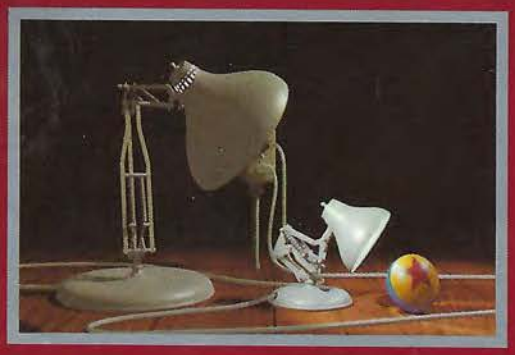
INTRODUCTION TO

COMPUTER GRAPHICS

FOLEY • VAN DAM • FEINER • HUGHES • PHILLIPS



INTRODUCTION TO COMPUTER GRAPHICS

James Foley, Georgia Institute of Technology
Andries van Dam, Brown University
Steven K. Feiner, Columbia University
John F. Hughes, Brown University
Richard L. Phillips, Los Alamos National Laboratory
and The University of Michigan

This introduction to computer graphics is an adaptation of *Computer Graphics: Principles and Practice, Second Edition*, which remains the most comprehensive and authoritative work in the field. While retaining the currency and accuracy of the larger book, this abbreviated version focuses on topics essential for all beginners in computer graphics and provides expanded explanations for readers with less technical background. Worked examples have been added to illustrate concepts and techniques, and program code has been written in the C language to enhance the book's usefulness.

Topic coverage includes basic graphics programming, hardware, and applications. Important algorithms are included to facilitate implementation of both 2D and 3D graphics. A separate chapter covers SPHIGS—a simplified dialect of the PHIGS 3D standard—and coincides with the availability of an updated version of the software. Another chapter presents a concise overview of interaction issues and techniques. Advanced material from the larger book has been condensed, and the mathematics needed for it has been explained carefully. The book is profusely illustrated, with more than 50 full-color images. The result is an accessible introduction to computer graphics, crafted to provide a solid foundation for futher work in this exciting field.

About the Authors

James Foley (Ph.D., University of Michigan) is professor and director of the Graphics, Visualization & Usability Center in The College of Computing at the Georgia Institute of Technology. **Andries van Dam** (Ph.D., University of Pennsylvania) is currently the Herbert Ballon University Professor and professor of computer science at Brown University, and on the Technical Advisory Boards of Electronic Book Technologies, Microsoft, and Sho Graphics. **Steven Feiner** (Ph.D., Brown University) is associate professor of computer science at Columbia University where he directs the computer graphics group. **John Hughes** (Ph.D., University of California, Berkeley) is associate professor (research) of computer science and mathematics at Brown University where he codirects the computer graphics group with Andries van Dam. **Richard Phillips** (Ph.D., University of Michigan), is principally responsible for this adaptation. Professor Emeritus at The University of Michigan, he is currently a scientist at Los Alamos National Laboratory. See the Preface for more about this distinguished team of authors.

ADDISON-WESLEY PUBLISHING COMPANY

90100>
9 780201 609219
ISBN 0-201-60921-5

Introduction to Computer Graphics

James D. Foley
Georgia Institute of Technology

Andries van Dam
Brown University

Steven K. Feiner
Columbia University

John F. Hughes
Brown University

Richard L. Phillips
Los Alamos National Laboratory
and The University of Michigan

ADDISON-WESLEY PUBLISHING COMPANY

Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam • Bonn
Sydney • Singapore • Tokyo • Madrid • San Juan • Milan • Paris

Sponsoring Editor *Peter S. Gordon*
Senior Production Supervisor *Jim Rigney*
Copy Editors *Lyn Dupré and Joyce Grandy*
Text Designer *Sandra Rigney*
Technical Art Consultant *Joseph K. Vetere*
Illustrators *C&C Associates and Tech Graphics*
Cover Designer *Eileen Hoff*
Senior Manufacturing Manager *Roy Logan*
Marketing Manager *Robert Donegan*

Cover images from "Luxo Jr.," by J. Lasseter, W. Reeves, E. Ostby, and S. Leffler. (Copyright © 1986 Pixar.) "Luxo" is a trademark of Jac Jacobsen Industrier.

This book is an abridged and modified version of *Computer Graphics: Principles and Practice,* Second Edition, by Foley, van Dam, Feiner, and Hughes, published in 1990 in the Addison-Wesley Systems Programming Series, IBM Editorial Board, consulting editors.

Library of Congress Cataloging-in-Publication data

Introduction to computer graphics / James D. Foley . . . [et al.] .].
p. cm.
Includes bibliographical references and index.
ISBN 0-201-60921-5
1. Computer graphics. I. Foley, James D., 1942– .
T385.I538 1993
006.6—dc20 93-16677
CIP

Reprinted with corrections February, 1994

4 5 6 7 8 9 10 MA 9897969594

Contents

2.2 BASIC INTERACTION HANDLING

Now that we know how to draw basic shapes and text, our next step is to learn how to write interactive programs that communicate effectively with the user, via input devices such as the keyboard and the mouse. First, we look at general guidelines for making effective and pleasant-to-use interactive programs; then we discuss the fundamental notion of logical (abstract) input devices. Finally, we look at SRGP's mechanisms for dealing with various aspects of interaction handling.

2.2.1 Human Factors

The designer of an interactive program must deal with many matters that do not arise in a noninteractive, batch program. They are the so-called **human factors** of a program, such as its interaction style (often called **look and feel**) and its ease of learning and of use, and they are as important as its functional completeness and correctness. Techniques for user–computer interaction that exhibit good human factors are studied in more detail in Chapter 8. The guidelines discussed there include these:

- Provide *simple and consistent* interaction sequences.
- *Do not overload the user* with too many different options and styles.
- *Show the available options clearly* at every stage of the interaction.
- *Give appropriate feedback* to the user.
- Allow the user to *recover gracefully* from mistakes.

We attempt to follow these guidelines for good human factors in our sample programs. For example, we typically use menus to allow the user to indicate which function to execute next, by using a mouse to pick a text button in a menu of such buttons. Also common are palettes (iconic menus) of basic geometric primitives, application-specific symbols, and fill patterns. Menus and palettes satisfy our first three guidelines in that their entries prompt the user with a list of available options and provide a single, consistent way of choosing among these options. Unavailable options may be either deleted temporarily or *grayed out* by being drawn in a low-intensity gray-scale pattern rather than a solid color (see Programming Project 2.14).

Feedback occurs at every step of a menu operation to satisfy the fourth guideline: The application program will *highlight* the menu choice or object selection—for example, display it in inverse video or framed in a rectangle—to draw attention to it. The package itself may also provide an *echo* in which an immediate response to the manipulation of an input device is given. For example, characters appear immediately at the position of the cursor as keyboard input is typed; as the mouse is moved on the table or desktop, a cursor echoes the corresponding location on the

screen. Graphics packages offer a variety of cursor shapes that can be used by the application program to reflect the state of the program. In many display systems, the cursor shape can be varied dynamically as a function of the cursor's position on the screen. In many word-processing programs, for example, the cursor is shown as an arrow in menu areas and as a blinking vertical bar in text areas.

Graceful error recovery, our fifth guideline, is usually provided through *cancel* and *undo/redo* features. They require the application program to maintain a record of operations and their inverse, corrective actions.

2.2.2 Logical Input Devices

Device types in SRGP. A major goal in designing graphics packages is device independence, which enhances portability of applications. SRGP achieves this goal for graphics output by providing primitives specified in terms of an abstract integer coordinate system, thus shielding the application from the need to set the individual pixels in the frame buffer. To provide a level of abstraction for graphics input, SRGP supports a set of **logical input devices** that shield the application from the details of the physical input devices available. Two logical devices are supported by SRGP:

- **Locator**, a device for specifying screen coordinates and the state of one or more associated buttons
- **Keyboard**, a device for specifying character string input

SRGP maps the logical devices onto the physical devices available (e.g., the locator could map to a mouse, joystick, tablet, or touch-sensitive screen). This mapping of logical to physical is familiar from conventional procedural languages and operating systems, in which I/O devices such as terminals, disks, and tape drives are abstracted to logical data files to achieve both device-independence and simplicity of application programming.

Device handling in other packages. SRGP's input model is essentially a subset of the GKS and PHIGS input models. SRGP implementations support only one logical locator and one keyboard device, whereas GKS and PHIGS allow multiple devices of each type. Those packages also support additional device types: the **stroke** device (returning a polyline of cursor positions entered with the physical locator), the **choice** device (abstracting a function-key pad and returning a key identifier), the **valuator** (abstracting a slider or control dial and returning a floating-point number), and the **pick** device (abstracting a pointing device, such as a mouse or data tablet, with an associated button to signify a selection, and returning the identification of the logical entity picked). Other packages, such as QuickDraw and the X Window System, handle input devices in a more device-dependent way that gives the programmer finer control over an individual device's operation, at the cost of greater application-program complexity and reduced portability to other platforms.

Chapter 8 elaborates further on the properties of logical devices. Here, we briefly summarize modes of interacting with logical devices in general, and then examine SRGP's interaction functions in more detail.

2.2.3 Sampling Versus Event-Driven Processing

Two fundamental techniques are used to receive information created by user interactions. In **sampling** (also called **polling**), the application program queries the current value of a logical input device (called the **measure** of the device) and continues execution. The sampling is performed regardless of whether the device's measure has changed since the last sampling; indeed, only by continuous sampling of the device will changes in the device's state be known to the application. This mode is costly for interactive applications, because they would spend most of their CPU cycles in tight sampling loops waiting for measure changes.

An alternative to the CPU-intensive polling loop is the **interrupt-driven** interaction; in this technique, the application enables one or more devices for input and then continues normal execution until interrupted by some input **event** (a change in a device's state caused by user action); control then passes asynchronously to an interrupt procedure, which responds to the event. For each input device, an **event trigger** is defined; the event trigger is the user action that causes an event to occur. Typically, the trigger is a button push, such as a press of the mouse button (**mouse down**) or a press of a keyboard key.

To free applications programmers from the tricky and difficult aspects of asynchronous transfer of control, many graphics packages, including GKS, PHIGS, and SRGP, offer **event-driven** interaction as a synchronous simulation of interrupt-driven interaction. In this technique, an application enables devices and then continues execution. In the background, the package monitors the devices and stores information about each event in an event queue (Fig. 2.11). The application, at its convenience, checks the event queue and processes the events in temporal order. In effect, the application specifies when it would like to be *interrupted*.

When an application checks the event queue, it specifies whether it would like to enter a wait state. If the queue contains one or more event reports, the head event (representing the event that occurred earliest) is removed, and its information is made available to the application. If the queue is empty and a wait state is not desired, the application is informed that no event is available and that it is free to continue execution. If the queue is empty and a wait state is desired, the application pauses until the next event occurs or until an application-specified maximum-wait-time interval passes. In effect, event mode replaces polling of the input devices with the much more efficient waiting on the event queue.

In summary, in sampling mode, the device is polled and an event measure is collected, regardless of any user activity. In event mode, the application either gets an event report from a prior user action or waits until a user action (or timeout) occurs. It is this *respond only when the user acts* behavior of event mode that is the essential difference between sampled and event-driven input. Event-driven programming may seem more complex than sampling, but you are already familiar with a similar technique used with the scanf function in an interactive C program:

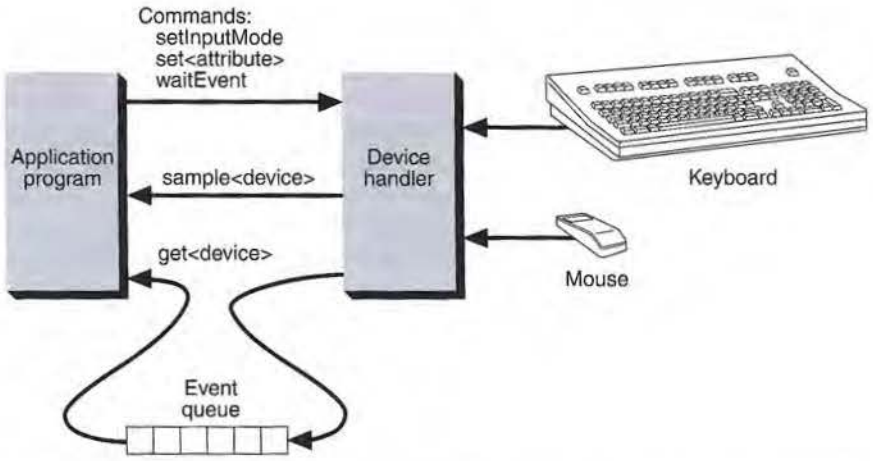


Figure 2.11 Sampling versus event-handling using the event queue.

C enables the keyboard, and the application waits in the scanf until the user has completed entering a line of text. Some environments allow the scanf statement to access characters that were typed and queued before the scanf was issued.

Simple event-driven programs in SRGP and in similar packages follow the reactive *ping-pong* interaction introduced in Section 1.4.3 and pseudocoded in Prog. 2.4; this interaction can be nicely modeled as a finite-state automaton. More complex styles of interaction, allowing simultaneous program and user activity, are discussed in Chapter 8.

Program 2.4

Event-driven interaction scheme.

```
initialize, including generating the initial image;
activate interactive device(s) in event mode;
do {          /* main event loop */
   wait for user-triggered event on any of several devices;
   switch ( which device caused event ) {
      case DEVICE_1: collect DEVICE_1 event measure data, process, respond;
      case DEVICE_2: collect DEVICE_2 event measure data, process, respond;
      ...
   }
}
while ( user does not request quit );
```

Event-driven applications typically spend most of their time in a wait state, since interaction is dominated by *think time* during which the user decides what to do next; even in fast-paced game applications, the number of events a user can generate in a second is a fraction of what the application could handle. Since SRGP typically implements event mode using true (hardware) interrupts, the wait state effectively uses no CPU time. On a multitasking system, the advantage is obvious: The event-mode application requires CPU time only for short bursts of

activity immediately following user action, thereby freeing the CPU for other tasks.

One other point, about correct use of event mode, should be mentioned. Although the queueing mechanism does allow program and user to operate asynchronously, the user should not be allowed to get too far ahead of the program, because each event should result in an echo as well as some feedback from the application program. It is true that experienced users have learned to use **typeahead** to type in parameters such as file names or even operating-system commands while the system is processing earlier requests, especially if at least a character-by-character echo is provided immediately. In contrast, **mouseahead** for graphical commands is generally not as useful (and is much more dangerous), because the user usually needs to see the screen updated to reflect the application model's current state before the next graphical interaction.

2.2.4 Sample Mode

Activating, deactivating, and setting the mode of a device. The following function is used to activate or deactivate a device; taking a device and a mode as parameters:

```
void SRGP_setInputMode ( inputDevice LOCATOR / KEYBOARD,
    inputMode INACTIVE / SAMPLE / EVENT);
```

Thus, to set the locator to sample mode, we call

```
SRGP_setInputMode (LOCATOR, SAMPLE);
```

Initially, both devices are inactive. Placing a device in a mode in no way affects the other input device—both may be active simultaneously and even then need not be in the same mode.

The locator's measure. The locator is a logical abstraction of a mouse or data tablet, returning the cursor position as a screen (x, y) coordinate pair, the number of the button that most recently experienced a transition, and the state of the buttons as a **chord** array (since multiple buttons can be pressed simultaneously). The second field lets the application know which button caused the trigger for that event.

```
typedef struct {
    point  position;
    enum {
        UP, DOWN
    } buttonChord[MAX_BUTTON_COUNT];          /*Typically 1–3*/
    int  buttonOfMostRecentTransition;
} locatorMeasure;
```

Having activated the locator in sample mode with the SRGP_setInputMode function, we can ask its current measure using

```
void  SRGP_sampleLocator ( locatorMeasure *measure );
```

Let us examine the prototype sampling application shown in Prog. 2.5, a simple *painting* loop involving only button 1 on the locator. Such painting entails leaving a trail of paint where the user has dragged the locator while holding down button 1; the locator is sampled in a loop as the user moves it. First, we must detect when the user starts painting by sampling the button until it is depressed; then we place the paint (a filled rectangle in our simple example) at each sample point until the user releases the button.

