

# **MAR GREGORIOS COLLEGE OF ARTS & SCIENCE**

Block No.8, College Road, Mogappair West, Chennai – 37

Affiliated to the University of Madras  
Approved by the Government of Tamil Nadu  
An ISO 9001:2015 Certified Institution



## **PG DEPARTMENT OF COMPUTER SCIENCE**

**SUBJECT NAME: COMPUTER GRAPHICS**

**SUBJECT CODE: PSDEC**

**SEMESTER: II**

**PREPARED BY: PROF.A.JEROME ROBINSON**

## E-CONTENT- COMPUTER GRAPHICS BY A.JEROME ROBINSON

### UNIT 1:

### INTRODUCTION TO COMPUTER GRAPHICS

The power and utility of computer graphics is widely recognized, and a broad range of graphics hardware and software systems is now available for applications in virtually all fields. Graphics capabilities for both two-dimensional and three dimensional applications are now common, even on general-purpose computers and handheld calculators. With personal computers, we can use a variety of interactive input devices and graphics software packages. For higher-quality applications, we can choose from a number of sophisticated special-purpose graphics hardware systems and technologies. In this chapter, we explore the basic features of graphics hardware components and graphics software packages.

### VIDEO DISPLAY DEVICES

Typically, the primary output device in a graphics system is a video monitor. Historically, the operation of most video monitors was based on the standard **cathode-ray tube (CRT)** design, but several other technologies exist. In recent years, **flat-panel** displays have become significantly more popular due to their reduced power consumption and thinner designs.

#### Refresh Cathode-Ray Tubes

Figure 1 illustrates the basic operation of a CRT. A beam of electrons (*cathode rays*), emitted by an electron gun, passes through focusing and deflection systems that direct the beam toward specified positions on the phosphor-coated screen. The phosphor then emits a small spot of light at each position contacted by the electron beam. Because the light emitted by the phosphor fades very rapidly, some method is needed for maintaining the screen picture. One way to do this is to store the picture information as a charge distribution within the CRT. This charge distribution can then be used to keep the phosphors activated. However, the most common method now employed for maintaining phosphor glow is to redraw the picture repeatedly by quickly directing the electron beam back over the same screen points. This type of display is called a **refresh CRT**, and the frequency at which a picture is redrawn on the screen is referred to as the **refresh rate**. The primary components of an electron gun in a CRT are the heated metal cathode and a control grid (Fig. 2). Heat is supplied to the cathode by directing a current through a coil of wire, called the filament, inside the cylindrical cathode structure. This causes electrons to be “boiled off” the hot cathode surface.

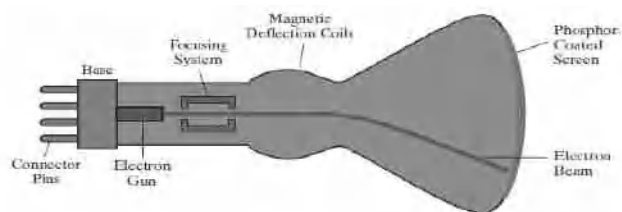


FIGURE 1  
Basic design of a magnetic-deflection CRT.

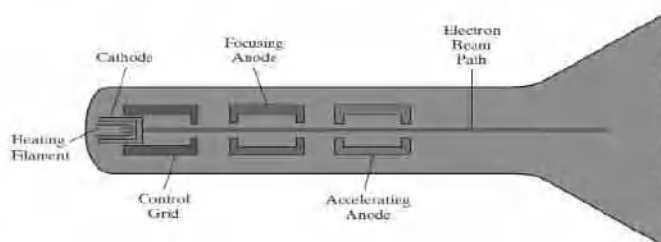


FIGURE 2  
Operation of an electron gun with an accelerating anode.

In the vacuum inside the CRT envelope, the free, negatively charged electrons are then accelerated toward the phosphor coating by a high positive voltage. The accelerating voltage can be generated with a positively charged metal coating on the inside of the CRT envelope near the phosphor screen, or an accelerating anode, as in Figure 2, can be used to provide the positive voltage. Sometimes the electron gun is designed so that the accelerating anode and focusing system are within the same unit.

Intensity of the electron beam is controlled by the voltage at the control grid, which is a metal cylinder that fits over the cathode. A high negative voltage applied to the control grid will shut off the beam by repelling electrons and stopping them from passing through the small hole at the end of the control grid structure. A smaller negative voltage on the control grid simply decreases the number of electrons passing through. Since the amount of light emitted by the phosphor coating depends on the number of electrons striking the screen, the brightness of a display point is controlled by varying the voltage on the control grid. This brightness, or intensity level, is specified for individual screen positions with graphics software commands. The focusing system in a CRT forces the electron beam to converge to a small cross section as it strikes the phosphor. Otherwise, the electrons would repel each other, and the beam would spread out as it approaches the screen. Focusing is accomplished with either electric or magnetic fields. With electrostatic focusing, the electron beam is passed through a positively charged metal cylinder so that electrons along the center line of the cylinder are in an equilibrium position. This arrangement forms an electrostatic lens, as shown in Figure 2, and the electron beam is focused at the center of the screen in the same way that an optical lens focuses a beam of light at a particular focal distance. Similar lens focusing effects can be accomplished with a magnetic field set up by a coil mounted around the outside of the CRT envelope, and magnetic lens focusing usually produces the smallest spot size on the screen.

Additional focusing hardware is used in high-precision systems to keep the beam in focus at all screen positions. The distance that the electron beam must travel to different points on the screen varies because the radius of curvature for most CRTs is greater than the distance from the focusing system to the screen center. Therefore, the electron beam will be focused properly only at the center of the screen. As the beam moves to the outer edges of the screen, displayed images become blurred. To compensate for this, the system can adjust the focusing according to the screen position of the beam.

As with focusing, deflection of the electron beam can be controlled with either electric or magnetic fields. Cathode-ray tubes are now commonly constructed with magnetic-deflection coils mounted on the outside of the CRT envelope, as illustrated in Figure 1. Two pairs of coils are used for this purpose. One pair is mounted on the top and bottom of the CRT neck, and the other pair is mounted on opposite sides of the neck. The magnetic field produced by each pair of coils results in a transverse deflection force that is perpendicular to both the direction of the magnetic field and the direction of

travel of the electron beam.

Horizontal deflection is accomplished with one pair of coils, and vertical

deflection with the other pair. The proper deflection amounts are attained by adjusting the current through the coils. When electrostatic deflection is used, two pairs of parallel plates are mounted inside the CRT envelope. One pair of plates is mounted horizontally to control vertical deflection, and the other pair is mounted vertically to control horizontal deflection (Fig. 3).

