We summarize the four SPHIGS rendering modes here; they are discussed much more fully in Chapters 14 through 16.

Wireframe rendering mode. WIREFRAME mode is the fastest but least realistic form of display. Objects are drawn as though made of wire, with only their edges showing. The visible (within the view volume) portions of all edges of all objects are shown in their entirety, with no hidden-edge removal. Primitives are drawn in temporal order—that is, in the order in which the traverser encounters them in the posted structure networks in the database; this order is affected by the display-priority determined by the view index, as mentioned in Section 7.3.4.

All edge attributes affect screen appearance in their designated way in this mode; in fact, when the edge flag is set to EDGE_INVISIBLE, fill areas and polyhedra are entirely invisible in this mode.

Shaded rendering modes. In its other three rendering modes, SPHIGS displays fill areas and polyhedra in a more realistic fashion by drawing fill areas and facets as filled polygons. The addition of shaded areas to the rendering process increases the complexity significantly, because spatial ordering becomes important—portions of objects that are hidden (because they are obscured by portions of "closer" objects) must not be displayed. Methods for determining visible surfaces (also known as hidden-surface removal) are discussed in Chapter 15.

For the three shaded rendering modes, SPHIGS "shades" the interior pixels of visible portions of the facets; the quality of the rendering varies with the mode. For FLAT shading, the mode used often in this chapter's figures, all facets of a polyhedron are rendered in the current interior color, without being influenced by any light sources in the scene. Visible portions of edges are shown (if the edge flag is EDGE_VISIBLE) as they would appear in WIREFRAME mode. If the interior color is set to match the screen background, only the edges show—this use of FLAT rendering, which produced Figs. 7.9(a) and 7.14(c), simulates wireframe with hidden-edge removal.

The two highest-quality rendering modes produce images illuminated by a light source;[10] illumination and shading models are discussed in Chapter 16. These images are nonuniformly "shaded;" the colors of the pixels are based on, but are not exactly, the value of the interior-color attribute. In LIT_FLAT mode, all the pixels on a particular facet have the same color, determined by the angle at which the light hits the facet. Because each facet is of a uniform color, the image has a "faceted" look, and the contrast between adjacent faces at their shared edge is noticeable. GOURAUD mode colors the pixels to provide a smooth shaded appearance that eliminates the faceted look.

In FLAT mode, the edge-flag attribute should be set to EDGE_VISIBLE, because, without visible edges, the viewer can determine only the silhouette boundary of the object. In the two highest-quality modes, however, edge visibility is usually turned off, since the shading helps the user to determine the shape of the object.

[10] The PHIGS+ extension provides many facilities for controlling rendering, including specification of the placement and colors of multiple light sources, of the material properties of objects characterizing their interaction with light, and so on; see Chapters 14 through 16.

7.9 STRUCTURE NETWORK EDITING FOR DYNAMIC EFFECTS

As with any database, we must be able not only to create and query (in order to display) the SPHIGS structure database, but also to edit it in a convenient way. An application edits a structure via the procedures described in this section; if the application also maintains an application model, it must ensure that the two representations are edited in tandem. *Motion dynamics* requires modification of viewing or modeling transformations; *update dynamics* requires changes in or replacement of structures. The programmer may choose to edit a structure's internal element list if the changes are relatively minor; otherwise, for major editing, it is common to delete and then to respecify the structure in its entirety.

In the remainder of this section, we present methods for intrastructure editing; see the SPHIGS reference manual for information on editing operations that affect entire structures (e.g., deletion, emptying), and for more detailed descriptions of the procedures presented here.

7.9.1 Accessing Elements with Indices and Labels

The rudimentary editing facilities of both SPHIGS and PHIGS resemble those of old-fashioned line-oriented program editors that use line numbers. The elements in a structure are indexed from 1 to N; whenever an element is inserted or deleted, the index associated with each higher-indexed element in the same structure is incremented or decremented. The unique *current element* is that element whose index is stored in the *element-pointer* state variable. When a structure is opened with the SPH_openStructure call, the element pointer is set to N (pointing to the last element) or to 0 for an empty structure. The pointer is incremented when a new element is inserted after the current element, and is decremented when the current element is deleted. The pointer may also be set explicitly by the programmer using absolute and relative positioning commands:

```
void SPH_setElementPointer (int index);
void SPH_offsetElementPointer (int delta);      /* + for forward, - for backward */
```

Because the index of an element changes when a preceding element is added or deleted in its parent structure, using element indices to position the element pointer is liable to error. Thus, SPHIGS allows an application to place "landmark" elements, called *labels*, within a structure. A label element is given an integer identifier when it is generated:

```
void SPH_label (int id);
```

The application can move the element pointer via

```
void SPH_moveElementPointerToLabel (int id);
```

The pointer is then moved forward in search of the specified label. If the end of the structure is reached before the label is found, the search terminates unsuccessfully. Thus, it is advisable to move the pointer to the very front of the structure (index 0) before searching for a label.

7.9.2 Intrastructure Editing Operations

The most common editing action is insertion of new elements into a structure. Whenever an element-generating procedure is called, the new element is placed immediately after the current element, and the element pointer is incremented to point to the new element.[11]

Another form of insertion entails copying all the elements of a given structure into the open structure (immediately after the current element):

void SPH_copyStructure (**int** *structureID*);

Elements are deleted by the following procedures:

void SPH_deleteElement (**void**);
void SPH_deleteElementsInRange (**int** *firstIndex*, **int** *secondIndex*);
void SPH_deleteElementsBetweenLabels (**int** *firstLabel*, **int** *secondLabel*);

In all cases, after the deletion is made, the element pointer is moved to the element immediately preceding the ones deleted, and all survivors are renumbered. The first procedure deletes the current element. The second procedure deletes the elements lying between and including the two specified elements. The third procedure is similar, but does not delete the two label elements.

Note that these editing facilities all affect an entire element or a set of elements; there are no provisions for selective editing of data fields within an element. Thus, for example, when a single vertex needs to be updated the programmer must respecify the entire polyhedron.

An editing example. Let us look at a modification of our simple street example. Our street now consists of only the first house and the cottage, the former being fixed and the latter being movable. We create a label in front of the cottage, so we can subsequently edit the transformation in order to move the cottage.

To move the cottage, we reopen the street structure, move the pointer to the label, and then offset to the transformation element, replace the transformation element, and close the structure. The screen is automatically updated after the structure is closed, to show the cottage in its new position. This code is shown in Fig. 7.22(a), and its sequence of operations is illustrated in (b).

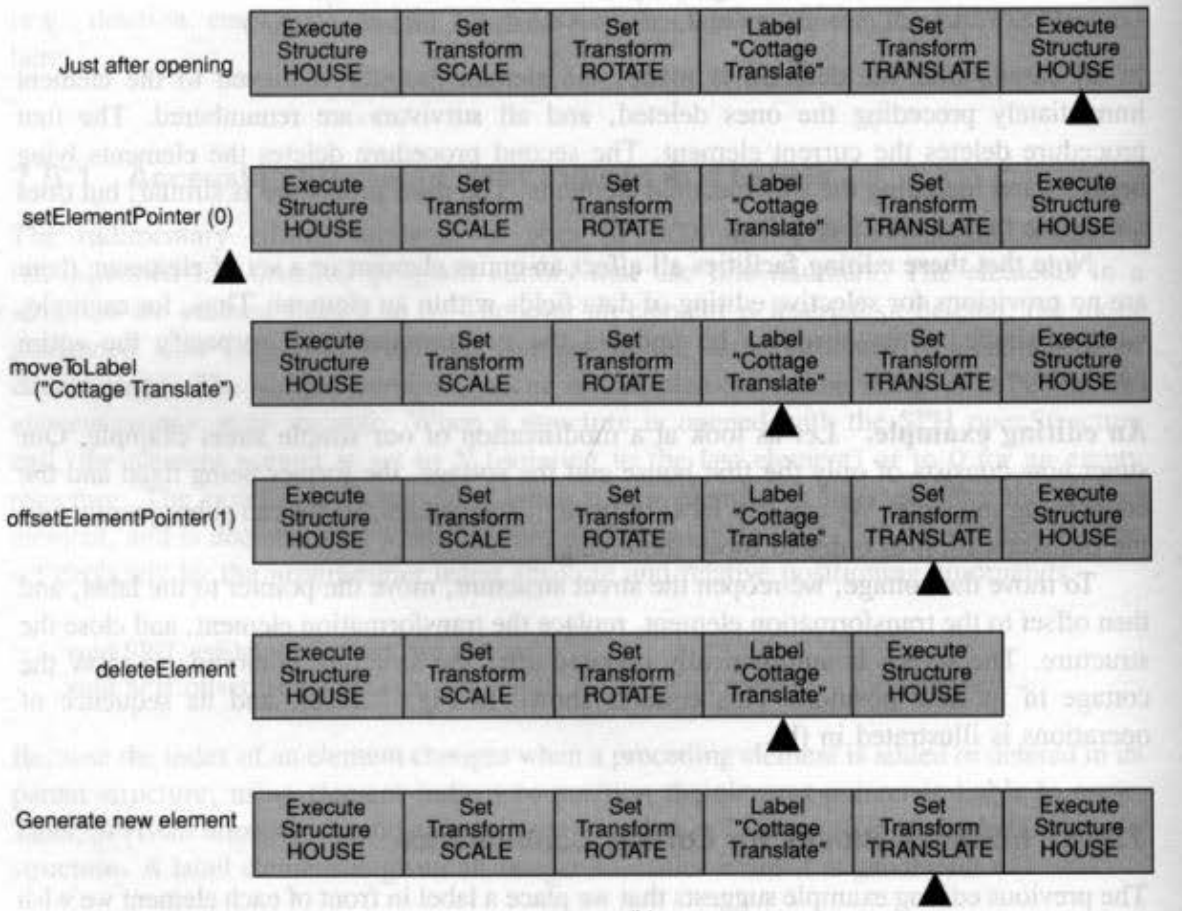
7.9.3 Instance Blocks for Editing Convenience

The previous editing example suggests that we place a label in front of each element we wish to edit, but creating so many labels is clearly too laborious. There are several techniques for avoiding this tedium. The first is to bracket an editable group of elements with two labels, and to use the labels in deleting or replacing the entire group. Another common technique is to group the set of elements in a fixed format and to introduce the group with a single label.

[11] We show the use of insert "mode" in our editing examples; however, SPHIGS also supports a "replace" editing mode in which new elements write over extant ones. See the reference manual for details.

```
SPH_openStructure (STREET_STRUCT);
    /* When a structure is opened, the element pointer is initially at its very end. We */
    /* must first move the pointer to the beginning, so we can search for labels. */
    SPH_setElementPointer (0);
    SPH_moveElementPointerToLabel (COTTAGE_TRANSLATION_LABEL);
    SPH_offsetElementPointer (1);      /* Pointer now points at transform element. */
    SPH_deleteElement ();              /* We replace here via a delete/insert combination */
    SPH_setLocalTransformation (newCottageTranslationMatrix, PRECONCATENATE);
SPH_closeStructure ();
```

(a)



(b)

Fig. 7.22 Editing operations. (a) Code performing editing. (b) Snapshot sequence of structure during editing. The black triangle shows the element pointer's position. (Syntax of calls abbreviated for illustrative purposes.)

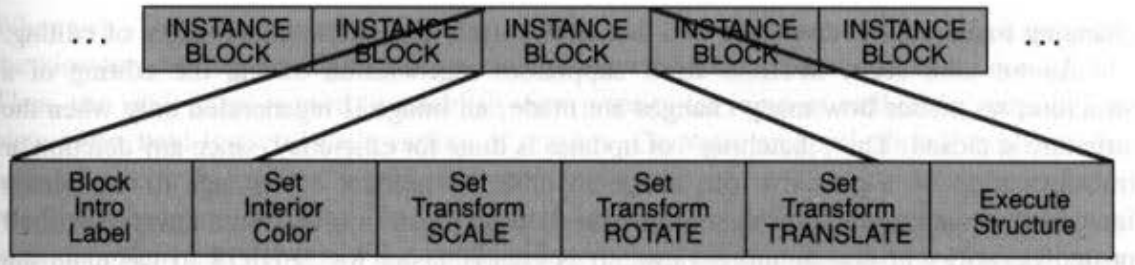


Fig. 7.23 Sample instance-block format.

To edit any member of the group, one moves the element pointer to the label, then offsets the pointer from that label into the group itself. Because the group's format is fixed, the offset is an easily determined small integer.

A special case of this technique is to design a standard way of instantiating substructures by preceding the structure-execution element with a common list of attribute-setting elements. A typical format of such a sequence of elements, called an *instance block,* is shown in Fig. 7.23; first comes the label uniquely identifying the entire block, then an interior-color setting, then the three basic transformations, and finally the invocation of the symbol structure.

We can create a set of symbolic constants to provide the offsets:

```
const int INTERIOR_COLOR_OFFSET = 1;
const int SCALE_OFFSET = 2;
const int ROTATION_OFFSET = 3;
const int TRANSLATION_OFFSET = 4;
```

Using the fixed format for the block guarantees that a particular attribute is modified in the same way for any instance. To change the rotation transformation of a particular instance, we use the following code:

```
SPH_openStructure (ID of structure to be edited);
    SPH_setElementPointer (0);
    SPH_moveElementPointerToLabel (the desired instance-block label);
    SPH_offsetElementPointer (ROTATION_OFFSET);
    SPH_deleteElement ();
    SPH_setLocalTransformation (newMatrix, mode);
SPH_closeStructure ();
```

Another nice feature of instance blocks is that the label introducing each block is easy to define: If the application keeps an internal database identifying all instances of objects, as is common, the label can be set to the unique number that the application itself uses to identify the instance internally.

7.9.4 Controlling Automatic Regeneration of the Screen Image

SPHIGS constantly updates the screen image to reflect the current status of its structure storage database and its view table. On occasion, however, we want to inhibit this regeneration, either to increase efficiency or to avoid presenting the user with a continuously

changing image that is confusing and that shows irrelevant intermediate stages of editing.

As we have seen, SPHIGS itself suppresses regeneration during the editing of a structure; no matter how many changes are made, an image is regenerated only when the structure is closed. This "batching" of updates is done for efficiency, since any deletion or transformation of a primitive can cause an arbitrary amount of damage to the screen image—damage that requires either selective damage repair or brute-force retraversal of all posted networks in one or more views. It is clearly faster for SPHIGS to calculate the cumulative effect of a number of consecutive edits just once, before regeneration.

A similar situation arises when several consecutive changes are made to different structures—for instance, when a structure and its substructures are deleted via consecutive calls to deleteStructure. To avoid this problem, an application can suppress automatic regeneration before making a series of changes, and allow it again afterward:

```
void SPH_setImplicitRegenerationMode (ALLOWED / SUPPRESSED value);
```

Even while implicit regeneration is suppressed, the application may explicitly demand a screen regeneration by calling

```
void SPH_regenerateScreen (void);
```

7.10 INTERACTION

Both SRGP's and SPHIGS's interaction modules are based on the PHIGS specification, and thus they have the same facilities for setting device modes and attributes, and for obtaining measures. The SPHIGS keyboard device is identical to that of SRGP, except that the echo origin is specified in NPC space with the z coordinate ignored. The SPHIGS locator device's measure has an additional field for the z coordinate, but is otherwise unchanged. SPHIGS also adds two new interaction facilities. The first is *pick correlation*, augmenting the locator functionality to provide identification of an object picked by the user. The second is the *choice* device, described in the reference manual, which supports menus. Section 10.1 provides a critical review of the PHIGS interaction devices in general.

7.10.1 Locator

The SPHIGS locator returns the cursor position in NPC coordinates, with $z_{NPC} = 0$.[12] It also returns the index of the highest-priority view whose viewport encloses the cursor.

```
typedef struct {
    point position;        /* [x, y, 0]NPC screen position */
    int viewIndex;         /* Index of view whose viewport encloses the cursor */
    int buttonOfMostRecentTransition;
    enum {UP, DOWN} buttonChord[MAX_BUTTON_COUNT];
} locatorMeasure;
```

[12] In PHIGS, the locator returns points in the 3D world-coordinate system. Many implementations, however, cannot return a meaningful z value; only high-performance workstations that support control dials and multiple real-time views can offer a comfortable user interface for pointing in 3D (see Chapter 8).

When two viewports overlap and the cursor position lies in the intersection of their bounds, the viewport having the highest index (in the view table) is returned in the second field. Thus, the view index is used to establish view priority for input as well as for output. The view-index field is useful for a variety of reasons. Consider an application that allows the user to specify bounds of a viewport interactively, much as one can move or resize a window manager's windows. In response to a prompt to resize, the user can pick any location within the viewport. The application program can then use the *viewIndex* field to determine which view was picked, rather than doing a point-in-rectangle test on viewport boundaries. The view index is also used in applications with some output-only views; such applications can examine the returned view index to determine whether the correlation procedure even needs to be called.

7.10.2 Pick Correlation

Because the SPHIGS programmer thinks in terms of modeled objects rather than of the pixels composing their images, it is useful for the application to be able to determine the identity of an object whose image a user has picked. The primary use of the locator, therefore, is to provide an NPC point for input to the pick-correlation procedure discussed in this section. As we saw with SRGP, pick correlation in a flat-earth world is a straightforward matter of detecting *hits*—primitives whose images lie close enough to the locator position to be considered chosen by the user. If there is more than one hit, due to overlapping primitives near the cursor, we disambiguate by choosing the one most recently drawn, since that is the one that lies "on top." Thus, a 2D pick correlator examines the primitives in inverse temporal order, and picks the first hit. Picking objects in a 3D, hierarchical world is a great deal more complex, for the reasons described next; fortunately, SPHIGS relieves an application of this task.

Picking in a hierarchy. Consider the complexity introduced by hierarchy. First, what information should be returned by the pick-correlation utility to identify the picked object? A structure ID is not enough, because it does not distinguish between multiple instances of a structure. Only the full *path*—a description of the complete ancestry from root to picked primitive—provides unique identification.

Second, when a particular primitive is picked, which level of the hierarchy did the user mean? For example, if the cursor is placed near one of our robot's thumbs, does the user mean to select the thumb, the arm, the upper body, or the entire robot? At times, the actual primitive is intended, at times the leaf structure is intended, and any other level is possibly intended, up to the very root! Some applications resolve this problem by providing a feedback mechanism allowing the user to step through the levels from primitive to root, in order to specify exactly which level is desired (see Exercise 7.13).

Comparison criterion. How is proximity to an object defined when the comparison should really be done in 3D? Since the locator device effectively yields a 2D NPC value, there is no basis for comparing the z coordinates of primitives to the locator position. Thus, SPHIGS can compare the cursor position only to the screen images of the primitives, not to the WC locations of the primitives. If a primitive is a hit, it is deemed a *candidate* for correlation. In wireframe mode, SPHIGS picks the very first candidate encountered during

traversal; the reason for this strategy is that there is no obvious depth information in a wireframe image, so the user does not expect pick correlation to take relative depth into account. (A side effect of the strategy is that it optimizes pick correlation.) In shaded rendering modes, SPHIGS picks the candidate whose *hit point* (the NPC point, on the primitive's normalized (3D NPC) surface, to which the user pointed directly) is closest to the viewpoint—the one "in front," as discussed in Section 7.12.2.

Pick-correlation utility. To perform pick correlation, the application program calls a SPHIGS pick-correlation utility[13] with an NPC point and a view index, typically returned by a previous interaction with the locator:

```
void SPH_pickCorrelate (
    point position, int viewIndex, pickInformation *pickInfo);
```

The returned information identifies the primitive picked and its ancestry via a *pick path*, as described by Pascal data types in Fig. 7.24.

When no primitive is close enough to the cursor position, the value of *pickLevel* returned is 0 and the *path* field is undefined. When *pickLevel* is greater than 0, it specifies the length of the path from the root to the picked primitive—that is, the primitive's depth within the network. In this latter case, entries [1] through [*pickLevel*] of the *path* array return the identification of the structure elements involved in the path leading from root to picked primitive. At the deepest level (entry [*pickLevel*]), the element identified is the primitive that was picked; at all other levels (entries [*pickLevel*–1] through [1]), the elements are all structure executions. Each entry in *path* identifies one element with a record that gives the structure ID of the structure containing the element, the index of the element

[13]Full PHIGS has the Pick logical input device that returns the same measure as the SPH_pickCorrelate procedure.

```
typedef struct {
    int structureID;
    int elementIndex;
    /* Enumerated type: polyline, polyhedron, execute-structure, etc. */
    elementTypeCode elementType;
    int pickID;
} pickPathItem;

typedef pickPathItem pickPath[MAX_HIERARCHY_LEVEL];

typedef struct {
    int pickLevel;
    pickPath path;
} pickInformation;
```

Fig. 7.24 Pick-path storage types.

in that structure, a code presenting the type of the element, and the pick ID of the element (discussed next).

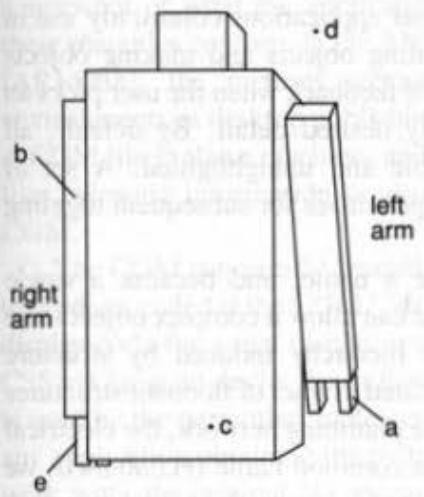
Figure 7.25 uses the structure network of Fig. 7.15 for the robot's upper body, and shows the pick information returned by several picks within the structure's displayed image.

How does the pick path uniquely identify each instance of a structure that is invoked arbitrarily many times within the hierarchy? For example, how do we distinguish a pick on the robot's left thumb from a pick on its right thumb? The pick paths for the two thumbs are identical except at the root level, as demonstrated by points *a* and *e* in Fig. 7.25.

The *pick identifier* can provide pick correlation at a finer resolution than does a structure ID. Although the element index can be used to identify individual elements, it is subject to change when the structure is edited. Therefore, using the pick ID is easier, because the pick ID is not affected by editing of other elements. It has a default value of 0 and is modally set within a structure. One generates a pick-ID element via

```
void SPH_setPickIdentifier (int id);
```

The pick-ID element is ignored during display traversal. Also, a pick ID has no notion of inheritance: it is initially 0 when SPHIGS begins the traversal of any structure, whether it is a root or a substructure. Because of these two aspects, pick IDs do not behave like attributes. Multiple primitives within a structure may have unique IDs or share the same one; this permits arbitrarily fine resolution of pick correlation within a structure, as needed by the application. Although labels and pick IDs are thus different mechanisms, the former used for editing and the latter for pick correlation, they are often used in conjunction. In particular, when structures are organized using the instance-block technique described in Section 7.9.2, a pick-ID element is also part of the block, and the pick ID itself is typically set to the same integer value as that of the block label.



Refer to the structure network shown in Fig. 7.15.

(a) level = 3
path[1] : struct UPPER _BODY, element 7
path[2] : struct ARM, element 3
path[3] : struct THUMB, element 1

(b) level = 2
path[1] : struct UPPER_BODY, element 11
path[2] : struct ARM, element 1

(c) level = 1
path[1] : struct UPPER_BODY, element 1

(d) level = 0

(e) level = 3
path[1] : struct UPPER_BODY, element 11
path[2] : struct ARM, element 3
path[3] : struct THUMB, element 1

Fig. 7.25 Example of pick correlation.

7.11 ADDITIONAL OUTPUT FEATURES

7.11.1 Attribute Bundles

Standard PHIGS provides a mechanism for setting attribute values indirectly. An application can, during its initialization sequence, store a collection of attribute values in an *attribute bundle*. Each type of primitive has its own type of bundle, and a PHIGS package provides storage for many bundles, each bundle identified by an integer ID. For example, we could store a "favorite" polyline attribute set in bundle 1. Subsequently, while editing a structure, we would prepare for the specification of a polyline primitive by inserting an element that, when executed during traversal, specifies that polyline attributes are to be taken from bundle 1 (rather than from the explicitly specified traversal attribute state).

Attribute bundles are often used as a "shorthand" to simplify the task of specifying attributes. Consider an application in which a large number of unrelated primitives must appear with identical attributes. Because the primitives are not related, the inheritance mechanism does not help. Indeed, without attribute bundles, the application would have to specify the desired attribute set redundantly, at various places throughout the structure networks.

Implementors of PHIGS packages sometimes initialize the attribute bundles in order to provide workstation-dependent preselected attribute sets that take advantage of the workstation's best features. The application programmer can choose to accept the bundles' initial values, as "suggestions" from the implementor, or to modify them with the bundle-editing commands. Changing definitions of bundles in the bundle table without changing structure networks is a simple mechanism for dynamically changing the appearance of objects.

7.11.2 Name Sets for Highlighting and Invisibility

SPHIGS supports two traditional feedback techniques that applications commonly use in conjunction with the SPHIGS picking facility: highlighting objects and making objects invisible. The former technique is typically used to provide feedback when the user picks an object; the latter declutters the screen by showing only desired detail. By default, all primitives that are part of a posted network are visible and unhighlighted. A set of primitives may be given an integer *name*, to identify the primitives for subsequent toggling of their visibility or highlighting status.

Because a group of unrelated primitives can share a name, and because a single primitive can have any number of names, the name feature can allow a complex object to be organized in several ways orthogonal to the structure hierarchy induced by structure invocation. For instance, an office-building model represented as a set of floor substructures could also be represented as a union of several systems: the plumbing network, the electrical wiring, and so on. Simply by giving all pipe primitives a common name (PLUMBING), we ensure that, even though the pipe primitives may be scattered among the actual structure hierarchy, we can nevertheless refer to them as a single unit.

When we want, say, to make the plumbing subsystem invisible, we add the name PLUMBING to the global *invisibility filter;* the screen is immediately updated to remove images of pipe objects. Similarly, by setting the invisibility filter to the names of all the

subsystems except the electrical subsystem, we can display the electrical subsystem in isolation. The highlighting filter works similarly. Both filters are initially empty, and are affected only by explicit calls that add or remove names.[14] Note that changing a filter, like changing a viewing specification, triggers screen regeneration; indeed, these operations change the rendered view of the CSS much as queries in traditional database programs are used to show different "views" of data.

The method used to bind names to primitives dynamically is very similar to the way attributes are assigned to primitives. SPHIGS maintains, as part of display traversal state, a *traversal name set* of zero or more names. A root inherits an empty name set. A child inherits the parent's name set when it is invoked, as it does attributes in general; thus, multiple instances of a building-block object can either share names or be named individually. The SPHIGS reference manual describes structure elements that, when executed during traversal, add names or remove names from this name set.

7.11.3 Picture Interchange and Metafiles

Although PHIGS and other standard graphics packages are system- and device-independent to promote portability, a given implementation of such packages in a particular environment is likely to be highly optimized in a nonportable way for performance reasons. The internal representation of the CSS, for example, may contain machine-specific information for structures and elements. To provide a medium of exchange among different implementations of PHIGS, the graphics standards committee has defined an archive file format. This portion of the standard is a machine- and environment-independent form of the contents of the CSS, without any viewing information. The PHIGS archive file thus is a portable snapshot of the structure database at a given time and permits PHIGS implementations to share geometric models.

PHIGS implementations may also support the writing of a metafile, which can contain a snapshot of what the application is currently presenting on the display surface. When these metafiles conform to the ANSI and ISO Computer Graphics Metafile (CGM) standard [ARNO88], the pictures contained in them can be transferred to such application environments as desktop publishing and interactive graphics art enhancement workstations. A CGM file is also a machine- and device-independent form of the CSS, but, unlike archive files, viewing information is also used in the creation of the picture represented in the CGM.

The CGM is typically created by having a PHIGS output device driver traverse the CSS to produce code for the CGM "virtual device," much as an ordinary device driver produces display code for a real display system. Other systems then can read the metafile into their CSS via an input device driver that converts from the prescribed metafile format to whatever is used by the particular implementation. Because the metafile is a 2D view of a 3D scene, any application obtaining the picture information via a CGM will have only the 2D view to work with: the original 3D geometry will be lost. If the 3D model is to be exchanged in a standard format, archive files must be used.

---

[14] The PHIGS detectability filter allows the application to specify primitives that cannot be picked. Moreover, PHIGS' filter scheme is more powerful, having separate inclusion and exclusion filters.

Other types of metafiles might be useful for debugging and backup purposes. An audit trail metafile is an historical transcript file containing a list of all calls to the graphics package procedures (and the values sent to them as parameters) in temporal order. It would, therefore, need to be run from start to finish, in order to recreate the CSS at the end of the session. Another type of transcript file records user actions: Running the application program with that transcript file reproduces the original sequence of PHIGS calls and thus the same CSS/image. No standards exist nor are standards planned for such transcript files, and neither a CGM nor a PHIGS archive file contains any historical information.

7.12 IMPLEMENTATION ISSUES

Much of the internal functionality of SPHIGS involves the maintenance (editing) and use (traversal) of the view table and the CSS. We do not discuss maintenance here, since it is essentially a conventional data-structures problem and not graphics-specific. Rather, this section focuses on the mechanics of displaying structures and of doing pick correlation.

The display traverser is invoked whenever the screen image must be updated. When implicit regeneration is allowed, the following operations prompt traversal: closing a structure, posting and unposting, changing a viewing transformation, changing rendering mode, and changing a filter. To generate the contents of a view, each structure posted to the view is traversed.

To display a posted structure network, SPHIGS visits the component structures' elements using a recursive descent, depth-first traversal, and performs the appropriate action for each element, based on the element's type. This display process that maps a model to an image on screen (or hardcopy) is referred to as display traversal in the context of PHIGS, but more generally as *rendering;* its implementation in software and/or hardware is referred to as the *rendering pipeline*.

Pick-correlation traversal is very similar to display traversal. The primitives encountered during traversal are compared to the locator position to find candidates. Traversal can be halted at the first candidate when wireframe rendering is being used; otherwise, a complete traversal is performed and the candidate closest to the viewpoint in z is chosen.

7.12.1 Rendering

The conceptual *rendering pipeline* that implements display traversal is illustrated in Fig. 7.26. Its first stage is the actual depth-first traversal of the CSS itself. (Alternatively, if an immediate-mode graphics package is used, the application may traverse the application model or generate primitives and attributes procedurally.) Each primitive encountered during traversal is passed through the remainder of the pipeline: First, the modeling transformations (described in Chapter 5) are applied to map the primitive from modeling coordinates to world coordinates. Then, the viewing operation is applied to transform and clip the primitive to the canonical view volume, and then to map it to the NPC parallelepiped (described in Chapter 6). Since these processes are independent of the display device and deal with vertex geometry in floating-point coordinates, this portion of

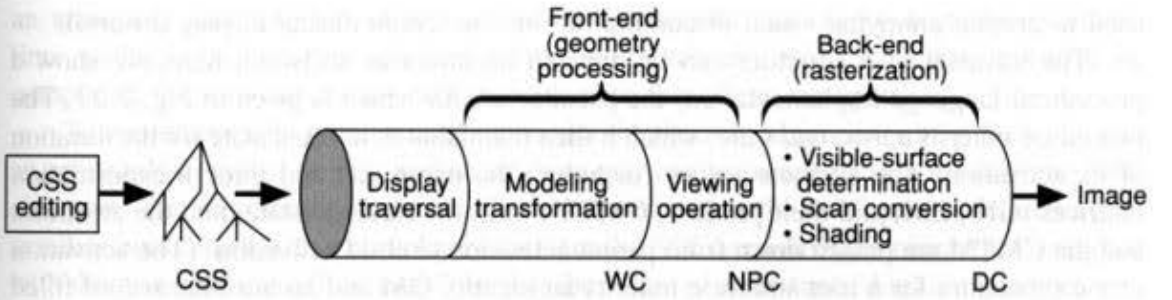


Fig. 7.26 The SPHIGS rendering pipeline.

the pipeline immediately following traversal is often referred to as the geometry-processing subsystem.

The back end of the pipeline takes transformed, clipped primitives and produces pixels; we will refer to this pixel processing as *rasterization*. This process is, of course, straightforward for wireframe mode: The NPC coordinates are easily mapped (via scaling and translating, with z ignored) to integer device coordinates, and then the underlying raster graphics package's line-drawing function is invoked to do the actual scan conversion. Shaded rendering, however, is quite complex, and is composed of three subprocesses: *visible-surface determination* (determining which portions of a primitive are actually visible from the synthetic camera's point of view), *scan conversion* (determining the pixels covered by a primitive's image), and *shading* (determining which color to assign to each covered pixel). The exact order in which these subprocesses are performed varies as a function of the rendering mode and implementation method. Detailed descriptions of the rasterization subprocesses are contained in Chapters 14 through 16.

Traversal. Since all stages of the rendering pipeline but the first one are covered in other chapters, we need to discuss only the traversal stage here. In a simple implementation, SPHIGS regenerates the screen by erasing it and then retraversing all posted roots (a list of which is stored with each view). Optimizing regeneration in order to traverse as little of the CSS as possible is quite difficult, because the effect of a trivial operation is potentially enormous. For example, it is difficult to determine how much of the screen must be regenerated due to the editing of a structure: It is possible that the structure is never invoked and thus has no effect on the screen, but it is also possible that it is a commonly invoked building block appearing in most of the views. Even when the implementation can determine that only a single view has been affected by an operation, refreshing that view may damage the images of objects in overlapping viewports of higher priority. Doing damage repair efficiently is, in general, a complicated task requiring considerable bookkeeping.

In implementations on high-performance workstations, where the image can be traversed and regenerated in a fraction of a second, the bookkeeping space and time overhead needed to regenerate the screen selectively is probably not worthwhile, and therefore complete regeneration is used most frequently. In either the complete or selective regeneration schemes, the double-buffering technique described later in this section can be

used to prevent annoying visual discontinuities on the screen during display traversal.

The traversal of a structure can be done in hardware or software; here, we show a procedural-language implementation, the pseudocode for which is given in Fig. 7.27. The procedure inherits a *traversal state*, which it then maintains as its local state for the duration of its activation. The attribute values (including the name set) and three transformation matrices (GM, LM, and their product, CMTM) form the traversal state, and the attributes and the CMTM are passed down from parent activation to child activation. (The activation of the procedure for a root structure inherits an identity GM and an attribute record filled with default values). The procedure visits each element in the structure and performs an operation based on the element's type, as indicated in Fig. 7.27.

Optimization via extent checking. The traversal strategy we have presented traverses a network's contents unconditionally; during traversal, all structure invocations are executed, and no part of the DAG is skipped. Frequently, however, not all of a network's objects are visible, since modeling and viewing tranformations in effect when the traversal is performed can cause large parts of a network to lie outside of the viewing volume.

What information do we need to trivially reject a subordinate structure during traversal? Say we have come to an element "Execute structure S" and can quickly compute the bounds of this instance, S_i, in NPC space. We could then ask the question, "Does S_i lie completely outside of the NPC viewport?" If it does, we skip over the structure-execution element and refrain from descending into structure S. Since structure S may be the root of

```
void TraverseStructureForDisplay (structID, attributeState, GM)
{
    LM = identity matrix;
    CMTM = GM;

    for (each element in structure structID) {
        switch (element type) {
            case attribute or name-set modification:
                update attributeState;
                break;
            case LM setting:
                replace or update LM;
                update CMTM by postconcatenating LM to GM;
                break;
            case primitive:
                pass primitive through rest of rendering pipeline;
                break;
            case execute structure:        /* A recursive call */
                TraverseStructureForDisplay
                    (ID of structure to be executed, attributeState, CMTM);
                break;
            default:                       /* ignore labels and pickIDs */
                break;
        }
    }
} /* TraverseStructureForDisplay */
```

Fig. 7.27 Pseudocode for display traversal.

an arbitrarily complex subnetwork, avoiding its traversal can reap potentially enormous time savings. In fact, this method allows us to trivially reject an entire network by comparing the root structure's NPC bounds with that of the viewport.

To implement this optimization, we need a simple method for computing the bounds of an arbitrarily complex object, and an efficient way of comparing these bounds to the NPC viewport. The representation that fits both these criteria is the *NPC extent*, defined as the smallest bounding box that completely encloses the NPC version of the object and is aligned with the principal axes. If we can show that the extent of an instance does not intersect with the viewport, we have proved that the instance is completely invisible. The extent is ideal because a viewport is itself an aligned box, and because it is very cheap to calculate the intersection of two aligned boxes (see Exercise 7.5). Trivial rejection is not the only optimization gained by the use of the NPC extent. We can also trivially accept—that is, discover instances that lie completely within the view volume and thus do not need to be clipped. The extent technique for trivial-accept/trivial-reject testing of substructures was first used in hardware in the BUGS system [VAND74] and is described also in [CLAR76].

Because instances are not explicitly stored in the CSS, we must calculate an instance's NPC extent from its structure's MC extent, which we store in the CSS. To perform extent checking during traversal, we thus first transform the MC extent to NPC, to determine the instance's NPC extent. Because a transformed extent box is not necessarily upright in NPC, we must determine the NPC extent of the transformed MC extent; that NPC extent is what we compare against the viewport.

To calculate the extent of a structure S, we must calculate the union of the extents of its primitives and descendants. We can calculate the extent of a polyhedron or fill area by traversing its MC vertex list, transformed by the local matrix, to determine minimum and maximum values of x, y, and z. These six numbers determine the extent: the box with one corner at $(x_{min}, y_{min}, z_{min})$ and the opposite corner at $(x_{max}, y_{max}, z_{max})$.

There is one other issue involved in maintaining extent information. When should the MC extent of a structure be calculated? It is not enough to do it whenever the structure is edited; a structure's MC extent is affected not only by its contents but also by the contents of any of its descendants. Thus, after each editing operation, an arbitrary number of structure extents must be recalculated. Moreover, recalculation requires traversal of an arbitrarily large subset of the CSS. We can optimize extent calculation by doing it not after each edit but during normal display traversal. This technique has the advantage of updating the extents of only those structures that are actually visible (part of a posted network), and it ensures that no structure's extent is calculated more than once in response to an arbitrarily large batch of editing operations. [SKLA90] presents this optimization technique in greater detail.

Animation and double-buffering. Software implementations of SPHIGS are well suited for rapidly producing animation prototypes, but are poorly suited for presenting high-quality real-time animations, because rendering is inherently time-consuming and the extent-checking optimization technique does not work well when a great deal of editing is performed between "frames." Animation prototypes are usually rendered in WIREFRAME mode, with automatic regeneration disabled, using a cyclical algorithm: The application

edits the structure database to describe the next scene, explicitly requests regeneration, then edits to describe the next scene, and so on. If the scene is not very complicated, WIREFRAME mode can be fast enough to present an almost–real-time animation of a modest number of primitives with just software. With some hardware assistance, higher-quality shading modes can perform in real-time.

One side effect of some simple implementations is that, between the "frames" of the animation, viewers see the screen being erased and can (more or less) see the objects being drawn as the traverser performs regeneration. An SPHIGS implementation can reduce this visual artifact by double buffering: using an offscreen canvas/bitmap to store the next frame while that frame is being drawn; then, when the regeneration is complete, the contents of that canvas can be copied onto the screen. When this technique is used on a system with a very fast copyPixel operation, the switch between frames is not noticeable, but there are still discontinuities in the motion of the objects in the scene. In fact, the *animation rate* (i.e., the number of frames per second) may decrease due to the increased overhead of the copyPixel, but the decrease in visual artifacts in many cases justifies the cost. Hardware double-buffering, as described in Section 4.4.1, is much better, since it avoids the copyPixel time.

7.12.2 Pick Correlation

In pick correlation, SPHIGS traverses those networks posted to the viewport in which the specified NPC point lies. The traversal is nearly identical to that performed during display; the modeling-transformation matrices are maintained, and much of the rendering pipeline is performed. Moreover, the pick ID, ignored during display traversal, is maintained as part of the local traversal state in the recursive traversal procedure. (The attribute group does not need to be maintained because during hit detection SPHIGS does not take into account attributes such as line thickness.)

Let us first examine pick correlation for wireframe rendering, where traversal is halted at the very first hit. Because traversal is recursive, the pick path is easily determined the moment the first hit is found: Each activation of the traversal procedure is responsible for one level of information in the pick path. Each primitive is transformed to NPC before being compared to the locator position for hit detection. (Hit-detection techniques are described later in this section.) When a hit is discovered by an activation of the procedure, it returns, and the recursion is unwound. Before returning, each activation stores its pick information into one level of a global pick-information array. (See Exercise 7.9.)

In shaded rendering modes, the order in which primitives are encountered during traversal has no bearing on whether they appear in front of or in back of other primitives whose images map to the same part of the screen. Therefore, the SPHIGS pick-correlation algorithm cannot simply choose the first hit detected. Rather, it must traverse all posted structure networks, maintaining a list of candidate hits. When traversal is completed, the candidates are compared to determine the candidate whose NPC hit point is closest to the viewpoint—that is, the one whose z coordinate is algebraically largest. To calculate a candidate's z coordinate at the hit point, we can plug the x and y coordinates of the locator measure into the appropriate equations for each primitive: the (parametric) line equation for edges, and the plane equation for facets (see Exercises 7.7 and 7.8). Another approach

using a hardware visible-surface determination algorithm is described in Section 15.4. Pick-correlation traversal can, of course, be optimized in a variety of ways, including the extent-checking techniques used to optimize display traversal.

Analytical hit detection. Two fundamental methods, analytical and clipping, are used for hit detection. For analytical hit detection, algebraic equations are used to determine whether the NPC primitive lies sufficiently close to the 2D NPC locator measure. We first convert to 2D by ignoring the z coordinate for an orthographic projection or use the perspective transform of Section 6.5.4 to create an orthographic view volume. Some examples of analytical techniques follow:

- In WIREFRAME mode, a function PtNearLineSegment is used to compute the distance in NPC from the cursor position to each edge of a facet or fill area, and to each line segment of a polyline. This same function would be used in shaded rendering modes for polyline primitives. The line equation is used for the computation (see Exercise 7.10).
- In shaded rendering modes, a function PtInPolygon is used to test for hits on fill areas and facets. One popular algorithm for determining if the NPC cursor position lies inside a polygon, based on the odd-parity rule described in Section 2.1.3, casts a ray from the locator position and determines how many times the ray intersects the polygon. The algorithm traverses the edge list, testing for intersections and special cases (e.g., intersections at vertices, edge–ray colinearity). The polygon scan-conversion algorithm described in Section 3.7 tackles a very similar problem and can be adapted for use as a PtInPolygon function (see Exercise 7.12). This algorithm handles the general case of concave and self-intersecting polygons. Optimized computational geometry algorithms are available if it can be guaranteed that polygons do not self-intersect, or that a horizontal ray intersects only two edges, or that the polygon is convex [PREP85].
- Hit detection for nongeometric text is most easily performed by comparing the locator position to the text's rectangular screen extent.
- Packages that support primitives such as ellipses and curves and surfaces require more complex pick-correlation algorithms of the sort mentioned in Chapters 11 and 19. Furthermore, the problems are similar to those encountered in ray tracing, as described in Chapter 15.

Hit detection via clipping. Some hardware clipping devices and optimized software clipping utilities return state information, allowing an application to determine whether any part of a primitive's image lies inside a 2D integer clip rectangle, without having actually to draw the primitive. An SPHIGS implementation can use this type of clipping to test for candidacy: The clip rectangle is set to a *pick window*—a small square surrounding the cursor position—and then traversal is performed. Each primitive (transformed to integer device coordinates) is given to the clipper, which returns a Boolean "hit-detection" result (see Exercise 7.11). Alternatively, if the clipper does not return any such state information, we can draw each primitive into an offscreen bitmap using the pick-window clip rectangle; if any pixels are changed, the primitive is deemed a candidate.

7.13 OPTIMIZING DISPLAY OF HIERARCHICAL MODELS

7.13.1 Elision

We can model a building as a parts hierarchy by saying that it consists of floors, the floors consist of offices, and so on; there are no primitives in the hierarchy's nodes until we get to the level of bricks, planks, and concrete slabs that consist of polyhedra. Although this representation might be useful for construction purposes, it is not as useful for display, where we sometimes wish to see a cruder, simplified picture that eliminates confusing and unneeded details (and that is faster to render). The term *elision* refers to the decision by a display traverser to refrain from descending into a substructure.

Pruning. In Section 7.12.1, we showed how a display traverser can avoid executing a subordinate structure by checking its NPC extent against the viewport, to determine whether the substructure lies wholly outside (i.e., is fully clipped by) the view volume. This type of elision, a typical feature of optimized display traversers, is called *pruning*.

Culling. In addition, a traverser can examine the size of the subordinate's NPC extent, and can choose to elide the substructure if that substructure's extent is so small that, after transformation and projection, the image of the object would be compressed into a few pixels. This type of elision is called *culling*; on systems that support it, the application typically can specify a minimum extent size, below which a substructure is culled.

When a substructure is pruned, it is not drawn at all; however, that is not the best approach for culling. The object is being culled not because it is invisible from the current point of view, but because its image is too tiny for its details to be discernible. Rather than not draw it at all, most implementations draw it as an abstract form, typically a parallelipiped representing its WC extent, or simply a rectangle (the 2D projection of its NPC extent).

Level-of-detail elision. Pruning and culling are optimization techniques, preventing traversal that is unnecessary (pruning) or that would produce an unusable image (culling). Elision can also be used to give the user control over the amount of detail presented in a view of the CSS. For example, a user examining our building model could specify a low level of detail in order to view the building as simply a set of parallelepipeds representing the individual floors, or could increase the level of detail so as to see in addition the walls that form office boundaries.

The MIDAS microprocessor architecture simulator [GURW81] was one of the earliest systems in which alternate representations of subobjects were traversed automatically by the display processor, depending on the size of the projection of the subobject on the screen. This logical-zoom facility made successively more detail appear as the user dynamically zoomed in on the processor block in the CPU architecture block diagram. Also, at increased zoom factors, one could see digits representing address, data, and control bytes moving from source to destination over system buses during the simulation of the instruction cycle.

Elision in MIDAS was implemented using a conditional-execution facility of the BUGS vector-graphics system [VAND74]: The hierarchical display list included a conditional substructure execution element that tested the screen extent of the substructure. A similar

feature described in the 1988 specification of PHIGS+ in the form of a conditional-execution element, allowing pruning and culling to be performed explicitly by elements within the CSS.

7.13.2 Structure Referral

Certain implementations of PHIGS and PHIGS+ allow a nonstandard form of structure execution, called *referral*, that bypasses the expensive state saving and restoring of the ExecuteStructure mechanism. Whereas we could argue that a better, transparent approach to increasing efficiency is to optimize the implementation of the ExecuteStructure operation itself (so that the operation saves only as much state as required at any given time), it is simpler to add a ReferStructure operation and to let the programmer use it for those cases where an invoked child structure does not have any attribute-setting elements.

A second use of ReferStructure is to allow a child to influence its parents' attributes. This is useful when a group of objects do not have a common parent, but do need to "inherit" the same group of attributes, including potentially an arbitrary number of modeling transformations. In this case, we can create a structure *A* that consists of transformation and appearance attribute settings, and then have each of the object structures refer to structure *A*. By editing structure *A* later, we indirectly affect all structures referring to *A*. If only appearance attributes need to be affected, PHIGS attribute bundles provide an alternative, standard mechanism for changing the appearance of diverse structures.

7.14 LIMITATIONS OF HIERARCHICAL MODELING IN PHIGS

Although this chapter has emphasized geometric modeling hierarchy, it is important to realize that hierarchy is only one of many forms of data representation. In this section, we discuss the limitations of hierarchy in general and in PHIGS in particular; in the next section, we present some alternatives to structure hierarchy.

7.14.1 Limitations of Simple Hierarchy

As mentioned in Section 1.7.2, some applications have no real structure for their data (e.g., data for scatter plots), or have at most a (partial) ordering in their data (e.g., a function represented algebraically). Many other applications are more naturally expressed as networks—that is, as general (directed) graphs (which may have hierarchial subnets). Among these are circuit and electrical-wiring diagrams, transportation and communications networks, and chemical-plant–piping diagrams. Another example of simple hierarchy's insufficiency for certain types of models is Rubik's cube, a collection of components in which the network and any hierarchy (say of layers, rows, and columns) is fundamentally altered after any transformation.

For other types of models, a single hierarchy does not suffice. For example, the pen holder on an (x, y) plotter is moved by, and therefore "belongs" to, both the horizontal and vertical arms. In short, whether the application model exhibits pure hierarchy, pure network without hierarchy, hierarchy in a network with cross-links, or multiple hierarchies, SPHIGS can be used to display it, but we may not want, or be able, to use structure hierarchy in its full generality.

7.14.2 Limitations of SPHIGS "Parameter Passing"

The black-box nature of structures is good for modularity but, as shown in our robot example, can be limiting. For example, how can we build a robot with two identical arms and have the robot use its right arm to pick up a cylinder from a table and move away with that cylinder? The pick-up operation can be performed by editing the arm structure to add an invocation of the cylinder structure, so that as the robot or arm moves subsequently, the cylinder moves along with it. But if we implement the robot by having a single arm structure invoked twice by the upper-body structure, the result of that operation would be that both the left and right arm would be holding a cylinder! Thus, we have no choice but to build two separate arm structures, with unique structure IDs, each invoked only once. Let us extend this example. If we wish to have an army of robots with independently controlled arms, we must build two unique arm structures for each robot! Obviously, here is a case where structure hierarchy is not as useful as we first thought.

The reason that structure hierarchy does not support instances of structures differing in the settings of transformations at various hierarchical levels is that structure hierarchy has neither the general parameter-passing mechanism of procedure hierarchy nor general flow-of-control constructs. Rather, it is essentially a data organization with rudimentary interactions between structures and at most limited conditional execution of structures (in PHIGS+). We have seen that a parent's transformations are inherited by all the children, and there is no provision for a particular child to be affected selectively.

By contrast, in a procedure hierarchy, a "root" procedure passes parameters that are either used directly by procedures the root calls, or passed down by those procedures to lower-level procedures. Thus, the root can pass down parameters arbitrarily deeply and selectively via intermediate procedures. Furthermore, with parameters, a procedure can control not just the data on which a lower-level procedure operates, but even the way in which the lower-level procedure operates. To change its operation, the lower-level procedure uses flow-of-control constructs to test parameters and selectively enables or disables code segments. Because of structure hierarchy's lack of general parameter-passing and flow-of-control mechanisms, our analogy between structure hierarchy and procedure hierarchy in the introduction was a superficial one.

By augmenting the attribute-binding model of PHIGS, we could specify attributes for selected object instances at arbitrary levels in a hierarchy. A system that has such a general mechanism is SCEFO [STRA88], which allows the programmer to specify an attribute that is to be applied when the traversal reaches a certain state, the state being represented by a pathname similar to the pick path returned by the PHIGS pick device. With this facility, it would be possible to control individually the colors or positions of the thumb instances in our robot, without having to create two virtually identical arm masters, by making use of the fact that the two thumb instances have unique pathnames.

Another limitation in the PHIGS parameter-passing mechanism is that it handles transformations and appearance attributes for which inheritance rules are very simple. It would not be easy to support operations more complex than geometric transformations; a more general model is needed to support set operations on solid primitives (Chapter 12), and deformation operations such as bend, taper, and twist (Chapter 20).

7.15 ALTERNATIVE FORMS OF HIERARCHICAL MODELING

We have just concluded that structure hierarchy is only one way—and not always the best way—to encode hierarchy. In this section, we discuss alternatives to structure hierarchy.

7.15.1 Procedure Hierarchy

In the spectrum from pure data hierarchy to pure procedure hierarchy, structure hierarchy is almost all the way at the data end, since it lacks general flow of control. By contrast, a template procedure (i.e., a procedure defining a template object, consisting of primitives or of calls to subordinate template procedures) can use parameters and arbitrary flow of control. Template procedures can be used in two different ways. First, they can be used with a retained-mode graphics package such as SPHIGS. Here, they are used as a means to the end of creating structure hierarchy. Second, they can be used to specify primitives and attributes to an immediate-mode graphics package. Here, they are not means to ends, but rather are ends in themselves; that is, they are the sole representation of the hierarchy used for display. In this case, display traversal is effected by procedure traversal—practical only if the CPU can provide a reasonable rate of retraversal. (In general, smooth dynamics requires at least 15 frames per second; 30 frames per second looks noticeably better.) The procedures themselves implement inheritance and maintain transformation matrices and attribute states, using techniques similar to those presented in our discussion of traversal implementation in Section 7.12. Newman's display procedures mentioned in Section 7.4 were an early example of the use of procedure hierarchy for dynamics.

Pick correlation is a bit tricky in dynamic procedure traversal, since it requires retraversing the procedure hierarchy. If the first candidate is to be chosen (i.e., if the rendering mode is wireframe), it is difficult to halt the traversal as soon as a hit is detected and to return from an arbitrary level of the procedure activation stack. If a nonwire frame rendering mode is used, there must be a way of interacting with the rendering pipeline to obtain the candidates and their z values corresponding to the cursor position. Each procedure is also complicated by the fact that it must be used both for display and for correlation traversal.

We can combine procedure and structure hierarchy by using template procedures to create structures. For example, our robot can be built using template procedures for each of the parts; each procedure creates display commands or a SPHIGS structure, depending on whether we choose immediate mode or retained structure mode, respectively. The upper-body procedure can pass parameters to the arm and thumb procedures to initialize their transformations individually and to allow the arms and thumbs to operate independently. In retained mode, each invocation of the robot template procedure thus creates a unique SPHIGS network, with unique names for all the child structures. For example, two arm structures would be created, with their own, separate invocations of thumbs. Leaf nodes such as the thumb can still be shared among multiple networks, since they can be instanced with individual transforms by their callers. To create unique structure IDs, we can assign a unique interval of integer space to each root, and can number its parts within that interval.

What are the limitations of using only procedure hierarchy? First, unless the programming environment supports dynamic code creation, it is difficult to edit or construct procedures at run time. Creating new data structures is far easier. Therefore, procedure hierarchy is typically used when the set of template objects is predefined and only the attributes need to be varied dynamically. Structures created by template procedures can, of course, be edited with all the PHIGS machinery.

Second, if the procedure hierarchy is used to create immediate-mode graphics, the CPU, involved in constantly retraversing the procedure hierarchy, is much less available for application processing. We can offload the CPU in a multiprocessor system by having another CPU dedicated to procedure traversal. Alternatively, in retained mode, structure-network traversal can be performed by a special-purpose hardware coprocessor or a separate CPU.

Third, even for minor edits, an immediate-mode procedure hierarchy must be retraversed in its entirety and all display primitives must be retransmitted to the graphics packages. For display systems connected over networks or communication lines, this requirement produces heavy communication traffic. It is far more efficient for minor edits, for both the CPU and the communications system, to store and manage a structure database in a display peripheral that can be modified incrementally and traversed rapidly.

7.15.2 Data Hierarchy

Unlike procedure hierarchy, data hierarchy is well suited to dynamic creation. Like template-procedure hierarchy, it can be used in conjunction with either immediate- or retained-mode graphics packages. If immediate mode is used, the CPU must retraverse the application model and drive the package sufficiently quickly to provide dynamic update rates. Objects are created and edited by changing the application model and retraversing it to update the screen. The application must do its own pick correlation, by retraversal. As with an immediate-mode procedure hierarchy, however, if the display subsystem is on a communications network, retransmittal of graphics commands for any update is considerably slower than is sending update commands to a structure database in the peripheral.

Like the structure-hierarchy technique, the data-hierarchy approach lacks the flexibility of the procedure-hierarchy method due to the absence of general flow-of-control mechanisms; these must be embodied via flags in the data structures. Object-oriented environments with run-time code creation and binding offer an attractive, totally general combination of data and procedure hierarchy; there is a natural match between the notion of an object–subobject hierarchy and that of a class–instance hierarchy. As processor performance improves, object-oriented environments are likely to become a dominant paradigm in dynamic computer graphics.

7.15.3 Using Database Systems

Since a general-purpose database has more power than does a special-purpose one, we should consider using standard database systems for computer graphics [WELL76; GARR80]. Unfortunately, such databases are designed to work with large volumes of data in secondary storage and to give response times measured on a human time scale. They are

designed to process user-input queries or even batch transactions with times measured, at best, in milliseconds, whereas real-time graphics demands microsecond access to elements. Using a memory-resident database would work best if the database were optimized for fast traversal and had built-in graphics data types and operators. At least, it would have to be able to invoke procedures for retrieved items, passing parameters extracted from fields in the database.

Although several systems have used relational databases for graphics, the limitations on the structure of the data imposed by the relational model, as well as the slowness of standard relational databases, have restricted these systems to research environments. As object-oriented environments are useful for combining data and procedures, such an environment used in conjunction with an object-oriented database has the potential for removing the restrictions of relational databases. The slow performance of object-oriented databases, however, may make them infeasible for real-time graphics in the near term.

7.16 SUMMARY

This chapter has given a general introduction to geometric models, emphasizing hierarchical models that represent parts assemblies. Although many types of data and objects are not hierarchical, most human-made objects are at least partly so. PHIGS and our adaptation, SPHIGS, are designed to provide efficient and natural representations of geometric objects stored essentially as hierarchies of polygons and polyhedra. Because these packages store an internal database of objects, a programmer can make small changes in the database with little effort, and the package automatically produces an updated view. Thus, the application program builds and edits the database, typically in response to user input, and the package is responsible for producing specified views of the database. These views use a variety of rendering techniques to provide quality–speed tradeoffs. The package also provides locator and choice input devices, as well as pick correlation to allow the selection of objects at any level in a hierarchy. Highlighting and visibility filters can be used for selective enabling and disabling as another form of control over the appearance of objects.

Because the nature of structures and the means for searching and editing them are restricted, such a special-purpose system is best suited to motion dynamics and light update dynamics, especially if the structure database can be maintained in a display terminal optimized to be a PHIGS peripheral. If much of the structure database must be updated between successive images, or if the application database can be traversed rapidly and there is no bottleneck between the computer and the display subsystem, it is more efficient to use a graphics package in immediate mode, without retaining information.

