercial musical-instrument controller, the VideoHarp. The VideoHarp has attracted widespread attention in electronic-music circles, and was recently featured on the cover of Computer Music Journal (Volume 14, No. 1, MIT Press).

Sensor Cube gesture-recognition technology has it's greatest potential impact in computer-aided design (CAD) and teleoperation. Current input devices with six degrees of freedom or less are inappropriate for the manipulation of virtual tools. By gaining additional ability to capture the gestures of skilled scientists, designers, and technicians, computers will become a better alternative to traditional manual methods of design. If desktop manufacturing workstations with gesture-recognition input devices having up to 12 degrees of freedom can do for designers what time-sharing did for the programmers of the punch-card era, human productivity might be enhanced considerably; possibly by orders of magnitude.

1. Background and Motivation For Gesture-Based Systems

1.1. Virtual Reality and Virtual Tools

By the time a human child begins to speak, it has already spent approximately eighteen months to two years learning how to identify objects, people, and actions. It can distinguish one parent from another. It can distinguish itself from other objects and people. It can grasp and manipulate objects. Spatial knowledge comes early, and preceeds language.

Many young children can thread a nut onto a bolt before they go to school. A child less than four years old can do this. The task requires more than six degrees of freedom per hand (ie - positioning and orientation of the object in three-space plus a grasping operation), and implies that manipulation of twelve or more independent parameters is not unusually difficult for a young human.

In contrast, most workstations available today allow simultaneous manipulation of only *two* independent parameters, using a mouse. One *can* specify and manipulate representations of three-space objects with a mouse; but decomposing a six-parameter task into at least three sequential two-parameter tasks is not only counterintuitive, time-consuming, and error-prone; it is a waste of time if we can find a better way. By analogy, we could probably show that anything one can do using a keyboard can also be done using a telegraph key. But most of us would not exchange our computer keyboards for telegraph keys, despite the fact that the latter is cheaper, simpler, smaller, and standardized.

These considerations have prompted several researchers to attempt to improve workstation interfaces with a view toward accommodating human gesturing and tool-manipulation ability. In section 1.3, we will describe several systems which permit manipulation of objects in three dimensions. We will discuss their usefulness and their drawbacks, and ask how they might evolve in the future. While much of the published literature on 3-D input devices concentrates on the videogame-like ambiance of virtual reality, we will move the emphasis toward the idea of virtual tools, a subset of virtual reality that concerns itself with the development of more productive tools for use in design. Design and the need for redesign are among the most costly components in the production of high-technology products such as airplanes, rockets and space vehicles, and of low-tech mass-produced products such as automobiles.

1.2. Virtual Tools

One problem associated with direct manipulation interfaces in a design environment is that the user may not be skilled or precise enough to achieve the desired result. We can alleviate this problem through the use of *virtual tools*. We define virtual tools as tools, displayed on a workstation's video monitor, which are analogous to the tools used in factories, machine shops, or design studios. They include, but are not limited to, tools for cutting, smoothing, shaping, or joining operations. Virtual tools would map multifinger two and three-space gestures into the operations performed by the "business end" of the tool (such as the blade of a cutting tool), with constraints imposed by the model of the tool itself, the material or workpiece being operated upon, and the objectives of the user. The virtual tool would allow us to sculpt a smooth 3-D surface, varying the curvature or even the smoothness of a curve as it is drawn.

Virtual tools might be used to add material to a workpiece, to cut material, or to extrude it. The motion of a tool might be low-pass filtered, with filter-cutoff frequency of the filter being controlled,for example, by the distance between two fingers.

As we evolve hierarchies of virtual tools, designer productivity will hopefully increase. If we can significantly shorten design time, customization will be easier... and it is important to realize in this context that the higher-order goods of mass production, the machines that make other machines, are often highly customized tools, made in small quantities, but requiring many design iterations over their useful lifetime. As we build the virtual tools that cut design time, learning time for the designer will also be shorter, in relation to productivity. This is especially true if the designer can see "immediate feedback" on his or her latest design at low cost.

1.3. Related Research In Gesture-Sensing Technology

How can we best capture human gestures for intuitive manipulation of spatial objects? There are several different approaches to solving this problem. First, let's look at several currently-available devices:

- The DataGlove (VPL Systems)
- The Dexterous Hand Master (Exos)
- The Spaceball (Spatial Systems)
- The Flying Mouse (SimGraphics Engineering Corp.)

The DataGlove and Dexterous Hand Master (DHM) both sense finger-flexing motions. The DataGlove also senses hand position and orientation using a "Polhemus sensor" developed by McDonell-Douglas. The Polhemus sensor determines position and orientation of the hand using an externally-generated oscillating electromagnetic field. The version of the DataGlove with a Polhemus sensor has the advantage that it can sense relatively large-scale hand positions and orientations. Knowing position and orientation of the palm of the hand, one can use knowledge of finger-joint flexure to determine fingertip position, for use in grasping and tool-manipulation applications. In addition, by inserting piezoelectric transducers in the fingertips of the glove, one could conceivably provide some degree of touch feedback. Force feedback is a more difficult problem. The DHM has the advantage that its determination of finger-joint flexure appears to be considerably more accurate and repeatable than that of production DataGloves. It has the disadvantage that it does not currently provide hand position and orientation, although this could probably be implemented if market demand warrants it. Users of the DHM assert that it is lighter and less encumbering than it looks, although the time required to fit it to the hand seems to preclude casual use.

The use of glove-like sensors to sense gestures poses some problems. Currently, these devices use a cable to transmit data from the glove to the workstation, making casual use difficult. More later about the importance of casual use.

Hand (and consequently fingertip) position sensing (as opposed to detection of finger-joint flexure) requires the use of the relatively-expensive Polhemus sensor, and its use can be complicated by the presence and movement of ferrous metals in the vicinity of the sensor. A variation of the DataGlove developed by for Nintendo games, the PowerGlove, uses sonar devices mounted in the glove, but this severely constrains the orientation of the hand.

In the case of the DataGlove, unless each user has his own glove, a workstation supporting the device must have multiple gloves available in order to support left and right-handed persons with varying hand sizes. The same is probably true for the Exos device. Neither device yet provides sufficiently accurate and repeatable fingertip position information for use in a virtual tool environment. These latter considerations are an argument against the use of glove-like devices in a virtual tool (as opposed to virtual-reality) environment. Nevertheless, for many applications, we should expect them to provide a reasonably cost-effective solution.

The Spaceball is essentially a 3-D joystick. It is a ball slightly larger than a tennis ball, mounted in such a way as to make extended use very comfortable. The spaceball is excellent for positioning and orienting displayed 3-D objects, and for modifying one's view of a stationary object. It has good accuracy and repeatability. Because it functions like a joystick, it has some of the disadvantages that the joystick has relative to a mouse, and it has only six degrees of freedom. Six degrees of freedom are adequate for positioning and orienting objects, but more degrees of freedom are required to manipulate virtual tools. Once the tool is positioned, there are more things we must do to make it work, and that is the problem.

The Flying Mouse is a three-button mouse with a Polhemus sensor inside, designed so that it is easy to pick up. One can position and orient it in space, and then press the buttons. This is almost good enough for virtual tools, but not quite. For virtual tools, one might prefer the buttons to be more analog, ie - pressure sensitive. A nice thing about the Flying Mouse is that it can function as a normal 2-D mouse when on a tabletop, a convenient feature. The builder, Simgraphics Engineering Corporation, is well aware of the importance of the design and CAD markets, and emphasizes development of software necessary for the future "virtual tool" environment.

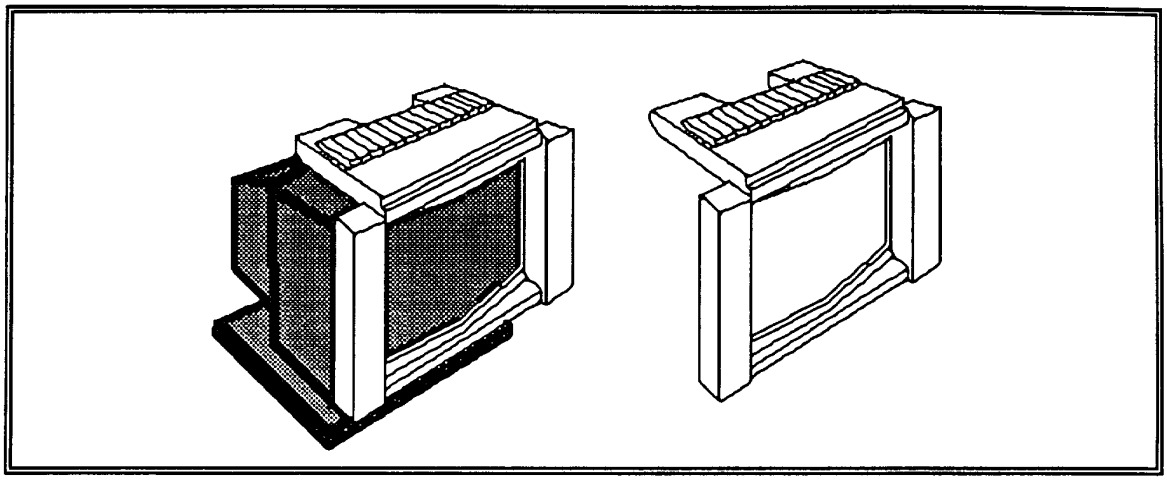
It is important to point out that the technologies we are describing are in their infancy, and constantly evolving. For this reason, many of the remarks pertaining to the products described above may become quickly outdated.

1.4. The Next Step: Vision-Based Gesture Sensing

The devices described in section 1.4 generally involve the use of mechanical, magnetic, or Hall-effect sensors in the sensing of palm position or finger flexure. A different approach to the problem of sensing multifinger gestures involves the use of vision-based systems.

Computer vision systems that analyze complex real-world scenes in real time remain beyond the state of the art. Nevertheless, in some applications, such as visual inspection, where scenes are specialized and predictable, systems are approaching feasibility (and a few systems are in commercial use).

At Sensor Frame Corporation in Pittsburgh, we have developed a device called a *Sensor Frame*, a 2-D optical finger-tracking device developed by the author and colleagues at Sensor Frame Corporation and Carnegie Mellon University. The prototype Sensor frame, using four sensors, reliably tracks up to three fingers at 30 Hz despite the fact that fingers sometimes block one-another from the point-of-view of some of the sensors. Tracking of *multiple* fingers is what distinguishes it from commonly-available touch screens. A drawing of the Sensor Frame, mounted on a monitor and in "standalone" mode, is shown below. The Videotape accompanying this report as Appendix C-1 shows the Sensor Frame in use.



The Mark IV Sensor Frame

Although the Sensor Frame represents a technology still in the early stages of it's development, it has aroused a fair amount of interest in industry, the press and media. In late 1988, CNN featured the Sensor Frame and VideoHarp in their AT&T Science and Technology series, and in 1989 Business Week featured both devices in their technology section. The Sensor Frame also appeared on the cover of NASA Tech Briefs, together with a feature article.

Unfortunately, production of the Sensor Frame, intended for September of 1989, was abruptly halted when the sensor manufacture halted delivery of optical dynamic-RAM sensors in the spring of 1989. This development is discussed in more detail in section 3.1. At present, we are developing the Sensor Cube, a 3D extension of the Sensor Frame, which will use one area CCD sensor to track up to three fingertips in three dimensions.

1.5. Future Applications of Gesture-Based Systems

Much of the motivation for building gesture-based systems can come from thinking about how we might apply them in the future in order to increase the productivity of designers. When we ask which gesture-sensing input devices will survive, we need to ask what future applications will require. Let's do a little thought experiment, and imagine what we would like our workstation to do for us if our objective were to design or modify a three-dimensional object, such as a machine-tool part, a piece of furniture, a molecule, or a nozzle for a rocket engine. We'll call this new type of workstation a *desktop manufacturing (DTM) workstation*, because it is intended to permit rapid prototyping of real-world objects. It would enable a designer to interactively specify or modify the shape of an object using spatial gestures and the *virtual tools* described above. Then it would build the object.

The DTM workstation would consist of the following components:

- A powerful CAD workstation that displays colored, shaded 3D objects, with full-motion video capability.
- A "3-D copier" similar to the stereolithography device manufactured by 3-D Systems Corporation. This device, or some future variation of it, will be used to fabricate a prototype or custom part quickly. There are currently at least three companies working on this aspect of DTM technology, and the number will probably increase.
- A 3-D gesture sensor, with gesture-recognition software and a *virtual-toolmaker's toolkit*.
- An optional 3-D laser scanner for scanning 3-D shapes.

2. Phase II Technical Objectives

Phase II Technical objectives consisted of the following:

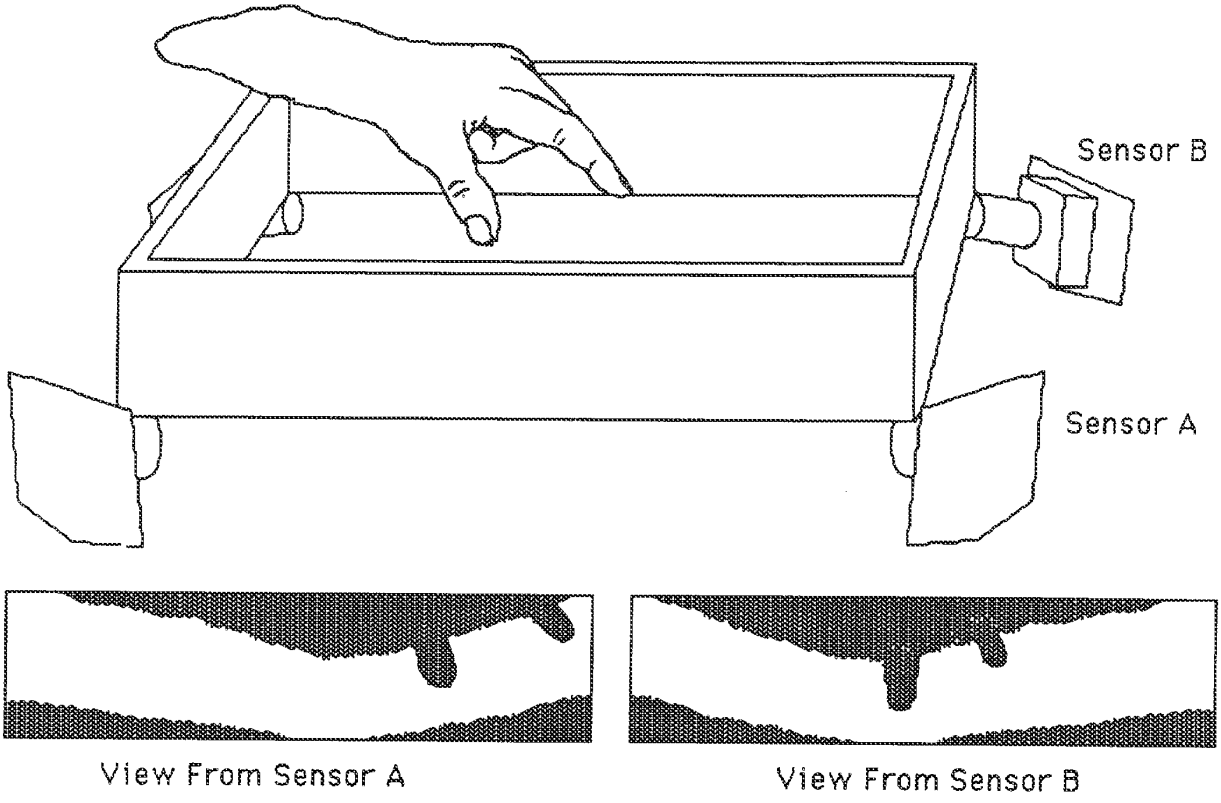
1. Development of Sensor Cube hardware and finger-tracking software.
2. Development of an intuitive interface for graphic-object manipulation.
3. Development of X-Window interface and UNIX device drivers for the Sensor Cube.
4. Development of soft control panels.

These objectives correspond to objectives 3.1.1 through 3.1.4, as described in our Phase II proposal for this project. Due to the sudden unavailability of DRAM sensors, as described in section 3.1 of this report, not all objectives were achieved in the form originally anticipated in the Statement of Work. Because the Sensor Cube design had to be modified significantly as a consequence of the sensor-availability problem, the resultant implementation delay precluded implementation of the 3D aspects of task 3.1.2.

3. Methodology, Observations, And Results

3.1. Development of Sensor Cube Hardware

In hardware terms, the *Sensor Cube* described in our NASA Phase II proposal was intended to be a thicker version of the Sensor Frame. A Sensor Cube was built with a 4.5" deep neon light source and four dynamic RAM (DRAM) sensors of the type used in the original Sensor Frame. This first Sensor Cube hardware was completed on schedule, about six months after the inception of Phase II. The first Sensor Cube prototype is shown schematically below, and in a videotape enclosed as Appendix C-2 of this report.



The First Prototype Sensor Cube, and Two Views From the Sensors

After completion of the first Sensor Cube prototype, work began on a UNIX interface for a Silicon-Graphics workstation, and at the same time for an X-Windows interface for an IBM RT workstation.

The UNIX and X-Window projects were essentially complete, in March of 1989, when Sensor Frame Corporation was abruptly informed by Micron Technology Corporation, the sole supplier of the DRAM sensors, that the fabrication of their line of DRAM sensors had been terminated. Our plans for commercial production of the Sensor Frame, intended to begin in August 1989, had to be abandoned. Both the then-current Sensor Frame and Sensor Cube designs made use of the 256K optical DRAMS supplied by Micron. All supplies of the 256K DRAMS had been committed to larger users by Micron before we were informed of the decision, leaving us with only five sensors; enough for our single prototype Sensor Frame, plus one spare. We were told that we would be able to obtain 100 of the smaller 64K DRAMs; however, we considered the 64K devices unsuitable for use either in a commercial Sensor Frame or in a Sensor Cube. Nevertheless, we bought the 100 64K devices, because we had a third product on the drawing boards that *could* use it; the VideoHarp.

It is perhaps relevant at this point to discuss the original reasons for the selection of DRAM sensors rather than charge-coupled devices (CCDs) as sensors, as well as the decision not to seek out another DRAM vendor to supply the optical DRAMs.

In 1982, when the first precursor of the Sensor Frame was built, CCDs were extremely expensive compared to DRAMs, with linear CCDs running in the thousand-dollar range. Further, CCDs require much more complex interface circuitry than do dynamic RAMs. In the early 80's, there were no integrated-circuit devices to provide the complex clock pulses, with their carefully-controlled slew rates, required by CCDs. Although integrated CCD clock and level-conversion chips became available in the mid-to-late 80's, the system cost of reasonable-quality CCDs is still considerably greater than the cost of optical DRAM chips. Further, the DRAM chips had several desirable properties that CCDs currently lack, one of the most important being addressability. In addition, it has not been any easier to obtain a second-sourced CCD than is was to obtain a second-sourced optical DRAM.

Although desperate, we were unable to convince Micron Technology to reverse their decision. They had little incentive to persue this still-relatively-small sensor market, having been awarded a virtual monopoly on the American DRAM market (along with Texas Instruments and IBM, the only remaining American DRAM manufacturers) by the U.S. Department of Commerce decision in 1988 to severely limit the importation of DRAMS from Japan.

As a consequence of this unfortunate event we did two things. First, since we could not manufacture Sensor Frames or Sensor Cubes, we decided to produce the VideoHarp, an optically-scanned musical instrument, which was the only one of the three products resulting from Sensor Frame technology that could make use of the available 64K DRAMs. Second, since we knew that we must switch to a different sensor technology after the 100th VideoHarp was built, and in order to build a commercial Sensor Cube (at present, we believe that it may be possible for a future VideoHarp and Sensor Cube to use the same area sensor), one of us (Paul McAvinney) attempted to find a way to build a Sensor Cube using fewer sensors. This effort succeeded shortly thereafter in the summer of 1989, when we developed a design using only one sensor and two mirrors.

The illustration below shows the a perspective view of the resultant single-sensor Sensor Cube mounted on a video monitor. Unlike the Sensor Frame, this design requires use of a gray-scale sensor such as a CCD or MOS area sensor. However, because it requires fewer sensors and associated optics, it will probably be cheaper to produce than the first Sensor Cube design. It's Z-axis depth will be about six inches, a significant improvement over the original design. In addition, the new design lends itself more easily to use as a two-handed teleoperation device. If two cubes are positioned side-by-side, they can share a controller.