Program 2.5 Sampling loop for painting.

```
set up color/pattern attributes, and brush size in halfBrushHeight and halfBrushWidth
SRGP_setInputMode(LOCATOR, SAMPLE);

/* First, sample until the button goes down.*/
do {
   SRGP_sampleLocator(locMeasure);
} while (locMeasure.buttonChord[0] == UP);

/*Perform the painting loop:
      Continuously place brush and then sample, until button is released.*/
do {
   rect = SRGP_defRectangle( locMeasure.position.x – halfBrushWidth,
                             locMeasure.position.y – halfBrushHeight,
                             locMeasure.position.x + halfBrushWidth,
                             locMeasure.position.y + halfBrushHeight );
   SRGP_fillRectangle (rect );
   SRGP_sampleLocator( &locMeasure );
} while ( locMeasure.buttonChord[0] == DOWN );
```

The results of this sequence are crude: The paint rectangles are arbitrarily close together or far apart, with their density completely dependent on how far the locator was moved between consecutive samples. The sampling rate is determined essentially by the speed at which the CPU runs the operating system, the package, and the application.

Sample mode is available for both logical devices; however, the keyboard device is almost always operated in event mode, so techniques for sampling it are not addressed here.

2.2.5 Event Mode

Using event mode for initiation of sampling loop. Although the two sampling loops of the painting example (one to detect the button-down transition, the other to paint until the button-up transition) certainly do the job, they put an unnecessary load on the CPU. Although overloading may not be a serious concern in a personal computer, it is not advisable in a system running multiple tasks, let alone doing time-sharing. Although it is certainly necessary to sample the locator repetitively for the painting loop itself (because we need to know the position of the locator at all times while the button is down), we do not need to use a sampling loop to wait for the button-down event that initiates the painting interaction. Event mode,

discussed next, can be used when there is no need for measure information while waiting for an event.

SRGP_waitEvent. At any time after SRGP_setInputMode has activated a device in event mode, the program may inspect the event queue by entering the wait state with

```
inputDevice SRGP_waitEvent ( int maxWaitTime );
```

The function returns immediately if the queue is not empty; otherwise, the parameter specifies the maximum amount of time (measured in $^1/_{60}$ second) for which the function should wait for an event to fill the queue. A negative *maxWaitTime* (specified by the symbolic constant INDEFINITE) causes the function to wait indefinitely, whereas a value of zero causes it to return immediately, regardless of the state of the queue.

The function returns the identity of the device that issued the head event, as LOCATOR, KEYBOARD, or NO_DEVICE. The special value NO_DEVICE is returned if no event was available within the specified time limit—that is, if the device timed out. The device type can then be tested to determine how the head event's measure should be retrieved (described later in this section).

The keyboard device. The trigger event for the keyboard device depends on the **processing mode** in which the keyboard device has been placed. EDIT mode is used when the application receives strings (e.g., file names, commands) from the user, who types and edits the string and then presses the Return key to trigger the event. In RAW mode, used for interactions in which the keyboard must be monitored closely, every key press triggers an event. The application uses the following function to set the processing mode:

```
void SRGP_setKeyboardProcessingMode ( keyboardMode EDIT / RAW );
```

In EDIT mode, the user can type entire strings, correcting them with the backspace key as necessary, and then use the Return (or Enter) key as trigger. This mode is used when the user is to type in an entire string, such as a file name or a figure label. All control keys except backspace and Return are ignored, and the measure is the string as it appears at the time of the trigger. In RAW mode, on the other hand, each character typed, including control characters, is a trigger and is returned individually as the measure. This mode is used when individual keyboard characters act as commands—for example, for moving the cursor, for simple editing operations, or for video-game actions. RAW mode provides no echo, whereas EDIT mode echoes the string on the screen and displays a **text cursor** (such as an underscore or block character) where the next character to be typed will appear. Each backspace causes the text cursor to back up and to erase one character.

When SRGP_waitEvent returns the device code KEYBOARD, the application obtains the measure associated with the event by calling

```
void SRGP_getKeyboard (char *measure , int buffersize);
```

When the keyboard device is active in RAW mode, its measure is always exactly one character in length. In this case, the first character of the measure string returns the RAW measure.

The program shown in Prog. 2.6 demonstrates the use of EDIT mode. It receives a list of file names from the user, deleting each file so entered. When the user enters a null string (by pressing Return without typing any other characters), the interaction ends. During the interaction, the program waits indefinitely for the user to enter the next string.

Although this code explicitly specifies where the text prompt is to appear, it does not specify where the user's input string is typed (and corrected with the backspace). The location of this keyboard echo is specified by the programmer, as discussed in Section 2.2.7.

The locator device. The trigger event for the locator device is a press or release of a mouse button. When SRGP_waitEvent returns the device code LOCATOR, the application obtains the measure associated with the event by calling

```
void SRGP_getLocator ( locatorMeasure *measure );
```

Typically, the **position** field of the measure is used to determine in which area of the screen the user designated the point. For example, if the locator cursor is in a rectangular region where a menu button is displayed, the event should be interpreted as a request for some action; if it is in the main drawing area, the point might be inside a previously drawn object to indicate it should be selected, or in an *empty* region to indicate where a new object should be placed.

Program 2.6

EDIT-mode keyboard interaction.

```
#define KEYMEASURE_SIZE 80
SRGP_setInputMode(KEYBOARD, EVENT);      /* Assume only keyboard is active */
SRGP_setKeyboardProcessingMode(EDIT);
pt = SRGP_defPoint( 100, 100 );
SRGP_text( pt, "Specify one or more files to be deleted; to exit press Return\n" );

/* main event loop */
do {
   inputDev = SRGP_waitEvent( INDEFINITE );
   SRGP_getKeyboard( measure , KEYMEASURE_SIZE );
   if (strcoll(measure, "" ))
      DeleteFile(measure);                  /* DeleteFile does confirmation, etc. */
}
while ( strcoll(measure, "" ) );
```

The pseudocode shown in Prog. 2.7 (similar to that shown previously for the keyboard) implements another use of the locator, letting the user specify points at which markers are to be placed. The user terminates the marker-placing loop by pressing the locator button while the cursor points to a screen button, a rectangle containing the text *quit*.

In this example, only the user's pressing of locator button 1 is significant; releases of the button are ignored. Note that the button must be released before the

next button-press event can take place—the event is triggered by a transition, not by a button state. Furthermore, to ensure that events coming from the other buttons do not disturb this interaction, the application tells SRGP which buttons are to trigger a locator event, by calling

```
void SRGP_setLocatorButtonMask ( int activeButtons );
```

Values for the button mask are LEFT_BUTTON_MASK, MIDDLE_BUTTON_MASK, and RIGHT_BUTTON_MASK. A composite mask is formed by logically or'ing individual values. The default locator-button mask is 1, but no matter what the mask is, all buttons always have a measure. On implementations that support fewer than three buttons, references to any nonexistent buttons are simply ignored by SRGP, and these buttons' measures always contain UP.

The function PickedQuitButton compares the measure position against the bounds of the quit button rectangle and returns a Boolean value signifying whether the user picked the quit button. This process is a simple example of **pick correlation,** as discussed in Section 2.2.6.

Program 2.7

Locator interaction.

```
#define QUIT 0
create the on-screen Quit button;
SRGP_setLocatorButtonMask( LEFT_BUTTON_MASK );
SRGP_setInputMode( LOCATOR, EVENT );          /* Assume only locator is active */
/* main event loop */
terminate = FALSE;
do {
    inputDev = SRGP_waitEvent( INDEFINITE );
    SRGP_getLocator( &measure );
    if (measure.buttonChord[QUIT] == DOWN ) {
        if PickedQuitButton( measure.position ) terminate  = TRUE;
        else
            SRGP_marker( measure.position );
    }
}
while ( !terminate );
```

Waiting for multiple events. The code fragments in Progs. 2.6 and 2.7 did not illustrate event mode's greatest advantage: the ability to wait for more than one device at the same time. SRGP queues events of enabled devices in chronological order and lets the application program take the first one off the queue when SRGP_waitEvent is called. Unlike hardware interrupts, which are processed in order of priorities, events are thus processed strictly in temporal order. The application examines the returned device code to determine which device caused the event.

The function shown in Prog. 2.8 allows the user to place any number of small circle markers anywhere within a rectangular drawing area. The user places a marker by pointing to the desired position and pressing button 1, then requests that the interaction be terminated either by pressing button 3 or by typing "q" or "Q".

Program 2.8

Use of several devices simultaneously.

```
#define PLACE_BUTTON 0
#define QUIT_BUTTON 2

generate initial screen layout;
SRGP_setInputMode( KEYBOARD, EVENT );
SRGP_setKeyboardProcessingMode( RAW );
SRGP_setInputMode( LOCATOR, EVENT );
SRGP_setLocatorButtonMask( LEFT_BUTTON_MASK | RIGHT_BUTTON_MASK );
                                        /* Ignore 2nd button */
/* Main event loop */
terminate = FALSE;
do {
   device = SRGP_waitEvent( INDEFINITE );
   switch ( device ) {
      case KEYBOARD:
         SRGP_getKeyboard( keyMeasure, lbuf );
         terminate = (keyMeasure[0] == 'q') || (keyMeasure[0] == 'Q');
         break;
      case LOCATOR: {
         SRGP_getLocator( &locMeasure );
         switch ( locMeasure.buttonOfMostRecentTransition ) {
            case PLACE_BUTTON:
               if (( locMeasure.buttonChord[PLACE_BUTTON] == DOWN )
                  && InDrawingArea( locMeasure.position ))
                     SRGP_marker( locMeasure.position );
               break;
            case QUIT_BUTTON:
               terminate = TRUE;
               break;
         }  /* button case */
      }  /* locator case */
   }  /* device case */
}
while (!terminate );
```

2.2.6 Pick Correlation for Interaction Handling

A graphics application customarily divides the screen area into regions dedicated to specific purposes. When the user presses the locator button, the application must determine exactly what screen button, icon, or other object was selected, if any, so that it can respond appropriately. This determination, called **pick correlation**, is a fundamental part of interactive graphics.

An application program using SRGP performs pick correlation by determining in which region the cursor is located, and then which object within that region, if any, the user is selecting. Points in an empty subregion might be ignored (if the point is between menu buttons in a menu, for example) or might specify the desired position for a new object (if the point lies in the main drawing area). Since a great many regions on the screen are upright rectangles, almost all the work for

pick correlation can be done by a simple, frequently used Boolean function that checks whether a given point lies in a given rectangle. The GEOM package distributed with SRGP includes this function (GEOM_ptInRect) as well as other utilities for coordinate arithmetic. (For more information on pick correlation, see Section 7.11.2.)

Let us look at a classic example of pick correlation. Consider a painting application with a **menu bar** across the top of the screen. This menu bar contains the names of pull-down menus, called menu **headers**. When the user picks a header (by placing the cursor on top of the header's text string and pressing a locator button), the corresponding **menu body** is displayed on the screen below the header and the header is highlighted. After the user selects an entry on the menu (by releasing the locator button), the menu body disappears and the header is unhighlighted. The rest of the screen contains the main drawing area in which the user can place and pick objects. The application, in creating each object, assigns it a unique positive integer identifier (ID) that is returned by the pick-correlation function for further processing of the object.

Program 2.9

High-level interaction scheme for menu handling.

```
void HighLevelInteractionHandler( locatorMeasure measureOfLocator )
{
    if ( GEOM_pointInRect( measureOfLocator.position, menuBarExtent ) {
            /* Find out which menu's header, if any, the user selected;
                Then, pull down that menu's body */
        menuID = CorrelateMenuBar( measureOfLocator.position );
        if ( menuID > 0 ) {
            chosenItemIndex = PerformPulldownMenuInteraction( menuID );
            if (chosenItemIndex > 0 )
                PerformActionChosenFromMenu(menuID, chosenItemIndex);
        }
    else        /* The user picked within the drawing area; detect what and respond */
    {
        objectID = CorrelateDrawingArea( measureOfLocator.position );
        if ( objectID > 0 ) ProcessObject( objectID );
    }
}
```

When a point is obtained from the locator via a button-down event, the high-level interaction-handling schema shown in Prog. 2.9 is executed; it is essentially a dispatching procedure that uses pick correlation within the menu bar or the main drawing area to divide the work among menu- and object-picking functions. First, if the cursor was in the menu bar, a subsidiary correlation procedure determines whether the user selected a menu header. If so, a procedure (detailed in Section 2.3.1) is called to perform the menu interaction; it returns an index specifying which item within the menu's body (if any) was chosen. The menu ID and item index together uniquely identify the action that should be taken in response. If the cursor was not in the menu bar but rather in the main drawing area, another subsidiary correlation procedure is called to determine what object was picked, if any. If an object was picked, a processing procedure is called to respond appropriately.

The function CorrelateMenuBar performs a finer correlation by calling GEOM_pointInRect once for each menu header in the menu bar; it accesses a data structure storing the rectangular screen extent of each header. The function CorrelateDrawingArea must do more sophisticated correlation because, typically, objects in the drawing area may overlap and are not necessarily rectangular.

2.2.7 Setting Device Measure and Attributes

Each input device has its own set of attributes, and the application can set these attributes to custom-tailor the feedback the device presents to the user. (The button mask presented earlier is also an attribute; it differs from those presented here in that it does not affect feedback.) Like output-primitive attributes, input-device attributes are set modally by specific functions. Attributes can be set at any time, whether or not the device is active.

In addition, each input device's measure, normally determined by the user's actions, can also be set by the application. Unlike input-device attributes, an input device's measure is reset to a default value when the device is deactivated; thus, upon reactivation, devices initially have predictable values, a convenience to the programmer and to the user. This automatic resetting can be overridden by explicitly setting a device's measure while it is inactive.

Locator echo attributes. Several types of echoes are useful for the locator. The programmer can control both echo type and cursor shape with

```
void SRGP_setLocatorEchoType ( echoType NO_ECHO / CURSOR /
    RUBBER_LINE /RUBBER_RECT );
```

The default is CURSOR, and SRGP implementations supply a cursor table from which an application selects a desired cursor shape (see the reference manual). A common use of the ability to specify the cursor shape dynamically is to provide feedback by changing the cursor shape according to the region in which the cursor lies. RUBBER_LINE and RUBBER_RECT echo are commonly used to specify a line or box. With these echoes set, SRGP automatically draws a continuously updated line or rectangle as the user moves the locator. The line or rectangle is defined by two points, the anchor point (another locator attribute) and the current locator position. Figure 2.12 illustrates the use of these two modes for user specification of a line and a rectangle.

In Fig. 2.12(a), the echo is a cross-hair cursor, and the user is about to press the locator button. The application initiates a rubber echo, anchored at the current locator position, in response to the button press. In parts (b) and (c), the user's movement of the locator device is echoed by the rubber primitive. The locator position in part (c) is returned to the application when the user releases the button, and the application responds by drawing a line or rectangle primitive and restoring normal cursor echo (see part d).

The anchor point for rubber echo is set with

```
void SRGP_setLocatorEchoRubberAnchor ( point position );
```

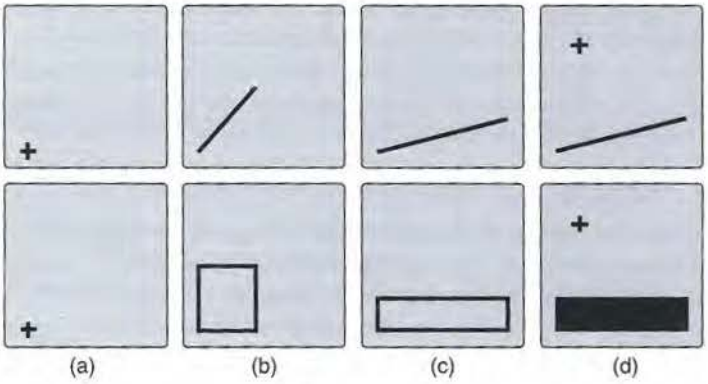


Figure 2.12 Rubber-echo scenarios.(a)Button press imitates echo. (b) Rubber primitive echoes locator device. (c) Locator position returns to application. (d) Application draws line and restores echo.