Spots of light are produced on the screen by the transfer of the CRT beam energy to the phosphor. When the electrons in the beam collide with the phosphor coating, they are stopped and their kinetic energy is absorbed by the phosphor.

Part of the beam energy is converted by friction into heat energy, and the remainder

causes electrons in the phosphor atoms to move up to higher quantum-energy levels. After a short time, the “excited” phosphor electrons begin dropping back to their stable ground state, giving up their extra energy as small quanta of light energy called photons. What we see on the screen is the combined effect of all the electron light emissions: a glowing spot that quickly fades after all the excited phosphor electrons have returned to their ground energy level. The frequency (or color) of the light emitted by the phosphor is in proportion to the energy difference between the excited quantum state and the ground state.

Different kinds of phosphors are available for use in CRTs. Besides color, a major difference between phosphors is their **persistence**: how long they continue to emit light (that is, how long it is before all excited electrons have returned to the ground state) after the CRT beam is removed. Persistence is defined as the time that it takes the emitted light from the screen to decay to one-tenth of its original intensity. Lower-persistence phosphors require higher refresh rates to maintain a picture on the screen without flicker. A phosphor with low persistence can be useful for animation, while high-persistence phosphors are better suited for displaying highly complex, static pictures. Although some phosphors have persistence values greater than 1 second, general-purpose graphics monitors are usually constructed with persistence in the range from 10 to 60 microseconds.

Figure 4 shows the intensity distribution of a spot on the screen. The intensity is greatest at the center of the spot, and it decreases with a Gaussian distribution out to the edges of the spot. This distribution corresponds to the cross-sectional electron density distribution of the CRT beam. The maximum number of points that can be displayed without overlap on a CRT is referred to as the **resolution**. A more precise definition of resolution is the number of points per centimeter that can be plotted

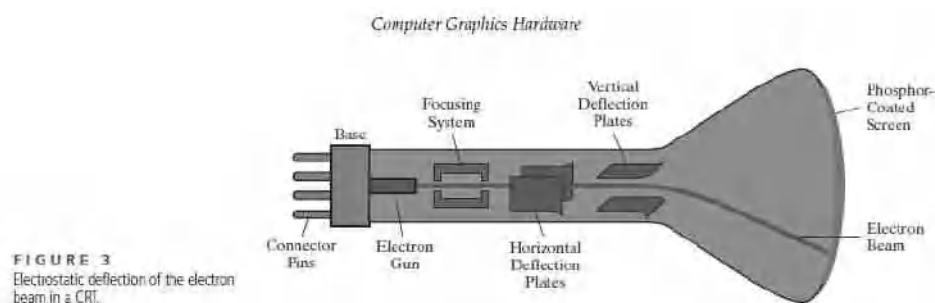


FIGURE 3  
Electrostatic deflection of the electron beam in a CRT.



horizontally and vertically, although it is often simply stated as the total number of points in each direction. Spot intensity has a Gaussian distribution (Fig. 4), so two adjacent spots will appear distinct as long as their separation is greater than the diameter at which each spot has an intensity of about 60 percent of that at the center of the spot. This overlap position is illustrated in Figure 5. Spot size also depends on intensity. As more electrons are accelerated toward the phosphor per second, the diameters of the CRT beam and the illuminated spot increase. In addition, the increased excitation energy tends to spread to neighboring phosphor atoms not directly in the path of the beam, which further increases the spot diameter.

Thus, resolution of a CRT is dependent on the type of phosphor, the intensity to be displayed, and the focusing and deflection systems. Typical resolution on high-quality systems is 1280 by 1024, with higher resolutions available on many systems. High-resolution systems are often referred to as *high-definition systems*.

The physical size of a graphics monitor, on the other hand, is given as the length of the screen diagonal, with sizes varying from about 12 inches to 27 inches or more.

A CRT monitor can be attached to a variety of computer systems, so the number of screen points that can actually be plotted also depends on the capabilities of the system to which it is attached.

### Raster-Scan Displays

The most common type of graphics monitor employing a CRT is the **raster-scan display**, based on television technology. In a raster-scan system, the electron beam is swept across the screen, one row at a time, from top to bottom. Each row is referred to as a **scan line**. As the electron beam moves across a scan line, the beam intensity is turned on and off (or set to some intermediate value) to create a pattern of illuminated spots. Picture definition is stored in a memory area called the **refresh buffer** or **frame buffer**, where the term **frame** refers to the total screen area.

This memory area holds the set of color values for the screen points. These stored color values are then retrieved from the refresh buffer and used to control the intensity of the electron beam as it moves from spot to spot across the screen. In this way, the picture is “painted” on the screen one scan line at a time, as demonstrated in Figure 6. Each screen spot that can be illuminated by the electron beam is referred to as a **pixel** or **pel** (shortened forms of **picture element**). Since the refresh buffer is used to store the set of screen color values, it is also sometimes called a **color buffer**. Also, other kinds of pixel information, besides color, are stored in buffer locations, so all the different buffer areas are sometimes referred to collectively as the “frame buffer.” The capability of a raster-scan system to store color information for each screen point makes it well suited for the realistic display of scenes containing subtle shading and color patterns. Home television sets and printers are examples of other systems using raster-scan methods.



FIGURE 4  
Intensity distribution of an illuminated phosphor spot on a CRT screen.

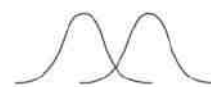
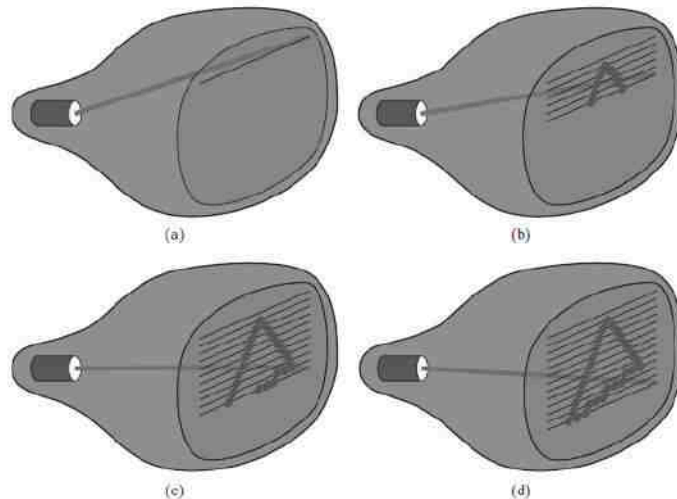


FIGURE 5  
Two illuminated phosphor spots are distinguishable when their separation is greater than the diameter at which a spot intensity has fallen to 60 percent of maximum.