Structure hierarchy lies between pure data and pure procedure hierarchy. It has the advantage of dynamic editing that is characteristic of data hierarchy. It also allows a simple form of parameter passing to substructures (of geometric or appearance attributes), using the attribute-traversal state mechanism. Because of the lack of general flow-of-control constructs, however, the parameter-passing mechanism is restricted, and structures cannot selectively set different attributes in different instances of a substructure. Instead, template procedures can be used to set up multiple copies of (hierarchical) structures that are identical in structure but that differ in the geometric or appearance attributes of substructures. Alternatively, they can be used to drive an immediate-mode package.

SPHIGS is oriented toward geometric models made essentially from polygons and polyhedra, especially those that exhibit hierarchy; in Chapters 11 and 12, we look at geometric models that have more complex primitives and combinations of primitives. Before turning to those more advanced modeling topics, we first consider interaction tools, techniques, and user interfaces.

EXERCISES

7.1 a. Complete the robot model of Fig. 7.16 by adding a base on which the upper body swivels and create a simple animation of its movement through a room.

b. Create an SPHIGS application producing an animation in which a one-armed robot approaches a table on which an object lies, picks up the object, and walks off with it (see Section 7.14 for the reason for specifying a one-armed robot).

7.2 Enhance the robot animation to provide user interaction. Let there be a number of objects on the table, and allow the user to choose (using the locator) the object that the robot should pick up.

7.3 Redesign the two-armed robot model so as to allow the thumbs on each arm to be controlled individually, so that each arm can pick up objects individually.

7.4 Enhance a robot animation so that three views of the animation are shown simultaneously, including one overhead orthographic view and one "robot's eye" view that shows us what the robot itself "sees" as it moves.

7.5 Design the addition of pruning elision to the recursive display traverser of Fig. 7.27. Assume the MC extent of a structure is stored in the structure's record. You must transform an MC extent box into an NPC extent box, meeting the requirement that extent boxes be aligned with the principal axes.

7.6 Update our recursive display traverser so that it maintains the MC extent information stored for each structure. Assume that, whenever a structure S is closed after being edited, a Boolean "extentObsolete" field in S's record is set. Assume also that functions are available that, given any primitive, return the primitive's NPC extent.

7.7 Design an algorithm for calculating analytically the hit point of a candidate line, given the line's NPC endpoints and the locator measure.

7.8 Design an algorithm for calculating analytically the hit point of a candidate fill area.

7.9 Design, using pseudocode, a recursive pick-correlation traverser that supports only wireframe mode.

7.10 Implement the function PtNearLineSegment analytically for use in pick correlation. To be a candidate, the line segment's image must come within P pixel widths of the locator position.

7.11 Implement the function PtNearLineSegment using clipping. Modify the Liang–Barsky clipping algorithm (of Fig. 3.45) to optimize it, because the clipped version of the segment is not needed—only a Boolean value is to be returned.

7.12 Implement the function PtInPolygon for use in pick correlation. Treat the special cases of rays that pass through vertices or are coincident with edges. See [PREP85] and [FORR85] for discussions of the subtleties of this problem.

7.13 Design a user interface for picking that lets the user indicate the desired level of a hierarchy. Implement and test your interface with the robot model by writing an application that allows the user to highlight portions of the robot's anatomy, from individual parts to whole subsystems.

8 Input Devices, Interaction Techniques, and Interaction Tasks

This is the first of three chapters on designing and implementing graphical user–computer interfaces. 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.

Interest in the quality of user–computer interfaces is new in the formal study of computers. The emphasis until the early 1980s was on optimizing two scarce hardware resources, computer time and memory. Program efficiency was the highest goal. With today's plummeting hardware costs and increasingly powerful graphics-oriented personal-computing environments (as discussed in Chapter 1), however, we can afford to optimize user efficiency rather than computer efficiency. Thus, although many of the ideas presented in this chapter require additional CPU cycles and memory space, the potential rewards in user productivity and satisfaction well outweigh the modest additional cost of these resources.

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. Indeed, in such critical applications as air-traffic control and nuclear-power-plant monitoring, a poor user interface can contribute to and even cause accidents of catastrophic proportions.

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 designer of an interactive graphics application must be sensitive to users' desire for easy-to-learn yet powerful interfaces.

On the other hand, in the future, the level of computer sophistication of workers will increase, as more users enter the workforce already computer-literate through computer use at home and school. Developers of some educational and game software will continue to design for the computer-naive user, while developers of workplace systems will be able to assume an increased awareness of general computer concepts.

In this chapter, we discuss basic elements of user interfaces: input devices, interaction techniques, and interaction tasks. Input devices were introduced in Chapters 2 and 4: here we elaborate on their use. 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.

In Chapter 9, we discuss the issues involved in putting together the building blocks into a complete user-interface design. The emphasis is on a top-down design approach; first, design objectives are identified, and the design is then developed through a stepwise refinement process. The pros and cons of various dialogue styles—such as what you see is what you get (WYSIWYG), command language, and direct manipulation—are discussed, and window-manager issues that affect the user interface are also described. Design guidelines, the dos and don'ts of interface design, are described and illustrated with various positive and negative examples. Many of the topics in Chapters 8 and 9 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], and Shneiderman [SHNE87]; the reference book by Salvendy [SALV87]; and the survey by Foley, Wallace, and Chan [FOLE84].

Many of the examples used in Chapters 8 and 9 are taken from the user interface of Apple Computer's Macintosh. Although the Macintosh user interface is not perfect, it is a huge improvement over previous commonly available interfaces. Developed in the early 1980s, the Macintosh was built primarily on pioneering work at Xerox Corporation's Palo Alto Research Center (PARC) in the mid-1970s, and has been imitated and in some cases extended by systems such as Microsoft Windows, Presentation Manager, NeXT's NeXT Step, the Commodore Amiga, Digital Research's GEM, and many others. (Much of this book was written on the Macintosh, using Microsoft Word, and many of the figures were drawn on the Macintosh using Freehand.)

Chapter 10 treats user-interface software. It is one thing to design graphic user interfaces that are easy to learn and fast to use; it is quite another to implement them. Having the right software tools is of critical importance. This chapter reviews the input-handling capabilities of SRGP and SPHIGS, and then discusses more general and more powerful input-handling capabilities. The internal structures and implementation strategies of window managers, a critical element in many high-quality user interfaces, are described. Finally, the key concepts of user-interface management systems (UIMSs) are presented. UIMSs provide a means for interface designers and implementors quickly to develop, try out, and modify their interface concepts, and thus decrease the cost of the

essential testing and refinement steps in user-interface development.

We focus in this chapter on input devices—those 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.

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.

By analogy with a natural language, single actions with input devices are similar to the individual letters of the alphabet from which words are formed. The sequence of input-device actions that makes up an interaction technique is analogous to the sequence of letters that makes up a word. A word is a unit of meaning; just as several interaction techniques can be used to carry out the same interaction task, so too words that are synonyms convey the same meaning. All the units of meaning entered by the user can be categorized as one of the four basic interaction tasks, just as words can be categorized as verb, noun, adjective, and so on. An interactive dialogue is made up of interaction-task sequences, just as a sentence is constructed from word sequences.

8.1 INTERACTION HARDWARE

Here, we introduce some interaction devices not covered in Section 4.6, 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 4.6, 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. Dialogue-level issues are discussed in Chapter 9, where we deal with constructing complete user interfaces from the building blocks introduced in this chapter. Much confusion can be avoided when we think about devices if we keep these three levels in mind.

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. Also omitted are the details of connecting devices to computers; by far the most common means is the serial asynchronous RS-232 terminal interface, generally making interfacing quite simple.

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 desk top, 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 light pen or 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, is creating 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 the *control-to-display ratio*, commonly called the C/D ratio [CHAP72]; it is the ratio between hand movement (the control) and cursor movement (the display). A large ratio is good for accurate positioning, but makes rapid movements tedious; a small ratio is good for speed but not for accuracy. Fortunately, for a relative positioning device, the ratio need not be constant, but can be changed adaptively as a function of control-movement speed. Rapid movements indicate the user is making a gross hand movement, so a small ratio is used; as the speed decreases, the C/D ratio is increased. This variation of C/D ratio can be set up so that a user can use a mouse to position a cursor accurately across a 15-inch screen without repositioning her wrist! For indirect discrete devices (cursor-control keys), there is a similar technique: the distance the cursor is moved per unit time is increased as a function of the time the key has been held down.

Precise positioning is difficult with direct devices, if the arm is unsupported and extended toward the screen. Try writing your name on a blackboard in this pose, and compare the result to your normal signature. This problem can be mitigated 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 the fingers can be used more effectively. Not all continuous indirect devices are equally satisfactory for drawing, however. Try writing your name with a joystick, a mouse, and a tablet pen stylus. Using the stylus is fastest, and the result is most pleasing.

Other interesting positioning devices include the Versatron *footmouse* [VERS84], which remains static on the floor: The user places the ball of his foot on the device, keeping his heel on the floor, and controls the footmouse with left–right and forward–backward movements. The experimental *mole* is a pivoted foot rest with integrated switches [PEAR86, PEAR88] that, like the footmouse, leaves the hands free. The Personics *headmouse* [PERS85] uses a head-mounted set of three microphones to measure the distance to a sound source, translating small rotational movements of the head into cursor movements. *Eye trackers* can determine where the eye is pointing and hence can cause a cursor to move or the object pointed at to be selected [BOLT80; BOLT84; WARE87]. These devices are often less accurate and considerably more expensive than are the more traditional devices, and thus would normally be considered for only hands-free applications. The 3D positioning devices discussed in Section 8.1.6 can also be used for 2D positioning.

8.1.2 Keyboard Devices

The well-known QWERTY keyboard has been with us for many years. It is ironic that this keyboard was originally designed to *slow down* typists, so that the typewriter hammers would not be so likely to jam. Studies have shown that the newer Dvořák keyboard [DVOR43], which places vowels and other high-frequency characters under the home

positions of the fingers, is somewhat faster than is the QWERTY design [GREE87]. It has not been widely accepted. Alphabetically organized keyboards are sometimes used when many of the users are nontypists. But more and more people are being exposed to QWERTY keyboards, and several experiments have shown no advantage of alphabetic over QWERTY keyboards [HIRS70; MICH71].

The *chord keyboard* has five keys similar to piano keys, and is operated with one hand, by pressing one or more keys simultaneously to "play a chord." With five keys, 31 different chords can be played. Learning to use a chord keyboard (and other similar stenographer-style keyboards) takes longer than learning the QWERTY keyboard, but skilled users can type quite rapidly, leaving the second hand free for other tasks. This increased training time means, however, that such keyboards are not suitable substitutes for general use of the standard alphanumeric keyboard.

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

8.1.3 Valuator Devices

Some valuators are *bounded*, like the volume control on a radio—the dial can be turned only so far before a stop is reached that prevents further turning. A bounded valuator inputs an absolute quantity. A continuous-turn potentiometer, on the other 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. For example, used with a headmouse (described in Section 8.1.1), a foot switch could easily provide functionality equivalent to a single-button mouse.

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 or not 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 very limited vocabularies—typically, they include only the digits and 50 to 100 words. Some discrete-word recognizers can recognize vocabularies of up to a few thousand different words after appropriate training. But if the user has a cold, the recognizer must be retrained. The user of a recognizer must pause for a fraction of a second after each word to cue the system that a word end has occurred; the pause is typically 100 to 200 milliseconds, and can be longer if the set of possible words is large. The more difficult task of recognizing connected speech from a limited vocabulary can also be performed by commercial hardware, but at a higher cost. The larger the vocabulary, however, the more artificial-intelligence techniques are needed to use the context and meaning of a sequence of sentences to remove ambiguity. A few systems with vocabularies of 20,000 or more words can recognize sentences such as "Write Mrs. Wright a letter right now!"

Voice synthesizers create waveforms that approximate, with varying degrees of realism, spoken words [KAPL85]. The simplest synthesizers use *phonemes*, the basic sound units that form words. This approach creates an artificial-sounding, inflection-free voice. More sophisticated phoneme-based systems add inflections. Other systems actually play back digitized speech patterns. They sound realistic, but require thousands of bytes of memory to store the digitized speech.

Now that several personal computers, including the Macintosh and NeXT, have standard sound synthesizers that can create both voice and music, speech feedback from computers is becoming quite common. Speech is best used to augment rather than to replace visual feedback, and is most effective when used sparingly. For instance, a training application could show a student a graphic animation of some process, along with a voice narration describing what is being seen. See [SIMP85; SIMP87] for an extensive review of speech recognition and generation, including additional guidelines for the effective application of these functions in user–computer interfaces.

Sound generators can be used to generate musical tones and other effects, which can call attention to specific situations, especially if the user is unlikely to be looking at the display. For instance, "printer out of paper" or "memory nearly full" alarms might be signaled by two different tones, in addition to messages on the screen. An attempt to reorient a line that has been constrained to be parallel to another line might cause a warning beep.

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 [GTCO82] senses not only stylus pressure but orientation as well. The resulting 5 degrees of freedom reported by the tablet can be used in various creative ways. For example, Bleser, Sibert, and McGee implemented the GWPaint system to simulate various artist's tools, such as an italic pen, that are sensitive to pressure and orientation [BLES88a]. Figure 8.1 shows the artistic creativity thus afforded.

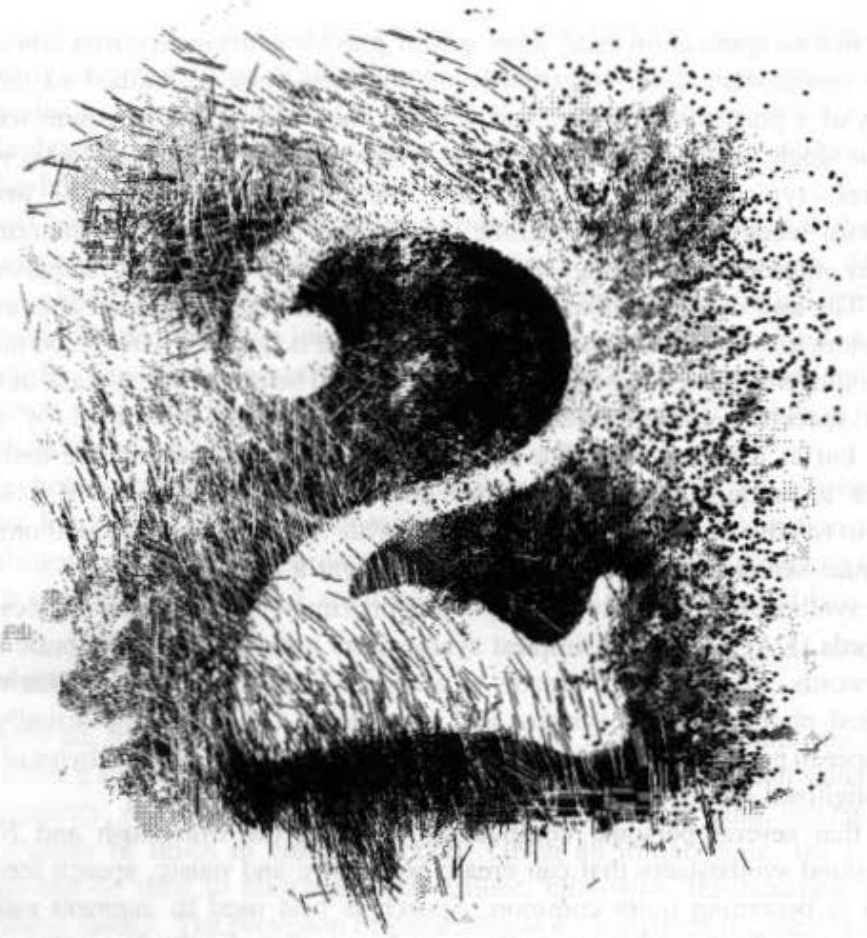


Fig. 8.1 *Numeral 2*, a drawing in the spirit of Jasper Johns, by Teresa Bleser. Drawn with the GWPaint program using a GTCO pressure- and tilt-sensitive tablet. (Courtesy of T. Bleser, George Washington University.)

Green [GREE85] applied optics principles to develop a tabletlike device that gives artists even more freedom than the pressure- and tilt-sensing tablet. The user paints on the tablet with brushes, hands, or anything else that is convenient. A television camera positioned below the tablet records the shape and motion of the brush wherever it contacts the tablet, and the resulting video signal is used to load the refresh buffer of a raster display. Resolution of 500 or 1000 units is achievable, depending on the television camera used.

An experimental touch tablet, developed by Buxton and colleagues, can sense multiple finger positions simultaneously, and can also sense the area covered at each point of contact [LEE85a]. The device is essentially a type of touch panel, but is used as a tablet on the work surface, not as a touch panel mounted over the screen. The device can be used in a rich variety of ways [BUXT85]. Different finger pressures correlate with the area covered at a point of contact, and are used to signal user commands: a light pressure causes a cursor to appear and to track finger movement; increased pressure is used, like a button-push on a mouse or puck, to begin feedback such as dragging of an object; decreased pressure causes the dragging to stop.

Another way to obtain more than just position information is to suspend a touch panel in front of a display using a few metal support strips with strain gauges [HERO78, MINS84]. Pressure applied to the touch panel translates into strain measured by the gauges. It is possible also to measure the direction of push and torque as well, by orienting the metal strips in appropriate directions. The measurements can be used to cause displayed objects to be rotated, scaled, and so on.

The decreasing costs of input devices and the increasing availability of computer power are likely to lead to the continuing introduction of novel interaction devices. Douglas Engelbart invented the mouse in the 1960s, and nearly 20 years passed before it became popular [PERR89]. What the next mouse will be is not yet clear, but we hope that it will have a much shorter gestation period.

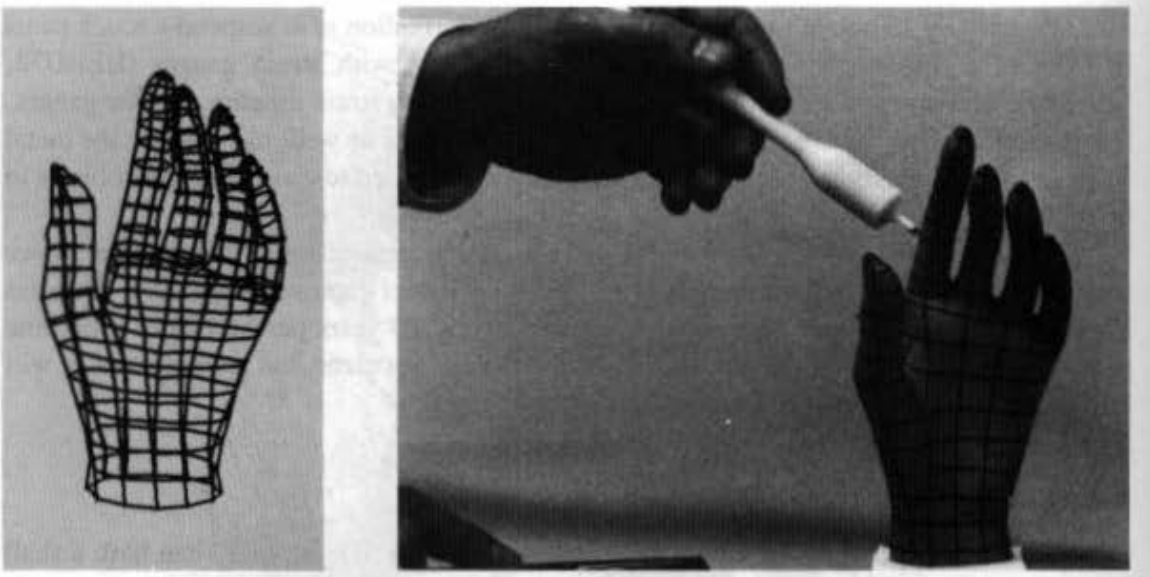
8.1.6 3D Interaction Devices

Some of the 2D interaction devices are readily extended to 3D. Joysticks can have a shaft that twists for a third dimension (see Fig. 4.38). 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.

The Spaceball (see Color Plate I.14) is a rigid sphere containing strain gauges. The user pushes or pulls the sphere in any direction, providing 3D translation and orientation. In this case, at least the directions of movement correspond to the user's attempts at moving the rigid sphere, although the hand does not actually move.

A number of devices, on the other hand, can record 3D hand movements. The experimental *Noll Box*, developed by Michael Noll, permits movement of a knob in a 12-inch cube volume, sensed by slider mechanisms linked to potentiometers. The Polhemus 3SPACE three-dimensional 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 8.2 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. 8.3, 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 I.15 illustrates this concept.



(a) (b)

Fig. 8.2 (a) A wireframe display of the result. (b) The Polhemus 3D position sensor being used to digitize a 3D object. (3Space digitizer courtesy of Polhemus, Inc., Colchester, VT.)

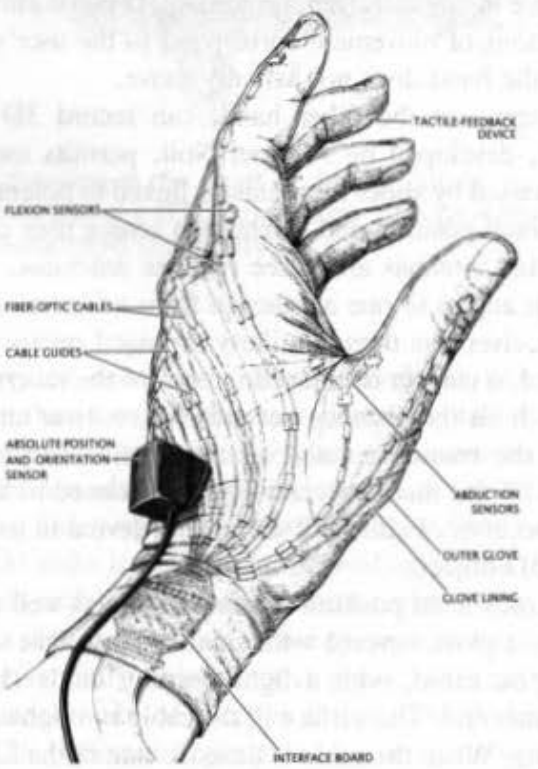


Fig. 8.3 The VPL DataGlove, showing the fiberoptic calbes that are used to sense finger movements, and the Polhemus position and orientation sensor. (From J. Foley, Interfaces for Advanced Computing, Copyright © 1987 by *SCIENTIFIC AMERICAN, Inc.* All rights reserved.)

Considerable effort has been directed toward creating what are often called *artificial realites* or *virtual realities*; these are completely computer-generated environments with realistic appearance, behavior, and interaction techniques [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 I.16 shows this combination of equipment.

Several other technologies can be used to record 3D positions. The sonic-tablet technology discussed in Section 4.6.1 can be extended to 3D to create a *sonic pen*. In one approach, three orthogonal strip microphones are used. The hand-held pen sparks 20 to 40 times per second, and the time for the sound to arrive at each of the three microphones determines the radius of the three cylinders on which the pen is located. The location is thus computed as the intersection of three cylinders. A similar approach uses three or four standard microphones; here, the pen location is computed as the intersection of spheres with centers at the microphones and radii determined by the time the sound takes to arrive at each microphone.

All these systems work in relatively small volumes—8 to 27 cubic feet. Optical sensors can give even greater freedom of movement [BURT74; FUCH77a]. 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 I.17. This technique could be extended to 3D through use of multiple cameras.

8.1.7 Device-Level Human Factors

Not all interaction devices of the same type are equivalent from a human-factors point of view (see [BUXT86] for an elaboration of this theme). For instance, mice differ in important ways. First, the physical shapes are different, ranging from a hemisphere to an elongated, low-profile rectangle. Buttons are positioned differently. Buttons on the side or front of a mouse may cause the mouse to move a bit when the buttons are pressed; buttons on the top of a mouse do not have this effect. The mouse is moved through small distances by wrist and finger movements, with the fingers grasping the mouse toward its front. Yet the part of the mouse whose position is sensed is often toward the rear, where fine control is least possible. In fact, a small leftward movement of the mouse under the fingertips can include a bit of rotation, so that the rear of the mouse, where the position sensors are, actually moves a bit to the right!

There is great variation among keyboards in design parameters, such as keycap shape, distance between keys, pressure needed to press a key, travel distance for key depression, key bounce, auditory feedback, the feeling of contact when the key is fully depressed, and the placement and size of important keys such as "return" or "enter." Improper choice of parameters can decrease productivity and increase error rates. Making the "return" key too small invites errors, as does placing a hardware "reset" key close to other keys. These and other design parameters are discussed in [KLEM71; GREE87], and have been the subject of recent international standardization efforts.

The tip of a short joystick shaft moves through a short distance, forcing use of a small C/D ratio; if we try to compensate by using a longer joystick shaft, the user cannot rest the heel of her hand on the work surface and thus does not have a steady platform from which to make fine adjustments. Accuracy and speed therefore suffer.

The implication of these device differences is that it is not enough for a user interface designer to specify a particular device class; specific device characteristics must be defined. Unfortunately, not every user interface designer has the luxury of selecting devices; often, the choice has already been made. Then the designer can only hope that the devices are well designed, and attempt to compensate in software for any hardware deficiencies.

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 the next section, 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 postioning.

A number of general issues transcend any one interaction technique. We first discuss the general issues; we introduce specific positioning techniques as illustrations.

Coordinate systems. An important issue in positioning is the coordinate system in which feedback is provided. If a locator device is moved to the right to drag an object, in which direction should the object move? There are at least three possibilities: the object could move along the increasing x direction in the screen-coordinate system, along the increasing x direction in world coordinates, or along the increasing x direction in the object's own coordinate system.

The first alternative, increasing screen-coordinate x direction, is the correct choice. For the latter two options, consider that the increasing x direction need not in general be along the screen coordinates' x axis. For instance, if the viewing transformation includes a 180° rotation, then the world coordinates' x axis goes in the opposite direction to the screen coordinates' x axis, so that the right-going movement of the locator would cause a left-going movement of the object. Try positioning with this type of feedback by turning your mouse 180°! Such a system would be a gross violation of the human-factors principle of stimulus-response compatibility (S-R compatibility), 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. Similar problems can occur if a data tablet is rotated with respect to the screen.

Resolution. The resolution required in a positioning task may vary from one part in a few hundred to one part in millions. Clearly, keyboard typein of an (x, y) pair can provide unlimited resolution: The typed digit strings can be as long as necessary. What resolution can cursor-movement techniques achieve? The resolution of tablets, mice, and so on is typically as least as great as the 500 to 2000 resolvable units of the display device. By using the window-to-viewport transformation to zoom in on part of the world, it is possible to arrange for one unit of screen resolution to correspond to an arbitrarily small unit of world-coordinate resolution.

Touch panels present other interesting resolution issues. Some panels are accurate to 1000 units. But the user's finger is about $\frac{1}{2}$-inch wide, so how can this accuracy be achieved? Using the first position the finger touches as the final position does not work. The user must be able to drag a cursor around on the screen by moving or rolling his finger while it is in contact with the touch panel. Because the finger obscures the exact position being indicated, the cursor arms can be made longer than normal, or the cursor can be offset from the actual point of contact. In an experiment, dragging an offset cursor was found to be more accurate, albeit slower, than was using the first point contacted [POTT88]. In general, the touch panel is not recommended for frequent high-resolution positioning tasks.

Grids. An important visual aid in many positioning tasks is a grid superimposed (perhaps at low intensity) on the work area, to help in aligning positions or objects. It can also be useful to force endpoints of primitives to fall on the grid, as though each grid point were surrounded by a gravity field. Gridding helps users to generate drawings with a neat appearance. To enforce gridding, the application program simply rounds locator coordinates to the nearest grid point (in some cases, only if the point is already close to a grid point). Gridding is usually applied in world coordinates. Although grids often are regular and span the entire display, irregular grids, different grids in different areas, as well as rotated grids, are all useful in creating figures and illustrations [BIER86a; FEIN82a].

Feedback. 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 8.4.

Direction preference. Some positioning devices impede movement in arbitrary directions; for example, certain joysticks and joyswitches give more resistance to movements off the principal axes than they do to those on the axes. This is useful only if the positioning task itself is generally constrained to horizontal and vertical movements.

Learning time. Learning the eye–hand coordination for indirect methods is essentially the same process as learning to steer a car. Learning time is a common concern but turns out to be a minor issue. Card and colleagues [CARD78] studied the mouse and joystick. They found that, although practice improved both error rates and speed, even the novices' performance was quite good. For instance, selection time with a mouse (move cursor to target, press button) decreased with extensive practice from 2.2 to 1.7 seconds. It is true, however, that some users find the indirect coordination very difficult, until they are explicitly taught.

One specific type of postioning task is *continuous positioning,* in which a sequence of positions is used to define a curve. The path taken by the locator is approximated by a connected series of very short lines, as shown in Fig. 8.5. So that the appearance of

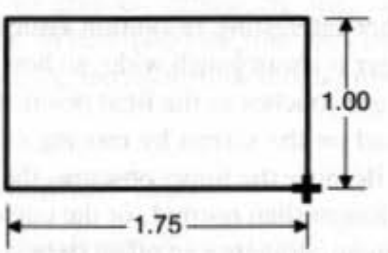


Fig. 8.4 Numeric feedback regarding size of an object being constructed. The height and width are changed as the cursor (+) is moved, so the user can adjust the object to the desired size.

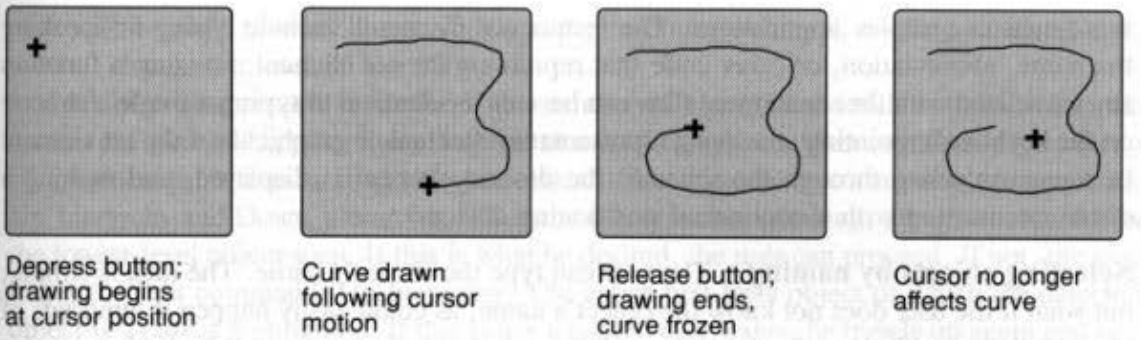


Fig. 8.5 Continuous sketching.

smoothness is maintained, more lines may be used where the radius of curvature is small, or individual dots may be displayed on the cursor's path, or a higher-order curve can be fitted through the points (see Chapter 11).

Precise continuous positioning is easier with a stylus than with a mouse, because the stylus can be controlled precisely with finger muscles, whereas the mouse is controlled primarily with wrist muscles. Digitizing of drawings is difficult with a mouse for the same reason; in addition, the mouse lacks both an absolute frame of reference and a cross-hair. On the other hand, a mouse requires only a small table area and is less expensive than a tablet.

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.

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 "relatively" modifier 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 the following section, 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 can help the user to remember just how the name is spelled, if he 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 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 he 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 the user wants, he travels up again and still more of the picture is highlighted. If he travels too far up the hierarchy, he reverses direction 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. 8.6

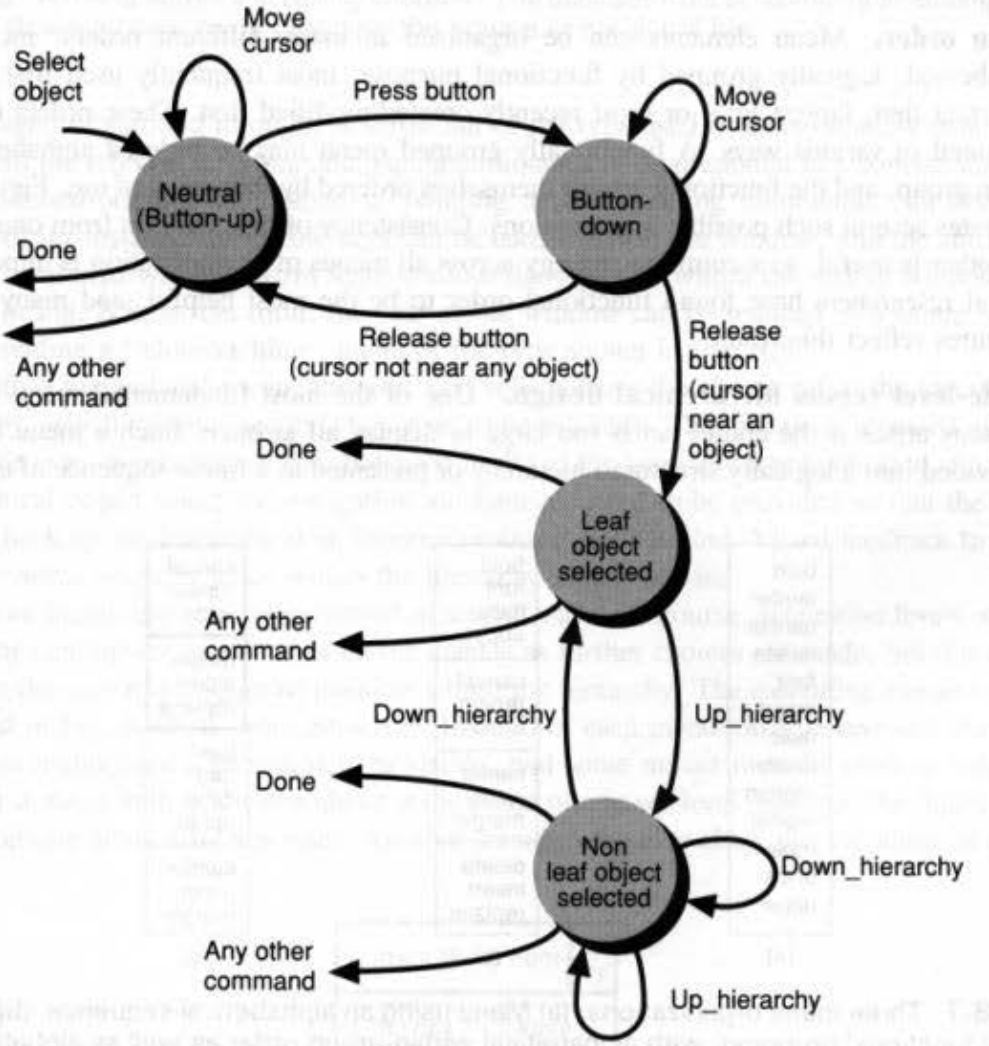


Fig. 8.6 State diagram for an object-selection technique for an arbitrary number of hierarchy levels. Up and Down are commands for moving up and down the hierarchy. In the state "Leaf object selected," the Down_hierarchy command is not available. The user selects an object by pointing at it with a cursor, and pressing and then releasing a button.

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.

Some text editors use a character–word–sentence–paragraph hierarchy. In the Xerox Star text editor, for instance, the user selects a character by positioning the screen cursor on the character and clicking the Select button on the mouse. To choose the word rather than the character, the user clicks twice in rapid succession. Further moves up the hierarchy are accomplished by additional rapid clicks.

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.

Menu order. Menu elements can be organized in many different orders, including alphabetical, logically grouped by functional purpose, most frequently used first, most important first, largest first, or most recently created/modified first. These orders can be combined in various ways. A functionally grouped menu may be ordered alphabetically within group, and the functional groups themselves ordered by frequency of use. Figure 8.7 illustrates several such possible organizations. Consistency of organization from one menu to another is useful, so a common strategy across all menus of an application is important. Several researchers have found functional order to be the most helpful, and many menu structures reflect this result.

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

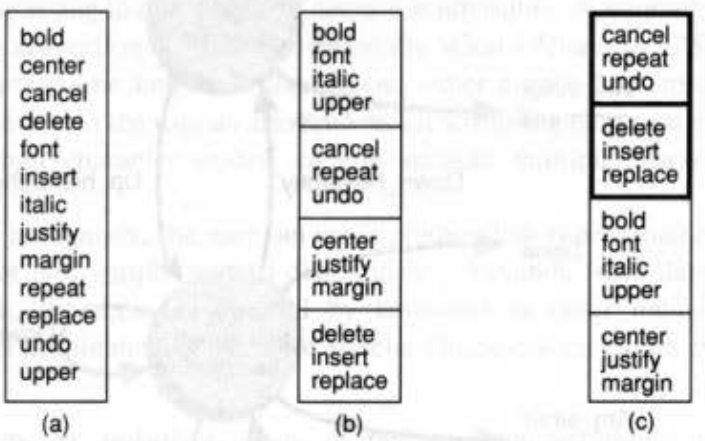


Fig. 8.7 Three menu organizations. (a) Menu using an alphabetical sequence. (b) Menu using functional grouping, with alphabetical within-group order as well as alphabetical-between-group order. (c) Menu with commands common to several different application programs placed at the top for consistency with the other application's menus; these commands have heavier borders. Menu items are some of those used in Card's menu-order experiment [CARD82].

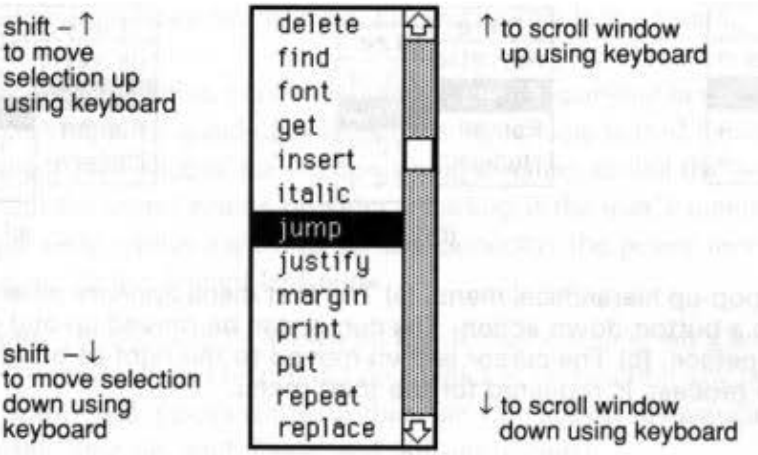


Fig. 8.8 A menu within a scrolling window. The user controls scrolling by selecting the up and down arrows or by dragging the square in the scroll bar.

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. 8.8. In the limit, the size of the window can be reduced to a single menu item, yielding a "slot-machine" menu of the type shown in Fig. 8.9.

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. 8.10, 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

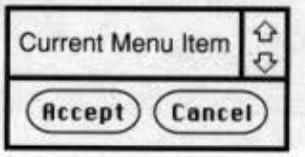


Fig. 8.9 A small menu-selection window. Only one menu item appears at a time. The scroll arrows are used to change the current menu item, which is selected when the Accept button is chosen.

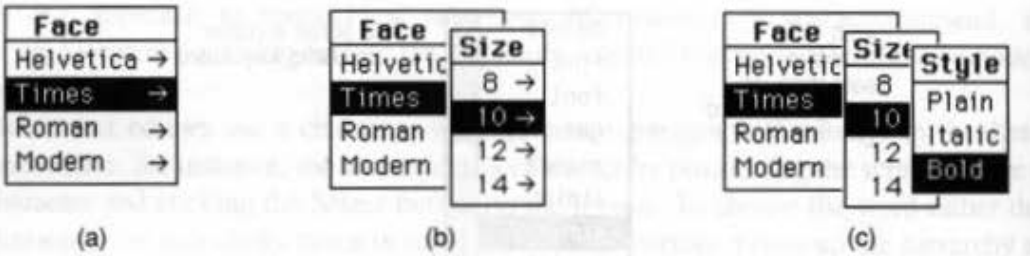


Fig. 8.10 A pop-up hierarchical menu. (a) The first menu appears where the cursor is, in response to a button-down action. The cursor can be moved up and down to select the desired typeface. (b) The cursor is then moved to the right to bring up the second menu. (c) The process is repeated for the third menu.

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

A *panel hierarchy* is another way to depict a hierarchy, as shown in Fig. 8.11; it takes up somewhat more room than the cascading hierarchy. If the hierarchy is not too large, an explicit tree showing the entire hierarchy can also be displayed.

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

Fig. 8.11 A hierarchical-selection menu. The leftmost column represents the top level; the children of the selected item in this column are shown in the next column; and so on. If there is no selected item, then the columns to the right are blank. (Courtesy of NeXT, Inc. © 1989 NeXT, Inc.)

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 if the user's memory fails. 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 can be shown on the display screen or on a second, auxiliary screen (Fig. 8.12); they can also be printed on a tablet or on function-key labels. Onscreen menus can be static and permanently visible, or can appear dynamically on request (tear-off, appearing, pop-up, pull-down, and pull-out menus).

A static menu printed on a tablet, as shown in Color Plate I.18, can easily be used in fixed-application systems. Use of a tablet or an auxiliary screen, however, requires that the user look away from the application display, and hence destroys visual continuity. The advantages are the saving of display space, which is often at a premium, and the accommodation of a large set of commands in one menu.

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

Fig. 8.12 A dual-display workstation. The two displays can be used to show the overview of a drawing on one and detail on the other, or to show the drawing on one and menus on the other. (Courtesy of Intergraph Corporation, Huntsville, Al.)

maintaining visual continuity. An attractive feature in pop-up menus is to highlight initially 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. Alternatively, if the menu is ordered by frequency of use, the most frequently used command can be highlighted initially and should also be in the middle (not at the top) of the menu, to minimize cursor movements in selecting other items.

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 Chapters 2 and 19.

Pop-up menus often can be context-sensitive. In several window-manager systems, if the cursor is in the window banner (the top heading of the window), commands involving window manipulation appear in the menu; if the cursor is in the window proper, commands concerning the application itself appear (which commands appear can depend on the type of object under the cursor); otherwise, commands for creating new windows appear in the menu. This context-sensitivity may initially be confusing to the novice, but is powerful once understood.

Unlike pop-up menus, pull-down and pull-out menus are anchored in a menu bar along an edge of the screen. The Apple Macintosh, Microsoft Windows, and Microsoft Presentation Manager all use pull-down menus. Macintosh menus, shown in Fig 8.13, also illustrate accelerator keys and context sensitivity. Pull-out menus, an alternative to pull-down menus, are shown in Fig. 8.14. Both types of menus have a two-level hierarchy: The menu bar is the first level, and the pull-down or pull-out menu is the second. Pull-down and pull-out menus can be activated explicitly or implicitly. In explicit activation, a button depression, once the cursor is in the menu bar, makes the second-level menu appear; the

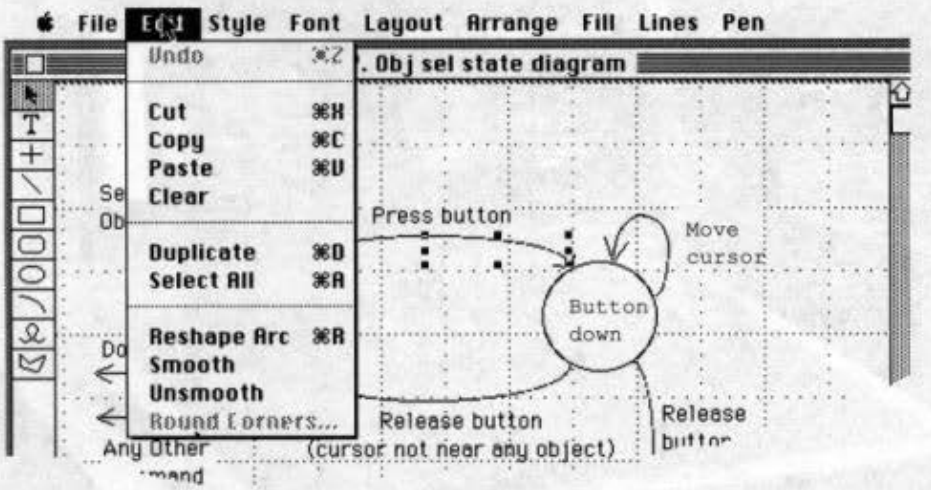


Fig. 8.13 A Macintosh pull-down menu. The last menu item is gray rather than black, indicating that it is currently not available for selection (the currently selected object, an arc, does not have corners to be rounded). The Undo command is also gray, because the previously executed command cannot be undone. Abbreviations are accelerator keys for power users. (Copyright 1988 Claris Corporation. All rights reserved.)

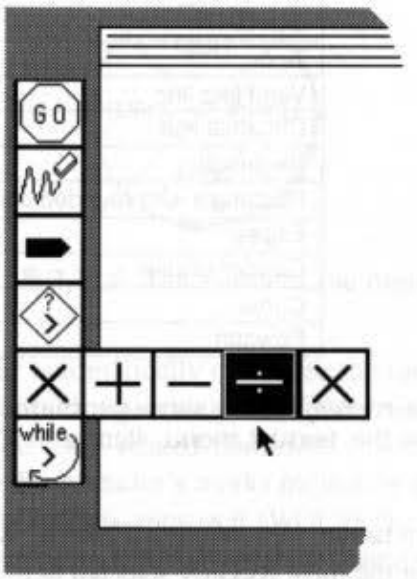


Fig. 8.14 A pull-out menu in which the leftmost, permanently displayed element shows the current selection. The newly selected menu item (reversed background) will become the current selection. This contrasts with most menu styles, in which the name of the menu is permanently displayed and the current selection is not shown after the adapted menu is dismissed. Menu is adapted from Jovanović's Process Visualization System [JOVA86]. (Courtesy of Branka Jovanović.)

cursor is moved on top of the desired selection and the button is then released. In implicit activation, moving the cursor into the heading causes the menu to appear; no button press is needed. Either selecting an entry or moving the cursor out of the menu area dismisses the menu. These menus, sometimes called "lazy" or "drop-down" menus, may also confuse new users by their seemingly mysterious appearance.

A full-screen menu can be a good or bad solution, depending on the context within which it is used. The disadvantage is that the application drawing will be obscured, removing context that might help the user to make an appropriate choice. Even this concern can be removed by using a raster display's look-up table to show the menu in a strong, bright color, over a dimmed application drawing [FEIN82a].

Visual representation. The basic decision on representation is whether menus use textual names or iconic or other graphical representations of elements of the choice set. Full discussion of this topic is deferred to the next chapter; however, note that iconic menus can be spatially organized in more flexible ways than can textual menus, because icons need not be long and thin like text strings; see Fig. 8.15. Also, inherently graphical concepts (particularly graphical attributes and geometrical primitives) are easily depicted.

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

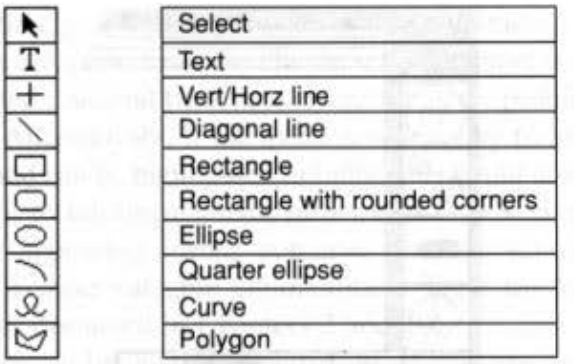


Fig. 8.15 Iconic and textual menus for the same geometric primitives. The iconic menu takes less space than does the textual menu. (Icons © 1988 Claris Corporation. All rights reserved.)

technique, patterned after the tuning buttons on car radios, is one way (Fig. 8.16). 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 she is 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.

Pop-up *pie menus* [CALL88], shown in Fig. 8.17, appear at the cursor. As the user moves the mouse from the center of the pie toward the desired selection, the target width becomes larger, decreasing the likelihood of error. Thus, the user has explicit control over the speed-versus-error tradeoff. In addition, the distance to each menu item is the same.

Pattern recognition. In selection techniques involving pattern recognition, the user makes sequences of movements with a continuous-positioning device, such as a tablet or

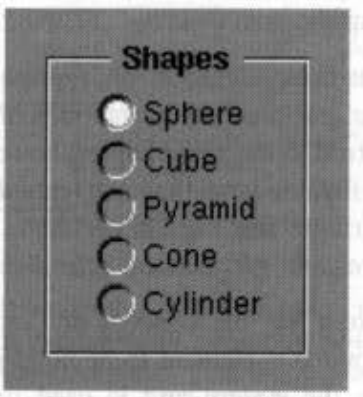


Fig. 8.16 Radio-button technique for selecting from a set of mutually exclusive alternatives. (Courtesy of NeXT, Inc. © 1989 NeXT, Inc.)

Fig. 8.17 A four-element pie menu.

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. Figure 8.18 shows one set of sketch patterns and their related commands, taken from Wallace's SELMA queueing analyzer [IRAN71]. Proofreader's marks indicating delete, capitalize, move, and so on are attractive candidates for this approach [WOLF87].

The technique requires no typing skill and preserves tactile continuity. Furthermore, if the command involves an object, the cursor position can be used for selection. The move command used in many Macintosh applications is a good example: the cursor is positioned on top of the object to be moved and the mouse button is pressed, selecting the object under

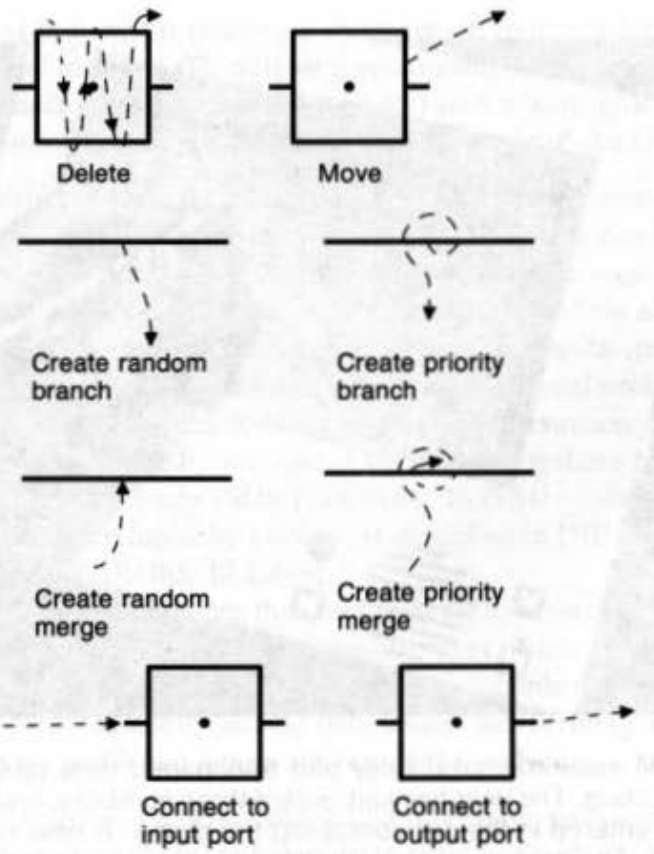


Fig. 8.18 Motions, indicated as dotted lines, that are recognized as commands. From Wallace's SELMA queuing analyzer [IRAN71].

the cursor (it is displayed in reverse video for feedback). As the user moves the mouse (still holding down the button), the object moves also. Releasing the mouse button detaches the object from the mouse. Skilled operators can work very rapidly with this technique, because hand movements between the work area and a command-entry device are eliminated. Given a data tablet and stylus, this technique can be used with at least several dozen patterns, but it is difficult for the user to learn a large number of different patterns.

Rhyne has recently combined a transparent tablet and liquid-crystal display into a prototype of a portable, lap-top computer [RHYN87]. Patterns are entered on the transparent tablet, and are recognized and interpreted as commands, numbers, and letters. The position at which information is entered is also significant. Figure 8.19 shows the device in use with a spreadsheet application.

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. Learning exotic key combinations, such as "shift-option-control-L," for some commands is not easy, however, and is left as an exercise for the regular user seeking the productivity gains that typically result. Putting a "cheat-sheet" template on the keyboard to remind users of

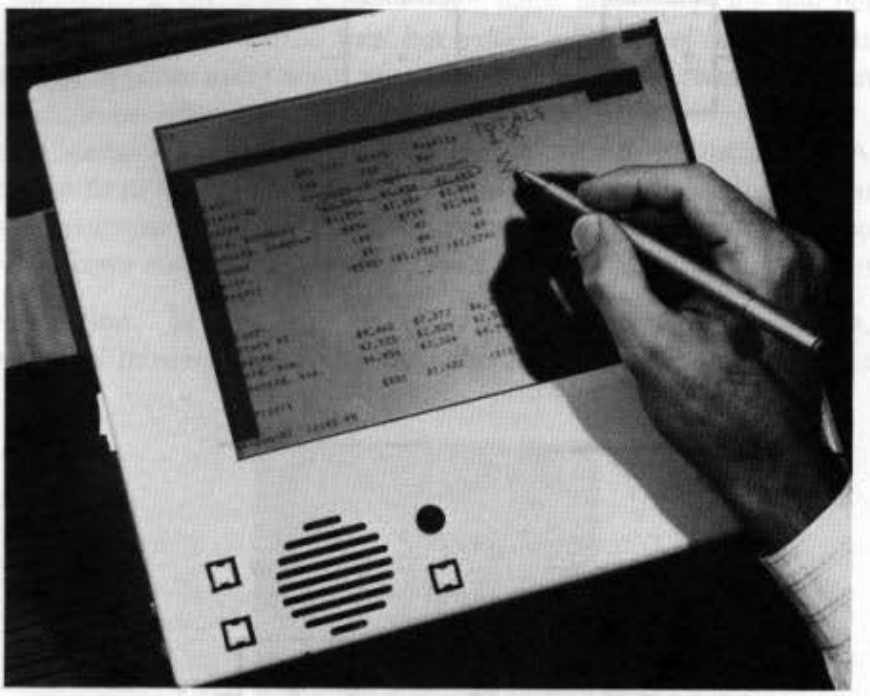


Fig. 8.19 An IBM experimental display and transparent data tablet. A spreadsheet application is executing. The user has just circled three numbers, and has indicated that their sum is to be entered in the cell containing the sigma. A new column heading has also been printed. The system recognizes the gestures and characters, and enters appropriate commands to the spreadsheet application. (Courtesy IBM T. J. Watson Research Laboratories.)

these obscure combinations can speed up the learning process. It is even sometimes possible to define chords that make some sense, decreasing learning time. 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.

One way to compensate for the lack of multiple buttons on a mouse is to use the temporal dimension to expand the possible meanings of one button—for instance, by distinguishing between a single click and two clicks made in rapid succession. If the meaning of two clicks is logically related to that of one click, this technique can be especially effective; otherwise, rote memory is needed to remember what one and two clicks mean. Examples of this technique are common: one click on a file icon selects it; two clicks opens the file. One click on the erase command enters erase mode; two clicks erase the entire screen. This technique can also be applied to each of the buttons on a multibutton mouse. Chording of mouse buttons or of keyboard keys with mouse buttons can also be used to provide the logical (but not necessarily human-factors) equivalent of more buttons. To be most useful, the organizing scheme for the chording patterns must be logical and easy to remember.

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.

Character recognition. The user prints characters with a continuous-positioning device, usually a tablet stylus, and the computer recognizes them. This is considerably easier than recognizing scanned-in characters, because the tablet records the sequence, direction, and sometimes speed and pressure of strokes, and a pattern-recognition algorithm can match these to stored templates for each character. For instance, the capital letter "A" consists of three strokes—typically, two downward strokes and one horizontal stroke. A recognizer can be trained to identify different styles of block printing: the parameters of each character are calculated from samples drawn by the user. Character recognizers have been used with interactive graphics since the early 1960s [BROW64; TEIT64]. A simplified adaptation of Teitelman's recognizer, developed by Ledeen, is described in [NEWM73]; a commercial system is described in [WARD85; BLES86].

It is difficult to block print more than one or two characters per second (try it!), so character recognition is not appropriate for massive input of text. We write cursive letters faster than we print the same characters, but there are no simple recognition algorithms for cursive letters: the great variability among individuals' handwrtiting and the difficulty of segmenting words into individual letters are two of the problems.

Menu selection. A series of letters, syllables, or other basic units is displayed as a menu. The user then inputs text by choosing letters from the menu with a selection device. This technique is attractive in several situations. First, if only a short character string is to be entered and the user's hands are already on a pointing device, then menu selection may be

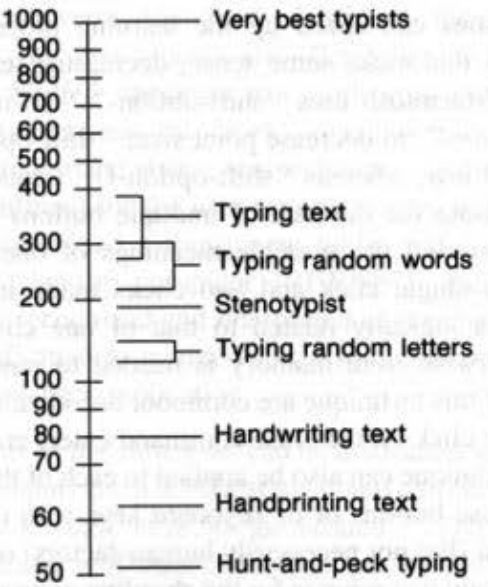


Fig. 8.20 Data-input speeds, in keystrokes per minute, of various techniques for entering text and numeric information. (Adapted from [VANC72, p. 335] and [CARD83, p. 61].)

faster than moving to the keyboard and back. Second, if the character set is large, this approach is a reasonable alternative to the keyboard.

Hierarchical menu selection can also be used with large character sets, such as are used in Chinese and Japanese. One such system uses the graphical features (strong horizontal line, strong vertical line, etc.) of the symbols for the hierarchy. A more common strategy is to enter the word in phonetic spelling, which string is then matched in a dictionary. For example, the Japanese use two alphabets, the katakana and hiragana, to type phonetically the thousands of kanji characters that their orthography borrows from the Chinese.