An application typically uses the *position* field of the measure obtained from the most recent locator-button–press event as the anchor position, since that button press typically initiates the rubber-echo sequence.

Locator measure control. The *position* portion of the locator measure is automatically reset to the center of the screen whenever the locator is deactivated. Unless the programmer explicitly resets it, the measure (and feedback position, if the echo is active) is initialized to that same position when the device is reactivated. At any time, whether the device is active or inactive, the programmer can reset the locator's measure (the *position* portion, not the fields concerning the buttons) by using

```
void SRGP_setLocatorMeasure ( point position );
```

Resetting the measure while the locator is inactive has no immediate effect on the screen, but resetting it while the locator is active changes the echo (if any) accordingly. Thus, if the program wants the cursor to appear initially at a position other than the center when the locator is activated, a call to SRGP_setLocatorMeasure with that initial position must precede the call to SRGP_setInputMode. This technique is commonly used to achieve continuity of cursor position: The last measure before the locator was deactivated is stored, and the cursor is returned to that position when it is reactivated.

Keyboard attributes and measure control. Unlike the locator, whose echo is positioned to reflect movements of a physical device, there is no obvious screen position for a keyboard device's echo. The position is thus an attribute (with an implementation-specific default value) of the keyboard device that can be set via

```
void SRGP_setKeyboardEchoOrigin ( point origin );
```

The default measure for the keyboard is automatically reset to the null string when the keyboard is deactivated. Setting the measure explicitly to a nonnull initial value just before activating the keyboard is a convenient way to present a default input string (displayed by SRGP as soon as echoing begins) that the user can accept as is or modify before pressing the Return key, thereby minimizing typing. The keyboard's measure, a character string, is set via

```
void SRGP_setKeyboardMeasure ( char *measure );
```

2.3 RASTER GRAPHICS FEATURES

By now, we have introduced most of the features of SRGP. This section discusses the remaining facilities that take particular advantage of raster hardware, especially the ability to save and restore pieces of the screen as they are overlaid by other images, such as windows or temporary menus. Such image manipulations are done under control of window- and menu-manager application programs. We also introduce offscreen bitmaps for storing windows and menus, and we discuss the use of clipping rectangles.

2.3.1 Canvases

The best way to make complex icons or menus appear and disappear quickly is to create them once in memory and then to copy them onto the screen as needed. Raster graphics packages do this by generating the primitives in invisible, offscreen bitmaps or pixmaps of the requisite size, called **canvases** in SRGP, and then copying the canvases to and from display memory. This technique is, in effect, a type of buffering. Moving blocks of pixels back and forth is faster, in general, than is regenerating the information, given the existence of the fast SRGP_copyPixel operation that we shall discuss soon.

An SRGP canvas is a data structure that stores an image as a 2D array of pixels. It also stores some control information concerning the size and attributes of the image. Each canvas represents its image in its own Cartesian coordinate system, which is identical to that of the screen shown in Fig. 2.1; in fact, the screen is itself a canvas, special solely because it is the only canvas that is displayed. To make an image stored in an off-screen canvas visible, the application must copy it onto the screen canvas. Beforehand, the portion of the screen image on which the new image—for example, a menu—will appear can be saved by copying the pixels in that region to an offscreen canvas. When the menu selection has taken place, the screen image is restored by copying back these pixels.

At any given time, there is one *currently active* canvas: the canvas into which new primitives are drawn and to which new attribute settings apply. This canvas may be the screen canvas (the default we have been using) or an offscreen canvas. The coordinates passed to the primitive functions are expressed in terms of the local coordinate space of the currently active canvas. Each canvas also has its own complete set of SRGP attributes, which affect all drawing on that canvas and are

(5.77)

R_z. See
both U

6 Viewing in 3D

The 3D viewing process is inherently more complex than is the 2D viewing process. In 2D, we simply specify a window on the 2D world and a viewport on the 2D view surface. Conceptually, objects in the world are clipped against the window and are then transformed into the viewport for display. The extra complexity of 3D viewing is caused in part by the added dimension and in part by the fact that display devices are only 2D. Although 3D viewing may seem overwhelming at first, it is less daunting when viewed as a series of easily understood steps, many of which we have prepared for in earlier chapters. Thus, we begin with a précis of the 3D viewing process to help guide you through this chapter.

6.1 THE SYNTHETIC CAMERA AND STEPS IN 3D VIEWING

A useful metaphor for creating 3D scenes is the notion of a **synthetic camera**, a concept illustrated in Fig. 6.1. We imagine that we can move our camera to any location, orient it in any way we wish, and, with a snap of the shutter, create a 2D image of a 3D object—the speedboat, in this case. At our bidding, the camera can become a motion-picture camera, enabling us to create an animated sequence that shows the object in a variety of orientations and magnifications. The camera, of course, is really just a computer program that produces an image on a display screen, and the object is a 3D dataset comprising a collection of points, lines, and surfaces. Figure 6.1 also shows that the camera and the 3D object each have their own coordinate system: u, v, n for the camera, and x, y, z for the object. We shall

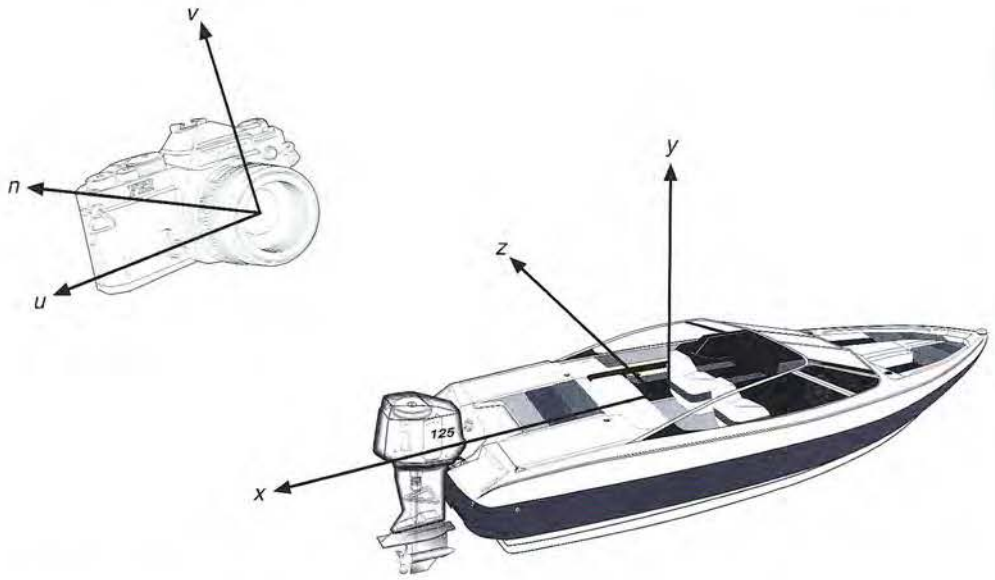


Figure 6.1 A synthetic camera photographing a 3D object.

discuss the significance of these coordinate systems later in this chapter. We note here that they provide an important independence of representation.

While the synthetic camera is a useful concept, there is a bit more to producing an image than just pushing a button. Creation of our "photo" is actually accomplished as a series of steps, which are described now.

- *Specification of projection type.* We resolve the mismatch between 3D objects and 2D displays by introducing **projections**, which transform 3D objects onto a 2D projection plane. Much of this chapter is devoted to projections: what they are, what their mathematics is, and how they are used in a current graphics subroutine package, PHIGS [ANSI88]. We concentrate on the two most important projections, **perspective** and **parallel orthographic**. The use of projections is also discussed further in Chapter 7.

- *Specification of viewing parameters.* Once a desired type of projection has been determined, we must specify the conditions under which we want to view the 3D real-world dataset, or the scene to be rendered. Given the world coordinates of the dataset, this information includes the position of the viewer's eye and the location of the viewing plane—the surface where the projection is ultimately displayed. We shall use two coordinate systems—that of the scene and another that we call the **viewing** or **eye coordinate system.** By varying any or all of these parameters, we can achieve any representation of the scene we wish, including viewing its interior, when that makes sense.

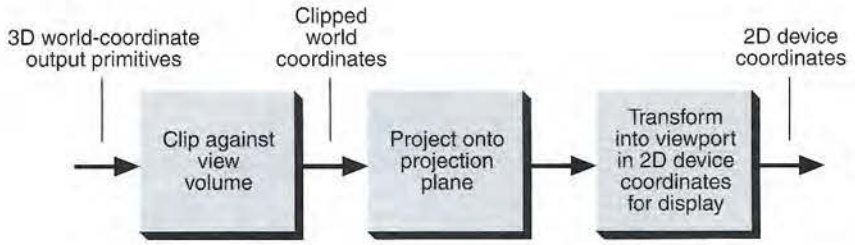


Figure 6.2 Conceptual model of the 3D viewing process.

- *Clipping in three dimensions.* Just as we must confine the display of a 2D scene to lie within the boundaries of our specified window, so too must we cull out portions of a 3D scene that are not candidates for ultimate display. We may, in fact, want to ignore parts of the scene that are behind us or are too far distant to be clearly visible. This action requires clipping against a view volume— a more complex process than that represented by the algorithms we have studied so far. Because of the wide variability of potential view volumes, we shall invest some effort in defining a canonical **view volume**—one against which we can efficiently apply a standardized clipping algorithm.

- *Projection and display.* Finally, the contents of the projection of the view volume onto the projection plane, called the **window,** are transformed (mapped) into the viewport for display.

Figure 6.2 shows the major steps in this conceptual model of the 3D viewing process, which is the model presented to the users of numerous 3D graphics subroutine packages. Just as with 2D viewing, a variety of strategies can be used to implement the viewing process. The strategies do not have to be identical to the conceptual model, as long as the results are those defined by the model. A typical implementation strategy for wire-frame line drawings is described in Section 6.6. For graphics systems that perform visible-surface determination and shading, a somewhat different pipeline, discussed in Chapter 14, is used.

6.2 PROJECTIONS

In general, projections transform points in a coordinate system of dimension n into points in a coordinate system of dimension less than n. In fact, computer graphics has long been used for studying n-dimensional objects by projecting them into 2D for viewing [NOLL67]. Here, we shall limit ourselves to the projection from 3D to 2D. The projection of a 3D object is defined by straight projection rays, called **projectors**, emanating from a **center of projection**, passing through each point of the object, and intersecting a **projection plane** to form the projection. In general,

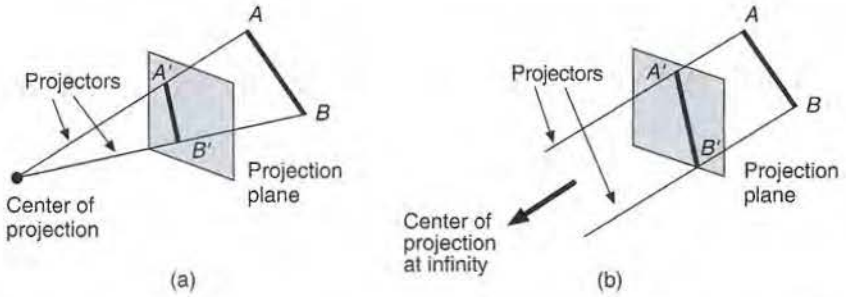


Figure 6.3 Two different projections of the same line. (a) Line AB and its perspective projection $A'B'$. (b) Line AB and its parallel projection $A'B'$. Projectors AA' and BB' are parallel.

the center of projection is a finite distance away from the projection plane. For some types of projections, however, it is convenient to think in terms of a center of projection that tends to be infinitely far away; we shall explore this concept further in Section 6.2.1. Figure 6.3 shows two different projections of the same line. Fortunately, the projection of a line is itself a line, so only line endpoints need to be projected.

The class of projections with which we deal here is known as **planar geometric projections,** because the projection is onto a plane rather than onto a curved surface, and uses straight rather than curved projectors. Many cartographic projections are either nonplanar or nongeometric.

Planar geometric projections, hereafter referred to simply as **projections**, can be divided into two basic classes: **perspective** and **parallel**. The distinction lies in the relation of the center of projection to the projection plane. If the distance from the one to the other is finite, then the projection is perspective; as the center of projection moves farther and farther away, the projectors passing through any particular object get closer and closer to being parallel to each other. Figure 6.3 illustrates these two cases. The parallel projection is so named because, with the center of projection infinitely distant, the projectors are parallel. When we define a perspective projection, we explicitly specify its **center of projection**; for a parallel projection, we give its **direction of projection**. The center of projection, being a point, has homogeneous coordinates of the form $(x, y, z, 1)$. Since the direction of projection is a vector (i.e., a difference between points), we can compute it by subtracting two points $d = (x, y, z, 1) - (x', y', z', 1) = (a, b, c, 0)$. Thus, **directions** and **points at infinity** correspond in a natural way. In the limit, a perspective projection whose center of projection tends to a point at infinity becomes a parallel projection.

The visual effect of a perspective projection is similar to that of photographic systems and of the human visual system, and is known as **perspective foreshortening**: The size of the perspective projection of an object varies inversely with the distance of that object from the center of projection. Thus, although the perspective projection of objects tends to look realistic, it is not particularly useful for recording the exact shape and measurements of the objects; distances cannot be taken

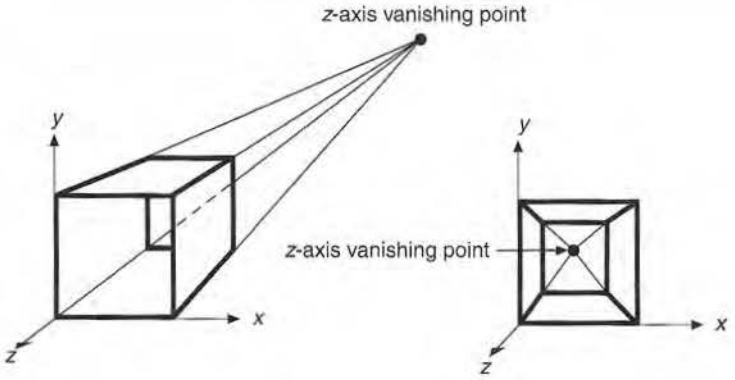


Figure 6.4 One-point perspective projections of a cube onto a plane cutting the z axis, showing vanishing point of lines perpendicular to projection plane.

from the projection, angles are preserved on only those faces of the object parallel to the projection plane, and parallel lines do not in general project as parallel lines.

The parallel projection is a less realistic view because perspective foreshortening is lacking, although there can be different constant foreshortenings along each axis. The projection can be used for exact measurements, and parallel lines do remain parallel. As in the perspective projection, angles are preserved only on faces of the object parallel to the projection plane.

The different types of perspective and parallel projections are discussed and illustrated at length in the comprehensive paper by Carlbom and Paciorek [CARL78]. In Sections 6.2.1 and 6.2.2, we summarize the basic definitions and characteristics of the more commonly used projections; we then move on, in Section 6.3, to understand how the projections are specified to PHIGS.

6.2.1 Perspective Projections

The perspective projections of any set of parallel lines that are not parallel to the projection plane converge to a **vanishing point**. In 3D, the parallel lines meet only at infinity, so the vanishing point can be thought of as the projection of a point at infinity. There is, of course, an infinity of vanishing points, one for each of the infinity of directions in which a line can be oriented.

If the set of lines is parallel to one of the three principal axes, the vanishing point is called an **axis vanishing point**. There are at most three such points, corresponding to the number of principal axes cut by the projection plane. For example, if the projection plane cuts only the z axis (and is therefore normal to it), only the z axis has a principal vanishing point, because lines parallel to either the y or x axes are also parallel to the projection plane and have no vanishing point.