Raster systems are commonly characterized by their resolution, which is the number of pixel positions that can be plotted. Another property of video monitors is



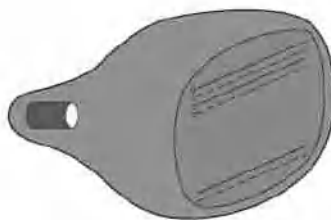
**FIGURE 6**  
A raster-scan system displays an object as a set of discrete points across each scan line.

**aspect ratio**, which is now often defined as the number of pixel columns divided by the number of scan lines that can be displayed by the system. (Sometimes this term is used to refer to the number of scan lines divided by the number of pixel columns.) Aspect ratio can also be described as the number of horizontal points to vertical points (or vice versa) necessary to produce equal-length lines in both directions on the screen. Thus, an aspect ratio of  $4/3$ , for example, means that a horizontal line plotted with four points has the same length as a vertical line plotted with three points, where line length is measured in some physical units such as centimeters. Similarly, the aspect ratio of any rectangle (including the total screen area) can be defined to be the width of the rectangle divided by its height. The range of colors or shades of gray that can be displayed on a raster system depends on both the types of phosphor used in the CRT and the number of bits per pixel available in the frame buffer. For a simple black-and-white system, each screen point is either on or off, so only one bit per pixel is needed to control the intensity of screen positions. A bit value of 1, for example, indicates that the electron beam is to be turned on at that position, and a value of 0 turns the beam off. Additional bits allow the intensity of the electron beam to be varied over a range of values between “on” and “off.” Up to 24 bits per pixel are included in high-quality systems, which can require several megabytes of storage for the frame buffer, depending on the resolution of the system. For example, a system with 24 bits per pixel and a screen resolution of 1024 by 1024 requires 3 MB of storage for the refresh buffer. The number of bits per pixel in a frame buffer is sometimes referred to as either the **depth** of the buffer area or the number of **bit planes**. A frame buffer with one bit per pixel is commonly called a **bitmap**, and a frame buffer with multiple bits per pixel is a **pixmap**, but these terms are also used to describe other rectangular arrays, where a bitmap is any pattern of binary values and a pixmap is a multicolor pattern. As each screen refresh takes place, we tend to see each frame as a smooth continuation of the patterns in the previous frame, so long as the refresh rate is not too low. Below about 24 frames per second, we can usually perceive a gap between successive screen images, and the picture appears to flicker. Old silent films, for example, show this effect because they were

photographed at a rate of 16 frames per second. When sound systems were developed in the 1920s, motionpicture film rates increased to 24 frames per second, which removed flickering and the accompanying jerky movements of the actors. Early raster-scan computer systems were designed with a refresh rate of about 30 frames per second. This produces reasonably good results, but picture quality is improved, up to a point, with higher refresh rates on a video monitor because the display technology on the monitor is basically different from that of film. A film projector can maintain the continuous display of a film frame until the next frame is brought into view. But on a video monitor, a phosphor spot begins to decay as soon as it is illuminated. Therefore, current raster-scan displays perform refreshing at the rate of 60 to 80 frames per second, although some systems now have refresh rates of up to 120 frames per second. And some graphics systems have been designed with a variable refresh rate. For example, a higher refresh rate could be selected for a stereoscopic application so that two views of a scene (one from each eye position) can be alternately displayed without flicker. But other methods, such as multiple frame buffers, are typically used for such applications.

Sometimes, refresh rates are described in units of cycles per second, or hertz (Hz), where a cycle corresponds to one frame. Using these units, we would describe a refresh rate of 60 frames per second as simply 60 Hz. At the end of each scan line, the electron beam returns to the left side of the screen to begin displaying the next scan line. The return to the left of the screen, after refreshing each scan line, is called the **horizontal retrace** of the electron beam. And at the end of each frame (displayed in 1/80 to 1/60 of a second), the electron beam returns to the upper-left corner of the screen (**vertical retrace**) to begin the next frame.

On some raster-scan systems and TV sets, each frame is displayed in two passes using an *interlaced* refresh procedure. In the first pass, the beam sweeps across every other scan line from top to bottom. After the vertical



**FIGURE 7**  
Interlaced scan lines on a raster-scan display. First, all points on the even-numbered (solid) scan lines are displayed; then all points along the odd-numbered (dashed) lines are displayed.

retrace, the beam then sweeps out the remaining scan lines (Fig. 7). Interlacing of the scan lines in this way allows us to see the entire screen displayed in half the time that it would have taken to sweep across all the lines at once from top to bottom.

This technique is primarily used with slower refresh rates. On an older, 30 frameper-second, non-interlaced display, for instance, some flicker is noticeable. But with interlacing, each of the two passes can be accomplished in 1/60 of a second, which brings the refresh rate nearer to 60 frames per second. This is an effective technique for avoiding flicker—provided that adjacent scan lines contain similar display information.

### **Random-Scan Displays**

When operated as a **random-scan display** unit, a CRT has the electron beam directed only to those parts of the screen where a picture is to be displayed.

Pictures are generated as

line drawings, with the electron beam tracing out the component lines one after the other. For this reason, random-scan monitors are also referred to as **vector displays** (or **stroke-writing displays** or **calligraphic displays**). The component lines of a picture can be drawn and refreshed by a random-scan system in any specified order (Fig. 8). A pen plotter operates in a similar way and is an example of a random-scan, hard-copy device.

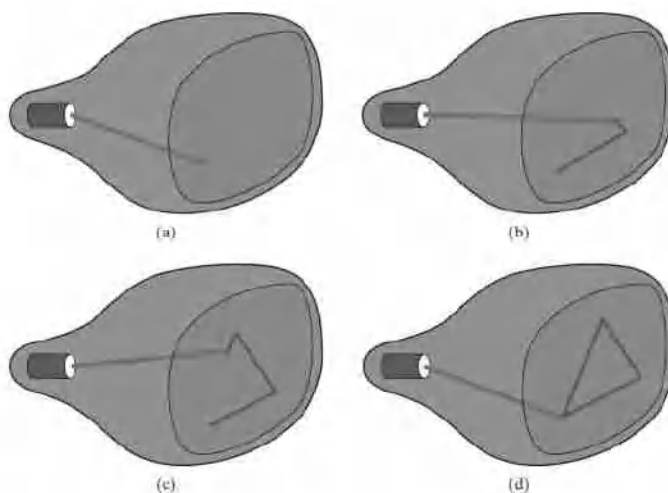
Refresh rate on a random-scan system depends on the number of lines to be displayed on that system. Picture definition is now stored as a set of line-drawing commands in an area of memory referred to as the **display list**, **refresh display file**, **vector file**, or **display program**. To display a specified picture, the system cycles through the set of commands in the display file, drawing each component line in turn. After all line-drawing commands have been processed, the system cycles back to the first line command in the list. Random-scan displays are designed to draw all the component lines of a picture 30 to 60 times each second, with up to 100,000 "short" lines in the display list. When a small set of lines is to be displayed, each refresh cycle is delayed to avoid very high refresh rates, which could burn out the phosphor.

