Computer Graphics

PRINCIPLES AND PRACTICE

Foley • van Dam • Feiner • Hughes

SECOND EDITION in C



THE SYSTEMS PROGRAMMING SERIES

EX-1021
Microsoft Inc. v. LiTL LLC

THE
SYSTEMS
PROGRAMMING
SERIES

Computer Graphics: Principles and Practice

SECOND EDITION in C

Foley • van Dam • Feiner • Hughes

SECOND EDITION IN C

Computer Graphics

PRINCIPLES AND PRACTICE

James D. Foley
Georgia Institute of Technology

Andries van Dam
Brown University

Steven K. Feiner
Columbia University

John F. Hughes
Brown University

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

Sponsoring Editor: Peter S. Gordon
Production Supervisor: Bette J. Aaronson
Production Supervisor for the C edition: Juliet Silveri
Copy Editor: Lyn Dupré
Text Designer: Herb Caswell
Technical Art Consultant: Joseph K. Vetere
Illustrators: C&C Associates
Cover Designer: Marshall Henrichs
Manufacturing Manager: Roy Logan

This book is in the **Addison-Wesley Systems Programming Series**
Consulting editors: IBM Editorial Board

Library of Congress Cataloging-in-Publication Data

Computer graphics: principles and practice / James D. Foley . . . [et al.]. — 2nd ed. in C.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-84840-6
1. Computer graphics. I. Foley, James D., 1942–
T385.C5735 1996
006.6'6—dc20 95-13631
CIP

Reprinted with corrections, July 1997.

Cover: "Dutch Interior," after Vermeer, by J. Wallace, M. Cohen, and D. Greenberg, Cornell University (Copyright © 1987 Cornell University, Program of Computer Graphics.)

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

The programs and applications presented in this book have been included for their instructional value. They are not guaranteed for any particular purpose. The publisher and the author do not offer any warranties or representations, nor do they accept any liabilities with respect to the programs or applications.

Reprinted with corrections November 1992, November 1993, and July 1995.

Copyright © 1996, 1990 by Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.

6 7 8 9 10-DOC-99 98 97

1 Introduction

Computer graphics started with the display of data on hardcopy plotters and cathode ray tube (CRT) screens soon after the introduction of computers themselves. It has grown to include the creation, storage, and manipulation of models and images of objects. These models come from a diverse and expanding set of fields, and include physical, mathematical, engineering, architectural, and even conceptual (abstract) structures, natural phenomena, and so on. Computer graphics today is largely *interactive:* The user controls the contents, structure, and appearance of objects and of their displayed images by using input devices, such as a keyboard, mouse, or touch-sensitive panel on the screen. Because of the close relationship between the input devices and the display, the handling of such devices is included in the study of computer graphics.

Until the early 1980s, computer graphics was a small, specialized field, largely because the hardware was expensive and graphics-based application programs that were easy to use and cost-effective were few. Then, personal computers with built-in raster graphics displays—such as the Xerox Star and, later, the mass-produced, even less expensive Apple Macintosh and the IBM PC and its clones—popularized the use of *bitmap graphics* for user-computer interaction. A *bitmap* is a ones and zeros representation of the rectangular array of points (*pixels* or *pels*, short for "picture elements") on the screen. Once bitmap graphics became affordable, an explosion of easy-to-use and inexpensive graphics-based applications soon followed. Graphics-based user interfaces allowed millions of new users to control simple, low-cost application programs, such as spreadsheets, word processors, and drawing programs.

The concept of a "desktop" now became a popular metaphor for organizing screen space. By means of a *window manager*, the user could create, position, and resize

rectangular screen areas, called *windows*, that acted as virtual graphics terminals, each running an application. This allowed users to switch among multiple activities just by pointing at the desired window, typically with the mouse. Like pieces of paper on a messy desk, windows could overlap arbitrarily. Also part of this desktop metaphor were displays of icons that represented not just data files and application programs, but also common office objects, such as file cabinets, mailboxes, printers, and trashcans, that performed the computer-operation equivalents of their real-life counterparts. *Direct manipulation* of objects via "pointing and clicking" replaced much of the typing of the arcane commands used in earlier operating systems and computer applications. Thus, users could select icons to activate the corresponding programs or objects, or select buttons on pull-down or pop-up screen menus to make choices. Today, almost all interactive application programs, even those for manipulating text (e.g., word processors) or numerical data (e.g., spreadsheet programs), use graphics extensively in the user interface and for visualizing and manipulating the application-specific objects. Graphical interaction via raster displays (displays using bitmaps) has replaced most textual interaction with alphanumeric terminals.

Even people who do not use computers in their daily work encounter computer graphics in television commercials and as cinematic special effects. Computer graphics is no longer a rarity. It is an integral part of all computer user interfaces, and is indispensable for visualizing two-dimensional (2D), three-dimensional (3D), and higher-dimensional objects: Areas as diverse as education, science, engineering, medicine, commerce, the military, advertising, and entertainment all rely on computer graphics. Learning how to program and use computers now includes learning how to use simple 2D graphics as a matter of routine.