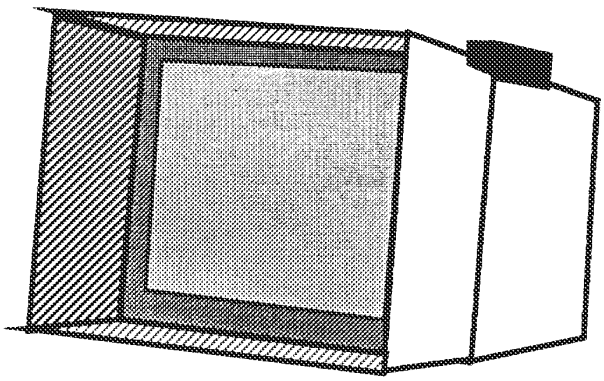
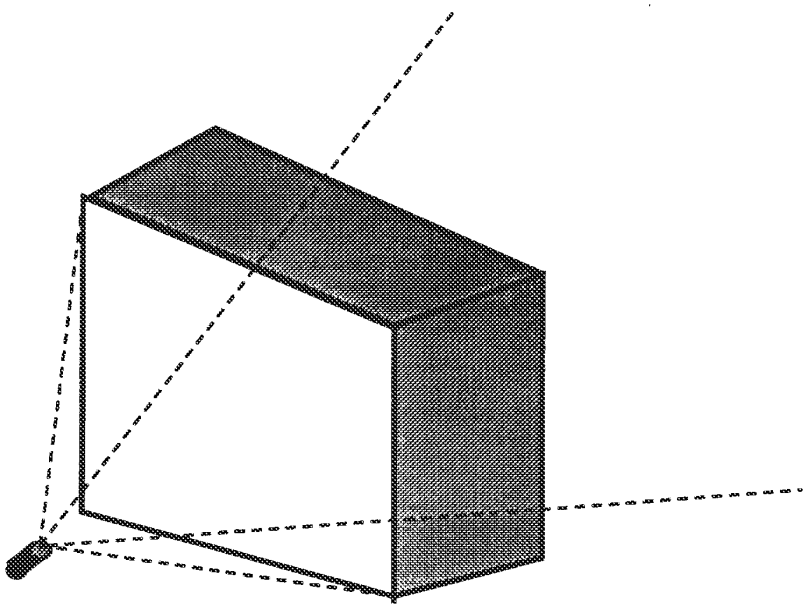


Figure 2: The Prototype Sensor Cube Mounted on a Video Monitor

Several important design considerations are driving the design of the second Sensor Cube. We list here the most important ones:

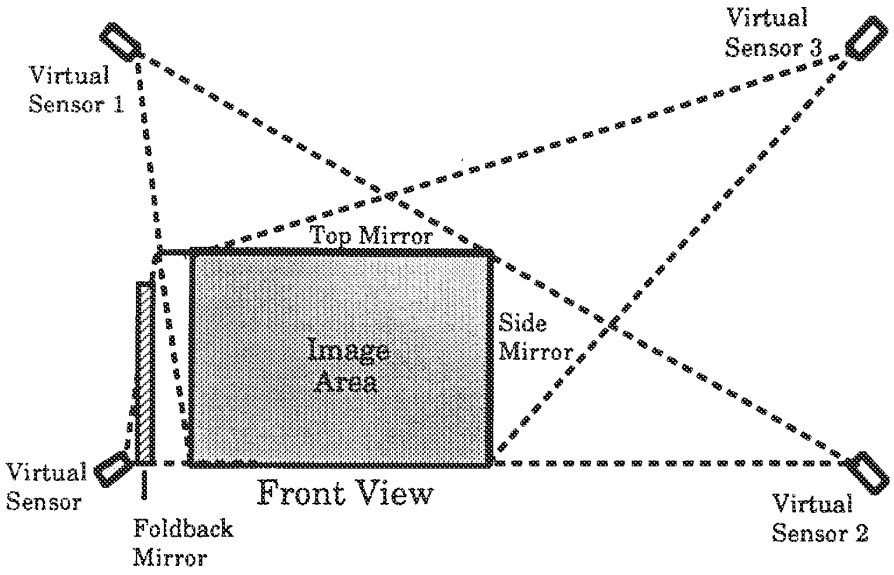
- The device must allow for at least ten degrees of freedom per hand, hopefully more. This will allow positioning and orientation of a virtual tool relative to a workpiece, followed by x-y manipulation of analog inputs on the tool itself by two opposed fingers. Even twelve degrees of freedom may not be too difficult to obtain.
- The device should allow casual use. This becomes especially important as increasingly powerful virtual tools permit a given operation to be completed in a short time, allowing the user to do something else which may not require the use of the gesture-sensing device. Good virtual tools should preclude the need for constant use, lessening concern about operator fatigue caused by holding one's hand in the air all day.
- The user's hands should be left free to use other devices, such as keyboards and telephones.
- Position of fingers relative to screen objects should be sensed.
- The device should be able to sense fingers in the vicinity of a video monitor. It should be attachable to the monitor, so that the user need not sacrifice desk space.
- It should operate independently of the video monitor, so that it can be mounted in another location (possibly for teleoperation-oriented applications) if the user so desires.
- It should be inexpensive in mass production, in order to encourage general use and standardization of application and user-interface software.

The next illustration shows a perspective schematic view of the new Sensor Cube design. The sensor is at lower left, and the shaded areas represent two mirrors at right angles.



Perspective View of Sensor Cube, With Monitor Screen At Rear

The next illustration shows the Sensor Cube from the front. Also shown are the positions of the *virtual sensors*. The scene produced in the single real sensor, at lower left, includes the scenes reflected from the mirrors along the top and right walls of the Sensor Cube enclosure. These virtual images may be treated geometrically as if they were images seen by the virtual sensors in the three positions shown. The net effect of the mirror system is to provide an image from four directions instead of just one. Since all sensors, real and virtual, look at the hand from a position near the plane of the video monitor, occlusion of fingers by the palm is minimized, except in the cases of extreme rotation of the hand.



Front View of Sensor Cube, Showing Virtual Sensors

The new Sensor Cube controller is currently under construction. Delays in delivery of support chips for the new design, based on a relatively inexpensive CCD designed by Texas Instruments, preclude the possibility of completion before the end of the NASA Phase II contract. Most U.S.- based CCD vendors have their CCD chips and support circuitry (and associated data sheets) produced in Japan for use in Japanese video cameras, and the U.S. wholesale market is small. As a consequence, some parts that have been on order for six months are still not being delivered.

A more long-term solution to the problem caused by the fact that there are currently no multiply-sourced area image sensors suitable for our designs is for us to design our own area sensor chip. This effort would make use of a scaleable CMOS process and the multi-foundry capabilities of the MOSIS prototyping service offered by the Information Sciences Institute at the University of Southern California (USC/ISI). PC-based software for MOSIS project-chip designs is available from commercial vendors at nominal cost.

Because of the uncertainty in the design schedule for this approach and our limited resources, we chose the more conservative approach of using commercial CCDs. Nevertheless, in the fall of 1989 we submitted a proposal to DARPA to fund a "smart" addressable MOS image sensor chip for use in gesture-based systems, but the proposal was rejected. We were told by DARPA that the proposal was considered technically sound, but that most if not all of their new funding had been reserved for HDTV and Star Wars projects. DARPA's approach may change, given the recent high-level shake-ups within the organization, but Sensor Frame Corporation intends to stake it's future on commercial product development (ie. - the VideoHarp), and fund new sensor development internally.

3.2. The Sensor Cube Finger-Tracking Algorithm

The algorithm for determining the spatial position and orientation of fingers in the Sensor Cube image area, stated here in somewhat oversimplified form, works as follows:

1) From the point-of-view of virtual sensor 3, the furthest sensor away from any sensed object, the scan line which intersects the mirrors at the greatest angle relative to the base-plane is read. This is guaranteed to sense a finger and allow it to be tracked at a z-axis value at or beyond the maximum guaranteed z-axis (Z_{MAX}) tracking value.

2) As any finger approaches Z_{MAX}, it is scanned by a "crosshair" pattern for each virtual sensor. One line of the crosshair is oriented *along the axis of the finger*, the angle being determined from previous scans. This is called the "longitudinal scan". For the simple case of a finger pointing directly along the Z axis, this value, taken from each virtual sensor, determines the position of the fingertip in the Z dimension. Information regarding fingertip position from each *longitudinal* scan is used to determine the height (above the fingertip) of the next *lateral* scan (see below).

3) The second scan is at right angles to the first, scanning across the *width* of the finger. This is called the "lateral scan". Information from each lateral scan is used to determine the lateral position of the next *longitudinal* scan.

In this method of tracking, each longitudinal scan corrects the position of the next lateral scan for a given finger, and vice-versa. Whether a frame-buffered image or an addressable sensor is used, the method allows us to locate fingers by scanning a relatively small fraction of the total number of pixels in the image, greatly reducing Sensor Cube controller processing requirements. In practice, two lateral scans of each finger may be needed to determine finger orientation accurately. When partial occlusion of a finger occurs, things become somewhat more complex. Experience with the Sensor Frame leads us to predict that we should not try to track more than three fingers at a time. This necessitates a style of gesturing which requires folding of the two smallest fingers into the palm. However, such a constraint appears to be easily learnable by most users. Further, the plane formed by three fingertips is useful for determining the orientation of a displayed object "grasped" by the hand. In the future, with more experience, we may try to relax the "three-finger" constraint.

3.3. Development of an Intuitive Interface for Graphic-Object Manipulation

Because the development of the Sensor Cube was delayed, the translation, rotation, grasping, and scaling of graphic objects in three dimensions was not possible. However, Appendix C-1, in the videotape attached to this report, shows how these capabilities were implemented using the Sensor Frame prototype for the two-dimensional case. We believe that when the Sensor Cube becomes operational, extension of these capabilities to the 3D case will not be difficult.

3.4. Development of an X-Window Interface and UNIX Device Drivers for the Sensor Cube.

X-Window and UNIX device-driver interfaces were successfully implemented for the Sensor Frame on IBM-RT and Silicon-Graphics IRIS workstations. The videotapes attached as appendices to this report show the effects of this implementation. Appendix B lists the implemented UNIX device-driver functions written in C.

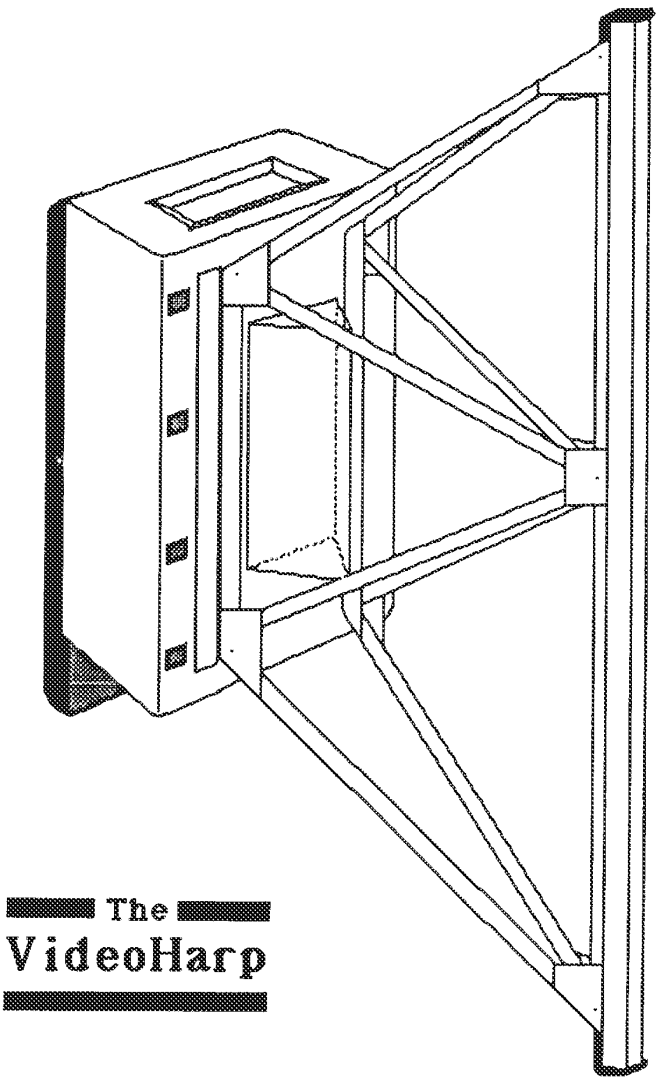
In general, it was found that the X-Windows interfaces (particularly on the IBM RT) were quite slow due to the excessive overhead of message passing between various X-Window components. This made multifinger tracking and screen update slow and difficult. The widely-acknowledged problem of excessive message-passing overhead has resulted in the recent appearance of terminals with processors dedicated to the efficient execution of X Windows.

Our implementation of the Sensor Frame on the Silicon-Graphics IRIS workstation was done using Sun's NEWS windowing system provided by Silicon Graphics.

3.5. Development of Soft Control Panels

The control-panel editing program was developed by researchers at Carnegie Mellon University under subcontract to Sensor Frame Corporation. An article describing this effort, "A Gesture Based User Interface Prototyping System", by Dr. Roger Dannenberg and Dale Amon of the School of Computer Science at Carnegie Mellon, was published in the Proceedings of the Second Annual ACM SIGGRAPH Symposium on User Interface Software and Technology, November 1989. That article is included in it's entirety as Appendix A of this report. The attached videotape (Appendix C-3) shows the operation of this system.

3.6. The VideoHarp

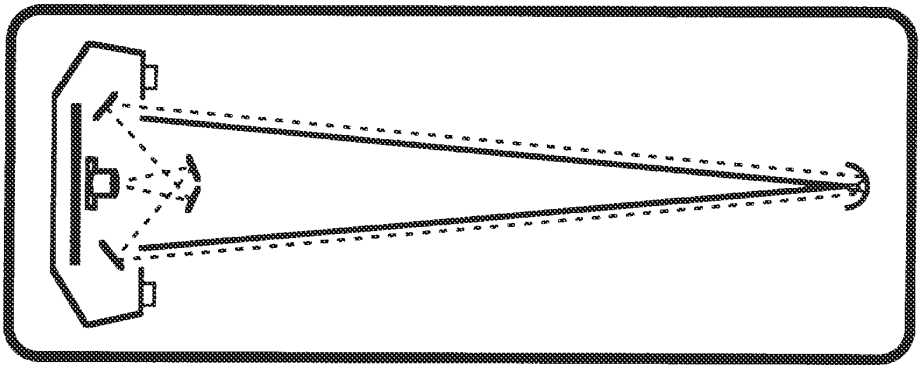


The VideoHarp is an optically scanned musical instrument which converts moving images of the fingers into music. Unlike keyboards and other mechanical sensing devices, the VideoHarp, because of the flexibility of it's optical scanning method, can recognize many classes of musical gestures, including bowing, strumming, keyboarding, and even conducting. A given class of gesture may be applied to any class of instrument timbre. For example, one could bow a horn or strum an organ. Using a fixed mechanical controller, such as a keyboard, one could produce non-keyboard sounds, such as the sound of a stringed instrument. However, even a keyboard with aftertouch cannot vary the timbre of a bowed note significantly as the note is played. Note that a cellist or violinist can press harder on the bow, play closer to the bridge of the instrument, and produce vibrato all at the same time. Keyboard controllers do not permit such quantitative, intuitive, and flexible control of many parameters at once.

The VideoHarp, because it can optically track all the player's fingers at once, allows control of many independent parameters. It permits a richness of timbral expression approaching, and often exceeding, that of traditional instruments. The playing surfaces of the VideoHarp can be divided up into *regions*. Each region possesses it's own attributes, such as which instrument is to be played, the width of keys, pitch and amplitude ranges, and many others too numerous to mention here.[1]

Some classes of gestures lend themselves well to *conducting*. For example, a stored score can be conducted using bowing motions. Each reversal of the bow causes the next note to be played. While one hand executes the bowing motions, the fingers of the other hand can be used to control additional aspects of timbre at the orchestral level[2]. This allows a novice user to obtain immediate musical results, and to express himself musically at a high level without having to learn all the nuances of the instrument. It is the *musical expression* which is important here, not the ability to specify which notes are to be played. That has already been done by the composer. Although a conductor may look at an orchestral score in order to plan what to do next, he is primarily interested in developing his own individualized expression or interpretation of the composition.

The following diagram illustrates the internal structure of the VideoHarp, as seen from above. The dashed line shows the light path from the light source (at right) to the sensor, at left. Mirrors are used to bend the light path so that both playing surfaces can be scanned by a single area sensor. Fingers placed against the playing surface block light from the light source, creating a shadow image on the sensor after being focused by a lens system (the cylindrical object at left).



The V2 VideoHarp, As Seen From Above

The VideoHarp can assume four different roles:

- **A Musical-Instrument Controller:** The VideoHarp is an optically-scanned free-hand gesture sensor adapted to the needs of the instrumentalist. It can be connected to any synthesizer with a MIDI input.

---

[1] For more information, see *The VideoHarp*, in Proceedings of the 14th International Computer Music Conference, Cologne Germany, 1988, Ed. Lishka and Fritsch.

[2] An orchestra can be thought of as a large instrument played by a conductor. The conductor does not specify the notes to be played, only *how* they are to be played.

- **A Conducting Controller:** The VideoHarp can capture gestures used in *conducting* a group of instruments, such as a quartet, an ensemble, or even a full orchestra.

- **A Composition Tool:** With it's ability to optically sense playing and conducting gestures of many types, the VideoHarp is an *enabling technology* which permits composers to experiment with the interaction between melody, tempo, timbre, and dynamics, with a flexibility and immediacy unmatched by current controllers.

- **A Complete Musical Instrument:** In the future, a VideoHarp with built-in synthesizer will be a complete musical instrument. At present, because there is no "standard" synthesizer, it is better to leave the choice of this device up to the user.

The VideoHarp has received national and international coverage in several publications, including Science News and Business Week. A color picture of the VideoHarp appeared recently on the cover of Computer Music Journal, which included a paper by the inventors.

4. Conclusions And Recommendations

We beleive that in the long run, vision-based gesture recognition systems such as the Sensor Cube will be widely used; first in design workstations, and later in personal computers, when full-motion video display of virtual tools and workpieces becomes inexpensive (this may happen relatively soon). We believe this for the following reasons:

- Casual, hands-free use of virtual tools will become increasingly important to users as the number, quality, cost, and utility of constrained virtual tools continues to shorten the design process and increase the number of people who will make use of it.
- Desktop Manufacturing (DTM) will allow fast prototyping, quick redesign, and inexpensive small-batch production of evolving products. As DTM becomes cheaper, a wider base of users will insist on standardized and portable virtual tools.
- Because each virtual tool must contain a description of the gesture-to-toolblade mapping, optical, rather than mechanical methods of gesture sensing permit the most flexible and repeatable interpretation of gestures having on the order of twelve degrees of freedom from a wide range of human hand and finger shapes.
- The Sensor Cube will be inexpensive in large quantities, and unobtrusive in casual use.

In the short run, we have to survive; we have had our problems obtaining a reliable supply of appropriate sensors and support circuits in a sensor market still dominated by video cameras for television applications. This situation has delayed construction of the Sensor Cube prototype (see Section 3.1), but things will probably improve. One Japanese image-sensor manufacturer has already requested that we submit a detailed proposal to them outlining our design requirements for a "smart" addressable area sensor. American IC manufacturers continue to lag in their understanding of the future role and importance of smart optical sensors which can detect and flag the pixel locations of image changes in the time domain.

We believe that the next great revolution in human productivity will be the result of a nonlinear increase in the utility and productivity of design tools. Good tools will make design more fun, and human creativity and productivity always profit when a process is viewed as being fun rather than work.

Appendix A: Reprint of Dannenberg/Amon SIGGRAPH Article

A Gesture Based User Interface Prototyping System

Roger B. Dannenberg and Dale Amon

School of Computer Science
Carnegie Mellon University
email: Roger.Dannenberg@cs.cmu.edu

Abstract

GID, for Gestural Interface Designer, is an experimental system for prototyping gesture-based user interfaces. GID structures an interface as a collection of "controls": objects that maintain an image on the display and respond to input from pointing and gesture-sensing devices. GID includes an editor for arranging controls on the screen and saving screen layouts to a file. Once an interface is created, GID provides mechanisms for routing input to the appropriate destination objects even when input arrives in parallel from several devices. GID also provides low level feature extraction and gesture representation primitives to assist in parsing gestures.

1. Introduction

Gestures, which can be defined as stylized motions that convey meaning, are used every day in a variety of tasks ranging from expressing our emotions to adjusting volume controls. Gestures are a promising approach to human-computer interaction because they often allow several parameters to be controlled simultaneously in an intuitive fashion. Gestures also combine the specification of operators, operands, and qualifiers into a single motion. For example, a single gesture might indicate "grab this assembly and move it to here, rotating it this much." Previous work on gesture based systems [1, 2, 6, 4, 12] has only begun to explore the potential of gestural input. We need a better understanding of how to construct gestural interfaces, and we need systems that allow us to prototype them rapidly in order to learn how to take advantage of gestures. Our work is a step toward these goals.

Building interactive systems based on gesture recognition is not a simple task. As we designed and implemented our system, we encountered several problems which do not arise in more conventional mouse-based systems. One problem is supporting multiple input devices, each of which might have many degrees of freedom. Unlike most mouse-based systems which can only engage in one interaction at a time, our system supports, for example, turning a knob and flipping a switch simultaneously.

Another problem is how to parse input into recognized gestures. We assume that gestures are specific to various interactive objects. For example, a switch displays an image of a toggle on the screen and can be "flipped" by a fingertip, but only if the finger travels across the image in the right direction. In this case, finger motion must be interpreted in the context of the interactive object, and a path (as opposed to instantaneous positions) defines the gesture.

Beyond these problems, we were also interested in making our prototyping environment easy to use, modular and extensible. Thus, we have been concerned with the issues of how to combine interactive objects in a screen-based interface, how to edit the layout and appearance of the interface, and how to encapsulate the behaviors of interactive objects and isolate them from other aspects of the system.

A final issue is the question of debugging support to aid in the implementation of new interactive objects. We use input logging to make bugs more reproducible and a combination of interpreted and compiled code to speed development.

We have completed a system, named GID for Gestural Interface Designer, in which one can interactively create and position instances of interactive objects such as menus, knobs and switches. One can interactively attach semantic actions to these objects. GID supports input from both a mouse and a free-hand sensor that can track multiple fingers. We are far from having the ultimate gesture based interface support environment, but we have developed interesting new techniques that are applicable to future gesture-based systems.

In section 2 we describe the structure of our prototyping system, and section 3 describes the handling of input from multiple devices. In section 4 we describe our general technique for processing input in order to recognize gestures. Section 5 describes in greater detail our develop-

ment techniques and the current implementation. Conclusions are presented in section 6 along with suggestions for future work.

2. The Interface Designer

This project extends an earlier effort called Interface Designer, or ID. The goal of ID was to provide a small, practical and portable system for creating screen-based interfaces by direct manipulation. ID was inspired by Jean-Marie Hullot's work at INRIA, a precursor to Interface Builder [5, 9]. A typical use of ID might be the following: by selecting a menu item, the user creates an instance of an object which displays a 3-D database. In order to manipulate the image, the user creates a few instances of sliders. A short Lisp expression is typed to supply an action for each slider, and labels of "azimuth", "altitude" and "pitch" are entered. Now, moving a slider causes a message to be sent to the display object and the image is updated accordingly.