Random-scan systems were designed for line-drawing applications, such as architectural and engineering layouts, and they cannot display realistic shaded scenes. Since picture definition is stored as a set of line-drawing instructions rather than as a set of intensity values for all screen points, vector displays generally have higher resolutions than raster systems. Also, vector displays produce smooth linedrawings because the CRT beam directly follows the line path. A raster system, by contrast, produces jagged lines that are plotted as discrete point sets. However, the greater flexibility and improved line-drawing capabilities of raster systems have resulted in the abandonment of vector technology.

### **Color CRT Monitors**

A CRT monitor displays color pictures by using a combination of phosphors that emit different-colored light. The emitted light from the different phosphors merges to form a single perceived color, which depends on the particular set of phosphors that have been excited.

One way to display color pictures is to coat the screen with layers of different colored

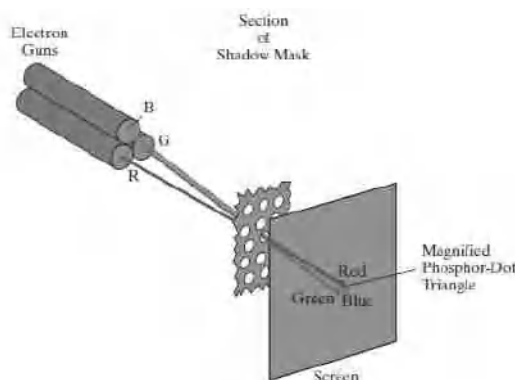


**FIGURE 8**  
A random-scan system draws the component lines of an object in any specified order.



phosphors. The emitted color depends on how far the electron beam penetrates into the phosphor layers. This approach, called the **beam-penetration** method, typically used only two phosphor layers: red and green. A beam of slow electrons excites only the outer red layer, but a beam of very fast electrons penetrates the red layer and excites the inner green layer. At intermediate beam speeds, combinations of red and green light are emitted to show two additional colors: orange and yellow. The speed of the electrons, and hence the screen color at any point, is controlled by the beam acceleration voltage. Beam penetration has been an inexpensive way to produce color, but only a limited number of colors are possible, and picture quality is not as good as with other methods.

**Shadow-mask** methods are commonly used in raster-scan systems (including color TV) because they produce a much wider range of colors than the beam-penetration method. This approach is based on the way that we seem to perceive colors as combinations of red, green, and blue components, called the **RGB color model**. Thus, a shadow-mask CRT uses three phosphor color dots at each pixel position. One phosphor dot emits a red light, another emits a green light, and the third emits a blue light. This type of CRT has three electron guns, one for each color dot, and a shadow-mask grid just behind the phosphor-coated screen. The light emitted from



**FIGURE 9**  
Operation of a delta-delta shadow-mask CRT. Three electron guns, aligned with the triangular color-dot patterns on the screen, are directed to each dot triangle by a shadow mask.

the three phosphors results in a small spot of color at each pixel position, since our eyes tend to merge the light emitted from the three dots into one composite color. Figure 9 illustrates the *delta-delta* shadow-mask method, commonly used in color CRT systems. The three electron beams are deflected and focused as a group onto the shadow mask, which contains a series of holes aligned with the phosphor-dot patterns. When the three beams pass through a hole in the shadow mask, they activate a dot triangle, which appears as a small color spot on the screen. The phosphor dots in the triangles are arranged so that each electron beam can activate only its corresponding color dot when it passes through the shadow mask. Another configuration for the three electron guns is an *in-line* arrangement in which the three electron guns, and the corresponding RGB color dots on the screen, are aligned along one scan line instead of in a triangular pattern. This in-line arrangement of electron guns is easier to keep in alignment and is commonly used in high-resolution color CRTs.

We obtain color variations in a shadow-mask CRT by varying the intensity levels of the three electron beams. By turning off two of the three guns, we get only the color coming from the single activated phosphor (red, green, or blue). When all three dots are activated with equal beam intensities, we see a white color. Yellow is produced with equal intensities from the green

and red dots only, magenta is produced with equal blue and red intensities, and cyan shows up when blue and green are activated equally. In an inexpensive system, each of the three electron beams might be restricted to either on or off, limiting displays to eight colors. More sophisticated systems can allow intermediate intensity levels to be set for the electron beams, so that several million colors are possible.

Color graphics systems can be used with several types of CRT display devices. Some inexpensive home-computer systems and video games have been designed for use with a color TV set and a radio-frequency (RF) modulator. The purpose of the RF modulator is to simulate the signal from a broadcast TV station. This means that the color and intensity information of the picture must be combined and superimposed on the broadcast-frequency carrier signal that the TV requires as input. Then the circuitry in the TV takes this signal from the RF modulator, extracts the picture information, and paints it on the screen. As we might expect, this extra handling of the picture information by the RF modulator and TV circuitry decreases the quality of displayed images.

**Composite monitors** are adaptations of TV sets that allow bypass of the broadcast circuitry. These display devices still require that the picture information be combined, but no carrier signal is needed. Since picture information is combined into a composite signal and then separated by the monitor, the resulting picture quality is still not the best attainable.

Color CRTs in graphics systems are designed as **RGB monitors**. These monitors use shadow-mask methods and take the intensity level for each electron gun (red, green, and blue) directly from the computer system without any intermediate processing. High-quality raster-graphics systems have 24 bits per pixel in the frame buffer, allowing 256 voltage settings for each electron gun and nearly 17 million color choices for each pixel. An RGB color system with 24 bits of storage per pixel is generally referred to as a **full-color system** or a **true-color system**.

### **Flat-Panel Displays**

Although most graphics monitors are still constructed with CRTs, other technologies are emerging that may soon replace CRT monitors. The term **flat-panel display** refers to a class of video devices that have reduced volume, weight, and power requirements compared to a CRT. A significant feature of flat-panel displays is that they are thinner than CRTs, and we can hang them on walls or wear them on our wrists. Since we can even write on some flat-panel displays, they are also available as pocket notepads. Some additional uses for flat-panel displays are as small TV monitors, calculator screens, pocket video-game screens, laptop computer screens, armrest movie-viewing stations on airlines, advertisement boards in elevators, and graphics displays in applications requiring rugged, portable monitors.