1.1 IMAGE PROCESSING AS PICTURE ANALYSIS

Computer graphics concerns the pictorial *synthesis* of real or imaginary objects from their computer-based models, whereas the related field of *image processing* (also called *picture processing*) treats the converse process: the *analysis* of scenes, or the *reconstruction* of models of 2D or 3D objects from their pictures. Picture analysis is important in many arenas: aerial surveillance photographs, slow-scan television images of the moon or of planets gathered from space probes, television images taken from an industrial robot's "eye," chromosome scans, X-ray images, computerized axial tomography (CAT) scans, and fingerprint analysis all exploit image-processing technology (see Color Plate I.1). Image processing has the subareas *image enhancement, pattern detection and recognition*, and *scene analysis and computer vision*. Image enhancement deals with improving image quality by eliminating noise (extraneous or missing pixel data) or by enhancing contrast. Pattern detection and recognition deal with detecting and clarifying standard patterns and finding deviations (distortions) from these patterns. A particularly important example is optical character recognition (OCR) technology, which allows for the economical bulk input of pages of typeset, typewritten, or even handprinted characters. Scene analysis and computer vision allow scientists to recognize and reconstruct a 3D model of a scene from several 2D images. An example is an industrial robot sensing the relative sizes, shapes, positions, and colors of parts on a conveyor belt.

enormous and is growing rapidly as computers with graphics capabilities become commodity products. Let's look at a representative sample of these areas.

- *User interfaces*. As we mentioned, most applications that run on personal computers and workstations, and even those that run on terminals attached to time-shared computers and network compute servers, have user interfaces that rely on desktop window systems to manage multiple simultaneous activities, and on point-and-click facilities to allow users to select menu items, icons, and objects on the screen; typing is necessary only to input text to be stored and manipulated. Word-processing, spreadsheet, and desktop-publishing programs are typical applications that take advantage of such user-interface techniques. The authors of this book used such programs to create both the text and the figures; then, the publisher and their contractors produced the book using similar typesetting and drawing software.
- *(Interactive) plotting in business, science, and technology*. The next most common use of graphics today is probably to create 2D and 3D graphs of mathematical, physical, and economic functions; histograms, bar and pie charts; task-scheduling charts; inventory and production charts; and the like. All these are used to present meaningfully and concisely the trends and patterns gleaned from data, so as to clarify complex phenomena and to facilitate informed decision making.
- *Office automation and electronic publishing*. The use of graphics for the creation and dissemination of information has increased enormously since the advent of desktop publishing on personal computers. Many organizations whose publications used to be printed by outside specialists can now produce printed materials inhouse. Office automation and electronic publishing can produce both traditional printed (hardcopy) documents and electronic (softcopy) documents that contain text, tables, graphs, and other forms of drawn or scanned-in graphics. Hypermedia systems that allow browsing of networks of interlinked multimedia documents are proliferating (see Color Plate I.2).
- *Computer-aided drafting and design*. In computer-aided design (CAD), interactive graphics is used to design components and systems of mechanical, electrical, electromechanical, and electronic devices, including structures such as buildings, automobile bodies, airplane and ship hulls, very large-scale-integrated (VLSI) chips, optical systems, and telephone and computer networks. Sometimes, the user merely wants to produce the precise drawings of components and assemblies, as for online drafting or architectural blueprints. Color Plate I.8 shows an example of such a 3D design program, intended for nonprofessionals: a "customize your own patio deck" program used in lumber yards. More frequently, however, the emphasis is on interacting with a computer-based model of the component or system being designed in order to test, for example, its structural, electrical, or thermal properties. Often, the model is interpreted by a simulator that feeds back the behavior of the system to the user for further interactive design and test cycles. After objects have been designed, utility programs can *postprocess* the design database to make parts lists, to process "bills of materials," to define numerical control tapes for cutting or drilling parts, and so on.
- *Simulation and animation for scientific visualization and entertainment*. Computer-produced animated movies and displays of the time-varying behavior of real and simulated

gray-scale or color systems. These techniques specify gradations in intensity of neighboring pixels at edges of primitives, rather than setting pixels to maximum or zero intensity only; see Chapters 3, 14, and 19 for further discussion of this important topic.

1.5.2 Input Technology

Input technology has also improved greatly over the years. The clumsy, fragile light pen of vector systems has been replaced by the ubiquitous mouse (first developed by office-automation pioneer Doug Engelbart in the mid-sixties [ENGE68]), the data tablet, and the transparent, touch-sensitive panel mounted on the screen. Even fancier input devices that supply not just (x, y) locations on the screen, but also 3D and even higher-dimensional input values (degrees of freedom), are becoming common, as discussed in Chapter 8. Audio communication also has exciting potential, since it allows hands-free input and natural output of simple instructions, feedback, and so on. With the standard input devices, the user can specify operations or picture components by typing or drawing new information or by pointing to existing information on the screen. These interactions require no knowledge of programming and only a little keyboard use: The user makes choices simply by selecting menu buttons or icons, answers questions by checking options or typing a few characters in a form, places copies of predefined symbols on the screen, draws by indicating consecutive endpoints to be connected by straight lines or interpolated by smooth curves, paints by moving the cursor over the screen, and fills closed areas bounded by polygons or paint contours with shades of gray, colors, or various patterns.

1.5.3 Software Portability and Graphics Standards

Steady advances in hardware technology have thus made possible the evolution of graphics displays from one-of-a-kind special output devices to the standard human interface to the computer. We may well wonder whether software has kept pace. For example, to what extent have early difficulties with overly complex, cumbersome, and expensive graphics systems and application software been resolved? Many of these difficulties arose from the primitive graphics software that was available, and in general there has been a long, slow process of maturation in such software. We have moved from low-level, *device-dependent* packages supplied by manufacturers for their particular display devices to higher-level, *device-independent* packages. These packages can drive a wide variety of display devices, from laser printers and plotters to film recorders and high-performance real-time displays. The main purpose of using a device-independent package in conjunction with a high-level programming language is to promote *application-program portability*. This portability is provided in much the same way as a high-level, machine-independent language (such as FORTRAN, Pascal, or C) provides portability: by isolating the programmer from most machine peculiarities and providing language features readily implemented on a broad range of processors. "Programmer portability" is also enhanced in that programmers can now move from system to system, or even from installation to installation, and find familiar software.