Perspective projections are categorized by their number of principal vanishing points and therefore by the number of axes the projection plane cuts. Figure 6.4

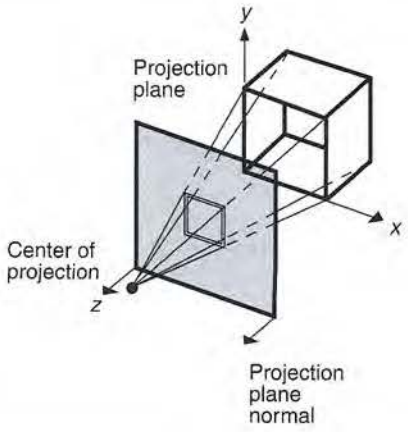


Figure 6.5 Construction of one-point perspective projection of cube onto plane cutting the z axis. The projection-plane normal is parallel to z axis. (Adapted from [CARL78], Association for Computing Machinery, Inc.; used by permission.)

shows two different one-point perspective projections of a cube. It is clear that they are one-point projections because lines parallel to the x and y axes do not converge; only lines parallel to the z axis do so. Figure 6.5 shows the construction of a one-point perspective with some of the projectors and with the projection plane cutting only the z axis.

Figure 6.6 shows the construction of a two-point perspective. Notice that lines parallel to the y axis do not converge in the projection. Two-point perspective is commonly used in architectural, engineering, industrial design, and advertising drawings. Three-point perspectives are used less frequently, since they add little realism beyond that afforded by the two-point perspective.

6.2.2 Parallel Projections

Parallel projections are categorized into two types, depending on the relation between the direction of projection and the normal to the projection plane. In **orthographic** parallel projections, these directions are the same (or are the reverse of each other), so the direction of projection is normal to the projection plane. For the **oblique** parallel projection, they are not.

The most common types of orthographic projections are the **front-elevation**, **top-elevation** or **plan-elevation**, and **side-elevation** projections. In all these, the projection plane is perpendicular to a principal axis, which is therefore the direction of projection. Figure 6.7 shows the construction of these three projections; they are often used in engineering drawings to depict machine parts, assemblies, and buildings, because distances and angles can be measured from them. Since each projection depicts only one face of an object, however, the 3D nature of the

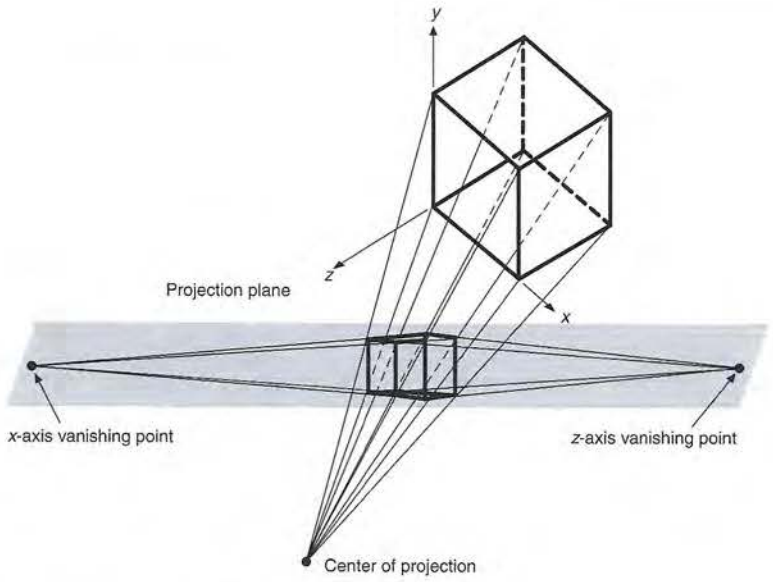


Figure 6.6 Two-point perspective projection of a cube. The projection plane cuts the *x* and *z* axes.

projected object can be difficult to deduce, even if several projections of the same object are studied simultaneously.

Axonometric orthographic projections use projection planes that are not normal to a principal axis and therefore show several faces of an object at once.

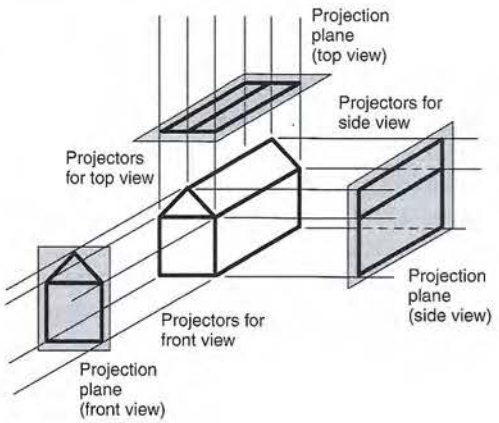


Figure 6.7 Construction of three orthographic projections.

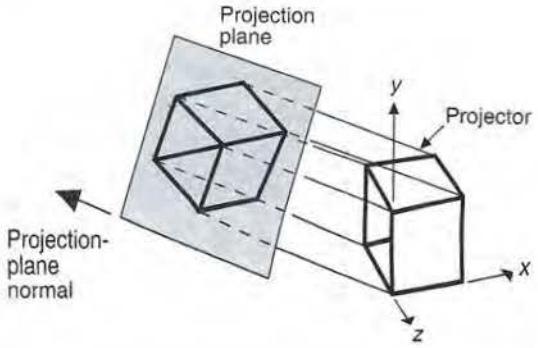


Figure 6.8 Construction of an isometric projection of a unit cube. (Adapted from [CARL78], Association for Computing Machinery, Inc.; used by permission.)

They resemble the perspective projection in this way, but differ in that the foreshortening is uniform, rather than being related to the distance from the center of projection. Parallelism of lines is preserved, but angles are not, and distances can be measured along each principal axis (in general, with different scale factors).

The **isometric projection** is a commonly used axonometric projection. The projection-plane normal (and therefore the direction of projection) makes equal angles with each principal axis. If the projection-plane normal is (d_x, d_y, d_z), then we require that $|d_x| = |d_y| = |d_z|$ or $\pm d_x = \pm d_y = \pm d_z$. There are just eight directions (one in each octant) that satisfy this condition. Figure 6.8 shows the construction of an isometric projection along one such direction, $(1, -1, -1)$.

The isometric projection has the useful property that all three principal axes are equally foreshortened, allowing measurements along the axes to be made to the same scale (hence the name: *iso* for equal, *metric* for measure). In addition, the projections of the principal axes make equal angles of 120° with one another.

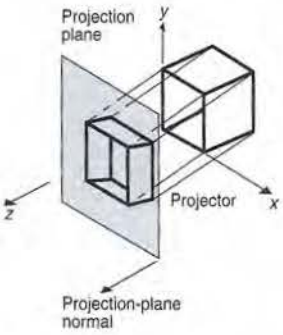


Figure 6.9 Construction of oblique projection. (Adapted from [CARL78], Association for Computing Machinery, Inc.; used by permission.)

Oblique projections, the second class of parallel projections, differ from orthographic projections in that the projection-plane normal and the direction of projection differ. Oblique projections combine properties of the front, top, and side orthographic projections with those of the axonometric projection: the projection plane is normal to a principal axis, so the projection of the face of the object parallel to this plane allows measurement of angles and distances. Other faces of the object project also, allowing distances along principal axes, but not angles, to be measured. Oblique projections are widely, although not exclusively, used in this text because of these properties and because they are easy to draw. Figure 6.9 shows the construction of an oblique projection. Notice that the projection-plane normal and the direction of projection are not the same. Several types of oblique projections are described in [FOLE90].

Figure 6.10 shows the logical relationships among the various types of projections. The common thread uniting all the projections is that they involve a projection plane and either a center of projection for the perspective projection, or a

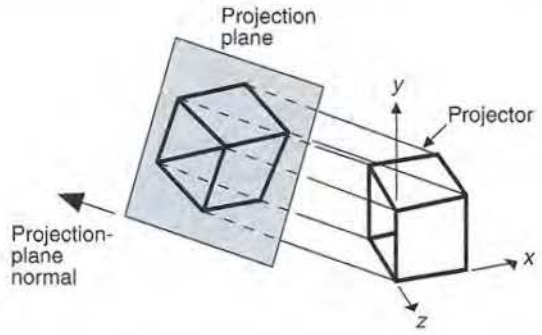


Figure 6.8 Construction of an isometric projection of a unit cube. (Adapted from [CARL78], Association for Computing Machinery, Inc.; used by permission.)

They resemble the perspective projection in this way, but differ in that the foreshortening is uniform, rather than being related to the distance from the center of projection. Parallelism of lines is preserved, but angles are not, and distances can be measured along each principal axis (in general, with different scale factors).

The **isometric projection** is a commonly used axonometric projection. The projection-plane normal (and therefore the direction of projection) makes equal angles with each principal axis. If the projection-plane normal is (d_x, d_y, d_z), then we require that $|d_x| = |d_y| = |d_z|$ or $\pm d_x = \pm d_y = \pm d_z$. There are just eight directions (one in each octant) that satisfy this condition. Figure 6.8 shows the construction of an isometric projection along one such direction, $(1, -1, -1)$.

The isometric projection has the useful property that all three principal axes are equally foreshortened, allowing measurements along the axes to be made to the same scale (hence the name: *iso* for equal, *metric* for measure). In addition, the projections of the principal axes make equal angles of 120° with one another.

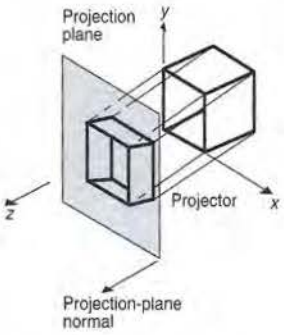


Figure 6.9
Construction of oblique projection. (Adapted from [CARL78], Association for Computing Machinery, Inc.; used by permission.)

Oblique projections, the second class of parallel projections, differ from orthographic projections in that the projection-plane normal and the direction of projection differ. Oblique projections combine properties of the front, top, and side orthographic projections with those of the axonometric projection: the projection plane is normal to a principal axis, so the projection of the face of the object parallel to this plane allows measurement of angles and distances. Other faces of the object project also, allowing distances along principal axes, but not angles, to be measured. Oblique projections are widely, although not exclusively, used in this text because of these properties and because they are easy to draw. Figure 6.9 shows the construction of an oblique projection. Notice that the projection-plane normal and the direction of projection are not the same. Several types of oblique projections are described in [FOLE90].

Figure 6.10 shows the logical relationships among the various types of projections. The common thread uniting all the projections is that they involve a projection plane and either a center of projection for the perspective projection, or a

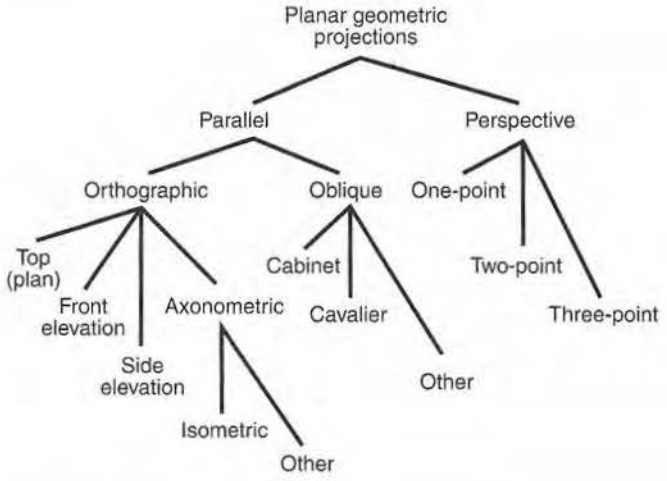


Figure 6.10 The subclasses of planar geometric projections. **Plan view** is another term for a top view. **Front** and **side** are often used without the term **elevation**.

direction of projection for the parallel projection. We can unify the parallel and perspective cases further by thinking of the center of projection as defined by the direction to the center of projection from some reference point, and the distance to the reference point. When this distance increases to infinity, the projection becomes a parallel projection. Hence, we can also say that the common thread uniting these projections is that they involve a projection plane, a direction to the center of projection, and a distance to the center of projection. In Section 6.3, we consider how to integrate some of these types of projections into the 3D viewing process.

6.3 SPECIFICATION OF AN ARBITRARY 3D VIEW

As suggested by Fig. 6.2, 3D viewing involves not just a projection, but also a view volume against which the 3D world is clipped. The projection and view volume together provide all the information that we need to clip and project into 2D space. Then, the 2D transformation into physical device coordinates is straightforward. We now build on the concepts of planar-geometric projection introduced in Section 6.2 to show how to specify a view volume. The viewing approach and terminology presented here is that used in PHIGS.

The projection plane, henceforth called the **view plane** to be consistent with the graphics literature, is defined by a point on the plane called the **view reference point (VRP)** and a normal to the plane called the **view-plane normal (VPN)**. The

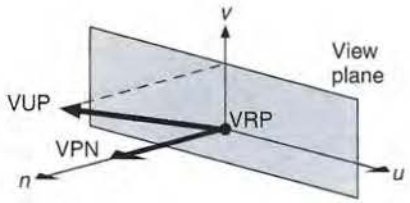


Figure 6.11 The view plane is defined by VPN and VRP; the *v* axis is defined by the projection of VUP along VPN onto the view plane. The *u* axis forms the right-handed VRC system with VPN and *v*.

view plane may be anywhere with respect to the world objects to be projected: It may be in front of, cut through, or be behind the objects.

Given the view plane, a window on the view plane is needed. The window's role is similar to that of a 2D window: Its contents are mapped into the viewport, and any part of the 3D world that projects onto the view plane outside of the window is not displayed. We shall see that the window also plays an important role in defining the view volume.

To define a window on the view plane, we need a means of specifying minimum and maximum window coordinates and the two orthogonal axes in the view plane along which to measure these coordinates. These axes are part of the 3D **viewing-reference coordinate (VRC)** system. The origin of the VRC system is the VRP. One axis of the VRC is VPN; this axis is called the *n* axis. A second axis of the VRC is found from the **view-up vector (VUP)**, which determines the *v*-axis direction on the view plane. The *v* axis is defined such that the projection of VUP parallel to VPN onto the view plane is coincident with the *v* axis. The *u*-axis direction is defined such that *u*, *v*, and *n* form a right-handed coordinate system, as in Fig. 6.11. The VRP and the two direction vectors VPN and VUP are specified in the right-handed world-coordinate system. (Some graphics packages use the *y* axis as VUP, but this convention is too restrictive and fails if VPN is parallel to the *y* axis, in which case VUP is undefined.)

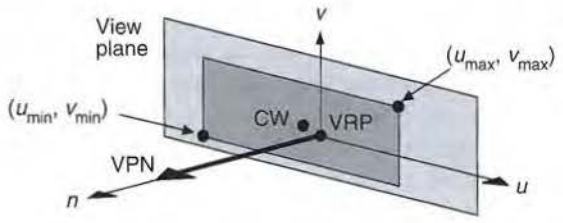


Figure 6.12 The viewing-reference coordinate system (VRC) is a right-handed system made up of the *u*, *v*, and *n* axes. The *n* axis is always the VPN. CW is the center of the window.

With the VRC system defined, the window's minimum and maximum u and v coordinates can be defined, as in Fig. 6.12. This figure illustrates that the window does not have to be symmetrical about the VRP, and explicitly shows the center of the window, CW.

The center of projection and direction of projection (DOP) are defined by a **projection reference point (PRP)** and an indicator of the projection type. If the projection type is perspective, then PRP is the center of projection. If the projection type is parallel, then the DOP is from the PRP to CW. The CW is in general not the VRP, which does not need even to be within the window bounds.

The PRP is specified in the VRC system, not in the world-coordinate system; thus, the position of the PRP relative to the VRP does not change as VUP or VRP is moved. The advantage of this scheme is that the programmer can specify the direction of projection required and then change VPN and VUP (hence changing VRC), without having to recalculate the PRP needed to maintain the desired projection. On the other hand, moving the PRP about to get different views of an object may be more difficult.

The **view volume** bounds that portion of the world that is to be clipped out and projected onto the view plane. For a perspective projection, the view volume is the semi-infinite pyramid with apex at the PRP and edges passing through the corners of the window. Figure 6.13 shows a perspective-projection view volume.

Positions behind the center of projection are not included in the view volume and thus are not projected. In reality, of course, our eyes see an irregularly shaped conelike view volume. However, a pyramidal view volume is mathematically more tractable, and is consistent with the concept of a rectangular viewport.