Evaluation of text-entry techniques. For massive input of text, the only reasonable substitute for a skilled typist working with a keyboard is an automatic scanner. Figure 8.20 shows experimentally determined keying rates for a variety of techniques. The hunt-and-peck typist is slowed by the perceptual task of finding a key and the ensuing motor task of moving to and striking it, but the trained typist has only the motor task of striking the key, preceded sometimes by a slight hand or finger movement to reach it. Speech input, not shown on the chart, is slow but attractive for applications where the hands must be free for other purposes, such as handling paperwork.

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; unfortunately, most graphics system manufacturers offer only rotary potentiometers. On the other hand, rotary potentiometers are easier to adjust. Availability of both linear and rotary potentiometers can help users to associate meanings with each device. It is important to use directions consistently: clockwise or upward movements normally increase a value.

With continuous-scale manipulation, the user points at the current-value indicator on a displayed gauge or scale, presses the selection button, drags the indicator along the scale to the desired value, and then releases the selection button. A pointer is typically used to indicate the value selected on the scale, and a numeric echo may be given. Figure 8.21 shows several such dials and their associated feedback. The range or precision of values entered in this way can be extended by using the positioning device as a relative rather than absolute device and by using a nonconstant C/D ratio, as discussed in Section 8.1.1. Then it becomes possible to increase a value by repeating a series of stroking actions: move to the right, lift mouse, move to the left, put mouse down, and so on. Thornton's number wheel [THOR79] is such a technique.

If the resolution needed is higher than the continuous-scale manipulation technique can provide, or if screen space is at a premium, an up–down counter arrangement can be used, as shown in Fig. 8.22.

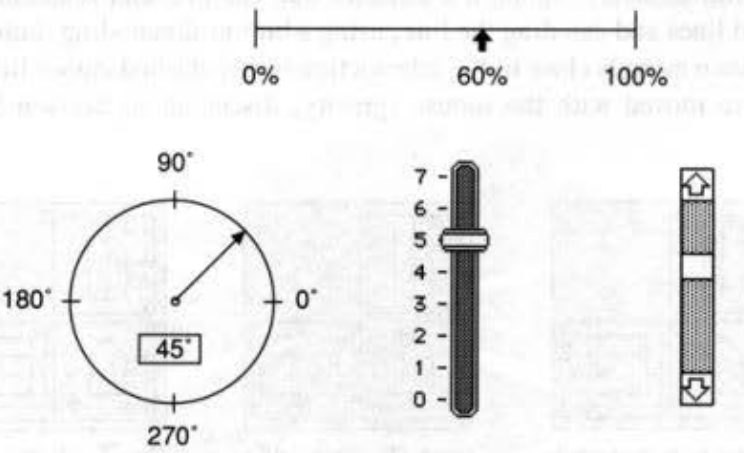


Fig. 8.21 Several dials that the user can use to input values by dragging the control pointer. Feedback is given by the pointer and, in two cases, by numeric displays. (Vertical sliders © Apple Computer, Inc.)

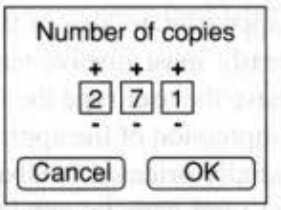


Fig. 8.22 An up–down counter for specifying a value. The user positions the cursor on "+" or "−" and holds down the selection button on the pointing device; the corresponding digit increases or decreases until the button is released.

8.2.6 3D Interaction Tasks

Two of the four interaction tasks described previously for 2D applications become more complicated in 3D: position and select. In this section, we also introduce an additional 3D interaction task: rotate (in the sense of orienting an object in 3-space). The major reason for the complication is the difficulty of perceiving 3D depth relationships of a cursor or object relative to other displayed objects. This contrasts starkly with 2D interaction, where the user can readily perceive that the cursor is above, next to, or on an object. A secondary complication arises because the commonly available interaction devices, such as mice and tablets, are only 2D devices, and we need a way to map movements of these 2D devices into 3D.

Display of stereo pairs, corresponding to left- and right-eye views, is helpful for understanding general depth relationships, but is of limited accuracy as a precise locating method. Methods for presenting stereo pairs to the eye are discussed in Chapters 14 and 18, and in [HODG85]. Other ways to show depth relationships are discussed in Chapters 14–16.

The first part of this section deals with techniques for positioning and selecting, which are closely related. The second part concerns techniques for interactive rotation.

Figure 8.23 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 (gravity, discussed in Section 8.3.2, can make

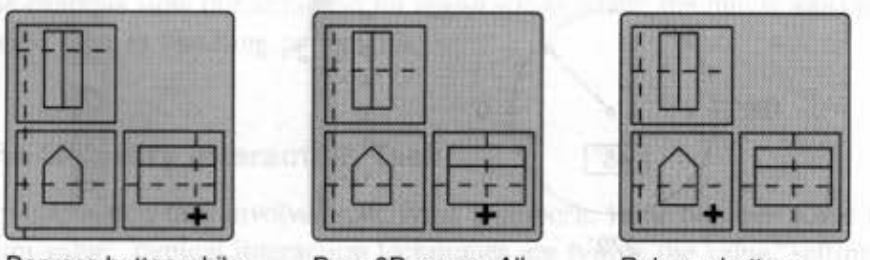


Fig. 8.23 3D positioning technique using three views of the same scene (a house). The 2D cursor (+) is used to select one of the dashed 3D cursor lines.

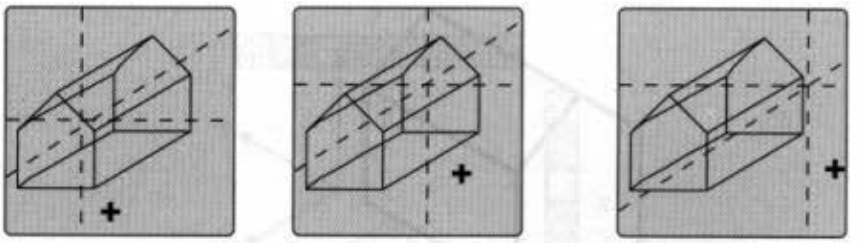


Fig. 8.24 Movement of the 3D cursor is controlled by the direction in which the 2D cursor is moved.

picking the intersection especially easy). 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.

Another possibility, developed by Nielson and Olsen [NIEL86] and depicted in Fig. 8.24, requires that all three principal axes project with nonzero length. A 3D cross-hair cursor, with cross-hairs parallel to the principal axes, is controlled by moving the mouse in the general direction of the projections of the three principal axes. Figure 8.25 shows how 2D locator movements are mapped into 3D: there are 2D zones in which mouse movements affect a specific axis. Of course, 3D movement is restricted to one axis at a time.

Both of these techniques illustrate ways to map 2D locator movements into 3D movements. We can instead use buttons to control which of the 3D coordinates are affected by the locator's 2 degrees of freedom. For example, the locator might normally control x and y; but, with the keyboard shift key depressed, it could control x and z instead (this requires that the keyboard be unencoded, as discused in Chapter 4). Alternatively, three buttons could be used, to bind the locator selectively to an axis. Instead of mapping from a

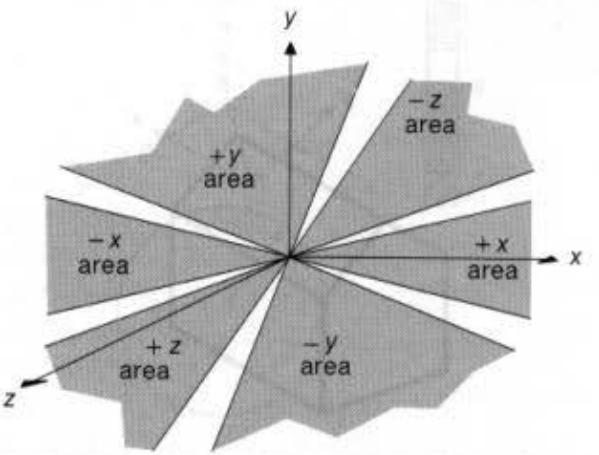


Fig. 8.25 The six regions of mouse movement, which cause the 3D cursor to move along the principal axes.

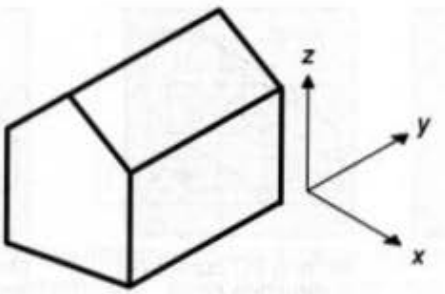


Fig. 8.26 The displayed local coordinate system of the house, which shows the three directions in which any translated object will move. To preserve stimulus-response compatibility, we can use the direction of mouse movements to determine the axes chosen, as in Fig. 8.25.

2D device into 3D, we could use a real 3D locator, such as the joysticks and trackballs discussed in Section 8.1.6

Constrained 3D movement is effective in 3D locating. Gridding and gravity can sometimes compensate for uncertainties in depth relationships and can aid exact placement. Another form of constraint is provided by those physical devices that make it easier to move along principal axes than in other directions. Some trackballs and joysticks have this property, which can also be simulated with the isometric strain-gauge and spaceball devices (Section 8.1.6).

Context-specific constraints are often more useful, however, than are these general constraints. It is possible to let the user specify that movements should be parallel to or on lines or planes other than the principal axes and planes. For example, with a method developed by Nielson and Olsen [NIEL86], the local coordinate system of the selected object defines the directions of movement as shown in Fig. 8.26. In a more general technique developed by Bier [BIER86b], the user places a coordinate system, called a *skitter*, on the surface of an object, again defining the possible directions of movement (Fig. 8.27). Another way to constrain movement to a particular plane is to give the user control over the view-plane orientation, and to limit translation to be parallel to the view plane.

One method of 3D picking—finding the output primitive that, for an (x, y) position

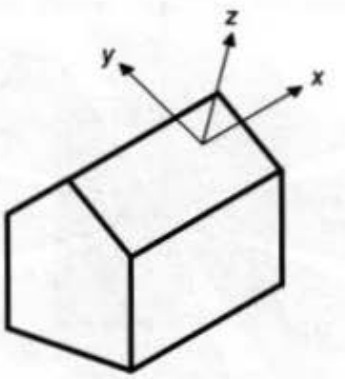


Fig. 8.27 The displayed coordinate system, placed interactively so that its (x, y) plane coincides with the plane of the roof, shows the three directions in which any translated object will move.

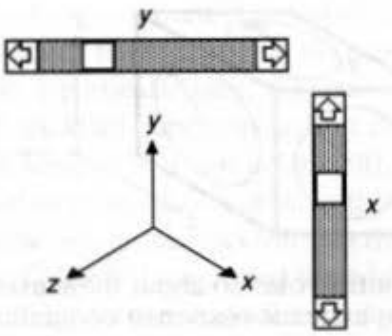


Fig. 8.28 Two slider dials for effecting rotation about the screen *x* and *y* axes.

determined by a 2D locator, has the maximum *z* value—was discussed in Chapter 7. Another method, which can be used with a 3D locator when wireframe views are shown—is to find the ouput primitive closest to the locator's (*x*, *y*, *z*) position.

As with locating and selection, the issues in 3D rotation are understanding depth relationships, mapping 2D interaction devices into 3D, and ensuring stimulus-response compatibility. 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 should normally 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. 8.28. The (*x*, *y*, *z*) coordinate system associated with the sliders is rotated as the sliders are moved to show the effect of the rotation. A 2D trackball can be used instead of the two sliders.

The two-axis rotation approach can be easily generalized to three axes by adding a dial for *z*-axis rotation, as in Fig. 8.29 (a dial is preferable to a slider for S-R compatibility).

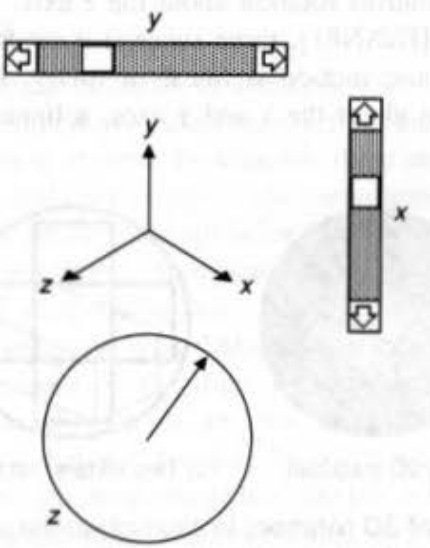


Fig. 8.29 Two slider dials for effecting rotation about the screen *x* and *y* axes, and a dial for rotation about the screen *z* axis. The coordinate system represents world coordinates and shows how world coordinates relate to screen coordinates.

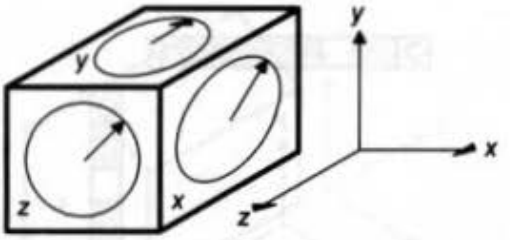
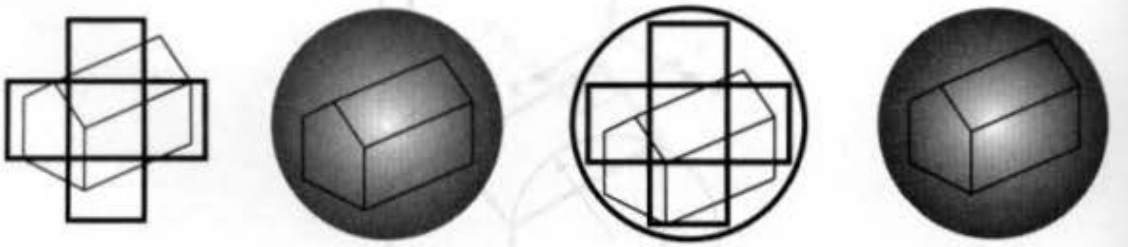


Fig. 8.30 Three dials to control rotation about three axes. The placement of the dials on the cube provides strong stimulus-response compatibility.

Even more S-R compatibility comes from the arrangement of dials on the faces of a cube shown in Fig. 8.30, which clearly suggests the axes controlled by each dial. Again, a 3D trackball could be used instead of the dials.

Mouse movements can be directly mapped onto object movements, without slider or dial intermediaries. The user can be presented a metaphor in which the two sliders of Fig. 8.28 are superimposed on top of the object being rotated, so that horizontal mouse movements are mapped into rotations about the screen-coordinate y axis, and vertical mouse movements are mapped into rotations about the screen-coordinate x axis (Fig. 8.31a). Diagonal motions have no effect. The slider dials are not really displayed; the user learns to imagine that they are present. Alternatively, the user can be told that an imaginary 2D trackball is superimposed on top of the object being rotated, so that the vertical, horizontal, or diagonal motions one would make with the trackball can be made instead with the mouse (Fig. 8.31b). Either of these methods provides two-axis rotation in 3D.

For three-axis rotations, three methods that closely resemble real-world concepts are particularly interesting. In the overlapping-sliders method [CHEN88], the user is shown two linear sliders overlapping a rotary slider, as in Fig. 8.31(c). Motions in the linear sliders control rotation about the x and y axes, while a rotary motion around the intersection of the two linear sliders controls rotation about the z axis. In a technique developed by Evans, Tanner, and Wein [EVAN81], three successive mouse positions are compared to determine whether the mouse motion is linear or rotary. Linear horizontal or vertical movements control rotation about the x and y axes, a linear diagonal movement rotates



(a) Two sliders (b) 2D trackball (c) Two sliders, rotary dial (d) 3D trackball

Fig. 8.31 Four methods of 3D rotation. In each case, the user makes movements with a 2D device corresponding to those that would be made if the actual devices were superimposed on the object. A 3D trackball can be twisted to give z-axis rotation, whereas a 2D trackball provides only two-axis rotation.

about both the x and y axes, and rotary movements control rotation about the z axis. While this is a relative technique and does not require that the movements be made directly over the object being rotated or in a particular area, the user can be instructed to use these motions to manipulate a 3D trackball superimposed on the object (Fig. 8.31d). In the virtual-sphere method, also developed by Chen [CHEN88], the user actually manipulates this superimposed 3D trackball in an absolute fashion as though it were real. With a mouse button down, mouse movements rotate the trackball exactly as your finger would move a real trackball. An experiment [CHEN88] comparing these latter two approaches showed no performance differences, but did yield a user preference for Chen's method.

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. An "apply" command is sometimes included in the box to cause the new values to be used

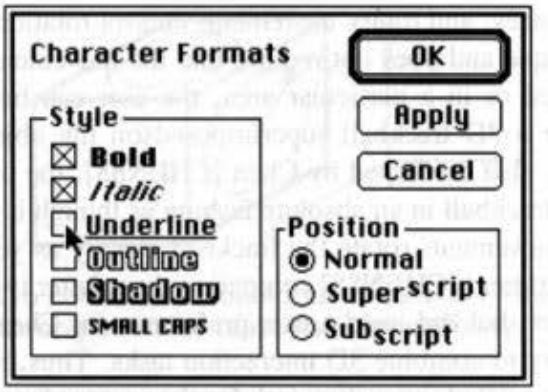


Fig. 8.32 A text-attribute dialogue box with several different attributes selected. The bold border of the "OK" button shows that the keyboard return key can be used as an alternative. The "Apply" button is used to apply new attibute values, so the user can observe their effects. "Cancel" is used to revert any changes to their previous values. Note that text attributes are described both by name and, graphically, by example. (Screen shots © 1983–1989 Microsoft Corporation. Reprinted with permission from Microsoft Corporation.)

without dismissing the box. More frequently, however, the dialogue box must be dismissed to apply the new settings. Figure 8.32 shows a dialogue box with several selected items highlighted.

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 8.33 show a rubberband line-drawing sequence. The user-action sequence is shown in the state diagram in Fig. 8.34. Notice that the state "rubberband" is active *only* while a button is held down. It is in this state that cursor movements cause the current line to change. See [BUXT85] for an informative discussion of the importance of matching the state transitions in an interaction technique with the transitions afforded by the device used with the technique.

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-

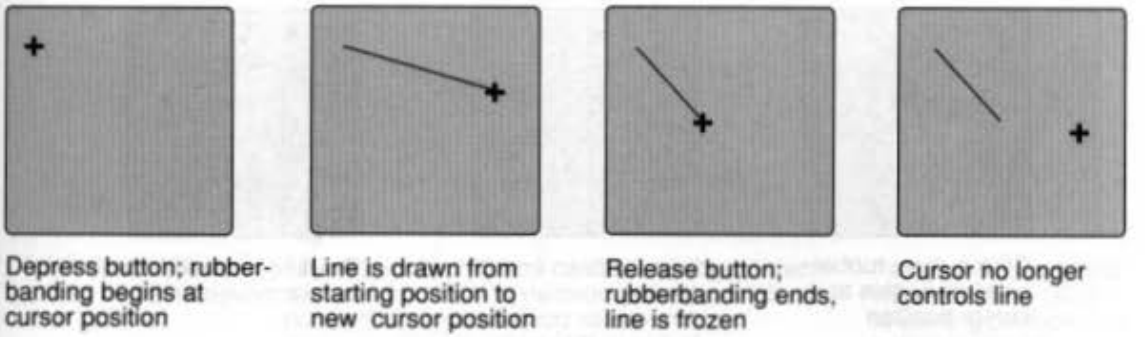


Fig. 8.33 Rubberband line drawing.

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. The *rubber-ellipse* technique creates an axis-aligned ellipse inside the rectangle defined by the initial and current cursor positions. A circle results if the rectangle is square—easily done with gridding. All these techniques have in common the user-action sequence of button-down, move locator and see feedback, button-up.

One interaction technique to create a polyline (a sequence of connected lines) is an extension of rubberbanding. After entering the polyline-creation command, the user clicks on a button to anchor each rubberbanded vertex. After all the vertices have been indicated, the user indicates completion, typically by a double click on a button without moving the locator, by a click on a second mouse button, or by entry of a new command. If the new command is from a menu, the last line segment of the polyline follows the cursor to the menu, and then disappears. Figure 8.35 depicts a typical sequence of events in creating a polyline; Fig. 8.36 is the accompanying state diagram.

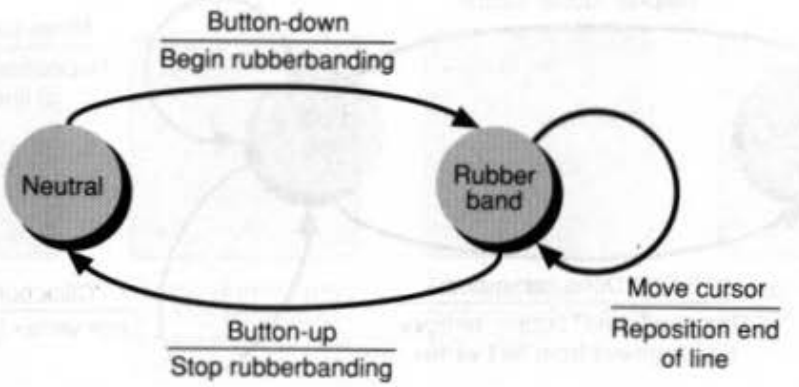


Fig. 8.34 State diagram for rubberband line drawing.

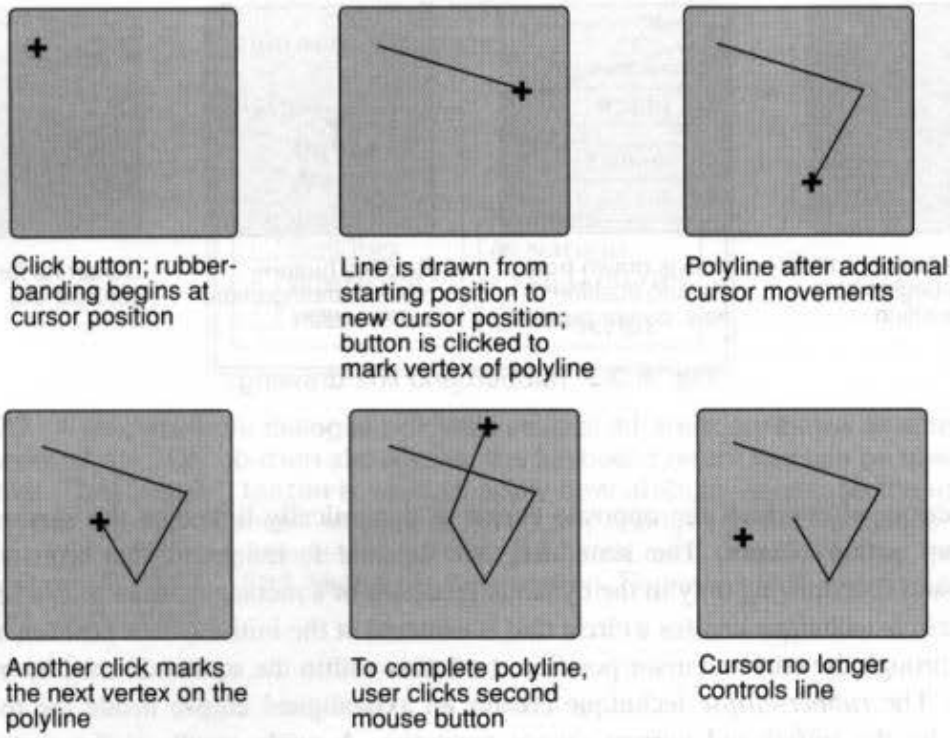


Fig. 8.35 Rubberband polyline sketching.

A polygon can be drawn similarly. In some cases, the user signals to the system that the polygon is complete by returning the cursor to the starting vertex of the polygon. In other cases, the user explicitly signals completion using a function key, and the system automatically closes the polygon. Figure 8.37 shows one way to create polygons.

Constraints of various types can be applied to the cursor positions in any of these techniques. For example, Fig. 8.38 shows a sequence of lines drawn using the same cursor

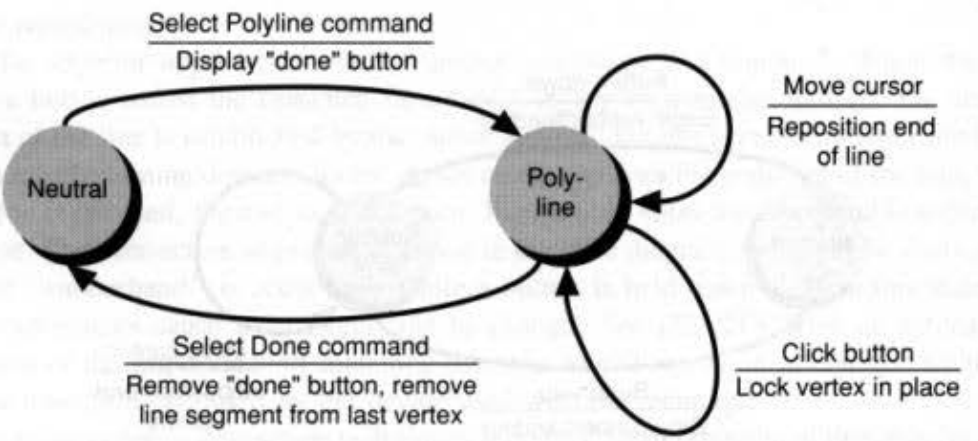


Fig. 8.36 State diagram for rubberband creation of a polyline.

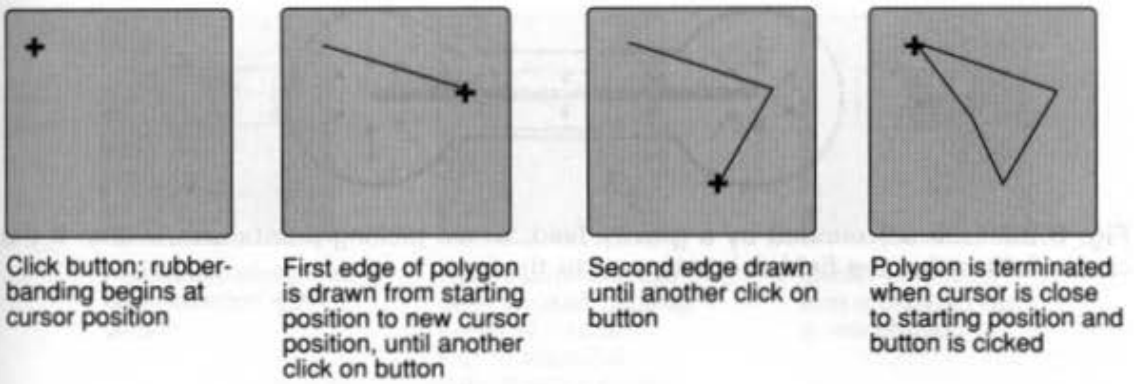


Fig. 8.37 Rubberband drawing of a polygon.

positions as in Fig. 8.33, 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.

Gravity is yet another form of constraint. When constructing drawings, we frequently want a new line to begin at the endpoint of, or on, an existing line. Matching an endpoint is easy if it was created using gridding, but otherwise is difficult without a potentially time-consuming zoom. The difficulty is avoided by programming an imaginary gravity field around each existing line, so that the cursor is attracted to the line as soon as it enters the gravity field. Figure 8.39 shows a line with a gravity field that is larger at the endpoints, so that matching endpoints is especially easy.

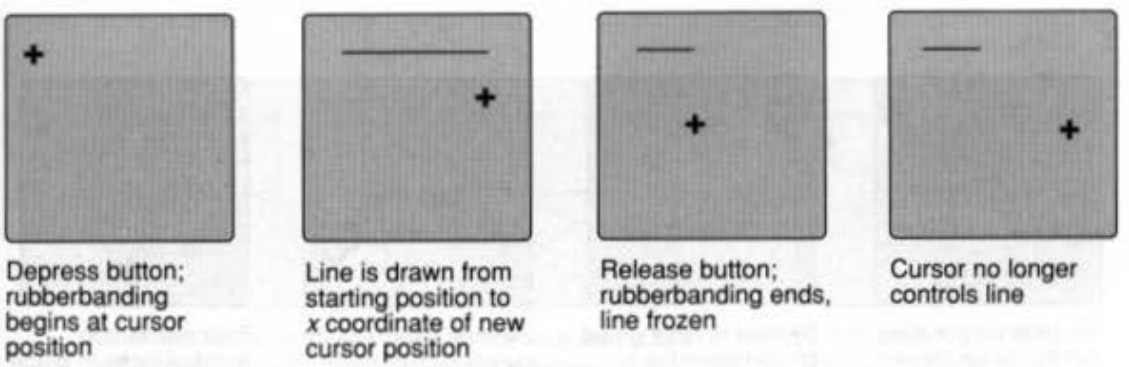


Fig. 8.38 Horizontally constrained rubberband line drawing.

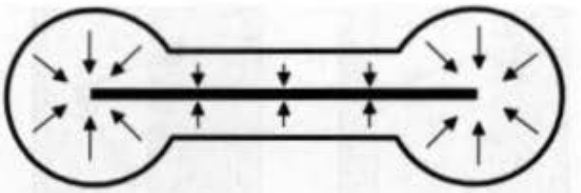


Fig. 8.39 Line surrounded by a gravity field, to aid picking points on the line: If the cursor falls within the field, it is snapped to the line.

8.3.3 Dynamic Manipulation

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

Dragging moves a selected symbol from one position to another under control of a cursor, as in Fig. 8.40. 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.

Dynamic rotation of an object can be done in a similar way, except that we must be able to identify the point or axis about which the rotation is to occur. A convenient strategy is to have the system show the current center of rotation and to allow the user to modify it as desired. Figure 8.41 shows one such scenario. Note that the same approach can be used for scaling, with the center of scaling, rather than that of rotation, being specified by the user.

The concept of *handles* is useful to provide scaling of an object, without making the user think explicitly about where the center of scaling is. Figure 8.42 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.

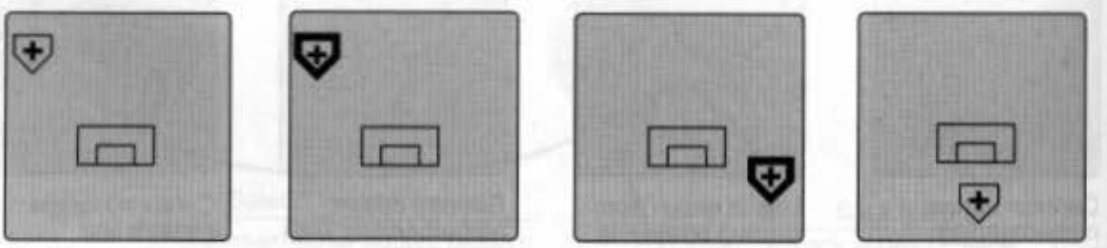


Fig. 8.40 Dragging a symbol into a new position.

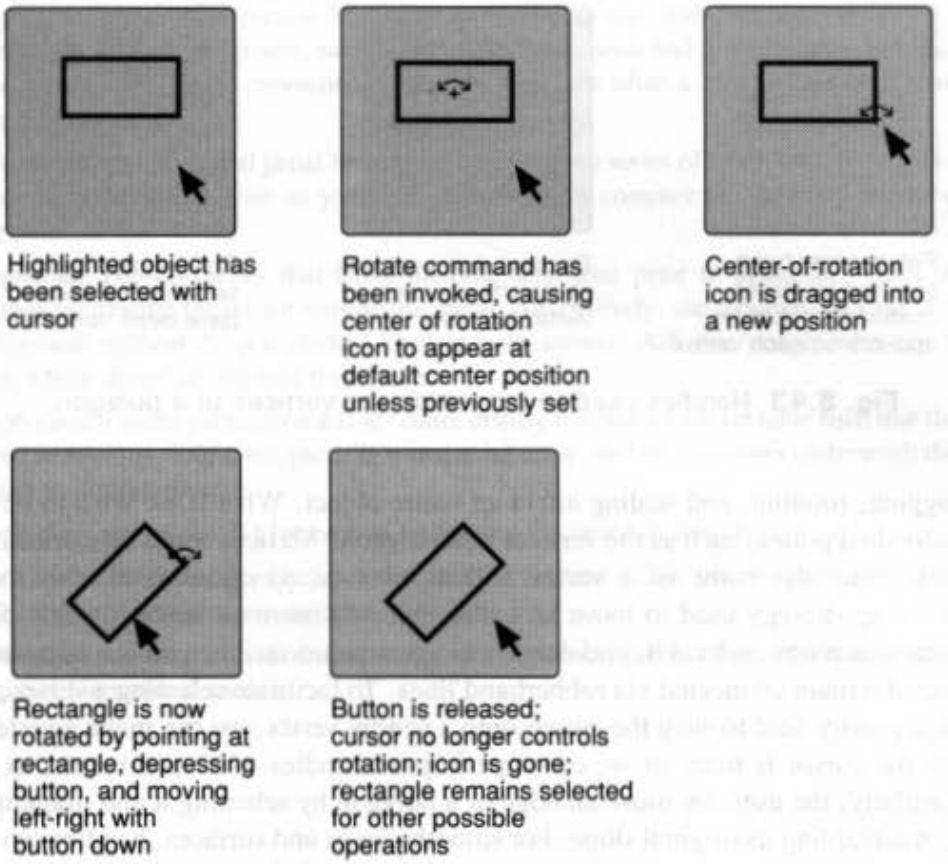


Fig. 8.41 Dynamic rotation.

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. (Blinking is another unique visual code, but tends to be distracting and annoying.)

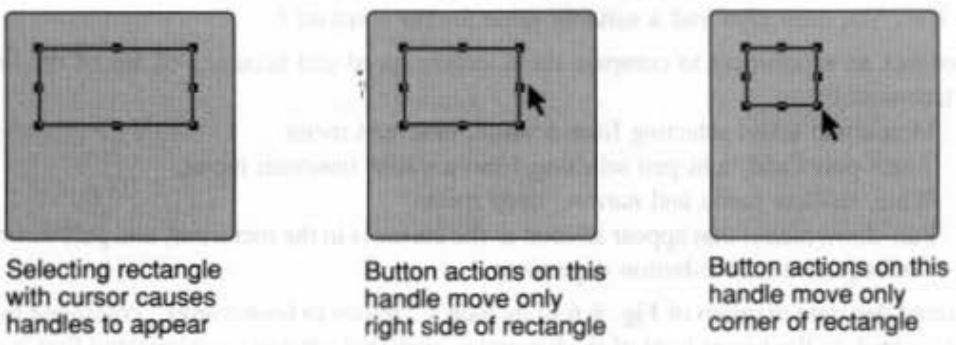


Fig. 8.42 Handles used to reshape objects.

Fig. 8.43 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 establish a gravity field to snap the cursor onto a nearby vertex, we can make a vertex blink whenever the cursor is near, or we can superimpose handles over each vertex, as in Fig. 8.43. 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 11.

In the next chapter, we discuss design issues involved in combining basic and composite interaction techniques into an overall user–computer dialogue.

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 Implement adaptive C/D ratio cursor tracking for use with a mouse or other relative-positioning device. Experiment with different relationships between mouse velocity v and the C/D ratio r: $r = k\,v$ and $r = k\,v^2$. You must also find a suitable value for the constant k.

8.3 Conduct an experiment to compare the selection speed and accuracy of any of the following pairs of techniques:

a. Mouse and tablet selecting from a static, onscreen menu
b. Touch-panel and light-pen selecting from a static, onscreen menu,
c. Wide, shallow menu and narrow, deep menu
d. Pull-down menus that appear as soon as the cursor is in the menu bar, and pull-down menus that require a mouse-button depression.

8.4 Extend the state diagram of Fig. 8.6 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.

8.5 Implement an autocompletion text-entry technique to use with an arbitrary list of words. Experiment with different word sets, such as the UNIX commands and proper names. Decide how to handle nonexistent matches, corrections typed by the user after a match has been made, and prompting for the user.

8.6 Implement cascading and panel heirarchical menus for a series of commands or for file-system subdirectories. What issues arise as you do this? Informally compare the selection speeds of each technique.

8.7 Implement pop-up menus that allow multiple selections prior to dismissal, which the user accomplishes by moving the cursor outside the menu. Alternatively, use a button click for dismissal. Which dismissal method do you prefer? Explain your answer. Ask five people who use the two techniques which dismissal method they prefer.

8.8 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.

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

8.10 For each of the locating techniques discussed in Section 8.2.6, identify the line or plane into which 2D locator movements are mapped.

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

9 Dialogue Design

We have described the fundamental building blocks from which the interface to an interactive graphics system is crafted—interaction devices, techniques, and tasks. Let us now consider how to assemble these building blocks into a usable and pleasing form. *User-interface design* is still at least partly an art, not a science, and thus some of what we offer is an attitude toward the design of interactive systems, and some specific dos and don'ts that, if applied creatively, can help to focus attention on the *human factors*, also called the *ergonomics*, of an interactive system.

The key goals in user-interface design are increase in speed of learning, and in speed of use, reduction of error rate, encouragement of rapid recall of how to use the interface, and increase in attractiveness to potential users and buyers.

Speed of learning concerns how long a new user takes to achieve a given proficiency with a system. It is especially important for systems that are to be used infrequently by any one individual: Users are generally unwilling to spend hours learning a system that they will use for just minutes a week!

Speed of use concerns how long an experienced user requires to perform some specific task with a system. It is critical when a person is to use a system repeatedly for a significant amount of time.

The *error rate* measures the number of user errors per interaction. The error rate affects both speed of learning and speed of use; if it is easy to make mistakes with the system, learning takes longer and speed of use is reduced because the user must correct any mistakes. However, error rate must be a separate design objective for applications in which even one error is unacceptable—for example, air-traffic control, nuclear-power-plant

control, and strategic military command and control systems. Such systems often trade off some speed of use for a lower error rate.

Rapid recall of how to use the system is another distinct design objective, since a user may be away from a system for weeks, and then return for casual or intensive use. The system should "come back" quickly to the user.

Attractiveness of the interface is a real marketplace concern. Of course, liking a system or a feature is not necessarily the same as being facile with it. In numerous experiments comparing two alternative designs, subjects state a strong preference for one design but indeed perform faster with the other.

It is sometimes said that systems cannot be both easy to learn and fast to use. Although there was certainly a time when this was often true, we have learned how to satisfy multiple design objectives. The simplest and most common approach to combining speed of use and ease of learning is to provide a "starter kit" of basic commands that are designed for the beginning user, but are only a subset of the overall command set. This starter kit is made available from menus, to facilitate ease of learning. All the commands, both starter and advanced, are available through the keyboard or function keys, to facilitate speed of use. Some advanced commands are sometimes put in the menus also, typically at lower levels of hierarchy, where they can be accessed by users who do not yet know their keyboard equivalents.

We should recognize that speed of learning is a relative term. A system with 10 commands is faster to learn than is one with 100 commands, in that users will be able to understand what each of the 10 commands does more quickly than they can what 100 do. But if the application for which the interface is designed requires rich functionality, the 10 commands may have to be used in creative and imaginative ways that are difficult to learn, whereas the 100 commands may map quite readily onto the needs of the application.

In the final analysis, meeting even one of these objectives is no mean task. There are unfortunately few absolutes in user-interface design. Appropriate choices depend on many different factors, including the design objectives, user characteristics, the environment of use, available hardware and software resources, and budgets. It is especially important that the user-interface designer's ego be submerged, so that the user's needs, not the designer's, are the driving factor. There is no room for a designer with quick, off-the-cuff answers. Good design requires careful consideration of many issues and patience in testing prototypes with real users.

9.1 THE FORM AND CONTENT OF USER–COMPUTER DIALOGUES

The concept of a *user–computer dialogue* is central to interactive system design, and there are helpful analogies between user–computer and person–person dialogues. After all, people have developed effective ways of communicating, and it makes sense to learn what we can from these years of experience. Dialogues typically involve gestures and words: In fact, people may have communicated with gestures, sounds, and images (cave pictures, Egyptian hieroglyphics) even before phonetic languages were developed. Computer graphics frees us from the limitations of purely verbal interactions with computers and enables us to use images as an additional communication modality.

Many attributes of person–person dialogues should be preserved in user–computer dialogues. People who communicate effectively share common knowledge and a common set of assumptions. So, too, there should be commonality between the user and the computer. Further, these assumptions and knowledge should be those of the user, not those of the computer-sophisticated user-interface designer. For instance, a biochemist studying the geometric structure of molecules is familiar with such concepts as atoms, bonds, dihedral angles, and residues, but does not know and should not have to know such concepts as linked lists, canvases, and event queues.

Learning to use a user interface is similar to learning to use a foreign language. Recall your own foreign-language study. Sentences came slowly, as you struggled with vocabulary and grammar. Later, as practice made the rules more familiar, you were able to concentrate on expanded vocabulary to communicate your thoughts more effectively. The new user of an interactive system must go through a similar learning process. Indeed, if new application concepts must be learned along with new grammar rules and vocabulary, the learning can be even more difficult. The designer's task, then, is to keep the user-interface rules and vocabulary simple, and to use concepts the user already knows or can learn easily.

The language of the user–computer dialogue should be efficient and complete, and should have natural sequencing rules. With an *efficient* language, the user can convey commands to the computer quickly and concisely. A *complete* language allows expression of any idea relevant to the domain of discourse. *Sequencing rules*, which define the order or syntax of the language, should have a minimum number of simple, easy-to-learn cases. Simple sequencing rules help to minimize training and allow the user to concentrate on the problem at hand; complex rules introduce discontinuities and distractions into the user's thought processes.

A user interface may be complete but inefficient; that is, expressing ideas may be difficult and time consuming. For example, a system for logic design needs to provide only a single building block, either the **nor** or the **nand**, but such a system will be laborious to use and thus inefficient. It is better to include in the system a facility for building up more complex commands from the few basic ones.

Extensibility can be exploited to make a language more efficient by defining new terms as combinations of existing terms. Extensibility is commonly provided in operating systems via scripts, cataloged procedures, or command files, and in programming languages via macros, but is less often found in graphics systems.

In person–person dialogue, one person asks a question or makes a statement, and the other responds, usually quite quickly. Even if a reply does not come immediately, the listener usually signals attentiveness via facial expressions or gestures. These are forms of *feedback*, a key component of user–computer dialogue. In both sorts of dialogue, the ultimate response may be provided either in words or with some gesture or facial expression—that is, with a graphic image.

Occasionally, too, the speaker makes a mistake, then says, "Oops, I didn't mean that," and the listener discards the last statement. Being able to undo mistakes is also important in user–computer dialogues.

In a conversation, the speaker might ask the listener for help in expressing a thought, or for further explanation. Or the speaker might announce a temporary digression to another

subject, holding the current subject in abeyance. These same capabilities should also be possible in user–computer dialogues.

With this general framework, let us define the components of the user–computer interface more specifically. Two languages constitute this interface. With one, the user communicates with the computer; with the other, the computer communicates with the user. The first language is expressed via actions applied to various interaction devices, and perhaps also via spoken words. The second language is expressed graphically through lines, points, character strings, filled areas, and colors combined to form displayed images and messages, and perhaps aurally through tones or synthesized words.

Languages have two major components: the meaning of the language, and the form of the language. The *meaning* of a language is its content, or its message, whereas the *form* is how that meaning is conveyed. In person–person dialogue, the meaning "I am happy" can be conveyed with the words "I am happy," or with the words "Ich bin glücklich," or with a smile. In user–computer dialogue, the meaning "delete temp9" might be conveyed by typing the command "DELETE temp9" or by dragging an icon representing file temp9 to a trashcan icon. The form of an interface is commonly called its "look and feel."

There are two elements to meaning in interface design: the conceptual and the functional. There are also two elements to form: sequencing and binding to hardware primitives. The user-interface designer must specify each of these four elements.

The *conceptual design* is the definition of the principal application concepts that must be mastered by the user, and is hence also called the *user's model* of the application. The conceptual design typically defines *objects*, *properties* of objects, *relationships* between objects, and *operations* on objects. In a simple text editor, the objects are characters, lines, and files, a property of a file is its name, files are sequences of lines, lines are sequences of characters, operations on lines are Insert, Delete, Move, and Copy, and the operations on files are Create, Delete, Rename, Print, and Copy. The conceptual design of a user interface is sometimes described by means of a metaphor or analogy to something with which the user is already familiar, such as a typewriter, Rolodex, drafting table and instruments, desktop, or filing cabinet. Although such analogies are often helpful for initial understanding, they can become harmful if they must be stretched unrealistically to explain the more advanced capabilities provided by the computer system [HALA82].

The *functional design* specifies the detailed functionality of the interface: what information is needed for each operation on an object, what errors can occur, how the errors are handled, and what the results of each operation are. Functional design is also called *semantic design*. It defines meanings, but not the sequence of actions or the devices with which the actions are conducted.

The *sequencing design*, part of the form of an interface, defines the ordering of inputs and outputs. Sequencing design is also called *syntactic design*. For input, the sequencing comprises the rules by which indivisible units of meaning (input to the system via interaction techniques) are formed into complete sentences. Units of meaning cannot be further decomposed without loss of meaning. For example, the mouse movements and mouse button clicks needed to make a menu selection do not individually provide information to the application.

For output, the notion of sequence includes spatial and temporal factors. Therefore, output sequencing includes the 2D and 3D layout of a display, as well as any temporal variation in the form of the display. The units of meaning in the output sequence, as in the input sequence, cannot be further decomposed without loss of meaning; for example, a transistor symbol has meaning for a circuit designer, whereas the individual lines making up the symbol do not have meaning. The meanings are often conveyed graphically by symbols and drawings, and can also be conveyed by sequences of characters.

The hardware *binding design*, also called the *lexical design*, is also part of the form of an interface. The binding determines how input and output units of meaning are actually formed from hardware primitives. The input primitives are whatever input devices are available, and the output primitives are the shapes (such as lines and characters) and their attributes (such as color and font) provided by the graphics subroutine package. Thus, for input, hardware binding is the design or selection of interaction techniques, as discussed in Chapter 8. For output, hardware binding design is the combining of display primitives and attributes to form icons and other symbols.

To illustrate these ideas, let us consider a simple furniture-layout program. Its conceptual design has as objects a room and different pieces of furniture. The relation between the objects is that the room contains the furniture. The operations on the furniture objects are Create, Delete, Move, Rotate, and Select; the operations on the room object are Save and Restore. The functional design is the detailed elaboration of the meanings of these relations and operations.

The sequence design might be to select first an object and then an operation on that object. The hardware-binding component of the input language might be to use a mouse to select commands from the menu, to select furniture objects, and to provide locations. The sequence of the output design defines the screen arrangement, including its partitioning into different areas and the exact placement of menus, prompts, and error messages. The hardware-binding level of the output design includes the text font, the line thickness and color, the color of filled regions, and the way in which output primitives are combined to create the furniture symbols.

Section 9.2 discusses some of the fundamental forms a user interface can take; Section 9.3 presents a set of design guidelines that applies to all four design levels. In Section 9.4, we present issues specific to input sequencing and binding; in Section 9.5, we describe visual design rules for output sequencing and binding. Section 9.6 outlines an overall methodology for user-interface design.

9.2 USER-INTERFACE STYLES

Three common styles for user–computer interfaces are *what you see is what you get, direct manipulation,* and *iconic*. In this section, we discuss each of these related but distinct ideas, considering their applicability, their advantages and disadvantages, and their relation to one another. There is also a brief discussion of other styles of user–computer interaction: menu selection, command languages, natural-language dialogue, and question–answer dialogue. These are not emphasized, because they are not unique to graphics. (Menus are the closest,

but their use certainly predates graphics. Graphics does, however, permit use of icons rather than of text as menu elements, and provides richer possibilites for text typefaces and fonts and for menu decorations.) None of these styles are mutually exclusive; successful interfaces often meld elements of several styles to meet design objectives not readily met by one style alone.

9.2.1 What You See Is What You Get

What you see is what you get, or *WYSIWYG* (pronounced wiz-ee-wig), is fundamental to interactive graphics. The representation with which the user interacts on the display in a WYSIWYG interface is essentially the same as the image ultimately created by the application. Most, but not all, interactive graphics applications have some WYSIWYG component.

Many text editors (most assuredly a graphics application) have WYSIWYG interfaces. Text that is to be printed in boldface characters is displayed in boldface characters. With a non-WYSIWYG editor, the user sees control codes in the text. For example,

In this sentence, we show @b(bold), @i(italic), and @ub(underlined bold) text.

specifies the following hardcopy output:

In this sentence, we show **bold,** *italic*, and **underlined bold** text.

A non-WYSIWYG specification of a mathematical equation might be something like

@f(@i(u)@sub(max) − @i(u)@sub(min),@i(x)@sub(max) − @i(x)@sub(min))

to create the desired result

$$\frac{u_{max} - u_{min}}{x_{max} - x_{min}}.$$

In such non-WYSIWYG systems, users must translate between their mental image of the desired results and the control codes. Confirmation that the control codes reproduce the mental image is not given until the coded input is processed.

WYSIWYG has some drawbacks. Whenever the spatial and intensity or color resolution of the screen differs from that of the hardcopy device, it is difficult to create an *exact* match between the two. Chapter 13 discusses problems that arise in accurately reproducing color. More important, some applications cannot be implemented with a pure WYSIWYG interface. Consider first text processing, the most common WYSIWYG application. Many text processors provide heading categories to define the visual characteristics of chapter, section, subsection, and other headings. Thus, "heading type" is an object property that must be visually represented. But the heading type is not part of the final hardcopy, and thus, by definition, cannot be part of the display either. There are simple solutions, such as showing heading-type codes in the left margin of the display, but they are counter to the WYSIWYG philosophy. It is for this reason that WYSIWYG is sometimes called "what you see is *all* you get." As a second example, the robot arm in Fig.

7.1 does not reveal the existence of hierarchical relationships between the robot's body, arms, and so on, and it certainly does not show these relationships. These examples are intended not as indictments of WYSIWYG but rather as reminders of its limitations.

9.2.2 Direct Manipulation

A *direct-manipulation user interface* is one in which the objects, attributes, or relations that can be operated on are represented visually; operations are invoked by actions performed on the visual representations, typically using a mouse. That is, commands are not invoked explicitly by such traditional means as menu selection or keyboarding; rather, the command is implicit in the action on the visual representation. This representation may be text, such as the name of an object or property, or a more general graphic image, such as an icon. Later in this section, we discuss the circumstances under which textual and iconic forms of visual representation are appropriate.

The Macintosh interface uses direct manipulation in part, as shown in Fig. 9.1. Disks and files are represented as icons. Dragging a file's icon from one disk to another copies the file from one disk to the other; dragging to the trashcan icon deletes the file. In the earlier Xerox Star, dragging a file to a printer icon printed the file. Shneiderman [SHNE83], who coined the phrase "direct manipulation," discusses other examples of this technique.

Direct manipulation is sometimes presented as being the best user-interface style. It is certainly quite powerful and is especially easy to learn. But the Macintosh interface can be slow for experienced users in that they are forced to use direct manipulation when another

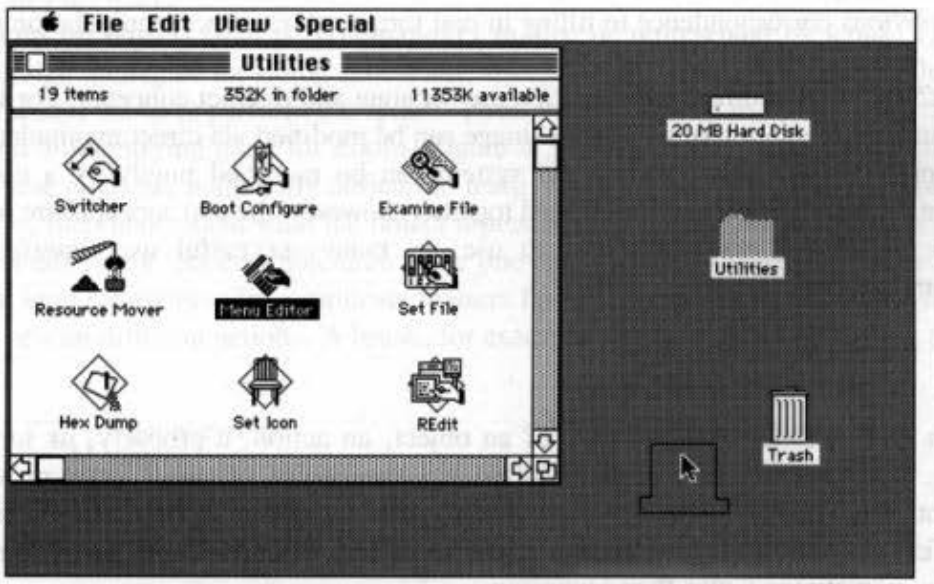


Fig. 9.1 The Macintosh screen. In the upper right is a disk icon; just below it is a directory icon, which is gray-toned to indicate that it is open. At the left is the open directory, with named icons representing the files within it. A file, represented by the icon outline around the cursor, is being dragged to the trashcan at the lower right. (Screen graphics © Apple Computer, Inc.)

style would generally be faster. Printing the file "Chapter 9" with direct manipulation requires the visual representation of the file to be found and selected, then the Print command is involved. Finding the file icon might involve scrolling through a large collection of icons. If the user knows the name of the file, typing "Print Chapter 9" is faster. Similarly, deleting all files of type "txt" requires finding and selecting each such file and dragging it to a trash can. Much faster is the UNIX-style command "rm *.txt", which uses the wild card * to find all files whose names end in ".txt."

An interface combining direct manipulation with command-language facilities can be faster to use than is one depending solely on direct manipulation. Note that direct manipulation encourages the use of longer, more descriptive names, and this tends to offset some of the speed gained from using typed commands. Some applications, such as programming, do not lend themselves to direct manipulation [HUTC86], except for simple introductory flowchart-oriented learning or for those constructs that in specialized cases can be demonstrated by example [MAUL89; MYER86].

Direct-manipulation interfaces typically incorporate other interface styles, usually commands invoked with menus or the keyboard. For instance, in most drafting programs, the user rotates an object with a command, not simply by pointing at it, grabbing a handle (as in Section 8.3.3), and rotating the handle. Indeed, it is often difficult to construct an interface in which all commands have direct-manipulation actions. This reinforces the point that a single interaction style may not be sufficient for a user interface: Mixing several styles is often better than is adhering slavishly to one style.

The form fill-in user interface is another type of direct manipulation. Here a form is filled in by pointing at a field and then typing, or by selecting from a list (a selection set) one of several possible values for the field. The limited functional domain of form fill-in and its obvious correspondence to filling in real forms makes direct manipulation a natural choice.

WYSIWYG and direct manipulation are separate and distinct concepts. For instance, the textual representation of a graphics image can be modified via direct manipulation, and the graphical image of a WYSIWYG system can be modified purely by a command-language interface. Especially when used together, however, the two concepts are powerful, easy to learn, and reasonably fast to use, as many successful user interfaces have demonstrated.

9.2.3 Iconic User Interfaces

An *icon* is a pictorial representation of an object, an action, a property, or some other concept. The user-interface designer often has the choice of using icons or words to represent such concepts. Note that the use of icons is not related to the direct-manipulation issue: Text can be directly manipulated just as well as icons can, and text can represent concepts, sometimes better than icons can.

Which is better, text or icons? As with most user-interface design questions, the answer is, "it depends." Icons have many advantages. Well-designed icons can be recognized more quickly than can words, and may also take less screen space. If carefully designed, icons can be language-independent, allowing an interface to be used in different countries.

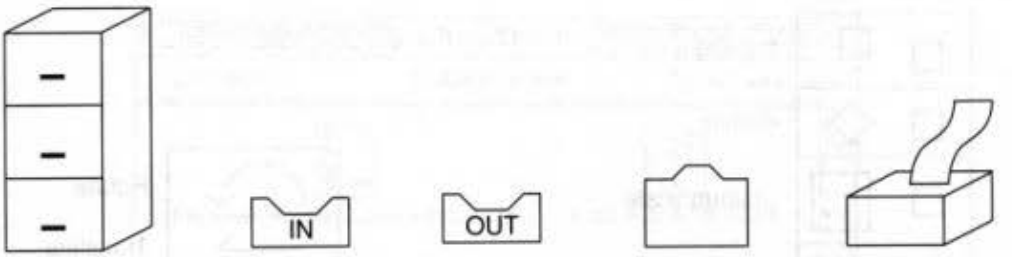


Fig. 9.2 Icons used to represent common office objects.

Icon design has at least three separate goals, whose importance depends on the specific application at hand:

1. *Recognition*—how quickly and accurately the meaning of the icon can be recognized
2. *Remembering*—how well the icon's meaning can be remembered once learned
3. *Discrimination*—how well one icon can be distinguished from another.

See [BEWL83] for a report on experiments with several alternative icon designs; see [HEME82; MARC84] for further discussion of icon-design issues.

Icons that represent objects can be designed relatively easily; Fig. 9.2 shows a collection of such icons from various programs. Properties of objects can also be represented easily if each of their values can be given an appropriate visual representation. This certainly can be done for the properties used in interactive graphics editors, such as line thickness, texture, and font. Numeric values can be represented with a gauge or dial icon, as in Fig. 8.21.

Actions on objects (that is, commands) can also be represented by icons. There are several design strategies for doing this. First, the command icon can represent the *object* used in the real world to perform the action. Thus, scissors can be used for Cut, a brush for Paste, and a magnifying glass for Zoom. Figure 9.3 shows a collection of such command icons. These icons are potentially difficult to learn, since the user must first recognize what the icon *is*, then understand what the object represented *does*. This two-step understanding process is inherently less desirable than is the one-step process of merely recognizing what object an icon represents. To complicate matters further, suppose that the object might be used for several different actions. A brush, for example, can be used for spreading paste (to

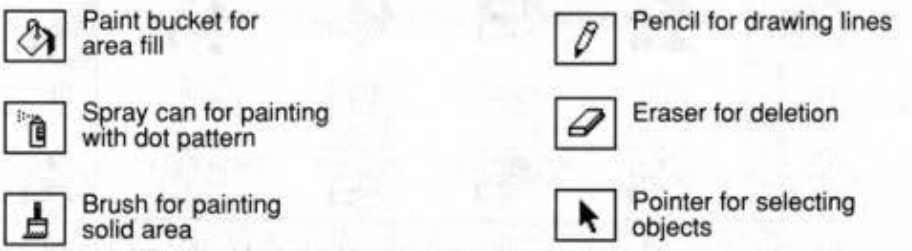


Fig. 9.3 Command icons representing objects used to perform the corresponding command. (Copyright 1988 Claris Corporation. All rights reserved.)