A general awareness of the need for standards in such device-independent graphics packages arose in the mid-seventies and culminated in a specification for a *3D Core*

graphics processor with a separate pixmap is introduced, and a wide range of graphics-processor functionalities is discussed in Section 4.3.3. Section 4.3.4 discusses ways in which the pixmap can be integrated back into the CPU's address space, given the existence of a graphics processor.

4.3.1 Simple Raster Display System

The simplest and most common raster display system organization is shown in Fig. 4.18. The relation between memory and the CPU is exactly the same as in a nongraphics computer system. However, a portion of the memory also serves as the pixmap. The video controller displays the image defined in the frame buffer, accessing the memory through a separate access port as often as the raster-scan rate dictates. In many systems, a fixed portion of memory is permanently allocated to the frame buffer, whereas some systems have several interchangeable memory areas (sometimes called *pages* in the personal-computer world). Yet other systems can designate (via a register) any part of memory for the frame buffer. In this case, the system may be organized as shown in Fig. 4.19, or the entire system memory may be dual-ported.

The application program and graphics subroutine package share the system memory and are executed by the CPU. The graphics package includes scan-conversion procedures, so that when the application program calls, say, SRGP_lineCoord (*x1*, *y1*, *x2*, *y2*), the graphics package can set the appropriate pixels in the frame buffer (details on scan-conversion procedures were given in Chapter 3). Because the frame buffer is in the address space of the CPU, the graphics package can easily access it to set pixels and to implement the PixBlt instructions described in Chapter 2.

The video controller cycles through the frame buffer, one scan line at a time, typically 60 times per second. Memory reference addresses are generated in synchrony with the raster scan, and the contents of the memory are used to control the CRT beam's intensity or

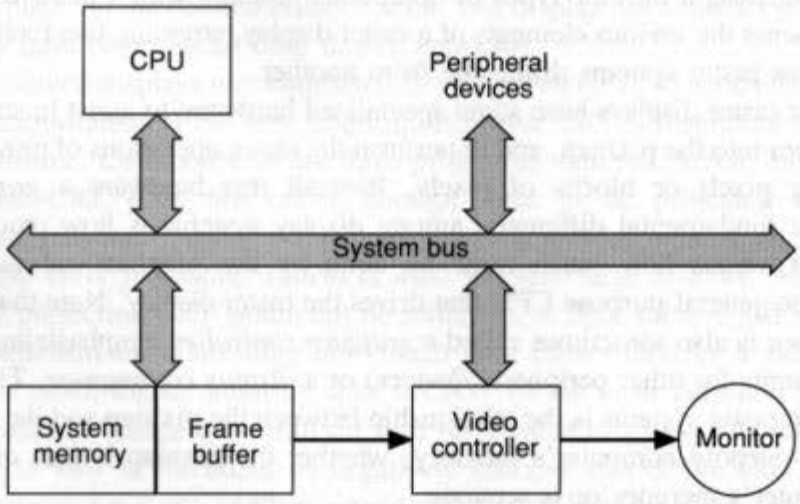


Fig. 4.18 A common raster display system architecture. A dedicated portion of the system memory is dual-ported, so that it can be accessed directly by the video controller, without the system bus being tied up.

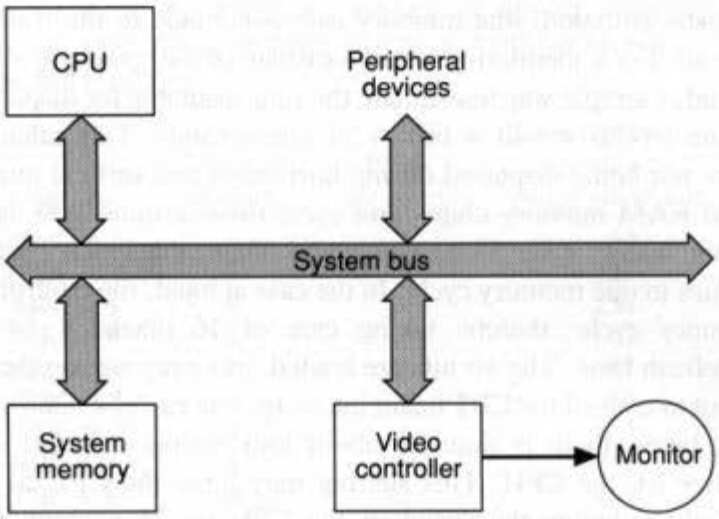


Fig. 4.19 A simple raster display system architecture. Because the frame buffer may be stored anywhere in system memory, the video controller accesses the memory via the system bus.

color. The video controller is organized as shown in Fig. 4.20. The raster-scan generator produces deflection signals that generate the raster scan; it also controls the X and Y address registers, which in turn define the memory location to be accessed next.