We can separate flat-panel displays into two categories: **emissive displays** and **nonemissive displays**. The emissive displays (or **emitters**) are devices that convert electrical energy into light. Plasma panels, thin-film electroluminescent displays, and light-emitting diodes are examples of emissive displays. Flat CRTs have also been devised, in which electron beams

are accelerated parallel to the screen and then deflected 90° onto the screen. But flat CRTs have not proved to be as successful as other emissive devices. Nonemissive displays (or **nonemitters**) use optical effects to convert sunlight or light from some other source into graphics patterns. The most important example of a nonemissive flat-panel display is a liquid-crystal device.

**Plasma panels**, also called **gas-discharge displays**, are constructed by filling the region between two glass plates with a mixture of gases that usually includes neon.

A series of vertical conducting ribbons is placed on one

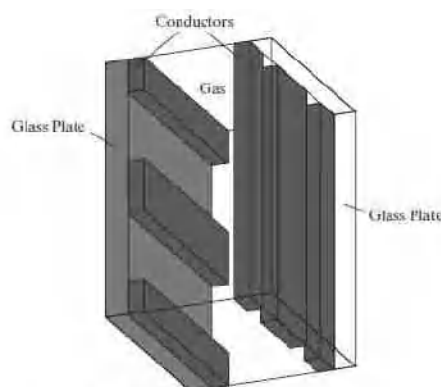


FIGURE 10 Basic design of a plasma-panel display device.

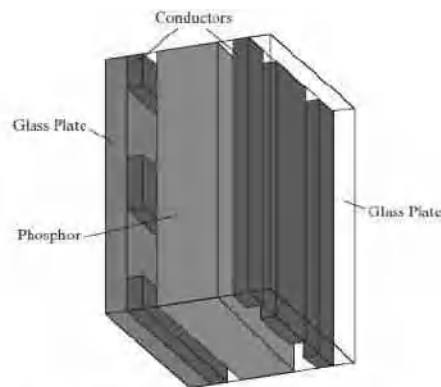


FIGURE 11 Basic design of a thin-film electroluminescent display device.

glass panel, and a set of horizontal conducting ribbons is built into the other glass panel (Fig. 10).

Firing voltages applied to an intersecting pair of horizontal and vertical conductors cause the gas at the intersection of the two conductors to break down into a glowing plasma of electrons and ions. Picture definition is stored in a refresh buffer, and the firing voltages are applied to refresh the pixel positions (at the intersections of the conductors) 60 times per second. Alternating-current methods are used to provide faster application of the firing voltages and, thus, brighter displays.

Separation between pixels is provided by the electric field of the conductors. One disadvantage of plasma panels has been that they were strictly monochromatic devices, but systems are now available with multicolor capabilities.

**Thin-film electroluminescent displays** are similar in construction to plasma panels. The difference is that the region between the glass plates is filled with a phosphor, such as zinc sulfide doped with manganese, instead of a gas (Fig. 11).

When a sufficiently high voltage is applied to a pair of crossing electrodes, the phosphor becomes a conductor in the area of the intersection of the two electrodes.

Electrical energy is absorbed by the manganese atoms, which then release the energy as a spot of light similar to the glowing plasma effect in a plasma panel.

Electroluminescent displays require more power than plasma panels, and good color displays are harder to achieve.

A third type of emissive device is the **light-emitting diode (LED)**. A matrix of diodes is arranged to form the pixel positions in the display,

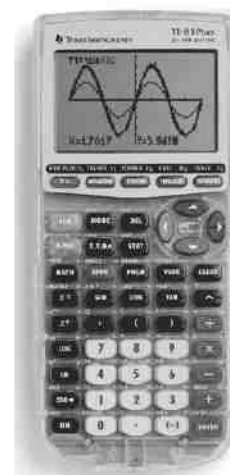


FIGURE 12 A handheld calculator with an LCD screen. (Courtesy of Texas Instruments)

and picture definition is stored in a refresh buffer. As in scan-line refreshing of a CRT, information is read from the refresh buffer and converted to voltage levels that are applied to the diodes to produce the light patterns in the display.

**Liquid-crystal displays (LCDs)** are commonly used in small systems, such as laptop computers and calculators (Fig. 12). These nonemissive devices produce a picture by passing polarized light from the surroundings or from an internal light source through a liquid-crystal material that can be aligned to either block or transmit the light.

The term *liquid crystal* refers to the fact that these compounds have a crystalline arrangement of molecules, yet they flow like a liquid. Flat-panel displays commonly use nematic (threadlike) liquid-crystal compounds that tend to keep the long axes of the rod-shaped molecules aligned. A flat-panel display can then be constructed with a nematic liquid crystal, as demonstrated in Figure 13. Two glass plates, each containing a light polarizer that is aligned at a right angle to the other plate, sandwich the liquid-crystal material.

Rows of horizontal,

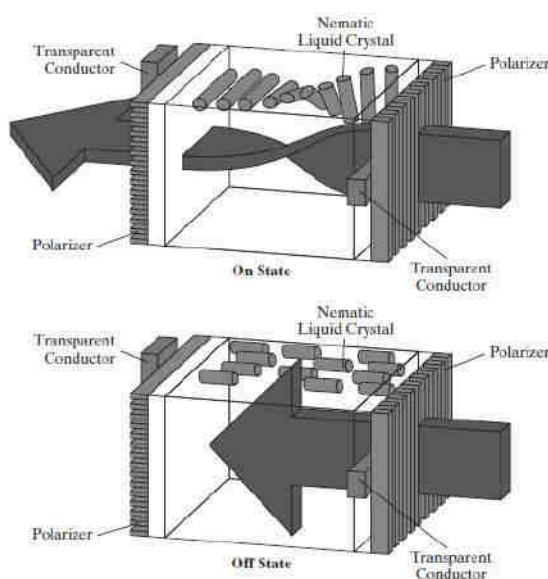


FIGURE 13  
The light-twisting, shutter effect used  
in the design of most LCD devices.

transparent conductors are built into one glass plate, and columns of vertical conductors are put into the other plate. The intersection of two conductors defines a pixel

position. Normally, the molecules are aligned as shown in the “on state” of Figure

13. Polarized light passing through the material is twisted so that it will pass through the opposite polarizer. The light is then reflected back to the viewer. To turn off the pixel, we apply a voltage to the two intersecting conductors to align the molecules so that the light is not twisted. This type of flat-panel device is referred to as a **passive-matrix** LCD. Picture definitions are stored in a refresh buffer, and the screen is refreshed at the rate of 60 frames per second, as in the emissive devices. Backlighting is also commonly applied using solid-state electronic devices, so that the system is not completely dependent on outside light sources.