For parallel projections, the view volume is an infinite parallelepiped with sides parallel to the direction of projection, which is the direction from the PRP to the center of the window. Figure 6.14 shows a parallel-projection view volume and its relation to the view plane, window, and PRP.

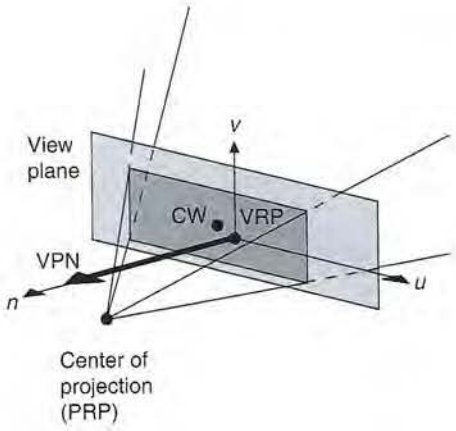


Figure 6.13 Semi-infinite pyramid view volume for perspective projection. CW is the center of the window.

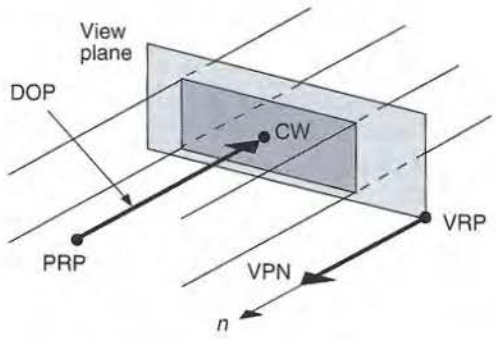


Figure 6.14 Infinite parallelepiped view volume of parallel orthographic projection. The VPN and direction of projection (DOP) are parallel. DOP is the vector from PRP to CW, and is parallel to the VPN.

At times, we might want the view volume to be finite, in order to limit the number of output primitives projected onto the view plane. Figures 6.15 and 6.16 show how the view volume is made finite with a **front clipping plane** and **back clipping plane**. These planes, sometimes called the **hither** and **yon planes,** are parallel to the view plane; their normal is the VPN. The planes are specified by the signed quantities **front distance** (F) and **back distance** (B) relative to the VRP and along the VPN, with positive distances in the direction of the VPN. For the view volume to be nonempty, the front distance must be algebraically greater than the back distance.

Limiting the view volume in this way can be useful to eliminate extraneous objects and to allow the user to concentrate on a particular portion of the world. Dynamic modification of either the front or rear distances can give the viewer a

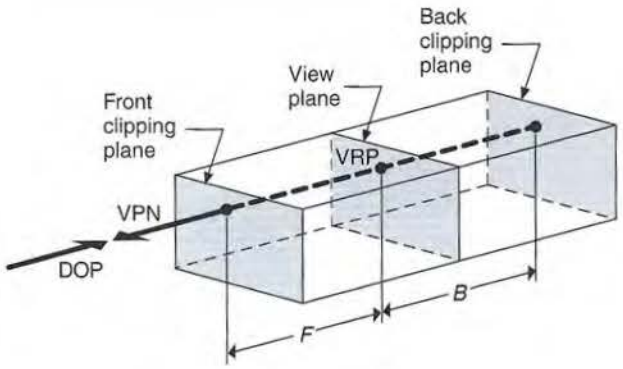


Figure 6.15 Truncated view volume for an orthographic parallel projection. DOP is the direction of projection.

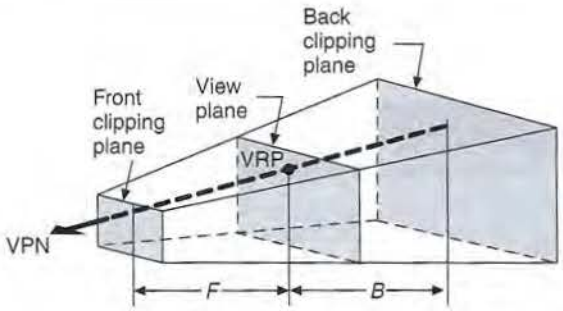


Figure 6.16 Truncated view volume for a perspective projection.

good sense of the spatial relationships between different parts of the object as these parts appear and disappear from view (see Chapter 12). For perspective projections there is an additional motivation. An object very distant from the center of projection projects onto the view surface as a "blob" of no distinguishable form. In displaying such an object on a plotter, the pen can wear through the paper; on a vector display, the CRT phosphor can be burned by the electron beam; and on a vector film recorder, the high concentration of light causes a fuzzy white area to appear. Also, an object very near the center of projection may extend across the window like so many disconnected pick-up sticks, with no discernible structure. Specifying the view volume appropriately can eliminate such problems.

How are the contents of the view volume mapped onto the display surface? First, consider the unit cube extending from 0 to 1 in each of the three dimensions of **normalized projection coordinates (NPC)**. The view volume is transformed into the rectangular solid of NPC, which extends from x_{min} to x_{max} along the x axis, from y_{min} to y_{max} along the y axis, and from z_{min} to z_{max} along the z axis. The front clipping plane becomes the z_{max} plane, and the back clipping plane becomes the z_{min} plane. Similarly, the u_{min} side of the view volume becomes the x_{min} plane, and the u_{max} side becomes the x_{max} plane. Finally, the v_{min} side of the view volume becomes the y_{min} plane, and the v_{max} side becomes the y_{max} plane. This rectangular solid portion of NPC, called a **3D viewport**, is within the unit cube.

The $z = 1$ face of this unit cube, in turn, is mapped into the largest square that can be inscribed on the display. To create a wire-frame display of the contents of the 3D viewport (which are the contents of the view volume), the z-component of each output primitive is simply discarded, and the output primitive is displayed. We shall see in Chapter 13 that hidden-surface removal simply uses the z-component to determine which output primitives are closest to the viewer and hence are visible.

PHIGS uses two 4×4 matrices, the view orientation matrix and the view mapping matrix, to represent the complete set of viewing specifications. The VRP, VPN, and VUP are combined to form the **view orientation matrix**, which transforms positions represented in world coordinates into positions represented in

VRC. This transformation takes the u, v, and n axes into the x, y, and z axes, respectively.

The view-volume specifications, given by PRP, u_{min}, u_{max}, v_{min}, v_{max}, F, and B, along with the 3D viewport specification, given by x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, and z_{max}, are combined to form the **view mapping matrix**, which transforms points in VRC to points in normalized projection coordinates. The subroutine calls that form the view orientation matrix and view mapping matrix are discussed in Section 7.3.4.

In Section 6.4, we see how to obtain various views using the concepts introduced in this section. In Section 6.5, the basic mathematics of planar geometric projections is introduced, whereas in Section 6.6, the mathematics and algorithms needed for the entire viewing operation are developed.

6.4 EXAMPLES OF 3D VIEWING

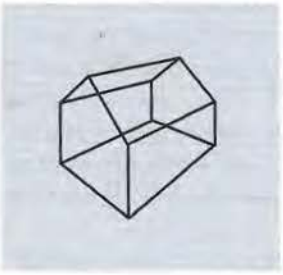


Figure 6.17
Two-point perspective projection of a house.

In this section, we consider how we can apply the basic viewing concepts introduced in Section 6.3 to create a variety of projections, such as that shown in Fig. 6.17. Because the house shown in this figure is used throughout this section, it will be helpful to remember its dimensions and position, which are indicated in Fig. 6.18. For each view discussed, we give a table showing the VRP, VPN, VUP, PRP, window, and projection type (perspective or parallel). The 3D viewport default, which is the unit cube in NPC, is assumed throughout this section. The notation (WC) or (VRC) is added to the table as a reminder of the coordinate system in which the viewing parameter is given. The form of the table is illustrated here for the default viewing specification used by PHIGS. The defaults are shown in Fig. 6.19(a). The view volume corresponding to these defaults is shown in Fig. 6.19(b). If the type of projection is perspective rather than parallel, then the view volume is the pyramid shown in Fig. 6.19(c).

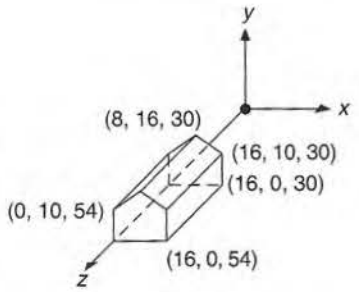


Figure 6.18 This house is used as an example of a world-coordinate dataset throughout this chapter. Its coordinates extend from 30 to 54 in z, from 0 to 16 in x, and from 0 to 16 in y.

8 Input Devices, Interaction Techniques, and Interaction Tasks

High-quality user interfaces are in many ways the *last frontier* in providing computing to a wide variety of users, since hardware and software costs are now low enough to bring significant computing capability to our offices and homes. Just as software engineering has recently given structure to an activity that once was totally ad hoc, so too the new area of user-interface engineering is generating user-interface principles and design methodologies.

The quality of the user interface often determines whether users enjoy or despise a system, whether the designers of the system are praised or damned, whether a system succeeds or fails in the market. The designer of an interactive graphics application must be sensitive to users' desire for easy-to-learn yet powerful interfaces.

The desktop user-interface metaphor, with its windows, icons, and pull-down menus, all making heavy use of raster graphics, is popular because it is easy to learn and requires little typing skill. Most users of such systems are not computer programmers and have little sympathy for the old style, hard-to-learn, keyboard-oriented command-language interfaces that many programmers take for granted. The process of designing, testing, and implementing a user interface is complex; see [FOLE90; SHNE86; MAYH90] for guidelines and methodologies.

We focus in this chapter on input devices, interaction technologies, and interaction tasks. These are the basic building blocks from which user interfaces are constructed. Input devices are the pieces of hardware by which a user enters information into a computer system. We have already discussed many such devices in Chapter 4. In this chapter, we introduce additional devices, and discuss reasons for preferring one device over another. In Section 8.1.6, we describe input devices oriented specifically toward 3D interaction. We continue to use the logical device

categories of locator, keyboard, choice, valuator, and pick used by SRGP, SPHIGS, and other device-independent graphics subroutine packages. We also discuss basic elements of user interfaces: **interaction techniques** and **interaction tasks.** Interaction techniques are ways to use input devices to enter information into the computer, whereas interaction tasks classify the fundamental types of information entered with the interaction techniques. Interaction techniques are the primitive building blocks from which a user interface is crafted.

An **interaction task** is the entry of a unit of information by the user. The four basic interaction tasks are **position, text, select,** and **quantify.** The unit of information input in a position interaction task is of course a position. Similarly, the text task yields a text string; the select task yields an object identification; and the quantify task yields a numeric value. Many different **interaction techniques** can be used for a given interaction task. For instance, a selection task can be carried out by using a mouse to select items from a menu, using a keyboard to enter the name of the selection, pressing a function key, or using a speech recognizer. Similarly, a single device can be used for different tasks: A mouse is often used for both positioning and selecting.

Interaction tasks are distinct from the logical input devices discussed in earlier chapters. Interaction tasks are defined by *what* the user accomplishes, whereas logical input devices categorize *how* that task is accomplished by the application program and the graphics package. Interaction tasks are user-centered, whereas logical input devices are a programmer and graphics-package concept.

Many of the topics in this chapter are discussed in much greater depth elsewhere; see the texts by Baecker and Buxton [BAEC87], Hutchins, Hollan, and Norman [HUTC86], Mayhew [MAYH90], Norman [NORM88], Rubenstein and Hersh [RUBE84], Shneiderman [SHNE86], and [FOLE90]; the reference book by Salvendy [SALV87]; and the survey by Foley, Wallace, and Chan [FOLE84].

8.1 INTERACTION HARDWARE

Here, we introduce some interaction devices not covered in Section 4.5, elaborate on how they work, and discuss the advantages and disadvantages of various devices. The presentation is organized around the logical-device categorization of Section 4.5, and can be thought of as a more detailed continuation of that section.

The advantages and disadvantages of various interaction devices can be discussed on three levels: device, task, and dialogue (i.e., sequence of several interaction tasks). The **device level** centers on the hardware characteristics per se, and does not deal with aspects of the device's use controlled by software. At the device level, for example, we note that one mouse shape may be more comfortable to hold than another, and that a data tablet takes up more space than a joystick.

At the **task level**, we might compare interaction techniques using different devices for the same task. Thus, we might assert that experienced users can often enter commands more quickly via function keys or a keyboard than via menu

selection, or that users can pick displayed objects more quickly using a mouse than they can using a joystick or cursor control keys.

At the **dialogue level**, we consider not just individual interaction tasks, but also sequences of such tasks. Hand movements between devices take time: Although the positioning task is generally faster with a mouse than with cursor-control keys, cursor-control keys may be faster than a mouse *if* the user's hands are already on the keyboard and will need to be on the keyboard for the next task in sequence after the cursor is repositioned.

Important considerations at the device level, discussed in this section, are the device footprints—(the **footprint** of a piece of equipment is the work area it occupies)—operator fatigue, and device resolution. Other important device issues—such as cost, reliability, and maintainability—change too quickly with technological innovation to be discussed here.

8.1.1 Locator Devices

It is useful to classify locator devices according to three independent characteristics: absolute or relative, direct or indirect, and discrete or continuous.

Absolute devices, such as a data tablet or touch panel, have a frame of reference, or origin, and report positions with respect to that origin. **Relative** devices—such as mice, trackballs, and velocity-control joysticks—have no absolute origin and report only changes from their former position. A relative device can be used to specify an arbitrarily large change in position: A user can move a mouse along the desktop, lift it up and place it back at its initial starting position, and move it again. A data tablet can be programmed to behave as a relative device: The first (x, y) coordinate position read after the pen goes from *far* to *near* state (i.e., close to the tablet) is subtracted from all subsequently read coordinates to yield only the change in x and y, which is added to the previous (x, y) position. This process is continued until the pen again goes to *far* state.

Relative devices cannot be used readily for digitizing drawings, whereas absolute devices can be. The advantage of a relative device is that the application program can reposition the cursor anywhere on the screen.

With a **direct** device—such as a touch screen—the user points directly at the screen with a finger or surrogate finger; with an **indirect** device—such as a tablet, mouse, or joystick—the user moves a cursor on the screen using a device not on the screen. New forms of eye–hand coordination must be learned for the latter; the proliferation of computer games in homes and arcades, however, have created an environment in which many casual computer users have already learned these skills. However, direct pointing can cause arm fatigue, especially among casual users.

A **continuous** device is one in which a smooth hand motion can create a smooth cursor motion. Tablets, joysticks, and mice are all continuous devices, whereas cursor-control keys are **discrete** devices. Continuous devices typically allow more natural, easier, and faster cursor movement than do discrete devices. Most continuous devices also permit easier movement in arbitrary directions than do cursor control keys.

Speed of cursor positioning with a continuous device is affected by th **control-to-display ratio**, commonly called the C/D ratio [CHAP72]; it is the rati between hand movement (the control) and cursor movement (the display). A larg ratio is good for accurate positioning, but makes rapid movements tedious; a sma ratio is good for speed but not for accuracy. Fortunately, for a relative positionin device, the ratio need not be constant, but can be changed adaptively as a functio of control-movement speed. Rapid movements indicate the user is making a gros hand movement, so a small ratio is used; as the speed decreases, the C/D ratio i increased. This variation of C/D ratio can be set up so that users can use a mouse t position a cursor accurately across a 15-inch screen without repositioning the wrist! For indirect discrete devices (cursor-control keys), there is a similar tech nique: The distance the cursor is moved per unit time is increased as a function o the time the key has been held down.

Precise positioning is difficult with direct devices, if the arm is unsupporte and extended toward the screen. Try writing your name on a blackboard in thi pose, and compare the result to your normal signature. This problem can be miti gated if the screen is angled close to horizontal. Indirect devices, on the other hand allow the heel of the hand to rest on a support, so that the fine motor control of th fingers can be used more effectively. Not all continuous indirect devices ar equally satisfactory for drawing, however. Try writing your name with a joystick, mouse, and a tablet pen stylus. Using the stylus is fastest, and the result is mos pleasing.