Assume that the frame buffer is addressed in x from 0 to x_{max} and in y from 0 to y_{max}; then, at the start of a refresh cycle, the X address register is set to zero and the Y register is set to y_{max} (the top scan line). As the first scan line is generated, the X address is incremented up through x_{max}. Each pixel value is fetched and is used to control the intensity of the CRT beam. After the first scan line, the X address is reset to zero and the Y address is decremented by one. The process continues until the last scan line ($y = 0$) is generated.

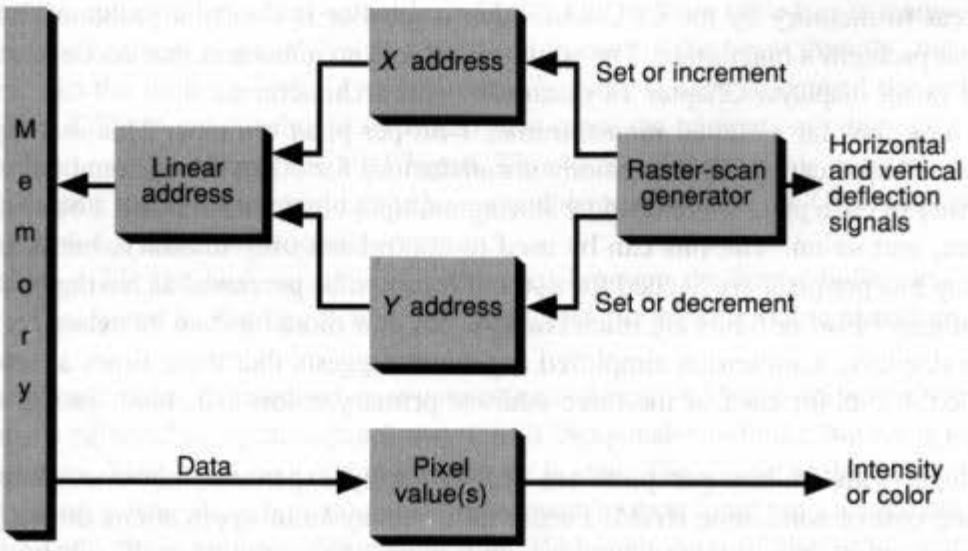


Fig. 4.20 Logical organization of the video controller.

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.

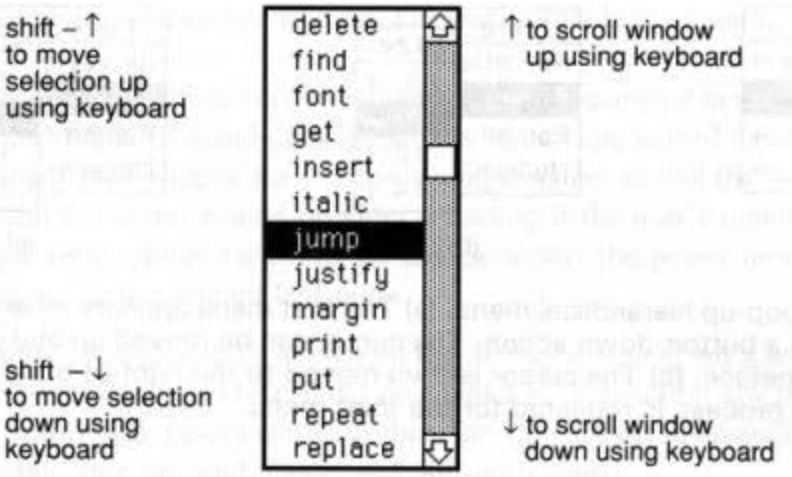


Fig. 8.8 A menu within a scrolling window. The user controls scrolling by selecting the up and down arrows or by dragging the square in the scroll bar.

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

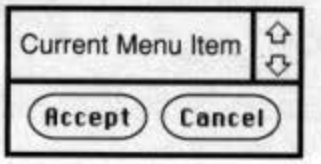


Fig. 8.9 A small menu-selection window. Only one menu item appears at a time. The scroll arrows are used to change the current menu item, which is selected when the Accept button is chosen.

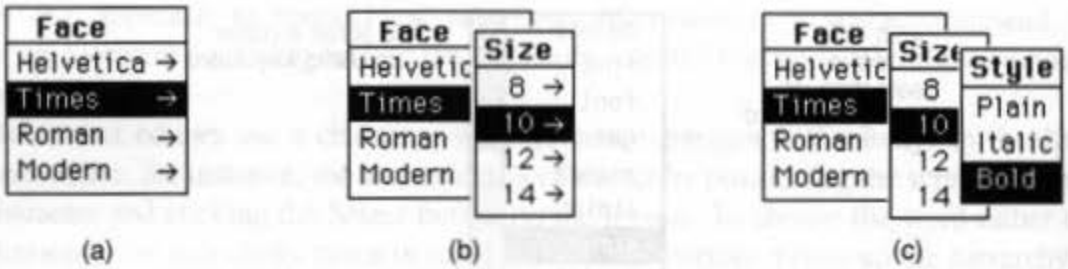


Fig. 8.10 A pop-up hierarchical menu. (a) The first menu appears where the cursor is, in response to a button-down action. The cursor can be moved up and down to select the desired typeface. (b) The cursor is then moved to the right to bring up the second menu. (c) The process is repeated for the third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