Colors can be displayed by using different materials or dyes and by placing a triad

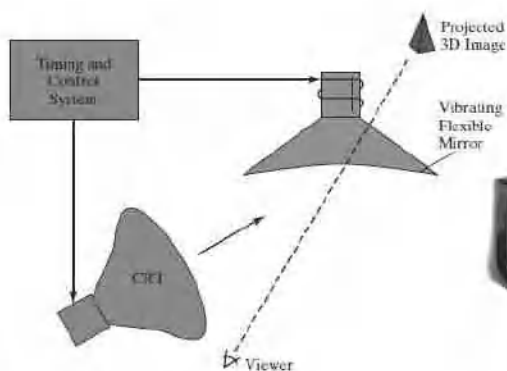


of color pixels at each screen location. Another method for constructing LCDs is to place a transistor at each pixel location, using thin-film transistor technology. The transistors are used to control the voltage at pixel locations and to prevent charge from gradually leaking out of the liquid-crystal cells. These devices are called **active-matrix** displays.

### Three-Dimensional Viewing Devices

Graphics monitors for the display of three-dimensional scenes have been devised using a technique that reflects a CRT image from a vibrating, flexible mirror (Fig. 14). As the varifocal mirror vibrates, it changes focal length. These

These vibrations are synchronized with the display of an object on a CRT so that each point on the object is reflected from the mirror into a spatial position corresponding to the distance of that point from a specified viewing location. This allows us to walk around an object or scene and view it from different sides.



**FIGURE 14**  
Operation of a three-dimensional display system using a vibrating mirror that changes focal length to match the depths of points in a scene.



**FIGURE 15**  
Glasses for viewing a stereoscopic scene in 3D. (Courtesy of X-PAND, XGD USA Inc.)

In addition to displaying three-dimensional images, these systems are often capable of displaying two-dimensional cross-sectional "slices" of objects selected at different depths, such as in medical applications to analyze data from ultrasonography and CAT scan devices, in geological applications to analyze topological and seismic data, in design applications involving solid objects, and in three-dimensional simulations of systems, such as molecules and terrain.

**Stereoscopic and Virtual-Reality Systems**

Another technique for representing a three-dimensional object is to display stereoscopic views of the object. This method does not produce true three-dimensional images, but it does provide a three-dimensional effect by presenting a different view to each eye of an observer so that scenes do appear to have depth.

To obtain a stereoscopic projection, we must obtain two views of a scene generated with viewing directions along the lines from the position of each eye (left and right) to the scene. We can construct the two views as computer-generated scenes with different viewing positions, or we can use a stereo camera pair to photograph an object or scene. When we simultaneously look at the left view with the left eye and the right view with the right eye, the two views merge into a single image and we perceive a scene with depth.

One way to produce a stereoscopic effect on a raster system is to display each of the two views on alternate refresh cycles. The screen is viewed through glasses, with each lens designed to act as a rapidly alternating shutter that is synchronized to block out one of the views. One such design (Figure 15) uses liquid-crystal shutters and an infrared emitter that synchronizes the glasses with the views on the screen.

Stereoscopic viewing is also a component in **virtual-reality** systems, where users can step into a scene and interact with the environment. A headset containing an optical system to generate the stereoscopic views can be used in conjunction with interactive input devices to locate and manipulate objects in the scene.

A sensing system in the headset keeps track of the viewer's position, so that the front and back of objects can be seen as the viewer "walks through" and interacts with the display. Another method for creating a virtual-reality environment is to use projectors to generate a scene within an arrangement of walls, where a viewer interacts with a virtual display using stereoscopic glasses and data gloves (Section 4).

Lower-cost, interactive virtual-reality environments can be set up using a graphics monitor, stereoscopic glasses, and a head-tracking device. The tracking device is placed above the video monitor and is used to record head movements, so that the viewing position for a scene can be changed as head position changes.

### **RASTER SCAN SYSTEMS**

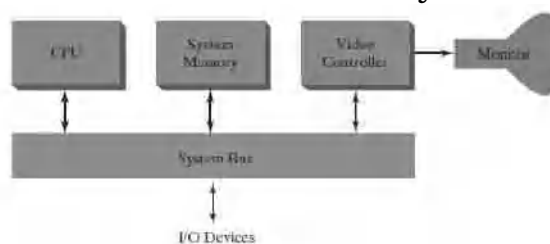
Interactive raster-graphics systems typically employ several processing units. In addition to the central processing unit (CPU), a special-purpose processor, called the **video controller** or **display controller**, is used to control the operation of the display device. Organization of a simple raster system is shown in Figure 16.

Here, the frame buffer can be anywhere in the system memory, and the video controller accesses the frame buffer to refresh the screen. In addition to the video controller, more sophisticated raster systems employ other processors as coprocessors and accelerators to implement various graphics operations.

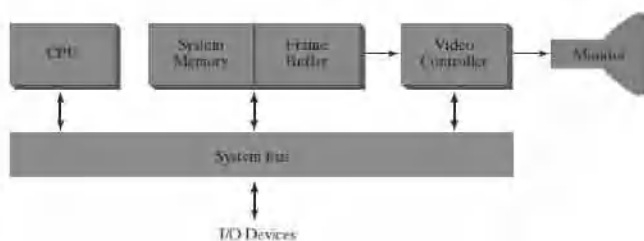
#### **Video Controller**

Figure 17 shows a commonly used organization for raster systems. A fixed area of the system memory is reserved for the frame buffer, and the video controller is given direct access to the frame-buffer memory.

Frame-buffer locations, and the corresponding screen positions, are referenced in



**FIGURE 16**  
Architecture of a simple raster-graphics system.



**FIGURE 17**  
Architecture of a raster system with a fixed portion of the system memory reserved for the frame buffer.

Cartesian coordinates. In an application program, we use the commands within a graphics software package to set coordinate positions for displayed objects relative to the origin of the Cartesian reference frame. Often, the coordinate origin is referenced at the lower-left corner of a screen display area by the software commands, although we can typically set the origin at any convenient location for a particular application. Figure 18 shows a two-dimensional Cartesian reference frame with the origin at the lower-left screen corner. The screen surface is then represented as the first quadrant of a two-dimensional system, with positive  $x$  values increasing from left to right and positive  $y$  values increasing from the bottom of the screen to the top. Pixel positions are then assigned integer  $x$  values that range from 0 to  $x_{\max}$  across the screen, left to right, and integer  $y$  values that vary from 0 to  $y_{\max}$ , bottom to top. However, hardware processes such as screen refreshing, as well as some software systems, reference the pixel positions from the top-left corner of the screen.