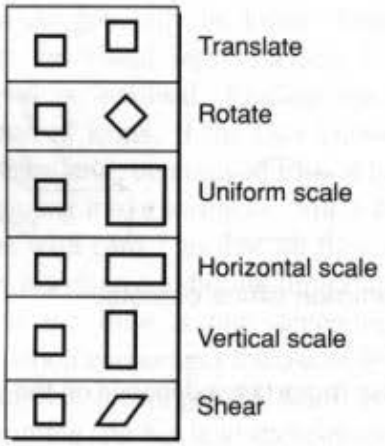


Fig. 9.4 Command icons indicating geometric transformations by showing a square before and after the commands are applied.

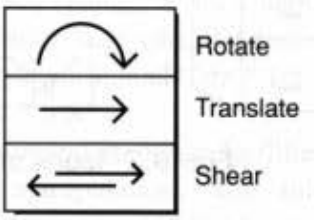


Fig. 9.5 Several abstract command icons for some of the actions depicted in Fig. 9.4. Not all geometric operations can be represented in this way.

paste something in place), and also for spreading paint (to color something). If both Paste and Paint could reasonably be commands in the same application, the brush icon could be ambiguous. Of course, sometimes only one interpretation will make sense for a given application.

Another design strategy for command icons is to show the command's *before and after* effects, as in Fig. 9.4 and Color Plates I.19–I.21. This works well if the representations for the object (or objects) are compact. If the command can operate on many different types of objects, however, then the specific object represented in the icon can mislead the user into thinking that the command is less general than it really is.

The NeXT user interface, implemented on a two-bit-per-pixel display, uses icons for a variety of purposes, as seen in Color Plate I.22.

A final design approach is to find a more *abstract representation* for the action. Typical examples are shown in Fig. 9.5. These representations can depend on some cultural-specific concept, such as the octagonal stop-sign silhouette, or can be more generic, such as X for Delete.

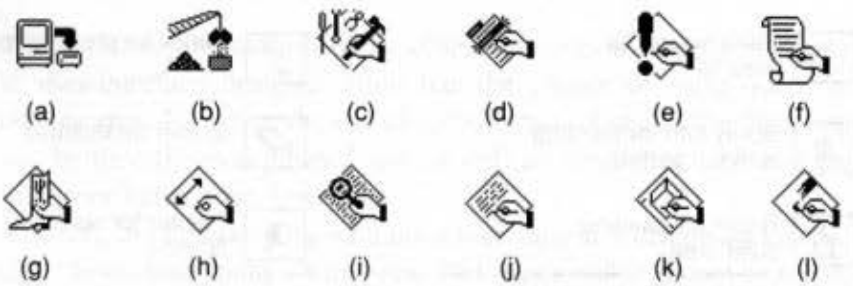


Fig. 9.6 Icons that represent Macintosh programs. What does each icon represent? In most cases, the icons suggest the type of information that is operated on or created. See Exercise 9.14 for the answers.

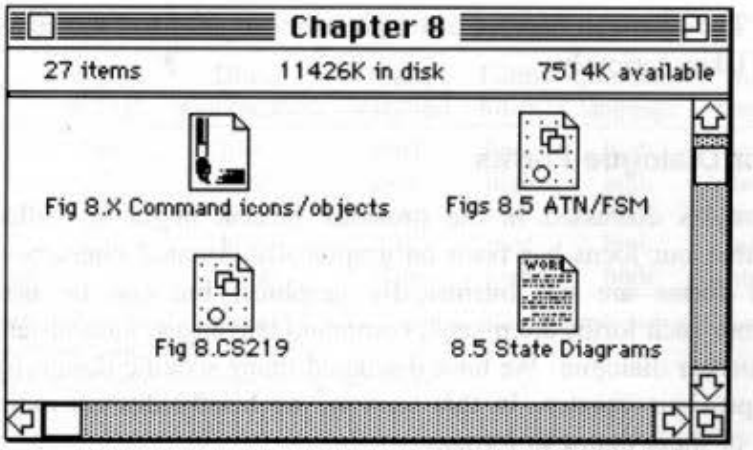


Fig. 9.7 The contents of a disk directory represented with icons and text. The icons help to distinguish one file from another. (Certain screen graphics © Apple Computer, Inc.)

Arbitrarily designed icons are not necessarily especially recognizable. Figure 9.6 shows a large number of icons used to represent Macintosh programs. We challenge you to guess what each program does! However, once learned, these icons seem to function reasonably well for remembering and discrimination.

Many visual interfaces to operating systems use icons to discriminate among files used by different application programs. All files created by an application share the same icon. If a directory or disk contains many different types of files, then the discrimination allowed by the icon shapes is useful (see Fig. 9.7). If all the files are of the same type, however, this discrimination is of no use whatsoever (see Fig. 9.8).

Icons can be poorly used. Some users dislike icons such as the trashcan, contending that such ideas are juvenile, "cute," and beneath their dignity. The designer may or may not agree with such an evaluation, but the user's opinion is usually more important than is

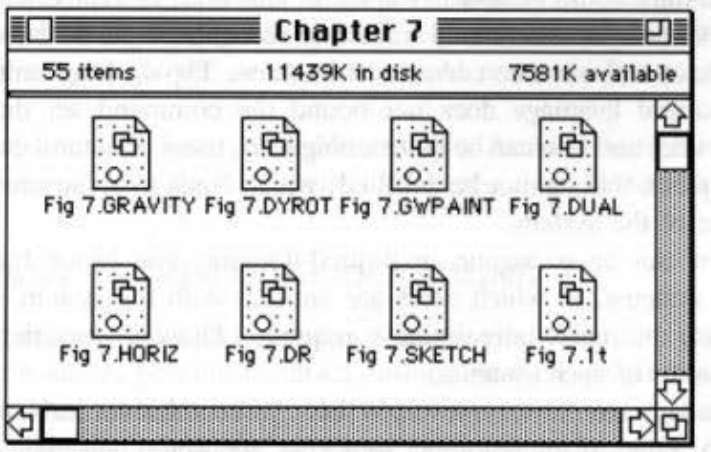


Fig. 9.8 The contents of a disk directory represented with icons and text. Since the files are all of the same type, the icons do not help to distinguish one file from another, and simply take up extra space. (Computer screen graphics © Apple Computer, Inc.)

the designer's. The user who dislikes a computer or program and thus develops a negative attitude is best taken seriously.

9.2.4 Other Dialogue Forms

The dialogue styles discussed in the previous section might be called "intrinsically graphical" in that our focus has been on graphically oriented interaction. A number of other dialogue forms are not intrinsically graphical but can be used in graphical applications. Four such forms are menus, command languages, natural-language dialogue, and question–answer dialogue. We have discussed many specific design issues concerning menus in the previous chapter. In this section, we briefly discuss more general issues involving each of these dialogue forms.

Menus are widely used in both graphical and nongraphical applications. In either case, however, the fundamental advantage of menus is that the user can work with what is called *recognition memory*, where visual images (textual or iconic menu items) are associated with already-familiar words and meanings. This contrasts with *recall memory*, where the user must recall from memory a command or concept in order to enter information into the computer. Menus reduce the memory load for users, and hence are especially attractive to novices. Menus, along with form fill-in, allow current selections to be indicated visually, further reducing the user's memory load and also allowing rapid input if the current selection is desired. On the other hand, menus limit the size of the selection set of alternatives, whereas some of the other dialogue styles do not.

Use of a *command language* is the traditional way to interact with a computer. This technique can accommodate large selection sets, is easy to extend (just add another command), and is fairly fast for experienced users who can type. Learning time is its major liability, with the need for typing skills a second factor. Errors are more likely with command languages than with menus, because of the possibility of typing and recall errors.

Natural-language dialogue is often proposed as the ultimate objective for interactive systems: If computers could understand our commands, typed or spoken in everyday English, then everyone would be able to use them. However, current voice recognizers with large vocabularies must be individually trained to recognize a particular user's voice; they also make mistakes, and must be corrected somehow. Typing long sentences is tedious. Also, because natural language does not bound the command set that an application program must handle, and also can be quite ambiguous, users of natural-language interfaces tend to make requests that cannot be fulfilled, which leads to frustration of the user and poor performance of the system.

This problem can be overcome in limited-domain (and hence limited-vocabulary) natural-language systems, in which users are familiar with the system's capabilities and hence are unlikely to make unreasonable requests. Drawing programs and operating systems are examples of such systems.

There is a fundamental flaw, however, in the argument that natural language interaction is the ultimate objective. If the argument were true, we would be satisfied to interact with one another solely by means of telephone and/or keyboard communications. It is for this reason that voice input of natural language to an interactive graphics application program is

TABLE 9.1 COMPARISON OF SEVEN USER INTERFACE STYLES

	WYSI-WYG*	Direct manipulation	Menu selection	Form fill-in	Command language	Natural language	Q/A dialogue
learning time	low	low	med	low	high	low	low
speed of use		med	med	high	high	med	low
error-proneness	low	low	low	low	high	high	low
extensibility	low	low	med	med	high	high	high
typing skill required		none	none	high	high	high**	high

*WYSIWYG has several blank fields because it is not a complete interface style, since it must be accompanied by some means of entering commands.
**Assuming keyboard input; none for voice-recognizer input.

most likely to be used in combination with other dialogue styles, to allow overlapped use of the voice and hands to speed interaction. After all, this is exactly how we work in the real world: we point at things and talk about them. This powerful concept was compellingly demonstrated a decade ago in the "Put-that-There" [BOLT80; BOLT84] program for manipulating objects. In this system, the user can move an object by pointing at it while saying "put that;" pointing elsewhere, and saying "there." A recent study of a VLSI design program using voice input of commands combined with mouse selection of objects and positions found that users worked 60 percent faster than did those who had just a mouse and keyboard [MART89].

Question–answer dialogue is computer-initiated, and the user response is constrained to a set of expected answers. Using a keyboard for input, the user can give any answer. If the set of expected answers is small, the question can include the possible answers; menu selection might even be provided instead of typing as a means for the user to respond. In the limit, question–answer dialogue becomes a sequential set of menu selections. A common failing of instances of this dialogue form is the inability to go back several steps to correct an answer. A more general problem with the sequentiality implied by this form is that of context: The user has only the context of the past and current questions to assist in interpreting the current question. With a form fill-in dialogue, by contrast, the user can see all the fields to be entered, and so can quickly tell, for instance, whether an apartment number in an address goes in the street-address field or in a separate apartment-number field.

Table 9.1 compares user-interface dialogue styles. A much more extensive discussion of the pros and cons of many of these styles can be found in [SHNE86].

9.3 IMPORTANT DESIGN CONSIDERATIONS

In this section, we describe a number of design principles to help ensure good human factors in a design: be consistent, provide feedback, minimize error possibilities, provide error recovery, accommodate multiple skill levels, and minimize memorization. Application of these principles is generally considered necessary, although by no means is sufficient, for a successful design. These and other principles are discussed more fully in [FOLE74; GAIN84; HANS71; MAYH90; RUBE84; SHNE86].

9.3.1 Be Consistent

A *consistent* system is one in which the conceptual model, functionality, sequencing, and hardware bindings are uniform and follow a few simple rules, and hence lack exceptions and special conditions. The basic purpose of consistency is to allow the user to *generalize* knowledge about one aspect of the system to other aspects. Consistency also helps to avoid the frustration induced when a system does not behave in an understandable and logical way. The best way to achieve this consistency is through a careful top-down design of the overall system.

Simple examples of consistency in the output portion of a user interface are

- The same codings are always employed. Colors always code information in the same way, just as red always means stop, and green always means go.
- System-status messages are shown at a logically (although not necessarily physically) fixed place.
- Menu items are always displayed in the same relative position within a menu, so that users can allow "muscle memory" to help in picking the desired item.

Examples of consistency in the input portion are

- Keyboard characters—such as carriage return, tab, line feed, and backspace—always have the same function and can be used whenever text is being input.
- Global commands—such as Help, Status, and Cancel—can be invoked at any time.
- Generic commands—such as Move, Copy, and Delete—are provided and can be applied, with predictable results, to any type of object in the system.

We should, however, remember Emerson's observation that "A foolish consistency is the hobgoblin of little minds" [EMER03]. Consistency can conflict with other design objectives. For instance, if dragging a file icon to the trash deletes the file, what should happen when a file icon is dragged to an electronic-mail outbox? Should the file be sent and then deleted, for consistency with dragging to the trash? Or should a copy of the file be sent? If a file is dragged to a printer icon to print the file, should the file be printed and then deleted, for consistency with dragging to the trash? Surely in these latter two cases the file should not be deleted. The *law of least astonishment*, a higher design principle, suggests that doing what the user is likely to consider normal or reasonable is more important than is maintaining pure consistency.

Figure 9.9 shows how state diagrams can help to identify inconsistency. We can see here that help is available only from the move state, not from the other states. A mixed strategy is used to let users change their minds once an action sequence has been initiated. From the move and delete states there is a Cancel command, whereas the rotate state has an Undo command. The sequence of object/operation varies: for Move and Delete, the sequence is operation then object, whereas it is the reverse for rotation. The feedback strategy is also mixed: dynamic for moving, static for rotation.

Reisner demonstrated experimentally an intuitively expected result: Given two functionally equivalent user interfaces, new users make fewer errors and learn more quickly with

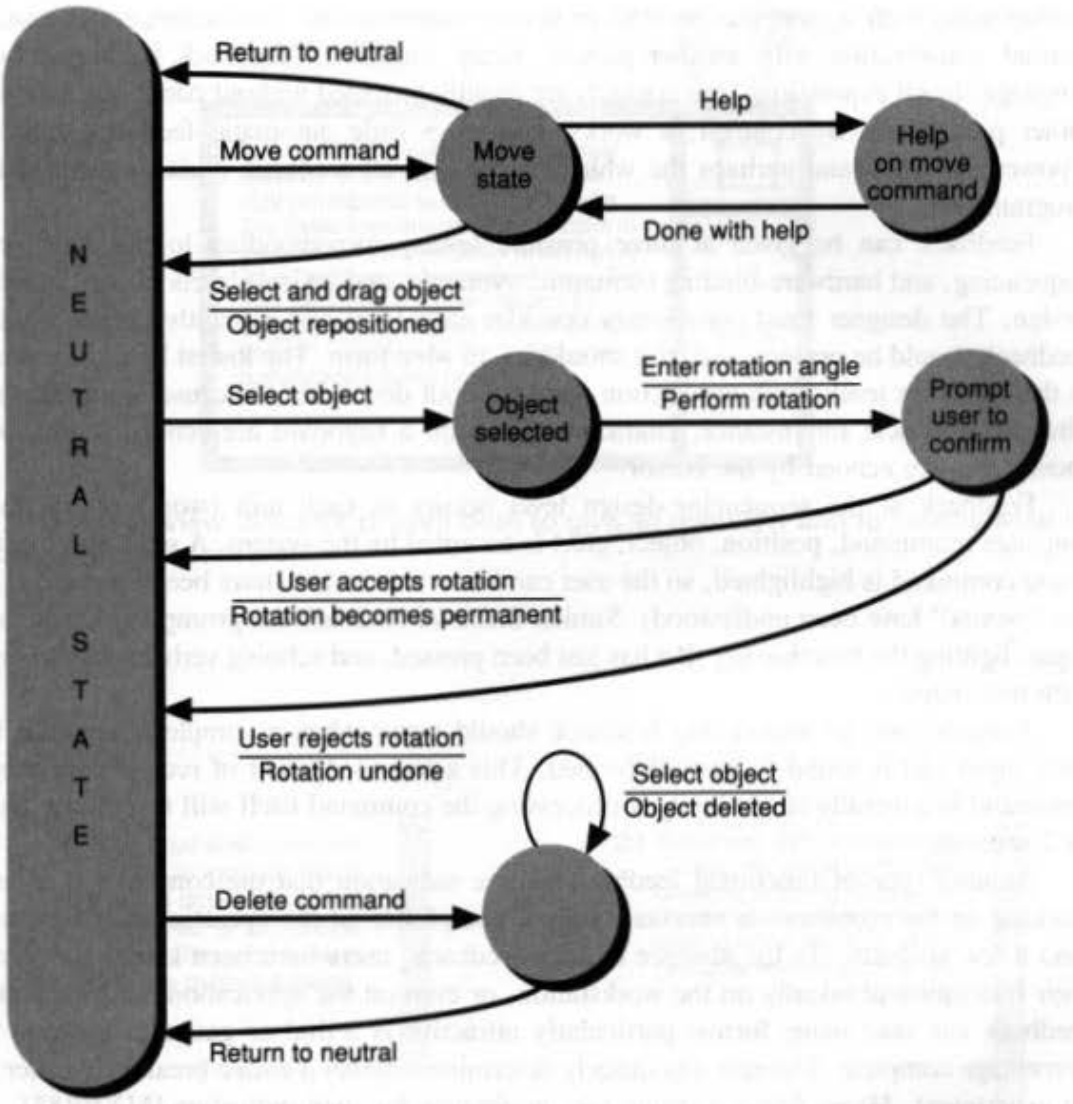


Fig. 9.9 State diagram of a user interface with an inconsistent syntax.

one that has a simpler syntactic structure [REIS82]. Thus, another useful design objective to apply is simply to minimize the number of different syntactic structures.

At the functional level, consistency requires the use of *generic commands* that apply across as broad a range as possible. For example, chairs and desks should be moved in the same way in the room-layout program discussed previously; files should be opened, deleted, and saved from within application programs with the same generic file-manipulation commands.

9.3.2 Provide Feedback

Have you ever tried conversing with a partner who neither smiles nor nods, and who responds only when forced to do so? It is a frustrating experience, because there is little or no indication that the partner understands what you are saying. Feedback is as essential in

conversation with a computer as it is in human conversation. The difference is that, in normal conversation with another person, many sources of feedback (gestures, body language, facial expressions, eye contact) are usually provided without conscious action by either participant. By contrast, a workstation gives little automatic feedback (just the "power on" light and perhaps the whir of a fan), so all feedback must be planned and programmed.

Feedback can be given at three possible levels, corresponding to the functional, sequencing, and hardware-binding (semantic, syntactic, and lexical) levels of user-interface design. The designer must consciously consider each level and explicitly decide whether feedback should be present, and, if it should be, in what form. The lowest level of feedback is the hardware level. Each user action with an input device should cause immediate and obvious feedback: for instance, characters typed on a keyboard are echoed, and mouse movements are echoed by the cursor.

Feedback at the sequencing-design level occurs as each unit (word) of the input language (command, position, object, etc.) is accepted by the system. A selected object or menu command is highlighted, so the user can know that actions have been accepted (i.e., the "words" have been understood). Similar forms of feedback are prompting for the next input, lighting the function key that has just been pressed, and echoing verbal (speech) input with text output.

Another form of sequencing feedback should occur when a complete sequence has been input and is found to be well formed. This acknowledgment of receipt of a proper command is generally needed only if processing the command itself will take more than 1 or 2 seconds.

Another type of functional feedback—some indication that the computer is at least working on the problem—is necessary only if completion of the operation will take more than a few seconds. (In the absence of such feedback, users have been known to express their frustration physically on the workstation, or even on the application designer!) Such feedback can take many forms; particularly attractive is a dial or gauge to indicate the percentage complete. The user can quickly determine whether a coffee break is in order. In an experiment, Myers found a strong user preference for such indicators [MYER85].

The most useful and welcome form of functional feedback tells the user that the requested operation has been completed. This is usually done with a new or modified display that explicitly shows the results of the operation.

It is useful to distinguish between problem-domain and control-domain feedback. *Problem-domain* feedback concerns the actual objects being manipulated: their appearance, their position, their existence. *Control-domain* feedback has to do with the mechanisms for controlling the interactive system: status, current and default values, menus, and dialogue boxes.

Problem-domain feedback is needed if users can see just part of a large drawing, so that they can know which part of the world is being displayed. Figure 9.10 shows one way to do this. The approach can be even more effective with two displays—one for the overview, the other for the detail. In either case, the rectangle in the overview indicates which part of the drawing is being shown in the detailed display. Panning and zooming are generally effected by dragging and resizing the overview rectangle. Figure 9.11 shows how increasing

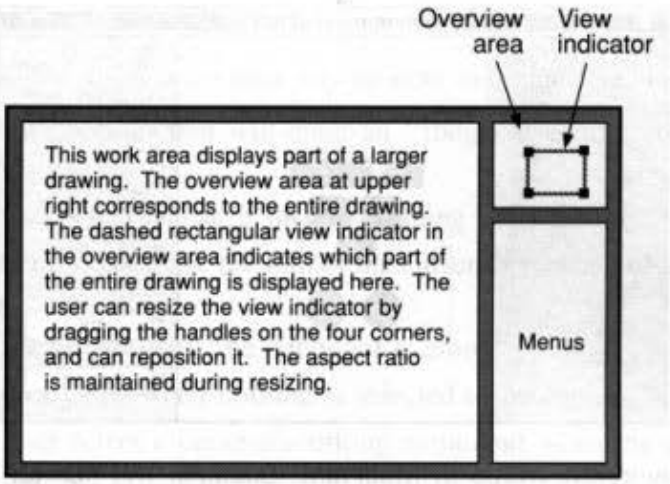


Fig. 9.10 The view indicator is used both to give an overview and to control what is displayed in the work area.

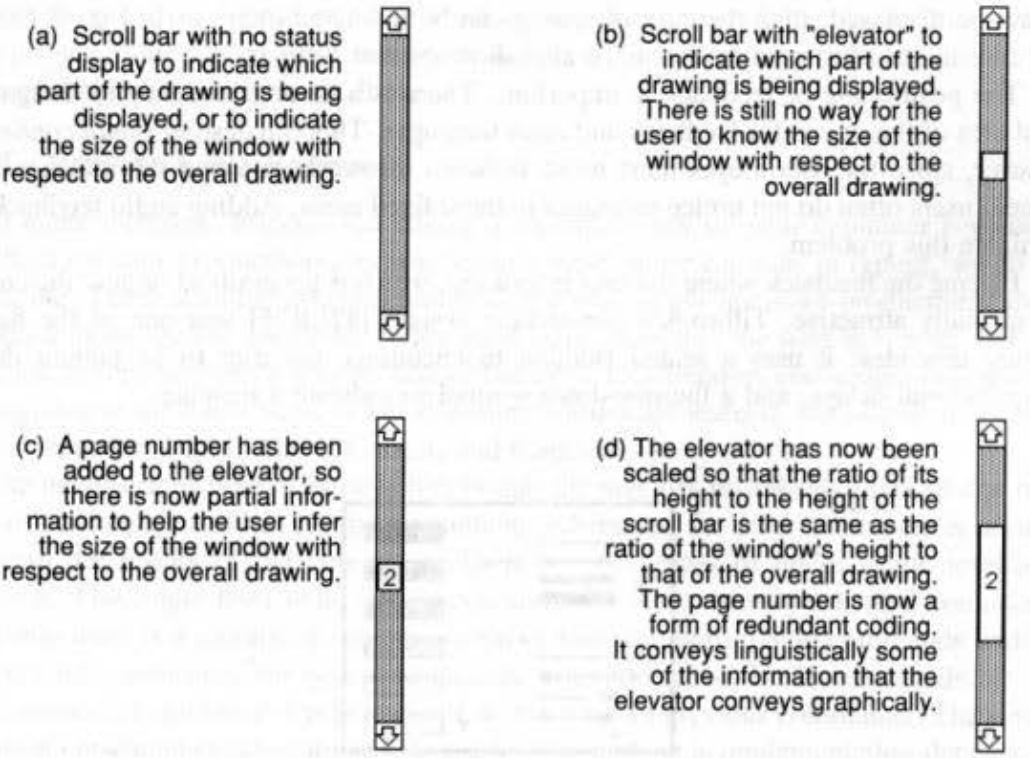


Fig. 9.11 Four different levels of feedback in scroll bars, ranging from none in (a) to redundant coding in (d).

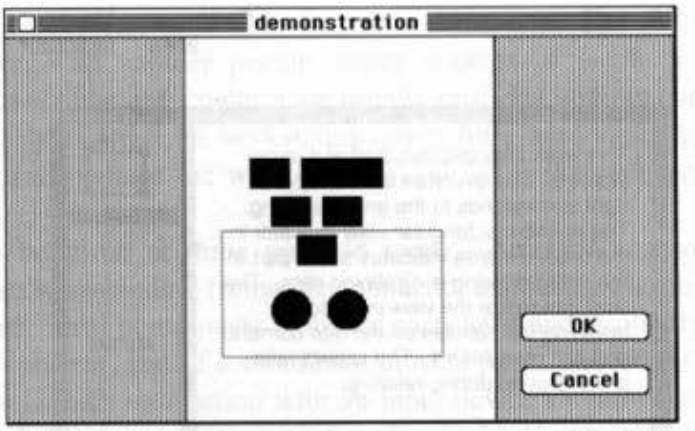


Fig. 9.12 The overview mode in MacPaint. Because the screen is very small, the overview alternates with the normal, more detailed view of the drawing. The user drags the rectangular dashed box to the desired area, and selects "OK" to see the detailed view of the enclosed area. (Copyright 1988 Claris Corporation. All rights reserved.)

amounts of feedback can be built into window scroll bars. Another approach to orienting the viewer is used in MacPaint, as shown in Fig. 9.12.

An important type of control-domain feedback is *current settings*. Current settings can be shown in a feedback area. If the menus or other tools by which settings are selected can always be displayed, then the current setting can be indicated there, as in Fig. 9.13. The pull-out menus illustrated in Fig. 8.14 also show current settings.

The positioning of feedback is important. There is a natural tendency to designate a fixed area of the screen for feedback and error messages. This can destroy *visual continuity*, however, since the user's eyes must move between the work area and the message area. Indeed, users often do not notice messages in these fixed areas. Adding audio feedback can eliminate this problem.

Placing the feedback where the user is looking, which is generally at or near the cursor, is especially attractive. Tilbrook's Newswhole system [TILB76] was one of the first to employ this idea; it uses a seated Buddha to encourage the user to be patient during computational delays, and a thumbs-down symbol to indicate a mistake.

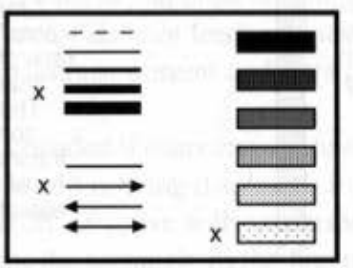


Fig. 9.13 In this menu of graphic attributes, the x indicates the current setting. The user has constant feedback if the menu can be permanently visible. If not, a more compact form of permanent feedback should be used.

9.3.3 Minimize Error Possibilities

Don't set the user up for a fall is another way to state this objective. For example,

- Do not offer menu options that will elicit an "illegal selection, command not valid now" message
- Do not let the user select Delete if there is nothing to be deleted
- Do not let the user try to change the font of the currently selected object if the object is not a text string
- Do not let the user Paste when the clipboard is empty
- Do not let the user Copy when nothing is selected to be copied
- Do not let the user select a curve-smoothing command when the currently selected object is not a curve.

In all these instances, the system should instead disable unavailable items and alert the user by changing the menu item's appearance—for instance, by making it gray instead of black.

These are all examples of *context sensitivity*: The system provides the user with only those commands that are valid in the current context or mode. When there is a context, the system should guide the user to work within that context and should make it difficult or impossible for the user to do things that are not permissible in that context.

Another aspect of this objective is to avoid *side effects*, which are results the user has not been led to expect. The classic side effect is the print command that also deletes the file being printed. Side effects arise from poor design or ineffective communication with the user regarding what a command does.

9.3.4 Provide Error Recovery

We all make mistakes. Imagine not having a backspace key on your computer keyboard! The effect on your productivity, as you became much more cautious in typing, would be devastating. There is ample experimental evidence that people are more productive if their mistakes can be readily corrected. With good error recovery, the user is free to explore unlearned system facilities without "fear of failure." This freedom encourages exploratory learning, one of the major ways in which system features are learned. We discuss four types of error recovery: Undo, Abort, Cancel, and Correct.

The most serious type of error is functional: the user has mistakenly invoked one or a series of commands and has obtained unanticipated results. An *Undo* command is needed to reverse the results of the command. There are two types of undo: single level and multilevel. The single-level undo can reverse only the most recently executed command. This Undo itself is a command, so the second of two successive Undo commands undoes the first Undo, returning the system to its state prior to the two Undo commands.

In contrast, a multilevel Undo operates on the stack of previous commands. The actual number of commands stacked up and thus able to be undone is implementation-dependent; in some cases, all commands since the session began are saved. With a multilevel undo, a *Redo* command is also needed, so that, if the user backs up too far in the command stack, the most recently undone command can be redone. Neither Undo nor Redo is entered on

the stack. Several tricky issues concerning Redo are discussed in [VITT84]. For a single level, Undo and Redo are mutually exclusive: One or the other, but not both, can be available at any given time. For multilevel, Undo and Redo can be available at the same time.

Users often need help in understanding the scope of the Undo command (i.e., how much work the command will undo). They also are often confused about whether Undo applies to windowing commands, such as scrolling, as well as to application commands. If the Undo command is in a menu, the menu-item text can indicate what will be undone; instead of "undo," the entry could be "undo copy" or "undo deletion." Kurlander and Feiner have developed a graphical way to show the history of user actions and the scope of Undo, shown in Color Plate I.23 [KURL88; KURL90].

A form of undo is sometimes provided as an *explicit-accept, explicit-reject* step. After a command is carried out and its results are shown on the display, the user must accept or reject the results before doing anything else. This step adds to the number of user actions required to accomplish a task, but does force the user to think twice before confirming acceptance of dangerous actions. However, an Undo command is usually preferable because an explicit action is required only to reject the results of a command; the command is implicitly accepted when the next command (other than the Undo command) is entered. Hence, we call undo an *implicit-accept, explicit-reject* strategy.

No matter how undo is provided, its implemention requires extra programming, especially for commands that involve major changes to data structures. An easier, although less satisfactory, alternative to undo is to require the user explicitly to confirm commands for which there is no undo. This is commonly used for the file-delete command.

A user may realize that a functional-level mistake has been made while a command is still being performed. This illustrates the need for an *Abort* command to terminate prematurely a currently executing command. Like Undo, Abort must restore the system to its exact state prior to initiation of the aborted command. In fact, Abort and Undo can be thought of as essentially the same command: They both reverse the most recently specified functional-level action. A user-interface design might make both actions available with the same name.

A less dramatic type of error occurs when the user is partway through specifying information required to carry out some command, and says, "Oops, I don't really want to do this!" A poorly designed interface gives the user no choice but to proceed with the command, after which an Undo or Abort (if available) is used to recover. A well-designed interface lets the user back out of such a situation with a *Cancel* command. This is especially common with a form fill-in dialogue, where a Cancel command is often available on the form, as in Fig. 8.32. Note that Cancel can also be thought of as a specialized Undo command, with the system reverting to the state prior to the current command.

In a less serious type of error, the user may want to *correct* one of the units of information needed for a command. The dialogue style in use determines how easy to make such corrections are. Command-language input can be corrected by multiple backspaces to the item in error, followed by reentry of the corrected information and all the information that was deleted. If the system has line-editing capabilities, then the cursor can be moved back to the erroneous information without the intervening information being deleted. Form fill-in allows simple corrections as well, whereas question–answer and menu dialogues are not so forgiving. The dynamic interaction techniques discussed in Chapter 8 provide a form

of error recovery: for instance, the position of an object being dragged into place is easy to change.

9.3.5 Accommodate Multiple Skill Levels

Many interactive graphics systems must be designed for a spectrum of users, ranging from the completely new and inexperienced user through the user who has worked with the system for thousands of hours. Methods of making a system usable at all skill levels are accelerators, prompts, help, extensibility, and hiding complexity.

New users normally are most comfortable with menus, forms, and other dialogue styles that provide considerable prompting, because this prompting tells them what to do and facilitates learning. More experienced users, however, place more value on speed of use, which requires use of function keys and keyboard commands. Fast interaction techniques that replace slower ones are called *accelerators*. Typical accelerators, such as one-letter commands to supplement mouse-based menu selection, have been illustrated in previous sections. The Sapphire window manager [MYER84], taking this idea even further, provides three rather than two ways to invoke some commands: pointing at different areas of the window banner and clicking different mouse buttons, a standard pop-up menu, and keyboard commands.

The Macintosh uses accelerators for some menu commands, as was shown in Fig. 8.13. Another approach is to number menu commands, so that a number can be typed from the keyboard, or a command can be selected with the cursor. Alternatively, the command name or abbreviation could be typed.

One of the fastest accelerators is the use of multiple clicks on a mouse button. For instance, the Macintosh user can select a file (represented as an icon) by clicking the mouse button with the cursor on the icon. Opening the file, the typical next step, can be done with a menu selection, an accelerator key, or an immediate second button click. The two rapid clicks are considerably faster than is either of the other two methods. From within applications, another scheme is used to open files, as illustrated in Fig. 9.14. The dialogue box permits a file name to be selected either by pointing or by typing. If the name is typed,

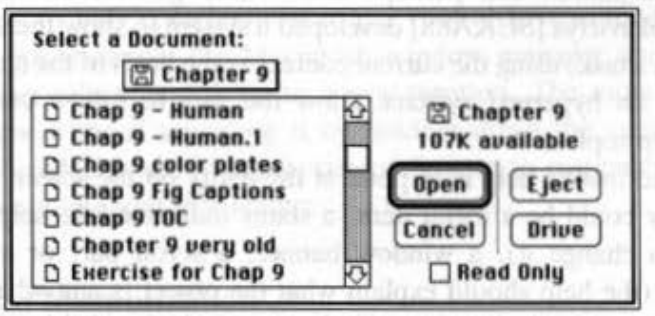


Fig. 9.14 Opening files from within a Macintosh program. The user enters the Open command, either by menu selection or with a two-key chord, causing the dialogue box to appear. The highlighted file can be opened with the "open" button or with the carriage-return key. The user can highlight a new file by selecting it with the cursor or by typing some or all of its name. Therefore, the user can open a file using only the keyboard, by entering the two-key chord, a partial file name, and the return key. (Computer screen graphics © Apple Computer, Inc.)

autocompletion permits the user to type only enough characters for the name to be specified unambiguously. Double clicking on a file name opens the file immediately.

Another form of accelerator is to provide command-line input as an alternative to the other styles. As users gain experience, they use the command line more and more. This transition can be aided by displaying the command-line equivalent of commands that are entered in other ways.

Unlike feedback, which acknowledges specific user actions, the purpose of *prompts* is to suggest what to do next. The more experienced the user, the less prompting is appropriate, especially if prompting is obtrusive and slows down the interaction or uses much of the screen. Many systems provide several levels of prompting controllable by the user; the inexperienced can be "led by the hand," whereas the experienced can proceed without the prompts.

Prompting can take many forms. The most direct is a displayed message that explains explicitly what to do next, such as "Specify location." A speech synthesizer can give explicit aural instructions to the user. Subtler forms of prompting are also available. On a function-key box, buttons eligible for selection can be illuminated. A prominent tracking cross or cursor can be displayed when a position must be input; a blinking underline cursor can indicate that a text string is to be input; a scale or dial can be displayed when a value is desired. Direct-manipulation graphical interfaces implicitly provide prompts: the icons that can be manipulated are the prompt.

A *help* facility allows the user to obtain additional information about system concepts, typical tasks, various commands and the methods used to invoke them, and interaction techniques. Ideally, help can be requested from any point in the interactive dialogue, always with the same mechanism. The return from help should leave the system in exactly the same state as when help was invoked, and the help should be context-sensitive. For example, if help is invoked while the system is awaiting a command, a list of commands available in this state should be shown (with menus or function keys, this may be unnecessary). The Help command followed by a command name should yield more information about the command. If help is requested while the parameters of a command are being entered, details about the parameters should be provided. A second Help command should produce more detailed information and perhaps allow more general browsing through online documentation. Sukaviriya [SUKA88] developed a system to show the user an animation of how to accomplish a task, using the current context as the basis of the animation. Some help capabilities based on hypertext systems allow the user to follow complex sets of links among various help topics.

An easy way to invoke help is to point at the entity on the screen about which help is desired. The entity could be a menu item, a status indicator (the help should explain the status and how to change it), a window banner, a scroll bar, or a previously created application object (the help should explain what the object is and what operations can be applied to it). This approach, however, can be used only for visible objects, not for more abstract concepts or for tasks that must be performed with a series of commands.

A help capability is appropriate even if prompts and menus are displayed, because it gives the user an opportunity to receive more detailed information than can be provided in a short prompt. Even experienced users forget details, particularly in a large and complex application.

Expert systems are beginning to be integrated into user interfaces to provide help that not only is context-sensitive, but also is tailored to individual user profiles. These profiles are developed automatically by the system as a new user and the system interact and learn more about each other, just as teachers learn about their students and custom-tailor their suggestions.

Making the user interface *extensible* means letting the user add additional functionality to the interface by defining new commands as combinations of existing commands. The key is to be able to save and replay sequences of user actions. A particularly appealing macro-definition capability is one in which user actions are automatically saved in a trace file. To create a macro, the user edits the trace file to identify the start and end of the macro, replace literals with parameters, and names the macro. Several commercial applications, such as Ashton-Tate's Full Impact, have such capabilities; Olsen has developed a particularly sophisticated prototype system [OLSE88].

Hiding complexity can allow new users to learn basic commands and to start doing productive work without becoming bogged down with specifying options, learning infrequently used specialized commands, or going through complicated start-up procedures. On the other hand, powerful systems of necessity have many commands, often with many variations. The solution to this quandary is to design the entire set of commands so that it has a small "starter kit" of commands. Default values (current or initial settings) that follow the law of least astonishment can often be useful to achieve this goal.

For example, a chart-making program should allow the user to request a pie chart, specify some data, and immediately see the chart. If the user is dissatisfied with some details of the chart layout, she should be able, say, to modify the radius of the pie, to change the color or texture of each pie slice, to add annotations, to change the text face or font used to display the data values, or to change the position of the data values displayed with each pie slice. But the user should not be forced to specify each of these explicitly when initially creating the chart.

Another design strategy is to make complicated and advanced commands available only via keyboard commands or function keys. This approach keeps the menus smaller and makes the system simpler and less intimidating. Alternatively, two or more sets of menus can be provided, each with successively more commands included.

Yet another way to hide complexity is to use control keys to modify the meaning of other commands. For instance, the Macintosh window manager normally activates the window that the user selects and drags to a new position. The more advanced user can reposition a window without activating it by holding down the control key (called the *command key* on the Macintosh, in appropriate deference to computer-naive users). New users simply are not told about this feature.

9.3.6 Minimize Memorization

Interface designs sometimes force unnecessary memorization. In one design-drafting system, objects are referred to by numeric rather than by alphanumeric names. To appreciate what that means, we can imagine an interactive operating system in which file names are numeric. The remembering and learning tool of mnemonic names would be unavailable, forcing rote memorization or the use of auxiliary written aids. Of course,

explicit picking of displayed objects or icons further eliminates the need for memorization. It is important to invoke the user's recognition rather than recall memory whenever possible.

In one interactive graphing system, a command such as "Plot years gross net" produces a trend chart of yearly gross income and net income on a single set of axes. A reasonable way to control the style of a line is to use a command such as "Linestyle net dash" (to plot the net income with a dashed line). Unhappily, the actual command is of the form "Linestyle 3 dash." The "3" refers to the third variable named in the most recent "Plot" command—in this case, net. Since the most recent Plot command is not generally on the screen, the user must remember the order of the parameters.

Some help systems completely obscure the work area, forcing the user to memorize the context in order to interpret the help message. Then, once he understands the help information, the user must remember it while returning to the context in which the error occurred. Window managers solve this problem; help information is in one window, the application is in another.

9.4 MODES AND SYNTAX

Loosely defined, a *mode* is a state or collection of states in which just a subset of all possible user-interaction tasks can be performed. Examples of modes are these:

- A state in which only commands applicable to the currently selected object are available
- A state in which a dialogue box must be completed before another operation can be performed
- A state for making drawings in a document-preparation system in which separate programs are used to edit text, to make drawings, and to lay out the document
- A state in which available commands are determined by the current data-tablet overlay.

Thus, modes provide a context within which the system and user operate.

There are two kinds of modes: harmful ones and useful ones. A *harmful mode*, as discussed by Tesler [TESL81] and by Smith and colleagues [SMIT82], lasts for a period of time, is not associated with any particular object, is not visible to the user, and serves no meaningful role. Harmful modes confuse users; users get stuck in them and cannot get out, or users forget in which mode they are and attempt to invoke commands that are not available, potentially creating errors. Modes that decrease user productivity are harmful.

On the other hand, *useful modes* narrow the choices of what to do next, so prompts and help can be more specific and menus can be shorter and thus easier to traverse. A well-organized mode structure can reduce the burden on the user's memory and can help to organize knowledge about the interface into categories based on the mode. Useful modes increase user productivity.

Useful modes clearly indicate the current mode, provide feedback to show what commands are available, and include an easy, obvious, and fast means for exiting from the mode. Window managers provide highly visible modes, since each window represents a different mode; mode switching is effected by deactivating one window and activating another.

Users can be made aware of short-lived modes by a state of heightened muscle tension while in the mode. The "button-down–dynamic feedback–button-up" interaction techniques discussed in Chapter 8 make the user aware of the mode through the muscle tension involved in holding down the mouse button [BUXT86].

Command-language syntax has a major influence on the mode structure of a user interface. The traditional prefix syntax of Command, parameter 1, . . ., parameter *n* locks the user into a mode as soon as the command is specified: Only parameters can be entered, possibly in a required order. Mechanisms for error correction (Section 9.3.4) are especially important here, because otherwise the user who erroneously selects a command must continue the potentially lengthy parameter-specification process.

One of the difficulties of prefix syntax is solved by a process called *factoring* or *orthogonalization*. Commands are provided for setting each of the parameters to a current value; parameters may also have an initial (default) value. Consider, for instance, the following unfactored command syntax, where an initial capital letter indicates a command, and lowercase letters indicate parameters:

```
Draw_line        point point line_style line_thickness line_intensity
```

We can factor out the attribute specifications into either three separate commands or one overall attribute-setting command. Hence, the user would go through the sequence

```
Set_attributes   attribute_values   {Only if the current attribute values
                                     are inappropriate}
Draw_line        point point
```

We can factor this sequence further by introducing the concept of a *current point*, which is selected by the user before she invokes the Draw_line command:

```
Set_attributes   attribute_values   {Only if the current attribute values
                                     are inappropriate}
Select_point     point              {Select start point}
Select_point     point              {Select end point}
Draw_line                           {Draw_line has no parameters—
                                     all have been factored out}
```

Completely parameterless commands are not necessarily desirable. Here, for example, specifying points and then telling the system to connect them by a line eliminates the ability to do rubberband line drawing.

What about applications in which the user tends to perform the same command several times in sequence, each time on different objects? This situation suggests using a *command-mode* syntax in which the command is entered once, followed by an arbitrary number of parameter sets for the command. For instance, the syntax to delete objects could be as follows:

```
Delete_object                       {Establish Delete_object command mode}
object
object
object
⋮
any command                         {Establish new command mode}
```

Delete_object establishes a deletion mode so that each object selected thereafter is deleted, until any other command is selected. Note that command mode implies a prefix syntax.

If we factor out the object parameter from the command whose syntax is

```
Delete_object        object
```

we introduce the concept of a *currently selected object*, or CSO. We also need a new command, Select_object, to create a mode in which there is a CSO; this CSO can then be operated on by the Delete_object command:

```
Select_object        object        {Use if no object is selected,
                                    or if another CSO is desired}
Delete_object                      {No parameter—the object
                                    has been factored out}
```

The parameter factoring has created a *postfix* syntax: The object is specified first by selection, and then the command is given. This is an especially attractive technique if the user is working with a pointing device, because we have a natural tendency to point at something before saying what to do with it; the converse order is much less natural [SUKA90].

The currently selected object is a useful concept, because the user can perform a series of operations on one object, and then move on to another. Furthermore, the Select_object command usually can be made implicit: The user simply points at the object to be selected and clicks a button. This means that factoring does not need to create extra steps.

Recall that command mode has a prefix syntax. Can its advantages be retained if a postfix syntax is preferred? The answer is yes, *if* a Repeat command, to repeat the last nonselect command, can be made available easily, say by a button-down on a multibutton mouse. If so, the user action sequence to delete several objects could be

```
Select_object        object        {Establish a CSO}
Delete_object                      {Delete the CSO}
Select_object        object        {A new CSO}
Repeat                             {Single button depression to delete}
Select_object        object        {A new CSO}
Repeat                             {Single button depression to delete}
⋮
```

Compare this to the action sequence that would be used with a true command mode:

```
Delete_object
object
object
object
⋮
```

Assuming that Select_object requires just a point and click, the extra steps needed to use Repeat_last_operation with an object mode, as opposed to using a command mode, are a single button-push per deletion.

Another sequencing alternative is the arbitrary *free-form syntax* (*nofix syntax*), which permits intermixing of different syntaxes. Whether the user specifies an object and then an action or an action and then an object, the action is carried out on the object. For example,

```
Set_attributes    attribute values
Select_object     object1          {Attributes applied to object1}
Set_attributes    attribute values
Select_object     object2          {Attributes applied to object2}
Select_object     object3
Set_attributes    attribute values {Attributes applied to object3}
```

Note that this syntactic structure cannot assume a currently selected object; if it did, then the second Set_attributes command would immediately operate on object1, rather than on object2.

Command and currently selected-object modes can be used with a free-form syntax *if* a Do_it command is added, so the user can tell the system to carry out the current command on the currently selected object. This command does, however, add another user action, as the following sequence illustrates:

```
Select_object     object           {Establish a CSO}
Set_attributes    attribute values {Establish a current command}
Do_it                              {CSO receives new attributes}
Copy                               {Establish a new current command}
Select_object     object           {Establish a new CSO}
Do_it                              {Copy CSO; copy is now the CSO}
Do_it                              {CSO copied; new copy is the CSO}
```

An alternative to this free-form syntax is a *mode-sensitive syntax*, which differentiates between the two sequences to make the Set_attributes command mode-sensitive:

```
Set_attributes    attribute values {No CSO at this point}
Select_object     object
```

and

```
Select_object     object           {Establish a CSO}
Set_attributes    attribute values
```

Mode-sensitivity is a special case of a more general context-sensitivity, by which the effect of a command depends on the current context. In the first of the preceding sequences, where there is no CSO when Set_attributes is used, the attribute values become the global default values that are applied when new objects are created. In the second sequence, where there is a CSO, the attribute values apply to the CSO and do not change the global default values. That is, the existence or nonexistence of a CSO, which is a mode, determines the effect of the command. This technique creates a more powerful set of commands without adding any explicit new commands; however, the user must have mode feedback to know how the command will behave. Also, some users are confused by this approach, because it seems inconsistent until the rules are understood.

The general concept of factoring is important for several reasons. First, new users do not need to be concerned with factored parameters that have default values, which improves learning speed. Values for factored parameters do not need to be specified unless the current values are unacceptable, which improves speed of use. Factoring out the object from the command creates the concept of a CSO, a natural one for interactive graphics with its pointing devices. Finally, factoring reduces or eliminates the short-term modes created by prefix commands with multiple parameters. Factoring has been incorporated into a user-interface design tool so that the designer can request that specific parameters be factored; the necessary auxiliary command (Select_object) is introduced automatically [FOLE89].

There are several variations on the CSO concept. First, when an object is created, it does not need to become the CSO if there is already a CSO. Similarly, when the CSO is deleted, some other object (the most recent CSO or an object close to the CSO) can become the new CSO. In addition, a currently selected set (CSS) made of up several selected objects can be used.

9.5 VISUAL DESIGN

The visual design of a user–computer interface affects both the user's initial impression of the interface and the system's longer-term usefulness. Visual design comprises all the graphic elements of an interface, including overall screen layout, menu and form design, use of color, information codings, and placement of individual units of information with respect to one another. Good visual design strives for clarity, consistency, and attractive appearance.

9.5.1 Visual Clarity

If the meaning of an image is readily apparent to the viewer, we have *visual clarity*. An important way to achieve visual clarity is to use the visual organization of information to reinforce and emphasize the underlying logical organization. There are just a few basic visual-organization rules for accomplishing this end. Their use can have a major influence, as some of the examples will show. These rules, which have been used by graphic designers for centuries [MARC80], were codified by the Gestalt psychologist Wertheimer [WERT39] in the 1930s. They describe how a viewer organizes individual visual stimuli into larger overall forms (hence the term *Gestalt*, literally "shape" or "form," which denotes an emphasis on the whole, rather than on the constituent parts).

The visual-organization rules concern similarity, proximity, closure, and good continuation. The rule of *similarity* states that two visual stimuli that have a common property are seen as belonging together. Likewise, the rule of *proximity* states that two visual stimuli that are close to each other are seen as belonging together. The rule of *closure* says that, if a set of stimuli almost encloses an area or could be interpreted as enclosing an area, the viewer sees the area. The *good-continuation* rule states that, given a juncture of lines, the viewer sees as continuous those lines that are smoothly connected.

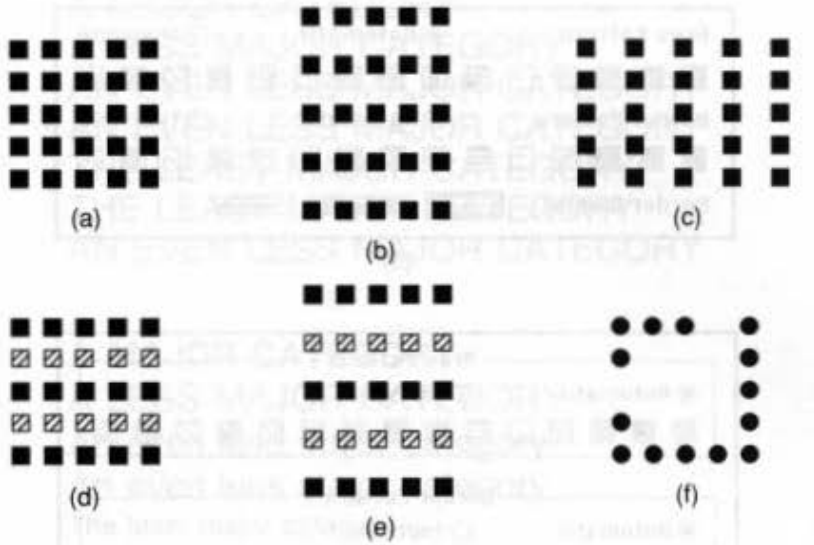


Fig. 9.15 Gestalt rules. In (a), the squares are undifferentiated. In (b), proximity induces a horizontal grouping; in (c), it induces a vertical grouping. In (d), similarity induces a horizontal grouping, which is further reinforced in (e) by a combination of proximity and similarity. In (f), closure induces a square of dots, even though two dots are missing.

Figures 9.15 and 9.16 give examples of these rules and also show how some of them can be combined to reinforce one another. Figure 9.17 shows a form before and after the visual organization rules have been applied. In part (a), everything is near to everything else, so the underlying logical groupings are unclear. Similarity (here, in the sense of being contained in a box) and proximity bind together the patterns and the choice buttons in (b). Closure completes the boxes, which are broken by the label.

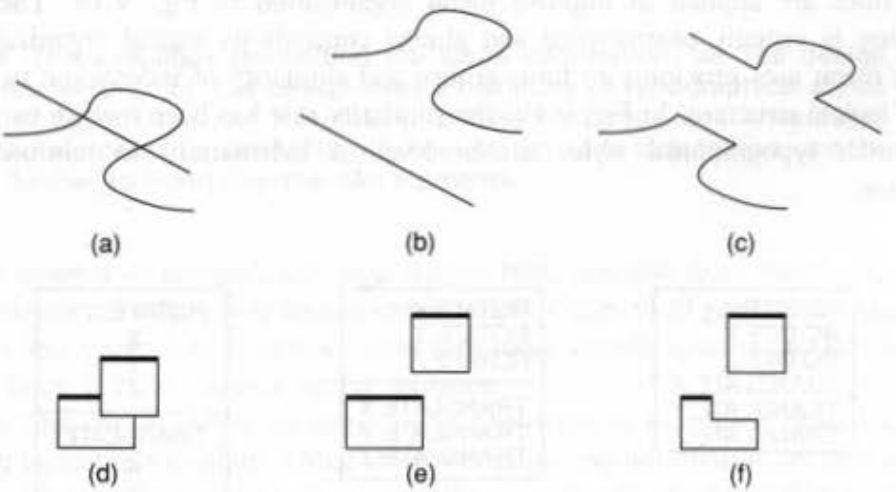


Fig. 9.16 More Gestalt rules. The two intersecting lines in (a) could be interpreted as shown in either (b) or (c). Good continuation favors (b). In a more applied context, the two overlapping windows of (d) could be interpreted as shown in either (e) or (f). Good continuation favors (e).

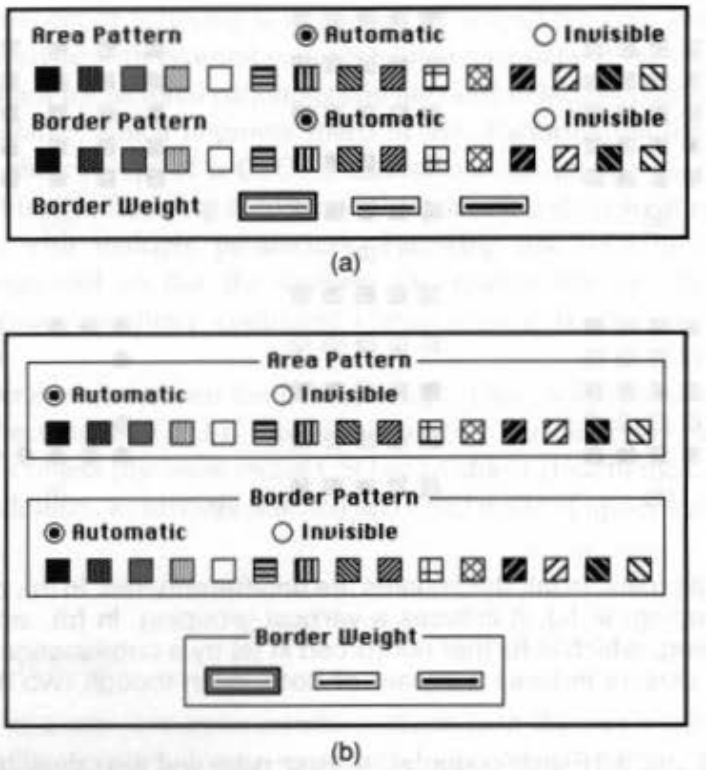


Fig. 9.17 (a) Form laid out with little attention to visual design. (b) Form created using visual grouping and closure to reinforce the logical relationships among the visual elements of the form. (Screen shots © 1983–1989 Microsoft Corporation. Reprinted with permission from Microsoft Corporation.)

The rules are applied to improve menu organization in Fig. 9.18. The leftmost organization is visually unstructured and almost conceals its logical organization. The rightmost menu uses proximity to form groups and similarity of indentation to show the two-level logical structure. In Fig. 9.19, the similarity rule has been used in two different ways (similar typographical style, similar level of indentation) to reinforce logical organization.

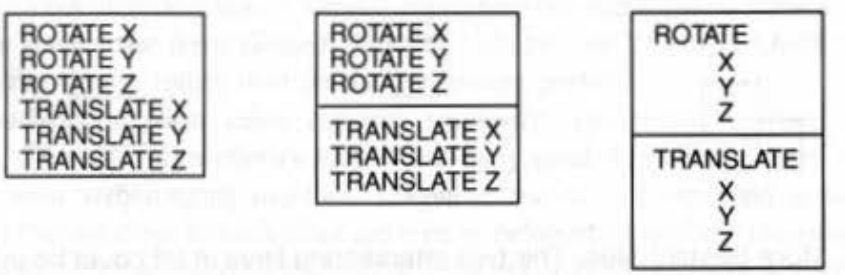


Fig. 9.18 Three designs for the same menu, showing application of visual design techniques.

A MAJOR CATEGORY
A LESS MAJOR CATEGORY
AN EVEN LESS MAJOR CATEGORY
AN EVEN LESS MAJOR CATEGORY
THE LEAST MAJOR CATEGORY
THE LEAST MAJOR CATEGORY
AN EVEN LESS MAJOR CATEGORY

(a)

A MAJOR CATEGORY
A LESS MAJOR CATEGORY
An even less major category
An even less major category
The least major category
The least major category
An even less major category

(b)

A MAJOR CATEGORY
A LESS MAJOR CATEGORY
An even less major category
An even less major category
The least major category
The least major category
An even less major category

(c)

Fig. 9.19 Three designs presenting the same information. (a) The design uses no visual reinforcement. (b) The design uses a hierarchy of typographical styles (all caps boldface, all caps, caps and lowercase, smaller font caps and lowercase) to bond together like elements by similarity. (c) The design adds indentation, another type of similarity, further to bond together like elements.

When ignored or misused, the organization rules can give false visual cues and can make the viewer infer the wrong logical organization. Figure 9.20 gives an example of false visual cues and shows how to correct them with more vertical spacing and less horizontal spacing. Figure 9.21(a) shows a similar situation.

Recall that the objective of using these principles is to achieve visual clarity by reinforcing logical relationships. Other objectives in placing information are *to minimize the eye movements* necessary as the user acquires the various units of information required for a task, and *to minimize the hand movements* required to move a cursor between the parts of the screen that must be accessed for a task. These objectives may be contradictory; the designer's task is to find the best solution.

Fig. 9.20 In (a), the list has a horizontal logical (alphabetical) organization, but a vertical visual organization is induced by the strong proximity relationship. In (b), the alphabetical organization is visually reinforced.

9.5.2 Visual Codings

In interface design, *coding* means creating visual distinctions among several different types of objects. Many different coding techniques are available: color, shape, size or length, typeface, orientation, intensity, texture, line width, and line style are all commonly used in computer graphics. A fundamental issue with any coding technique is to determine how many different categories a particular technique can encode. As more code values are introduced, the possibility of the viewer confusing one value with another increases. The use of a legend, indicating the meaning of each code value, can decrease the error rate.