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

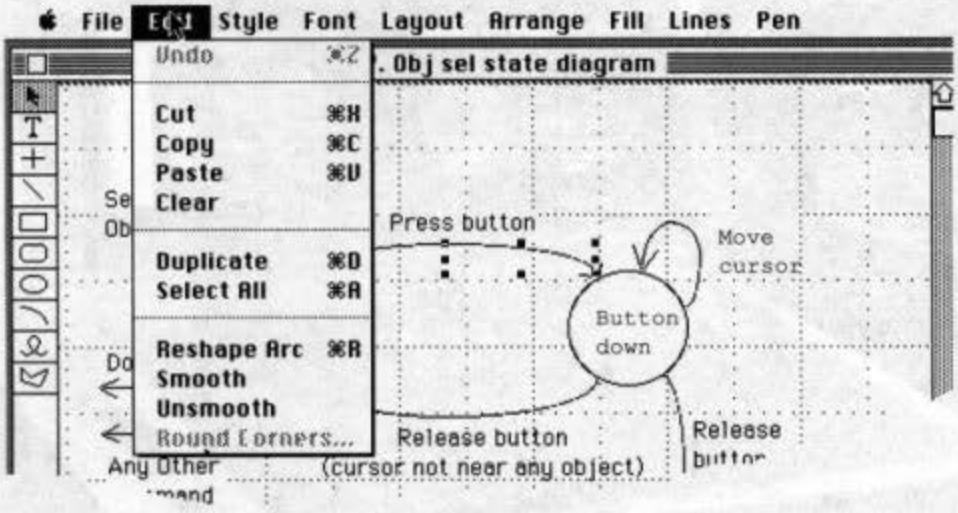


Fig. 8.13 A Macintosh pull-down menu. The last menu item is gray rather than black, indicating that it is currently not available for selection (the currently selected object, an arc, does not have corners to be rounded). The Undo command is also gray, because the previously executed command cannot be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

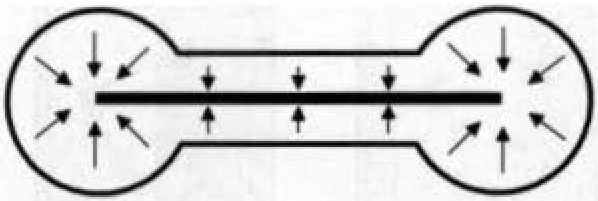


Fig. 8.39 Line surrounded by a gravity field, to aid picking points on the line: If the cursor falls within the field, it is snapped to the line.

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.

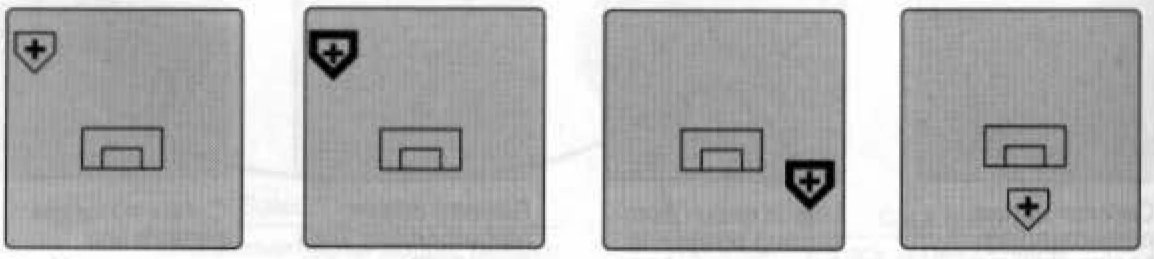
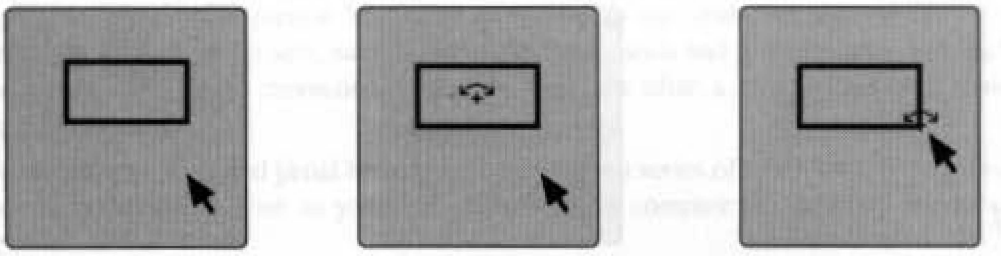


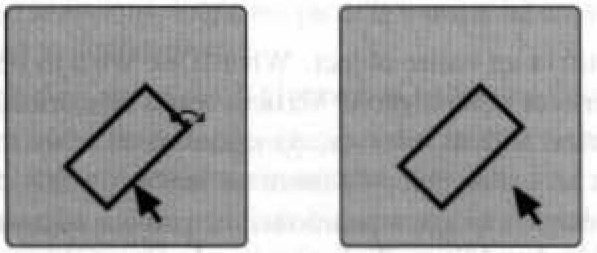
Fig. 8.40 Dragging a symbol into a new position.



Highlighted object has been selected with cursor

Rotate command has been invoked, causing center of rotation icon to appear at default center position unless previously set

Center-of-rotation icon is dragged into a new position

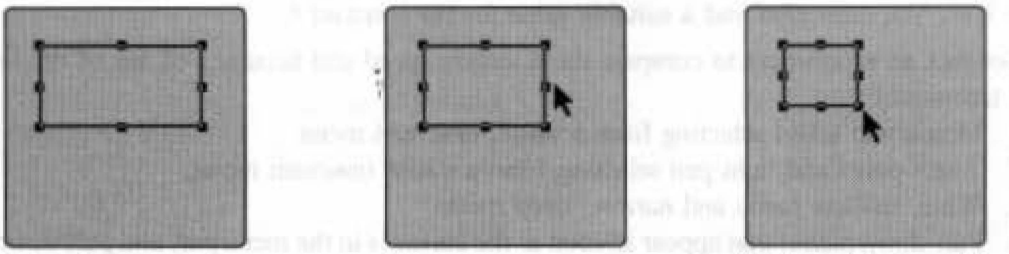


Rectangle is now rotated by pointing at rectangle, depressing button, and moving left-right with button down

Button is released; cursor no longer controls rotation; icon is gone; rectangle remains selected for other possible operations

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.)