In Figure 19, the basic refresh operations of the video controller are diagrammed. Two registers are used to store the coordinate values for the screen pixels. Initially, the  $x$  register is set to 0 and the  $y$  register is set to the value for the top scan line. The contents of the frame

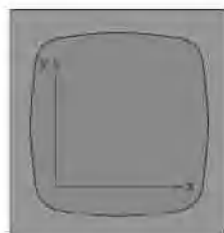


FIGURE 18  
A Cartesian reference frame with origin at the lower-left corner of a video monitor.

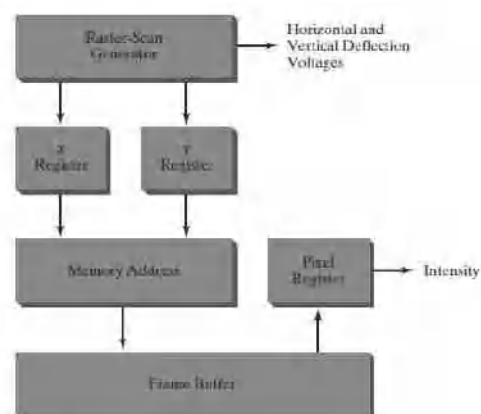


FIGURE 19  
Basic video-controller refresh operations.

buffer at this pixel position are then retrieved and used to set the intensity of the CRT beam. Then the  $x$  register is incremented by 1, and the process is repeated for the next pixel on the top scanline. This procedure continues for each pixel along the top scan line. After the last pixel on the top scan line has been processed, the  $x$  register is reset to 0 and the  $y$  register is set to the value for the next scan line down from the top of the screen. Pixels along this scan line are then processed in turn, and the procedure is repeated for each successive scan line. After cycling through all pixels along the bottom scan line, the video controller resets the registers to the first pixel position on the top scan line and the refresh process starts over.

Since the screen must be refreshed at a rate of at least 60 frames per second, the simple procedure illustrated in Figure 19 may not be accommodated by typical RAM chips if the cycle time is too slow. To speed up pixel processing, video controllers can retrieve multiple pixel values from the refresh buffer on each pass. The multiple pixel intensities are then stored in a separate register and used to control the CRT beam intensity for a group of adjacent pixels. When that group of pixels has been processed, the next block of pixel values is retrieved from the frame buffer.

A video controller can be designed to perform a number of other operations. For various applications, the video controller can retrieve pixel

values from different memory areas on different refresh cycles. In some systems, for example, multiple frame buffers are often provided so that one buffer can be used for refreshing while pixel values are being loaded into the other buffers. Then the current refresh buffer can switch roles with one of the other buffers. This provides a fast mechanism for generating real-time animations, for example, since different views of moving objects can be successively loaded into a buffer without interrupting a refresh cycle. Another video-controller task is the transformation of blocks of pixels, so that screen areas can be enlarged, reduced, or moved from one location to another during the refresh cycles. In addition, the video controller often contains a lookup table, so that pixel values in the frame buffer are used to access the lookup table instead of controlling the CRT beam intensity directly.

This provides a fast method for changing screen intensity values. Finally, some systems are designed to allow the video controller to mix the frame-buffer image with an input image from a television camera or other input device.

### Raster-Scan Display Processor

Figure 20 shows one way to organize the components of a raster system that contains a separate **display processor**, sometimes referred to as a **graphics controller** or a **display coprocessor**.

The purpose of the display processor is to free the CPU from the graphics chores.

In addition to the system memory, a separate display-processor memory area can be provided.

A major task of the display processor is digitizing a picture definition given in an application program into a set of pixel values for storage in the frame buffer. This digitization process is called **scan conversion**. Graphics commands specifying straight lines and other geometric objects are scan converted into a set of discrete points, corresponding to screen pixel positions. Scan converting a straight-line segment, for example, means that we have to locate the pixel positions closest to the line path and store the color for each position in the frame buffer. Similar methods are used for scan converting other objects in a picture definition. Characters can be defined with rectangular pixel grids, as in Figure 21, or they can be defined with outline shapes, as in Figure 22. The array size for character grids can vary from

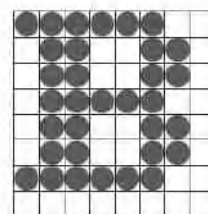


FIGURE 21  
A character defined as a rectangular grid of pixel positions.



FIGURE 22  
A character defined as an outline shape.

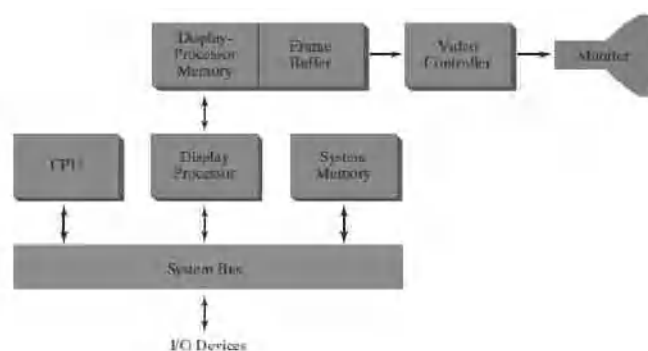


FIGURE 20  
Architecture of a raster-graphics system with a display processor.



about 5 by 7 to 9 by 12 or more for higher-quality displays. A character grid is displayed by superimposing the rectangular grid pattern into the frame buffer at a specified coordinate position. For characters that are defined as outlines, the shapes are scan-converted into the frame buffer by locating the pixel positions closest to the outline.

Display processors are also designed to perform a number of additional operations.

These functions include generating various line styles (dashed, dotted, or solid), displaying color areas, and applying transformations to the objects in a scene. Also, display processors are typically designed to interface with interactive input devices, such as a mouse.

In an effort to reduce memory requirements in raster systems, methods have been devised for organizing the frame buffer as a linked list and encoding the color information. One organization scheme is to store each scan line as a set of number pairs. The first number in each pair can be a reference to a color value, and the second number can specify the number of adjacent pixels on the scan line that are to be displayed in that color. This technique, called **run-length encoding**, can result in a considerable saving in storage space if a picture is to be constructed mostly with long runs of a single color each. A similar approach can be taken when pixel colors change linearly. Another approach is to encode the raster as a set of rectangular areas (**cell encoding**). The disadvantages of encoding runs are that color changes are difficult to record and storage requirements increase as the lengths of the runs decrease. In addition, it is difficult for the display controller to process the raster when many short runs are involved. Moreover, the size of the frame buffer is no longer a major concern, because of sharp declines in memory costs. Nevertheless, encoding methods can be useful in the digital storage and transmission of picture information.

## RANDOM SCAN SYSTEMS

### Random-Scan Displays

When operated as a **random-scan display** unit, a CRT has the electron beam directed only to those parts of the screen where a picture is to be displayed.