Many experiments have been conducted to determine how many code values in different coding techniques can be used and still allow almost error-free code recognition (without a legend). For 95-percent error-free performance, 10 colors, 6 area sizes, 6 lengths, 4 intensities, 24 angles, and 15 geometric shapes are the most that can be used [VANC72]. Of course, the code values must be appropriately spaced; see [VANC72, pp. 70–71] for a list of appropriate colors.

If it is important for the viewer to distinguish among different types of information, then it is appropriate to use redundant coding: the use of two different codes to represent the same information. Part (c) of Fig. 9.19 is redundantly coded. Figure 9.22 shows a triply redundant code. Color is normally used redundantly with some other code, to accommodate color-blind users.

Before we can select a code, we must know how many code levels are needed. It is also important to understand whether the information being coded is nominative, ordinal, or ratio. *Nominative* information simply designates, or names, different types of things, such as different types of planes or ships. Nominative information has no notion of greater than or less than. *Ordinal* information is ordered and has a greater than and less than relation. But no metric is defined on ordinal information; there is no notion of varying distances

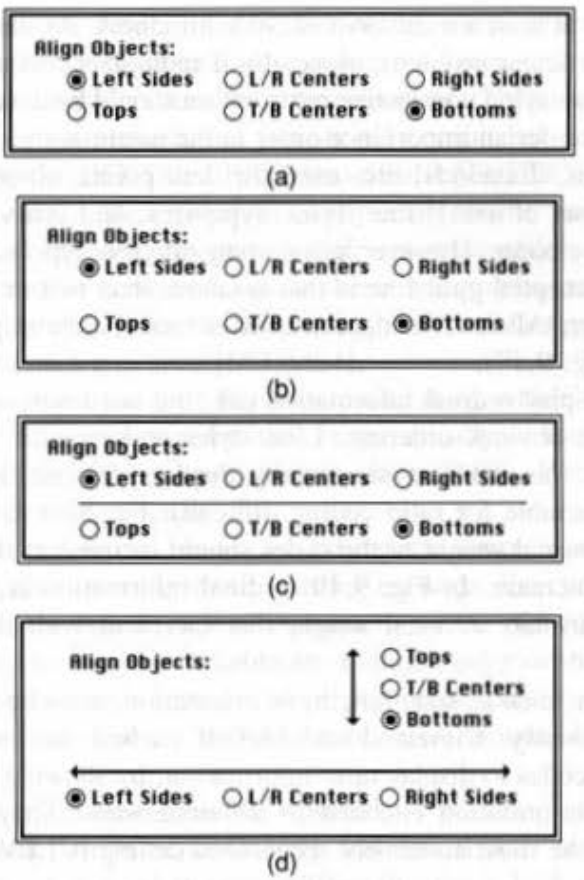


Fig. 9.21 A dialogue box for aligning objects. In (a), the visual cues group the buttons into three groups of two, rather than the proper two groups of three. In (b), the vertical spacing has been increased and the visual cues are correct. In (c), a horizontal rule instead of vertical spacing is used to achieve the same effect in less total space. In (d), the options are rearranged and arrows are used to emphasize the spatial correspondence of each button set to the associated meanings. (Copyright 1988 Claris Corporation. All rights reserved.)

between categories. *Ratio* information has such a metric; examples are temperature, height, weight, and quantity.

For a fixed number of nominative-code values, color is distinguished considerably more accurately than are shape and size, and somewhat more accurately than is intensity [CHRI75]. This suggests that color should be used for coding, but recall that 6 percent to 8

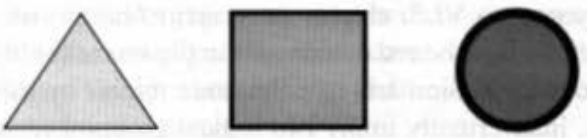


Fig. 9.22 A triply redundant code using line thickness, geometric shape, and interior fill pattern. Any one of the three codes is sufficient to distinguish between the three code values.

percent of males have at least a mild form of color blindness. As discussed in Chapter 13, this need not be a particular problem, especially if redundant coding is used.

Codes used for displaying nominative information should be devoid of any ordering, so that the viewer cannot infer an importance order in the information. Different shapes (such as the squares, circles, diamonds, etc. used for data points when several variables are plotted on the same set of axes), line styles, typefaces, and cross-hatching patterns are appropriate nominative codes. However, using many different typefaces creates a confusing image. A generally accepted guideline is that no more than two or three faces should be used in a single image. Also, differing densities of cross-hatching patterns can create an apparent ordering (Fig. 9.17).

Codes used to display ordinal information can, but need not, vary continuously, but must at least have an obvious ordering. Line styles and area-fill patterns with varying densities can be used, as can text size (many displays provide only a few text sizes, making use of this variable for ratio coding difficult). For both ratio and ordinal information, the apparent visual weight of the codes should increase as the values of the information being coded increase. In Fig. 9.19, ordinal information is being coded, and the typographical hierarchy has a visual weight that increases with the importance of the category.

Ratio information, such as size, length, or orientation, must be presented with a code that can vary continuously. Cleveland and McGill studied the use of several different continuously varying codes to display ratio information, by showing experimental subjects graphs of the same information encoded in different ways. They found the following rankings, where 1 is the most accurately recognized coding [CLEV84; CLEV85]:

1. Position along a common scale
2. Position on identical, nonaligned scales
3. Length
4. Angle between two lines, and line slope
5. Area
6. Volume, density, and color saturation
7. Color hue.

Similarly, Ware and Beatty [WARE88] found that color is effective in grouping objects, but is not effective as a ratio code.

If color were used both to group menu items and to code information in the work area (say, to distinguish layers on a VLSI chip or geographic features on a map), then the user might incorrectly conclude that the red commands in the menu could be applied to only the red elements in the work area. Similarly, a color code might, by using some bright colors and some dark colors, inadvertently imply two logical groupings, one of brighter objects, the other of darker objects. The similarity rule discussed earlier is really at the heart of coding. All like information should be coded with the same code value; all unlike information should have some other code value.

Coding of quantitative data is just a part of the more general field of displaying quantitative data. When data presentations—such as bar, pie, and trend charts—are being designed, many further considerations become important. These are beyond the scope of this text, but are important enough that you should consult key references. The lavishly illustrated books by Bertin [BERT81; BERT83] and Tufte [TUFT83] discuss how to convey quantitative data effectively. Bertin systematically analyzes the visual codes, shows how they can be used effectively, and categorizes different presentation schemes. Tufte argues for minimality in decorative accoutrements to charts and graphs, and for emphasis on the data being conveyed. He also traces the fascinating history of data presentation since 1700. Schmid provides additional guidance [SCHM84].

Mackinlay incorporated some of Bertin's ideas, along with Cleveland and McGill's results into APT, an expert system that automatically creates data presentations [MACK86]. Color Plate I.24 is an example from APT. We expect to see more developments in this promising area.

Closely related to coding are means for calling the viewer's attention to a particular piece of information, such as an error or warning message, the currently selected object, the current command, the failed piece of equipment, or the planes on a collision course. Some attention-getting techniques available are a unique color or shape, a blinking or pulsating or rotating cursor, and reverse video. A unique color was found to be more effective for attracting the viewer's attention than was a unique shape, size, or intensity [CHRI75].

Attention-getting mechanisms can be misused. A pulsating cursor (that is, a cursor whose size continuously varies between large and small) does indeed attract the user's attention. But it also tends to hold attention. When the user is looking at something else on the screen, the pulsating cursor, even though seen only peripherally, is distracting rather than helpful.

Coding of qualitative information is another important research area for user interface design. Work by Feiner and Seligmann [FEIN85; SELI89] explores the automated design of pictures that explain how to perform actions in 3D environments. Based on input about the information the pictures are supposed to communicate and who will be viewing them, a rule-based system determines the objects to include, their properties and rendering style, and the virtual camera parameters that are input to a 3D graphics system that draws the pictures. Color Plate I.25 is an example of a picture generated automatically for a maintenance and repair application.

9.5.3 Visual Consistency

Consistent application of visual-organization rules and codings, and consistent combination of visual elements into higher-level graphic objects and icons, constitute another important element of visual design. Visual consistency is, of course, part of the overall theme of consistency discussed in Section 9.3.1.

Visual elements can be thought of as letters in a graphic alphabet, to be combined into "words" whose meanings should be obvious to the viewer. For instance, dialogue boxes for Macintosh applications are constructed from a small graphic alphabet. Figures 8.16, 8.32,

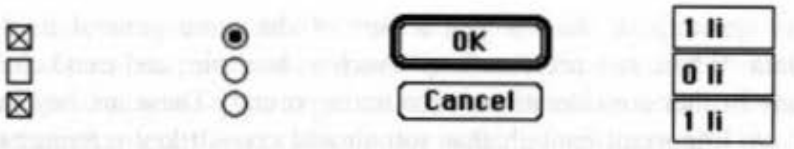


Fig. 9.23 The graphic alphabet used in many Macintosh applications. The square choice boxes indicate alternatives, of which several may be selected at once. The round choice circles, called "radio buttons," indicate mutually exclusive alternatives; only one may be selected. The rounded-corner rectangles indicate actions that can be selected with the mouse. In addition, the action surrounded by the bold border can be selected with the return key on the keyboard. The rectangles indicate data fields that can be edited. (© Apple Computer, Inc.)

9.14, 9.17, and 9.21 are examples of these dialogue boxes, and Fig. 9.23 shows their graphic alphabet. Similarly, Fig. 9.24 shows the use of a small graphic alphabet to build icons, and Fig. 9.25 shows a single-element graphic alphabet.

Consistency must be maintained among as well as within single images; a consistent set of rules must be applied from one image to another. In coding, for example, it is unacceptable for the meaning of dashed lines to change from one part of an application to another. For placement consistency, keep the same information in the same relative position from one image or screen to the next, so that the user can locate information more quickly.

9.5.4 Layout Principles

Individual elements of a screen not only must be carefully designed, but also, to work together, must all be well placed in an overall context. Three basic layout rules are balance,

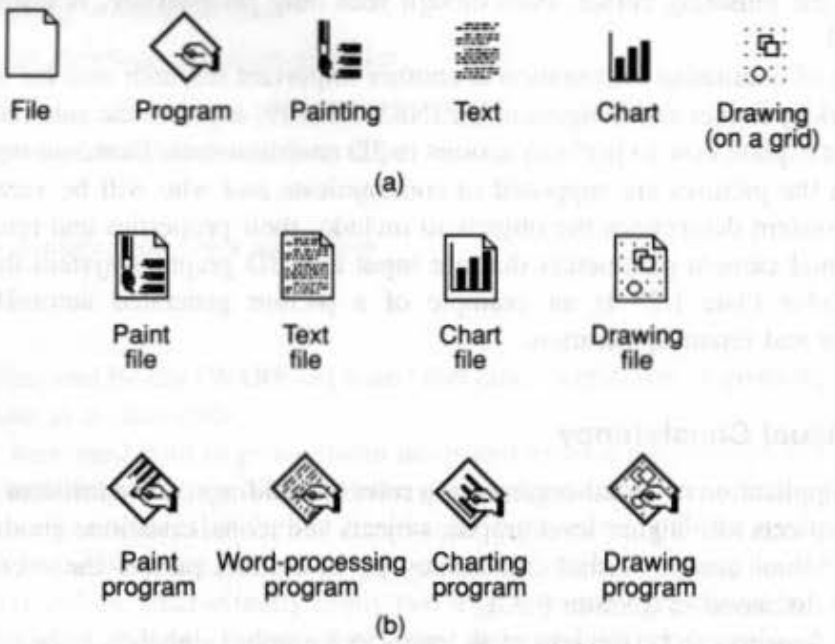


Fig. 9.24 (a) A graphics alphabet. (b) Icons formed by combining elements of the alphabet.

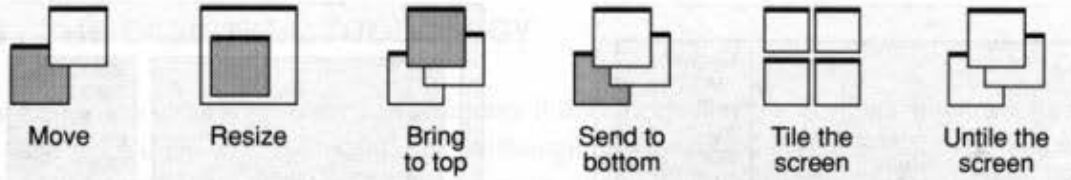


Fig. 9.25 Several different icons, all created from a single shape representing a window.

gridding, and proportion. Figure 9.26 shows two different designs for the same screen. Design (a) is *balanced*, nicely framing the center and drawing the eye to this area. Design (b) is unbalanced, and unnecessarily draws the eye to the right side of the area. Design (b) also has a slight irregularity in the upper right corner: the base lines of the scroll bar arrow and the pointer icon are not quite aligned. The eye is needlessly drawn to such meaningless discontinuities.

Figure 9.27 shows the benefits of using empty space between different areas, and also illustrates the concept of *gridding*; in cases (b) and (c), the sides of the three areas are all aligned on a grid, so there is a neatness, an aesthetic appeal, lacking in (a) and (d). Figure 9.28 further emphasizes the detrimental effects of not using a grid. [FEIN88] discusses an expert system that generates and uses design grids.

Proportion deals with the size of rectangular areas that are laid out on a grid. Certain ratios of the lengths of a rectangle's two sides are more aesthetically pleasing than are others, and have been used since Greco-Roman times. The ratios are those of the square, which is 1:1; of the square root, 1:1.414; of the golden rectangle, 1:1.618; and of the double square, 1:2. The double square is especially useful, because two horizontal double squares can be placed next to a vertical double square to maintain a grid. These and other design rules are discussed in [MARC80; MARC84; PARK88].

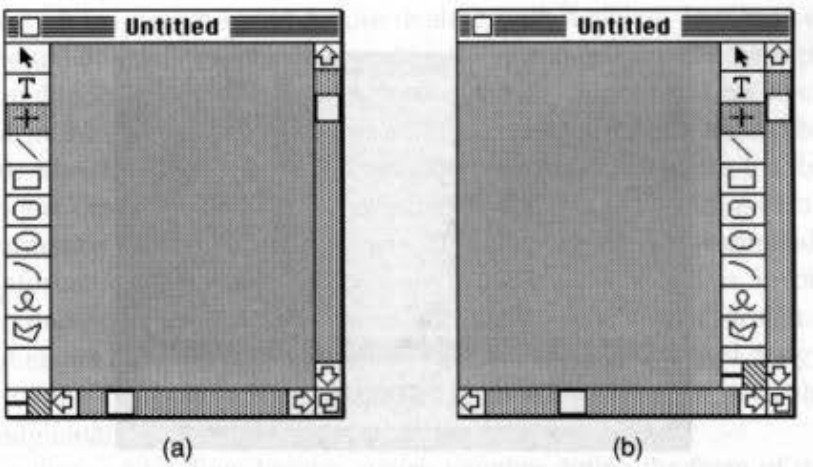


Fig. 9.26 Two alternative screen designs. Design (a) is balanced; design (b) emphasizes the right side. (Copyright 1988 Claris Corporation. All rights reserved.)

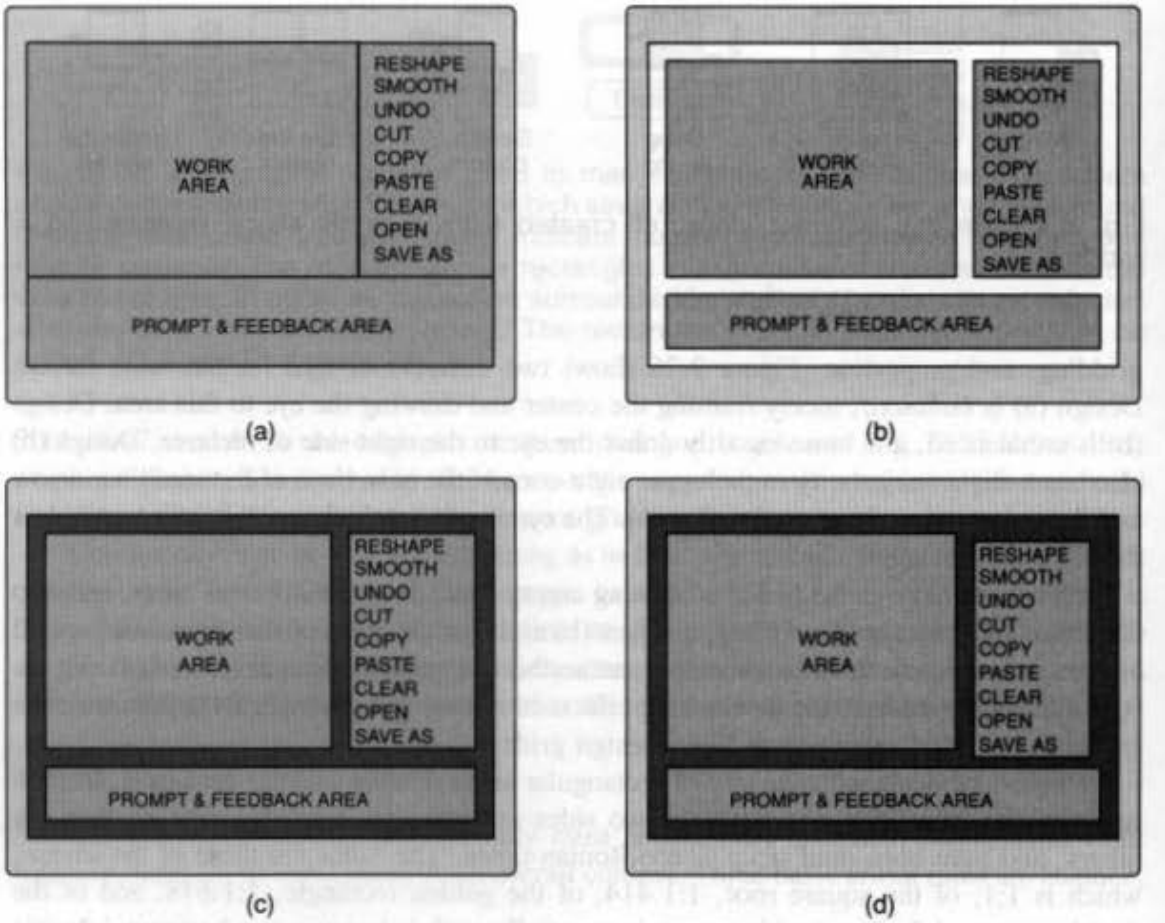


Fig. 9.27 Four screen designs. (a) A typical initial design. (b) Border area has been added. (c) The border has been strengthened to separate the three areas further. (d) The deleterious effect of not aligning elements on a grid is obvious.

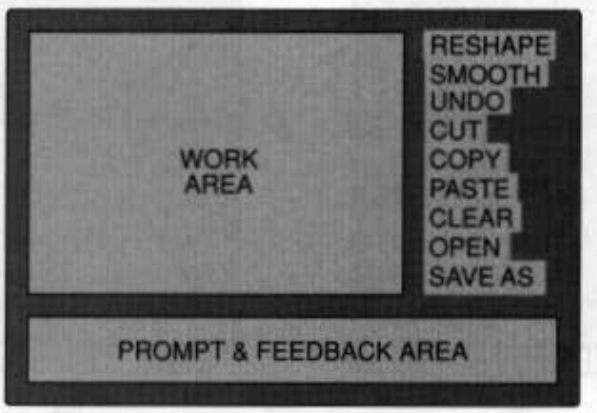


Fig. 9.28 Removing the box around the menu area creates the meaningless but attention-getting ragged-right border.

9.6 THE DESIGN METHODOLOGY

Many ideas have been presented in Chapters 8 and 9. How can a designer integrate them and work with them in a structured way? Although user-interface design is still in part an art rather than a science, we can at least suggest an organized approach to the design process. In this section, we give an overview of the key elements of such a methodology.

The first step in designing an interface is to decide what the interface is meant to accomplish. Although at first this statement may seem trite, poor requirements definitions have doomed numerous user-interface design projects at an early stage. Understanding user requirements can be accomplished in part by studying how the problem under consideration is currently solved. Another successful approach is for the designer to learn how to perform the tasks in question. The objective is to understand what prospective users currently do, and, more important, why they do it.

We do not mean to imply that the interface should exactly mimic current methods. The reason for understanding why prospective users work as they do is often to develop new and better tools. We should recognize, however, that it is sometimes better to mimic old ways to avoid massive retraining or to avoid morale problems with an existing workforce. A typical strategy is first to mimic existing methods, and also to make new methods available; over time, users can be trained in the new or augmented capabilities.

User characteristics must also be identified. What skills and knowledge do the users have? Are the users knowledgable about their work but computer-naive? Are they touch-typists? Will the users typically be eager to learn the system, or will they be reluctant? Will usage be sporadic or regular, full-time or part-time? It is important when assessing the user population to remember that what you, the system designer, would want or like is not necessarily the same as what those for whom the system is being designed might want. Your users are not necessarily created in your own image.

When the requirements have been worked out, a top-down design is next completed by working through the design levels discussed in Section 9.1: conceptual, functional, sequencing, and binding. The rationale for top-down design of user interfaces is that it is best to work out global design issues before dealing with detailed, low-level issues.

The conceptual design is developed first. Ideally, several alternative conceptual designs are developed and evaluated on the basis of how well they will allow users to carry out the tasks identified in the requirements definition. High-frequency tasks should be especially straightforward. Simplicity and generality are other criteria appropriate for the designs.

The functional design focuses on the commands and what they do. Attention must be paid to the information each command requires, to the effects of each command, to the new or modified information presented to the user when the command is invoked, and to possible error conditions. Figure 9.29 shows the focus of functional design. Notice the notations about errors being "engineered out" by subsequent lower-level design decisions. One objective of the functional design is to minimize the number of possible errors, by defining the individual commands appropriately.

The sequencing and binding designs, which together define the form of the interface, are best developed together as a whole, rather than separately. The design involves first selecting an appropriate set of dialogue styles, and then applying these styles to the specific

Function:	Add_symbol_instance
Parameters:	Symbol_identifier Symbol_position
Description:	An instance of the symbol is created and is added to the figure at the desired position. The instance becomes the currently selected object (CSO). The previous CSO, if any, is no longer selected.
Feedback:	The instance is seen on the display and is highlighted because it is selected. (If there was a CSO, it is no longer highlighted.)
Errors:	1. The Symbol_identifier is unknown (engineered out by use of a menu selection to choose symbol). 2. The Symbol_position is outside the viewport (engineered out by constraining the positioning device feedback to be within the viewport).

Fig. 9.29 A typical functional specification for a command. The annotations with the errors are added after interaction techniques are identified as part of designing the form of the interface.

functionality. Sequences of screens, sometimes called *storyboards*, can be used to define the visual and some of the temporal aspects of these designs. State diagrams, as discussed in Section 9.3.1 and in Chapter 8, are also helpful in detailing user-action sequences.

The interface form can be defined by a *style guide*, a written codification of many of the elements of user-interface form. The most common motivation for developing a style guide is to ensure a "look and feel" consistency within and among applications. Some elements of the style guide can be implemented in libraries of interaction techniques (Chapter 10); other elements must be accommodated by the designers and programmers. Many style guides exist, among them guides for the Macintosh [APPL87], Open Software Foundation's OSF/MOTIF [OPEN89], NASA's Transportable Application Executive [BLES88b], and DEC's XUI [DIGI89].

The whole design process is greatly aided by interleaving design with user-interface prototyping. A user-interface prototype is a quickly created version of some or all of the final interface, often with very limited functionality. The emphasis is on speedy implementation and speedy modifiability. At the start, some design questions will seem to be unanswerable; once a prototype is available, however, the answers may become apparent. Prototyping is often superior to using a design document, since it gives users a more specific frame of reference, within which they can talk about their needs, likes, and dislikes. HyperCard and Smalltalk are used extensively for rapid prototyping, as are some of the software tools discussed in Section 10.6.

Prototyping can begin as soon as a conceptual design is worked out, and the elements of the functional design and dialogue style can be developed concurrently. It is important to

follow the Bauhaus dictum, *form follows function*, lest the user-interface style dictate the capabilities of the overall system. As soon as even some modest elements of the interface are developed, potential users should be exposed to them, to elicit suggestions for improvements to the interface. As modifications are made and as the prototype becomes more comprehensive, users should again work with the system. This iterative cycle has come to be viewed as essential to the development of high-quality user-interface software. Further discussion of prototyping and iterative development can be found in [ALAV84; HART89].

EXERCISES

9.1 Determine how several commands in an interactive graphics application program with which you are familiar could be made into direct-manipulation operations.

9.2 Examine several interactive graphics application programs and characterize their dialogue style. List the ways in which the interfaces do and do not follow the design guidelines discussed in this chapter. Identify the design level for each point you list; for instance, consistent use of color is at the hardware-binding level.

9.3 The conceptual model for many word processors is partially based on an analogy with typewriters. List ways in which this analogy might create difficulties for a user who attempts to carry it further than is realistic.

9.4 Analyze a user interface to determine what methods, if any, are provided for error correction. Categorize the methods according to the four types discussed in Section 9.3.4.

9.5 What is the form of the state diagram representing a completely modeless interface?

9.6 Design and implement a simple graphics editor with the following functionality: create, delete, and move lines; move endpoints of lines; change the line style (dash, dotted, solid) of existing lines; set the line-style mode for lines that have not yet been created. Design this system to support two or more of the following five syntaxes, and include a command to switch from one syntax to another: object mode, command mode, object mode with Repeat_last_operation operation, free-form syntax, and free-form syntax with modes and a Do_it command.

9.7 Conduct either informal or formal controlled experiments with each of the syntaxes implemented in Exercise 9.6. Test ease of learning and speed of use. Give your users five predefined tasks to perform. Identify tasks that will be faster to perform with one syntax than with the others.

9.8 Study three different interactive graphics application programs.

a. Identify the classes of modes and syntaxes used in each, using the definitions in Section 9.4. A single application may have several types of modes or syntaxes. If so, are there clear distinctions showing which to use when?

b. Identify the factoring, if any, that has occurred. Are there additional possibilities for factoring? Do you think the user interface would be improved by application of the factoring?

9.9 Consider the three window organizations shown in Fig. 9.30. Count the number of mouse movements and mouse-button depressions needed to move the lower-right corner. In each case, assume that the cursor starts in the center of the window and must be returned to the center of the window. Counting mouse movements as 1.1 seconds and button depressions as 0.2 seconds, how long does each window organization take? Does this result mean that one organization is better than the others?

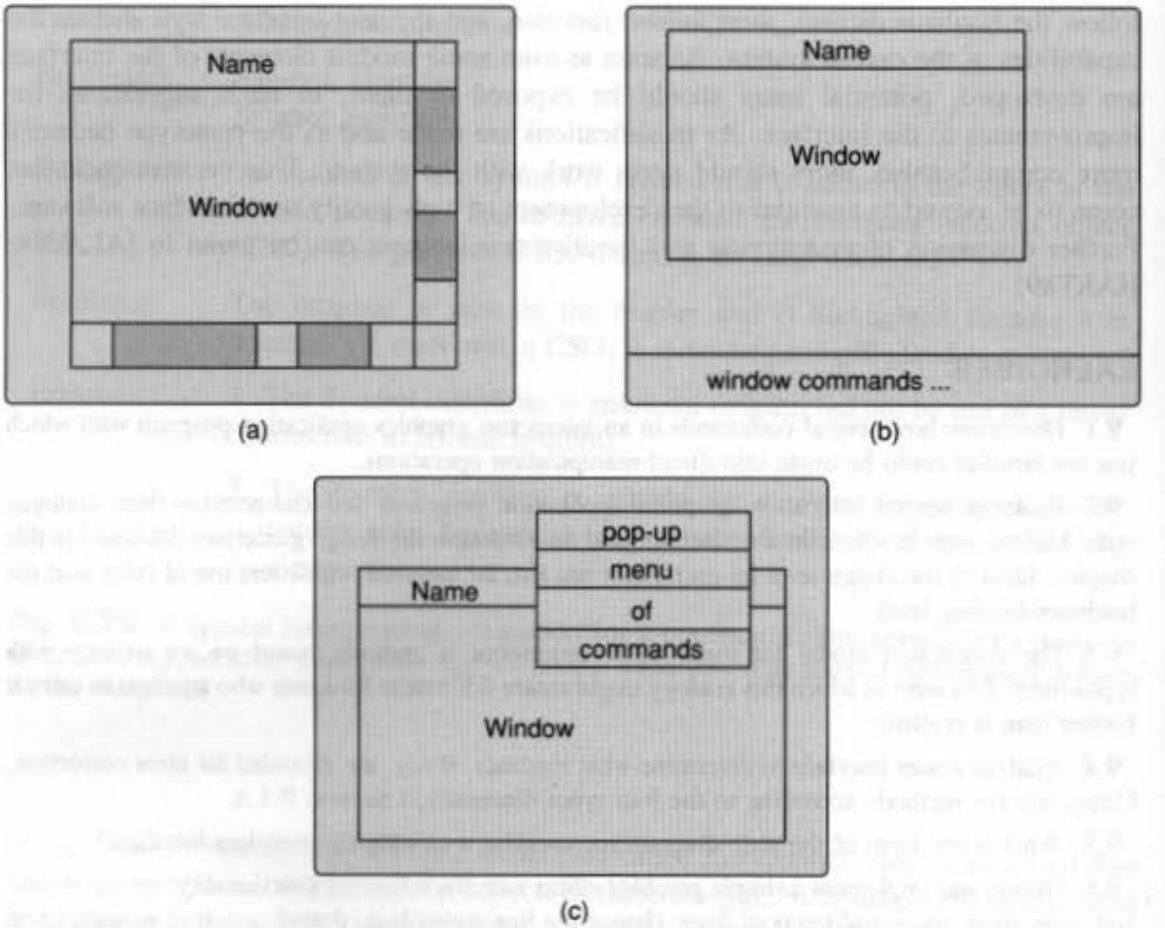


Fig. 9.30 Three means to invoke window commands. In (a) (Macintosh style), regions of the window dressing are used. To resize, the user selects and drags the lower-right corner. In (b), the commands are in the command area at the bottom of the screen. To resize, the user selects a command, and then drags the lower-right corner of the window. In (c), a pop-up menu appears at the cursor when a mouse button is pressed. To resize, the user selects the command from the pop-up menu, and drags the lower-right corner of the window.

9.10 Implement a single-level Undo command for an interactive graphics application program you have written. Decide which of several implementation strategies you will use, and justify your choice. The strategies include (1) after each command, make a complete copy of the application data structure, state variables, and so on; (2) save a record of what was changed in the application data structure; (3) save all the commands since logon, replay them to effect undo; (4) save the application data structure every 10 minutes, plus save all commands since the last such save operation; and (5) save, for each user command, the one or several commands needed to undo the user command. Would your choice differ for a multi-level Undo command?

9.11 Examine an interactive graphics application. How many commands does it have? List the "starter kit" of commands with which a new user can do a simple piece of work. How big is the starter kit with respect to the overall command set? List the ways in which defaults or other methods have been used to minimize the complexity of the starter kit. In your opinion, does the system have a good starter kit?

9.12 Explore other ways of redesigning Fig. 9.17(a) to reinforce the logical relationships of the visual elements.

9.13 Study the dialogue-box design of an application. Which methods are used to reinforce logical structure with visual structure? Can you further improve the design?

9.14 Figure 9.6 shows 12 icons that represent Macintosh programs: (a) disk copy utility, (b) resource mover, (c) icon editor, (d) menu editor, (e) alert/dialog editor, (f) edit program, (g) boot configure, (h) switcher, (i) examine file, (j) MacWrite, (k) MacDraw, and (l) MacPaint. Some of the icons indicate the associated program better than others do.

a. Design an alternative set of icons to represent the 12 programs.
b. Show the set of icons to 10 programmers who are not familiar with the Macintosh, and ask them to guess what each icon means. Tabulate the results.
c. Tell 10 programmers who are not familiar with the Macintosh what each icon means. Give them 2 minutes to study the icons. Ten minutes later, show them the icons again, and ask them what the icons mean. Tabulate the results.
d. Repeat parts (b) and (c) for the icons in the figure. What conclusions can you draw from your data?

9.15 Analyze the visual design of a graphics application. What is the visual alphabet? What visual codings are used? What visual hierarchies are established? Redesign some of the visual cues to emphasize further the underlying logical relationships.

9.16 List 10 specific examples of codings in computer graphics applications you have seen. Is the coded information nominative, ordinal, or ratio? Are the code values appropriate to the information? Are there any false codings?

9.17 Examine three different window managers. What visual code indicates which window is the "listener"—that is, the window to which keyboard input is directed? What visual code indicates which processes are active, as opposed to blocked? What other visual codings are used?

10 User Interface Software

The first two chapters on user interfaces, Chapters 8 and 9, concentrated on the external characteristics of user–computer interfaces. Here, we examine the software components, beyond the basic graphics packages already discussed, that are used in implementing interfaces. Figure 10.1 shows the various levels of user-interface software, and suggests the roles for each. The figure shows that the application program has access to all software levels; programmers can exploit the services provided by each level, albeit with care, because calls made to one level may affect the behavior of another level. The operating-system level is not discussed in this text, and the basics of graphics subroutine packages have already been described. Some input features of device-independent graphics subroutine packages are compared and evaluated in Section 10.1. Window-management systems, discussed in Sections 10.2 to 10.4, manage the resources of screen space and interaction devices so that several applications or multiple views of the same application can share the display. Some window-management systems have an integral graphics subroutine package that provides device-independent abstractions, whereas others simply pass graphics calls through to the underlying graphics hardware or software.

The interaction techniques discussed in Chapter 8 are useful in many applications, but require careful development to provide a pleasing look and feel. Interaction-technique toolkits, treated in Section 10.5, are built on window-management systems to give the application developer a common set of techniques. The final layer, the user-interface management system (UIMS), discussed in Section 10.6, provides additional generic user-interface support at the sequencing level of design (Section 9.1). UIMSs speed up implementation of a user interface, and facilitate making rapid changes during the interface-debugging process discussed in Section 9.6.

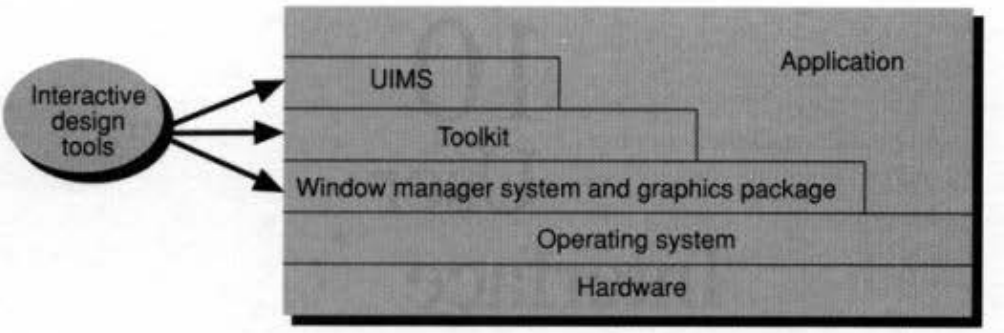


Fig. 10.1 Levels of user-interface software. The application program has access to the operating system, window-manager system and graphics package, toolkit, and user-interface management system (UIMS). The interactive design tools allow nonprogrammers to design windows, menus, dialogue boxes, and dialogue sequences.

10.1 BASIC INTERACTION-HANDLING MODELS

In this section, we elaborate on the interaction-handling capabilities of contemporary device-independent graphics subroutine packages, as introduced in Chapters 2 (SRGP) and 7 (SPHIGS). The sampling and event-driven processing in these two packages is derived from GKS [ANSI85b; ENDE86] and PHIGS [ANSI88], which share a common interaction model. Window-management systems use an event mechanism similar to, but more powerful than, the GKS/PHIGS model discussed in this section.

GKS and PHIGS have six classes of logical input devices, and there may be more than one device of each class associated with a workstation (a display and associated interaction devices). Each of the logical input devices can operate in one of three modes: sample, request, and event. Each device has an associated *measure*, which is the type of information returned by the device. The devices and their measures are as follows:

Device	Measure
locator	position in world coordinates
pick	pick path for SPHIGS, segment identification for GKS
choice	integer indicating the choice
valuator	real number
string	character string (called keyboard device in SRGP and SPHIGS)
stroke	sequence of positions in world coordinates

SRGP and SPHIGS use measures that are slightly different from these.

In *request mode*, the application program requests input from a device, and the graphics package returns control and the measure of the device only after the user has performed an action with the device. The action is called the *trigger*. The specific trigger action for each logical device class is implementation-dependent, but is typically a button-push. For instance, a mouse button triggers locator or pick devices, and the return key triggers the string device.

Request mode can be used with only one device at a time, and is intended to support the limited functionality of older graphics terminals, which are typically connected to computers via RS-232 interfaces. Interaction techniques such as keyboard accelerators

cannot be used, because the application program must know in advance from which device to request input. In addition, as the measure is modified (by moving the mouse to change the locator's measure, say), the application program generally cannot provide dynamic feedback, because the application program does not regain control until the trigger action occurs. This difficulty can be eliminated by defining the trigger action as a small change in the measure.

In *sample mode*, a single device is sampled, and the measure of the device is immediately returned. We can permit the user to select one of several devices to use, by polling all eligible devices, as follows:

```
terminate = FALSE;
while (!terminate) {
    SamplePick (&status, &segmentName);
    /* status = OK means had successful pick; segmentName is identification */
    /* of picked item */
    Process pick input
    SampleString (string);
    Process string input
    /* terminate set to TRUE as part of processing string or pick */
}
```

Sampling in this way is dangerous, however. If the user makes several more inputs while the first is being processed, they will never be seen by the application program, since it stops sampling while processing the first input. Also, the sequence of user events, which is often essential to maintain, might be lost in sampling. Unlike request mode, however, sample mode is well suited for dynamic feedback from the application program, because no trigger action is required to return the device measure to the application program.

Event mode avoids the problems of sample and request modes, by allowing input to be accepted asynchronously from several different devices at once. As discussed in Section 2.3.6 of Chapter 2, the application program first enables all devices whose use is to be permitted. Once the devices are enabled, a trigger action for any of them places an event report on an input queue, in order of occurrence. As seen in Fig. 10.2, the application

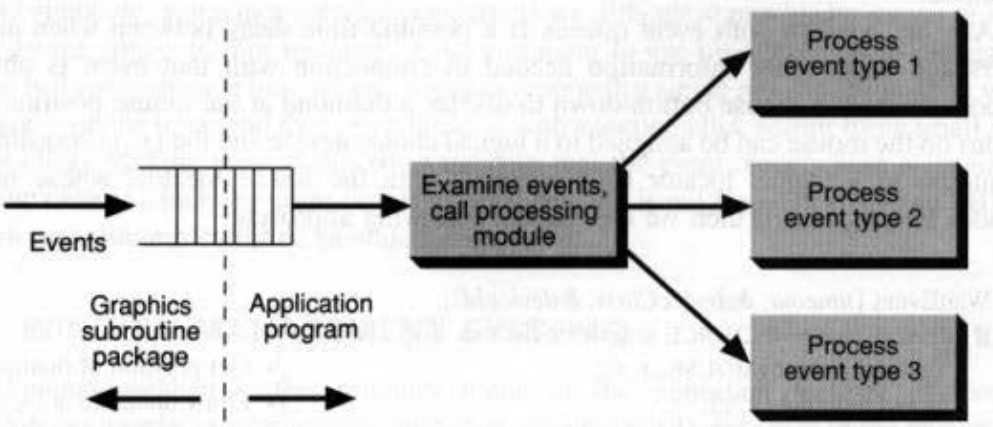


Fig. 10.2 The application program removes events from the queue and dispatches control to the appropriate procedure, which processes them.

program checks the queue to see what user actions have actually occurred, and processes the events as they are removed from the queue.

The following code fragment reimplements the previous polling example in event mode:

```
terminate = FALSE;
while (!terminate) {
    WaitEvent (timeout, &deviceClass, &deviceId);          /* Wait for user action */
    switch (deviceClass) {
        case pick: Process pick;
            break;
        case string: Process string;
            break;
    } /* terminate set to TRUE in processing of pick or string */
}
```

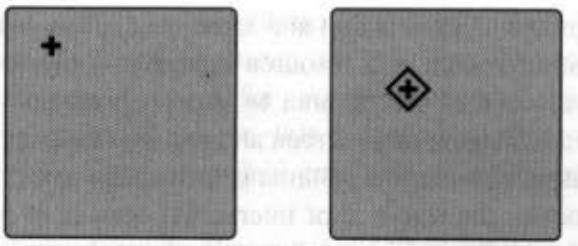
Unlike request-mode input, event-mode input is asynchronous: Once a device is enabled, the application program can be executing while the user is concurrently inputting information with the enabled input devices. This is sometimes called *typeahead*, or, when done with a mouse, *mouseahead*.

The typeahead capability of the event-queue mechanism provides an opportunity to speed up interactions with the computer. Suppose a button-press (choice logical device) is used to scroll through a drawing. Each button-press scrolls, say, x inches. If the user presses the button more rapidly than the scroll can occur, events build up in the queue. The application program can look for multiple successive button events on the queue; if there are n such events, then a single scroll of nx inches can be performed, and will be much faster than n scrolls of x inches each.

Care must be taken to manage the event queue properly. If the first of two events on the queue causes the program to enable a different set of logical input devices and then to call WaitEvent, the program now may not be expecting the second event, leading to unexpected results. The call FlushDeviceEvents is provided to alleviate this problem; the application program can empty the event queue to ensure that the queue contains nothing unexpected. However, flushing the queue may leave the user wondering why the second event was never processed.

Another concern with event queues is a possible time delay between when an event occurs and when other information needed in connection with that event is obtained. Suppose we want a mouse button-down to display a diamond at the mouse position. If the buttons on the mouse can be assigned to a logical choice device and the (x, y) coordinates of the mouse to a logical locator (this contrasts with the SRGP locator, whose measure includes button status), then we can use the following approach:

```
WaitEvent (timeout, &deviceClass, &deviceId);
if (deviceClass == CHOICE && deviceId == 1) {
    SampleLocator (MOUSE, x, y);                    /* Get position of diamond */
    DrawDiamond (x, y);                             /* Draw diamond at (x, y) */
}
```



At time t_1, the user makes a choice, with a choice device, to draw a diamond. The user then moves cursor to go on to another task.

At time t_2, locator is sampled and diamond is drawn. Locator has moved between t_1 and t_2.

Fig. 10.3 The effect of a time delay between an event and sampling of the cursor position associated with that event.

The problem is that, between time t_1 (when the WaitEvent procedure returns) and time t_2 (when the SampleLocator procedure obtains the (x, y) coordinates), seconds may have elapsed. In this time, the user may have moved the locator some distance, causing the unexpected result shown in Fig. 10.3. Substantial delays can easily occur on a time-shared computer if another program takes control of the processor, and on any computer that supports virtual memory when page-fault interrupts occur. Thus, although we would like the application program to be uninterruptible during the interval from t_1 to t_2, we cannot guarantee that it will be so.

In GKS and PHIGS, the risk of this time delay can be reduced and even eliminated by activating several logical devices with the same trigger. If we define the trigger to be a button-click on any of the three buttons, and associate this trigger with both the three-button choice device and the locator device, then both events will be placed on the queue (in unspecified order) at the same time. The device-dependent driver under the graphics package can do this faster than the application program can execute the preceding code segment, so the likelihood of being interrupted is less. If the operating system grants the device driver the privilege of disabling interrupts, then there will never be a time delay.

Unfortunately, some user-interface constructs are difficult to provide by means of these logical-device concepts. For instance, it is convenient to use time intervals to distinguish between two commands. Thus, we can select an icon with a single mouse button-click while the cursor is on the icon, and then open the icon with a second click within some small Δt of the first click. Making these distinctions requires that the event reports have a *timestamp* giving the time at which the event occurred. This concept is not found in GKS and PHIGS, although a timestamp could be provided readily.

10.2 WINDOW-MANAGEMENT SYSTEMS

A *window-management system* provides many of the important features of modern user-computer interfaces: applications that show results in different areas of the display, the ability to resize the screen areas in which those applications are executing, pop-up and pull-down menus, and dialogue boxes.

The window-management system is first and foremost a resource manager in much the same way that an operating system is a resource manager—only the types of resources differ. It allocates the resource of screen area to various programs that seek to use the screen, and then assists in managing these screen areas so that the programs do not interfere with one another. This aspect of window systems is further discussed in Section 10.3. The window system also allocates the resource of interaction devices to programs that require user input, and then routes the flow of input information from the devices to the event queue of the appropriate program for which the input is destined. Input handling is discussed further in Section 10.4.

Our objective with these three sections is to provide an overview of key window-management concepts: The most comprehensive treatment of the subject is [STEI89], and a historical development overview is given in [TEIT86], in a book of relevant papers [HOPG86b].

A window-management system has two important parts. The first is the *window manager*, with which the end user interacts to request that windows be created, resized, moved, opened, closed, and so on. The second is the underlying functional component, the *window system*,which actually causes windows to be created, resized, moved, opened, closed, and so on.

The window manager is built on top of the window system: The window manager uses services provided by the window system. The window manager is to its underlying window system as a command-line interpreter is to its underlying operating-system kernel. Also built on top of the window system are higher-level graphics packages and application programs. The programs built on the window system are sometimes called *client* programs, which in turn use the capabilities of the window system, itself sometimes called the *server* program. In some server–client window-management systems, such as the X Window System [SCHE88a] and NeWS [SUN87], the window manager itself appears to the window system as just another client program. In other systems, there is a closer relationship between the window manager and window system than there is between a client and server.

Some window systems, including the X Window System and NeWS, are designed to be *policy-free,* meaning that multiple window managers, each with a different look and feel, can be built on top of the window system. The window manager, not the window system, determines how windows look, and how the user interacts with windows. A policy-free window system would support all the window styles of Fig. 9.30, as well as others. Just as many different application programs can be built on top of a graphics package, many different window managers can be built on top of a policy-free window system: The window manager and graphics application program both control external appearance and behavior. For this approach to be possible, the window system must be designed to carry out a wide range of window-manager policies (see Exercise 10.10). Of course, in a specific environment, the window manager and application programs need to have a common user-interface look and feel.

If the programming interface to the window system is cleanly defined and is implemented via an interprocess communication capability, then clients of the window system can reside on computers different from that of the window system, provided the computers are connected by a high-speed network. If the window manager is

itself just another client of the window system, then it too can reside on another computer. The use of interprocess communications in this way allows computation-intensive applications to reside on a powerful computer, while the user interacts with the application from a workstation. In this regard, the server–client model is just a sophisticated instance of a virtual terminal protocol; such protocols in general share this advantage.

A window-management system does not need to be built on the server–client model. For instance, the Macintosh has no well-defined separation between the window manager and window system. Such separation was not necessary for the single-active-process, single-processor design objective of the Macintosh, and would have led to additional run-time overhead.

In window systems that provide for use of interprocess communications between the window manager and window system, such as the X Window System, NeWS, and Andrew [MORR86], the interface must be designed to minimize communications delays. Several strategies can help us to meet this objective. First, asynchronous rather than synchronous communications can be used between the client and server whenever possible, so that, when the client sends a message to the server, the client does not need to wait for a reply before resuming processing and sending another message. For example, when the client program calls DrawLine, a message is sent to the server, and control returns immediately to the client.

We can sometimes realize a modest savings by minimizing the number of separate messages that must be sent between the server and client. Most network communication protocols have a minimum packet size, typically 16 to 32 bytes, and the time to send larger numbers of bytes is often proportionately less than the time needed to send the minimum packet. There is thus an advantage in batching messages, as provided in some systems with a BeginBatchOfUpdates and EndBatchOfUpdates subroutine call; all the calls made between the BeginBatch and EndBatch calls are transmitted in one message. There is also an advantage in designing single messages that replace multiple messages, such as a single message that sets multiple graphics attributes.

A third way to minimize communication is to move more functionality and generality into the server. The commands most clients send to their window system are fairly primitive: draw line, create window, copy pixmap. In the X Window System, for instance, many commands are needed to create a menu or dialogue box. A more robust and powerful strategy is to send commands to the server as programs written in a language that can be interpreted efficiently. Thus, the commands can be very general and can carry out any functionality the language can express. The cost of this generality is, of course, the time taken to execute the programs interpretively, and any additional space needed for the interpreter—a modest price with contemporary computers. The benefit can be a dramatic decrease in communications traffic. The strategy of moving more generality and functionality into the workstation is not new; exactly the same issues were discussed two decades ago when distributed graphics systems were first being built [FOLE71; FOLE76; VAND74].

This third strategy is used in the NeWS window system, which accepts as commands programs written in an extended version of the PostScript language [ADOB85a; ADOB85b]. PostScript combines traditional programming-language constructs (variables,

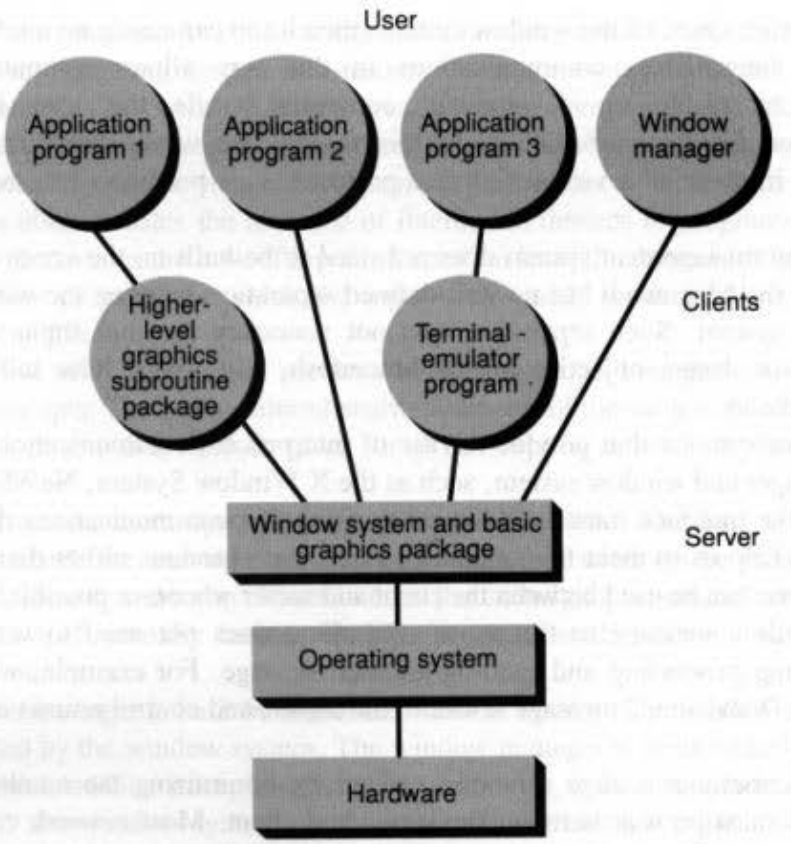


Fig. 10.4 The relationship of the window system to the operating system and application programs.

data structures, expressions, assignments, control flow, I/O) with imbedded graphics capabilities for drawing output primitives, clipping against arbitrary regions, and transforming primitives. NeWS adds extensions for processes, input, and windows. The language is further discussed in Section 19.9.

For a dialogue box to be defined in NeWS, a PostScript program defining the box is sent to the server when a program begins execution; each time the dialogue box is to appear, a short message is sent to invoke the program. This strategy avoids resending the box's definition each time. Similarly, programs to perform time-critical operations, such as rubberband drawing of lines (Chapter 8) or curves (Chapter 11), can be sent to the server, avoiding the time delays involved in each round-trip message between the server and client needed to update the rubberband line.

A graphics package is often integrated with the window system, typically a 2D nonsegmented package with capabilities similar to SRGP of Chapter 2. If the underlying hardware has 3D or segmentation capabilities, then the window-system level might pass on graphics calls to the hardware. Figure 10.4 shows how the window system and its graphics package typically relate to other system components; Fig. 10.5 shows the relationships among windows, clients, and input events. User-generated events involving windows—resizing, repositioning, pushing, popping, scrolling, and so on—are routed by the window system to the window manager; other events are routed to the appropriate application program.

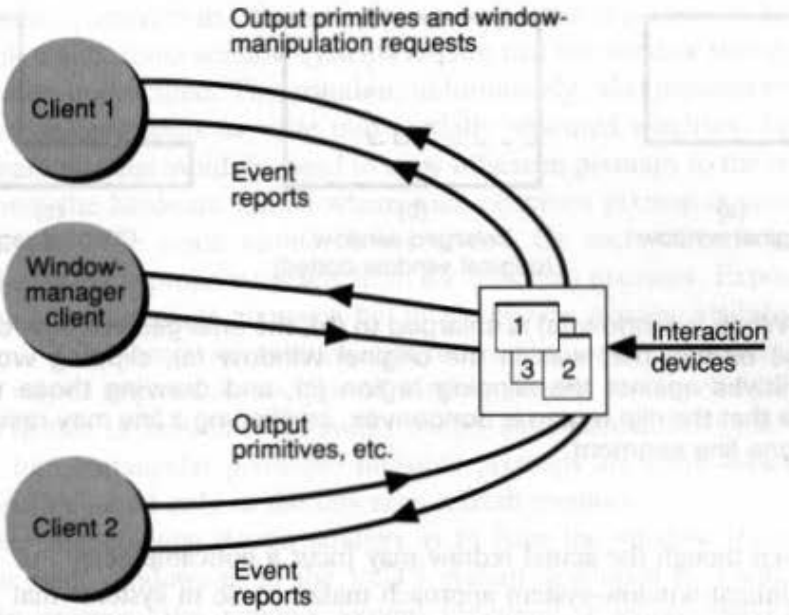


Fig. 10.5 Another view of the relationship among windows, clients, and events. Each client outputs to a window; input events are routed to the appropriate client's event queue.

10.3 OUTPUT HANDLING IN WINDOW SYSTEMS

The output resource allocated by the window system to its client programs is screen space, which must be managed so that clients do not interfere with one another's use of the screen. The strategies by which this allocation is made vary considerably from one window system to another, but fit into three broad categories. The main difference is in how parts of a window that have just been exposed (when the window is made larger, uncovered, or scrolled) are displayed. The strategies place progressively more responsibility for making this decision on the window system itself, such that the client is progressively less aware of the existence of the windows. The system may also have to manage a look-up table so as to avoid conflicts between clients.

A minimal window system takes no responsibility for drawing newly exposed portions of a window; rather, it sends a "window-exposed" event to the client responsible for the window. Such a window system does not save the occluded portions of windows. When the client sends output to a window, output primitives are clipped against the window system's window (which corresponds to the graphics package's viewport). If a line is drawn in a partially visible window, only the visible part of the line is drawn.

When the client responsible for the window receives the window-exposed event, the client can handle the event by establishing a clipping window on the entire portion of the world that is to be displayed in the window system's window, and then processing all the output primitives, or by using a smaller clipping window corresponding to just the newly exposed portion of the window, as shown in Fig. 10.6. Alternatively, the client can save the pixmaps of obscured windows to avoid regenerating the window's contents when the window is next uncovered. The client would simply display some or all of the saved pixmap with a PixBlt from the saved pixmap to the screen pixmap. In some cases, the window system repaints the window backgrounds and borders to provide the illusion of quick

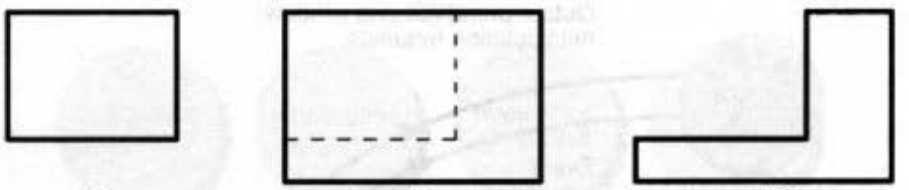


Fig. 10.6 When a window (a) is enlarged to (b), the enlarged window can be updated by using the bitmap that was in the original window (a), clipping world-coordinate output primitives against the clipping region (c), and drawing those within the clip region. Note that the clip region is nonconvex, so clipping a line may result in display of more than one line segment.

response, even though the actual redraw may incur a noticable delay.

This minimal-window-system approach makes sense in systems that support diverse application data models, ranging from pixmaps through structured hierarchies to more complex relational models. The window system itself cannot have a model that is well matched for all applications. Instead, the application is given total responsibility for providing efficient storage and redrawing capabilities.

The Macintosh window system is an example of this minimalist design [APPL85]. Algorithms embedded in its Quickdraw graphics package allow clipping to arbitrary regions, such as those shown in Fig. 10.6. NeWS and X also support arbitrary-shaped windows. Some of the clipping algorithms needed to support arbitrary windows are discussed in Section 19.7.

More memory-rich window systems save the obscured parts of windows, so that the client does not need to display newly exposed portions of a window. Other window systems give the client a choice of whether or not obscured window parts are to be saved. In any case, there is a question of how much of the obscured window is saved. Typically, the maximum possible size of the window is saved, which is usually the size of the entire screen. Some window systems save a pixmap larger than the display itself, although this approach becomes more expensive as the pixmap becomes larger or deeper. Decreasing memory prices, however, are having a dramatic effect on what is cost-effective. The client must be involved in redrawing if the window is scrolled away from the part of the world that has been saved as a pixmap, or if the view must be rescaled.