8.1.2 Keyboard Devices

The well-known QWERTY keyboard has been with us for many years. It is ironi that this keyboard was originally designed to *slow down* typists, so that the type writer hammers would not be so likely to jam. Studies have shown that the newe Dvořák keyboard [DVOR43], which places vowels and other high-frequenc characters under the home positions of the fingers, is somewhat faster than is th QWERTY design [GREE87]. It has not been widely accepted. Alphabeticall organized keyboards are sometimes used when many of the users are nontypist But more and more people are being exposed to QWERTY keyboards, and severa experiments have shown no advantage of alphabetic over QWERTY keyboard [HIRS70; MICH71].

Other keyboard-oriented considerations, involving not hardware but softwa design, are arranging for a user to enter frequently used punctuation or correctio characters without needing to press the control or shift keys simultaneously, an assigning dangerous actions (such as delete) to keys that are distant from other fr quently used keys.

8.1.3 Valuator Devices

Some valuators are **bounded**, like the volume control on a radio—the dial can turned only so far before a stop is reached that prevents further turning. A bounc valuator inputs an absolute quantity. A continuous-turn potentiometer, on the oth

hand, can be turned an **unbounded** number of times in either direction. Given an initial value, the unbounded potentiometer can be used to return absolute values; otherwise, the returned values are treated as relative values. The provision of some sort of echo enables the user to determine what relative or absolute value is currently being specified. The issue of C/D ratio, discussed in the context of positioning devices, also arises in the use of slide and rotary potentiometers to input values.

8.1.4 Choice Devices

Function keys are a common choice device. Their placement affects their usability: Keys mounted on the CRT bezel are harder to use than are keys mounted in the keyboard or in a nearby separate unit. A foot switch can be used in applications in which the user's hands are engaged yet a single switch closure must be frequently made.

8.1.5 Other Devices

Here we discuss some of the less common, and in some cases experimental, 2D interaction devices. Voice recognizers, which are useful because they free the user's hands for other uses, apply a pattern-recognition approach to the waveforms created when we speak a word. The waveform is typically separated into a number of different frequency bands, and the variation over time of the magnitude of the waveform in each band forms the basis for the pattern matching. However, mistakes can occur in the pattern matching, so it is especially important that an application using a recognizer provide convenient correction capabilities.

Voice recognizers differ in whether they must be trained to recognize the waveforms of a particular speaker, and whether they can recognize connected speech as opposed to single words or phrases. Speaker-independent recognizers have vocabularies that include the digits and up to 1000 words.

The data tablet has been extended in several ways. Many years ago, Herot and Negroponte used an experimental pressure-sensitive stylus [HERO76]: High pressure and a slow drawing speed implied that the user was drawing a line with deliberation, in which case the line was recorded exactly as drawn; low pressure and fast speed implied that the line was being drawn quickly, in which case a straight line connecting the endpoints was recorded. A more recent commercially available tablet [WACO93] incorporates such a pressure-sensitive stylus. The resulting three degrees of freedom reported by the tablet can be used in various creative ways.

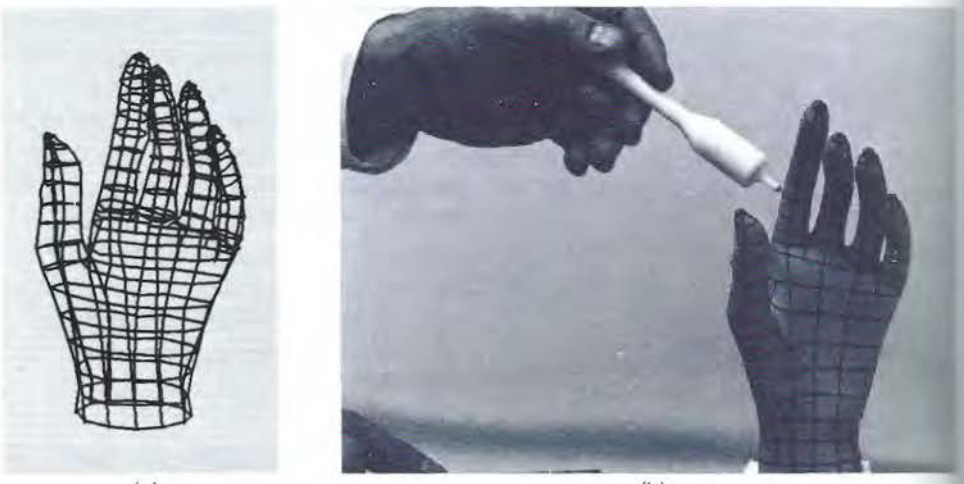
8.1.6 3D Interaction Devices

Some of the 2D interaction devices are readily extended to 3D. Joysticks can have a shaft that twists for a third dimension (see Fig. 4.15). Trackballs can be made to sense rotation about the vertical axis in addition to that about the two horizontal axes. In both cases, however, there is no direct relationship between hand movements with the device and the corresponding movement in 3-space.

A number of devices can record 3D hand movements. For example, the Polhemus 3SPACE 3D position and orientation sensor uses electromagnetic coupling between three transmitter antennas and three receiver antennas. The transmitter antenna coils, which are at right angles to one another to form a Cartesian coordinate system, are pulsed in turn. The receiver has three similarly arranged receiver antennas; each time a transmitter coil is pulsed, a current is induced in each of the receiver coils. The strength of the current depends both on the distance between the receiver and transmitter and on the relative orientation of the transmitter and receiver coils. The combination of the nine current values induced by the three successive pulses is used to calculate the 3D position and orientation of the receiver. Figure 8.1 shows this device in use for one of its common purposes: digitizing a 3D object.

The DataGlove records hand position and orientation as well as finger movements. As shown in Fig. 8.2, it is a glove covered with small, lightweight sensors. Each sensor is a short length of fiberoptic cable, with a light-emitting diode (LED) at one end and a phototransistor at the other end. The surface of the cable is roughened in the area where it is to be sensitive to bending. When the cable is flexed, some of the LED's light is lost, so less light is received by the phototransistor. In addition, a Polhemus position and orientation sensor records hand movements. Wearing the DataGlove, a user can grasp objects, move and rotate them, and then release them, thus providing very natural interaction in 3D [ZIMM87]. Color Plate 6 illustrates this concept.

Considerable effort has been directed toward creating what are often called **artificial realities** or **virtual realities**; these are completely computer-generated environments with realistic appearance, behavior, and interaction techniques



(a) (b)

Figure 8.1 (a) The Polhemus 3D position sensor being used to digitize a 3D object. (b) A wireframe display of the result. (3Space digitizer courtesy of Polhemus, Inc., Colchester, VT.)

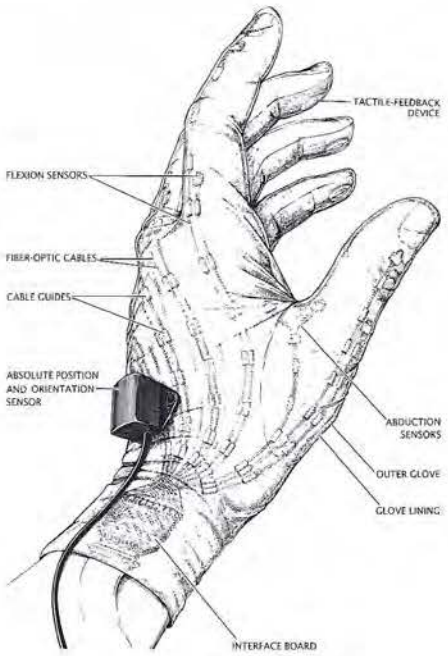


Figure 8.2 The VPL DataGlove, showing the fiberoptic cables that are used to sense finger movements, and the Polhemus position and orientation sensor. (From J. Foley, Interfaces for Advanced Computing, Copyright @ 1987 by *Scientific American, Inc.* All rights reserved.)

[FOLE87]. In one version, the user wears a head-mounted stereo display to show proper left- and right-eye views, a Polhemus sensor on the head allows changes in head position and orientation to cause changes to the stereo display, a DataGlove permits 3D interaction, and a microphone is used for issuing voice commands. Color Plate 7 shows this combination of equipment.

Several other technologies can be used to record 3D positions. In one, using optical sensors, LEDs are mounted on the user (either at a single point, such as the fingertip, or all over the body, to measure body movements). Light sensors are mounted high in the corners of a small, semidarkened room in which the user works, and each LED is intensified in turn. The sensors can determine the plane in which the LED lies, and the location of the LED is thus at the intersection of three planes. (A fourth sensor is normally used, in case one of the sensors cannot see the LED.) Small reflectors on the fingertips and other points of interest can replace the LEDs; sensors pick up reflected light rather than the LED's emitted light.

Krueger [KRUE83] has developed a sensor for recording hand and finger movements in 2D. A television camera records hand movements; image-processing techniques of contrast-enhancement and edge detection are used to find the

outline of the hand and fingers. Different finger positions can be interpreted as commands, and the user can grasp and manipulate objects, as in Color Plate 8. This technique could be extended to 3D through use of multiple cameras.

8.2 BASIC INTERACTION TASKS

With a basic interaction task, the user of an interactive system enters a unit of information that is meaningful in the context of the application. How large or small is such a unit? For instance, does moving a positioning device a small distance enter a unit of information? Yes, if the new position is put to some application purpose, such as repositioning an object or specifying the endpoint of a line. No, if the repositioning is just one of a sequence of repositionings as the user moves the cursor to place it on top of a menu item: Here, it is the menu choice that is the unit of information.

Basic interaction tasks (BITs) are indivisible; that is, if they were decomposed into smaller units of information, the smaller units would not in themselves be meaningful to the application. BITs are discussed in this section. In Section 8.3, we treat composite interaction tasks (CITs), which are aggregates of the basic interaction tasks described here. If one thinks of BITs as atoms, then CITs are molecules.

A complete set of BITs for interactive graphics is positioning, selecting, entering text, and entering numeric quantities. Each BIT is described in this section, and some of the many interaction techniques for each are discussed. However, there are far too many interaction techniques for us to give an exhaustive list, and we cannot anticipate the development of new techniques. Where possible, the pros and cons of each technique are discussed; remember that a specific interaction technique may be good in some situations and poor in others.

8.2.1 The Position Interaction Task

The positioning task involves specifying an (x, y) or (x, y, z) position to the application program. The customary interaction techniques for carrying out this task involve either moving a screen cursor to the desired location and then pushing a button, or typing the desired position's coordinates on either a real or a simulated keyboard. The positioning device can be direct or indirect, continuous or discrete, absolute or relative. In addition, cursor-movement commands can be typed explicitly on a keyboard, as Up, Left, and so on, or the same commands can be spoken to a voice-recognition unit. Furthermore, techniques can be used together—a mouse controlling a cursor can be used for approximate positioning, and arrow keys can be used to move the cursor a single screen unit at a time for precise positioning.

There are two types of positioning tasks, spatial and linguistic. In a **spatial** positioning task, the user knows where the intended position is, in spatial relation to nearby elements, as in drawing a line between two rectangles or centering an object between two others. In a **linguistic** positioning task, the user knows the

numeric values of the (x, y) coordinates of the position. In the former case, the user wants feedback showing the actual position on the screen; in the latter case, the coordinates of the position are needed. If the wrong form of feedback is provided, the user must mentally convert from one form to the other. Both forms of feedback can be provided by displaying both the cursor and its numeric coordinates, as in Fig. 8.3.

8.2.2 The Select Interaction Task—Variable-Sized Set of Choices

The selection task is that of choosing an element from a **choice set**. Typical choice sets are commands, attribute values, object classes, and object instances. For example, the line-style menu in a typical paint program is a set of attribute values, and the object-type (line, circle, rectangle, text, etc.) menu in such programs is a set of object classes. Some interaction techniques can be used to select from any of these four types of choice sets; others are less general. For example, pointing at a visual representation of a set element can serve to select it, no matter what the set type. On the other hand, although function keys often work quite well for selecting from a command, object class, or attribute set, it is difficult to assign a separate function key to each object instance in a drawing, since the size of the choice set is variable, often is large (larger than the number of available function keys), and changes quite rapidly as the user creates and deletes objects.

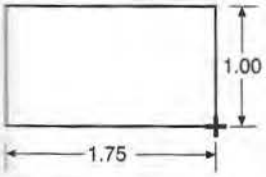


Figure 8.3
Numeric feedback regarding size of an object being constructed. The height and width are changed as the cursor (+) is moved, so the user can adjust the object to the desired size.

We use the terms (*relatively*) *fixed-sized choice set* and *varying-sized choice set*. The first term characterizes command, attribute, and object-class choice sets; the second, object-instance choice sets. The modifier *relatively* recognizes that any of these sets can change as new commands, attributes, or object classes (such as symbols in a drafting system) are defined. But the set size does not change frequently, and usually does not change much. Varying-sized choice sets, on the other hand, can become quite large, and can change frequently.

In this section, we discuss techniques that are particularly well suited to potentially large varying-sized choice sets; these include naming and pointing. In Section 8.2.3, we discuss selection techniques particularly well suited to (relatively) fixed-sized choice sets. These sets tend to be small, except for the large (but relatively fixed-sized) command sets found in complex applications. The techniques discussed include typing or speaking the name, abbreviation, or other code that represents the set element; pressing a function key associated with the set element (this can be seen as identical to typing a single character on the keyboard); pointing at a visual representation (textual or graphical) of the set element in a menu; cycling through the set until the desired element is displayed; and making a distinctive motion with a continuous positioning device.

Selecting objects by naming. The user can type the choice's name. The idea is simple, but what if the user does not know the object's name, as could easily happen if hundreds of objects are being displayed, or if the user has no reason to know names? Nevertheless, this technique is useful in several situations. First, if the user is likely to know the names of various objects, as a fleet commander would know

the names of the fleet's ships, then referring to them by name is reasonable, and can be faster than pointing, especially if the user might need to scroll through the display to bring the desired object into view. Second, if the display is so cluttered that picking by pointing is difficult *and* if zooming is not feasible (perhaps because the graphics hardware does not support zooming and software zoom is too slow), then naming may be a choice of last resort. If clutter is a problem, then a command to turn object names on and off would be useful.

Typing allows us to make multiple selections through wild-card or don't-care characters, if the choice set elements are named in a meaningful way. Selection by naming is most appropriate for experienced, regular users, rather than for casual, infrequent users.

If naming by typing is necessary, a useful form of feedback is to display, immediately after each keystroke, the list (or partial list, if the full list is too long) of names in the selection set matching the sequence of characters typed so far. This display can trigger memory of how the name is spelled, if the user has recalled the first few characters. As soon as an unambiguous match has been typed, the correct name can be automatically highlighted on the list. Alternatively, the name can be automatically completed as soon as an unambiguous match has been typed. This technique, called **autocompletion**, is sometimes disconcerting to new users, so caution is advisable. A separate strategy for name typein is spelling correction (sometimes called **Do What I Mean**, or DWIM). If the typed name does not match one known to the system, other names that are close to the typed name can be presented to the user as alternatives. Determining closeness can be as simple as searching for single-character errors, or can include multiple-character and missing-character errors.

With a voice recognizer, the user can speak, rather than type, a name, abbreviation, or code. Voice input is a simple way to distinguish commands from data. Commands are entered by voice, the data are entered by keyboard or other means. In a keyboard environment, this feature eliminates the need for special characters or modes to distinguish data and commands.

Selecting objects by pointing. Any of the pointing techniques mentioned in the introduction to Section 8.2 can be used to select an object, by first pointing and then indicating (typically via a button-push) that the desired object is being pointed at. But what if the object has multiple levels of hierarchy, as did the robot of Chapter 7? If the cursor is over the robot's hand, it is not clear whether the user is pointing at the hand, the arm, or the entire robot. Commands like Select_robot and Select_arm can be used to specify the level of hierarchy. On the other hand, if the level at which the user works changes infrequently, the user will be able to work faster with a separate command, such as Set_selection_level, used to change the level of hierarchy.