Selecting rectangle with cursor causes handles to appear

Button actions on this handle move only right side of rectangle

Button actions on this handle move only corner of rectangle

Fig. 8.42 Handles used to reshape objects.

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.

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

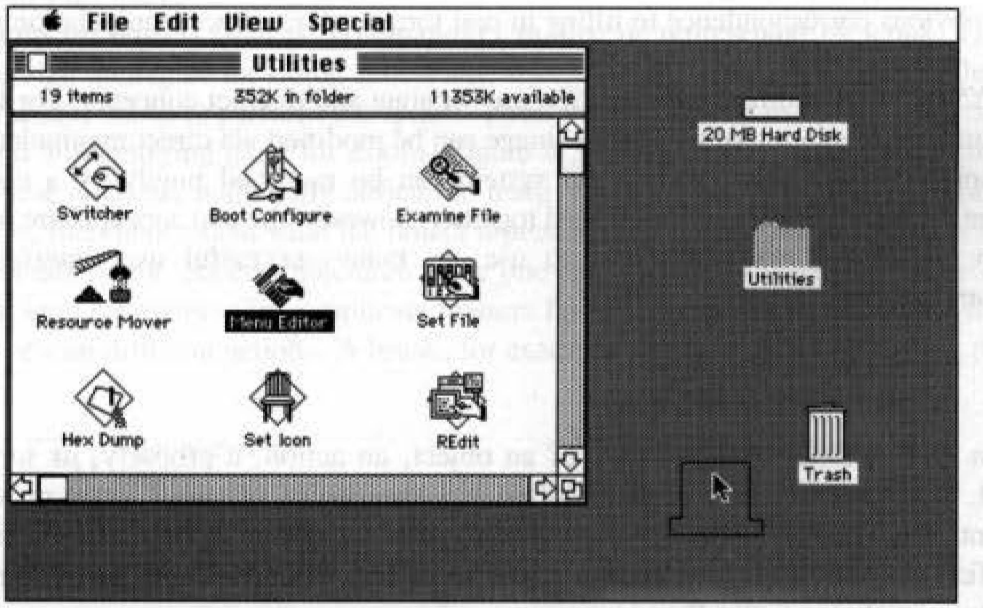


Fig. 9.1 The Macintosh screen. In the upper right is a disk icon; just below it is a directory icon, which is gray-toned to indicate that it is open. At the left is the open directory, with named icons representing the files within it. A file, represented by the icon outline around the cursor, is being 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.

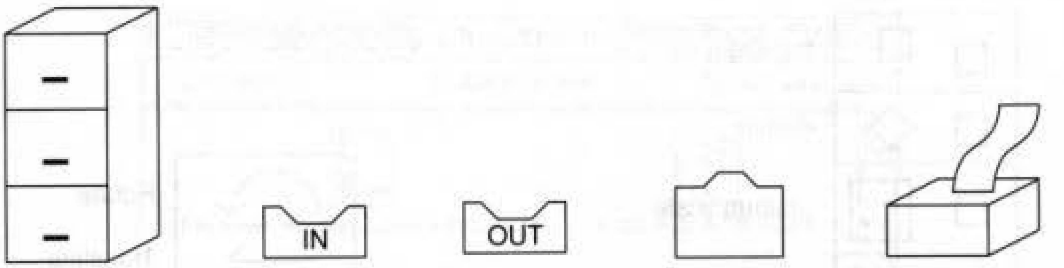


Fig. 9.2 Icons used to represent common office objects.

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

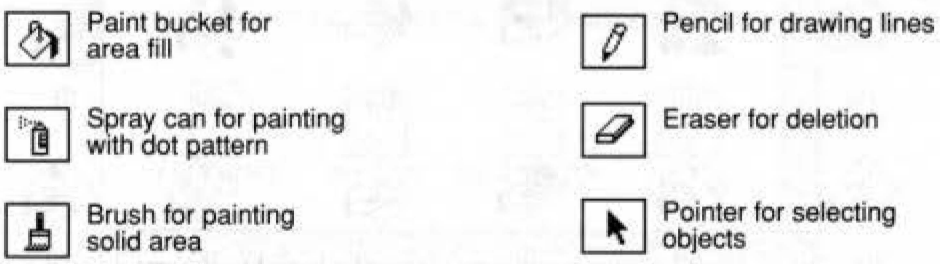


Fig. 9.3 Command icons representing objects used to perform the corresponding command. (Copyright 1988 Claris Corporation. All rights reserved.)