The basic internal structure of ID introduces no significant improvements over other object-oriented event-driven interface systems such as MacApp [11] or Cardelli's user interface system [3]. It will be described here, however, for clarity.

ID represents the screen as a tree of objects. At the root is a screen object that contains a set of window objects. Each window object may contain a set of control objects. One type of control object is the control group, which serves to collect a set of control objects into an aggregate. Other types of control objects include sliders, buttons and switches of various styles. (See figure 2-1.)

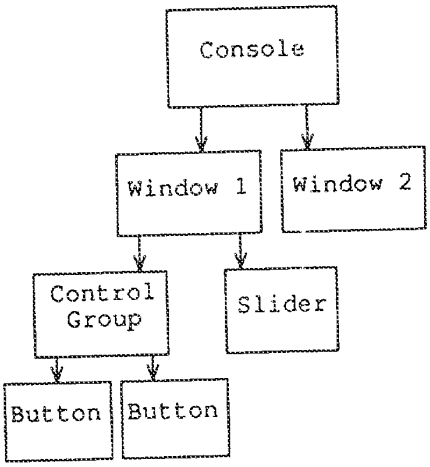


Figure 2-1: An ID control object tree.

In addition to the hierarchy implied by this tree, there is also a class hierarchy arranged so that classes can inherit much of their behavior. (See figure 2-2.) The InputControl class encapsulates generic behavior of objects that handle input from the user and manage some sort of image on the screen. PictureControls, a subclass of InputControls, actually draw images. These include classes such as Switch and Slider. Another subclass of InputControl is ControlGroup, which implements the search for an input handler. New interactive controls are typically created by subclassing PictureControl or one of its subclasses. Output-only "controls" have also been defined as subclasses of Control. For example, class 3dPict draws a wire-frame rendering of a 3-D data base which is loaded from a file.

```
Object
    Control
        InputControl
            PictureControl
                Button
                Switch
                Slider
                FontDev
            ControlGroup
                Console
                Window
                Menu
                MenuCard
                PictureGroup
            MenuItem
        OutputOnlyControl
```

Figure 2-2: Interface Designer class hierarchy.

In normal operation, ID has a single main loop that waits for input and delivers it to the appropriate destination. Each input event is represented by a window identifier, a device type (e.g. mouse or keyboard), coordinates (if any), and other data. This event is passed to the root of the tree where a search for a recipient begins. Typically, each node which is not a leaf node (a PictureControl) passes the event to each of its children until one of them accepts the input event.

To make this recursive search reasonably efficient, a ControlGroup object rejects mouse input which falls outside of its bounding box, and windows reject input unless the event's window identifier matches. Even with these optimizations, it is too inefficient to search the object tree from the root for each mouse-moved event during a dragging operation. Instead, a context mechanism is used.

In ID, the handler for input is found at the top of a *context stack*. An object can grab future input events by pushing a new context onto the stack to direct future input to the object. For example, a dragging operation would start with a mouse-down event that would be handled in the normal way. Upon receiving the mouse-down event, the object that handles the dragging operation pushes the context stack and becomes the target of future input. All successive mouse-move events go directly to the object. When mouse-up is received, the object pops the current context to restore input processing to normal.

The context stack has two uses in addition to temporarily grabbing mouse input. The context stack is used for nested

pop-up windows and also for implementing an "edit" mode in which control objects can be created, moved, copied, and deleted. In edit mode, we want to be able to select controls without invoking their normal operations. This is accomplished by pushing a special "edit context" which routes all input to an editor that can manipulate the on-screen objects.

3. Parallel Input Handling

We used ID as the basis for GID, our gesture-based system. GID was designed to be used with a Sensor Frame [7] as the gesture sensing device. The Sensor Frame tracks multiple objects (normally fingers) in a plane positioned just above the face of a CRT display. The "plane" actually has some thickness, so three coordinates are used to locate each visible finger. When a finger enters the field of view, it is assigned a unique identifier called the *finger identifier* Each time the finger moves, the new coordinates of the finger and the finger identifier are transmitted from the Sensor Frame to the host computer. Ideally, when a finger enters the field of view of the Sensor Frame, it is assigned a number which it retains for the entire time it remains in view. Since the Sensor Frame may be tracking multiple fingers in parallel, coordinate changes for several fingers may be interleaved in time.

In our gesture-based system, we wanted be able to handle multiple finger gestures acting on a single object, for example, turning a knob. We also wanted to allow users to operate a control with each hand. The stack-based context mechanism described in the previous section, however, does not allow inputs to be directed to several objects. We could simply pass all input to the root of the object tree, but again, the search overhead would be too high.

Our solution is to maintain a more general mapping from input events to objects. Each context contains a list of input templates, each of which has an associated handling object. Input templates consist of a window identifier, device type, and finger identifier. If all elements of the template match corresponding elements of an input event (the template may have "don't care" values) then the event is sent to the indicated handling object. If no template matches, then input is sent to a default handling object, also specified in the current context. As a result, we can have:

- two fingers operating a knob (input from either finger is forwarded immediately to the knob object),
- another finger moving toward a switch (input from this finger goes to the root of the object tree as usual. The switch object may change the current context and take future input directly when the finger gets close), and
- a simultaneous mouse click on a button (this input would work its way through the object tree from the root to the button object).

In some cases, one might want to effect a global context change, such as a pop-up dialog box which preempts all controls. This is accomplished by pushing a new context on the stack. This may redirect input from an object with a gesture in progress. We avoid problems here by sending a "finger up" event to the old handling object and a "finger down" event to the new handling object whenever a finger changes windows.

4. Gesture Representation and Processing

Since individual finger coordinates do not convey any dynamic aspects of gestures, the first stage of processing Sensor Frame input data is to represent the path of each finger by a set of features. The features are then interpreted by controls. The current set of features includes a piece-wise linear approximation of the path, the point where the path first crosses into an "activation radius", and the cumulative angular change.

4.1. Initial Processing

The x,y,z coordinates are supplied by the Sensor Frame as integers but are translated to floating point for further processing. The x,y,z portion of the input data is referred to hereafter as a *Raw Data Point* or *RDP*.

Normally, the default handling object for RDP's is the root of the object tree. The tree is searched after each input; however, when the RDP falls within the bounding box of a control object, the object responds by putting a template in the current context that will direct future events with the same finger identifier to the object. Future matching events will arrive at the object where they are added to a table associated with both the object and the finger identifier. This table of RDP's is called an *open vector*.

4.2. Path Decomposition

The next step is to process the open vector of RDP's to obtain a segmented[1] representation. This representation simultaneously provides data reduction and immunity from jitter.

For convenience, we want our approximation to be continuous; that is, each segment begins where the previous one ended, and all endpoints coincide with data points (RDP's). The algorithm for constructing the approximation is straightforward: as each RDP is added to the open vector, and error measure is computed. When the error measure exceeds a constant threshold, a segment from the first to the next-to-last point is added to the path and the open vector is adjusted to contain the last two RDP's. This algorithm can be described as "greedy without backtracking" since we pack as many RDP's into each segment as possible (limited by the error threshold) and we

[1] In this discussion, a *segment* is an ordered pair of points, e.g. RDP's, and a point is an x, y, z triple.

never try alternative assignments of RDP's to segments.

Figure 4-1 illustrates the process. The segment from point 1 to point 3 falls below the error threshold, but a segment from point 1 to point 4 exceeds the threshold. Therefore, the segment [point 1, point 3] is added to the path, and a new open vector [point 3, point 4] is started. This is extended to point 5 and then to point 6.

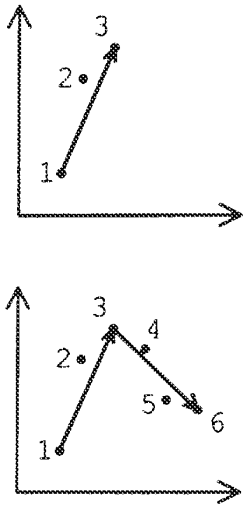


Figure 4-1: Fitting vectors to a set of points.

The error measure is:

$$error = \frac{1}{n}\sqrt{(\sum_{i=1}^{n}|D_x(p_i)|)^2+(\sum_{i=1}^{n}|D_y(p_i)|)^2+(\sum_{i=1}^{n}|D_z(p_i)|)^2}$$

where $D_x(p_i)$ is the x-component of the shortest vector from an RDP p_i to the proposed segment $[p_1, p_n]$ from point p_1 to p_n. We elected not to take a sum-of-squares in the innermost summation to save a bit of computation, and the resulting path decomposition seems to work well. The distance from a point to a line can be computed without trigonometric or square root functions as shown in Appendix I.

4.3. The Activation Volume

Gesture analysis is performed if an open vector passes into the volume defined by an activation radius and an activation center. Such processing will continue so long as succeeding RDP's remain within that volume.

An activation center is not necessarily static. For example, the knob on a slider has an activation center that moves along with it. The value associated with the device class is in this case a default initial value for the slider location.

Because we are polling the Sensor Frame from the application program, we cannot guarantee that we will catch all (or any) relevant RDP's within a possibly small *activation volume*. This is particularly true if the finger is traveling quickly. However, by setting the size of the bounding box large enough, we can guarantee we will at least pick up endpoints of a path segment that intersects this volume. The same distance algorithm (see Appendix I) used for path decomposition is then used to see if the point of closest approach of the path to the *activation center* is less than the *activation radius*.

4.4. Gestures

Once an RDP falls within the activation radius, the gesture features are examined by the corresponding object. Response to gestures is programmed procedurally for each type of control.

A toggle switch (or any other control affected by a simple linear motion), can be moved if the direction of travel of a finger path (A) matches the preferred axis of travel of a device (B). We define a maximum angle (θ_{max}) between the two and see if the actual angle (θ_{act}) is within bounds.

The actual angular error can be found using the definition of the vector dot product:

$$\mathbf{A}\cdot\mathbf{B} = |A||B|\cos(\theta_{act})$$

and rearranging to solve for $\cos(\theta_{act})$:

$$\cos(\theta_{act}) = (\mathbf{A}\cdot\mathbf{B})/(|A||B|)$$

If the inequality:

$$\cos(\theta_{act}) \le \cos(\theta_{max})$$

holds, then the movement of the finger is close enough to the preferred direction to cause a state change. Note that $\cos(\theta_{max})$ is a constant that can be precalculated, thus we avoid calculating transcendentals at run time by comparing cosines of angles instead of the angles themselves and by using the equation:

$$\mathbf{A}\cdot\mathbf{B} = A_xB_x + A_yB_y + A_zB_z$$

The knob rotation gesture consists of one or two fingers moving within the activation radius of the knob. Once it is determined that a finger path crosses the activation radius, an angle from the center of the knob to the finger is computed and saved. Each location change within the activation radius results in a recalculation of the angle, and the angle of the knob is updated by the angular difference. When there are two fingers within the activation radius, the knob is updated when either finger moves; the overall knob rotation is effectively the average rotation of the two fingers.

5. System Considerations

5.1. Implementation Languages

Our Sensor Frame interface, gesture recognition software, and graphics primitives are all implemented in the C programming language. Graphical and interactive objects, as well as the top-level input handling routines, are im-

plemented in XLISP, a lisp interpreter with built-in support for objects.

Although we would have preferred a compiled lisp, this work was begun at a time when our workstation environment was in a state of rapid change. During the course of the project, we ported XLISP to three machine types and implemented our graphics interface on two window managers. The fact that XLISP is a relatively small C program made it easy to port and to extend with the additional graphics and I/O primitives we needed.

5.2. Input Diagnostics

For diagnostic purposes, input of raw position data points is done through a device-independent module that allows input to come from a Sensor Frame, to be partially simulated by a mouse, or to be played back from a file that was "recorded" on a previous run with a mouse or a Sensor Frame. Bugs that appear only in long runs can be reproduced by playing back the log file during a debugging session.

The interface is implemented in such a way that regardless of which device is being used as the pointing device, the window menu is still available via the mouse. Commands are available to display every RDP as a small box on the screen; to print the results of every Sensor Frame input to a diagnostic window; to select a prerecorded file, a mouse or the Sensor Frame as the source of input; or to begin or end recording data for future playback.

6. Results and Conclusions

In the process of building GID, we have encountered several problems which are worth further study. One problem is how to organize prototyping software such as GID to allow controls to be operated in "run" mode and edited in "edit" mode. It seems inappropriate to implement editing within each object (Should a slider contain code for editing its size, placement, label, etc?), but a modular approach is preferable to a monolithic editor that captures all input in edit mode. In GID, we divert input when in "edit" mode, but we have specific editing methods in various subclasses of Control. One alternative is to implement all interactive behavior outside of control objects as in Garnet [8].

Another problem is that we have no high-level procedures for recognizing complex gestures: our recognizers must be hand-coded using fairly low-level representations. A promising alternative is the pattern recognition approach being pursued by Dean Rubine [10].

We know of no window managers that support multiple cursors. Ideally, the window manager should track each finger with a cursor and also determine what window contains each visible finger. Currently, the overhead of cursor tracking and mapping input to windows from outside of the window manager (X11) causes significant performance problems.

The present resolution of the Sensor Frame is only about 160 x 200 points. While this provides plenty of resolution relative to the size of controls displayed on the screen, greater resolution is needed in order to accurately measure the direction of motion and to minimize jitter.

The organization of GID prevents a single gesture from being received by multiple controls simultaneously. We do not feel this is a serious limitation, but it could be avoided by utilizing a more complete mapping from RDP's to objects. Rather than searching the object tree depth first, we could use hashing or a linear search of all objects to locate potentially overlapping bounding boxes which contain each RDP. Input events would then be duplicated and sent to each "interested" object. This technique was tried in an earlier system and allowed, for example, two adjacent switches to be flipped by moving a finger between them.

We note that some window managers might assist in the implementation of controls: if each control is implemented as a sub-window, then the search for a handler could be performed by the window manager. This technique will *not* work if we want input to reach multiple controls because current window managers will map input to only one window even if there is overlap. Furthermore, window managers typically assume a single pointing device, and extensive modification would be required to handle input from the Sensor Frame or some other gesture sensing device.

In conclusion, we have implemented a system for prototyping gesture-based user interfaces. The system is capable of editing its own interface, and applications are typically built by extension. The system allows us to experiment with screen layout and with multiple input devices without programming, and the system is extensible so that new interaction techniques can be integrated and evaluated. We have found piecewise linear approximations to paths to be an appropriate representation for simple gestures, and our vector software can be reused by different control objects.

7. Acknowledgments

We would like to thank Paul McAvinney, who played a crucial role in this research as inventor of the Sensor Frame and promoter of the value of gestures. Portions of this work were supported by the Sensor Frame Corporation as part of a project sponsored by NASA. Workstations used in this study were made available through an equipment grant from IBM. Thanks are also due to to Gerald Agin for the time he spent discussing curve fitting algorithms and analytic geometry.

References

1. R. A. Bolt. *The Human Interface: where people and computers meet*. Lifetime Learning Publications, 1984.

2. Frederick P. Brooks, Jr. Grasping Reality Through Illusion - Interactive Graphics Serving Science. CHI '88 Proceedings, May, 1988, pp. 1-11.

3. Luca Cardelli. Building User Interfaces by Direct Manipulation. Tech. Rept. 22, Digital Equipment Corporation Systems Research Center Research Report, Oct., 1987.

4. S. S. Fisher, M. McGreevy, J. Humphries, and W. Robinett. Virtual Environment Display System. ACM Workshop on Interactive 3D Grahpics, Association for Computing Machinery, 1986, pp. 77-87.

5. Jean-Marie Hullot. *Interface Builder*. Santa Barbara, CA, 1987.

6. Myron W. Krueger. *Artificial Reality*. Addison-Wesley, Reading, MA, 1983.

7. Paul McAvinney. U.S. Patent No. 4,746,770; Method and Apparatus for Isolating and Manipulating Graphic Objects On Computer Video Monitor. May 24, 1988.

8. Brad A. Myers. Encapsulating Interactive Behaviors. Human Factors in Computing Systems, SIGCHI'89, Austin, TX, April, 1989, pp. (to appear).

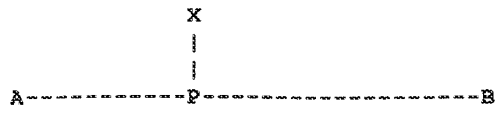
9. NeXT, Inc. *Interface Builder*. Palo Alto, CA, 1988. (online, preliminary documentation).

10. Dean Rubine. The Automatic Recognition of Gestures. (thesis proposal, Carnegie Mellon University School of Computer Science).

11. Kurt J. Schmucker. *Object-oriented programming for the Macintosh*. Hayden Book Co., Hasbrouck Heights, N.J., 1986.

12. David Weimer and S. K. Ganapathy. A Synthetic Visual Environment With Hand Gesturing and Voice Input. CHI'89 Conference Proceedings, Association for Computing Machinery's Special Interest Group on Computer Human Interaction, 1989, pp. 235-240.

I. Minimum Distance Between a Point and Segment



Parameterize equation of AB:

$$V(k) = (1-k)A + kB$$

for $0 \le k \le 1$, and $A \le V \le B$ so that V is any point on the segment between A and B.

Release the constraint on k for the time being, and let P be the point nearest X on AB: $P = V(k_p)$.

This gives us Equation 1:

$$Eqn\ 1 \quad P = (1-k_p)A + k_pB$$

or, in expanded form:

$$P = A - k_pA + k_pB$$

We want a line normal to AB that passes through X. By definition the dot product is zero if $\angle APX = 90^o$, so for $XP \perp AP$ we have:

$$(P-X)\cdot(P-A) = 0$$

Now substitute for P:

$$(A - k_pA + k_pB - X)\cdot(-k_pA + k_pB) = 0$$

expand terms:

$$(-k_p+k_p^2)(A\cdot A) + (-k_p^2+k_p-k_p^2)(A\cdot B) + (k_p^2)(B\cdot B) + k_p(A\cdot X) - k_p(B\cdot X) = 0$$

divide through by k_p and simplify:

$$(-1+k_p)(A\cdot A) + (-2k_p+1)(A\cdot B) + k_p(B\cdot B) + (A\cdot X) - (B\cdot X) = 0$$

arrange terms for easier reduction:

$$-(1-k_p)(A\cdot A) + [(-k_p+1)(A\cdot B) - k_p(A\cdot B)] + k_p(B\cdot B) + (A\cdot X) - (B\cdot X) = 0$$

apply distributive property of dot product:

$$(1-k_p)[A\cdot(B-A)] + k_p[(B-A)\cdot B] = [X\cdot(B-A)]$$

collect terms:

$$k_p[[-A\cdot(B-A)] + [(B-A)\cdot B]] + [A\cdot(B-A)] = [X\cdot(B-A)]$$

apply distributive property of dot product again:

$$k_p[(B-A)\cdot(B-A)] = [(X-A)\cdot(B-A)]$$

solve for k_p:

$$Eqn\ 2 \quad k_p = \frac{(B-A)\cdot(X-A)}{(B-A)\cdot(B-A)}$$

Note that if $k_p < 0$, the nearest point to X is A. If $k_p > 1$, it is B. Otherwise solve Eqn 1 with value of k_p from Eqn 2 to get the nearest point.

Appendix B: Sensor Frame UNIX Device Driver Library Functions

Following is a list of the Sensor Frame UNIX device driver C-callable functions:

sf_open(connection)
sf_close(sf_fd)

sf_perror(string)

sf_scale(sf_fd, xmin, xmax, ymin, ymax, zmin, zmax)
sf_query_scale(sf_fd, xmin, xmax, ymin, ymax, zmin, zmax)

sf_enable(sf_fd, types, boolean)
sf_q_enable(sf_fd, types)

sf_queue(sf_fd, types, boolean)
sf_q_queue(sf_fd, types)

sf_poll_once(event_structure)
sf_poll_all(boolean, sf_fd, event_structure)

sf_qtest( )
sf_qread(event_structure)
sf_qflush( )
sf_qadd(event_structure)
sf_qpush(event_structure)

sf_user_input_handler(sf_fd, user_function_address)

sf_filter(sf_fd, filter_structure)
sf_q_filter(sf_fd, filter_id, filter_structure)

sf_toss( )

Appendix C: Contents of Sensor Frame Videotape (VHS Format)

Appendix C consists of a VHS Videotape showing the Following:

- Appendix C-1: The Sensor Frame
- Appendix C-2: The First and Second Prototype Sensor Cubes
- Appendix C-3: The Gesture Based User Interface Prototyping System (GID)
- Appendix C-4: The VideoHarp

Copies of the Videotape are available upon request from Sensor Frame Corporation.

DOI:10.1145/1610252.1610277

ThinSight: A Thin Form-Factor Interactive Surface Technology

By Shahram Izadi, Steve Hodges, Alex Butler, Darren West, Alban Rrustemi, Mike Molloy and William Buxton

ABSTRACT

ThinSight is a thin form-factor interactive surface technology based on optical sensors embedded inside a regular liquid crystal display (LCD). These augment the display with the ability to sense a variety of objects near the surface, including fingertips and hands, to enable multitouch interaction. Optical sensing also allows other physical items to be detected, allowing interactions using various tangible objects. A major advantage of ThinSight over existing camera and projector-based systems is its compact form-factor, making it easier to deploy in a variety of settings. We describe how the ThinSight hardware is embedded behind a regular LCD, allowing sensing without degradation of display capability, and illustrate the capabilities of our system through a number of proof-of-concept hardware prototypes and applications.

1. INTRODUCTION

Touch input using a single point of contact with a display is a natural and established technique for human computer interaction. Research over the past decades,[3] and more recently products such as the iPhone and Microsoft Surface, have shown the novel and exciting interaction techniques and applications possible if multiple simultaneous touch points can be detected.