A different approach is needed if the number of hierarchical levels is unknown to the system designer and is potentially large (as in a drafting system, where symbols are made up of graphics primitives and other symbols). At least two user commands are required: Up_hierarchy and Down_hierarchy. When the user selects something, the system highlights the lowest-level object seen. If this is what

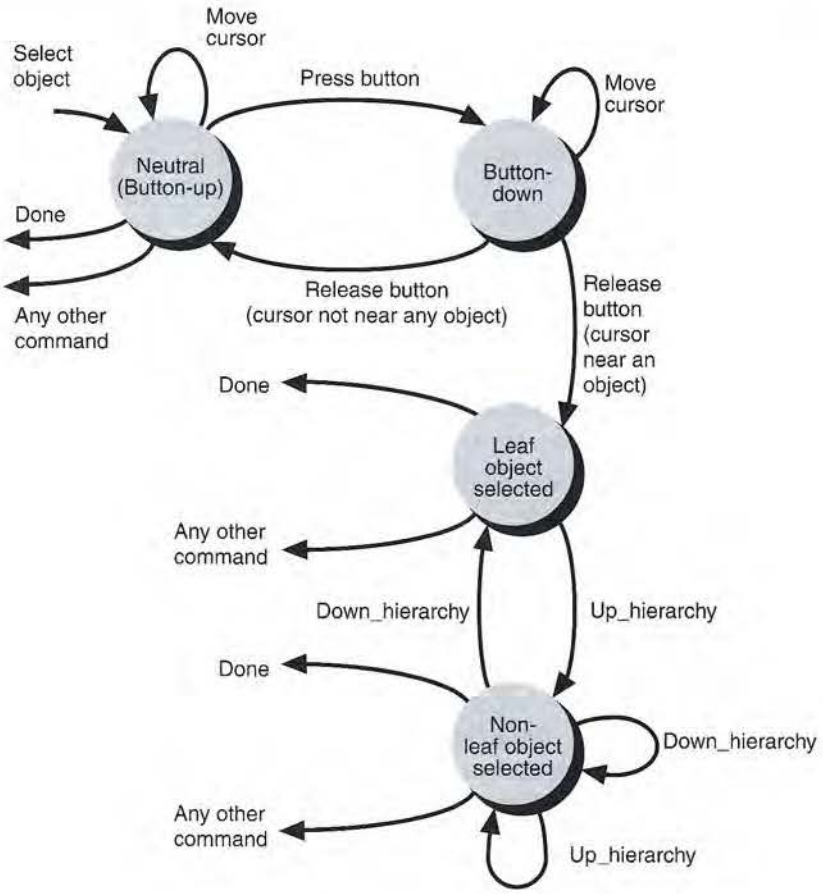


Figure 8.4 State diagram for an object-selection technique for an arbitrary number of hierarchy levels. Up and Down are commands for moving up and down the hierarchy. In the state "Leaf object selected," the Down_hierarchy command is not available. The user selects an object by pointing at it with a cursor, and pressing and then releasing a button.

desired, the user can proceed. If not, the user issues the first command: Up_hierarchy. The entire first-level object of which the detected object is a part is highlighted. If this is not what is wanted, the user travels up again and still more of the picture is highlighted. If the user travels too far up the hierarchy, direction is reversed with the Down_hierarchy command. In addition, a Return_to_lowest_level command can be useful in deep hierarchies, as can a hierarchy diagram in another window, showing where in the hierarchy the current selection is located. The state diagram of Fig. 8.4 shows one approach to hierarchical selection. Alternatively, a single command, say Move_up_hierarchy, can skip back to the originally selected leaf node after the root node is reached.

8.2.3 The Select Interaction Task—Relatively Fixed-Sized Choice Set

Menu selection is one of the richest techniques for selecting from a relatively fixed-sized choice set. Here we discuss several key factors in menu design.

Single-level versus hierarchical design. One of the most fundamental menu design decisions arises if the choice set is too large to display all at once. Such a menu can be subdivided into a logically structured hierarchy or presented as a linear sequence of choices to be paged or scrolled through. A scroll bar of the type used in many window managers allows all the relevant scrolling and paging commands to be presented in a concise way. A fast keyboard-oriented alternative to pointing at the scrolling commands can also be provided; for instance, the arrow keys can be used to scroll the window, and the shift key can be combined with the arrow keys to move the selection within the visible window, as shown in Fig. 8.5.

With a hierarchical menu, the user first selects from the choice set at the top of the hierarchy, which causes a second choice set to be available. The process is repeated until a leaf node (i.e., an element of the choice set itself) of the hierarchy tree is selected. As with hierarchical object selection, navigation mechanisms need to be provided so that the user can go back up the hierarchy if an incorrect subtree was selected. Visual feedback to give the user some sense of place within the hierarchy is also needed.

Menu hierarchies can be presented in several ways. Of course, successive levels of the hierarchy can replace one another on the display as further choices are made, but this does not give the user much sense of position within the hierarchy. The **cascading hierarchy**, as depicted in Fig. 8.6, is more attractive. Enough of each menu must be revealed that the complete highlighted selection path is visible, and some means must be used to indicate whether a menu item is a leaf node or is the name of a lower-level menu (in the figure, the right-pointing arrow fills this role). Another arrangement is to show just the name of each selection made thus

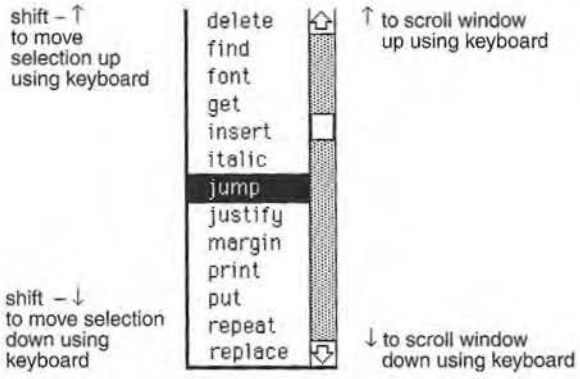


Figure 8.5 A menu within a scrolling window. The user controls scrolling by selecting the up and down arrows or by dragging the square in the scroll bar.

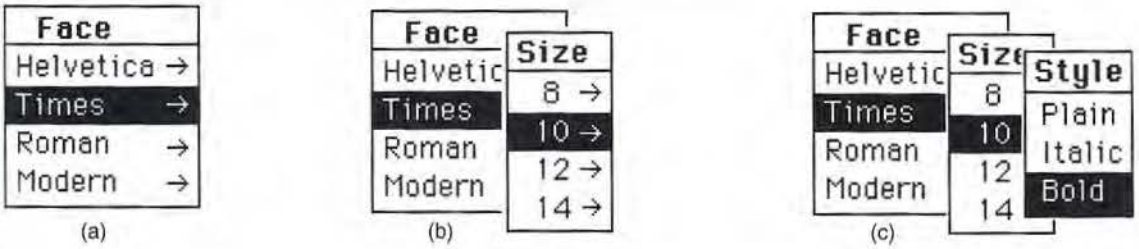


Figure 8.6 A pop-up hierarchical menu. (a) The first menu appears where the cursor is, in response to a button-down action. The cursor can be moved up and down to select the desired typeface. (b) The cursor is then moved to the right to bring up the second menu. (c) The process is repeated for the third menu.

far in traversing down the hierarchy, plus all the selections available at the current level.

When we design a hierarchical menu, the issue of depth versus breadth is always present. Snowberry et al. [SNOW83] found experimentally that selection time and accuracy improve when broader menus with fewer levels of selection are used. Similar results are reported by Landauer and Nachbar [LAND85] and by other researchers. However, these results do not necessarily generalize to menu hierarchies that lack a natural, understandable structure.

Hierarchical menu selection almost demands an accompanying keyboard or function-key accelerator technique to speed up selection for more experienced (so-called **power**) users. This is easy if each node of the tree has a unique name, so that the user can enter the name directly, and the menu system provides a backup should the user's memory fail. If the names are unique only within each level of the hierarchy, the power user must type the complete path name to the desired leaf node.

Menu placement. Menus shown on the display screen can be static and permanently visible, or can appear dynamically on request (tear-off, appearing, pop-up, pull-down, and pull-out menus).

A pop-up menu appears on the screen when a selection is to be made, either in response to an explicit user action (typically pressing a mouse or tablet puck button), or automatically because the next dialogue step requires a menu selection. The menu normally appears at the cursor location, which is usually the user's center of visual attention, thereby maintaining visual continuity. An attractive feature in pop-up menus is the initial highlighting of the most recently made selection from the choice set *if* the most recently selected item is more likely to be selected a second time than is another item, positioning the menu so the cursor is on that item.

Pop-up and other appearing menus conserve precious screen space—one of the user-interface designer's most valuable commodities. Their use is facilitated by a fast RasterOp instruction, as discussed in Chapter 2.

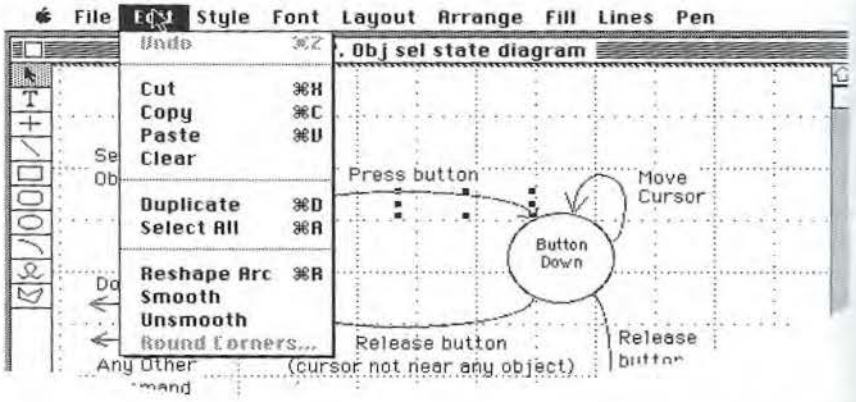


Figure 8.7 A Macintosh pull-down menu. The last menu item is gray rather than black, indicating that it is currently not available for selection (the currently selected object, an arc, does not have corners to be rounded). The Undo command is also gray, because the previously executed command cannot be undone. Abbreviations are accelerator keys for power users. (Copyright 1988 Claris Corporation. All rights reserved.)

Figure 8.8 Radio-button technique for selecting from a set of mutually exclusive alternatives. (Courtesy of NeXT, Inc. © 1989 NeXT, Inc.)

Unlike pop-up menus, pull-down menus are anchored in a menu bar along the top of the screen. All the popular graphical user interfaces—the Apple Macintosh, Microsoft Windows, OPEN LOOK, and Motif—use pull-down menus. Macintosh menus, shown in Fig. 8.7, also illustrate accelerator keys and context sensitivity.

Current selection. If a system has the concept of *currently selected element* of a choice set, menu selection allows this element to be highlighted. In some cases, an initial default setting is provided by the system and is used unless the user changes it. The currently selected element can be shown in various ways. The **radio-button** interaction technique, patterned after the tuning buttons on car radios, is one way (Fig. 8.8). Again, some pop-up menus highlight the most recently selected item and place it under the cursor, on the assumption that the user is more likely to reselect that item than to select any other entry.

Size and shape of menu items. Pointing accuracy and speed are affected by the size of each individual menu item. Larger items are faster to select, as predicted by Fitts' law [FITT54; CARD83]; on the other hand, smaller items take less space and permit more menu items to be displayed in a fixed area, but induce more errors during selection. Thus, there is a conflict between using small menu items to preserve screen space versus using larger ones to decrease selection time and to reduce errors.

Pattern recognition. In selection techniques involving pattern recognition, the user makes sequences of movements with a continuous-positioning device, such as a tablet or mouse. The pattern recognizer automatically compares the sequence with a set of defined patterns, each of which corresponds to an element of the

selection set. Proofreader's marks indicating delete, capitalize, move, and so on are attractive candidates for this approach [WOLF87].

Recent advances in character recognition algorithms have led to pen-based operating systems and notepad computers, such as Apple's Newton. Patterns are entered on a tablet, and are recognized and interpreted as commands, numbers, and letters.

Function keys. Elements of the choice set can be associated with function keys. (We can think of single-keystroke inputs from a regular keyboard as function keys.) Unfortunately, there never seem to be enough keys to go around! The keys can be used in a hierarchical-selection fashion, and their meanings can be altered using chords, say by depressing the keyboard shift and control keys along with the function key itself. For instance, Microsoft Word on the Macintosh uses "shift-option->" to increase point size and the symmetrical "shift-option-<" to decrease point size; "shift-option-I" italicizes plain text and unitalicizes italicized text, whereas "shift-option-U" treats underlined text similarly.

8.2.4 The Text Interaction Task

The text-string input task entails entering a character string to which the application does not ascribe any special meaning. Thus, typing a command name is *not* a text-entry task. In contrast, typing legends for a graph and typing text into a word processor *are* text input tasks. Clearly, the most common text-input technique is use of the QWERTY keyboard.

8.2.5 The Quantify Interaction Task

The quantify interaction task involves specifying a numeric value between some minimum and maximum value. Typical interaction techniques are typing the value, setting a dial to the value, and using an up–down counter to select the value. Like the positioning task, this task may be either linguistic or spatial. When it is linguistic, the user knows the specific value to be entered; when it is spatial, the user seeks to increase or decrease a value by a certain amount, with perhaps an approximate idea of the desired end value. In the former case, the interaction technique clearly must involve numeric feedback of the value being selected (one way to do this is to have the user type the actual value); in the latter case, it is more important to give a general impression of the approximate setting of the value. This is typically accomplished with a spatially oriented feedback technique, such as display of a dial or gauge on which the current (and perhaps previous) value is shown.

One means of entering values is the potentiometer. The decision of whether to use a rotary or linear potentiometer should take into account whether the visual feedback of changing a value is rotary (e.g., a turning clock hand) or linear (e.g., a rising temperature gauge). The current position of one or a group of slide potentiometers is much more easily comprehended at a glance than are those of rotary potentiometers, even if the knobs have pointers. On the other hand, rotary potentiometers are easier to adjust. Availability of both linear and rotary potentiometers

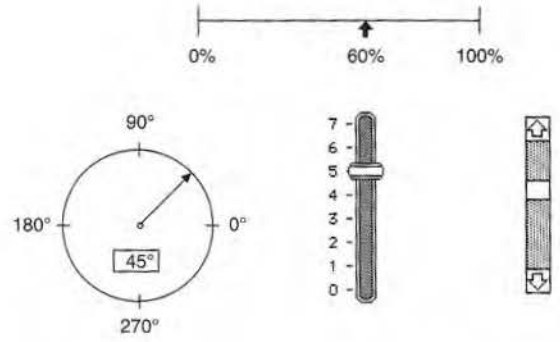


Figure 8.9 Several dials that the user can employ to input values by dragging the control pointer. Feedback is given by the pointer and, in two cases, by numeric displays. (Vertical sliders © Apple Computer, Inc.)

can help users to associate meanings with each device. It is important to use dir tions consistently: Clockwise or upward movements normally increase a value.

With continuous-scale manipulation, the user points at the current-value in cator on a displayed gauge or scale, presses the selection button, drags the indi tor along the scale to the desired value, and then releases the selection button. pointer is typically used to indicate the value selected on the scale, and a nume echo may be given. Figure 8.9 shows several such dials and their associated fe back.

8.2.6 3D Interaction Tasks

Two of the four interaction tasks described previously for 2D applications beco more complicated in 3D: position and select. The first part of this section de with a technique for positioning and selecting, which are closely related. In t section, we also introduce an additional 3D interaction task: rotate (in the sens orienting an object in 3-space). The major reason for the complication is the di culty of perceiving 3D depth relationships of a cursor or object relative to ot displayed objects. This contrasts starkly with 2D interaction, where the user readily perceive that the cursor is above, next to, or on an object. A second complication arises because the commonly available interaction devices, such mice and tablets, are only 2D devices, and we need a way to map movement these 2D devices into 3D.

Display of stereo pairs, corresponding to left- and right-eye views, is hel for understanding general depth relationships, but is of limited accuracy as a cise locating method. Methods for presenting stereo pairs to the eye are discu in Chapter 12, and in [HODG85]. Other ways to show depth relationships are cussed in Chapters 12–14.