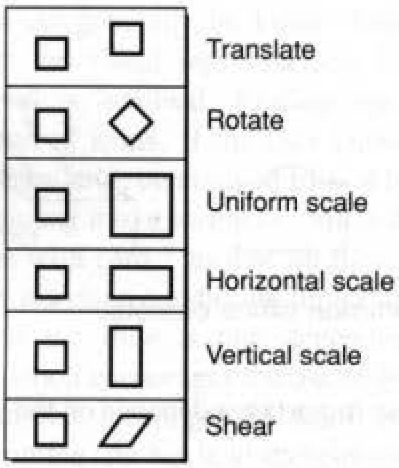


Fig. 9.4 Command icons indicating geometric transformations by showing a square before and after the commands are applied.

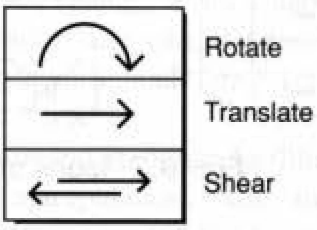


Fig. 9.5 Several abstract command icons for some of the actions depicted in Fig. 9.4. Not all geometric operations can be represented in this way.

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.

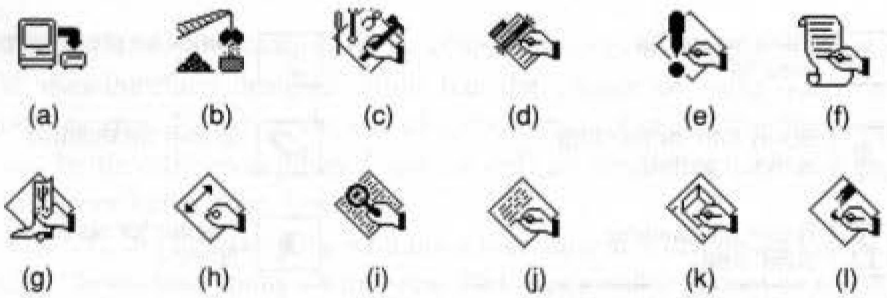


Fig. 9.6 Icons that represent Macintosh programs. What does each icon represent? In most cases, the icons suggest the type of information that is operated on or created. See Exercise 9.14 for the answers.

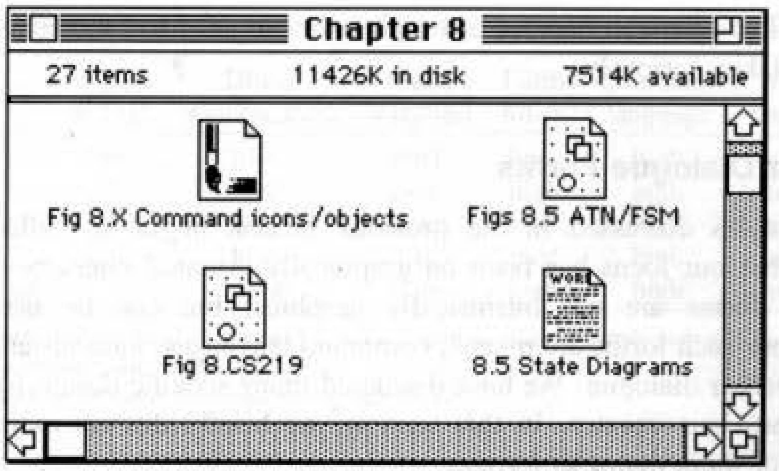


Fig. 9.7 The contents of a disk directory represented with icons and text. The icons help to distinguish one file from another. (Certain screen graphics © Apple Computer, Inc.)

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

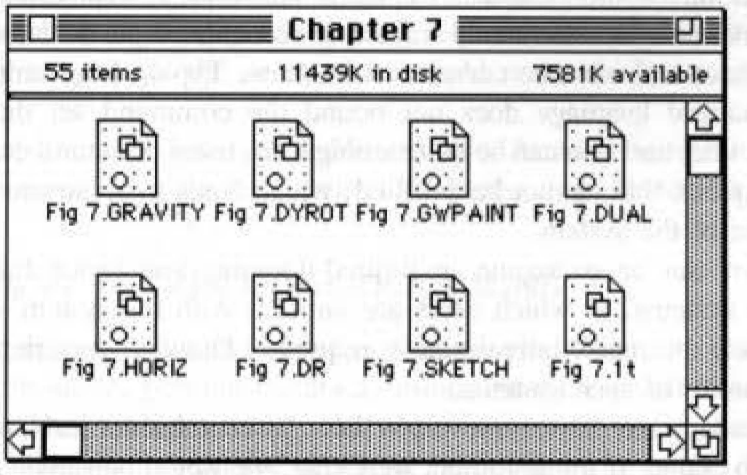


Fig. 9.8 The contents of a disk directory represented with icons and text. Since the files are all of the same type, the icons do not help to distinguish one file from another, and simply take up extra space. (Computer screen graphics © Apple Computer, Inc.)

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,

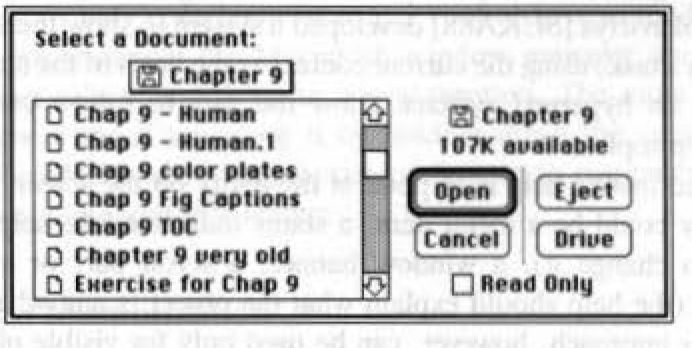


Fig. 9.14 Opening files from within a Macintosh program. The user enters the Open command, either by menu selection or with a two-key chord, causing the dialogue box to appear. The highlighted file can be opened with the "open" button or with the carriage-return key. The user can highlight a new file 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.)