Various technologies have been proposed for multitouch sensing in this way, some of which extend to detection of physical objects in addition to fingertips. Systems based on optical sensing have proven to be particularly powerful in the richness of data captured and the flexibility they can provide. As yet, however, such optical systems have predominately been based on cameras and projectors and require a large optical path in front of or behind the display. This typically results in relatively bulky systems—something that can impact adoption in many real-world scenarios. While capacitive overlay technologies, such as those in the iPhone and the Dell XT Tablet PC, can support thin form-factor multitouch, they are limited to sensing only fingertips.

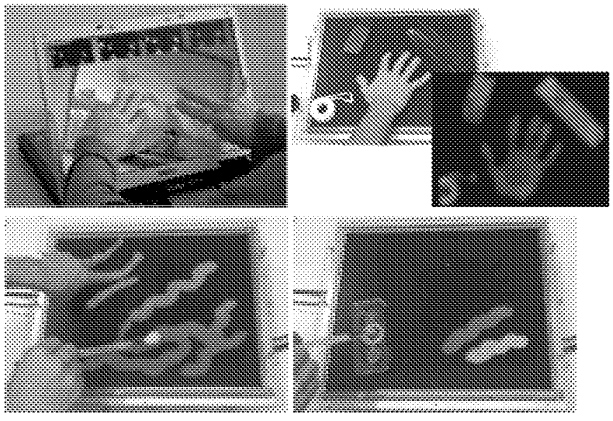
ThinSight is a novel interactive surface technology which is based on optical sensors integrated into a thin form-factor LCD. It is capable of imaging multiple fingertips, whole hands, and other objects near the display surface as shown in Figure 1. The system is based upon custom hardware embedded behind an LCD, and uses infrared (IR) light for sensing without degradation of display capability.

In this article we describe the ThinSight electronics and the modified LCD construction which results. We present two prototype systems we have developed: a multitouch laptop and a touch-and-tangible tabletop (both shown in Figure 1). These systems generate rich sensor data which can be processed using established computer vision techniques to prototype a wide range of interactive surface applications.

As shown in Figure 1, the shapes of many physical objects, including fingers, brushes, dials, and so forth, can be "seen" when they are near the display, allowing them to enhance multitouch interactions. Furthermore, ThinSight allows interactions close-up or at a distance using active IR pointing devices, such as styluses, and enables IR-based communication through the display with other electronic devices.

We believe that ThinSight provides a glimpse of a future where display technologies such as LCDs and organic light emitting diodes (OLEDs) will cheaply incorporate optical sensing pixels alongside red, green and blue (RGB) pixels in

Figure 1. ThinSight brings the novel capabilities of surface computing to thin displays. Top left: photo manipulation using multiple fingers on a laptop prototype (note the screen has been reversed in the style of a Tablet PC). Top right: a hand, mobile phone, remote control and reel of tape placed on a tabletop ThinSight prototype, with corresponding sensor data far right. Note how all the objects are imaged through the display, potentially allowing not only multitouch but tangible input. Bottom left and right: an example of how such sensing can be used to support digital painting using multiple fingertips, a real brush and a tangible palette to change paint colors.



Original versions of this paper appeared in *Proceedings of the 2007 ACM Symposium on User Interface Software and Technology* as "ThinSight: Versatile Multi-touch Sensing for Thin Form-factor Displays" and in *Proceedings of the 2008 IEEE Workshop on Horizontal Interactive Human Computer Systems* as "Experiences with Building a Thin Form-Factor Touch and Tangible Tabletop."

a similar manner, resulting in the widespread adoption of such surface technologies.

2. OVERVIEW OF OPERATION

2.1. Imaging through an LCD using IR light

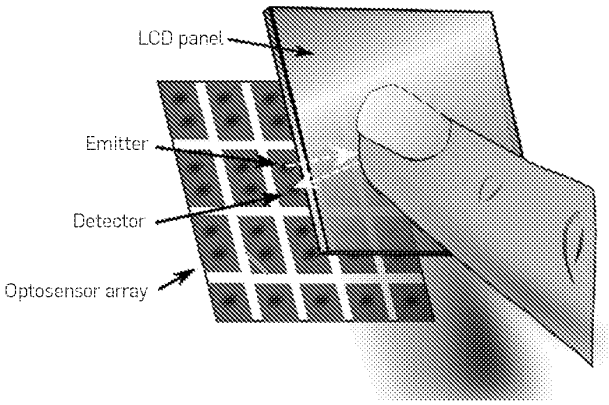
A key element in the construction of ThinSight is a device known as a retro-reflective optosensor. This is a sensing element which contains two components: a light emitter and an optically isolated light detector. It is therefore capable of both emitting light and, at the same time, detecting the intensity of incident light. If a reflective object is placed in front of the optosensor, some of the emitted light will be reflected back and will therefore be detected.

ThinSight is based around a 2D grid of retro-reflective optosensors which are placed behind an LCD panel. Each optosensor emits light that passes right through the entire panel. Any reflective object in front of the display (such as a fingertip) will reflect a fraction of the light back, and this can be detected. Figure 2 depicts this arrangement. By using a suitably spaced grid of retro-reflective optosensors distributed uniformly behind the display it is therefore possible to detect any number of fingertips on the display surface. The raw data generated is essentially a low resolution grayscale "image" of what can be seen through the display, which can be processed using computer vision techniques to support touch and other input.

A critical aspect of ThinSight is the use of retro-reflective sensors that operate in the infrared part of the spectrum, for three main reasons:

- Although IR light is attenuated by the layers in the LCD panel, some still passes through the display.[5] This is largely unaffected by the displayed image.
- A human fingertip typically reflects around 20% of incident IR light and is therefore a quite passable "reflective object."
- IR light is not visible to the user, and therefore does not detract from the image being displayed on the panel.

Figure 2. The basic principle of ThinSight. An array of retro-reflective optosensors is placed behind an LCD. Each of these contains two elements: an emitter which shines IR light through the panel; and a detector which picks up any light reflected by objects such as fingertips in front of the screen.



2.2. Further features of ThinSight

ThinSight is not limited to detecting fingertips in contact with the display; any suitably reflective object will cause IR light to reflect back and will therefore generate a "silhouette." Not only can this be used to determine the location of the object on the display, but also its orientation and shape, within the limits of sensing resolution. Furthermore, the underside of an object may be augmented with a visual mark—a barcode of sorts—to aid identification.

In addition to the detection of passive objects via their shape or some kind of barcode, it is also possible to embed a very small infrared transmitter into an object. In this way, the object can transmit a code representing its identity, its state, or some other information, and this data transmission can be picked up by the IR detectors built into ThinSight. Indeed, ThinSight naturally supports bidirectional IR-based data transfer with nearby electronic devices such as smartphones and PDAs. Data can be transmitted from the display to a device by modulating the IR light emitted. With a large display, it is possible to support several simultaneous bidirectional communication channels in a spatially multiplexed fashion.

Finally, a device which emits a collimated beam of IR light may be used as a pointing device, either close to the display surface like a stylus, or from some distance. Such a pointing device could be used to support gestures for new forms of interaction with a single display or with multiple displays. Multiple pointing devices could be differentiated by modulating the light generated by each.

3. THE THINSIGHT HARDWARE

3.1. The sensing electronics

The prototype ThinSight circuit board depicted in Figure 3 uses Avago HSDL-9100 retro-reflective infrared sensors. These devices are especially designed for proximity sensing —an IR LED emits infrared light and an IR photodiode generates a photocurrent which varies with the amount of incident light. Both emitter and detector have a center wavelength of 940 nm.

A 7 × 5 grid of these HSDL-9100 devices on a regular 10 mm pitch is mounted on custom-made 70 × 50 mm 4-layer printed circuit board (PCB). Multiple PCBs can be tiled together to support larger sensing areas. The IR detectors are interfaced directly with digital input/output lines on a PIC18LF4520 microcontroller.

The PIC firmware collects data from one row of detectors at a time to construct a "frame" of data which is then transmitted to the PC over USB via a virtual COM port. To connect multiple PCBs to the same PC, they must be synchronized to ensure that IR emitted by a row of devices on one PCB does not adversely affect scanning on a neighboring PCB. In our prototype we achieve this using frame and row synchronization signals which are generated by one of the PCBs (the designated "master") and detected by the others ("slaves").

Note that more information on the hardware can be found in the full research publications.[7,10]

Figure 3. Top: the front side of the sensor PCB showing the 7×5 array of IR optosensors. The transistors that enable each detector are visible to the right of each optosensor. Bottom: the back of the sensor PCB has little more than a PIC microcontroller, a USB interface and FETs to drive the rows and columns of IR emitting LEDs. Three such PCBs are used in our ThinSight laptop while there are thirty in the tabletop prototype.

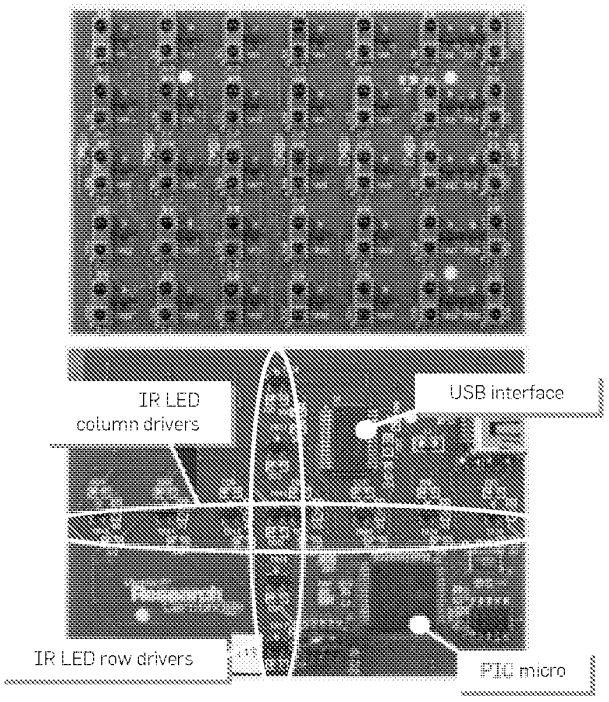
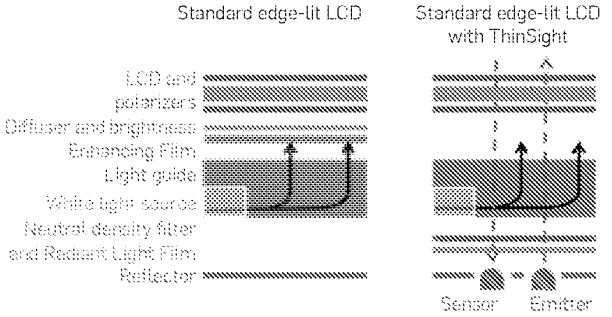


Figure 4. Typical LCD edge-lit architecture shown left. The LCD comprises a stack of optical elements. A white light source is typically located along one or two edges at the back of the panel. A white reflector and transparent light guide direct the light toward the front of the panel. The films help scatter this light uniformly and enhance brightness. However, they also cause excessive attenuation of IR light. In ThinSight, shown right, the films are substituted and placed behind the light guide to minimize attenuation and also reduce noise caused by LCD flexing upon touch. The sensors and emitters are placed at the bottom of the resulting stack, aligned with holes cut in the reflector.



3.2. LCD technology overview

To understand how the ThinSight hardware is integrated into a display panel, it is useful to understand the construction and operation of a typical LCD. An LCD panel is made up of a stack of optical components as shown in Figure 4. At the front of the panel is a thin layer of liquid crystal material which is sandwiched between two polarizers. The polarizers are orthogonal to each other, which means that any light which passes through the first will naturally be blocked by the second, resulting in dark pixels. However, if a voltage is applied across the liquid crystal material at a certain pixel location, the polarization of light incident on that pixel is twisted through 90° as it passes through the crystal structure. As a result it emerges from the crystal with the correct polarization to pass through the second polarizer. Typically, white light is shone through the panel from behind by a backlight and red, green, and blue filters are used to create a color display. In order to achieve a low profile construction while maintaining uniform lighting across the entire display and keeping cost down, the backlight is often a large "light guide" in the form of a clear acrylic sheet which sits behind the entire LCD and which is edge-lit from one or more sides. The light source is often a cold cathode fluorescent tube or an array of white LEDs. To maximize the efficiency and uniformity of the lighting, additional layers of material may be placed between the light guide and the LCD. Brightness enhancing film (BEF) "recycles" visible light at suboptimal angles and polarizations and a diffuser smoothes out any local nonuniformities in light intensity.

3.3. Integration with an LCD panel

We constructed our ThinSight prototypes using a variety of desktop and laptop LCD panels, ranging from 17" to 21". Two of these are shown in Figures 5 and 6. Up to 30 PCBs were tiled to support sensing across the entire surface. In instances where large numbers of PCBs were tiled, a custom hub circuit based on an FPGA was designed to collect and aggregate the raw data captured from a number of tiled sensors and transfer this to the PC using a single USB channel. These tiled PCBs are mounted directly behind the light guide. To ensure that the cold cathode does not cause any stray IR light to emanate from the acrylic light guide, we placed a narrow piece of IR-blocking film between it and the backlight. We cut small holes in the white reflector behind the light guide to coincide with the location of every IR emitting and detecting element.

During our experiments we found that the combination of the diffuser and BEF in an LCD panel typically caused excessive attenuation of the IR signal. However, removing these materials degrades the displayed image significantly: without BEF the brightness and contrast of the displayed image is reduced unacceptably; without a diffuser the image appears to "float" in front of the backlight and at the same time the position of the IR emitters and detectors can be seen in the form of an array of faint dots across the entire display.

To completely hide the IR emitters and detectors we required a material that lets IR pass through it but not visible light, so that the optosensors could not be seen but would operate normally. The traditional solution would be

Figure 5. Our laptop prototype. Top: Three PCBs are tiled together and mounted on an acrylic plate, to give a total of 105 sensing pixels. Holes are also cut in the white reflector shown on the far left. Bottom left: an aperture is cut in the laptop lid to allow the PCBs to be mounted behind the LCD. This provides sensing across the center of the laptop screen. Bottom right: side views of the prototype—note the display has been reversed on its hinges in the style of a Tablet PC.

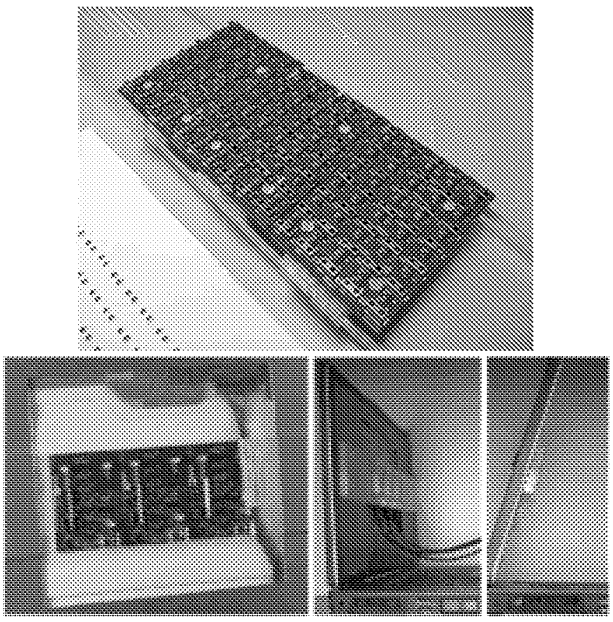
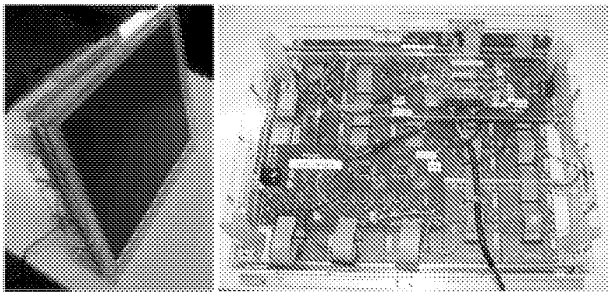


Figure 6. The ThinSight tabletop hardware as viewed from the side and behind. Thirty PCBs (in a 5×6 grid) are tiled with columns interconnected with ribbon cable and attached to a hub board for aggregating data and inter-tile communication. This provides a total of 1050 discrete sensing pixels across the entire surface.



to use what is referred to as a "cold mirror." Unfortunately these are made using a glass substrate which means they are expensive, rigid and fragile and we were unable to source a cold mirror large enough to cover the entire tabletop display. We experimented with many alternative materials including tracing paper, acetate sheets coated in emulsion paint, spray-on frosting, thin sheets of white polythene and mylar. Most of these are unsuitable either because of a lack of IR transparency or because the optosensors can be seen through them to some extent. The solution we settled on was the use of Radiant Light Film by 3M (part number CM500), which largely lets IR light pass through while reflecting visible light without the disadvantages of a true cold mirror. This was combined with the use of a grade "0" neutral density filter, a visually opaque but IR transparent diffuser, to even out the distribution rear illumination and at the same time prevent the "floating" effect. Applying the Radiant Light Film carefully is critical since minor imperfections (e.g. wrinkles or bubbles) are highly visible to the user—thus we laminated it onto a thin PET carrier. One final modification to the LCD construction was to deploy these films *behind* the light guide to further improve the optical properties. The resulting LCD layer stack-up is depicted in Figure 4 right.

Most LCD panels are not constructed to resist physical pressure, and any distortion which results from touch interactions typically causes internal IR reflection resulting in "flare." Placing the Radiant Light Film and neutral density filter behind the light guide improves this situation, and we also reinforced the ThinSight unit using several lengths of extruded aluminum section running directly behind the LCD.

4. THINSIGHT IN OPERATION

4.1. Processing the raw sensor data

Each value read from an individual IR detector is defined as an integer representing the intensity of incident light. These sensor values are streamed to the PC via USB where the raw data undergoes several simple processing and filtering steps in order to generate an IR image that can be used to detect objects near the surface. Once this image is generated, established image processing techniques can be applied in order to determine coordinates of fingers, recognize hand gestures, and identify object shapes.

Variations between optosensors due to manufacturing and assembly tolerances result in a range of different values across the display even without the presence of objects on the display surface. To make the sensor image uniform and the presence of additional incident light (reflected from nearby objects) more apparent, we subtract a "background" frame captured when no objects are present, and normalize relative to the image generated when the display is covered with a sheet of white reflective paper.

We use standard bicubic interpolation to scale up the sensor image by a predefined factor (10 in our current implementation). For the larger tabletop implementation this results in a 350 × 300 pixel image. Optionally, a Gaussian filter can be applied for further smoothing, resulting in a grayscale "depth" image as shown in Figure 7.

4.2. Seeing through the ThinSight display

The images we obtain from the prototype are quite rich, particularly given the density of the sensor array. Fingers and hands within proximity of the screen are clearly identifiable. Examples of images captured through the display are shown in Figures 1, 7 and 8.

Figure 7. The raw ThinSight sensor data shown left and after interpolation and smoothing right. Note that the raw image is a very low resolution, but contains enough data to generate the relatively rich image at right.

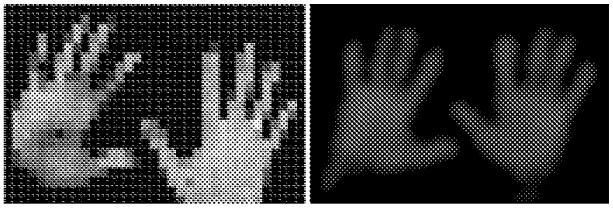
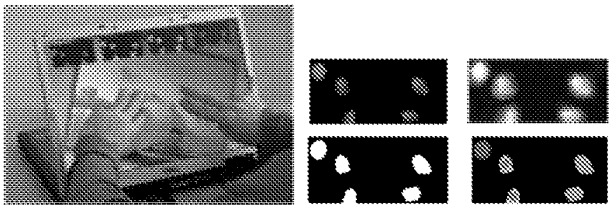


Figure 8. Fingertips can be sensed easily with ThinSight. Left: the user places five fingers on the display to manipulate a photo. Right: a close-up of the sensor data when fingers are positioned as shown at left. The raw sensor data is: (1) scaled-up with interpolation, (2) normalized, (3) thresholded to produce a binary image, and finally (4) processed using connected components analysis to reveal the fingertip locations.



Fingertips appear as small blobs in the image as they approach the surface, increasing in intensity as they get closer. This gives rise to the possibility of sensing both touch and hover. To date we have only implemented touch/no-touch differentiation, using thresholding. However, we can reliably and consistently detect touch to within a few millimeters for a variety of skin tones, so we believe that disambiguating hover from touch would be possible.

In addition to fingers and hands, optical sensing allows us to observe other IR reflective objects through the display. Figure 1 illustrates how the display can distinguish the shape of many reflective objects in front of the surface, including an entire hand, mobile phone, remote control, and a reel of white tape. We have found in practice that many objects reflect IR.

A logical next step is to attempt to uniquely identify objects by placement of visual codes underneath them. Such codes have been used effectively in tabletop systems such as the Microsoft Surface and various research prototypes[12, 28] to support tangible interaction. We have also started preliminary experiments with the use of such codes on ThinSight, see Figure 9.

Active electronic identification schemes are also feasible. For example, cheap and small dedicated electronic units containing an IR emitter can be stuck onto or embedded inside objects that need to be identified. These emitters will produce a signal directed to a small subset of the display sensors. By emitting modulated IR it is possible to transmit a unique identifier to the display.

4.3. Communicating through the ThinSight display

Beyond simple identification, an embedded IR transmitter also provides a basis for supporting richer bidirectional communication with the display. In theory any IR modulation scheme, such as the widely adopted IrDA standard, could be supported by ThinSight. We have implemented a DC-balanced modulation scheme which allows retro-reflective object sensing to occur *at the same time* as data transmission. This required no additions or alterations to the sensor PCB, only changes to the microcontroller firmware. To demonstrate our prototype implementation of this, we built a small embedded IR transceiver based on a low power MSP430 microcontroller, see Figure 10. We encode 3 bits of data in the IR transmitted from the ThinSight pixels to control an RGB LED fitted to the embedded receiver. When the user touches various soft buttons on the ThinSight display, this in turn transmits different 3 bit codes from ThinSight pixels to cause different colors on the embedded device to be activated.