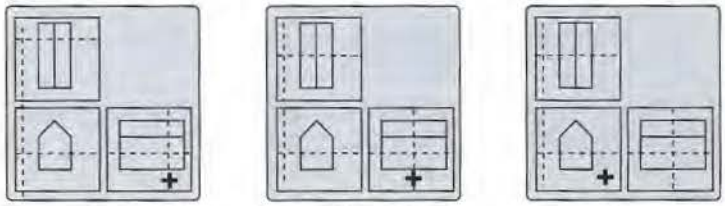


Figure 8.10 3D positioning technique using three views of the same scene (a house). The 2D cursor (+) is used to select one of the dashed 3D cursor lines.

Figure 8.10 shows a common way to position in 3D. The 2D cursor, under control of, say, a mouse, moves freely among the three views. The user can select any one of the 3D cursor's dashed lines and can drag the line using a button-down–drag–button-up sequence. If the button-down event is close to the intersection of two dashed cursor lines, then both are selected and are moved with the mouse. Although this method may appear restrictive in forcing the user to work in one or two dimensions at a time, it is sometimes advantageous to decompose the 3D manipulation task into simpler lower-dimensional tasks. Selecting as well as locating is facilitated with multiple views: Objects that overlap and hence are difficult to distinguish in one view may not overlap in another view.

As with locating and selecting, the issues in 3D rotation are understanding depth relationships, mapping 2D interaction devices into 3D, and ensuring stimulus-response compatibility (S-R compatibility)[1]. An easily implemented 3D rotation technique provides slider dials or gauges that control rotation about three axes. S-R compatibility suggests that the three axes normally should be in the screen-coordinate system—x to the right, y increasing upward, z out of (or into) the screen [BRIT78]. Of course, the center of rotation either must be explicitly specified as a separate step, or must be implicit (typically the screen-coordinate origin, the origin of the object, or the center of the object). Providing rotation about the sceen's x and y axes is especially simple, as suggested in Fig. 8.11(a). The (x, y, z) coordinate system associated with the sliders is rotated as the sliders are moved to show the effect of the rotation. The two-axis rotation approach can be easily generalized to three axes by adding a dial for z-axis rotation (a dial is preferable to a slider for S-R compatibility). Even more S-R compatibility comes from the arrangement of dials on the faces of a cube shown in Fig. 8.11(b), which clearly suggests the axes controlled by each dial. A 3D trackball could be used instead of the dials.

[1] The human-factors principle, which states that system responses to user actions must be in the same direction or same orientation, and that the magnitude of the responses should be proportional to the actions.

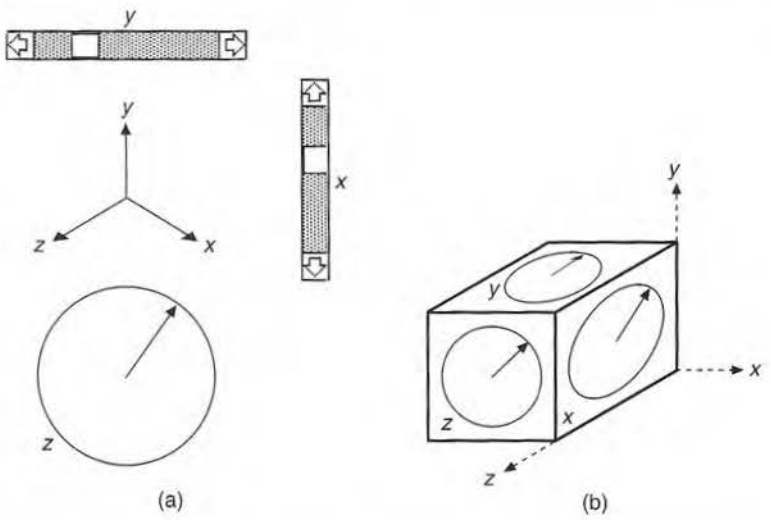


Figure 8.11 Two approaches to 3D rotation. (a) Two slider dials for effecting rotation about the screen's x and y axes, and a dial for rotation about the screen's z axis. The coordinate system represents world coordinates and shows how world coordinates relate to screen coordinates. (b) Three dials to control rotation about three axes. The placement of the dials on the cube provides strong stimulus-response compatibility.

It is often necessary to combine 3D interaction tasks. Thus, rotation requires a select task for the object to be rotated, a position task for the center of rotation, and an orient task for the actual rotation. Specifying a 3D view can be thought of as a combined positioning (where the eye is), orientation (how the eye is oriented), and scaling (field of view, or how much of the projection plane is mapped into the viewport) task. We can create such a task by combining some of the techniques we have discussed, or by designing a *fly-around* capability in which the viewer flies an imaginary airplane around a 3D world. The controls are typically pitch, roll, and yaw, plus velocity to speed up or slow down. With the fly-around concept, the user needs an overview, such as a 2D plan view, indicating the imaginary airplane's ground position and heading.

8.3 COMPOSITE INTERACTION TASKS

Composite interaction tasks (CITs) are built on top of the basic interaction tasks (BITs) described in the previous section, and are actually combinations of BITs integrated into a unit. There are three major forms of CITs: dialogue boxes, used to specify multiple units of information; construction, used to create objects requiring two or more positions; and manipulation, used to reshape existing geometric objects.

8.3.1 Dialogue Boxes

We often need to select multiple elements of a selection set. For instance, text attributes, such as italic, bold, underline, hollow, and all caps, are not mutually exclusive, and the user may want to select two or more at once. In addition, there may be several sets of relevant attributes, such as typeface and font. Some of the menu approaches useful in selecting a single element of a selection set are not satisfactory for multiple selections. For example, pull-down and pop-up menus normally disappear when a selection is made, necessitating a second activation to make a second selection.

This problem can be overcome with dialogue boxes, a form of menu that remains visible until explicitly dismissed by the user. In addition, dialogue boxes can permit selection from more than one selection set, and can also include areas for entering text and values. Selections made in a dialogue box can be corrected immediately. When all the information has been entered into the dialogue box, the box is typically dismissed explicitly with a command. Attributes and other values specified in a dialogue box can be applied immediately, allowing the user to preview the effect of a font or line-style change.

8.3.2 Construction Techniques

One way to construct a line is to have the user indicate one endpoint and then the other; once the second endpoint is specified, a line is drawn between the two points. With this technique, however, the user has no easy way to try out different line positions before settling on a final one, because the line is not actually drawn until the second endpoint is given. With this style of interaction, the user must invoke a command each time an endpoint is to be repositioned.

A far superior approach is **rubberbanding**, discussed in Chapter 2. When the user pushes a button (often the tipswitch on a tablet stylus, or a mouse button), the starting position of the line is established by the cursor (usually but not necessarily controlled by a continuous-positioning device). As the cursor moves, so does the endpoint of the line; when the button is released, the endpoint is frozen. Figure 8.12 shows a rubberband line-drawing sequence. The *rubberband* state is active *only* while a button is held down. It is in this state that cursor movements cause the current line to change.

An entire genre of interaction techniques is derived from rubberband line drawing. The **rubber-rectangle** technique starts by anchoring one corner of a rectangle with a button-down action, after which the opposite corner is dynamically linked to the cursor until a button-up action occurs. The state diagram for this technique differs from that for rubberband line drawing only in the dynamic feedback of a rectangle rather than a line. The **rubber-circle** technique creates a circle that is centered at the initial cursor position and that passes through the current cursor position, or that is within the square defined by opposite corners. All these techniques have in common the user-action sequence of button-down, move locator and see feedback, button-up.

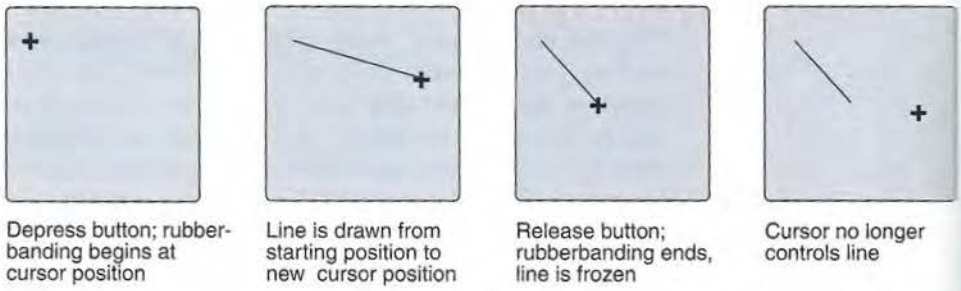


Figure 8.12 Rubberband line drawing.

Constraints of various types can be applied to the cursor positions in any of these techniques. For example, Fig. 8.13 shows a sequence of lines drawn using the same cursor positions as in Fig. 8.12, but with a horizontal constraint in effect. A vertical line, or a line at some other orientation, can also be drawn in this manner. Polylines made entirely of horizontal and vertical lines, as in printed circuit boards, VLSI chips, and some city maps, are readily created; right angles are introduced either in response to a user command, or automatically as the cursor changes direction. The idea can be generalized to any shape, such as a circle, ellipse, or any other curve; the curve is initialized at some position, then cursor movements control how much of the curve is displayed. In general, the cursor position is used as input to a constraint function whose output is then used to display the appropriate portion of the object.

8.3.3 Dynamic Manipulation

It is not sufficient to just create lines, rectangles, and so on. In many situations, the user must be able to modify previously created geometric entities.

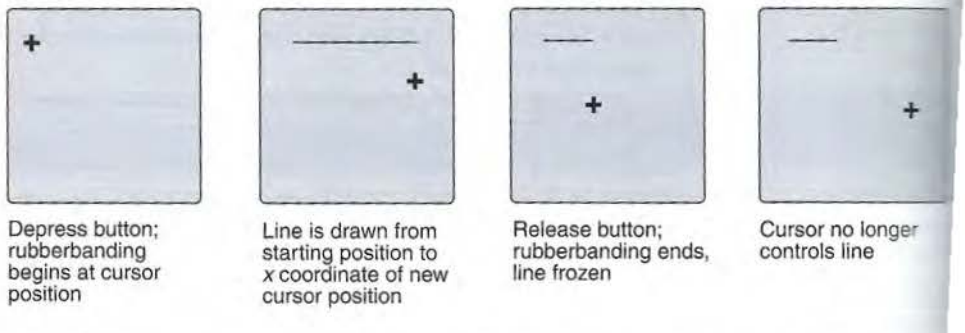


Figure 8.13 Horizontally constrained rubberband line drawing.

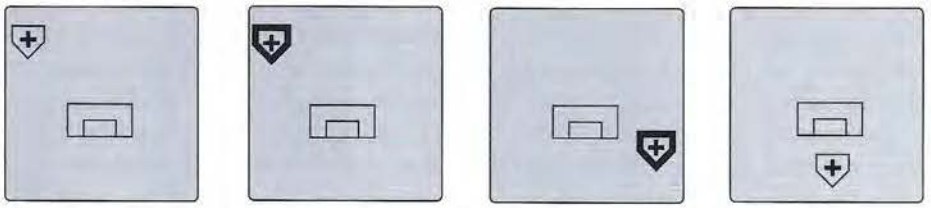


Figure 8.14 Dragging a symbol into a new position.

Dragging moves a selected symbol from one position to another under control of a cursor, as in Fig. 8.14. A button-down action typically starts the dragging (in some cases, the button-down is also used to select the symbol under the cursor to be dragged); then, a button-up freezes the symbol in place, so that further movements of the cursor have no effect on it. This button-down–drag–button-up sequence is often called **click-and-drag** interaction.

The concept of **handles** is useful to provide scaling of an object. Figure 8.15 shows an object with eight handles, which are displayed as small squares at the corners and on the sides of the imaginary box surrounding the object. The user selects one of the handles and drags it to scale the object. If the handle is on a corner, then the corner diagonally opposite is locked in place. If the handle is in the middle of a side, then the opposite side is locked in place.

When this technique is integrated into a complete user interface, the handles appear only when the object is selected to be operated on. Handles are also a unique visual code to indicate that an object is selected, since other visual codings (e.g., line thickness, dashed lines, or changed intensity) might also be used as part of the drawing itself.

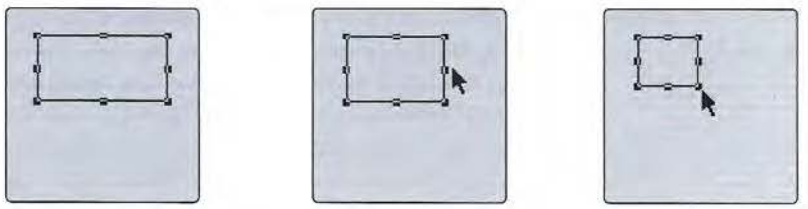


Figure 8.15 Handles used to reshape objects.

Figure 8.16 Handles used to reposition the vertices of a polygon.

Dragging, rotating, and scaling affect an entire object. What if we wish to be able to move individual points, such as the vertices of a polygon? Vertices could be named, and the user could enter the name of a vertex and its new (x, y) coordinates. But the same point-and-drag strategy used to move an entire object is more attractive. In this case, the user points to a vertex, selects it, and drags it to a new position. The vertices adjacent to the one selected remain connected via rubberband lines. To facilitate selecting a vertex, we can make a vertex blink whenever the cursor is near, or we can superimpose handles over each vertex, as in Fig. 8.16. Similarly, the user can move an edge of a polygon by selecting it and dragging, with the edge maintaining its original slope. For smooth curves and surfaces, handles can also be provided to allow the user to manipulate points that control the shape, as discussed further in Chapter 9.

8.4 INTERACTION-TECHNIQUE TOOLKITS

The look and feel of a user–computer interface is determined largely by the collection of interaction techniques provided for it. Recall that interaction techniques implement the hardware binding portion of a user–computer interface design. Designing and implementing a good set of interaction techniques is time consuming: Interaction-technique toolkits, which are subroutine libraries of interaction techniques, are mechanisms for making a collection of techniques available for use by application programmers. This approach, which helps to ensure a consistent look and feel among application programs, is clearly a sound software-engineering practice.

Interaction-technique toolkits can be used not only by application programs but also by the resident window manager, which is after all just another program. Using the same toolkit across the board is an important and commonly used approach to providing a look and feel that unifies both multiple applications and

the windowing environment itself. For instance, the menu style used to select window operations should be the same style used within applications.

A toolkit can be implemented on top of a **window-management system** [FOLE90]. In the absence of a window system, toolkits can be implemented directly on top of a graphics subroutine package; however, because elements of a toolkit include menus, dialogue boxes, scroll bars, and the like, all of which can conveniently be implemented in windows, the window system substrate is normally used. Widely used toolkits include the Macintosh toolkit [APPL85], OSF/Motif [OPEN89] and InterViews [LINT89] for use with the X Window System, and several toolkits that implement OPEN LOOK [SUN89]. Color Plate 9 shows the OSF/Motif interface. Color Plate 10 shows the OPEN LOOK interface.

SUMMARY

We have presented some of the most important concepts of user interfaces: input devices, interaction techniques, and interaction tasks. There are many more aspects of user interface techniques and design, however, that we have not discussed. Among these are the pros and cons of various dialogue styles—such as what you see is what you get (WYSIWYG), command language, and direct manipulation—and window-manager issues that affect the user interface. [FOLE90] has a thorough treatment of those topics.

Exercises

8.1 Examine a user–computer interface with which you are familiar. List each interaction task used. Categorize each task into one of the four BITs of Section 8.2. If an interaction does not fit this classification scheme, try decomposing it further.

8.2 Extend the state diagram of Fig. 8.4 to include a "return to lowest level" command that takes the selection back to the lowest level of the hierarchy, such that whatever was selected first is selected again.

8.3 Implement a menu package on a color raster display that has a look-up table such that the menu is displayed in a strong, bright but partially transparent color, and all the colors underneath the menu are changed to a subdued gray.

8.4 Implement any of the 3D interaction techniques discussed in this chapter.

8.5 Draw the state diagram that controls pop-up hierarchical menus. Draw the state diagram that controls panel hierarchical menus.