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.

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.

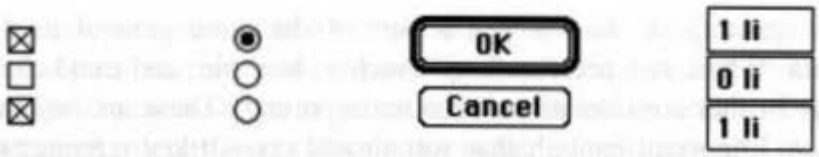


Fig. 9.23 The graphic alphabet used in many Macintosh applications. The square choice boxes indicate alternatives, of which several may be selected at once. The round choice circles, called "radio buttons," indicate mutually exclusive alternatives; only one may be selected. The rounded-corner rectangles 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,

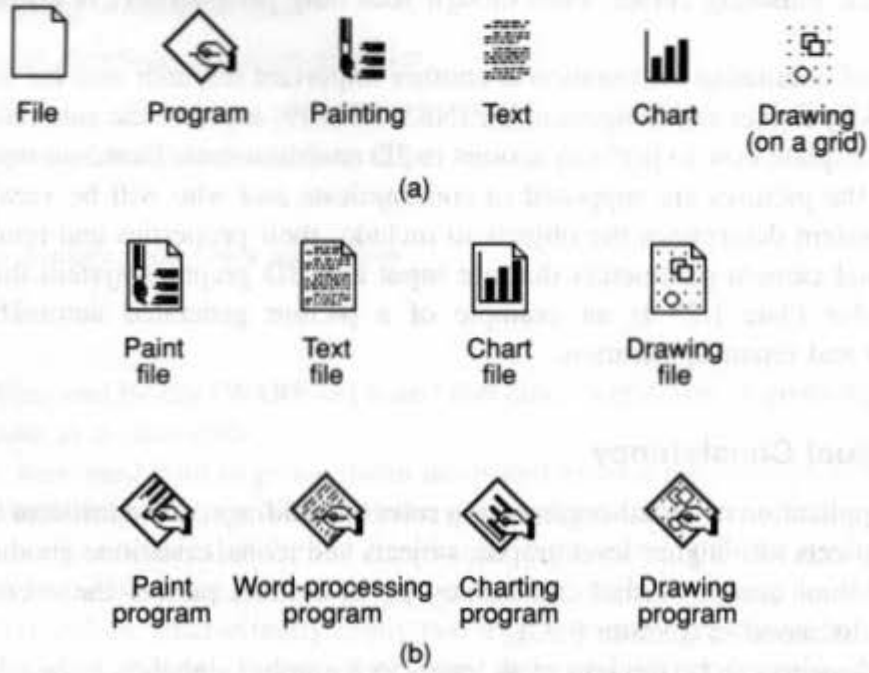


Fig. 9.24 (a) A graphics alphabet. (b) Icons formed by combining elements of the alphabet.

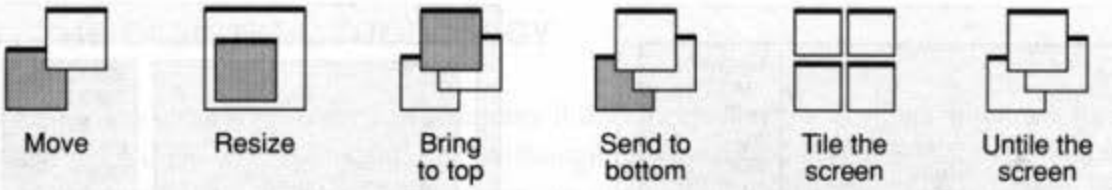


Fig. 9.25 Several different icons, all created from a single shape representing a window.

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].

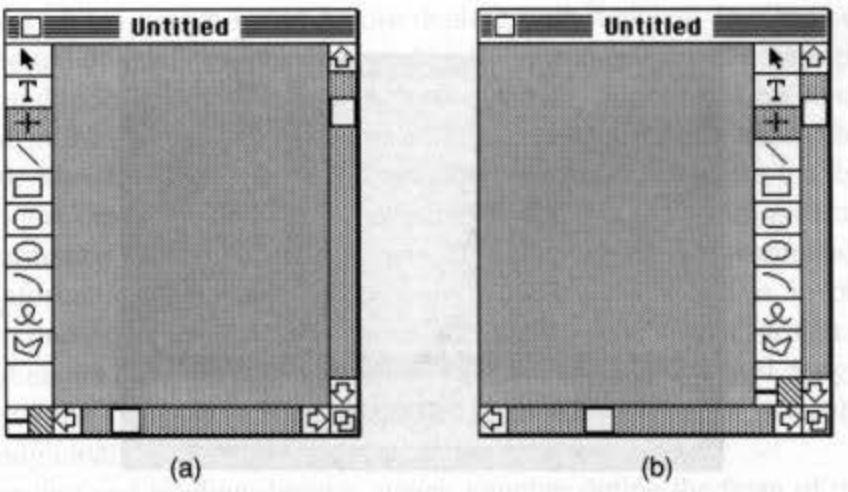


Fig. 9.26 Two alternative screen designs. Design (a) is balanced; design (b) emphasizes the right side. (Copyright 1988 Claris Corporation. All rights reserved.)