It is theoretically possible to transmit and receive different data simultaneously using different columns on the

Figure 9. An example 2" diameter visual marker and the resulting ThinSight image after processing.

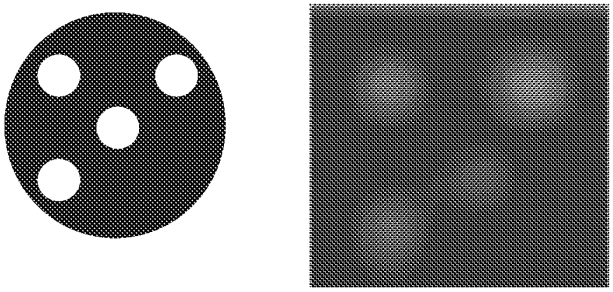
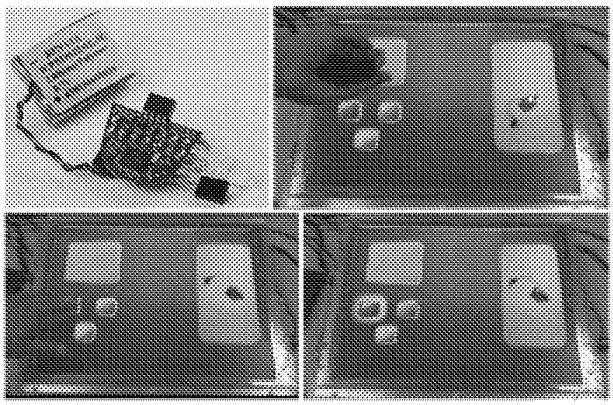


Figure 10. Using ThinSight to communicate with devices using IR. Top left: an embedded microcontroller/IR transceiver/RGB LED device. Bottom left: touching a soft button on the ThinSight display signals the RGB LED on the embedded device to turn red (bottom right). Top right: A remote control is used to signal from a distance the display whichin turn sends an IR command to the RGB device to turn the LED blue.



display surface, thereby supporting spatially multiplexed bidirectional communications with multiple local devices and reception of data from remote gesturing devices. Of course, it is also possible to time multiplex communications between different devices if a suitable addressing scheme is used. We have not yet prototyped either of these multiple-device communications schemes.

4.4. Interacting with ThinSight

As shown earlier in this section, it is straightforward to sense and locate multiple fingertips using ThinSight. In order to do this we threshold the processed data to produce a binary image. The connected components within this are isolated, and the center of mass of each component is calculated to generate representative X, Y coordinates of each finger. A very simple homography can then be applied to map these fingertip positions (which are relative to the sensor image) to onscreen coordinates. Major and minor axis analysis or more detailed shape analysis can be performed to determine orientation information. Robust fingertip tracking algorithms or optical flow techniques[28] can be employed to add stronger heuristics for recognizing gestures.

Using these established techniques, fingertips are sensed to within a few millimeters, currently at 23 frames/s. Both hover and touch can be detected, and could be disambiguated by defining appropriate thresholds. A user therefore need not apply any force to interact with the display. However, it is also possible to estimate fingertip pressure by calculating the increase in the area and intensity of the fingertip "blob" once touch has been detected.

Figure 1 shows two simple applications developed using ThinSight. A simple photo application allows multiple images to be translated, rotated, and scaled using established multifinger manipulation gestures. We use distance and angle between touch points to compute scale factor and rotation deltas. To demonstrate some of the capabilities of ThinSight beyond just multitouch, we have built an example paint application that allows users to paint directly on the surface using both fingertips and real paint brushes. The latter works because ThinSight can detect the brushes' white bristles which reflect IR. The paint application also supports a more sophisticated scenario where an artist's palette is placed on the display surface. Although this is visibly transparent, it has an IR reflective marker on the underside which allows it to be detected by ThinSight, whereupon a range of paint colors are rendered underneath it. The user can change color by "dipping" either a fingertip or a brush into the appropriate well in the palette. We identify the presence of this object using a simple ellipse matching algorithm which distinguishes the larger palette from smaller touch point "blobs" in the sensor image. Despite the limited resolution of ThinSight, it is possible to differentiate a number of different objects using simple silhouette shape information.

5. DISCUSSION AND FUTURE WORK

We believe that the prototype presented in this article is an interesting proof-of-concept of a new approach to multitouch and tangible sensing for thin displays. We have already described some of its potential; here we discuss a number of additional observations and ideas which came to light during the work.

5.1. Fidelity of sensing

The original aim of this project was simply to detect fingertips to enable multi-touch-based direct manipulation. However, despite the low resolution of the raw sensor data, we still detect quite sophisticated object images. Very small objects do currently "disappear" on occasion when they are midway between optosensors. However, we have a number of ideas for improving the fidelity further, both to support smaller objects and to make object and visual marker identification more practical. An obvious solution is to increase the density of the optosensors, or at least the density of IR detectors. Another idea is to measure the amount of reflected light under different lighting conditions—for example, simultaneously emitting light from neighboring sensors is likely to cause enough reflection to detect smaller objects.

5.2. Frame rate

In informal trials of ThinSight for a direct manipulation task, we found that the current frame rate was reasonably acceptable to users. However, a higher frame rate would not only produce a more responsive UI which will be important for some applications, but would make temporal filtering more practical thereby reducing noise and improving subpixel accuracy. It would also be possible to sample each detector under a number of different illumination conditions as described above, which we believe would increase fidelity of operation.

5.3. Robustness to lighting conditions

The retro-reflective nature of operation of ThinSight combined with the use of background substitution seems to give reliable operation in a variety of lighting conditions, including an office environment with some ambient sunlight. One common approach to mitigating any negative effects of ambient light, which we could explore if necessary, is to emit modulated IR and to ignore any nonmodulated offset in the detected signal.

5.4. Power consumption

The biggest contributor to power consumption in ThinSight is emission of IR light; because the signal is attenuated in both directions as it passes through the layers of the LCD panel, a high intensity emission is required. For mobile devices, where power consumption is an issue, we have ideas for improvements. We believe it is possible to enhance the IR transmission properties of an LCD panel by optimizing the materials used in its construction for this purpose—something which is not currently done. In addition, it may be possible to keep track of object and fingertip positions, and limit the most frequent IR emissions to those areas. The rest of the display would be scanned less frequently (e.g. at 2–3 frames/s) to detect new touch points.

One of the main ways we feel we can improve on power consumption and fidelity of sensing is to use a more

sophisticated IR illumination scheme. We have been experimenting with using an acrylic overlay *on top* of the LCD and using IR LEDs for edge illumination. This would allow us to sense multiple touch points using standard Frustrated Total Internal Reflection (FTIR),[5] but not objects. We have, however, also experimented with a material called Endlighten which allows this FTIR scheme to be extended to diffuse illumination, allowing both multitouch and object sensing with far fewer IR emitters than our current setup. The overlay can also serve the dual purpose of protecting the LCD from flexing under touch.

6. RELATED WORK

The area of interactive surfaces has gained particular attention recently following the advent of the iPhone and Microsoft Surface. However, it is a field with over two decades of history.[3] Despite this sustained interest there has been an evident lack of off-the-shelf solutions for detecting multiple fingers and/or objects on a display surface. Here, we summarize the relevant research in these areas and describe the few commercially available systems.

6.1. Camera-based systems

One approach to detecting multitouch and tangible input is to use a video camera placed in front of or above the surface, and apply computer vision algorithms for sensing. Early seminal work includes Krueger's VideoDesk[13] and the DigitalDesk,[26] which use dwell time and a microphone (respectively) to detect when a user is actually touching the surface. More recently, the Visual Touchpad[17] and C-Slate[9] use a stereo camera placed above the display to more accurately detect touch. The disparity between the image pairs determines the height of fingers above the surface. PlayAnywhere[28] introduces a number of additional image processing techniques for front-projected vision-based systems, including a shadow-based touch detection algorithm, a novel visual bar code scheme, paper tracking, and an optical flow algorithm for bimanual interaction.

Camera-based systems such as those described above obviously require direct line-of-sight to the objects being sensed which in some cases can restrict usage scenarios. Occlusion problems are mitigated in PlayAnywhere by mounting the camera off-axis. A natural progression is to mount the camera *behind* the display. HoloWall[18] uses IR illuminant and a camera equipped with an IR pass filter behind a diffusive projection panel to detect hands and other IR-reflective objects in front of it. The system can accurately determine the contact areas by simply thresholding the infrared image. TouchLight[27] uses rear-projection onto a holographic screen, which is also illuminated from behind with IR light. A number of multitouch application scenarios are enabled including high-resolution imaging capabilities. Han[5] describes a straightforward yet powerful technique for enabling high-resolution multitouch sensing on rear-projected surfaces based on FTIR. Compelling multitouch applications have been demonstrated using this technique. The Smart Table[22] uses this same FTIR technique in a tabletop form factor.

The Microsoft Surface and ReacTable[12] also use rear-projection, IR illuminant and a rear mounted IR camera to monitor fingertips, this time in a horizontal tabletop form-factor. These systems also detect and identify objects with IR-reflective markers on their surface.

The rich data generated by camera-based systems provides extreme flexibility. However, as Wilson discusses[28] this flexibility comes at a cost, including the computational demands of processing high resolution images, susceptibility to adverse lighting conditions and problems of motion blur. However, perhaps more importantly, these systems require the camera to be placed at some distance from the display to capture the entire scene, limiting their portability, practicality and introducing a setup and calibration cost.

6.2. Opaque embedded sensing

Despite the power of camera-based systems, the associated drawbacks outlined above have resulted in a number of parallel research efforts to develop a non-vision-based multitouch display. One approach is to embed a multitouch sensor of some kind behind a surface that can have an image projected onto it. A natural technology for this is capacitive sensing, where the capacitive coupling to ground introduced by a fingertip is detected, typically by monitoring the rate of leakage of charge away from conductive plates or wires mounted behind the display surface.

Some manufacturers such as Logitech and Apple have enhanced the standard laptop-style touch pad to detect certain gestures based on more than one point of touch. However, in these systems, using more than two or three fingers typically results in ambiguities in the sensed data. This constrains the gestures they support. Lee et al.[14] used capacitive sensing with a number of discrete metal electrodes arranged in a matrix configuration to support multitouch over a larger area. Westerman[25] describes a sophisticated capacitive multitouch system which generates x-ray-like images of a hand interacting with an opaque sensing surface, which could be projected onto. A derivative of this work was commercialized by Fingerworks.

DiamondTouch[4] is composed of a grid of row and column antennas which emit signals that capacitively couple with users when they touch the surface. Users are also capacitively coupled to receivers through pads on their chairs. In this way the system can identify which antennas behind the display surface are being touched and by which user, although a user touching the surface at two points can produce ambiguities. The SmartSkin[21] system consists of a grid of capacitively coupled transmitting and receiving antennas. As a finger approaches an intersection point, this causes a drop in coupling which is measured to determine finger proximity. The system is capable of supporting multiple points of contact by the same user and generating images of contact regions of the hand. SmartSkin and DiamondTouch also support physical objects, but can only identify an object when a user touches it. Tactex provide another interesting example of an opaque multitouch sensor, which uses transducers to measure surface pressure at multiple touch points.[23]

6.3. Transparent overlays

The systems above share one major disadvantage: they all rely on front-projection for display. The displayed image will therefore be broken up by the user's fingers, hands and arms, which can degrade the user experience. Also, a large throw distance is typically required for projection which limits portability. Furthermore, physical objects can only be detected in limited ways, if object detection is supported at all.

One alternative approach to address some of the issues of display and portability is to use a transparent sensing overlay in conjunction with a self-contained (i.e., not projected) display such as an LCD panel. DualTouch[19] uses an off-the-shelf transparent resistive touch overlay to detect the position of two fingers. Such overlays typically report the average position when two fingers are touching. Assuming that one finger makes contact first and does not subsequently move, the position of a second touch point can be calculated. An extension to this is provided by Loviscach.[16]

The Philips Entertaible[15] takes a different "overlay" approach to detect up to 30 touch points. IR emitters and detectors are placed on a bezel around the screen. Breaks in the IR beams detect fingers and objects. The SMART DViT[22] and HP TouchSmart[6] utilize cameras in the corners of a bezel overlay to support sensing of two fingers or styluses. With such line of sight systems, occlusion can be an issue for sensing.

The Lemur music controller from JazzMutant[11] uses a proprietary resistive overlay technology to track up to 20 touch points simultaneously. More recently, Balda AG and N-Trig[20] have both released capacitive multitouch overlays, which have been used in the iPhone and the Dell XT, respectively. These approaches provide a robust way for sensing multiple fingers touching the surface, but do not scale to whole hand sensing or tangible objects.

6.4. The need for intrinsically integrated sensing

The previous sections have presented a number of multitouch display technologies. Camera-based systems produce very rich data but have a number of drawbacks. Opaque sensing systems can more accurately detect fingers and objects, but by their nature rely on projection. Transparent overlays alleviate this projection requirement, but the fidelity of sensing is reduced. It is difficult, for example, to support sensing of fingertips, hands and objects.

A potential solution which addresses all of these requirements is a class of technologies that we refer to as "intrinsically integrated" sensing. The common approach behind these is to distribute sensing across the display surface, integrating the sensors with the display elements. Hudson[3] reports on a prototype 0.7" monochrome display where LED pixels double up as light sensors. By operating one pixel as a sensor while its neighbors are illuminated, it is possible to detect light reflected from a fingertip close to the display. The main drawbacks are the use of visible illuminant during sensing and practicalities of using LED-based displays. SensoLED uses a similar approach with visible light, but this time based on polymer LEDs and photodiodes. A 1" diagonal sensing polymer display has been demonstrated.[2]

Planar[1] and Toshiba[24] were among the first to develop LCD prototypes with integrated visible light photosensors, which can detect the shadows resulting from fingertips or styluses on the display. The photosensors and associated signal processing circuitry are integrated directly onto the LCD substrate. To illuminate fingers and other objects, either an external light source is required—impacting on the profile of the system—or the screen must uniformly emit bright visible light—which in turn will disrupt the displayed image.

The motivation for ThinSight was to build on the concept of intrinsically integrated sensing. We have extended the work above using invisible (IR) illuminant to allow simultaneous display and sensing, building on current LCD and IR technologies to make prototyping practical in the near term. Another important aspect is support for much larger thin touch-sensitive displays than is provided by intrinsically integrated solutions to date, thereby making it more practical to prototype multitouch applications.

7. CONCLUSION

In this article we have described a new technique for optically sensing multiple objects, including fingertips, through thin form-factor displays. Optical sensing allows rich "camera-like" data to be captured by the display and this is processed using computer vision techniques. This supports new types of human computer interfaces that exploit zero-force multi-touch and tangible interaction on thin form-factor displays such as those described in Buxton.[3] We have shown how this technique can be integrated with off-the-shelf LCD technology, making such interaction techniques more practical and deployable in real-world settings.

We have many ideas for potential refinements to the ThinSight hardware, firmware, and PC software. In addition to such incremental improvements, we also believe that it will be possible to transition to an integrated "sensing and display" solution which will be much more straightforward and cheaper to manufacture. An obvious approach is to incorporate optical sensors directly onto the LCD backplane, and as reported earlier early prototypes in this area are beginning to emerge.[24] Alternatively, polymer photodiodes may be combined on the same substrate as polymer OLEDs[2] for a similar result. The big advantage of this approach is that an array of sensing elements can be combined with a display at very little incremental cost by simply adding "pixels that sense" in between the visible RGB display pixels. This would essentially augment a display with optical multitouch input "for free," enabling truly widespread adoption of this exciting technology.

Acknowledgments

We thank Stuart Taylor, Steve Bathiche, Andy Wilson, Turner Whitted and Otmar Hilliges for their invaluable input. ◙

References

1. Abileah, A., Green, P. Optical sensors embedded within AMLCD panel: design and applications. In *Proceedings of EDT'07* (San Diego, California, August 04, 2007), ACM, New York, NY.
2. Börgi, L. et al. Optical proximity and touch sensors based on monolithically integrated polymer photodiodes and polymer LEDs. *Org. Electron. 7* (2006).
3. Buxton, B. Multi-Touch Systems That I Have Known and Loved (2007), http://www.billbuxton.com/multitouchOverview.html.
4. Dietz, P., Leigh, D. DiamondTouch: a multi-user touch technology. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology* (Orlando, FL, Nov. 11–14, 2001), UIST '01. ACM, NY, 219–226.
5. Han, J.Y. Low-cost multi-touch sensing through frustrated total internal reflection. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology* (Seattle, WA, Oct. 23–26, 2005), UIST '05. ACM, NY, 115–118.
6. HP TouchSmart, http://www.hp.com/united-states/campaigns/touchsmart/.
7. Hodges, S., Izadi, S., Butler, A., Rrustemi, A., Buxton, B. ThinSight: Versatile multi-touch sensing for thin form-factor displays. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology* (Newport, RI, Oct. 07–10, 2007), UIST '07. ACM, NY, 259–268.
8. Hudson, S.E. 2004. Using light emitting diode arrays as touch-sensitive input and output devices. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology* (Santa Fe, NM, Oct. 24–27, 2004), UIST '04. ACM, NY, 287–290.
9. Izadi, S. et al. C-Slate: A multi-touch and object recognition system for remote collaboration using horizontal surfaces. In *IEEE Workshop on Horizontal Interactive Human-Computer Systems* (Rhode Island, Oct. 2007), IEEE Tabletop 2007, IEEE, 3–10.
10. Izadi, S. et al. Experiences with building a thin form-factor touch and tangible tabletop. In *IEEE Workshop on Horizontal Interactive Human-Computer Systems* (Amsterdam, Holland, Oct. 2008), Tabletop 2008, IEEE, 181–184.
11. JazzMutant Lemur. http://www.jazzmutant.com/lemur_overview.php.
12. Jordà, S., Geiger, G., Alonso, M., Kaltenbrunner, M. The reacTable: Exploring the synergy between live music performance and tabletop tangible interfaces. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction* (Baton Rouge, LA, Feb. 15–17, 2007), TEI '07. ACM, NY, 139–146.
13. Krueger, M. Artificial Reality 2. Addison-Wesley Professional (1991). ISBN 0-201-52260-8.
14. Lee, S., Buxton, W., Smith, K.C. S. A multi-touch three dimensional touch-sensitive tablet. *SIGCHI Bull 16*, 4 (Apr. 1985), 21–25.
15. van Loenen, E. et al. Entertaible: a solution for social gaming experiences. In *Tangible Play Workshop, TUI Conference* (2007).
16. Loviscach, J. Two-finger input with a standard touch screen. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology* (Newport, RI, USA, October 07–10, 2007). UIST '07. ACM, New York, NY, 169–172.
17. Malik, S., Laszlo, J. Visual touchpad: A two-handed gestural input device. In *Proceedings of the 6th International Conference on Multimodal Interfaces* (State College, PA, Oct. 13–15, 2004), ICMI '04. ACM, NY, 289–296.
18. Matsushita, N., Rekimoto, J. 1997. HoloWall: Designing a finger, hand, body, and object sensitive wall. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology* (Banff, Alberta, Canada, Oct. 14–17, 1997), UIST '97. ACM, NY, 209–210.
19. Matsushita, N., Ayatsuka, Y., Rekimoto, J. 2000. Dual touch: A two-handed interface for pen-based PDAs. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology* (San Diego, California, US, Nov. 06–08, 2000), UIST '00. ACM, New York, NY, 211–212.
20. N-trig, duo-touch sensor, http://www.n-trig.com/.
21. Rekimoto, J. SmartSkin: An infrastructure for freehand manipulation on interactive surfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Changing Our World, Changing Ourselves* (Minneapolis, MN, Apr. 20–25, 2002), CHI '02. ACM, NY, 113–120.
22. Smart Technologies, http://smarttech.com (2009).
23. Tactex Controls Inc. Array Sensors. http://www.tactex.com/products_array.php.
24. Matsushita, T. LCD with Finger Shadow Sensing, http://www3.toshiba.co.jp/tm_dsp/press/2005/05-09-29.htm.
25. Westerman, W. Hand Tracking, Finger Identification and Chordic Manipulation on a Multi-Touch Surface. PhD thesis, University of Delaware (1999).
26. Wellner, P. Interacting with Paper on the Digital Desk. *CACM 36*, 7 (1993) 86–96.
27. Wilson, A.D. TouchLight: An imaging touch screen and display for gesture-based interaction. In *Proceedings of the 6th International Conference on Multimodal Interfaces* (State College, PA, Oct. 13–15, 2004), ICMI '04. ACM, NY, 69–76.
28. Wilson, A.D. PlayAnywhere: A compact interactive tabletop projection-vision system. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology* (Seattle, WA, Oct. 23–26, 2005), UIST '05. ACM, NY, 83–92.

Shahram Izadi, Steve Hodges, Alex Butler, Darren West, Alban Rrustemi, Mike Molloy, and William Buxton ({shahrami, shodges, dab}@microsoft.com), Microsoft Research Cambridge, U.K.

VIDEOPLACE--An Artificial Reality

Myron W. Krueger, Thomas Gionfriddo and Katrin Hinrichsen
Computer Science Department
University of Connecticut
Storrs, Connecticut 06268

Abstract

The human-machine interface is generalized beyond traditional control devices to permit physical participation with graphic images. The VIDEOPLACE System combines a participant's live video image with a computer graphic world. It also coordinates the behavior of graphic objects and creatures so that they appear to react to the movements of the participant's image in real-time. A prototype system has been implemented and a number of experiments with aesthetic and practical implications have been conducted.

Introduction

This paper describes a number of experiments in alternate modes of human-machine interaction. The premise is that interaction is a central, not peripheral, issue in computer science. We must explore this domain for insight as well as immediate application. It is as important to suggest new applications as it is to solve the problems associated with existing ones. Research should anticipate future practicality and not be bound by the constraints of the present.