A slightly different strategy is for the window manager to keep, for each window, an offscreen pixmap containing the entire window. Whenever part of a window is unobscured, the appropriate subarea of the offscreen pixmap is copied to the screen by a PixBlt. This strategy is slow for window updating, because partially obscured windows can be written into by client programs. Thus, after a client program writes into a window (which is the offscreen pixmap), the window system must copy the visible part of the window to the screen. Alternatively, the window system can directly scan convert new output primitives into both the offscreen pixmap and the visible part of the window in the onscreen pixmap, by clipping each output primitive against two clip regions: one for the visible part of the pixmap, the other for the entire offscreen pixmap. Updates to a completely unobscured window can be done faster by updating only the visible, onscreen version of the window;

the window is then copied to its offscreen pixmap only when it is about to be obscured. To avoid this whole issue, some window systems require that the window that is being written into be completely unobscured. This solution, unfortunately, also prevents multiprocessing when an active process needs to write into partially obscured windows. Several special-purpose hardware systems avoid the need to copy offscreen pixmaps to the screen pixmap. In these systems, the hardware knows where each offscreen pixmap is stored and which portion of each is to be made visible on the screen. On each refresh cycle, the video controller picks up the appropriate pixels from the offscreen pixmaps. Exposing more of a window is done not by copying pixmaps, but by giving new window-visibility information to the hardware. These hardware solutions are further discussed in Chapter 18.

A second way to implement this type of window system, developed by Pike [PIKE83] and discussed further in Section 19.8, avoids storing any information twice. Each window is partitioned into rectangular pixmaps. Invisible pixmaps are saved offscreen, whereas visible pixmaps are saved only in the onscreen refresh memory.

Yet another fundamental design strategy is to have the window system maintain a display list for each window, as in the VGTS system developed by Lantz and Nowicki [LANT84]. In essence, the window system maintains a display-list–based graphics package, such as SPHIGS, as part of the window system. Whenever a window needs to be redrawn, the display list is traversed and clipped. Fast scan-conversion hardware is desirable for this approach, so that redraw times do not become prohibitively long. Pixmap-oriented applications, such as paint programs, do not benefit from this approach, although VGTS does include a pixmap primitive.

A frequent concern is the effect of a window-resize operation on the amount of information shown in a window: what happens to the world-coordinate window when the window-manager window is resized? There are two possibilities, and the client program should be able to cause either to occur. The first possibility is that, when the user resizes the window, the world-coordinate window changes size correspondingly. The net effect is that the user sees more or less of the world, according to whether the window was made larger or smaller, but always at the same scale, as depicted in Fig. 10.7(c). The second possibility is that, when the user resizes the window, the world-coordinate window size stays fixed. Thus, as the window is enlarged, the same amount of the world is seen, but at a larger scale. In one approach, a uniform scaling is applied to the world, even if the aspect ratios of the world window and window-system window are different; this can make some part of the window-system window go unused, as in Fig. 10.7(d). Alternatively, a nonuniform scaling can be applied, distorting the contents of the world window to fit the window-system window, as in Fig. 10.7(e).

Several window systems use hierarchical windows—that is, windows that contain subwindows—as shown in Fig. 10.8. Subwindows are generally contained within their parents. Hierarchical windows can be used to implement dialogue boxes and forms of the type shown in Chapter 8. The entire dialogue box is defined as a window, and then each field, radio button, and check box is defined as a separate subwindow, and mouse button-down events are enabled for each one. When the user selects any of the subwindows, the event report contains the name of that subwindow. A typical restriction is that subwindows be contained within their parent window, so if the dialogue box is to be moved outside of the application window, the box cannot be implemented as a subwindow to the application window.

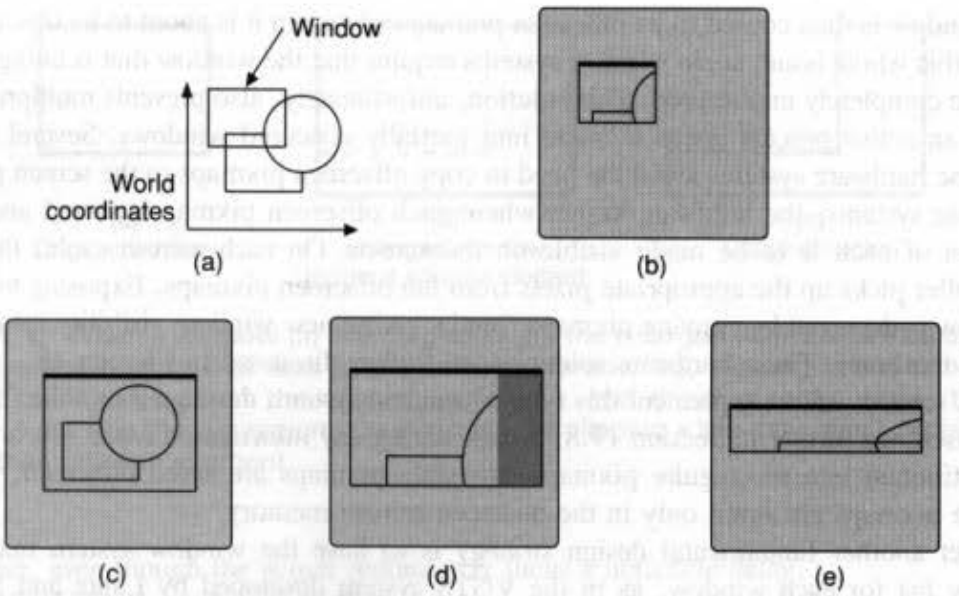


Fig. 10.7 Relationships between world-coordinate window and window-manager window. (a) A world-coordinate scene; (b) its view through a window. In (c), when the window-manager window is enlarged, more of the world is seen: the world-coordinate window is enlarged. In (d), enlarging the window-manager window creates an enlarged view of the contents of the world-coordinate window. The enlargement is done with uniform scaling, so that part of the window-manager window (gray tone) is unused. In (e), enlarging the window-manager window also creates an enlarged view of the contents of the world-coordinate window, but with nonuniform scaling so as to fill the entire window-manager window.

The design of a window-hierarchy system involves many subtleties, such as determining the effect on children of resizing a parent. Also, if a client process spawns a subprocess that then creates windows, the subprocess's windows could be considered subwindows of the spawning process's window, except that the subprocess may have its own event queue to receive input from its windows. See [ROSE83; SCHE86] for a more extensive discussion of hierarchies.

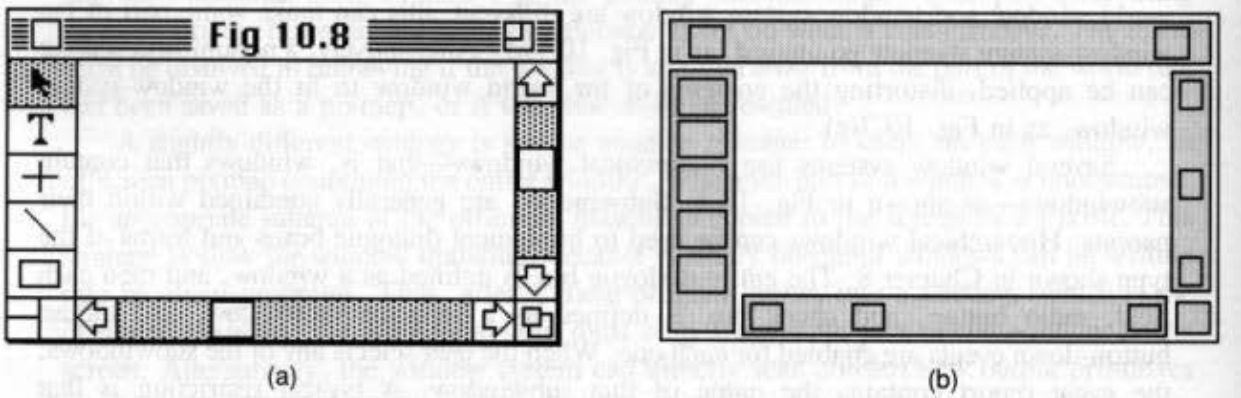


Fig. 10.8 (a) The window for a drawing program; (b) division of the window into a hierarchy of windows. In (b), a contained window is a child of the containing window. (Copyright 1988 Claris Corporation. All rights reserved.)

The subroutine calls with which a typical window system must deal follow:

CreateWindow (*name*)	Create a new window; it becomes the current window
SetPosition (*xmin*, *ymin*)	Set position of current window
SetSize (*height*, *width*)	Set size of current window
SelectWindow (*name*)	Make this the current window
ShowWindow	Make the current window visible, on top of all others
HideWindow	Make the current window invisible; expose any windows the current window was hiding
SetTitle (*char_string*)	Set displayed title of current window to *char_string*
GetPosition (*xmin*, *xmax*)	Get position of the current window
GetSize (*height*, *width*)	Get size of the current window
BringToTop	Put the current window on top of all other windows
SendToBottom	Send the current window to the bottom, behind all others
DeleteWindow	Delete the current window

The other output resource allocated by a window system is look-up table entries. Imagine a window system running on an 8-bit per pixel hardware system, with two window-system clients each wanting to have a look-up table. With two clients, each could be given half the entries (128), but then the number of look-up table entries per client depends on the number of clients. A fixed upper bound on the number of clients could be established to determine the number of entries per client, but if there are in fact fewer clients, then some of the look-up table entries will be wasted. A single client at a time could be given exclusive use of the look-up table—often the client whose window contains the cursor. This solution is viable, but suffers in that the overall screen appearance can change dramatically as the table is given first to one client, then to another, and so on.

Another solution is to allocate not look-up table entries, but rather colors. If a client asks for 100 percent blue, and some entry already contains this color, the client is given the same index to use (but is not allowed to change the contents of the entry). If no entry contains 100 percent blue and there are free entries, one is allocated. Otherwise, the entry with the color closest to that requested is allocated. The danger is that the distance between the requested and actual color might be quite large; not being able to change the look-up table is also a disadvantage. Unfortunately, there is no generally satisfactory solution.

10.4 INPUT HANDLING IN WINDOW SYSTEMS

The input resource being allocated and managed by the window system for that system's clients is the set of input devices and the events the devices generate. The window system must know to which client different types of events are to be routed. The process of routing events to the proper client is sometimes called *demultiplexing*, since events destined for different clients arrive in sequential order from a single source and must then be fanned out to different clients.

The types of events are those discussed in Section 10.1, plus additional events that are specific to window systems. Some window systems generate *window-enter* and *window-leave* events, which allow a user interface to highlight the window containing the cursor without the overhead of constantly sampling the pointer device. Window systems that do not retain a record of what is displayed in each window generate *window-damage* events whenever a window needs to be redrawn. Damage occurs if the window is enlarged,

uncovered, or scrolled. The window-enter and window-leave events are generated by the window system in direct response to user actions, whereas the window-damage event is a secondary event generated when a client requests that a window be changed. All these events are routed to the client's event queue. Some user actions, such as closing a window, can cause damage events to be to sent to a large number of clients. The information in the event report is similar to that discussed in Section 10.1, but is augmented with additional event types and window-specific information.

If a window hierarchy exists, child windows can be manipulated just like the parent window, and can have associated events. The window name associated with an event is that of the lowest-level window in the hierarchy that contains the cursor and for which the event was enabled. This means that different types of events can be associated with different windows. Every event placed in the event queue has as part of its record the name of the window with which it is associated. It is also possible to report an event in a subwindow not just to the subwindow, but also to all the windows that contain the subwindow. The client would do this by enabling the same event for all the windows in the hierarchical path from the outermost containing window to the lowest-level subwindow. Thus, multiple event reports, each with a different window name, will be placed in the client's event queue.

With hierarchical windows, a pop-up menu can be defined as a main window subdivided into as many subwindows as there are menu items. As the cursor moves out of the window of menu item i into the window of menu item $i + 1$, a leave-window event is placed on the event queue with the window name of menu item i, followed by an enter-window event with the window name of menu item $i + 1$. The client program processes the first event by undoing the highlight feedback on menu item i. It processes the second event similarly, creating the highlight feedback on menu item $i + 1$. If the cursor enters some region of the pop-up menu (such as a title area at the top) that is not overlaid by a subwindow, then the enter-window event includes the name of the pop-up menu window. Window hierarchies tend to be used in this way, but user response time can be degraded by the processing time needed to manage the hierarchy and its many events. A NeWS-style window system does not generate such a large number of events to be processed, because the feedback can be handled within the server.

Hierarchical windows can be used for selection of displayed objects in much the same way as SPHIGS structures can be. Of course, subwindows are not as general as structures, but they are convenient for picking rectangularly shaped, hierarchically nested regions (NeWS and X support nonrectangular windows). Other uses for hierarchical windows include causing the cursor shape to change as the cursor moves from one part of the screen to another, and allowing pick detection on the handles sometimes used for manipulating graphics objects (see Fig. 8.42).

Two basic approaches are widely used by window systems to route events to clients: real-estate–based and listener routing. Many window systems actually provide both strategies, and allow the window manager to specify which to use. Some also allow a policy provided by the window manager to be used in place of either of these two basic approaches. *Real-estate–based* routing looks at which window the cursor is in when an event occurs; all events are directed to the client that created the window and include the name of the window as part of the event report.

To do real-estate–based routing, the window system must maintain a data structure that stores the bounds of each window, as shown in Fig. 10.9. When an event occurs, the data

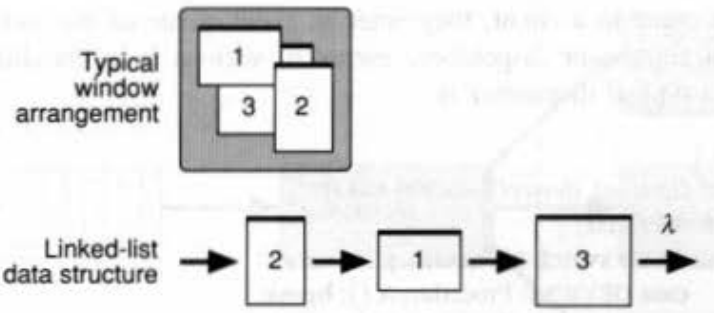


Fig. 10.9 Data structure used by the window manager to determine in which window the cursor is located, by searching for the first window on the list that brackets the cursor position.

structure is searched for the visible window containing the cursor position. If the data structure is ordered, with the most recently fully revealed window always brought to the head, then the search can terminate with the first window that contains the cursor (x, y) position. For hierarchical windows, a more complex data structure and search is needed (see Exercise 10.2).

Listener event routing, also known as *click-to-type* routing, is done when one client tells the window system to route all events of a certain type to another client (the receiving client can be, but does not need to be, the client that makes the request). For instance, the window manager can have a command that allows the user to route all keyboard input to the client that owns a particular window. The window manager implements the command by directing the window system to route keyboard events to the appropriate client program. Keyboard events are those most commonly routed explicitly, but even a mouse button-down event can be redirected.

Event distribution can cause unexpected results for users. Suppose, for instance, that the user accidentally double-clicks on a window's close (also called "go-away') button, although only a single click is needed to close the window. The window system routes the first click to the window manager, which closes the window. The second click is next routed to whatever was underneath the close button of the now-closed window, perhaps in turn selecting a menu command!

Message-transmission delays in a network can also wreak havoc with the user. Consider a drawing program executing in window A. To draw a rubberband line, the user generates a button-down event in window A, and then moves the cursor to wherever the line will end. This point might be outside of window A, if the line is to end in an area of the drawing that is not visible through the window. In anticipation of the cursor going out of its window, the drawing program sends the window system a request that *all* button-up events come to it, no matter where they occur. Now if the user does move the cursor outside window A, we have a *race condition*. Will the mouse-up event occur before or after the window system receives the request that all mouse-up events go to the drawing program? If the mouse-up occurs before the request is received, the event will go to whatever window the cursor is in, causing an unexpected result. If the mouse-up occurs after the request is received, all is well. The only certain way to avoid these race conditions is to require the client to tell the server that event i has been processed before the server will send event $i + 1$ to the client.

Once events come to a client, they enter an event queue of the sort shown in Figure 10.2. The client routes, or dispatches, events to various event-handling routines. The pseudocode for a typical dispatcher is

```
while (!quit) {
    WaitEvent (timeout, deviceClass, deviceId);
    switch (deviceClass) {
        case CLASS1: switch (deviceId) {
                case DEVICE1: ProcedureA (); break;
                case DEVICE2: ProcedureB (); break;
            }
        case CLASS2: switch (deviceId) {
etc.
```

As events occur and the program moves to different states, the procedures called in response to a particular event may change, further complicating the program logic.

The *dispatcher model* (also called the *notifier model*) enhances the input-handling system with a procedure that responds to the user actions, as shown in Fig. 10.10. The application program *registers* a procedure with the notifier and tells the notifier under what conditions the procedure is to be called. Procedures called by the notifier are sometimes called *callback procedures*, because they are called back by the notifier. With hierarchical windows, different callback procedures can be associated with locator events occurring in different parts of a dialogue box or in different parts of a menu. These callback procedures may modify the event before it is reported to the application, in which case they are called *filter procedures*.

The input subroutine calls with which a typical window system must deal include

EnableEvents (*eventList*)	Enable the listed set of events
WaitEvent (*timeout, eventType, windowName, eventRecord*)	Get the next event from the event queue
SetInputFocus (*window, eventList*)	Direct all input events of the type on *eventList* to *window*
CursorShape (*pixmap, x, y*)	*pixmap* defines the cursor shape; *x, y* give the position in the cursor pixmap used for reporting the cursor position

Typical types of events that can be placed on the event queue follow:

KeyPress	Keyboard key pressed
KeyRelease	Keyboard key released
ButtonPress	Locator (such as mouse) button pressed
ButtonRelease	Locator button released
Motion	Cursor has moved
LeaveNotify	Cursor has left window
EnterNotify	Cursor has entered window
WindowExpose	Window has been partially or completely exposed
ResizeRequest	Window resizing has been requested
Timer	Previously specified time or time increment has occurred

Each of these event types has a timestamp (see Section 10.1 to understand why this is needed), the name of the window where the cursor was when the event occurred, and other

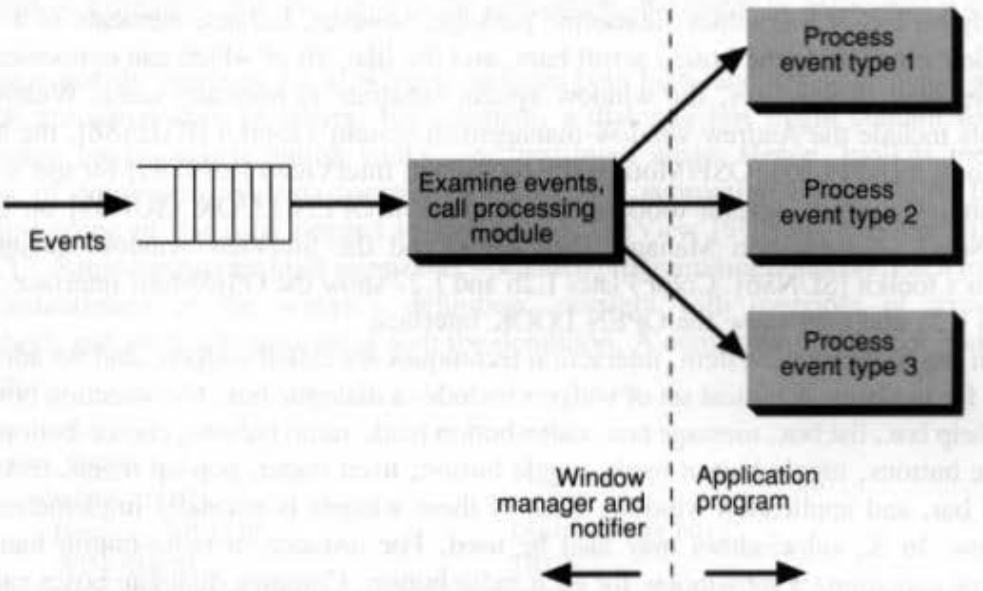


Fig. 10.10 The window manager's notifier examines the event queue and calls the procedure previously registered to process a particular event.

event-specific information, such as the new window size for a ResizeRequest.

This brief overview of window-management systems has excluded important topics, such as ensuring that the window system is sufficiently general that any and all types of window-manager policies can be provided. Also important is whether the window system is a separate process from the clients, is a subroutine library linked in with the clients, or is a part of the operating system. These and other issues are more fully discussed in [LANT87; SCHE86; SCHE88a; STEI89].

10.5 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 window manager, which is after all just another client 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.

As shown in Fig. 10.1, the toolkit can be implemented on top of the window-management system. 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 Andrew window-management system's toolkit [PALA88], the Macintosh toolkit [APPL85], OSF/Motif [OPEN89a] and InterViews [LINT89] for use with the X Window System, several toolkits that implement OPEN LOOK [SUN89] on both X and NeWS, Presentation Manager [MICR89], and the SunView window-management system's toolkit [SUN86]. Color Plates I.26 and I.27 show the OSF/Motif interface. Color Plates I.28 and I.29 show the OPEN LOOK interface.

In the X Window System, interaction techniques are called *widgets*, and we adopt this name for use here. A typical set of widgets includes a dialogue box, file-selection box, alert box, help box, list box, message box, radio-button bank, radio buttons, choice-button bank, choice buttons, toggle-button bank, toggle button, fixed menu, pop-up menu, text input, scroll bar, and application window. Each of these widgets is normally implemented as a window. In X, subwindows may also be used. For instance, a radio-button bank is a window containing a subwindow for each radio button. Complex dialogue boxes can have dozens of subwindows. An application window may have subwindows for scroll bars, resize buttons, and so on, as in Fig. 10.8.

Interaction-technique toolkits typically have notifiers of the type discussed in Section 10.4 to invoke callback procedures when events occur in their subwindows. The procedures are, in some cases, part of the toolkit—for instance, procedures to highlight the current menu item, to select and deselect radio buttons, and to scroll a list or file-selection box. They can also be provided by the application; for instance, there are procedures to carry out a command selected from a menu, to check the validity of each character as it is typed into a text input area, or simply to record the fact that a button has been selected. Figure 10.11

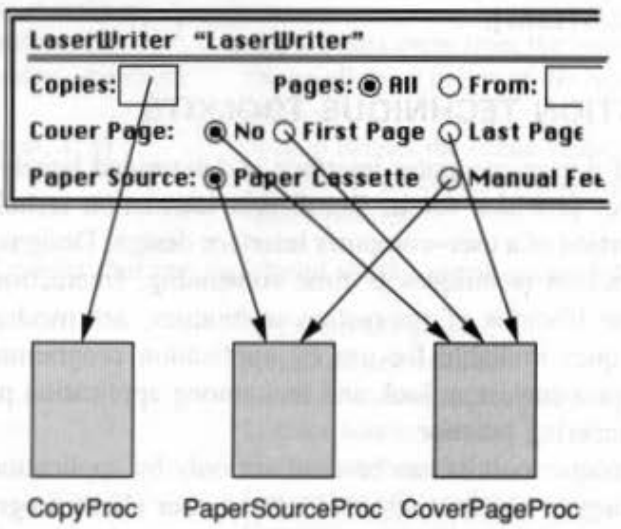


Fig. 10.11 Callback procedures associated with widgets in a dialogue box. CopyProc checks to ensure that each character is numeric and that the total number entered does not exceed some upper bound. PaperSourceProc manages the radio-button bank for the paper source to ensure that one and only one button is on and to maintain the current selection. CoverPageProc performs a similar function for the cover-page radio-button bank. (Screen graphics © Apple Computer, Inc.)

shows part of a dialogue box and some of the procedures that might be associated with the box.

Notice that the previous list of widgets includes both high- and low-level items, some of which are composites of others. For example, a dialogue box might contain several radio-button banks, toggle-button banks, and text-input areas. Hence, toolkits include a means of composing widgets together, typically via subroutine calls. Figure 10.12 shows just some of the code needed to specify the SunView [SUN86] dialogue box of Fig. 10.13. Some toolkits are built using object-oriented programming concepts: Each widget is an instantiation of the widget's definition, possibly with overrides of some of the methods and attributes associated with the definition. A composite consists of multiple instances.

```
print_frame =
  window_create(
    frame,    FRAME,                    {Surrounding box}
    WIN_SHOW,                           TRUE,
    FRAME_NO_CONFIRM,                   TRUE,
    FRAME_SHOW_LABEL,                   TRUE,
    FRAME_LABEL,     "Print",           {Header at top of window}
  0);                                   {Zero means end of list}

print_panel =                           {Panel inside the window}
  window_create(print_frame,            PANEL,
    WIN_ROWS,                           PRINT_WIN_ROWS,
    WIN_COLUMNS,                        PRINT_WIN_COLS,
  0);

print_uickb_name =                      {Header at top of panel}
  panel_create_item(print_panel,        PANEL_MESSAGE,
    PANEL_LABEL_STRING,                 "UICKB: Untitled",
    PANEL_ITEM_X,                       ATTR_COL(PRINT_NAME_COL),
    PANEL_ITEM_Y,                       ATTR_ROW(PRINT_NAME_ROW),
  0);

print_report_choice_item =
  panel_create_item(print_panel,        PANEL_CHOICE,
  {List of mutually exclusive options}
    PANEL_ITEM_X,                       ATTR_COL(PRINT_REPORT_COL),
    PANEL_ITEM_Y,                       ATTR_ROW(PRINT_REPORT_ROW),
    PANEL_LABEL_STRING,                 "Report",
    PANEL_LAYOUT,                       PANEL_VERTICAL,{Or horizontal}
    PANEL_CHOICE_STRINGS,
      "Completeness", "Consistency", "Schema", 0,
    PANEL_NOTIFY_PROC,                  print_report_choice_proc,
    {Name of callback procedure}
  0);
```

Fig. 10.12 Some of the SunView code needed to specify the dialogue box of Fig. 10.13.

Fig. 10.13 Dialogue box created using the SunView window-manager system's toolkit. The code specifying this box is shown in Fig. 10.12. (Courtesy of Kevin Murray, The George Washington University.)

Creating composites by programming, no matter what the mechanism, is tedious. Interactive editors, such as those shown in Figs. 10.14 and 10.15, allow composites to be created and modified quickly, facilitating the rapid prototyping discussed in Section 9.6. Cardelli has developed a sophisticated interactive editor that allows spatial constraints between widgets to be specified [CARD88]. At run time, when the dialogue box's size can be changed by the user, the constraints are used to keep the widgets neatly spaced.

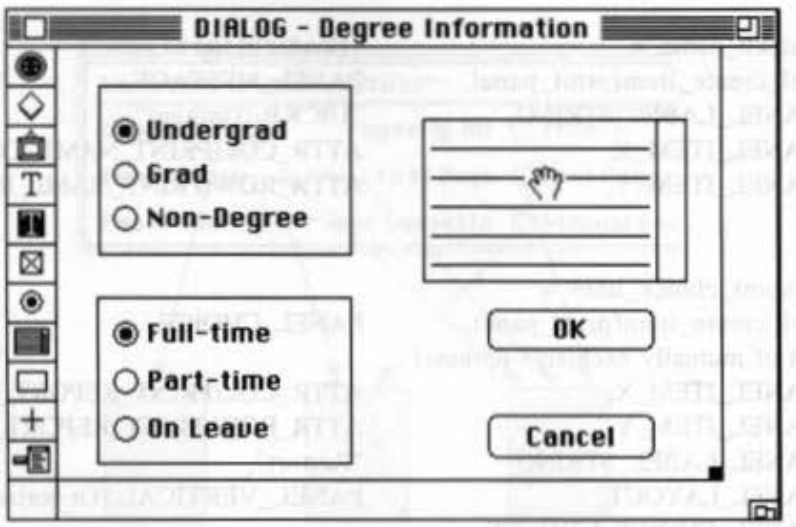
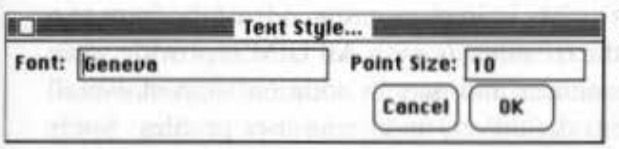


Fig. 10.14 The SmethersBarnes Prototyper dialogue-box editor for the Macintosh. A scrolling-list box is being dragged into position. The menu to the left shows the widgets that can be created; from top to bottom, they are buttons, icons, pictures, static text, text input, check boxes, radio buttons, scrolling lists, rectangles (for visual grouping, as with the radio-button banks), lines (for visual separation), pop-up menus, and scroll bars. (Courtesy of SmethersBarnes, Inc.)

Fig. 10.15 An interactive editor for designing windows. The size, position, border style, title, and presence of the "go away" box can all be controlled. The editor shows the window at its actual size; the text file describing the window is at the upper right, and a dialogue box for controlling the window characteristics is at the upper left. The window's size and position can be modified by direct manipulation, in which case the values in the dialogue box are modified. The text file is written out as the permanent record of the window's design. The I.D. and Ref# form a name by which the application program can refer to the window. (Screen graphics © Apple Computer, Inc.)

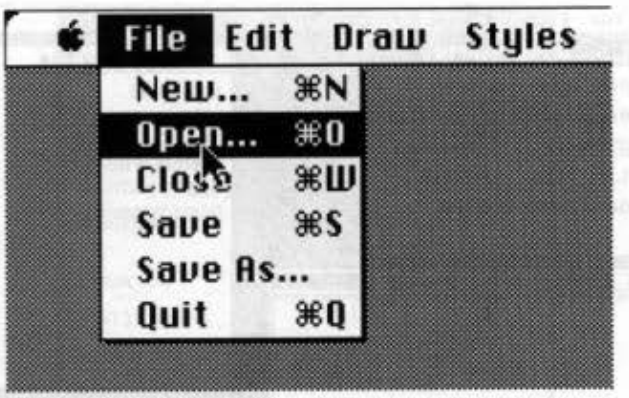
The output of these editors is a representation of the composite, either as data structures that can be translated into code, or as code, or as compiled code. In any case, mechanisms are provided for linking the composite into the application program. Programming skills are not needed to use the editors, so the editors are available to user-interface designers and even to sophisticated end users. These editors are typical of the interactive design tools shown in Fig. 10.1.

Another approach to creating menus and dialogue boxes is to use a higher-level programming-language description. In Mickey [OLSE89], an extended Pascal for the Macintosh, a dialogue box is defined by a record declaration. The data type of each record item is used to determine the type of widget used in the dialogue box; enumerated types become radio-button banks, character strings become text inputs, Booleans become checkboxes, and so on. Figure 10.16 shows a dialogue box and the code that creates it. An




```
type
  Str40     = string[40]
  textStyle = record
                font : Str40;
                points ('Name = 'Point Size'') : integer
              end
```

Fig. 10.16 A dialogue box created automatically by Mickey from the extended Pascal record declaration. (Courtesy of Dan Olsen, Jr., Brigham Young University.)




```
procedure NewDrawing (
    (* Menu = File Name = 'New...' Key = N *)
    DrawFile : OutFileDesc);                    {Name of dialogue box to be shown.}
procedure OpenDrawing (
    (* Menu = File Name = 'Open...' Key = O *)
    DrawFile : InFileDesc);                     {Name of dialogue box to be shown.}
procedure CloseDrawing;
    (* Menu = File Name = 'Close' Key = W *)
procedure SaveDrawing;
    (* Menu = File Name = 'Save' Key = S *)
procedure SaveDrawingAs (
    (* Menu = File Name = 'Save As...' *)
    DrawFile : OutFileDesc);                    {Name of dialogue box to be shown.}
```

Fig. 10.17 A menu created automatically by Mickey from the extended Pascal record declaration. (Courtesy of Dan Olsen, Jr., Brigham Young University.)

interactive dialogue-box editor can be used to change the placement of widgets. Figure 10.17 shows a menu and the code from which it is generated.

Peridot [MYER86; MYER88] takes a radically different approach to toolkits. The interface designer creates widgets and composite widgets interactively, by example. Rather than starting with a base set of widgets, the designer works with an interactive editor to create a certain look and feel. Examples of the desired widgets are drawn, and Peridot infers relationships that allow instances of the widget to adapt to a specific situation. For instance, a menu widget infers that its size is to be proportional to the number of items in the menu choice set. To specify the behavior of a widget, such as the type of feedback to be given in response to a user action, the designer selects the type of feedback from a Peridot menu, and Peridot generalizes the example to all menu items.

10.6 USER-INTERFACE MANAGEMENT SYSTEMS

A user-interface management system (UIMS) assists in implementing at least the form of a user interface, and in some cases portions of the meaning as well. All UIMSs provide some means of defining admissible user-action sequences and may in addition support overall screen design, help and error messages, macro definition, undo, and user profiles. Some recent UIMSs also manage the data associated with the application. This is in contrast to interaction technique toolkits, which provide far less support.

UIMSs can increase programmer productivity (in one study, up to 50 percent of the code in interactive programs was user-interface code [SUTT78]), speed up the development

process, and facilitate iterative refinement of a user interface as experience is gained in its use. As suggested in Fig. 10.1, the UIMS is interposed between the application program and the interaction-technique toolkit. The more powerful the UIMS, the less the need for the application program to interact directly with the operating system, window system, and toolkit.

In some UIMSs, user-interface elements are specified in a programming language that has specialized operators and data types. In others, the specification is done via interactive graphical editors, thus making the UIMS accessible to nonprogrammer interface designers.

Applications developed on top of a UIMS are typically written as a set of subroutines, often called *action routines* or *semantic action routines*. The UIMS is responsible for calling appropriate action routines in response to user inputs. In turn, the action routines influence the dialogue—for instance, by modifying what the user can do next on the basis of the outcome of a computation. Thus, the UIMS and the application share control of the dialogue—this is called the *shared-control* model. A UIMS in which the action routines have no influence over the dialogue is said to follow an *external-control* model; control resides solely in the UIMS. External control is not as powerful a model as is shared control.

UIMSs vary greatly in the specific capabilities they provide to the user–interface designer, but the one essential ingredient is a dialogue-sequence specification, to control the order in which interaction techniques are made available to the end user. For this reason, in the next section, we turn our attention to dialogue sequencing; then, in Section 10.6.2, we discuss more advanced UIMS concepts. Further background on UIMSs can be found in [HART89; MYER89; OLSE84b; OLSE87].

10.6.1 Dialogue Sequencing

Permissible sequences of user actions can be defined in a variety of ways: via transition networks (also called state diagrams), recursive transition networks, event languages, or by example, where the designer demonstrates the allowable action sequences to the system and the system "learns" what sequences are possible. Common to all these methods is the concept of a user-interface *state* and associated user actions that can be performed from that state. The notion of state has been discussed in Section 9.3.1 (state diagrams) and in Section 9.4 (modes). Each of the specification methods encodes the user-interface state in a slightly different way, each of which generalizes to the use of one or more *state variables*.

If a context-sensitive user interface is to be created, the system response to user actions must depend on the current state of the interface. System responses to user actions can include invocation of one or several action routines, changes in one or more of the state variables, and enabling, disabling, or modifying interaction techniques or menu items in preparation for the next user action. Help should also be dependent on the current state. Since the outcome of computations performed by the action routines should affect user-interface behavior, the action routines must be able to modify the state variables. Thus, state is at the heart of context-sensitivity, a concept central to contemporary user interfaces.

The simplest and least powerful, but nevertheless useful, sequence specification method is the *transition network* or *state diagram*. Transition networks have a single state variable, an integer indicating the current state. User actions cause transitions from one

state to another; each transition has associated with it zero or more action routines that are called when the transition occurs. In addition, states can have associated action routines executed whenever the state is entered. This shorthand is convenient for actions that are common to all transitions entering a state.

The action routines can affect the current state of the transition network in one of two ways. First, they can place events in the event queue, which in turn drives the interaction handling. This approach implicitly modifies the state, although to ensure that the state change is immediate, the event must be put at the front of the queue, not at the back. Alternatively, the action routines can modify the state more directly by simply setting the state variable to a new value. The first approach is cleaner from a software-engineering view, whereas the second is more flexible but more error-prone.

A number of UIMSs are based on state diagrams [JACO83; JACO85; SCHU85; RUBE83; WASS85]. Some of these provide interactive transition-network editors, which makes the networks simple to specify. The first UIMS, developed by Newman and called *The Reaction Handler*, included such an editor [NEWM68]. A simple transition-network–driven UIMS is easy to implement—see Exercise 10.8.

Transition networks are especially useful for finding sequencing inconsistencies, as discussed in Section 9.3.1, and can easily be used to determine the number of steps required to complete a task sequence. Thus, they also serve as means of predicting how good a particular design will be, even before the complete interface is implemented. Consider, for example, the simple case of explicit versus implicit acceptance of results. Figure 10.18 represents a one-operand command with explicit acceptance and rejection of the results; Fig. 10.19 shows implicit acceptance and explicit rejection. In the first case, three steps are always required: enter command, enter operand, accept. In the second case, only two steps are normally needed: enter command, enter operand. Three steps are needed only when an error has been made. Minimizing steps per task is one goal in interface design, especially for experienced users, since (not surprisingly) the speed with which experienced users can input commands is nearly linearly related to the number of discrete steps (keystrokes, hand movements) required [CARD80].

Transition networks, however, have drawbacks. First, the user-interface state is typically based on a number of state variables, and having to map all possible combinations of values of these variables onto a single state is awkward and nonintuitive for the

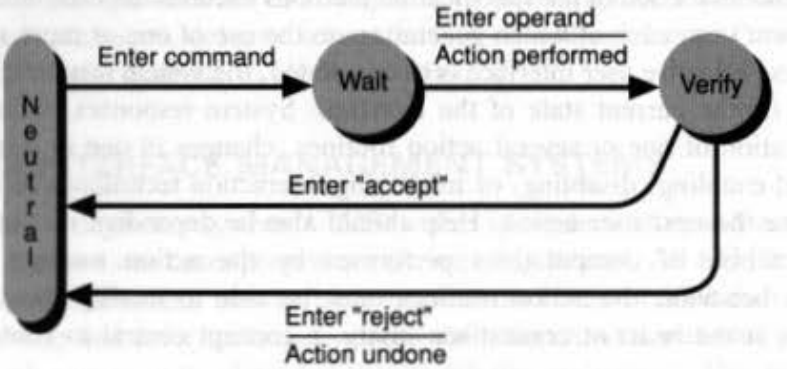


Fig. 10.18 Transition network for a dialogue with explicit acceptance and rejection of results.

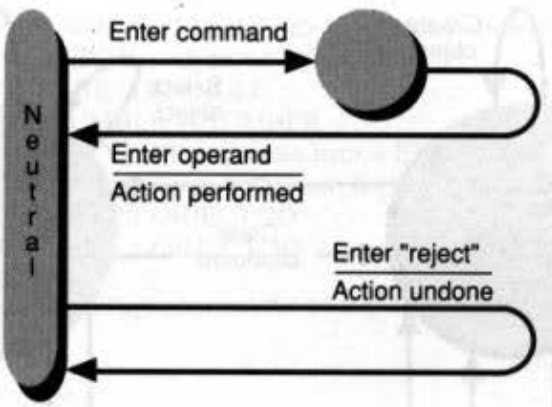


Fig. 10.19 Transition network for a dialogue with implicit acceptance and explicit rejection of results.

user-interface designer. For example, if commands are to behave in one way when there is a currently selected object (CSO) and in another way when there is no CSO, the number of states must be doubled to encode the "CSO exists–does not exist" condition. These types of context-sensitivities can expand the state space and make the transition networks difficult to create and understand. Figure 10.20, for example, shows a transition network for a simple application having the following commands:

- Select an object (establishes a CSO)
- Deselect the CSO (so there is no CSO)
- Create an object (establishes a CSO)
- Delete the CSO (so there is no CSO)
- Copy the CSO to the clipboard (requires a CSO, makes the clipboard full)
- Paste from the clipboard (requires that the clipboard be full, creates a CSO)
- Clear the clipboard (requires that the clipboard be full, empties the clipboard).

Four states are needed to encode the two possible conditions of the clipboard and the CSO. Notice also that whether or not any objects exist at all also should be encoded, since objects must exist for the command Select_object to be available in the starting state. Four more states would be needed to encode whether any objects do or do not exist.

Concurrency creates a similar state-space growth problem. Consider two user-interface elements—say, two concurrently active dialogue boxes—each with its own "state" encoding the selections currently allowable or currently made. If each dialogue-box state can be encoded in 10 states, their combination requires 100 states; for three dialogue boxes, 1000 states are needed; and so on. This exponential growth in state space is unacceptable. Jacob [JACO86] combines transition networks with object-oriented programming concepts to specify complete user interfaces while limiting the state-space explosion. Objects are self-contained entities within the interface, and each object has its own transition network to specify its behavior, which is independent of that of other objects. The UIMS portion of HUTWindows, the Helsinki University of Technology Window Manager and UIMS, uses a similar strategy [KOIV88].

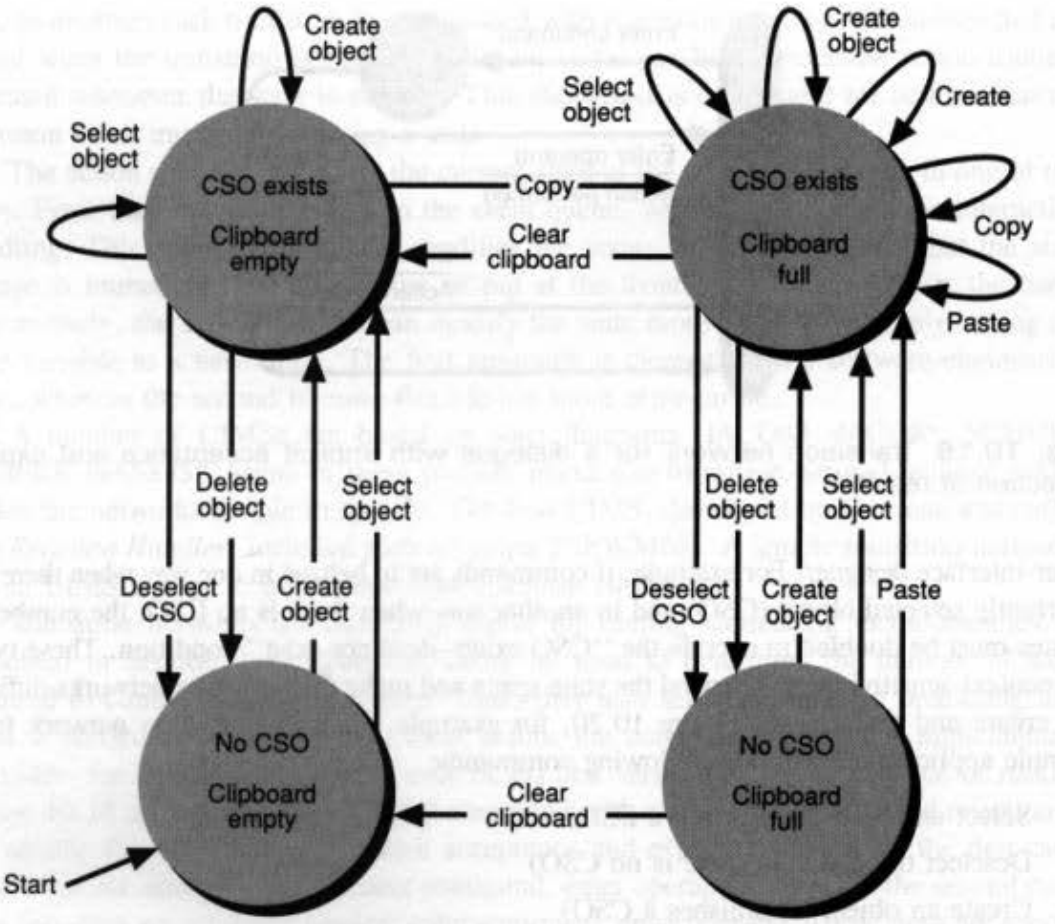


Fig. 10.20 A transition network with four states. Not all commands are available in all states. In general, action routines associated with transitions should appear on diagrams of this sort, with the names of the user actions (user commands in this case); we omit them here because the actions are obvious.

Globally available commands similarly enlarge the transition network. If help is to be globally available, each state must have an associated help state, a transition to the help state, and a reverse transition back to the originating state. This is also needed for the help to be context-sensitive. Undo must be done similarly, except that the transition from an undo state returns to a state different from the one from which it was entered. As the number of transitions relative to the number of states increases, we end up with complex "spaghetti" transition networks.

Various specialized constructs have been developed to simplify transition networks. For instance, we can alleviate the help problem by using subnetworks, in a fashion analogous to subroutines, to hide localized repetitive detail. Transition networks that can call subnetworks recursively are called *recursive transition networks*. The state variables in this case are the entire stack of saved states, plus the state of the currently active transition network. Several other powerful diagramming techniques, all derived from transition networks, are described in [WELL89].

```
<command> ::= <create> | <polyline> | <delete> | <move> | STOP
<create> ::= CREATE + <type> + <position>
<type> ::= SQUARE | TRIANGLE
<position> ::= NUMBER + NUMBER
<polyline> ::= POLYLINE + <vertex list> + END_POLY
<vertex_list> ::= <position> | <vertex_list> + <position>
<delete> ::= DELETE + OBJECT_ID
<move> ::= MOVEA + OBJECT_ID + <position>
```

Fig. 10.21 Backus–Naur form representation of the sequencing rules for a simple user interface.

Backus–Naur form (BNF) can also be used to define sequencing, and is equivalent in representational power to recursive transition networks (both are equivalent to push-down automata). BNF, illustrated in Fig. 10.21, can also be shown diagrammatically as the diagrams of Fig. 10.22. It is difficult to read BNF and to obtain a good overview of the sequencing rules, but BNF form can be processed to provide an evaluation of certain aspects of user-interface quality [BLES82, REIS82], or to generate command-language parsers [JOHN78]. Several older UIMSs were based on BNF specifications [HANA80; OLSE83; OLSE84a].

Transition networks, whether recursive or not, encode user-inteface state in a small number of state variables. *Augmented transition networks* (ATNs), a more flexible derivative of transition networks, encode user-interface state by which node of the ATN is active and by what the values of explicit state variables are. Responses can be the calling of action routines, the setting of these explicit state variables, or the changing of the node of the ATN that is active. Of course, the state variables can also be set by action routines. Figure 10.23 shows an ATN in which the Boolean state variable *cb*, set by several of the transitions, is used to affect flow of control from one state to another. The variable *cb* is **true** if the clipboard is full.

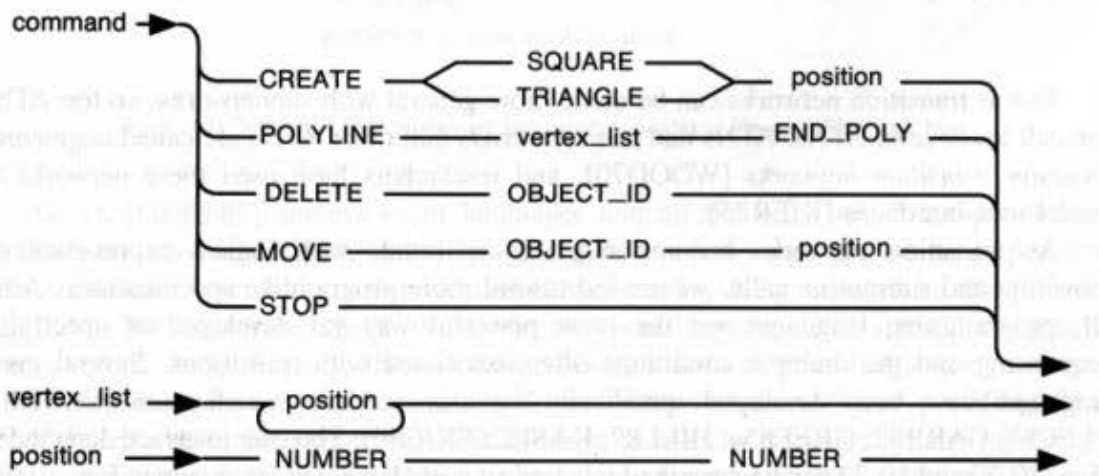


Fig. 10.22 A diagrammatic representation of the Backus–Naur form equivalent to that in Fig. 10.21.

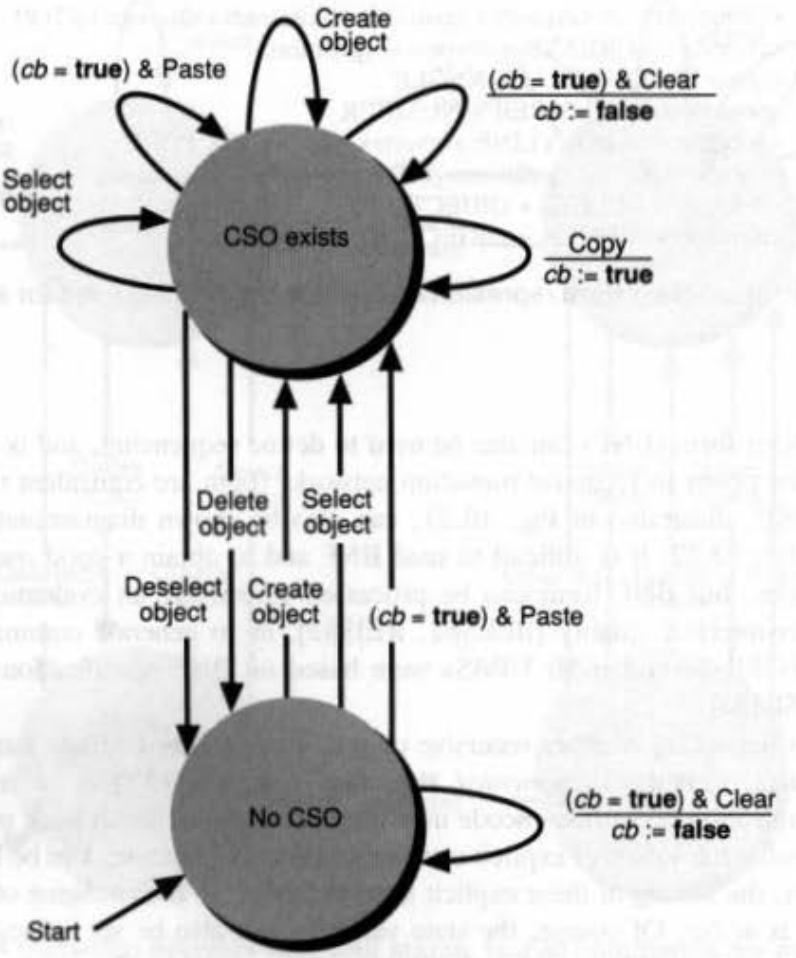


Fig. 10.23 An augmented transition network (ATN) representing the same user interface as that in Fig. 10.20. Transitions can be conditionally dependent on the value of explicit state variables (the Boolean *cb* in this case), and can also set state variables. In general, the application program can also set state variables.

Just as transition networks can be made more general with subnetworks, so too ATNs can call lower-level ATNs. ATNs that can recursively call other ATNs are called *augmented recursive transition networks* [WOOD70], and researchers have used these networks to model user interfaces [KIER85].

As transition networks become more complicated, with logical expressions on transition and subroutine calls, we are led toward more programlike specifications. After all, programming languages are the most powerful way yet developed of specifying sequencing and the multiple conditions often associated with transitions. Several *event languages* have been developed specifically for user-interface specification [CARD85; FLEC87; GARR82; GREE85a; HILL87; KASI82; SIOC89]. The user interface depicted in Figs. 10.20 and 10.23 can be described in a typical event language, as shown in Fig. 10.24. Note that event languages, unlike traditional programming languages, have no explicit flow of control. Instead, whenever an **if** condition becomes true, the associated actions are executed. Thus, the event language is a production-rule system.

```
if Event = SelectObject then
  begin
    cso := true
    perform action routine name
  end
if Event = DeselectCSO and cso = true then
  begin
    cso := false
    perform action routine name
  end
if Event = CreateObject then
  begin
    cso := true
    perform action routine name
  end
if Event = DeleteCSO and cso = true then
  begin
    cso := false
    perform action routine name
  end
if Event = CopyCSO and cso = true then
  begin
    cb := true
    perform action routine name
  end
if Event = PasteClipboard and cb = true then
  begin
    cso := true
    perform action routine name
  end
if Event = ClearClipboard and cb = true then
  begin
    cb := false
    perform action routine name
  end
```

Fig. 10.24 A typical event language, with a Pascal-like syntax.

Green [GREE87] surveys event languages and all the other sequence-specification methods we have mentioned, and shows that general event languages are more powerful than are transition networks, recursive transition networks, and grammars; he also provides algorithms for converting these forms into an event language. ATNs that have general computations associated with their arcs are also equivalent to event languages.

If event languges are so powerful, why do we bother with the various types of transition networks? Because, for simple cases, it is easier to work with diagrammatic representations. One of the goals of UIMSs is to allow nonprogrammers who specialize in user-interface design to be directly involved in creating an interface. This goal is probably best met with transition-network–oriented tools that are easier to use, although somewhat less powerful. Networks provide a useful, time-proven tool for laying out a dialogue, and

they appear to help the designer to document and understand the design. The diagrammatic representations are especially compelling if user actions are performed on interaction objects such as menus, dialogue boxes, and other visible objects. Then diagrams of the type shown in Fig. 10.25 can be created interactively to define dialogue sequencing. If needed, conditions (such as the *cb* = **true** in Fig. 10.23) can be associated with the arcs. Figure 10.26 shows one way to establish a link on such a diagram.

A quite different way to define syntax is by example. Here, the user-interface designer places the UIMS into a "learning" mode, and then steps through all acceptable sequences of actions (a tedious process in complex applications, unless the UIMS can infer general rules from the examples). The designer might start with a main menu, select an item from it, and then go through a directory to locate the submenu, dialogue box, or application-specific object to be presented to the user in response to the main menu selection. The object appears on the screen, and the designer can indicate the position, size, or other attributes that the object should have when the application is actually executed. The designer goes on to perform some operation on the displayed object and again shows what object should appear next, or how the displayed object is to respond to the operation; the designer repeats this process until all actions on all objects have been defined. This technique works for sequencing through items that have already been defined by the interface designer, but is not sufficiently general to handle arbitrary application functionality. User-interface software tools with some degree of by-example sequencing specification include Menulay [BUXT83], TAE Plus [MILL88c] and the SmethersBarnes Prototyper [COSS89]. Peridot, mentioned earlier, builds interaction techniques, (i.e., hardware bindings) by example.

10.6.2 Advanced UIMS Concepts

UIMSs have tended to focus on sequence control and visual design. Transition networks provide a good basis for sequencing, and interactive editors are just right for visual design.

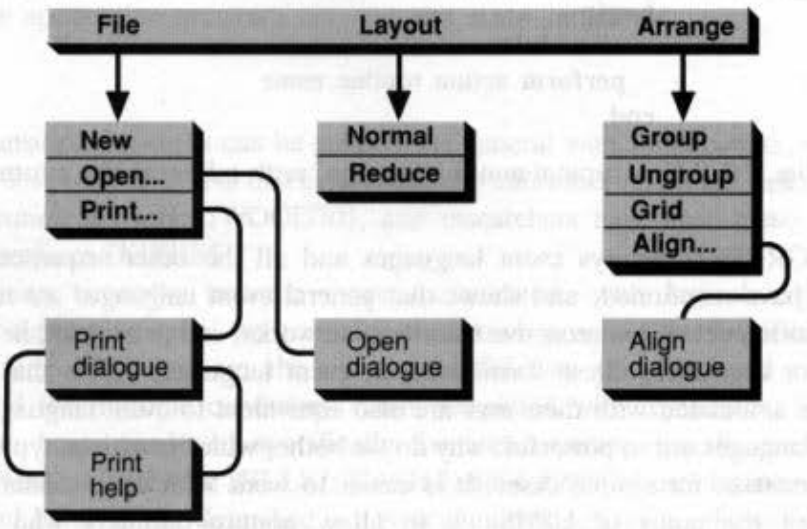


Fig. 10.25 Several menus and dialogue boxes linked together. The return paths from dialogue boxes to the main menu are not shown.

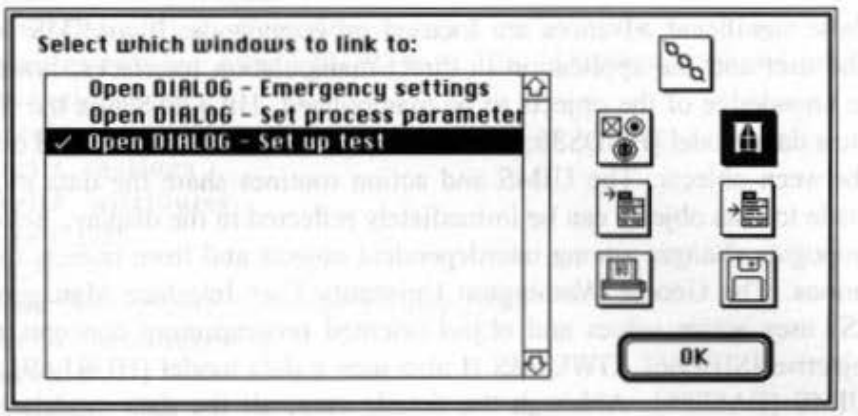


Fig. 10.26 Linking together different interaction techniques using the SmethersBarnes Prototyper. A menu item is being linked to the dialogue box "Set up test" checked in the list. The icons on the right are used to select the class of responses to be linked to the menu selection. Reading left to right underneath the "link" icon, the possibilities are enable or disable check boxes, radio buttons, and buttons in dialogue boxes; open a window or dialogue box (the class of response selected); enable or disable a menu; enable or disable a menu item; open a print dialogue box; or open a file dialogue box. (Courtesy of SmethersBarnes, Inc.)

As discussed in Chapter 9, however, a user-interface design includes conceptual, functional, sequencing, and hardware-binding levels. Much recent UIMS development has begun to address the functional and conceptual designs as well. Thus, there has come to be more focus on combining sequencing control with a higher-level model of objects and commands, and also on integrating intelligent help systems into the UIMS.

Representations at a higher level than that of transition networks are clearly needed. Consider how difficult it would be to add to the transition network of Fig. 10.20 new states to record whether any objects have yet been created. It would also be difficult to apply some of the dialogue modifications, such as CO, currently selected command, and factored attributes, discussed in Section 9.4. And the sequencing specifications provide no information about what operations can be performed on what objects, and certainly give no glimmer of whart parameters are needed to perform an operation.

The first step away from a sequencing orientation and toward higher levels of abstraction was taken by COUSIN [HAYE83; HAYE84], which automatically generates menus and dialogue boxes from a specification of commands, parameters, and parameter data types. The innovation of COUSIN is in defining all the parameters needed by a command as an integral unit. COUSIN has enough information that a prefix or postfix syntax could also be generated. Green took a similar approach, adding preconditions and postconditions to specify the semantics of user commands [GREE85b]. Olsen's MIKE system [OLSE86] declares commands and parameters, also generating a user interface in a fashion similar to COUSIN. In addition, MIKE supports direct manipulation of objects to specify positions, and can cause commands to be executed when a button-down event occurs in a window or subwindow.

All these significant advances are focused on commands. If a UIMS is to mediate between the user and the application in direct-manipulation interfaces, however, it must have some knowledge of the objects to be manipulated. HIGGINS was the first UIMS to incorporate a data model [HUDS86; HUDS87; HUDS88], one that is based on objects and relations between objects. The UIMS and action routines share the data model, so that changes made to data objects can be immediately reflected in the display. *Active values* are used to propogate changes among interdependent objects and from objects to their visual representations. The George Washington University User Interface Management System (GWUIMS) uses active values and object-oriented programming concepts to achieve a similar objective [SIBE86]. GWUIMS II also uses a data model [HURL89], as does the Serpent UIMS [BASS88]. Although the details vary, all the data models make use of object-oriented programming concepts and active values, and are closely related to developments in database-management systems in the area of semantic data models [HULL87].

The User Interface Design Environment (UIDE) project [FOLE89] has developed a new user-interface specification method integrating some elements of these recent developments to include a data model, the commands that can be applied to each type of object in the data model, the parameters needed by the commands, the parameter data types, the conditions under which commands are available for use (that is, command preconditions), and the changes that occur to state variables when a command is executed (that is, command postconditions) [FOLE87; FOLE88]. To illustrate the method, we start with the sample application developed in Section 10.6.1. We add a data model, which is here a single class of objects with two subclasses, square and triangle. In addition, there are two distinguished instances of objects, the CSO and the clipboard object, both of which may or may not exist at any given time. The specification is shown in Fig. 10.27. The preconditions are the conditions on state variables that must be satisfied for a command to be invoked, whereas the postconditions are changes in state variables.

Not only is this specification sufficient to create automatically an operational interface to the application's action routines, but also it is represented such that

- Menu items can be enabled and disabled, using preconditions
- Users can be told why a command is disabled, again using preconditions
- Users can be told what to do to enable a command, by back chaining to determine what commands must be invoked to satisfy the preconditions of the command in question
- Users can be given a partial explanation of what a command does, using the postconditions
- Some user-interface design-consistency rules can be checked
- Different interaction techniques can be assigned for use in specifying commands and command parameters
- Speed of use of the interface can be predicted for various task sequences and for various interaction techniques.

Another way to define user interfaces consisting of interconnected processing modules is with data-flow diagrams. For instance, the NeXT Interface Builder, shown in Fig. 10.28,

```
class object                                    {First the data model}
  subclasses triangle, square;
  actions CreateObject, SelectObject;
  attributes position range [0..10] x [0..10]   {Attribute name and data type}
class triangle, square;
  superclass object;
  inherits  actions
  inherits  attributes
instance CSO
  of object
  actions DeselectCSO, DeleteCSO, CopyCSO
  inherits  attributes
instance CB
  of object
  actions ClearClipboard, Paste
  inherits  attributes

{Initial values for state variables}
initial Number (object) := 0; csoExists := false; cbFull := false;

{Actions on objects, with preconditions, postconditions, and parameters}
precondition Number (object) ≠ 0;
SelectObject (object);
postcondition csoExists := true;

precondition csoExists := true;
DeselectCSO (CSO);
postcondition csoExists := false;

precondition;
CreateObject (position, object);
postcondition Number (object) := Number (object) +1; csoExists := true;

precondition csoExists := true;
DeleteCSO (CSO);
postcondition Number (object) = Number (object) - 1; csoExists := false;

precondition csoExists := true;
CopyCSO (CSO);
postcondition  cbFull := true;

precondition cbFull := true;
Paste (CB);
postcondition csoExists := true;

precondition cbFull := true;
ClearClipboard (CB);
postcondition cbFull := false;
```

Fig. 10.27 A high-level specification of a user interface incorporating a data model, sequencing information, and command parameters.

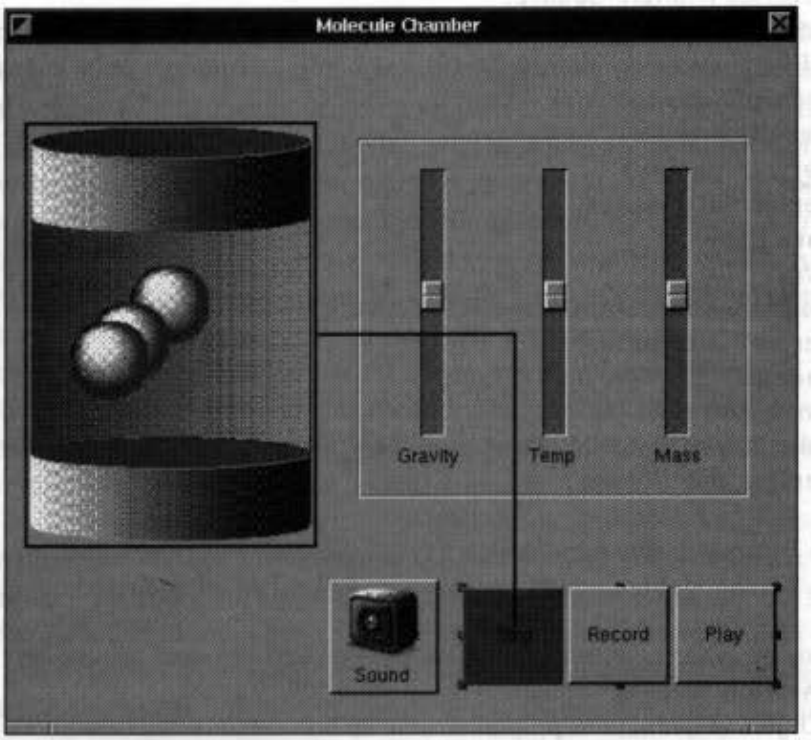


Fig. 10.28 The NeXT Interface Builder, showing a connection being made. The user has already selected the type of message to be sent from the Stop button, and has drawn the connection to the cylindrical chamber to indicate the destination of the message. (Courtesy of NeXT, Inc. © 1989 NeXT, Inc.)

allows objects to be interconected so that output messages from one object are input to another object. Type checking is used to ensure that only compatible messages are sent and received.

Data-flow diagrams can also be used to specify the detailed behavior of some or all of a user interface, although doing so takes on considerable programming flavor and suffers the same problems of scale seen with flowcharts and transition networks. Work is this area is surveyed in [BORN86a]; a more recent project is described in [SMIT88]. A specialized system, for scientific data visualization, is shown in Color Plate I.30.

UIMSs are finding their way into regular use. Early UIMSs suffered from rigid interaction styles that did not allow custom-tailoring to suit users' needs and were overly dependent on transition networks. Commercial UIMSs are now used on a large scale, and are becoming as essential to developing interactive graphics application programs as are graphics subroutine packages, window managers, and interaction-technique toolkits.

EXERCISES

10.1 Study the user interfaces to two different window systems. Categorize each with respect to the design issues discussed in Section 9.3.

10.2 Devise the search mechanism for real-estate–based event routing with overlapping main windows, in which each main window can contain a hierarchy of spatially nested subwindows.

10.3 Survey three window-management systems to determine whether they

a. Have hierarchical windows
b. Implement a server–client model and, if they do, whether the implementation allows the server and client to be distributed in a network
c. Provide real-estate or listener input event dispatching, or some combination thereof
d. Integrate the graphics subroutine package with the window-management system, or pass graphics calls directly through to graphics hardware.

10.4 Write an interactive dialogue box or menu editor.

10.5 Implement the concepts demonstrated in MICKY [OLSE89] with a programming language and toolkit available to you.

10.6 Examine several user interfaces with which you are familiar. Identify a set of user-interface state variables used in implementing each user interface. How many of these state variables are used in the user interface to provide context-sensitive menus, help, and so on?

10.7 Document the dialogue of a user interface to a paint or drawing program. Do this (a) with state diagrams, and (b) with the specialized language introduced in Section 10.6.1. Which method did you find easier? Why? Compare your opinions with those of your classmates. Which method is easier for answering questions such as "How do I draw a circle?" "What do I do after an error message appears?"

10.8 Write a transition-network–based UIMS. Every transition in the state diagram should be represented by a state-table entry with the following information:

- Current state number
- Next state
- Event which causes the transition
- Name of procedure to call when the transition occurs.

Events should include selection of a command from a menu, typing of a command name, mouse movement, mouse button-down, and mouse button-up. You should automatically display a menu containing all possible commands (derive this list from the events in the state table), enabling only those choices available from the current state.

10.9 For each of the extensions to state diagrams discussed in Section 9.3, determine whether the modifications create a push-down automaton (is it bounded or unbounded?) or a Turing machine.

10.10 Carefully study a window-management system that includes a policy-free window system. Examine several window managers to determine whether they can be implemented with the window system. For instance, some window systems provide for borders around windows for scroll bars, a heading, and perhaps selection buttons. For the window system to be completely policy-free, you must be able to specify separately the width of the borders on the four sides of the window.

11 Representing Curves and Surfaces

Smooth curves and surfaces must be generated in many computer graphics applications. Many real-world objects are inherently smooth, and much of computer graphics involves modeling the real world. Computer-aided design (CAD), high-quality character fonts, data plots, and artists' sketches all contain smooth curves and surfaces. The path of a camera or object in an animation sequence (Chapter 21) is almost always smooth; similarly, a path through intensity or color space (Chapters 16 and 13) must often be smooth.

The need to represent curves and surfaces arises in two cases: in modeling existing objects (a car, a face, a mountain) and in modeling "from scratch," where no preexisting physical object is being represented. In the first case, a mathematical description of the object may be unavailable. Of course, one can use as a model the coordinates of the infinitely many points of the object, but this is not feasible for a computer with finite storage. More often, we merely approximate the object with pieces of planes, spheres, or other shapes that are easy to describe mathematically, and require that points on our model be close to corresponding points on the object.

In the second case, when there is no preexisting object to model, the user creates the object in the modeling process; hence, the object matches its representation exactly, because its only embodiment is the representation. To create the object, the user may sculpt the object interactively, describe it mathematically, or give an approximate description to be "filled in" by some program. In CAD, the computer representation is used later to generate physical realizations of the abstractly designed object.

This chapter introduces the general area of *surface modeling*. The area is quite broad, and only the three most common representations for 3D surfaces are presented here: polygon mesh surfaces, parametric surfaces, and quadric surfaces. We also discuss

Fig. 11.1 A 3D object represented by polygons.

parametric curves, both because they are interesting in their own right and because parametric surfaces are a simple generalization of the curves.

Solid modeling, introduced in the next chapter, is the representation of volumes completely surrounded by surfaces, such as a cube, an airplane, or a building. The surface representations discussed in this chapter can be used in solid modeling to define each of the surfaces that bound the volume.

A *polygon mesh* is a set of connected polygonally bounded planar surfaces. Open boxes, cabinets, and building exteriors can be easily and naturally represented by polygon meshes, as can volumes bounded by planar surfaces. Polygon meshes can be used, although less easily, to represent objects with curved surfaces, as in Fig. 11.1; however, the representation is only approximate. Figure 11.2 shows the cross-section of a curved shape and the polygon mesh representing it. The obvious errors in the representation can be made arbitrarily small by using more and more polygons to create a better piecewise linear approximation, but this increases space requirements and the execution time of algorithms processing the representation. Furthermore, if the image is enlarged, the straight edges again become obvious. (Forrest calls this problem "geometric aliasing" [FORR80], by analogy to the general notion of aliasing discussed in Chapters 3 and 14.)

Parametric polynomial curves define points on a 3D curve by using three polynomials in a parameter t, one for each of x, y, and z. The coefficients of the polynomials are selected such that the curve follows the desired path. Although various degrees of polynomials can be used, we present only the most common case, cubic polynomials (that have powers of the parameter up through the third). The term *cubic curve* will often be used for such curves.

Parametric bivariate (two-variable) *polynomial surface patches* define the coordinates of points on a curved surface by using three bivariate polynomials, one for each of x, y, and z. The boundaries of the patches are parametric polynomial curves. Many fewer bivariate polynomial surface patches than polygonal patches are needed to approximate a curved

Fig. 11.2 A cross-section of a curved object and its polygonal representation.

surface to a given accuracy. The algorithms for working with bivariate polynomials, however, are more complex than are those for polygons. As with curves, polynomials of various degrees can be used, but we discuss here only the common case of polynomials that are cubic in both parameters. The surfaces are accordingly called *bicubic surfaces*.

Quadric surfaces are those defined implicitly by an equation $f(x, y, z) = 0$, where f is a quadric polynomial in x, y, and z. Quadric surfaces are a convenient representation for the familiar sphere, ellipsoid, and cylinder.

The next chapter, on solid modeling, incorporates these representations into systems to represent not just surfaces, but also bounded (solid) volumes. The surface representations described in this chapter are used, sometimes in combination with one another, to bound a 3D volume.

11.1 POLYGON MESHES

A *polygon mesh* is a collection of edges, vertices, and polygons connected such that each edge is shared by at most two polygons. An edge connects two vertices, and a polygon is a closed sequence of edges. An edge can be shared by two adjacent polygons, and a vertex is shared by at least two edges. A polygon mesh can be represented in several different ways, each with its advantages and disadvantages. The application programmer's task is to choose the most appropiate representation. Several representations can be used in a single application: one for external storage, another for internal use, and yet another with which the user interactively creates the mesh.

Two basic criteria, space and time, can be used to evaluate different representations. Typical operations on a polygon mesh are finding all the edges incident to a vertex, finding the polygons sharing an edge or a vertex, finding the vertices connected by an edge, finding the edges of a polygon, displaying the mesh, and identifying errors in representation (e.g., a missing edge, vertex, or polygon). In general, the more explicitly the relations among polygons, vertices, and edges are represented, the faster the operations are and the more space the representation requires. Woo [WOO85] has analyzed the time complexity of nine basic access operations and nine basic update operations on a polygon-mesh data structure.

In the rest of this section, several issues concerning polygon meshes are discussed: representing polygon meshes, ensuring that a given representation is correct, and calculating the coefficients of the plane of a polygon.

11.1.1 Representing Polygon Meshes

In this section, we discuss three polygon-mesh representations: explicit, pointers to a vertex list, and pointers to an edge list. In the *explicit* representation, each polygon is represented by a list of vertex coordinates:

$$P = ((x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_n, y_n, z_n)).$$

The vertices are stored in the order in which they would be encountered traveling around the polygon. There are edges between successive vertices in the list and between the last and first vertices. For a single polygon, this is space-efficient; for a polygon mesh, however,

much space is lost because the coordinates of shared vertices are duplicated. Even worse, there is no explicit representation of shared edges and vertices. For instance, to drag a vertex and all its incident edges interactively, we must find all polygons that share the vertex. This requires comparing the coordinate triples of one polygon with those of all other polygons. The most efficient way to do this would be to sort all N coordinate triples, but this is at best an $N\log_2 N$ process, and even then there is the danger that the same vertex might, due to computational roundoff, have slightly different coordinate values in each polygon, so a correct match might never be made.

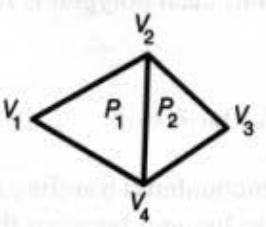
With this representation, displaying the mesh either as filled polygons or as polygon outlines necessitates transforming each vertex and clipping each edge of each polygon. If edges are being drawn, each shared edge is drawn twice; this causes problems on pen plotters, film recorders, and vector displays due to the overwriting. A problem may also be created on raster displays if the edges are drawn in opposite directions, in which case extra pixels may be intensified.

Polygons defined with *pointers to a vertex list*, the method used by SPHIGS, have each vertex in the polygon mesh stored just once, in the vertex list $V = ((x_1, y_1, z_1), \ldots, (x_n, y_n, z_n))$. A polygon is defined by a list of indices (or pointers) into the vertex list. A polygon made up of vertices 3, 5, 7, and 10 in the vertex list would thus be represented as $P = (3, 5, 7, 10)$.

This representation, an example of which is shown in Fig. 11.3, has several advantages over the explicit polygon representation. Since each vertex is stored just once, considerable space is saved. Furthermore, the coordinates of a vertex can be changed easily. On the other hand, it is still difficult to find polygons that share an edge, and shared polygon edges are still drawn twice when all polygon outlines are displayed. These two problems can be eliminated by representing edges explicitly, as in the next method.

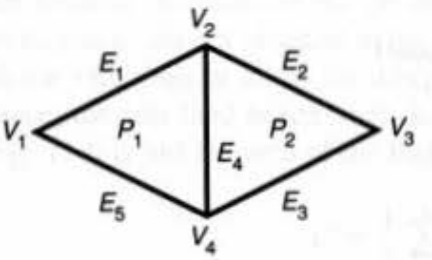
When defining polygons by *pointers to an edge list*, we again have the vertex list V, but represent a polygon as a list of pointers not to the vertex list, but rather to an edge list, in which each edge occurs just once. In turn, each edge in the edge list points to the two vertices in the vertex list defining the edge, and also to the one or two polygons to which the edge belongs. Hence, we describe a polygon as $P = (E_1, \ldots, E_n)$, and an edge as $E = (V_1, V_2, P_1, P_2)$. When an edge belongs to only one polygon, either P_1 or P_2 is null. Figure 11.4 shows an example of this representation.

Polygon outlines are shown by displaying all edges, rather than by displaying all polygons; thus, redundant clipping, transformation, and scan conversion are avoided. Filled polygons are also displayed easily. In some situations, such as the description of a 3D honeycomblike sheet-metal structure, some edges are shared by three polygons. In such



$V = (V_1, V_2, V_3, V_4) = ((x_1, y_1, z_1), \ldots, (x_4, y_4, z_4))$
$P_1 = (1, 2, 4)$
$P_2 = (4, 2, 3)$

Fig. 11.3 Polygon mesh defined with indexes into a vertex list.



$V = (V_1, V_2, V_3, V_4) = ((x_1, y_1, z_1), \ldots, (x_4, y_4, z_4))$
$E_1 = (V_1, V_2, P_1, \lambda)$
$E_2 = (V_2, V_3, P_2, \lambda)$
$E_3 = (V_3, V_4, P_2, \lambda)$
$E_4 = (V_4, V_2, P_1, P_2)$
$E_5 = (V_4, V_1, P_1, \lambda)$
$P_1 = (E_1, E_4, E_5)$
$P_2 = (E_2, E_3, E_4)$

Fig. 11.4 Polygon mesh defined with edge lists for each polygon (λ represents null).

cases, the edge descriptions can be extended to include an arbitrary number of polygons: $E = (V_1, V_2, P_1, P_2, \ldots, P_n)$.

In none of these three representations (i.e., explicit polygons, pointers to vertices, pointers to an edge list), is it easy to determine which edges are incident to a vertex: All edges must be inspected. Of course, information can be added explicitly to permit determining such relationships. For instance, the winged-edge representation used by Baumgart [BAUM75] expands the edge description to include pointers to the two adjoining edges of each polygon, whereas the vertex description includes a pointer to an (arbitrary) edge incident on the vertex, and thus more polygon and vertex information is available. This representation is discussed in Chapter 12.

11.1.2 Consistency of Polygon-Mesh Representations

Polygon meshes are often generated interactively, such as by operators digitizing drawings, so errors are inevitable. Thus, it is appropriate to make sure that all polygons are closed, all edges are used at least once but not more than some (application-defined) maximum, and each vertex is referenced by at least two edges. In some applications, we would also expect the mesh to be completely connected (any vertex can be reached from any other vertex by moving along edges), to be topologically planar (the binary relation on vertices defined by edges can be represented by a planar graph), or to have no holes (there exists just one boundary—a connected sequence of edges each of which is used by one polygon).

Of the three representations discussed, the explicit-edge scheme is the easiest to check for consistency, because it contains the most information. For example, to make sure that all edges are part of at least one but no more than some maximum number of polygons, the code in Fig. 11.5 can be used.

This procedure is by no means a complete consistency check. For example, an edge used twice in the same polygon goes undetected. A similar procedure can be used to make sure that each vertex is part of at least one polygon; we check whether at least two different edges of the same polygon refer to the vertex. Also, it should be an error for the two vertices of an edge to be the same, unless edges with zero length are allowed.

The relationship of "sharing an edge" between polygons is a binary equivalence relation and hence partitions a mesh into equivalence classes called *connected components*.

```
for (each edge E_j in set of edges)
    use_count_j = 0;
for (each polygon P_i in set of polygons)
    for (each edge E_j of polygon P)
        use_count_j++;
for (each edge E_j in set of edges) {
    if (use_count_j == 0)
        Error ();
    if (use_count_j > maximum)
        Error ();
}
```

Fig. 11.5 Code to ensure that all edges of explicit polygon representation are used between 1 and *maximum* times.

One usually expects a polygon mesh to have a single connected component. Algorithms for determining the connected components of a binary relation are well known [SEDG88].

More detailed testing is also possible; one can check, for instance, that each polygon referred to by an edge E_i refers in turn back to the edge E_i. This ensures that all references from polygons to edges are complete. Similarly, we can check that each edge E_i referred to by a polygon P_i also refers back to polygon P_i, which ensures that the references from edges to polygons are complete.

11.1.3 Plane Equations

When working with polygons or polygon meshes, we frequently need to know the equation of the plane in which the polygon lies. In some cases, of course, the equation is known implicitly through the interactive construction methods used to define the polygon. If it is not known, we can use the coordinates of three vertices to find the plane. Recall the plane equation

$$Ax + By + Cz + D = 0. \tag{11.1}$$

The coefficients A, B, and C define the normal to the plane, $[A \quad B \quad C]$. Given points P_1, P_2, and P_3 on the plane, that plane's normal can be computed as the vector cross-product $P_1P_2 \times P_1P_3$ (or $P_2P_3 \times P_2P_1$, etc.). If the cross-product is zero, then the three points are collinear and do not define a plane. Other vertices, if any, can be used instead. Given a nonzero cross-product, D can be found by substituting the normal $[A \quad B \quad C]$ and any one of the three points into Eq. (11.1).

If there are more than three vertices, they may be nonplanar, either for numerical reasons or because of the method by which the polygons were generated. Then another technique for finding the coefficients A, B, and C of a plane that comes close to all the vertices is better. It can be shown that A, B, and C are proportional to the signed areas of the projections of the polygon onto the (y, z), (x, z), and (x, y) planes, respectively. For example, if the polygon is parallel to the (x, y) plane, then $A = B = 0$, as expected: The

projections of the polygon onto the (y, z) and (x, z) planes have zero area. This method is better because the areas of the projections are a function of the coordinates of all the vertices and so are not sensitive to the choice of a few vertices that might happen not to be coplanar with most or all of the other vertices, or that might happen to be collinear. For instance, the area (and hence coefficient) C of the polygon projected onto the (x, y) plane in Fig. 11.6 is just the area of the trapezoid A_3, minus the areas of A_1 and A_2. In general,

$$C = \frac{1}{2}\sum_{i=1}^{n}(y_i + y_{i\oplus 1})(x_{i\oplus 1} - x_i), \tag{11.2}$$

where the operator $\oplus$ is normal addition except that $n \oplus 1 = 1$. The areas for A and B are given by similar formulae, except the area for B is negated (see Exercise 11.1).

Eq. (11.2) gives the sum of the areas of all the trapezoids formed by successive edges of the polygons. If $x_{i\oplus 1} < x_i$, the area makes a negative contribution to the sum. The sign of the sum is also useful: if the vertices have been enumerated in a clockwise direction (as projected onto the plane), then the sign is positive; otherwise, it is negative.

Once we determine the plane equation by using all the vertices, we can estimate how nonplanar the polygon is by calculating the perpendicular distance from the plane to each vertex. This distance d for the vertex at (x, y, z) is

$$d = \frac{Ax + By + Cz + D}{\sqrt{A^2 + B^2 + C^2}}. \tag{11.3}$$

This distance is either positive or negative, depending on which side of the plane the point is located. If the vertex is on the plane, then $d = 0$. Of course, to determine only on which side of a plane a point is, only the sign of d matters, so division by the square root is not needed.

The plane equation is not unique; any nonzero multiplicative constant k changes the equation, but not the plane. It is often convenient to store the plane coefficients with a normalized normal; this can be done by choosing

$$k = \frac{1}{\sqrt{A^2 + B^2 + C^2}}, \tag{11.4}$$

which is the reciprocal of the length of the normal. Then, distances can be computed with Eq. (11.3) more easily, since the denominator is 1.

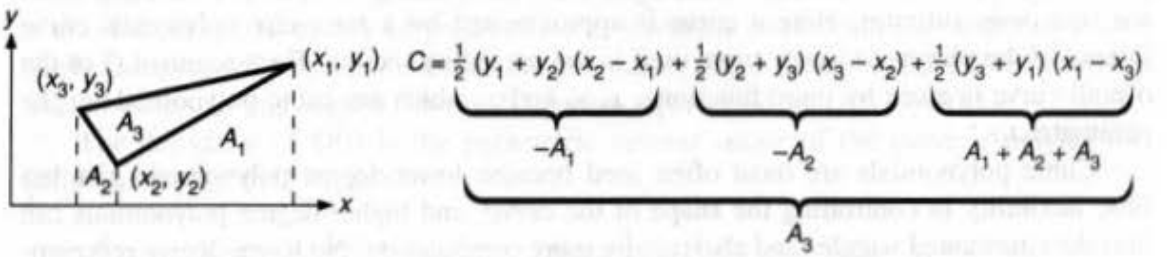


Fig. 11.6 Calculating the area C of a triangle using Eq. (11.2).

11.2 PARAMETRIC CUBIC CURVES

Polylines and polygons are first-degree, piecewise linear approximations to curves and surfaces, respectively. Unless the curves or surfaces being approximated are also piecewise linear, large numbers of endpoint coordinates must be created and stored to achieve reasonable accuracy. Interactive manipulation of the data to approximate a shape is tedious, because many points have to be positioned precisely.

In this section, a more compact and more manipulable representation of piecewise smooth curves is developed; in the following section, the mathematical development is generalized to surfaces. The general approach is to use functions that are of a higher degree than are the linear functions. The functions still generally only approximate the desired shapes, but use less storage and offer easier interactive manipulation than do linear functions.

The higher-degree approximations can be based on one of three methods. First, one can express y and z as *explicit* functions of x, so that $y = f(x)$ and $z = g(x)$. The difficulties with this are that (1) it is impossible to get multiple values of y for a single x, so curves such as circles and ellipses must be represented by multiple curve segments; (2) such a definition is not rotationally invariant (to describe a rotated version of the curve requires a great deal of work and may in general require breaking a curve segment into many others); and (3) describing curves with vertical tangents is difficult, because a slope of infinity is difficult to represent.

Second, we can choose to model curves as solutions to *implicit* equations of the form $f(x, y, z) = 0$; this is fraught with its own perils. First, the given equation may have more solutions than we want. For example, in modeling a circle, we might want to use $x^2 + y^2 = 1$, which is fine. But how do we model a half circle? We must add constraints such as $x \geq 0$, which cannot be contained within the implicit equation. Furthermore, if two implicitly defined curve segments are joined together, it may be difficult to determine whether their tangent directions agree at their join point. Tangent continuity is critical in many applications.

These two mathematical forms do permit rapid determination of whether a point lies on the curve or on which side of the curve the point lies, as was done in Chapter 3. Normals to the curve are also easily computed. Hence, we shall briefly discuss the implicit form in Section 11.4.

The *parametric representation* for curves, $x = x(t)$, $y = y(t)$, $z = z(t)$ overcomes the problems caused by functional or implicit forms and offers a variety of other attractions that will become clear in the remainder of this chapter. Parametric curves replace the use of geometric slopes (which may be infinite) with parametric tangent vectors (which, we shall see, are never infinite). Here a curve is approximated by a *piecewise polynomial* curve instead of the piecewise linear curve used in the preceding section. Each segment Q of the overall curve is given by three functions, x, y, and z, which are cubic polynomials in the parameter t.

Cubic polynomials are most often used because lower-degree polynomials give too little flexibility in controlling the shape of the curve, and higher-degree polynomials can introduce unwanted wiggles and also require more computation. No lower-degree representation allows a curve segment to interpolate (pass through) two specified endpoints with specified derivatives at each endpoint. Given a cubic polynomial with its four coefficients,

four knowns are used to solve for the unknown coefficients. The four knowns might be the two endpoints and the derivatives at the endpoints. Similarly, the two coefficients of a first-order (straight-line) polynomial are determined by the two endpoints. For a straight line, the derivatives at each end are determined by the line itself and cannot be controlled independently. With quadratic (second-degree) polynomials, and hence three coefficients, two endpoints and one other condition, such as a slope or additional point, can be specified.

Also, parametric cubics are the lowest-degree curves that are nonplanar in 3D. You can see this by recognizing that a second-order polynomial's three coefficients can be completely specified by three points and that three points define a plane in which the polynomial lies.

Higher-degree curves require more conditions to determine the coefficients and can "wiggle" back and forth in ways that are difficult to control. Despite this, higher-degree curves are used in applications—such as the design of cars and planes—in which higher-degree derivatives must be controlled to create surfaces that are aerodynamically efficient. In fact, the mathematical development for parametric curves and surfaces is often given in terms of an arbitrary degree n. In this chapter, we fix n at 3.

The cubic polynomials that define a curve segment $Q(t) = [x(t)\ y(t)\ z(t)]$ are of the form

$$\begin{aligned} x(t) &= a_x t^3 + b_x t^2 + c_x t + d_x, \\ y(t) &= a_y t^3 + b_y t^2 + c_y t + d_y, \\ z(t) &= a_z t^3 + b_z t^2 + c_z t + d_z, \qquad 0 \le t \le 1. \end{aligned} \tag{11.5}$$

To deal with finite segments of the curve, without loss of generality, we restrict the parameter t to the [0, 1] interval.

With $T = [t^3 \quad t^2 \quad t \quad 1]$, and defining the matrix of coefficients of the three polynomials as

$$C = \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{bmatrix}, \tag{11.6}$$

we can rewrite Eq. (11.5) as

$$Q(t) = [x(t) \quad y(t) \quad z(t)] = T \cdot C. \tag{11.7}$$

This provides a compact way to express the Eq. (11.5).

Figure 11.7 shows two joined parametric cubic curve segments and their polynomials; it also illustrates the ability of parametrics to represent easily multiple values of y for a single value of x with polynomials that are themselves single valued. (This figure of a curve, like all others in this section, shows 2D curves represented by $[x(t) \quad y(t)]$.)

The derivative of $Q(t)$ is the parametric *tangent vector* of the curve. Applying this definition to Eq. (11.7), we have

$$\begin{aligned} \frac{d}{dt}Q(t) = Q'(t) &= \left[\frac{d}{dt}x(t) \quad \frac{d}{dt}y(t) \quad \frac{d}{dt}z(t)\right] = \frac{d}{dt}T \cdot C = [3t^2 \quad 2t \quad 1 \quad 0] \cdot C \\ &= [3a_x t^2 + 2b_x t + c_x \quad 3a_y t^2 + 2b_y t + c_y \quad 3a_z t^2 + 2b_z t + c_z]. \end{aligned} \tag{11.8}$$

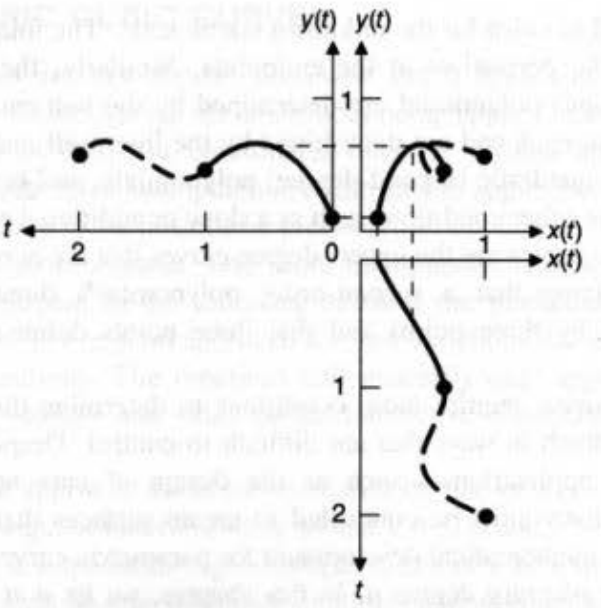


Fig. 11.7 Two joined 2D parametric curve segments and their defining polynomials. The dashed lines between the (x,y) plot and the $x(t)$ and $y(t)$ plots show the correspondence between the points on the (x,y) curve and the defining cubic polynomials. The $x(t)$ and $y(t)$ plots for the second segment have been translated to begin at $t = 1$, rather than at $t = 0$, to show the continuity of the curves at their join point.

If two curve segments join together, the curve has G^0 *geometric continuity*. If the directions (but not necessarily the magnitudes) of the two segments' tangent vectors are equal at a join point, the curve has G^1 geometric continuity. In computer-aided design of objects, G^1 continuity between curve segments is often required. G^1 continuity means that the geometric slopes of the segments are equal at the join point. For two tangent vectors TV_1 and TV_2 to have the same direction, it is necessary that one be a scalar multiple of the other: $TV_1 = k \cdot TV_2$, with $k > 0$ [BARS88].

If the tangent vectors of two cubic curve segments are equal (i. e., their directions *and* magnitudes are equal) at the segments' join point, the curve has first-degree continuity in the parameter t, or *parametric continuity*, and is said to be C^1 continuous. If the direction and magnitude of $d^n/dt^n[Q(t)]$ through the nth derivative are equal at the join point, the curve is called C^n *continuous*. Figure 11.8 shows curves with three different degrees of continuity. Note that a parametric curve segment is itself everywhere continuous; the continuity of concern here is at the join points.

The tangent vector $Q'(t)$ is the velocity of a point on the curve with respect to the parameter t. Similarly, the second derivative of $Q(t)$ is the acceleration. If a camera is moving along a parametric cubic curve in equal time steps and records a picture after each step, the tangent vector gives the velocity of the camera along the curve. The camera velocity and acceleration at join points should be continuous, to avoid jerky movements in the resulting animation sequence. It is this continuity of acceleration across the join point in Fig. 11.8 that makes the C^2 curve continue farther to the right than the C^1 curve, before bending around to the endpoint.

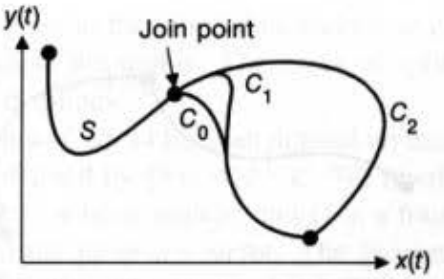


Fig. 11.8 Curve segment S joined to segments C_0, C_1, and C_2 with the 0, 1, and 2 degrees of parametric continuity, respectively. The visual distinction between C_1 and C_2 is slight at the join, but obvious away from the join.

In general, C^1 continuity implies G^1, but the converse is generally not true. That is, G^1 continuity is generally less restrictive than is C^1, so curves can be G^1 but not necessarily C^1. However, join points with G^1 continuity will appear just as smooth as those with C^1 continuity, as seen in Fig. 11.9.

There is a special case in which C^1 continuity does *not* imply G^1 continuity: When both segments' tangent vectors are [0 0 0] at the join point. In this case, the tangent vectors are indeed equal, but their directions can be different (Fig. 11.10). Figure 11.11 shows this concept in another way. Think again of a camera moving along the curve; the camera velocity slows down to zero at the join point, the camera changes direction while its velocity is zero, and the camera accelerates in the new direction.

The plot of a parametric curve is distinctly different from the plot of an ordinary function, in which the independent variable is plotted on the x axis and the dependent variable is plotted on the y axis. In parametric curve plots, the independent variable t is

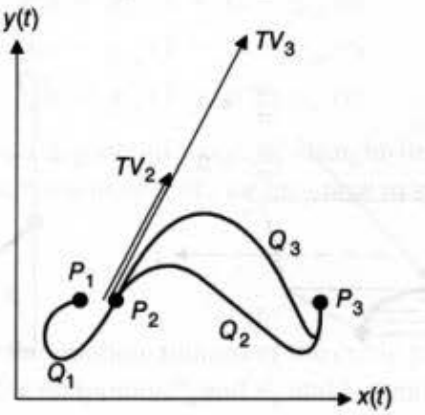


Fig. 11.9 Curve segments Q_1, Q_2, and Q_3 join at the point P_2 and are identical except for their tangent vectors at P_2. Q_1 and Q_2 have equal tangent vectors, and hence are both G^1 and C^1 continuous at P_2. Q_1 and Q_3 have tangent vectors in the same direction, but Q_3 has twice the magnitude, so they are only G^1 continuous at P_2. The larger tangent vector of Q_3 means that the curve is pulled more in the tangent-vector direction before heading toward P_3. Vector TV_2 is the tangent vector for Q_2, TV_3 is that for Q_3.

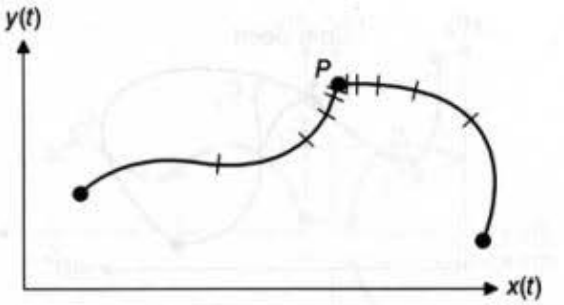


Fig. 11.10 The one case for which C^1 continuity does not imply G^1 continuity: the tangent vector (i.e., the parametric velocity along the curve) is zero at the join joint P. Each tick mark shows the distance moved along the curve in equal time intervals. As the curve approaches P, the velocity goes to zero, then increases past P.

never plotted at all. This means that we cannot determine, just by looking at a parametric curve plot, the tangent vector to the curve. It is possible to determine the direction of the vector, but not its magnitude. This can be seen as follows: if $\gamma(t)$, $0 \le t \le 1$ is a parametric curve, its tangent vector at time 0 is $\gamma'(0)$. If we let $\eta(t) = \gamma(2t)$, $0 \le t \le \frac{1}{2}$, then the parametric plots of γ and η are identical. On the other hand, $\eta'(0) = 2\,\gamma'(0)$. Thus, two curves that have identical plots can have different tangent vectors. This is the motivation for the definition of geometric continuity: For two curves to join smoothly, we require only that their tangent-vector directions match, not that their magnitudes match.

A curve segment $Q(t)$ is defined by constraints on endpoints, tangent vectors, and continuity between curve segments. Each cubic polynomial of Eq. (11.5) has four coefficients, so four constraints will be needed, allowing us to formulate four equations in the four unknowns, then solving for the unknowns. The three major types of curves discussed in this section are *Hermite*, defined by two endpoints and two endpoint tangent vectors; *Bézier*, defined by two endpoints and two other points that control the endpoint tangent vectors; and several kinds of *splines*, each defined by four control points. The

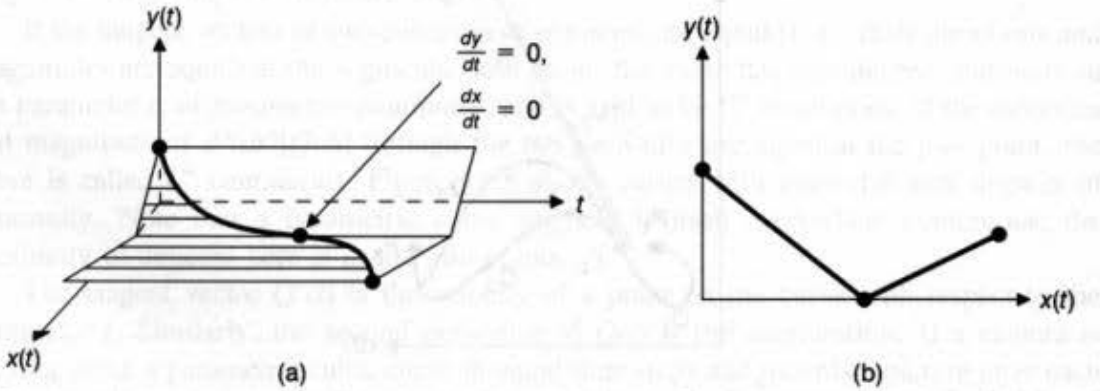


Fig. 11.11 (a) View of a 2D parametric cubic curve in 3D (x, y, t) space, and (b) the curve in 2D. At the join, the velocity of both parametrics is zero; that is, $dy/dt = 0$ and $dx/dt = 0$. You can see this by noting that, at the join, the curve is parallel to the t axis, so there is no change in either x or y. Yet at the join point, the parametrics are C^1 continuous, but are not G^1 continuous.

splines have C^1 and C^2 continuity at the join points and come close to their control points, but generally do not interpolate the points. The types of splines are uniform B-splines, nonuniform B-splines, and β-splines.

To see how the coefficients of Eq. (11.5) can depend on four constraints, we recall that a parametric cubic curve is defined by $Q(t) = T \cdot C$. We rewrite the coefficient matrix as $C = M \cdot G$, where M is a 4×4 *basis matrix*, and G is a four-element column vector of geometric constraints, called the *geometry vector*. The geometric constraints are just the conditions, such as endpoints or tangent vectors, that define the curve. We use G_x to refer to the column vector of just the x components of the geometry vector. G_y and G_z have similar definitions. M or G, or both M and G, differ for each type of curve.

The elements of M and G are constants, so the product $T \cdot M \cdot G$ is just three cubic polynomials in t. Expanding the product $Q(t) = T \cdot M \cdot G$ gives

$$Q(t) = [x(t) \quad y(t) \quad z(t)] = [t^3 \quad t^2 \quad t \quad 1] \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} \begin{bmatrix} G_1 \\ G_2 \\ G_3 \\ G_4 \end{bmatrix}. \quad (11.9)$$

Multiplying out just $x(t) = T \cdot M \cdot G_x$ gives

$$\begin{aligned} x(t) = {} & (t^3 m_{11} + t^2 m_{21} + t\, m_{31} + m_{41}) g_{1x} + (t^3 m_{12} + t^2 m_{22} + t\, m_{32} + m_{42}) g_{2x} \\ & + (t^3 m_{13} + t^2 m_{23} + t\, m_{33} + m_{43}) g_{3x} + (t^3 m_{14} + t^2 m_{24} + t\, m_{34} + m_{44}) g_{4x}. \end{aligned} \quad (11.10)$$

Equation (11.10) emphasizes that the curve is a weighted sum of the elements of the geometry matrix. The weights are each cubic polynomials of t, and are called *blending functions*. The blending functions B are given by $B = T \cdot M$. Notice the similarity to a piecewise linear approximation, for which only two geometric constraints (the endpoints of the line) are needed, so each curve segment is a straight line defined by the endpoints G_1 and G_2:

$$\begin{aligned} x(t) &= g_{1x}(1 - t) + g_{2x}(t), \\ y(t) &= g_{1y}(1 - t) + g_{2y}(t), \\ z(t) &= g_{1z}(1 - t) + g_{2z}(t). \end{aligned} \quad (11.11)$$

Parametric cubics are really just a generalization of straight-line approximations.

To see how to calculate the basis matrix M, we turn now to specific forms of parametric cubic curves.

11.2.1 Hermite Curves

The Hermite form (named for the mathematician) of the cubic polynomial curve segment is determined by constraints on the endpoints P_1 and P_4 and tangent vectors at the endpoints R_1 and R_4. (The indices 1 and 4 are used, rather than 1 and 2, for consistency with later sections, where intermediate points P_2 and P_3 will be used instead of tangent vectors to define the curve.)

To find the *Hermite basis matrix* M_H, which relates the *Hermite geometry vector* G_H to the polynomial coefficients, we write four equations, one for each of the constraints, in the four unknown polynomial coefficients, and then solve for the unknowns.

Defining G_{H_x}, the x component of the Hermite geometry matrix, as

$$G_{H_x} = \begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix}_x, \tag{11.12}$$

and rewriting $x(t)$ from Eqs. (11.5) and (11.9) as

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x = T \cdot C_x = T \cdot M_H \cdot G_{H_x} \cdot = [t^3 \quad t^2 \quad t \quad 1]\, M_H \cdot G_{H_x}, \tag{11.13}$$

the constraints on $x(0)$ and $x(1)$ are found by direct substitution into Eq. (11.13) as

$$x(0) = P_{1_x} = [0 \quad 0 \quad 0 \quad 1]\, M_H \cdot G_{H_x}, \tag{11.14}$$

$$x(1) = P_{4_x} = [1 \quad 1 \quad 1 \quad 1]\, M_H \cdot G_{H_x}. \tag{11.15}$$

Just as in the general case we differentiated Eq. (11.7) to find Eq. (11.8), we now differentiate Eq. (11.13) to get $x'(t) = [3t^2 \quad 2t \quad 1 \quad 0]M_H \cdot G_{H_x}$. Hence, the tangent-vector–constraint equations can be written as

$$x'(0) = R_{1_x} = [0 \quad 0 \quad 1 \quad 0]\, M_H \cdot G_{H_x}, \tag{11.16}$$

$$x'(1) = R_{4_x} = [3 \quad 2 \quad 1 \quad 0]\, M_H \cdot G_{H_x}. \tag{11.17}$$

The four constraints of Eqs. (11.14), (11.15), (11.16), and (11.17) can be rewritten in matrix form as

$$\begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix}_x = G_{H_x} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix} M_H \cdot G_{H_x}. \tag{11.18}$$

For this equation (and the corresponding expressions for y and z) to be satisfied, M_H must be the inverse of the 4×4 matrix in Eq. (11.18)

$$M_H = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}. \tag{11.19}$$

M_H, which is of course unique, can now be used in $x(t) = T \cdot M_H \cdot G_{H_x}$ to find $x(t)$ based on the geometry vector G_{H_x}. Similarly, $y(t) = T \cdot M_H \cdot G_{H_y}$ and $z(t) = T \cdot M_H \cdot G_{H_z}$, so we can write

$$Q(t) = [x(t) \quad y(t) \quad z(t)] = T \cdot M_H \cdot G_H, \tag{11.20}$$

where G_H is the column vector

$$\begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix}.$$

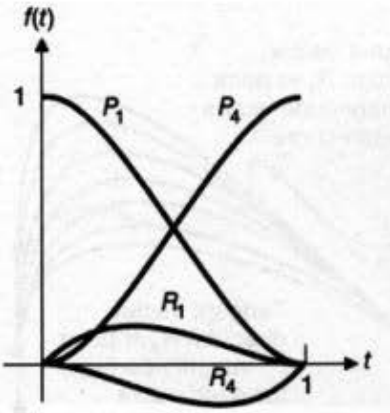


Fig. 11.12 The Hermite blending functions, labeled by the elements of the geometry vector that they weight.

Expanding the product $T \cdot M_H$ in $Q(t) = T \cdot M_H \cdot G_H$ gives the *Hermite blending functions* B_H as the polynomials weighting each element of the geometry vector:

$$Q(t) = T \cdot M_H \cdot G_H = B_H \cdot G_H$$
$$= (2t^3 - 3t^2 + 1)P_1 + (-2t^3 + 3t^2)P_4 + (t^3 - 2t^2 + t)R_1 + (t^3 - t^2)R_4. \quad (11.21)$$

Figure 11.12 shows the four blending functions. Notice that, at $t = 0$, only the function labeled P_1 is nonzero: only P_1 affects the curve at $t = 0$. As soon as t becomes greater than zero, R_1, P_4, and R_4 begin to have an influence. Figure 11.13 shows the four functions weighted by the y components of a geometry vector, their sum $y(t)$, and the curve $Q(t)$.

Figure 11.14 shows a series of Hermite curves. The only difference among them is the length of the tangent vector R_1: the directions of the tangent vectors are fixed. The longer the vectors, the greater their effect on the curve. Figure 11.15 is another series of Hermite curves, with constant tangent-vector lengths but with different directions. In an interactive

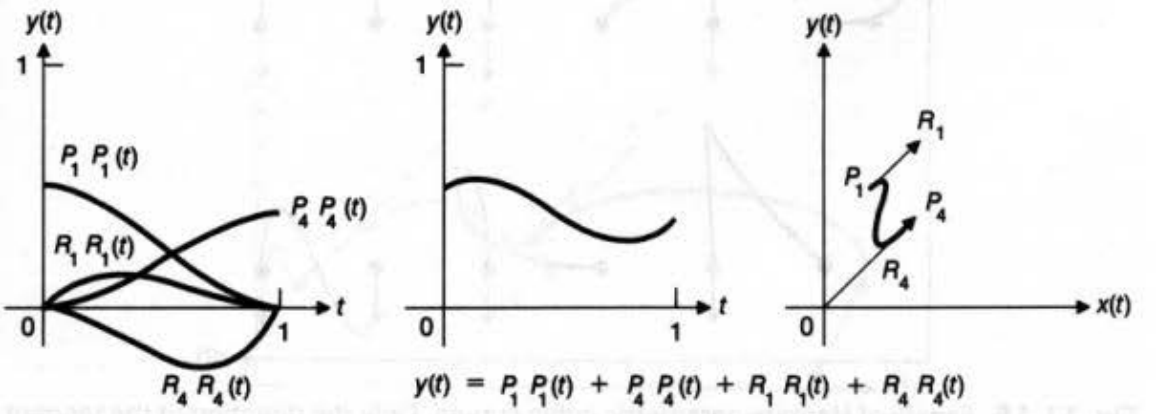


Fig. 11.13 A Hermite curve showing the four elements of the geometry vector weighted by the blending functions (leftmost four curves), their sum *y(t)*, and the 2D curve itself (far right). *x(t)* is defined by a similar weighted sum.

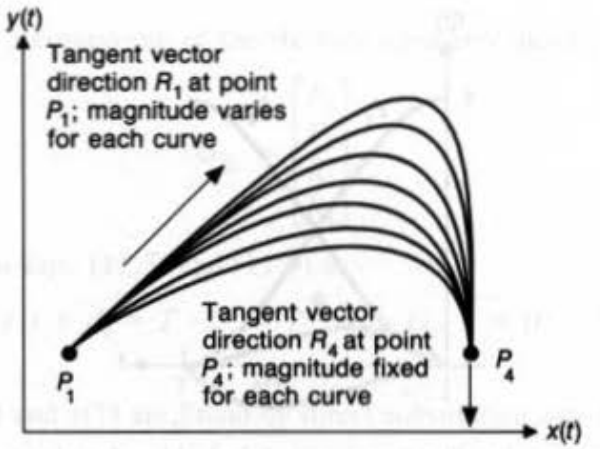


Fig. 11.14 Family of Hermite parametric cubic curves. Only R_1, the tangent vector at P_1, varies for each curve, increasing in magnitude for the higher curves.

graphics system, the endpoints and tangent vectors of a curve are manipulated interactively by the user to shape the curve. Figure 11.16 shows one way of doing this.

For two Hermite cubics to share a common endpoint with G^1 (geometrical) continuity, as in Fig. 11.17, the geometry vectors must have the form

$$\begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix} \text{ and } \begin{bmatrix} P_4 \\ P_7 \\ kR_4 \\ R_7 \end{bmatrix}, \text{ with } k > 0. \tag{11.22}$$

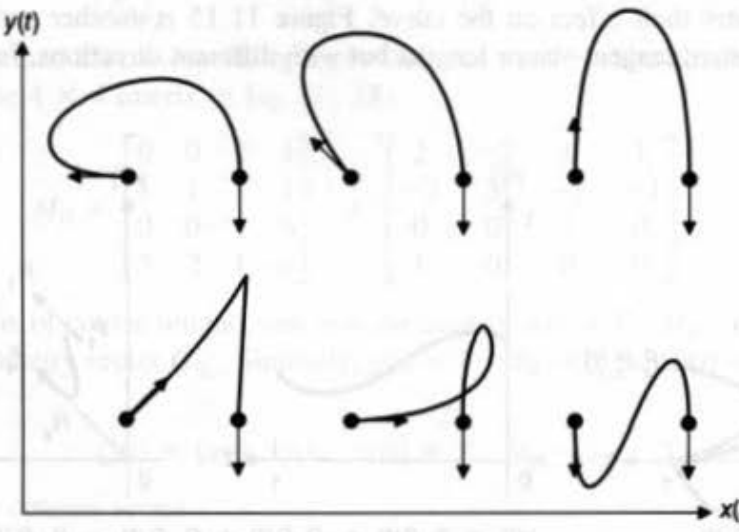


Fig. 11.15 Family of Hermite parametric cubic curves. Only the direction of the tangent vector at the left starting point varies; all tangent vectors have the same magnitude. A smaller magnitude would eliminate the loop in the one curve.

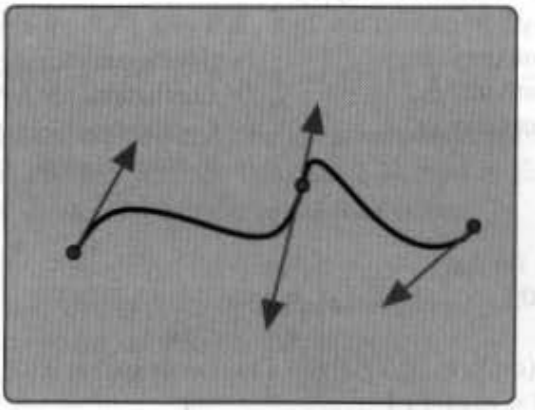


Fig. 11.16 Two Hermite cubic curve segments displayed with controls to facilitate interactive manipulation. The endpoints can be repositioned by dragging the dots, and the tangent vectors can be changed by dragging the arrowheads. The tangent vectors at the join point are constrained to be collinear (to provide C^1 continuity): The user is usually given a command to enforce C^0, C^1, G^1, or no continuity. The tangent vectors at the $t = 1$ end of each curve are drawn in the reverse of the direction used in the mathematical formulation of the Hermite curve, for clarity and more convenient user interaction.

That is, there must be a shared endpoint (P_4) and tangent vectors with at least equal directions. The more restrictive condition of C^1 (parametric) continuity requires that $k = 1$, so the tangent vector direction and magnitude must be equal.

Hermite and other similar parametric cubic curves are simple to display: We evaluate Eq. (11.5) at n successive values of t separated by a step size δ. Figure 11.18 gives the code. The evaluation within the **begin** . . . **end** takes 11 multiplies and 10 additions per 3D point. Use of Horner's rule for factoring polynomials,

$$f(t) = at^3 + bt^2 + ct + d = ((at + b)t + c)t + d, \tag{11.23}$$

reduces the effort slightly to nine multiplies and 10 additions per 3D point. In Section 11.2.9, we shall examine much more efficient ways to display these curves.

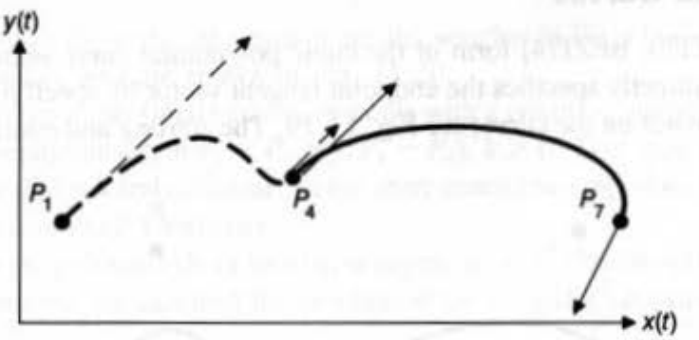


Fig. 11.17 Two Hermite curves joined at P_4. The tangent vectors at P_4 have the same direction but different magnitudes, yielding G^1 but not C^1 continuity.

```
typedef double CoefficientArray[4];
void DrawCurve (
        CoefficientArray cx,          /* Coefficients for x(t): C_x = M • G_x */
        CoefficientArray cy,          /* Coefficients for y(t): C_y = M • G_y */
        CoefficientArray cz,          /* Coefficients for z(t): C_z = M • G_z */
        int n)                        /* Number of steps */
{
    int i;
    double δ = 1.0 / n;
    double t = 0;

    MoveAbs3 (cx[3], cy[3], cz[3]);   /* t = 0: start at x(0), y(0), z(0) */
    for (i = 0; i < n; i++) {
        double t2, t3, x, y, z;

        t += δ;
        t2 = t * t;
        t3 = t2 * t;
        x = cx[0] * t3 + cx[1] * t2 + cx[2] * t + cx[3];
        y = cy[0] * t3 + cy[1] * t2 + cy[2] * t + cy[3];
        z = cz[0] * t3 + cz[1] * t2 + cz[2] * t + cz[3];
        DrawAbs3 (x, y, z);
    }
} /* DrawCurve */
```

Fig. 11.18 Program to display a cubic parametric curve.

Because the cubic curves are linear combinations (weighted sums) of the four elements of the geometry vector, as seen in Eq. (11.10), we can transform the curves by transforming the geometry vector and then using it to generate the transformed curve, which is equivalent to saying that the curves are invariant under rotation, scaling, and translation. This strategy is more efficient than is generating the curve as a series of short line segments and then transforming each individual line. The curves are *not* invariant under perspective projection, as will be discussed in Section 11.2.5.