Unlike most computer science professionals, who have been content to rely on traditional computer languages and the hundred year old keyboard as the means of input, designers of graphic systems have long recognized the importance of the human-machine interface. Even so, most innovations, including the light pen, joy stick, data tablet and track ball have been dictated by the minimum needs of immediate graphics applications.

There have been few experiments motivated by a purely intellectual desire to explore the means through which people and machines might interact, independent of specific applications. One such novel approach was Ivan Sutherland's head-mounted stereo displays which sensed the orientation of the viewer's head and displayed what would be seen in a simulated graphic environment from each position. [SUTH68]

Another unique approach was taken with the GROPE system at the University of North Carolina. It provided force feedback to a remote manipulator that could be used to pick up graphic blocks. [BATT72] In addition, there have been the well funded efforts of the Architecture Machine Group at MIT, including the Dataland project, MovieMap and "Put that there". [BOLT79, 80, 81], [LIPP80]

Finally, my work in Responsive Environments, beginning in 1969 and continuing to the present, has allowed a participant's movements around a room to be translated into actions in a projected graphic scene generated by the computer. [KRUE77, 83]

This paper describes one of my early experiments with Responsive Environments, the VIDEOPLACE project currently under development and applications planned for the near future.

Abstract Versus Concrete Intelligence

The observation underlying this research is that there are two quite different aspects of human intelligence. The first is the logical, deductive, explicitly rational process that we associate with abstract symbolic reasoning. While the technically inclined take great pride in this skill, a large fraction of the population has no interest in developing it. The second is the facility for understanding, navigating and manipulating the physical world. This ability is part of our basic human heritage.

As a greater percentage of the population becomes involved in the use of computers, it is natural to expect the manner of controlling computers to move away from the programming model and closer to the perceptual process we use to accomplish our goals in the physical world.

Early Responsive Environments

In 1969, I began to explore the idea of physical participation in a graphic world using the paradigm of a Responsive Environment. A Responsive Environment is an empty room in which a single participant's movements are perceived by the computer which responds through visual displays and electronic sound. Since 1970, video projection of computer graphic images has been used to provide the visual response.

PSYCHIC SPACE

In PSYCHIC SPACE, a Responsive Environment created in 1971, sensing of the participant's behavior was accomplished through a grid of hundreds of pressure sensors placed in the floor. As the participant walked around the room, the computer scanned the floor and detected the movement of his feet. The person's position in the room was then used to control an interaction in a graphic scene which was displayed on an 8'x10' rear-screen video projection.

In one PSYCHIC SPACE interaction, the participant's movements in the room were used to control the movements of a symbol on the video screen. After a few minutes, allocated for exploration of this phenomenon, a second symbol appeared. The participant, inevitably wondering what would happen if he walked his symbol over to the intruder's position, moved until the two symbols coincided. At that point, the second symbol disappeared and a maze appeared with the participant's symbol at the starting point. (Fig. 1)

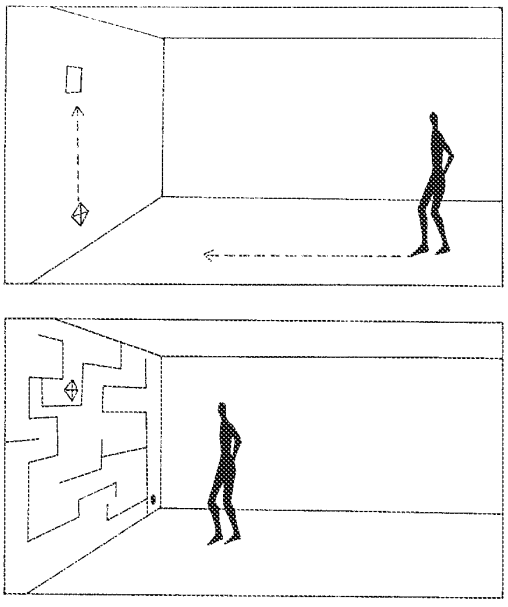


Fig. 1

Again, inevitably, the participant tried to walk the maze. However, after a few minutes the participant would realize that since there were no physical boundaries in the room, there was nothing to prevent cheating. When this realization struck, the participant, typically with some ceremony, raised his foot and planted it on the other side of one of the graphic boundaries. However, the maze program had anticipated this response and stretched that boundary elastically. Subsequent cheating attempts were greeted with a number of other gambits. The participant's symbol might fall apart; the whole maze could move; or, a specific boundary would disappear and a new one would appear elsewhere. By the end of the experience, the participant could have encountered as many as forty different variations on the maze theme.(KRUE77, 83)

PSYCHIC SPACE was presented as an aesthetic work in the Union Gallery at the University of Wisconsin. It suggested a new art form in which the participant's expectations about cause and effect could be used to create interesting and entertaining experiences, quite unlike anything that existed at that time and still different in spirit from the video games of today.

VIDEOPLACE

Concept

In 1970, I combined computer graphic images, created by an artist using a data tablet, with the live image of people. Observing their reactions to this computer graphic graffiti led to the formulation of the VIDEOPLACE concept.

VIDEOPLACE is a computer graphic environment in which the participant sees his or her live image projected on a video screen. It may be alone on the screen, or there may be images of other people at different locations. In addition, there may be graphic objects and creatures which interact with the participant's image.

When people see their image displayed with a graphic object, they feel a universal and irresistible desire to reach out and touch it. (Fig. 2) Furthermore, they expect the act of touching to affect the graphic world. By placing each participant against a neutral background, it is possible to digitize the image of his silhouette and to recognize the moment when it touches a graphic object. The system can then cause the object to move, apparently in response to the participant's touch.

It is also possible for the computer to analyze the participant's image and to alter its appearance on the screen. By either analog or digital techniques, the participant's image can be scaled and rotated and placed anywhere on the screen. Thus, in principle, the participant could climb graphic mountains, swim in graphic seas, or defy gravity and float around the screen. The potential for new forms of interaction within this model is very rich, with certain application as an art form, likely application in education and telecommunication, as well as arguable application for general human-machine interaction.

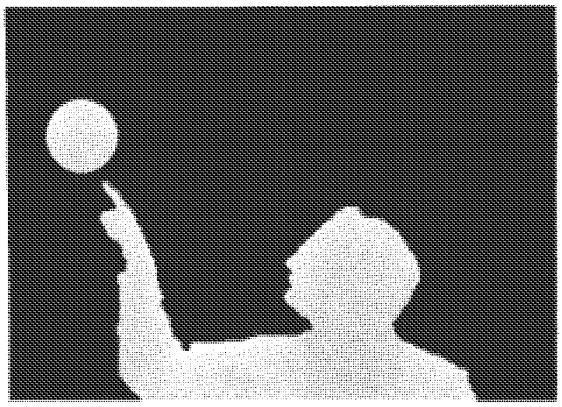


Fig. 2

Prototype System

A prototype VIDEOPLACE system has been constructed. Since understanding the movements of the participant's image appeared to be the most challenging issue, much of the initial effort was focussed on solving this problem. Graceful mechanisms for specifying and controlling the desired interactive relationships have been developed. To date, only very simple graphics have been used because of the very modest resources available and the fact that until recently commercial equipment did not emphasize high speed manipulation of raster data. To a great extent, we have worked with graphic hardware of our own construction which provides a number of features important to our interactions. In addition, we have recently acquired three Silicon Graphics workstations, which will greatly enhance our ability to create and manipulate realistic three-dimensional scenes.

The software employed to control the interactions is quite unusual. We believe the methodology is of general interest for graphics and other real-time applications. We treat the overall system as a model of a real-time intelligence. It is divided into two major components: the Cognitive System which runs on a VAX11/780 and the Reflex System which consists of a group of closely coupled dedicated processors operating on a specialized bus structure.

The Reflex System handles instantaneous decision making. The plan is for the Cognitive System to monitor the events in the Environment and the decisions of the Reflex System, in order to understand what is happening in semantic terms and then to make strategic decisions that will alter the future character of the interaction.

Although all of the communications are established, the Cognitive System is not yet performing this monitoring function. However, it has totally altered the programming process. Instead of writing a separate program for each interaction, we describe the desired causal relationships in conceptual terms. This conceptual representation is then translated into a form that the Reflex System can interpret in real-time. The long term objective is to develop an online real-time intelligence that understands the participant's behavior and the interaction in human terms.

CRITTER, A VIDEOPLACE Interaction

In one current interaction, the participant is joined by a single graphic creature on the screen. The behavior of this creature is very complex and context dependent. The intent is to produce the sensation of an intelligent and witty interaction between creature and the participant.

Initially, the creature sees the participant and chases his image about the screen. If the participant moves rapidly towards it, the creature, nicknamed CRITTER, moves to avoid contact. If the human holds out a hand, CRITTER will land on it and climb up the person's silhouette. As it climbs, its posture adapts to the contour of the human form. When it finally scales the person's head, it does a triumphant jig.

Once this immediate goal is reached, the creature considers the current orientation of the person's arms. If one of the hands is raised, it does a flying somersault and lands on that hand. If the hand is extended to the side but not above the horizontal, CRITTER dives off the head, rolls down the arm, grabs the finger and dangles from it. When the person shakes his hand, CRITTER falls off and dives to the bottom of the screen. Each time it climbs to the top of the participant's head, it is in a different state and is prepared to take a different set of actions. (Fig. 3a-h)

The CRITTER experience will soon be enhanced in a number of ways. Hardware has been built that shrinks the human image down to CRITTER size. The smaller size increases the number of relationships that can exist between the participant and the creature. Simple graphic scenes are being added. Both human and graphic entities will interact with these graphic props by moving among them, climbing them or hiding behind them. The new displays will provide a capability for three-dimensional scenery which can be navigated in real-time.

Practical Applications

The interface described is a deliberately informal one. The resemblance to video games might seem frivolous to the hard-nosed computer scientist used to catering to the needs of government agencies and three letter companies. However, games are a multi-billion dollar industry and by that measure practical. More importantly, games provide an extremely compelling interface whose advantages should be considered for more standard applications. Therefore, before adapting the techniques described to fit a more familiar practical context, we will examine their potential in the current VIDEOPLACE environment.

Computer Aided Instruction

In our culture, education is a sedentary activity imposed on naturally active creatures. Sti-

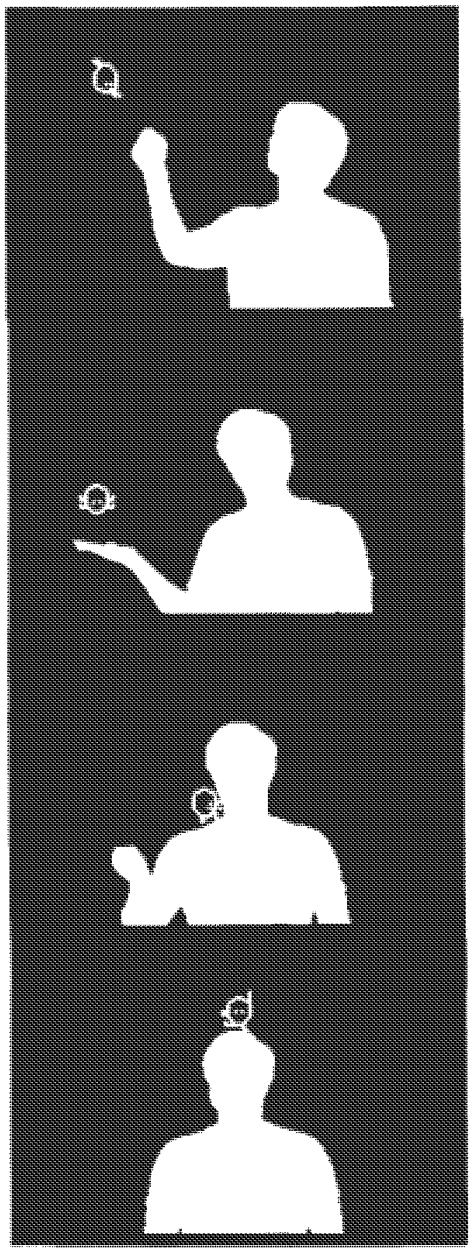


Fig. 3a-d

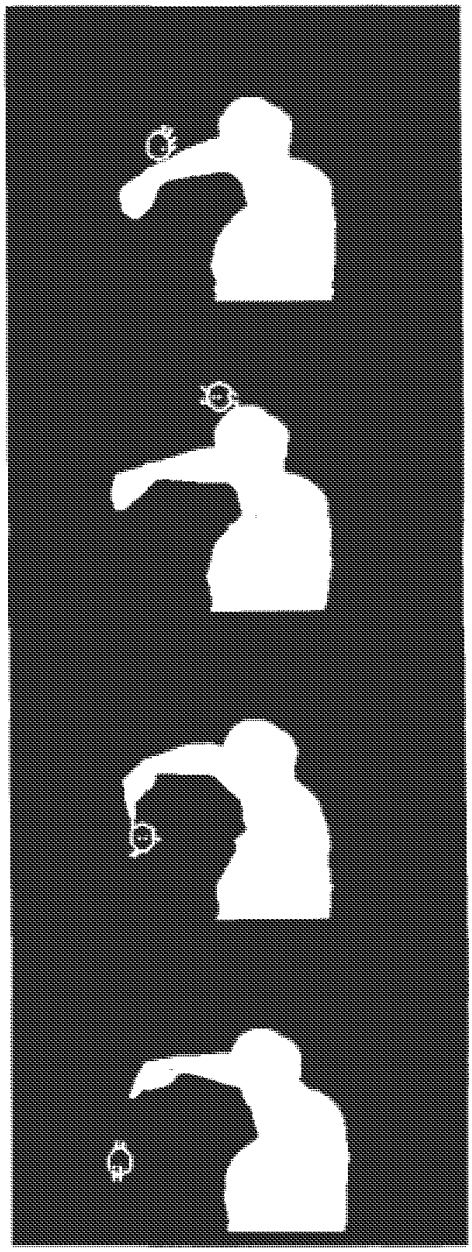


Fig. 3e-h

fling this energy is the first task of every elementary school teacher. As an alternative, VIDEOPLACE could be used to create a physically active form of Computer Aided Instruction in which the computer is used not to teach traditional material, but to alter what, as well as how, we teach.

In one proposal, which I first made formally to NSF in 1975, elementary school children were to be placed in the role of scientists landing on an alien planet. VIDEOPLACE would be used to define an artificial reality in which the laws of cause and effect are composed by the programmer. The task of the children would be to discover these laws. They would enter the Environment singly, interact with it and make individual observations of its rules. Under the guidance of their teacher, they would discuss their experiences and present their opinions. They would compare notes and formulate theories. Since each child would behave differently, in the VIDEOPLACE, individuals would have unique experiences and produce conflicting theories. They would argue and then revisit the Environment, executing critical experiments to resolve which theories were correct under which conditions. Thus, students would learn how obser-

vation leads to hypothesis formation, prediction, testing and reformulation. They would learn the process of scientific thought rather than memorizing vocabulary and performing mechanical calculations as they often do now.

Telecommunications

VIDEOPLACE was originally conceived and implemented as a telecommunications environment allowing people in different places to share a common video experience. While the possibility of such graphic interaction may seem unnecessary to communication, we should remember two points. First, a hundred years ago the telephone seemed to have no advantage over the telegraph which could transmit the content of messages equally well. Second, since communication between friends or business associates is not limited to words, it is clearly desirable to provide a place in which individuals who are geographically separated can share a common visual environment.

An example of this use of VIDEOPLACE is described in Artificial Reality (KRUE 83). A two-way computer graphic and live video telecommunications link was used to solve an engineering problem. In this experiment, the graphic images from two computers were viewed by television and combined by standard video techniques. Each participant pointed to the image on his local screen. The images of both of the participants' hands were combined with the graphic image, allowing them to gesture as naturally as if they were sitting together at a table. For the signal processing task at hand, the communication was complete.

Computing by Hand

A number of technologies are competing for space on the modern professional's desk. Telephones, answering devices, modems and computer terminals with touch screens are all candidates for the desk top. From the user's point of view, an empty desk is preferable. Two technology trends augur the removal of the computer terminal from the desk's surface. First, the keyboard will ultimately succumb to voice input. Second, flat screen displays of adequate resolution already exist. They are likely to be placed on a wall behind the desk, not on it, making touch screen input awkward.

The VIDEOPLACE techniques described in this paper can be used to duplicate any touch screen capability. A video camera pointed down at a desk surface can be used to create a VIDEODESK environment that will have several advantages over a touch screen.

In the VIDEOPLACE system, the user's hands can be used for any traditional graphics application. Since the system can detect when a person's hand touches a particular object, pointing and selection can be controlled. Similarly, a finger can be used to position the selected object in a design. A finger can also be used to draw on the screen, for example, to connect components in a logic design. We have already implemented simple menu selection, typing and finger painting systems. (Fig. 4)

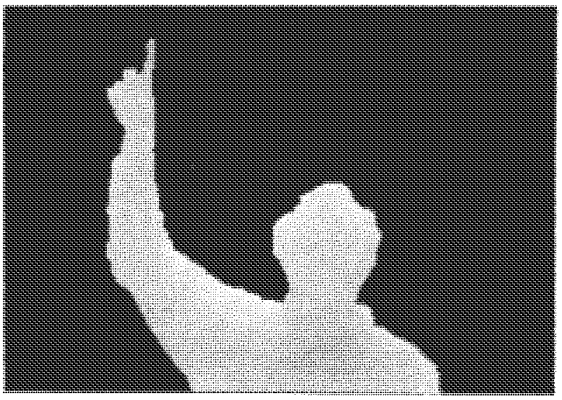


Fig. 4

Video input offers more than a simple alternative to other pointing techniques. With the exception of the recent development of three-dimensional input devices, virtually all pointing devices are limited to two degrees of freedom. However, on the VIDEODESK, two hands can be used in concert to increase the user's bandwidth. In fact, in one common graphic application, it is easy to see the use for eight or more degrees of freedom. B-spline curves are used widely to design car bodies, ships hulls, turbine blades, etc. These curves are defined in terms of a relatively small number of control points. The user controls the shape of the curve by moving these points. With existing input devices only one point can be moved at a time. On the VIDEODESK, the tips of the index fingers and thumbs can be used to manipulate four control points simultaneously. (Fig. 5)

Conclusion

VIDEOPLACE is not so much a solution to existing problems, as an effort to stretch our thinking about the human-machine interface. We have already entered an era where most of the people using computers are no longer programmers in the traditional sense. We can look to a day when most of the people interacting with computers will not be users in the current sense.

Since computers are becoming less expensive than the people who use them, we can expect that as much computing power will be dedicated to providing a pleasing human-machine interface as is actually used to accomplish the user's application. As computer interaction becomes the dominant mode of performing work and transacting business, it becomes a significant ingredient in our quality of life. It is time to give the aesthetics of human-machine interaction serious thought.

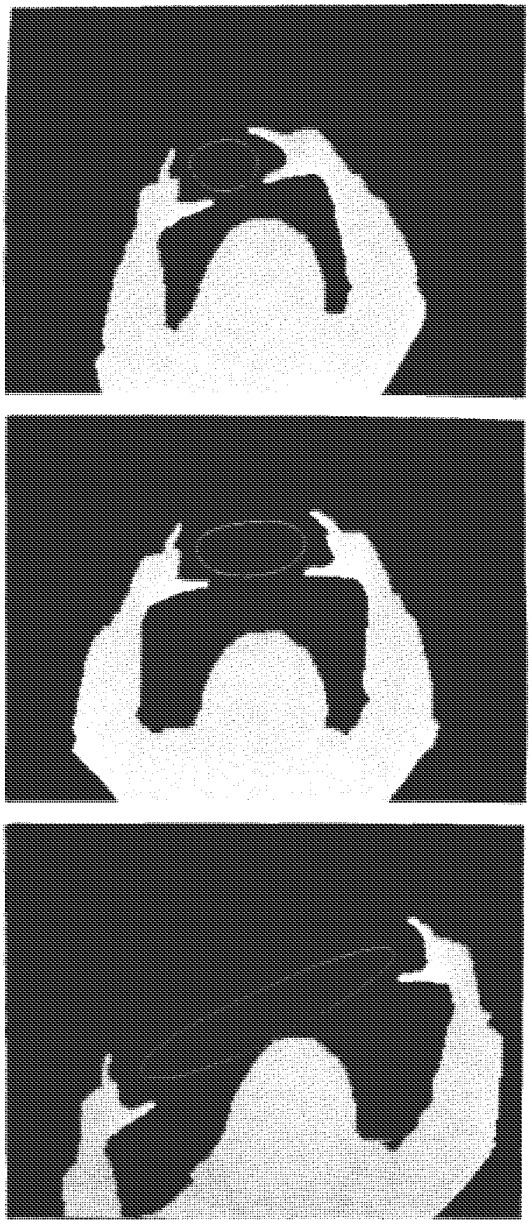


Fig. 5

Bibliography

[BATT72] Batter, James J. & Brooks, Frederick P. Jr., "GROPE-1," IFIPS 71, pp. 759-765.
[BOLT81] Bolt, Richard A., "Gaze Orchestrated Dynamic Windows." SIGGRAPH 81. pp. 32-42.
[BOLT80] Bolt, Richard A., "Put That There: Voice and Gesture at the Graphic Interfaces." SIGGRAPH 80. pp. 262-270.
[BOLT79] Bolt, Richard A., "Spatial Data Management." MIT 1979.
[CROC81] Crockett, D.W., "Triform Modules." CACM. Vol 24, No 6. pp. 344-350.
[CULL82] Cullingford, R.E., Krueger, M.W., Selfridge, M., "Automated Explanations in a CAD System." IEEE Transactions on SMC. January 1982.
[HAYE77] Hayes-Roth, F. and Lesser, V., "Focus of Attention in the Hearsay-II Speech Understanding System." IJCAI 77. pp. 27-35.
[KRUE83] Krueger, M.W., Artificial Reality. Addison-Wesley, 1983. 312 pp.
[KRUE81] Krueger, M.W., Cullingford, R.E. & Bellavance, D.A., "Control Issues in a CAD System with Expert Knowledge." Proceedings SMC. Oct. 1981.
[KRUE77] Krueger, M.W., "Responsive Environments." AFIPS 1977. 46:423-429.
[KRUE75] Krueger, M.W., IVAM, Annual Reports of NOAA Contract #5-315156 1975-1978.
[LIPP80] Lippman, Andrew. "Movie Maps: An Application of the Optical Disc to Computer Graphics." SIGGRAPH 81. pp. 109-120.
[SCHA77] Schank, R. and Abelson, R.P., "Scripts, Plans, Goals and Understanding." Erlbaum Press, 1977.
[SUTH68] Sutherland, Ivan. 1968. "A Head-Mounted Three-Dimensional Display." FJCC AFIPS 33-1:7575-764.

Brown, E., Buxton, W. & Murtagh, K. (1990). Windows on tablets as a means of achieving virtual input devices. In D. Diaper et al. (Eds), *Human-Computer Interaction - INTERACT '90.* Amsterdam: Elsevier Science Publishers B.V. (North-Holland), 675-681.

WINDOWS ON TABLETS AS A MEANS OF ACHIEVING VIRTUAL INPUT DEVICES

Ed BROWN, William A.S. BUXTON and Kevin MURTAGH

Computer Systems Research Institute,
University of Toronto,
Toronto, Ontario,
Canada M5S 1A4

Users of computer systems are often constrained by the limited number of physical devices at their disposal. For displays, window systems have proven an effective way of addressing this problem. As commonly used, a window system partitions a single physical display into a number of different virtual displays. It is our objective to demonstrate that the model is also useful when applied to input.

We show how the surface of a single input device, a tablet, can be partitioned into a number of virtual input devices. The demonstration makes a number of important points. First, it demonstrates that such usage can improve the power and flexibility of the user interfaces that we can implement with a given set of resources. Second, it demonstrates a property of tablets that distinguishes them from other input devices, such as mice. Third, it shows how the technique can be particularly effective when implemented using a touch sensitive tablet. And finally, it describes the implementation of a prototype an "input window manager" that greatly facilitates our ability to develop user interfaces using the technique.

The research described has significant implications on direct manipulation interfaces, rapid prototyping, tailorability, and user interface management systems.

1. INTRODUCTION

A significant trend in user interface design is away from the discrete, serial nature of what we might call a digital approach, towards the continuous, spatial properties of an analogue approach.

Direct Manipulation systems are a good example of this trend. With such systems, controls and functions (such as scroll bars, buttons, switches and potentiometers) are represented as graphical objects which can be thought of as virtual devices. A number of these are illustrated in Fig. 1.
The impression is that of a number of distinct devices, each with its own specialized function, and occupying its own dedicated space. While powerful, the impression is an illusion, since virtually all interactions with these devices is via only one or two physical devices: the keyboard and the mouse.

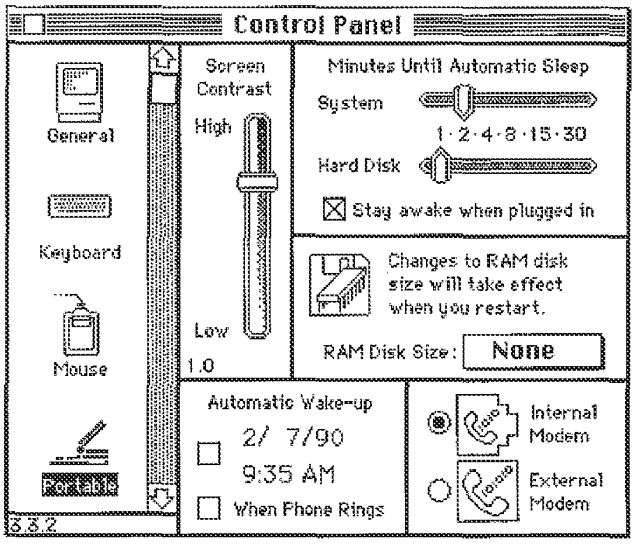


Figure 1: Virtual Devices in the Macintosh Control Panel

The figure shows graphical objects such as potentiometers, radio buttons and icons. Each functions as a distinct device. Interaction, however, is via one of two physical devices: the mouse or keyboard.

The strength of the illusion, however, speaks well for its effectiveness. Nevertheless, this paper is rooted in a belief that direct manipulation systems can be improved by expanding the design space to better afford turning this illusion into reality. Distinct controls for specific functions, provide the potential to improve the directness of the user's access (such as through decreased homing time and exploiting motor memory). Input functions are moved from the display to the work surface, thereby freeing up valuable screen real-estate. Because they are dedicated, physical controls can be specialized to a particular function, thereby providing the possibility to improve the quality of the manipulation .

While one may agree with the general concepts being expressed, things generally break down when we try to put these ideas into practice. Given the number of different functions and virtual devices that are found in typical direct manipulation systems, having a separate physical controller for each would generally be unmanageable. Our desks (which are already crowded) would begin to look like an aircraft cockpit or a percussionist's studio. Clearly, the designer must be selective in what functions are assigned to dedicated controllers. But even then, the practical management of the resources remains a problem.

The contribution of the current research is to describe a way in which this approach to designing the control structures can be supported. To avoid the explosion of input transducers, we introduce the notion of virtual input devices that are spatially distinct. We do so by partitioning the surface of one physical device into a number of separate regions, each of which emulates the function of a separate controller. This is analogous on the input side to windows on displays.
We highlight the properties that are required of the input technology to support such windows, and discuss why certain types of touch tablets are particularly suited for this type of interaction.

Finally, we discuss the functionality that would be required by a user interface management system to support the approach. We do so by describing the implementation of a working prototype system.

2. RELATIONSHIP TO PREVIOUS PRACTICE

The idea of virtual devices is not new. One of the most innovative approaches was the virtual keyboard developed by Ken Knowlton (1975, 1977a,b) at Bell Laboratories. Knowlton developed a system using half-silvered mirrors to permit the functionality of keyboards to be dynamically reconfigured. Partitioning a tablet surface into regions is also not new. Tablet mounted menus, as seen in many CAD systems, are one example of existing practice.

Our contribution:

- makes this model explicit
- develops it beyond current common practice
- develops some of the design issues (such as input transducers)
- demonstrates its utility
- and presents a prototype User Interface Management (UIMS) utility to support its use.

3. RELEVANT PROPERTIES OF INPUT TRANSDUCERS

The technique of "input windows" involves a mapping of different functions to distinct physical locations in the control space. This mapping can only be supported by input transducers that possess the following two properties:

- *Position Sensitive:* They must give absolute coordinates defining position, rather than a measure of motion (as with mice).
- *Fixed Planar Coordinate System:* Position must be measured in terms of a two dimensional Cartesian space.

Hence digitizing tablets will work, but mice, trackballs, and joysticks will not. Within the class of devices which meet these two criteria (including light pens, graphics tablets, touch screens), touch technologies (and especially touch tablets) have noteworthy potential.

Control systems that employ multiple input devices generally have two important properties:

- *Eyes-Free Operation:* Sufficient kinesthetic feedback is provided to permit the operation of the control, leaving the eyes free to perform some other task, such as monitoring a display.
- *Simultaneous Access:* More than one device can be operated at a time, as in driving a car (steering wheel and gear lever) or operating an audio mixing console (where multiple faders might be accessed simultaneously).

In many design situations, these properties are useful, if not essential. In mixing a colour in a paint program, one might assign a potentiometer to each of hue, saturation and value. In performing the task, it is reasonable to expect that the artist generally is better served by focusing visual attention on the colour produced rather than the potentiometers controlling its components values. Driving a car would be impossible if operating the steering wheel required visual attention.

Simultaneous access is also important in many situations. Within the domain of human-computer interaction, for example, Buxton and Myers (1986) demonstrate benefits in tasks similar to those demanded in text editing and

CAD.

4. THE AFFORDANCES OF TOUCH TABLETS

Touch tablets are interesting in that they can be designed and employed in such a way as to afford eyes-free operation and simultaneous access. As well, they can meet our constraints of providing absolute position information in a planar coordinate system. In this, they are rare among input transducers.

The primary attribute of touch technologies that affords eyes-free operation is their having no intermediate hand-held transducer (such as a stylus or puck). Sensing is with the finger. Consequently, physical templates can be placed over a touch tablet (as illustrated in Fig. 2) and provide the same type of kinesthetic feedback that one obtains from the frets on a guitar or the cracks between the keys of a piano. This was demonstrated in Buxton, Hill and Rowley (1985). Because of the ability to memorize the position of virtual devices and sense their boundaries, usage is very different than that where a stylus is used, or where the virtual devices are delimited on the tablet surface graphically, and cannot be felt.

An interesting result from our studies, however, is the degree to which eyes-free control can be exercised on a touch tablet which is partitioned into a number of virtual devices, but which has no graphical or physical templates on the tablet surface.

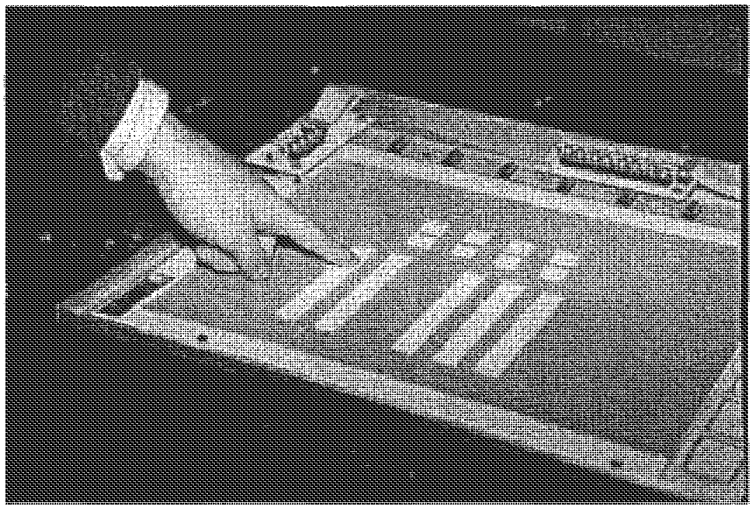


Figure 2: Using a template with a touch tablet

A cut-out template is being placed over a touch tablet. Each cut-out represents a different virtual device on a prototype operating console. The user can operate each device "eyes-free" since boundaries of the virtual devices can be felt (due to the raised edges of the template). If the tablet can sense more than one point of contact at a time, multiple virtual devices can be operated at once. (From Buxton, Hill, & Rowley, 1985).

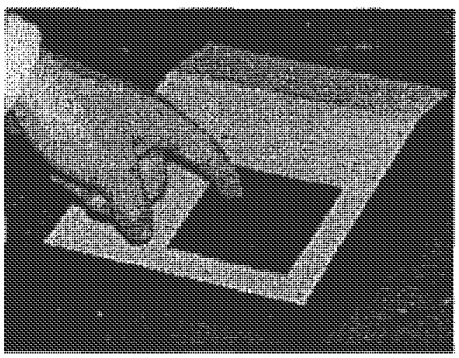


Figure 3: A 3"x3" Touch Tablet

A touch tablet of this size has the important property that it is on the same spatial scale as the hand. Therefore, control and access over its surface falls within the bounds of the relatively highly developed fine motor skills of the fingers, even if the palm is resting in a fixed (home?) position.

Using a 3"x3" touch sensitive touch tablet (shown in Fig. 3), our informal experience suggests that with very little training users can easily discriminate regions to a resolution of up to 1/3 of the tablet surface's vertical or horizontal dimensions. Thus, one can implement three virtual linear potentiometers by dividing the surface into three uniform sized rows or columns, or, for example, one can implement nine virtual push-button switches by partitioning the tablet surface into a 3x3 matrix.

If the surface is divided into smaller regions, such as a 4x4 grid, the result will be significantly more errors, and longer learning time. In such cases, using the virtual devices will require visual attention. The desired eyes-free operability is lost.

These limits are illustrated in Fig. 4. For example, we see that nine buttons for playing tick-tack-toe can work rather well, while a sixteen button numerical button keypad does not. Similarly, three virtual linear faders to control Hue, Saturation and Value work, while four such potentiometers do not.

Our belief is that the performance that we are observing is due to the size of the tablet as it relates to the size of the hand, and the degree of fine motor skills developed in the hand by virtue of everyday living. Being sensitive to these limits is very important as we shall see later when we discuss "dynamic windows." Because of this importance, these limits of motor control warrant more formal study.[1]

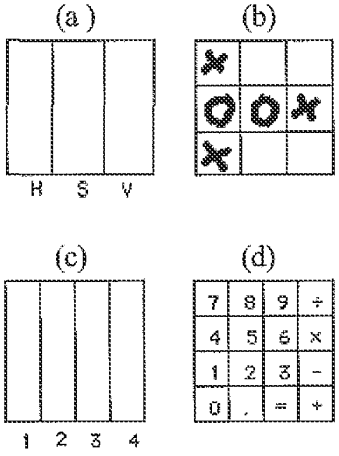


Figure 4: Grids on Touch Tablets

Four mappings of virtual devices are made onto a touch tablet. In (a) and (c), the regions represent linear potentiometers. The surface is partitioned into 3 and 4 regions, respectively. In (b) and (d) the surface is partitioned into a matrix of push buttons (3x3 and 4x4, respectively). Using a 3"x3" touch tablet without templates, our informal experience is that users can resolve virtual devices relatively easily, eyes-free, when the tablet is divided into up to 3 regions in either or both dimensions. This is the situation illustrated in (a) and (b). However, resolving virtual devices where the surface is more finely divided, as in (c) and (d), presents considerably more load. Eyes-free operation requires far more training, and errors are more frequent. The limits on this discrimination warrant more formal study.

Finally, there is the issue of parallel access. Touch technologies have the potential to support multiple virtual devices simultaneously. Again, this is largely by virtue of their not demanding any hand-held intermediate transducer. If, for example, I am holding a stylus in my hand, the affordances of the device bias my expectations towards wanting to draw only one line at a time. In contrast, if I were using finger paints, I would have no such restrictive expectations.

A similar effect is at play in interacting with virtual devices implemented on touch tablets. Consider the template shown in Fig. 2. Nothing biases the user against operating more than one of the virtual linear potentiometers at a time. In fact, experience in the everyday world of such potentiometers would lead one to expect this to be allowed. Consequently, if it is not allowed, the designer must pay particular attention to avoiding probable errors that would result from this false expectation.

Being able to activate more than one virtual device at a time opens up a new possibilities in control and prototyping. The mock up of instrument control consoles is just one example. The biggest obstacle restricting the exploitation of this potential is the lack of a touch tablet that is capable of sensing multiple points. However, Lee, Buxton and Smith (1985) have demonstrated a working prototype of such a transducer, and it is hoped that the applications described in this current paper will help stimulate more activity in this direction.

In summary, we have seen that position sensitive planar devices readily support spatially distinct virtual input devices. Further, we have seen that touch technologies, and touch tablets in particular, have affordances which are particularly well suited to this type of interaction. Finally, it has been shown that a touch tablet capable of sensing more than one point of contact at a time would enable the simultaneous operation of multiple virtual devices.

5. VIRTUAL INPUT TRANSDUCERS

In current "menu on the tablet" practice, there is typically just one device driver which returns a single stream of coordinates. The application must decode the data according to the current partitioning of the tablet. This is all ad hoc, as are the means of specifying the boundaries of the various partitions. There are few tools, and little flexibility.

In our approach, the data from each virtual device is transmitted to the application as if it were coming from an independent physical device with its own driver. If the region is a button device, its driver transmits state changes.

If it is a 1-D relative valuator, it transmits one dimension of relative data in stream mode. All of this is accomplished by placing a "window manager" between the device driver for the sensing transducer and the application.[2] Hence, applications can be constructed independent of how the virtual devices are implemented, thereby maintaining all of the desired properties of device independence. Furthermore, this is accomplished with a uniform set of tools that allows one to define the various regions and the operational behaviour of each region.

6. WHAT ABOUT DYNAMIC WINDOWS?

Window managers for displays can support the dynamic creation, manipulation, and destruction of windows. Is it reasonable to consider comparable functionality for input windows?

Our research (Buxton, Hill & Rowley, 1985) has demonstrated that under certain circumstances, the mapping of virtual devices onto the tablet surface can be dynamically altered. For example, in a paint system, the tablet may be a 2D pointing device in one context, and in another (such as when mixing colours) may have three linear potentiometers mapped onto it.

Changing the mapping of virtual devices onto the tablet surface restricts or precludes the use of physical templates. However, this is not always a problem. If visual (but not tactile) feedback is required, then a touch sensitive flat panel display can provide graphical feedback as to the current mapping. This is standard practice in many touch screen "soft machine" systems.

As has already been discussed, under certain circumstances, some touch tablets can be used effectively without physical or graphical templates. This can be illustrated using a paint mixing example. Since there are three components to colour, three linear potentiometers are used. As in Fig. 4(b), the potentiometers are vertically oriented so that there is no confusion: up is increase, down is decrease. The potentiometers are, left-to-right, Hue, Saturation, and Value (H, S & V in the figure). This ordering is consistent with the conventional order in speech, consequently there is little or no confusion for the user.

The example illustrates three conditions for using virtual devices without templates:

- a low number of devices;
- careful layout;
- strong compatibility between the virtual devices and the application.

Our objective is not to encourage or legitimize the arbitrary use of menus on tablet surfaces. As many CAD systems illustrate, this often leads to bad user interface design. What we hope we have done is identify a technique which, when used in the appropriate context, will result in an improved user interface.

7. UIMS's AND VIRTUAL DEVICES

User Interface Management Systems, or UIMS's, are sets of tools designed to support iterative development of user interfaces through all phases of development (Tanner & Buxton, 1983; Buxton, Lamb, Sherman & Smith, 1983). Ideally, this includes specification, design, implementation, testing, evaluation and redesign. Typically, UIMS's provide tools for the layout of graphic interfaces, control low level details of input and output, and (more rarely) provide monitoring facilities to aid in evaluation of the interfaces developed.

We have developed an *input window manager* (IWM). The tool consists of a "meta device" that provides for quick specification of the layout and behavior of the virtual devices. The specified configuration functions independent of the application. Users employ a *gesture-based trainer* to "show" the system the location and type of virtual device being specified. Hence, for example, adding a new template involves little more than tracing its outline on the control surface, defining the virtual device types and ranges, and attaching them to application parameters. Since the implementation of new devices can be achieved as quickly as they can be laid out on the tablet, this tool provides a new dimension of *system tailorability*.

In order to support iterative development, the tool should allow the user to suspend the application program, change the input configuration (by invoking a special process to control the virtual devices), and then proceed with the application program using the altered input configuration.

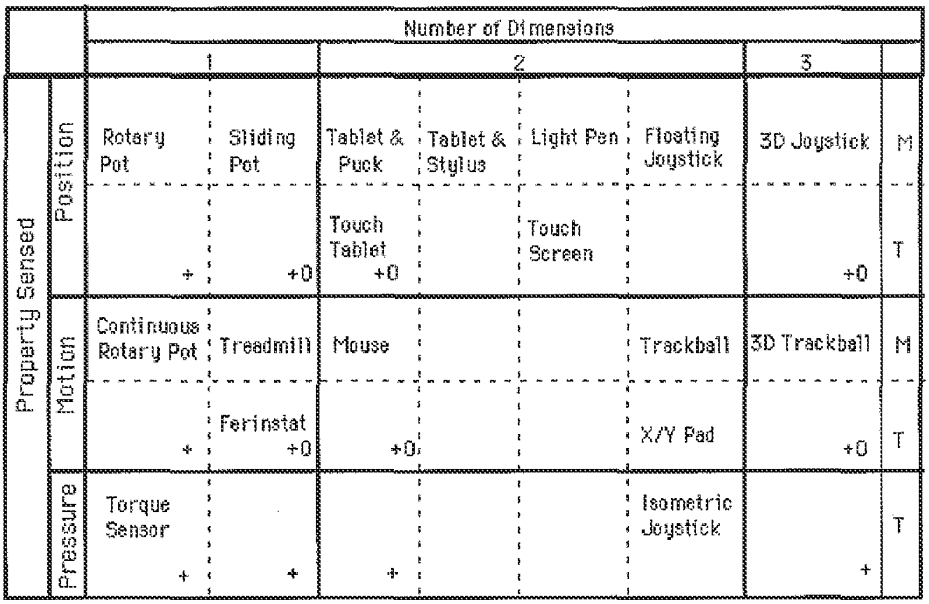


Figure 5. Taxonomy of Hand-Controlled Continuous Input Devices.

Cells represent input transducers with particular properties. Primary rows (solid lines) categorize property sensed (position, motion or pressure). Primary columns categorize number of dimensions transduced. Secondary rows (dashed lines) differentiate devices using a hand-held intermediate transducer (such as a puck or stylus) from those that respond directly to touch - the mediated (M) and touch (T) rows, respectively. Secondary columns group devices roughly by muscle groups employed, or the type of motor control used to operate the device. Cells marked with a "+" can be easily be emulated using virtual devices on a multi-touch tablet. Cells marked with a "O" indicate devices that have been emulated using a conventional digitizing tablet. After Buxton (1983).

8. THE REPERTOIRE OF SUPPORTED VIRTUAL DEVICES

The impact of the physical device used on the quality of interaction has been discussed by Buxton (1983). The objective, therefore, is to make available as broad a repertoire of "virtual" devices as possible from a limited number of physical transducers. We based our initial prototype on a conventional graphics tablet, and have designed to include future support for both single and multiple touch-sensitive tablets. The repertoire of virtual

devices supported by our prototype is indicated in Fig 5.

9. A PROTOTYPE INPUT WINDOW MANAGER

The architecture of the IWM that we have implemented is depicted in Figure 6. The user interacts with the IWM at two separate points indicated by ovals in the diagram. The *Trainer* program, provides for configuring the input control structure. The *application* exists outside of the IWM, and the workings of the IWM are incidental to it (other than the interface to the request handler).