11.2.2 Bézier Curves

The Bézier [BEZI70; BEZI74] form of the cubic polynomial curve segment, named after Pierre Bézier, indirectly specifies the endpoint tangent vector by specifying two intermediate points that are not on the curve; see Fig. 11.19. The starting and ending tangent vectors

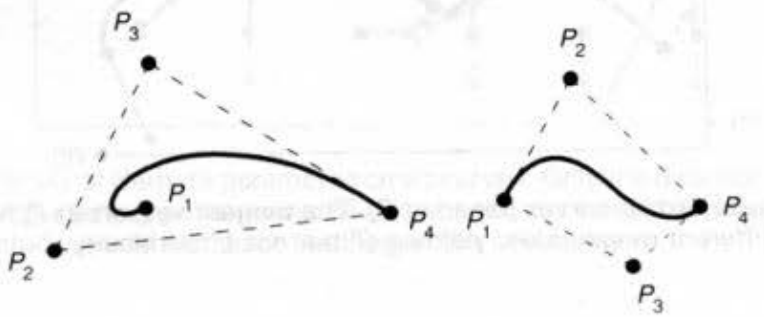


Fig. 11.19 Two Bézier curves and their control points. Notice that the convex hulls of the control points, shown as dashed lines, do not need to touch all four control points.

are determined by the vectors P_1P_2 and P_3P_4 and are related to R_1 and R_4 by

$$R_1 = Q'(0) = 3(P_2 - P_1),\ R_4 = Q'(1) = 3(P_4 - P_3). \tag{11.24}$$

The Bézier curve interpolates the two end control points and approximates the other two. See Exercise 11.12 to understand why the constant 3 is used in Eq. (11.24).

The *Bézier geometry vector* G_B, consisting of four points, is

$$G_B = \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}. \tag{11.25}$$

Then the matrix M_{HB} that defines the relation $G_H = M_{HB} \cdot G_B$ between the Hermite geometry vector G_H and the Bézier geometry vector G_B is just the 4×4 matrix in the following equation, which rewrites Eq. (11.24) in matrix form:

$$G_H = \begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix} = M_{HB} \cdot G_B. \tag{11.26}$$

To find the *Bézier basis matrix* M_B, we use Eq. (11.20) for the Hermite form, substitute $G_H = M_{HB} \cdot G_B$, and define $M_B = M_H \cdot M_{HB}$:

$$Q(t) = T \cdot M_H \cdot G_H = T \cdot M_H \cdot (M_{HB} \cdot G_B) = T \cdot (M_H \cdot M_{HB}) \cdot G_B = T \cdot M_B \cdot G_B. \tag{11.27}$$

Carrying out the multiplication $M_B = M_H \cdot M_{HB}$ gives

$$M_B = M_H \cdot M_{HB} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \tag{11.28}$$

and the product $Q(t) = T \cdot M_B \cdot G_B$ is

$$Q(t) = (1 - t)^3P_1 + 3t(1 - t)^2P_2 + 3t^2(1 - t)P_3 + t^3P_4. \tag{11.29}$$

The four polynomials $B_B = T \cdot M_B$, which are the weights in Eq. (11.29), are called the *Bernstein polynomials*, and are shown in Fig. 11.20.

Figure 11.21 shows two Bézier curve segments with a common endpoint. G^1 continuity is provided at the endpoint when $P_3 - P_4 = k(P_4 - P_5)$, $k > 0$. That is, the three points P_3, P_4, and P_5 must be distinct and collinear. In the more restrictive case when $k = 1$, there is C^1 continuity in addition to G^1 continuity.

If we refer to the polynomials of two curve segments as x^l (for the left segment) and x^r (for the right segment), we can find the conditions for C^0 and C^1 continuity at their join point:

$$x^l(1) = x^r(0),\quad \frac{d}{dt}x^l(1) = \frac{d}{dt}x^r(0). \tag{11.30}$$

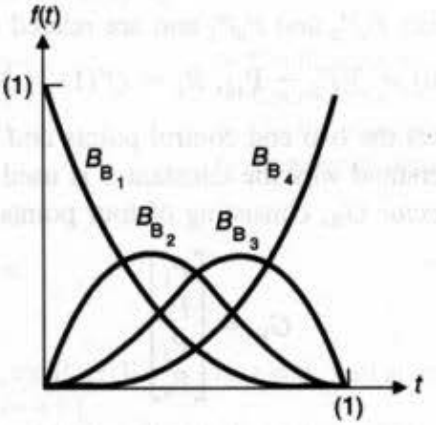


Fig. 11.20 The Bernstein polynomials, which are the weighting functions for Bézier curves. At $t = 0$, only B_{B_1} is nonzero, so the curve interpolates P_1; similarly, at $t = 1$, only B_{B_4} is nonzero, and the curve interpolates P_4.

Working with the x component of Eq. (11.29), we have

$$x^1(1) = x^r(0) = P_{4_x}, \frac{d}{dt}x^1(1) = 3(P_{4_x} - P_{3_x}), \frac{d}{dt}x^r(0) = 3(P_{5_x} - P_{4_x}). \quad (11.31)$$

As always, the same conditions are true of y and z. Thus, we have C^0 and C^1 continuity when $P_4 - P_3 = P_5 - P_4$, as expected.

Examining the four B_B polynomials in Eq. (11.29), we note that their sum is everywhere unity and that each polynomial is everywhere nonnegative for $0 \leq t < 1$. Thus, $Q(t)$ is just a weighted average of the four control points. This condition means that each curve segment, which is just the sum of four control points weighted by the polynomials, is completely contained in the *convex hull* of the four control points. The convex hull for 2D curves is the convex polygon formed by the four control points: Think of it as the polygon formed by putting a rubberband around the points (Fig. 11.19). For 3D curves, the convex hull is the convex polyhedron formed by the control points: Think of it as the polyhedron formed by stretching a rubber sheet around the four points.

This convex-hull property holds for all cubics defined by weighted sums of control points if the blending functions are nonnegative and sum to one. In general, the weighted average of n points falls within the convex hull of the n points; this can be seen intuitively

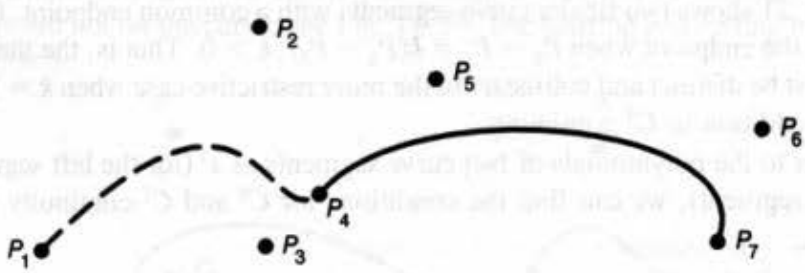


Fig. 11.21 Two Bézier curves joined at P_4. Points P_3, P_4, and P_5 are collinear. Curves are the same as those used in Fig. 11.17.

for $n = 2$ and $n = 3$, and the generalization follows. Another consequence of the fact that the four polynomials sum to unity is that the value of the fourth polynomial for any value of t can be found by subtracting the first three from unity.

The convex-hull property is also useful for clipping curve segments: Rather than clip each short line piece of a curve segment to determine its visibility, we first apply a polygonal clip algorithm to clip the convex hull or its extent against the clip region. If the convex hull (extent) is completely within the clip region, so is the entire curve segment. If the convex hull (extent) is completely outside the clip region, so is the curve segment. Only if the convex hull (extent) intersects the clip region does the curve segment itself need to be examined.

11.2.3 Uniform Nonrational B-Splines

The term *spline* goes back to the long flexible strips of metal used by draftspersons to lay out the surfaces of airplanes, cars, and ships. "Ducks," weights attached to the splines, were used to pull the spline in various directions. The metal splines, unless severely stressed, had second-order continuity. The mathematical equivalent of these strips, the *natural cubic spline*, is a C^0, C^1, and C^2 continuous cubic polynomial that interpolates (passes through) the control points. This is 1 more degree of continuity than is inherent in the Hermite and Bézier forms. Thus, splines are inherently smoother than are the previous forms.

The polynomial coefficients for natural cubic splines, however, are dependent on all n control points; their calculation involves inverting an $n + 1$ by $n + 1$ matrix [BART87]. This has two disadvantages: moving any one control point affects the entire curve, and the computation time needed to invert the matrix can interfere with rapid interactive reshaping of a curve.

B-splines, discussed in this section, consist of curve segments whose polynomial coefficients depend on just a few control points. This is called *local control*. Thus, moving a control point affects only a small part of a curve. In addition, the time needed to compute the coefficients is greatly reduced. B-splines have the same continuity as natural splines, but do not interpolate their control points.

In the following discussion we change our notation slightly, since we must discuss an entire curve consisting of several curve segments, rather than its individual segments. A curve segment need not pass through its control points, and the two continuity conditions on a segment come from the adjacent segments. This is achieved by sharing control points between segments, so it is best to describe the process in terms of all the segments at once.

Cubic B-splines approximate a series of $m + 1$ control points $P_0, P_1, \ldots P_m$, $m \geq 3$, with a curve consisting of $m - 2$ cubic polynomial curve segments $Q_3, Q_4, \ldots Q_m$. Although such cubic curves might be defined each on its own domain $0 \leq t < 1$, we can adjust the parameter (making a substitution of the form $t = t + k$) so that the parameter domains for the various curve segments are sequential. Thus, we say that the parameter range on which Q_i is defined is $t_i \leq t < t_{i+1}$, for $3 \leq i \leq m$. In the particular case of $m = 3$, there is a single curve segment Q_3 that is defined on the interval $t_3 \leq t < t_4$ by four control points, P_0 to P_3.

For each $i \geq 4$, there is a join point or *knot* between Q_{i-1} and Q_i at the parameter value t_i; the parameter value at such a point is called a *knot value*. The initial and final points at t_3

and t_{m+1} are also called knots, so that there is a total of $m - 1$ knots. Figure 11.22 shows a 2D B-spline curve with its knots marked. A closed B-spline curve is easy to create: The control points P_0, P_1, P_2 are repeated at the end of the sequence—P_0, P_1, . . . P_m, P_0, P_1, P_2.

The term *uniform* means that the knots are spaced at equal intervals of the parameter t. Without loss of generality, we can assume that $t_3 = 0$ and the interval $t_{i+1} - t_i = 1$. Nonuniform nonrational B-splines, which permit unequal spacing between the knots, are discussed in Section 11.2.4. (In fact, the concept of knots is introduced in this section to set the stage for nonuniform splines.) The term *nonrational* is used to distinguish these splines from rational cubic polynomial curves, discussed in Section 11.2.5, where $x(t)$, $y(t)$, and $z(t)$ are each defined as the ratio of two cubic polynomials. The "B" stands for basis, since the splines can be represented as weighted sums of polynomial basis functions, in contrast to the natural splines, for which this is not true.

Each of the $m - 2$ curve segments of a B-spline curve is defined by four of the $m + 1$ control points. In particular, curve segment Q_i is defined by points P_{i-3}, P_{i-2}, P_{i-1}, and P_i. Thus, the *B-spline geometry vector* G_{Bs_i} for segment Q_i is

$$G_{\mathrm{Bs}_i} = \begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix}, \quad 3 \le i \le m. \tag{11.32}$$

The first curve segment, Q_3, is defined by the points P_0 through P_3 over the parameter range $t_3 = 0$ to $t_4 = 1$, Q_4 is defined by the points P_1 through P_4 over the parameter range $t_4 = 1$ to $t_5 = 2$, and the last curve segment, Q_m, is defined by the points P_{m-3}, P_{m-2}, P_{m-1}, and P_m over the parameter range $t_m = m - 3$ to $t_{m+1} = m - 2$. In general, curve segment Q_i begins somewhere near point P_{i-2} and ends somewhere near point P_{i-1}. We shall see that the B-spline blending functions are everywhere nonnegative and sum to unity, so the curve segment Q_i is constrained to the convex hull of its four control points.

Just as each curve segment is defined by four control points, each control point (except for those at the beginning and end of the sequence P_0, P_1, . . . P_m) influences four curve

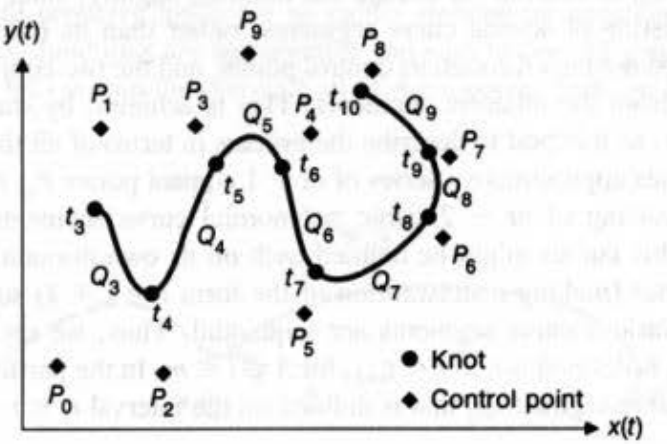


Fig. 11.22 A B-spline with curve segments Q_3 through Q_9. This and many other figures in this chapter were created with a program written by Carles Castellsaquè.

segments. Moving a control point in a given direction moves the four curve segments it affects in the same direction; the other curve segments are totally unaffected (see Fig. 11.23). This is the local control property of B-splines and of all the other splines discussed in this chapter.

If we define T_i as the row vector $[(t - t_i)^3 \quad (t - t_i)^2 \quad (t - t_i) \quad 1]$, then the B-spline formulation for curve segment i is

$$Q_i(t) = T_i \cdot M_{Bs} \cdot G_{Bs_i}, \quad t_i \le t < t_{i+1}. \tag{11.33}$$

The entire curve is generated by applying Eq. (11.33) for $3 \le i \le m$.

The *B-spline basis matrix*, M_{Bs}, relates the geometrical constraints G_{Bs} to the blending functions and the polynomial coefficients:

$$M_{Bs} = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}. \tag{11.34}$$

This matrix is derived in [BART87].

The B-spline blending functions B_{Bs} are given by the product $T_i \cdot M_{Bs}$, analogously to the previous Bézier and Hermite formulations. Note that the blending functions for each curve segment are exactly the same, because for each segment i the values of $t - t_i$ range from 0 at $t = t_i$ to 1 at $t = t_{i+1}$. If we replace $t - t_i$ by t, and replace the interval $[t_i, t_{i+1}]$ by $[0, 1]$, we have

$$\begin{aligned} B_{Bs} = T \cdot M_{Bs} &= [B_{Bs_{-3}} \quad B_{Bs_{-2}} \quad B_{Bs_{-1}} \quad B_{Bs_0}] \\ &= \frac{1}{6}[-t^3 + 3t^2 - 3t + 1 \quad 3t^3 - 6t^2 + 4 \quad -3t^3 + 3t^2 + 3t + 1 \quad t^3] \\ &= \frac{1}{6}[(1 - t)^3 \quad 3t^3 - 6t^2 + 4 \quad -3t^3 + 3t^2 + 3t + 1 \quad t^3], \ 0 \le t < 1. \end{aligned} \tag{11.35}$$

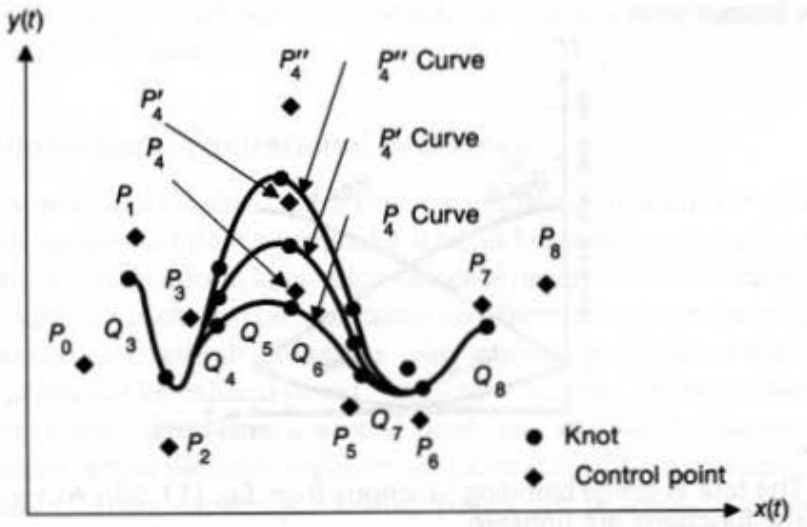


Fig. 11.23 A B-spline with control point P_4 in several different locations.

Figure 11.24 shows the B-spline blending functions B_{Bs}. Because the four functions sum to 1 and are nonnegative, the convex-hull property holds for each curve segment of a B-spline. See [BART87] to understand the relation between these blending functions and the Bernstein polynomial basis functions.

Expanding Eq. (11.33), again replacing $t - t_i$ with t at the second equals sign, we have

$$\begin{aligned} Q_i(t - t_i) &= T_i \cdot M_{Bs} \cdot G_{Bs_i} = T \cdot M_{Bs} \cdot G_{Bs_i} \\ &= B_{Bs} \cdot G_{Bs_i} = B_{Bs_{-3}} \cdot P_{i-3} + B_{Bs_{-2}} \cdot P_{i-2} + B_{Bs_{-1}} \cdot P_{i-1} + B_{Bs_0} \cdot P_i \\ &= \frac{(1-t)^3}{6} P_{i-3} + \frac{3t^3 - 6t^2 + 4}{6} P_{i-2} + \frac{-3t^3 + 3t^2 + 3t + 1}{6} P_{i-1} \\ &\quad + \frac{t^3}{6} P_i, \; 0 \le t < 1. \end{aligned} \tag{11.36}$$

It is easy to show that Q_i and Q_{i+1} are C^0, C^1, and C^2 continuous where they join. When we consider the x components of the adjacent segments, which are $x_i(t - t_i)$ and $x_{i+1}(t - t_{i+1})$ (y and z, as always, are analogous), it is necessary to show only that, at the knot t_{i+1} where they join,

$$x_i(t_{i+1}) = x_{i+1}(t_{i+1}), \quad \frac{d}{dt}x_i(t_{i+1}) = \frac{d}{dt}x_{i+1}(t_{i+1}), \quad \text{and} \quad \frac{d^2}{dt^2}x_i(t_{i+1}) = \frac{d^2}{dt^2}x_{i+1}(t_{i+1}). \tag{11.37}$$

Recalling the substitution of t for $t - t_i$, Eq. (11.37) is equivalent to showing that

$$\begin{aligned} x_i\big|_{t-t_i=1} &= x_{i+1}\big|_{t-t_{i+1}=0}, \\ \frac{d}{dt}x_i\big|_{t-t_i=1} &= 1\,\frac{d}{dt}x_{i+1}\big|_{t-t_{i+1}=0}, \\ \frac{d^2}{dt^2}x_i\big|_{t-t_i=1} &= \frac{d^2}{dt^2}\,x_{i+1}\big|_{t-t_{i+1}=0}. \end{aligned} \tag{11.38}$$

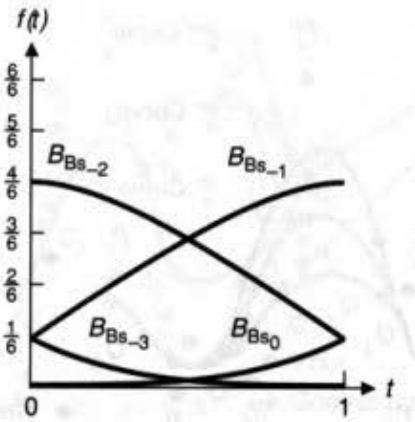


Fig. 11.24 The four B-spline blending functions from Eq. (11.35). At $t = 0$ and $t = 1$, just three of the functions are nonzero.

We demonstrate the equivalence by working with the x component of Eq. (11.36), and its first and second derivatives, to yield

$$x_i|_{t-t_i=1} = x_{i+1}|_{t-t_i=1} = \frac{P_{i-2_x} + 4P_{i-1_x} + P_{i_x}}{6}, \tag{11.39}$$

$$\frac{d}{dt}x_i|_{t-t_i=1} = \frac{d}{dt}x_{i+1}|_{t-t_{i+1}=0} = \frac{-P_{i-2_x} + P_{i_x}}{2} \tag{11.40}$$

$$\frac{d^2}{dt^2}x_i|_{t-t_i=1} = \frac{d^2}{dt^2}x_{i+1}|_{t-t_{i+1}=0} = P_{i-2_x} - 2P_{i-1_x} + P_{i_x}. \tag{11.41}$$

The additional continuity afforded by B-splines is attractive, but it comes at the cost of less control of where the curve goes. The curve can be forced to interpolate specific points by replicating control points; this is useful both at endpoints and at intermediate points on the curve. For instance, if $P_{i-2} = P_{i-1}$, the curve is pulled closer to this point because curve segment Q_i is defined by just three different points, and the point $P_{i-2} = P_{i-1}$ is weighted twice in Eq. (11.36)—once by $B_{\mathrm{Bs}_{-2}}$ and once by $B_{\mathrm{Bs}_{-1}}$.

If a control point is used three times—for instance, if $P_{i-2} = P_{i-1} = P_i$—then Eq. (11.36) becomes

$$Q_i(t) = B_{\mathrm{Bs}_{-3}} \cdot P_{i-3} + (B_{\mathrm{Bs}_{-2}} + B_{\mathrm{Bs}_{-1}} + B_{\mathrm{Bs}_0}) \cdot P_i. \tag{11.42}$$

Q_i is clearly a straight line. Furthermore, the point P_{i-2} is interpolated by the line at $t = 1$, where the three weights applied to P_i sum to 1, but P_{i-3} is not in general interpolated at $t = 0$. Another way to think of this is that the convex hull for Q_i is now defined by just two distinct points, so Q_i has to be a line. Figure 11.25 shows the effect of multiple control points at the interior of a B-spline. The price of interpolating the points in part (*c*) is loss of G^1 continuity, even though Eq. (11.40) shows that C^1 continuity is preserved (but with a value of zero). This is a case where C^1 continuity does not imply G^1 continuity, as discussed in Section 11.2.

Another technique for interpolating endpoints, *phantom vertices*, is discussed in [BARS83; BART87]. We shall see that, with nonuniform B-splines, discussed in the next section, endpoints and internal points can be interpolated in a more natural way than they can with the uniform B-splines.

11.2.4 Nonuniform, Nonrational B-Splines

Nonuniform, nonrational B-splines differ from the uniform, nonrational B-splines discussed in the previous section in that the parameter interval between successive knot values need not be uniform. The nonuniform knot-value sequence means that the blending functions are no longer the same for each interval, but rather vary from curve segment to curve segment.

These curves have several advantages over uniform B-splines. First, continuity at selected join points can be reduced from C^2 to C^1 to C^0 to none. If the continuity is reduced to C^0, then the curve interpolates a control point, but without the undesirable effect of uniform B-splines, where the curve segments on either side of the interpolated control point are straight lines. Also, starting and ending points can be easily interpolated exactly,

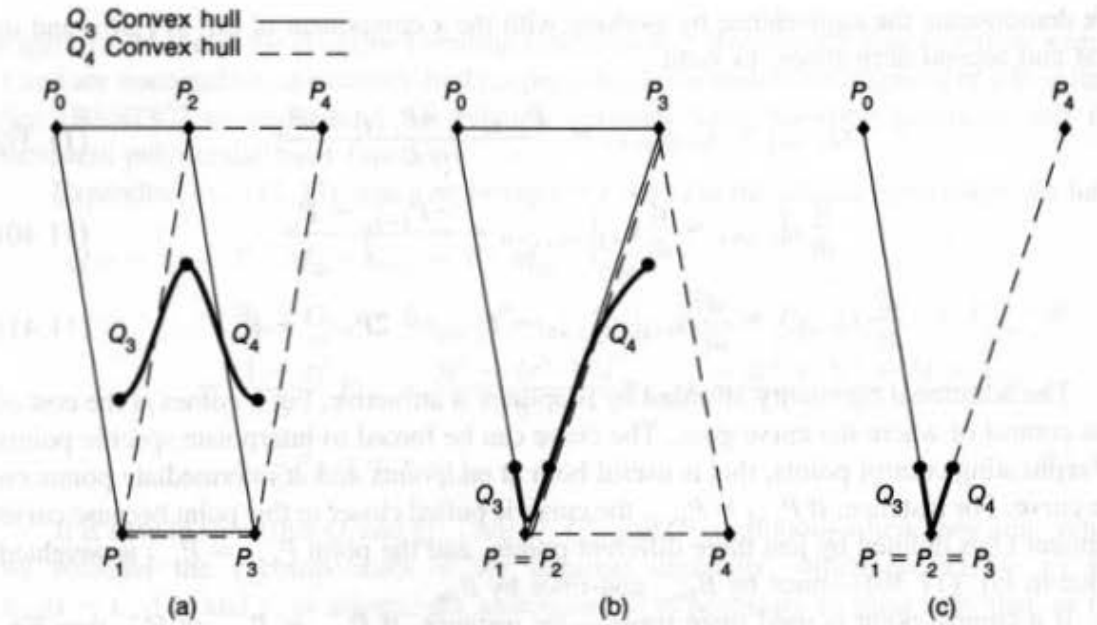


Fig. 11.25 The effect of multiple control points on a uniform B-spline curve. In (a), there are no multiple control points. The convex hulls of the two curves overlap; the join point between Q_3 and Q_4 is in the region shared by both convex hulls. In (b), there is a double control point, so the two convex hulls share edge P_2P_3; the join point is therefore constrained to lie on this edge. In (c), there is a triple control point, and the two convex hulls are straight lines that share the triple point; hence, the join point is also at the triple point. Because the convex hulls are straight lines, the two curve segments must also be straight lines. There is C^2 but only G^0 continuity at the join.

without at the same time introducing linear segments. As is further discussed in Section 11.2.7, it is possible to add an additional knot and control point to nonuniform B-splines, so the resulting curve can be easily reshaped, whereas this cannot be done with uniform B-splines.

The increased generality of nonuniform B-splines requires a slightly different notation than that used for uniform B-splines. As before, the spline is a piecewise continuous curve made up of cubic polynomials, approximating the control points P_0 through P_m. The *knot-value sequence* is a nondecreasing sequence of knot values t_0 through t_{m+4} (that is, there are four more knots than there are control points). Because the smallest number of control points is four, the smallest knot sequence has eight knot values and the curve is defined over the parameter interval from t_3 to t_4.

The only restriction on the knot sequence is that it be nondecreasing, which allows successive knot values to be equal. When this occurs, the parameter value is called a *multiple knot* and the number of identical parameter values is called the *multiplicity* of the knot (a single unique knot has multiplicity of 1). For instance, in the knot sequence (0, 0, 0, 0, 1, 1, 2, 3, 4, 4, 5, 5, 5, 5), the knot value 0 has multiplicity four; value 1 has multiplicity 2; values 2 and 3 have multiplicity 1; value 4 has multiplicity 2; and value 5 has multiplicity 4.

Curve segment Q_i is defined by control points P_{i-3}, P_{i-2}, P_{i-1}, P_i and by blending functions $B_{i-3,4}(t)$, $B_{i-2,4}(t)$, $B_{i-1,4}(t)$, $B_{i,4}(t)$, as the weighted sum

$$Q_i(t) = P_{i-3} \cdot B_{i-3,4}(t) + P_{i-2} \cdot B_{i-2,4}(t) + P_{i-1} \cdot B_{i-1,4}(t) + P_i \cdot B_{i,4}(t)$$
$$3 \le i \le m, \quad t_i \le t < t_{i+1}. \tag{11.43}$$

The curve is not defined outside the interval t_3 through t_{m+1}. When $t_i = t_{i+1}$ (a multiple knot), curve segment Q_i is a single point. It is this notion of a curve segment reducing to a point that provides the extra flexibility of nonuniform B-splines.

There is no single set of blending functions, as there was for other types of splines. The functions depend on the intervals between knot values and are defined recursively in terms of lower-order blending functions. $B_{i,j}(t)$ is the jth-order blending function for weighting control point P_i. Because we are working with fourth-order (that is, third-degree, or cubic) B-splines, the recursive definition ends with $B_{i,4}(t)$ and can be easily presented in its "unwound" form. The recurrence for cubic B-splines is

$$B_{i,1}(t) = \begin{cases} 1, & t_i \le t < t_{i+1} \\ 0, & \text{otherwise,} \end{cases}$$

$$B_{i,2}(t) = \frac{t - t_i}{t_{i+1} - t_i} B_{i,1}(t) + \frac{t_{i+2} - t}{t_{i+2} - t_{i+1}} B_{i+1,1}(t),$$

$$B_{i,3}(t) = \frac{t - t_i}{t_{i+2} - t_i} B_{i,2}(t) + \frac{t_{i+3} - t}{t_{i+3} - t_{i+1}} B_{i+1,2}(t),$$

$$B_{i,4}(t) = \frac{t - t_i}{t_{i+3} - t_i} B_{i,3}(t) + \frac{t_{i+4} - t}{t_{i+4} - t_{i+1}} B_{i+1,3}(t). \tag{11.44}$$

Figure 11.26 shows how Eq. (11.44) can be used to find the blending functions, using the knot vector (0, 0, 0, 0, 1, 1, 1, 1) as an example. The figure also makes clear why eight knot vectors are needed to compute four blending functions. $B_{3,1}(t)$ is unity on the interval $0 \le t < 1$. All other $B_{i,1}(t)$ are zero. $B_{2,2}(t)$ and $B_{3,2}(t)$ are linear ramps, and are the blending functions for linear interpolation between two points. Similarly, $B_{1,3}(t)$, $B_{2,3}(t)$, and $B_{3,3}(t)$ are quadratics, and are the blending functions for quadratic interpolation. For this particular knot vector, the $B_{i,4}(t)$ are the Bernstein polynomials, that is, the Bezier blending functions; compare them to those in Fig. 11.20. Also, for this knot vector, the curve interpolates the control points P_0 and P_3, and is in fact a Bézier curve, with the tangent vector at the endpoints determined by the vectors P_0P_1 and P_2P_3.

Computing the blending functions takes time. By restricting B-spline knot sequences to have intervals that are either 0 or 1, it is possible to store just a small number of matrices corresponding to Eq. (11.44), which covers all such possible knot configurations. This eliminates the need to reevaluate Eq. (11.44) for each curve segment.

It can be shown that the blending functions are nonnegative and sum to one, so nonuniform B-spline curve segments lie within the convex hulls of their four control points. For knots of multiplicity greater than one, the denominators can be zero because successive knot values can be equal: division by zero is defined to yield zero.

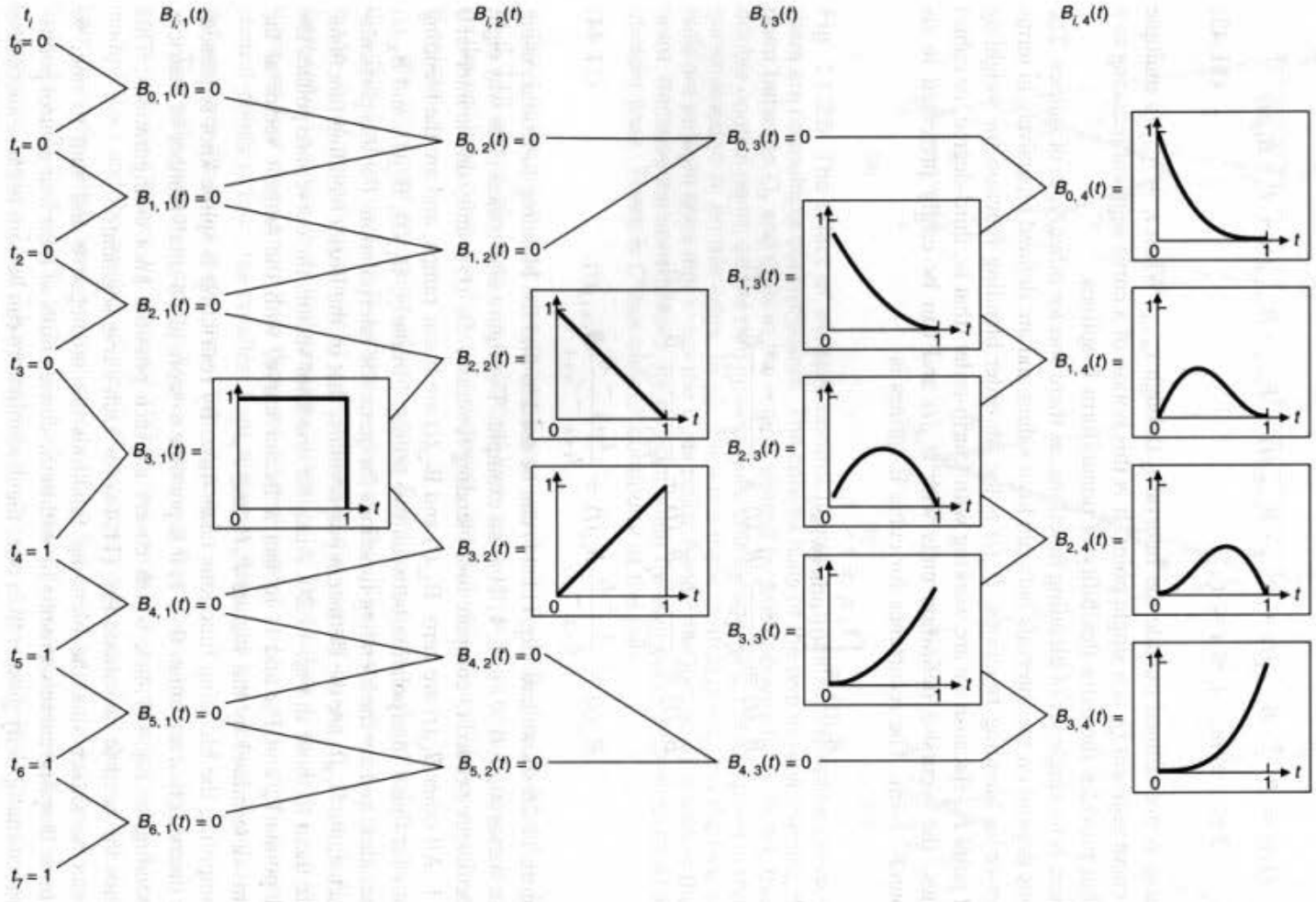


Fig. 11.26 The relationships defined by Eq. (11.44) between the knot vector (0, 0, 0, 0, 1, 1, 1, 1) and the blending functions $B_{i,1}(t)$, $B_{i,2}(t)$, $B_{i,3}(t)$, and $B_{i,4}(t)$.

Increasing knot multiplicity has two effects. First, each knot value t_i will automatically lie within the convex hull of the points P_{i-3}, P_{i-2}, and P_{i-1}. If t_i and t_{i+1} are equal, they must lie in the convex hull of P_{i-3}, P_{i-2}, and P_{i-1}, *and* in the convex hull of P_{i-2}, P_{i-1}, and P_i. This means they must actually lie on the line segment between P_{i-2} and P_{i-1}. In the same way, if $t_i = t_{i+1} = t_{i+2}$, then this knot must lie *at* P_{i-1}. If $t_i = t_{i+1} = t_{i+2} = t_{i+3}$, then the knot must lie both at P_{i-1} and at P_i—the curve becomes broken. Second, the multiple knots will reduce parametric continuity: from C^2 to C^1 continuity for one extra knot (multiplicity 2); from C^1 to C^0 continuity for two extra knots (multiplicity 3); from C^0 to no continuity for three extra knots (multiplicity 4).

Figure 11.27 provides further insight for a specific case. Part (a) shows the case when all knots have multiplicity 1. Each curve segment is defined by four control points and four blending functions, and adjacent curve segments each share three control points. For instance, curve segment Q_3 is defined by points P_0, P_1, P_2, and P_3; curve segment Q_4 is defined by points P_1, P_2, P_3 and P_4; and curve segment Q_5 is defined by points P_2, P_3, P_4, and P_5. Part (b) shows a double knot, $t_4 = t_5$, for which the curve segment Q_4 has zero length. Segments Q_3 and Q_5 are thus adjacent but share only two control points, P_2 and P_3; the two curve segments hence have less "in common," as implied by the loss of 1 degree of continuity. For the triple knot in part (c), only control point P_3 is shared in common: the one that the two curve segments now interpolate. Because only one control point is shared, we can expect only one constraint, C^0 continuity, to be satisfied at the join. The knot of multiplicity 4, shown in part (d), causes a discontinuity, or break, in the curve. Hence, several disjoint splines can be represented by a single knot sequence and set of control points. Figure 11.28 provides additional understanding of the relations among knots, curve segments, and control points. Table 11.1 summarizes the effects of multiple control points and multiple knots.

TABLE 11.1 COMPARISON OF THE EFFECTS OF MULTIPLE CONTROL POINTS AND OF MULTIPLE KNOTS

Multiplicity	Multiple control points	Multiple knots
1	C^2 G^{2*}	C^2 G^{2*}
2	C^2 G^1 Knots constrained to a smaller convex hull.	C^1 G^1 Knots constrained to a smaller convex hull of fewer control points.
3	C^2 G^0 Curve interpolates the triple control point. Curve segments on either side of the join are linear.	C^0 G^0 Curve interpolates control point. Can control shape of curve segments on either side of the join.
4	C^2 G^0 Curve interpolates the quadruple control points. Curve segments on either side of the join are linear and interpolate the control points on either side of the join.	There is a discontinuity in the curve. Curve stops on one control point, resumes at next. Can control shape of curve segments on either side of the discontinuity.

*Except for special case discussed in Section 11.2.

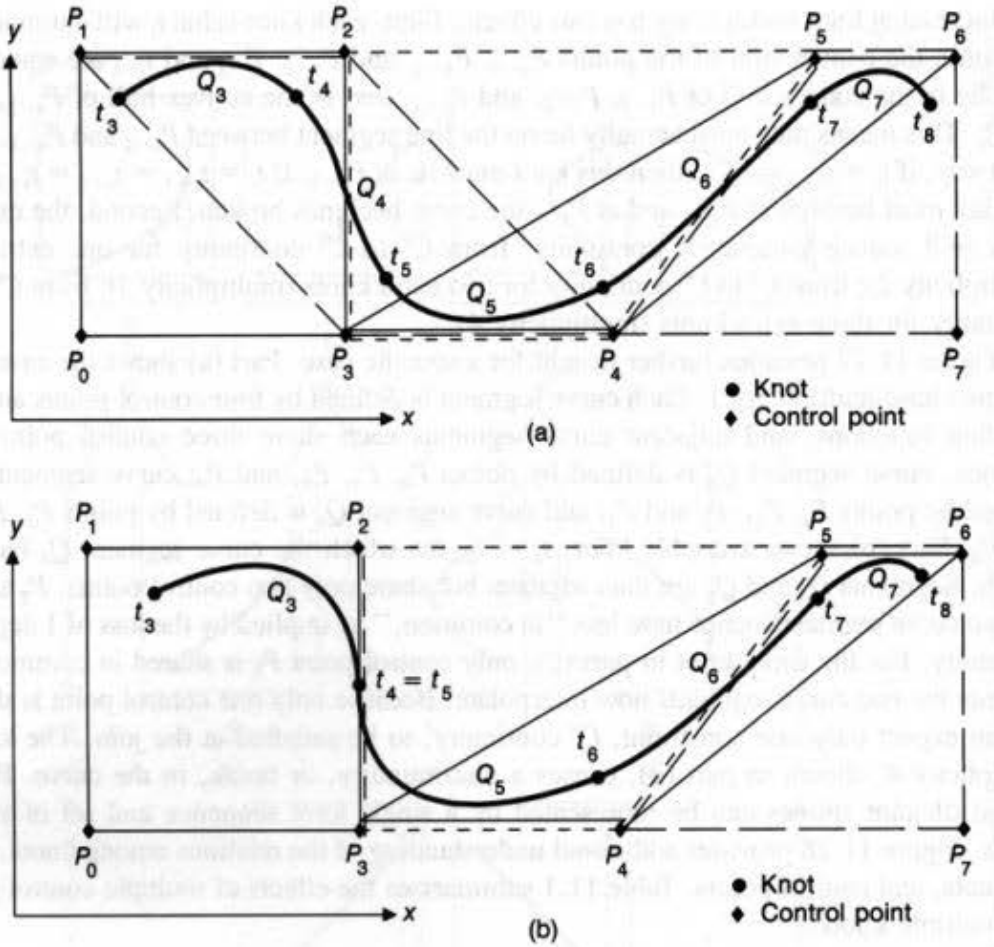
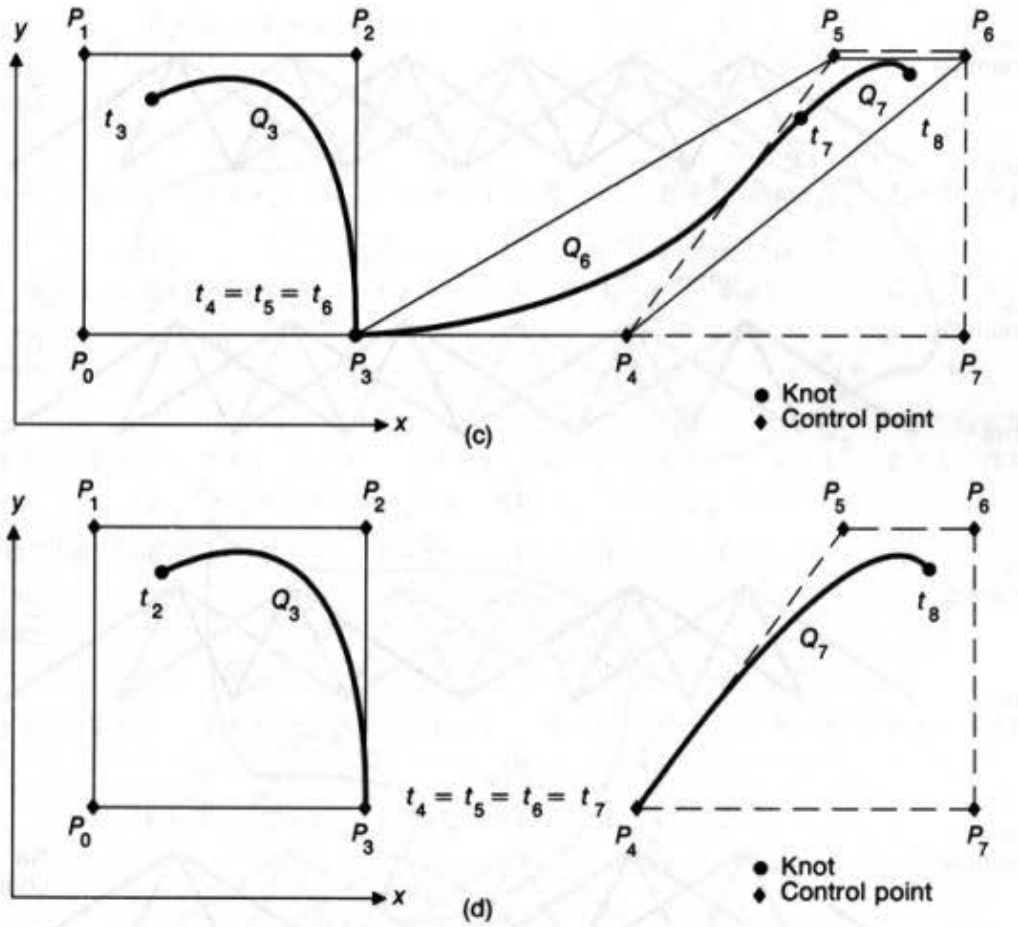


Fig. 11.27 The effect of multiple knots. In (a), with knot sequence (0, 1, 2, 3, 4, 5), there are no multiple knots; all the curve segments join with C^2 and G^2 continuity. The convex hulls containing each curve segment are also shown. In (b), with knot sequence (0, 1, 1, 2, 3, 4), there is a double knot, so curve segment Q_4 degenerates to a point. The convex hulls containing Q_3 and Q_5 meet along the edge P_2P_3, on which the join point is forced to lie. The join has C^1 and G^2 continuity. In (c), with knot sequence (0, 1, 1, 1,

Figure 11.29 illustrates the complexity of shapes that can be represented with this technique. Notice part (a) of the figure, with knot sequence (0, 0, 0, 0, 1, 1, 1, 1): The curve interpolates the endpoints but not the two intermediate points, and is a Bézier curve. The other two curves also start and stop with triple knots. This causes the tangent vectors at the endpoints to be determined by the vectors P_0P_1 and $P_{m-1}P_m$, giving Bézier-like control to the curves at the start and stop points.

Interactive creation of nonuniform splines typically involves pointing at control points, with multiple control points indicated simply by successive selection of the same point. Figure 11.30 shows a way of specifying knot values interactively. Another way is to point directly at the curve with a multibutton mouse: A double click on one button can indicate a double control point; a double click on another button, a double knot.



2, 3), there is a triple knot, so curve segments Q_4 and Q_5 degenerate to points. The convex hulls containing Q_3 and Q_6 meet only at P_3, where the join point is forced to be located. The two curve segments share only control point P_3, with C_0 continuity. In (d), with knot sequence (0, 1, 1, 1, 1, 2), there is a quadruple knot, which causes a discontinuity in the curve because curve segments Q_3 and Q_7 have no control points in common.

11.2.5 Nonuniform, Rational Cubic Polynomial Curve Segments

General rational cubic curve segments are ratios of polynomials:

$$x(t) = \frac{X(t)}{W(t)}, \quad y(t) = \frac{Y(t)}{W(t)}, \quad z(t) = \frac{Z(t)}{W(t)}, \tag{11.45}$$

where $X(t)$, $Y(t)$, $Z(t)$, and $W(t)$ are all cubic polynomial curves whose control points are defined in homogeneous coordinates. We can also think of the curve as existing in homogeneous space as $Q(t) = [X(t) \quad Y(t) \quad Z(t) \quad W(t)]$. As always, moving from homogeneous space to 3-space involves dividing by $W(t)$. Any nonrational curve can be transformed to a rational curve by adding $W(t) = 1$ as a fourth element. In general, the

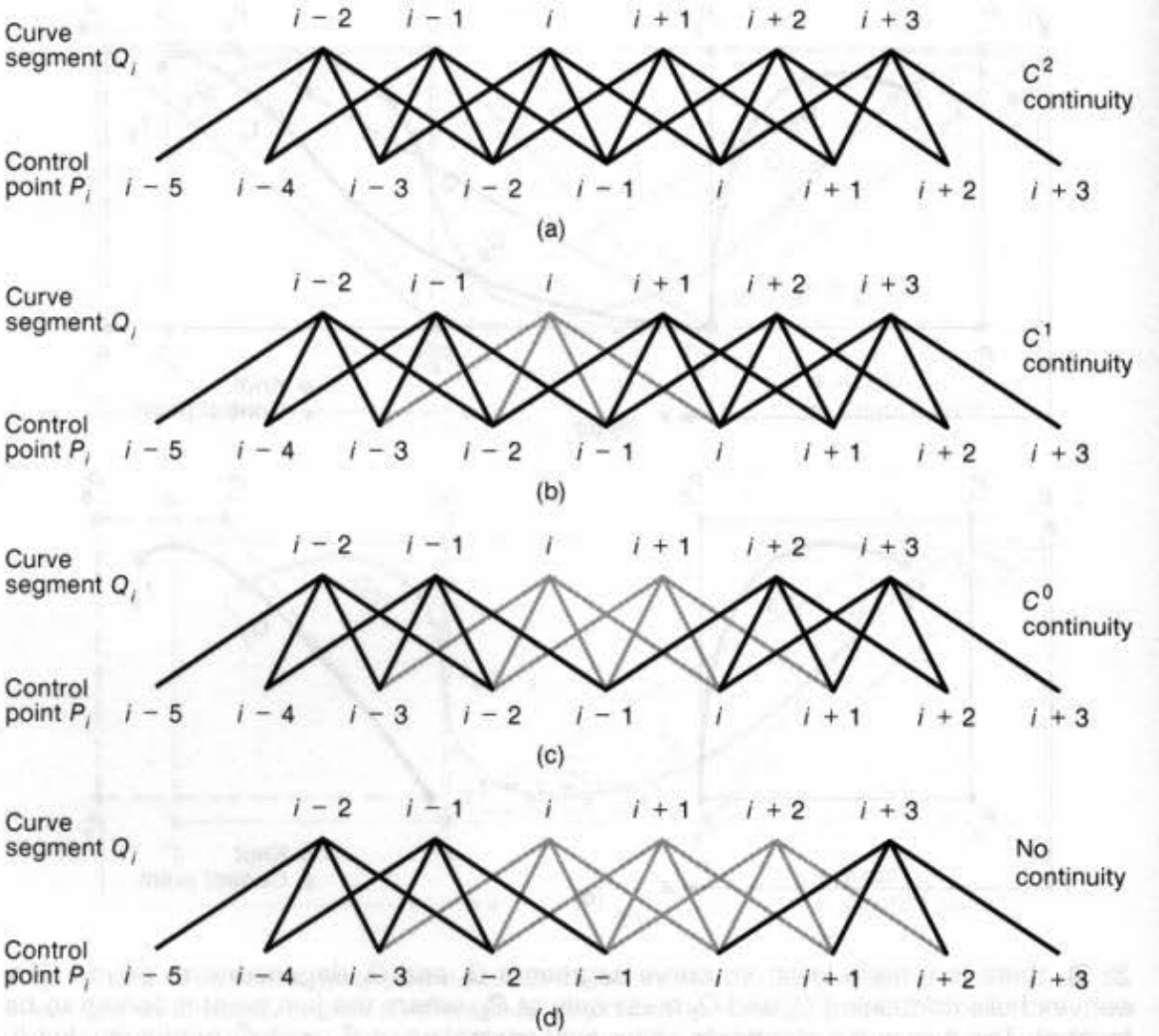


Fig. 11.28 The relationship among curve segments, control points, and multiple knots for nonuniform B-splines. Lines connect curve segments to their control points; gray lines are used for curve segments that do not appear because their knot interval is zero (i.e., the knot multiplicity is greater than one), causing them to have zero length. In (a), all knots are single. In (b), there is a double knot, so segment i is not drawn. In (c), there is a triple knot, so two segments are not drawn; thus, the single point, $i - 1$, is held in common between adjacent segments $i - 1$ and $i + 2$. In (d), with a quadruple knot, segments $i - 1$ and $i + 3$ have no points in common, causing the curve to be disconnected between points $i - 1$ and i.

polynomials in a rational curve can be Bézier, Hermite, or any other type. When they are B-splines, we have nonuniform rational B-splines, sometimes called *NURBS* [FORR80].

Rational curves are useful for two reasons. The first and most important reason is that they are invariant under rotation, scaling, translation and perspective transformations of the control points (nonrational curves are invariant under only rotation, scaling, and translation). This means that the perspective transformation needs to be applied to only the control

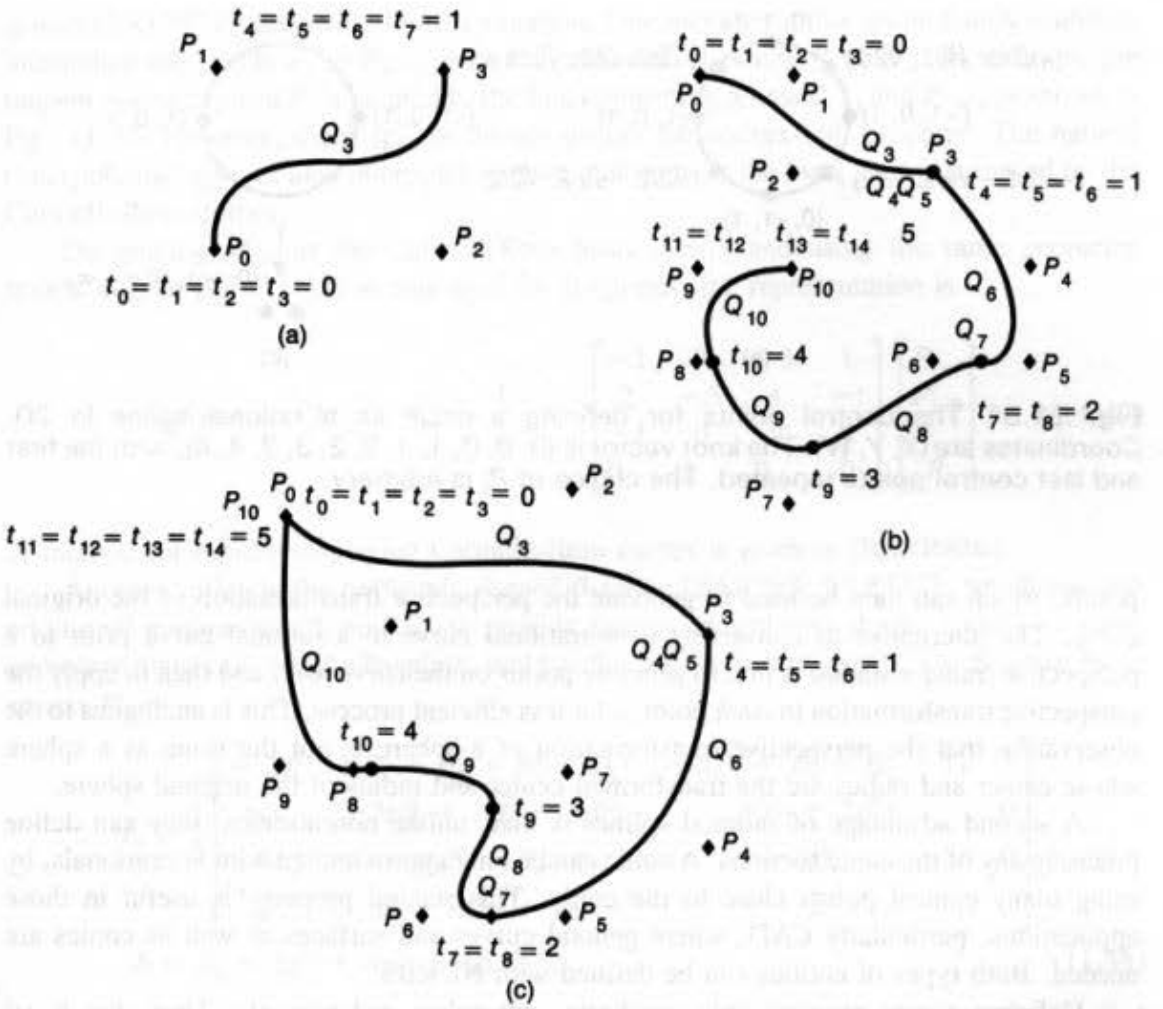


Fig. 11.29 Examples of shapes defined using nonrational B-splines and multiple knots. Part (a) is just a Bézier curve segment, with knot sequence (0, 0, 0, 0, 1, 1, 1, 1), and hence just one curve segment, Q_3. Parts (b) and (c) have the same knot sequence, (0, 0, 0, 0, 1, 1, 1, 2, 2, 3, 4, 5, 5, 5, 5) but different control points. Each curve has curve segments Q_3 to Q_{10}. Segments Q_4, Q_5, and Q_7 are located at multiple knots and have zero length.

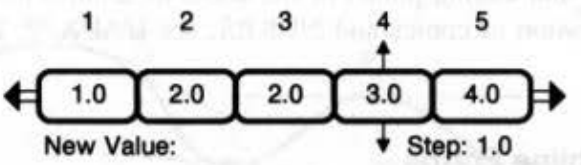


Fig. 11.30 An interaction technique developed by Carles Castellsaguè for specifying knot values. The partial knot sequence is shown, and can be scrolled left and right with the horizontal arrows. One knot value, selected with the cursor, can be incremented up and down using the vertical arrows, in increments specified by value Step. The selected knot value can also be replaced with a new typed-in value.

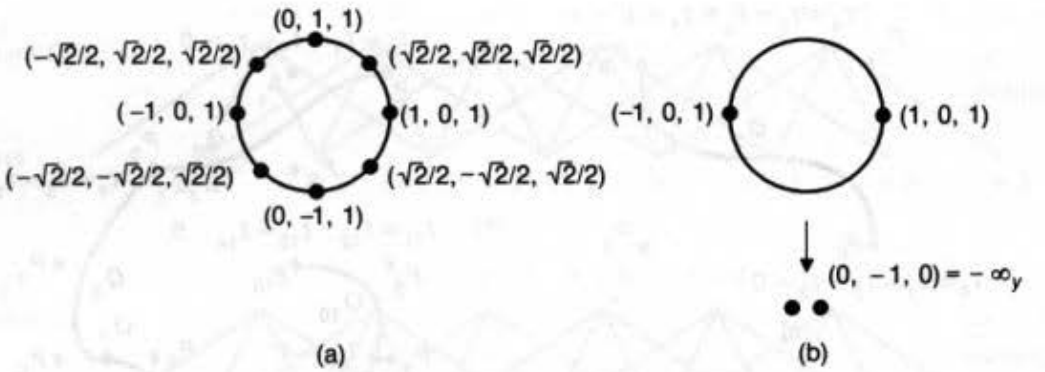


Fig. 11.31 The control points for defining a circle as a rational spline in 2D. Coordinates are (X, Y, W). The knot vector is (0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4), with the first and last control points repeated. The choice of P_0 is arbitrary.

points, which can then be used to generate the perspective transformation of the original curve. The alternative to converting a nonrational curve to a rational curve prior to a perspective transformation is first to generate points on the curve itself and then to apply the perspective transformation to *each* point, a far less efficient process. This is analogous to the observation that the perspective transformation of a sphere is not the same as a sphere whose center and radius are the transformed center and radius of the original sphere.

A second advantage of rational splines is that, unlike nonrationals, they can define precisely any of the conic sections. A conic can be only approximated with nonrationals, by using many control points close to the conic. This second property is useful in those applications, particularly CAD, where general curves and surfaces as well as conics are needed. Both types of entities can be defined with NURBS.

Defining conics requires only quadratic, not cubic, polynomials. Thus, the $B_{i,3}(t)$ blending functions from the recurrence Eq. (11.44) are used in the curve of the form

$$Q_i(t) = P_{i-2}B_{i-2,3}(t) + P_{i-1}B_{i-1,3}(t) + P_iB_{i,3}(t)$$
$$2 \le i \le m, \quad t_i \le t < t_{i+1}. \tag{11.46}$$

Two ways of creating a unit circle centered at the origin are shown in Fig. 11.31. Note that, with quadratic B-splines, a double knot causes a control point to be interpolated, and triple knots fix the starting and ending points of the curve on control points.

For further discussion of conics and NURBS, see [FAUX79; BÖHM84; TILL83].

11.2.6 Other Spline Forms

Very often, we have a series of positions and want a curve smoothly to interpolate (pass through) them. This might arise with a series of points read from a data tablet or mouse, or a series of 3D points through which a curve or camera path is to pass. The Catmull–Rom family of interpolating or approximating splines [CATM74a], also called *Overhauser*