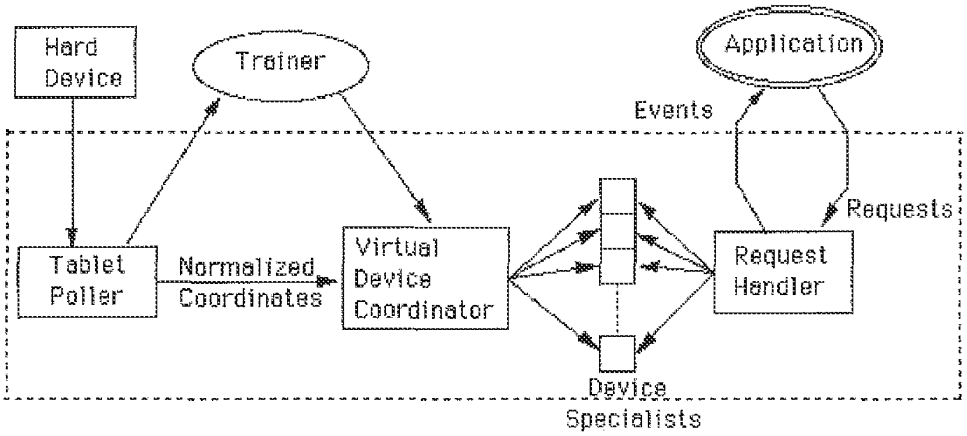


Figure 6. Architecture of a Prototype Input Window Manager

The *tablet poller* monitors the activity on the physical device, filters redundant information, and normalizes the data points before passing them on. The normalized format allows use of a range of physical devices simply by changing the tablet poller for the specific device.

The *virtual device coordinator* is active if the current activity is not a trainer session. It uses the incoming tablet data and the configuration provided by a trainer session to identify the virtual device to which the incoming data belongs. It passes the appropriate information on to the device specialist (device driver) for that virtual device. The *device specialist* determines the effect of the input and signals the *request handler* appropriately.

The virtual devices are accessed by the application program through two communication routines. One routine allows the activation and deactivation of various types of event signals. The other routine accesses the *event-queue*, returning the specifics of the last event to be signaled. A number of requests are available to the activation routine, including discrete status checks on a device, turning the device "on" or "off" for continuous event signaling, and a utility shutdown request.

The request handler module interprets and acts on requests from the application program, altering or extracting information of the device specialists as needed. It posts appropriate events to the event queue.

Finally, the architecture is such that much of the underlying software can reside in a dedicated processor, thereby freeing up resources on the machine running the main application. This includes the part of the tablet poller, the internal representation of the current mapping of the virtual devices onto the tablet, and the virtual device coordinator.

10. CONCLUSION

This paper has discussed one way of making direct manipulation interfaces more direct and manipulation more effective. The general approach has been to extend the number of discrete and continuous controllers which can be tied to different functions. This is accomplished through spatially distinct virtual devices, and an input window management system. In the process, a number of properties of input devices have been discussed, and a prototype system presented. The results have important implications on the usability and tailorability of systems, and the architecture of UIMS's.

The work described has been exploratory. Nevertheless, we feel that the results are sufficiently compelling to suggest that more formal investigations of the issues discussed are warranted. We hope that the current work will help serve as a catalyst to such research.

ACKNOWLEDGEMENTS

The work described in this paper has been supported by the Natural Science and Engineering Research Council of Canada and Xerox PARC. This support is gratefully acknowledged. We would also like to acknowledge the contribution of Ralph Hill, Peter Rowley and Abigail Sellen. Finally, we would like to thank Tom Milligan for his help in proof-reading the final manuscript.

NOTES

1 It must be emphasized that the limits discussed here were obtained through informal study. We intend only to suggest that there is something interesting and useful here, rather than to imply that these are experimentally derived data.

2 We thank Alain Fournier for first suggesting the analogy with window managers.

REFERENCES

Anson, E. (1982). The Device Model of Interaction. Computer Graphics, (16,3), 107-114.

Buxton, W. (1983). Lexical and Pragmatic Considerations of Input Structures. Computer Graphics, (17,1), 31-37

Buxton W., Hill R., & Rowley P. (1985). Issues and Techniques in Touch-Sensitive Tablet Input. Computer Graphics, 19(3), 215 - 224.

Buxton, W., Lamb, M., Sherman, D., & Smith, K.C. (1982). Towards a Comprehensive User Interface Management System. Computer Graphics, (16,3), 99-106.

Buxton, W. & Myers, B. (1986). A Study in Two-Handed Input. Proceedings of CHI'86 Conference on Human Factors in Computing Systems, 321-326.

Evans, K. Tanner, P., & Wein, M. (1981). Tablet-based Valuators that Provide One, Two, or Three Degrees of Freedom. Computer Graphics (15,3), 91-97.

Kasik, D. (1982). A User Interface Management System. Computer Graphics, (16,3), 99-106.

Knowlton, K. (1975). Virtual Pushbuttons as a Means of Person- Machine Interaction. Proc. IEEE Conf. on Computer Graphics, Pattern Matching, and Data Structure., 350-351.

Knowlton, K. (1977a). Computer Displays Optically Superimposed on Input Devices. The Bell System Technical Journal (56,3), 367-383.

Knowlton, K. (1977b). Prototype for a Flexible Telephone Operator's Console Using Computer Graphics. 16mm film, Bell Labs, Murray Hill, NJ.

Lee S., Buxton, W., & Smith, K.C. (1985). A Multi-Touch Three Dimensional Touch Tablet. Proceedings of CHI'85 Conference on Human Factors in Computing Systems, 21 - 25.

Myers, B. (1984a), Strategies for Creating an Easy to Use Window Manager with Icons. Proceedings of Graphics Interface '84, Ottawa, May, 1984, 227 - 233.

Myers, B. (1984b), The User Interface for Sapphire. IEEE Computer Graphics and Applications, 4 (12), 13 - 23.

Pike, R. (1983). Graphics in Overlapping Bitmap Layers. Computer Graphics, 17 (3), 331 - 356.

Tanner, P.P. & Buxton, W. (1985). Some Issues in Future User Interface Management System (UIMS) Development. In Pfaff, G. (Ed.), User Interface Management Systems, Berlin: Springer Verlag, 67 - 79.


File Edit View Favorites Tools Help
Upload
A Multi-Touch Three Dimensional Touch-Tablet
Bill Buxton
5,286
Like
About


Casio AT-550 Touch Screen Calculator Watch (1984)
Bill Buxton
18,167
Share this video
Prof. Bill Buxton Predicts the End of Personal Computers.
Is the Pebble Smart Watch A Good Buy? - Walt Mossberg
2013 Korea Open (ms-final) XU Xin - MA Long [Full Match/Short
The Basics of Volleyball - Serving
Casio Pro Trek 2500T Review
Volleyball Techniques and Tactics to Win the Serve
Marvin Minsky: Mind As Society (excerpt) - Thinking Allowed DVD
$40 ebay CASIO CALCULATOR WATCH DBC-32D-1A unboxing &

Electronic Patent Application Fee Transmittal

Application Number:	
Filing Date:	
Title of Invention:	Capacitive Responsive Electronic Switching Circuit
First Named Inventor/Applicant Name:	Byron Hourmand
Filer:	Brian A. Carlson
Attorney Docket Number:	5796183RX2

Filed as Small Entity

ex parte reexam Filing Fees

Description	Fee Code	Quantity	Amount	Sub-Total in USD($)
Basic Filing:				
REQUEST FOR EX PARTE REEXAMINATION	2812	1	6000	6000
Pages:				
Claims:				
Reexamination Independent Claims	2821	5	210	1050
REEXAMINATION CLAIMS IN EXCESS OF TWENTY	2822	65	40	2600
Miscellaneous-Filing:				
Petition:				
Patent-Appeals-and-Interference:				

Description	Fee Code	Quantity	Amount	Sub-Total in USD($)
Post-Allowance-and-Post-Issuance:				
Extension-of-Time:				
Miscellaneous:				
		Total in USD ($)		**9650**

Electronic Acknowledgement Receipt	
EFS ID:	17754459
Application Number:	90013106
International Application Number:	
Confirmation Number:	9188
Title of Invention:	Capacitive Responsive Electronic Switching Circuit
First Named Inventor/Applicant Name:	Byron Hourmand
Customer Number:	25962
Filer:	Brian A. Carlson
Filer Authorized By:	
Attorney Docket Number:	5796183RX2
Receipt Date:	24-DEC-2013
Filing Date:	
Time Stamp:	15:03:12
Application Type:	Reexam (Patent Owner)

Payment information:

Submitted with Payment	yes
Payment Type	Deposit Account
Payment was successfully received in RAM	$9650
RAM confirmation Number	1558
Deposit Account	501065
Authorized User	
The Director of the USPTO is hereby authorized to charge indicated fees and credit any overpayment as follows: Charge any Additional Fees required under 37 C.F.R. Section 1.17 (Patent application and reexamination processing fees)	

File Listing:

Document Number	Document Description	File Name	File Size(Bytes)/ Message Digest	Multi Part /.zip	Pages (if appl.)
1	Transmittal of New Application	PTO_NAR-5796183RX2_Reexam_Request_Form_PTO.pdf	125658 f7f01010949a811011ceaadb8c6aeb2ec4c2c095	no	3
Warnings:					
Information:					
2	Receipt of Original Ex Parte Reexam Request	PTO_NAR-5796183RX2_Request_for_Reexam_PTO.pdf	350144 e4cb2e4cddcd654c0466c77053e629754043d421	no	34
Warnings:					
Information:					
3	Reexam Miscellaneous Incoming Letter	PTO_NAR-5796183RX2_Exhibit_A_PTO2.pdf	6723309 a60672b73e9aa2bdb0a60c5ee3b11f12caa63bb5	no	37
Warnings:					
Information:					
4	Reexam Miscellaneous Incoming Letter	PTO_NAR-5796183RX2_Exhibit_B_PTO2.pdf	17839163 5d37c96622cb2804ea03b3b41d6022278b5f07ba	no	136
Warnings:					
Information:					
5	Reexam Miscellaneous Incoming Letter	PTO_NAR-5796183RX2_Exhibit_C_PTO2.pdf	2031975 164bc7c660d83257e8969243352c131ad2e385f0	no	13
Warnings:					
Information:					
6	Reexam Miscellaneous Incoming Letter	PTO_NAR-5796183RX2_Exhibit_D_PTO2.pdf	2558063 6b2c2a6bc334b4e355e5cbc6c24bc59df7711fa2	no	16
Warnings:					
Information:					
7	Reexam Miscellaneous Incoming Letter	PTO_NAR-5796183RX2_Exhibit_E_PTO2.pdf	309214 6d597cfba861476043777049d6835394e24262dc	no	2
Warnings:					
Information:					
8	Preliminary Amendment	PTO_NAR-5796183RX2_Amend_Acc_Reexam_Req_PTO.pdf	683580 179abf2337afcc80e41cd06652a3000e163f9d8a	no	142
Warnings:					
Information:					

9	Copy of patent for which reexamination is requested	PTO_NAR-5796183RX2_183Patent_PTO.PDF	2272260 4f5329791f53446bb65436b84cf199105373e181	no	36
Warnings:					
Information:					
10	Assignee showing of ownership per 37 CFR 3.73.	PTO_NAR-5796183RX2_373_Form_PTO.pdf	123283 128c4c53bfe9c2b814cf0c633f8402e2330c74ce	no	3
Warnings:					
Information:					
11	Power of Attorney	PTO_NAR-5796183RX2_Power_of_Atty_PTO.PDF	302528 b07c6a63a6c63630476018b064dc44f4d895b527	no	1
Warnings:					
Information:					
12	Transmittal Letter	PTO_NAR-5796183RX2_IDS_Cover_Ltr_PTO.pdf	15259 c305711d1f9f296a4b83e31e8170fb106c6f7957	no	1
Warnings:					
Information:					
13	Information Disclosure Statement (IDS) Form (SB08)	PTO_NAR-5796183RX2_SB08_Form_PTO.pdf	695955 883bc66b5537b8497a0e5f97f27f0062d0ebba5d	no	7
Warnings:					
Information:					
14	Non Patent Literature	PTO_NPL_Buxton_InvitedPaper_PTO.pdf	409495 312f0417fac8701faeee3f50d9fa0e57b54f51b4	no	5
Warnings:					
Information:					
15	Non Patent Literature	PTO_NPL_Hinckley_PTO.pdf	248635 55da6e593e3e6b72c4b4f769a1e91ea0307c3ca2	no	4
Warnings:					
Information:					
16	Non Patent Literature	PTO_NPL_Lee_Thesis_PTO.pdf	16890598 ab9acf8a342960946582eb3e3b7206f9fbc6e5d8	no	118
Warnings:					
Information:					
17	Non Patent Literature	PTO_NPL_Hillis_AHighRes_PTO.pdf	1879168 aa40075463062ed7a2ebde71f2f34ec049f517ef	no	12
Warnings:					
Information:					

18	Non Patent Literature	PTO_NPL_Lee_AMulti-Touch_PTO.pdf	874510 df53aec5188e8773b1722ba895f8184e3172a8fc	no	5
Warnings:					
Information:					
19	Non Patent Literature	PTO_NPL_Hlady_PTO.pdf	2257242 e0d2386694a86e80874a7684a55c3ae92dbfa5f2	no	7
Warnings:					
Information:					
20	Non Patent Literature	PTO_NPL_Sasaki_PTO.pdf	5118466 ce673fbb29fbbde7cffba06f5f0337047b2307cb	no	5
Warnings:					
Information:					
21	Non Patent Literature	PTO_NPL_Callahan_PTO.pdf	1192812 adee8f6a5fb28823fc4295ececcb97cfe4790778	no	6
Warnings:					
Information:					
22	Non Patent Literature	PTO_NPL_Casio_Ad_PTO.pdf	321343 bab17c766e91da884c0f3235447ae2f22270bcb4	no	1
Warnings:					
Information:					
23	Non Patent Literature	PTO_NPL_Casio_Manual_PTO.pdf	1939711 b598ae16cb083a0d9b7db30bf8e478164238871c	no	14
Warnings:					
Information:					
24	Non Patent Literature	PTO_NPL_Smith_Bit-slice_PTO.pdf	1363333 d82b4cdfb5dea978274a51afe2da349f570da041	no	7
Warnings:					
Information:					
25	Non Patent Literature	PTO_NPL_Boie_PTO.pdf	1522073 574b22e97a3475322299d25a826d128ec602dae4	no	9
Warnings:					
Information:					
26	Non Patent Literature	PTO_NPL_CliveThomson_PTO.pdf	219994 2c5bd016bd4ef378eccdac8ac2c124ae3efdfca6	no	3
Warnings:					
Information:					

27	Non Patent Literature	PTO_NPL_National_Research_Council_PTO.pdf	830527 f8ce9991051b72736e5e6b0797e3c0a4eee216ff	no	85
Warnings:					
Information:					
28	Non Patent Literature	PTO_NPL_Buxton_IssuesTech_PTO.pdf	2176107 f4d1d89e7255ef9edbf37a4c3fd385507f4bbc5d	no	10
Warnings:					
Information:					
29	Non Patent Literature	PTO_NPL_Buxton_LargeDisplays_PTO.pdf	699993 1aface304e3752dc2b700ebdd3d6f104dd26235d	no	8
Warnings:					
Information:					
30	Non Patent Literature	PTO_NPL_Buxton_LexicalPrag_PTO.pdf	1531125 781e65ebd8d45a45f6a0e30d1b5a5a61a24fc7a1	no	7
Warnings:					
Information:					
31	Non Patent Literature	PTO_NPL_LightBeam_PTO.pdf	81930 5027e8f24277c185a160634ffe254a811889e25b	no	2
Warnings:					
Information:					
32	Non Patent Literature	PTO_NPL_Buxton_Multi-Touch_PTO.pdf	643791 ee40592d58cadcfb1894732afd286d0035b0df7d	no	22
Warnings:					
Information:					
33	Non Patent Literature	PTO_NPL_Herot_PTO.pdf	1550938 87b7288c423780d930b4a5c062013ee322bd3249	no	7
Warnings:					
Information:					
34	Non Patent Literature	PTO_NPL_Wolfeld_PTO.pdf	15411551 619e70b30024524a72a6181ac4f6317a0334bb9e	no	68
Warnings:					
Information:					
35	Non Patent Literature	PTO_NPL_Lewis_PTO.pdf	613852 ecc53097f693967b592bad1026125a16ec91dec5	no	6
Warnings:					
Information:					

36	Non Patent Literature	PTO_NPL_Nakatani_PTO.pdf	1112483 5c2d9a7ff7d21d354e8097017e0608c9aa16f7aa	no	5
Warnings:					
Information:					
37	Non Patent Literature	PTO_NPL_Rubine_PTO.pdf	3162925 557a95160c204d5233ff036d96b23bc10b8f680d	no	285
Warnings:					
Information:					
38	Non Patent Literature	PTO_NPL_Kurtenbach_PTO.pdf	1670439 0dc0ca4099f4b05bb0bdf6025f87a999e2de6e14	no	201
Warnings:					
Information:					
39	Non Patent Literature	PTO_NPL_Hopkins_PTO.pdf	113021 1967f1f1645e82db2843bf6c0557227911ede095	no	8
Warnings:					
Information:					
40	Non Patent Literature	PTO_NPL_Buxton_LongNose_PTO.pdf	27893 621d2f1a416496295bdcf582e2595a3bb4cb29eb	no	3
Warnings:					
Information:					
41	Non Patent Literature	PTO_NPL_Buxton_MadDash_PTO.pdf	37401 d513532e5b9fc379095305b223f5f487e34d0055	no	3
Warnings:					
Information:					
42	Non Patent Literature	PTO_NPL_NASA_Graphic_Manipulator_PTO.pdf	4547824 87aa779ba0b2789b7b017d0a92784dbd0b59ad9d	no	28
Warnings:					
Information:					
43	Non Patent Literature	PTO_NPL_Izadi_PTO.pdf	1221374 ab87fdb52a4317631f99f54206dd79f0f187ce87	no	9
Warnings:					
Information:					
44	Non Patent Literature	PTO_NPL_Krueger_PTO.pdf	1080548 c5868e5addea39874c8a8b01c14e5be5fb1e35f7	no	6
Warnings:					
Information:					

45	Non Patent Literature	PTO_NPL_Brown_PTO.pdf	1150381 7cd9999fde19f7f57a96158c178b3277a5930bc4	no	11
Warnings:					
Information:					
46	Non Patent Literature	PTO_NPL_YouTube_Video1_PTO.pdf	478495 8caeaa56b47adc32cceca7a37f5062e1458b2656	no	1
Warnings:					
Information:					
47	Non Patent Literature	PTO_NPL_YouTube_Video2_PTO.pdf	611272 155f817a147f94608c33dc9f0a5e86b94823d38c	no	1
Warnings:					
Information:					
48	Fee Worksheet (SB06)	fee-info.pdf	33467 2041622c6292997365ba41e75802557ae353b4e6	no	2
Warnings:					
Information:					
		Total Files Size (in bytes):	107055108		

This Acknowledgement Receipt evidences receipt on the noted date by the USPTO of the indicated documents, characterized by the applicant, and including page counts, where applicable. It serves as evidence of receipt similar to a Post Card, as described in MPEP 503.

New Applications Under 35 U.S.C. 111
If a new application is being filed and the application includes the necessary components for a filing date (see 37 CFR 1.53(b)-(d) and MPEP 506), a Filing Receipt (37 CFR 1.54) will be issued in due course and the date shown on this Acknowledgement Receipt will establish the filing date of the application.

National Stage of an International Application under 35 U.S.C. 371
If a timely submission to enter the national stage of an international application is compliant with the conditions of 35 U.S.C. 371 and other applicable requirements a Form PCT/DO/EO/903 indicating acceptance of the application as a national stage submission under 35 U.S.C. 371 will be issued in addition to the Filing Receipt, in due course.

New International Application Filed with the USPTO as a Receiving Office
If a new international application is being filed and the international application includes the necessary components for an international filing date (see PCT Article 11 and MPEP 1810), a Notification of the International Application Number and of the International Filing Date (Form PCT/RO/105) will be issued in due course, subject to prescriptions concerning national security, and the date shown on this Acknowledgement Receipt will establish the international filing date of the application.

UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

Bib Data Sheet

CONFIRMATION NO. 9188

SERIAL NUMBER	FILING OR 371(c) DATE	CLASS	GROUP ART UNIT	ATTORNEY DOCKET NO.
90/013,106	12/24/2013 RULE	307	3992	5796183RX2

AIA (First Inventor to File): YES

INVENTORS

5796183, Residence Not Provided;
NARTRON CORPORATION, REED CITY, MI;

APPLICANTS

5796183, Residence Not Provided;
NARTRON CORPORATION, REED CITY, MI;

**** CONTINUING DATA ***************************

This application is a REX of 08/601,268 01/31/1996 PAT 5796183

**** FOREIGN APPLICATIONS ********************

** SMALL ENTITY **

		STATE OR COUNTRY	SHEETS DRAWING	TOTAL CLAIMS	INDEPENDENT CLAIMS
Foreign Priority claimed	☐ yes ☐ no				
35 USC 119 (a-d) conditions met	☐ yes ☐ no ☐ Met after Allowance				
Verified and Acknowledged	Examiner's Signature / Initials			32	8

ADDRESS
25962

TITLE

Capacitive Responsive Electronic Switching Circuit

FILING FEE RECEIVED		
6000	FEES: Authority has been given in Paper No. _____ to charge/credit DEPOSIT ACCOUNT No. _____ for following:	☐ All Fees ☐ 1.16 Fees (Filing) ☐ 1.17 Fees (Processing Ext. of time) ☐ 1.18 Fees (Issue) ☐ Other _____ ☐ Credit