Pictures are generated as line drawings, with the electron beam tracing out the component lines one after the other. For this reason, random-scan monitors are also referred to as **vector displays** (or **stroke-writing displays** or **calligraphic displays**). The component lines of a picture can be drawn and refreshed by a random-

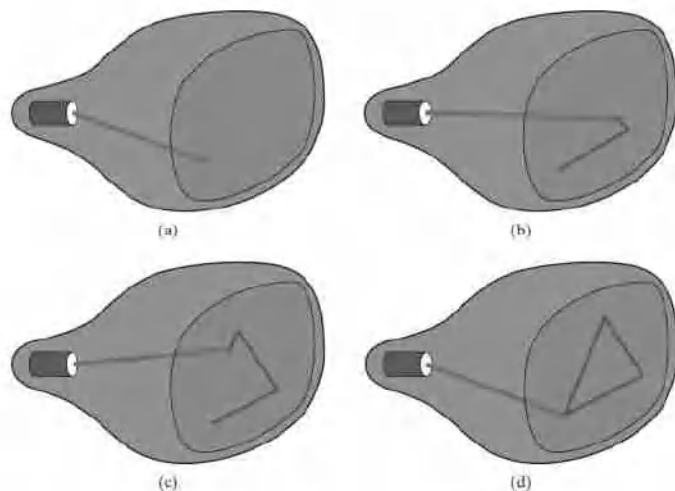


FIGURE 8  
A random-scan system draws the component lines of an object in any specified order.

scan system in any specified order (Fig. 8). A pen plotter operates in a similar way and is an example of a random-scan, hard-copy device.

Refresh rate on a random-scan system depends on the number of lines to be displayed on that system. Picture definition is now stored as a set of line-drawing commands in an area of memory referred to as the **display list, refresh display file, vector file, or display program**. To display a specified picture, the system cycles through the set of commands in the display file, drawing each component line in turn. After all line-drawing commands have been processed, the system cycles back to the first line command in the list. Random-scan displays are designed to draw all the component lines of a picture 30 to 60 times each second, with up to 100,000 "short" lines in the display list. When a small set of lines is to be displayed, each refresh cycle is delayed to avoid very high refresh rates, which could burn out the phosphor.

Random-scan systems were designed for line-drawing applications, such as architectural and engineering layouts, and they cannot display realistic shaded scenes. Since picture definition is stored as a set of line-drawing instructions rather than as a set of intensity values for all screen points, vector displays generally have higher resolutions than raster systems. Also, vector displays produce smooth line drawings because the CRT beam directly follows the line path. A raster system, by contrast, produces jagged lines that are plotted as discrete point sets. However, the greater flexibility and improved line-drawing capabilities of raster systems have resulted in the abandonment of vector technology.

### **INTERACTIVE INPUT DEVICES**

Graphics workstations can make use of various devices for data input. Most systems have a keyboard and one or more additional devices specifically designed for interactive input. These include a mouse, trackball, spaceball, and joystick. Some other input devices used in particular applications are digitizers, dials, buttonboxes, data gloves, touch panels, image scanners, and voice systems.

#### **Keyboards, Button Boxes, and Dials**

An alphanumeric **keyboard** on a graphics system is used primarily as a device for entering text strings, issuing certain commands, and selecting menu options. The keyboard is an efficient device for inputting such nongraphic data as picture labels associated with a graphics display. Keyboards can also be provided with features to facilitate entry of screen coordinates, menu selections, or graphics functions.

Cursor-control keys and function keys are common features on general-purpose keyboards. Function keys allow users to select frequently accessed operations with a single keystroke, and cursor-control keys are convenient for selecting a displayed object or a location by positioning the screen cursor. A keyboard can also contain other types of cursor-positioning devices, such as a trackball or joystick, along with a numeric keypad for fast entry of numeric data. In addition to these features, some keyboards have an ergonomic design that provides adjustments for relieving operator fatigue.

For specialized tasks, input to a graphics application may come from a set of buttons, dials, or switches that select data values or customized graphics operations.

Buttons and switches are often used to input predefined functions, and dials are common devices for entering scalar values. Numerical values within some defined range are selected for input with dial rotations. A potentiometer is used to measure dial rotation, which is then converted to the corresponding numerical value.

### Mouse Devices

A **mouse** is a small handheld unit that is usually moved around on a flat surface to position the screen cursor. One or more buttons on the top of the mouse provide a mechanism for communicating selection

**FIGURE 23**  
A wireless computer mouse designed with many user-programmable controls  
(Courtesy of Logitech®)



information to the computer; wheels or rollers on the bottom of the mouse can be used to record the amount and direction of movement. Another method for detecting mouse motion is with an optical sensor. For some optical systems, the mouse is moved over a special mousepad that has a grid of horizontal and vertical lines. The optical sensor detects movement across the lines in the grid. Other optical mouse systems can operate on any surface. Some mouse systems are cordless, communicating with computer processors using digital radio technology.

Since a mouse can be picked up and put down at another position without change in cursor movement, it is used for making relative changes in the position of the screen cursor. One, two, three, or four buttons are included on the top of the mouse for signaling the execution of operations, such as recording cursor position or invoking a function. Most general-purpose graphics systems now include a mouse and a keyboard as the primary input devices.

Additional features can be included in the basic mouse design to increase the number of allowable input parameters and the functionality of the mouse.

The Logitech G700 wireless gaming mouse in Figure 23 features 13 separately programmable control inputs. Each input can be configured to perform a wide range of actions, from traditional single-click inputs to macro operations containing multiple key strokes, mouse events, and pre-programmed delays between operations. The laser-based optical sensor can be configured to control the degree of sensitivity to motion, allowing the mouse to be used in situations requiring different levels of control over cursor movement. In addition, the mouse can hold up to five different configuration profiles to allow the configuration to be switched easily when changing applications.

### Trackballs and Space balls

A **trackball** is a ball device that can be rotated with the fingers or palm of the hand to produce screen-cursor movement. Potentiometers, connected to the ball, measure the amount and direction of rotation. Laptop keyboards are often equipped with a trackball to eliminate the extra space required by a mouse. A trackball also can be mounted on other devices, or it

can be obtained as a separate add-on unit that contains two or three control buttons.

An extension of the two-dimensional trackball concept is the **spaceball**, which provides six degrees of freedom. Unlike the trackball, a spaceball does not actually move. Strain gauges measure the amount of pressure applied to the spaceball to provide input for spatial positioning and orientation as the ball is pushed or pulled in various directions. Spaceballs are used for three-dimensional positioning and selection operations in virtual-reality systems, modeling, animation, CAD, and other applications.

### **Joysticks**

Another positioning device is the **joystick**, which consists of a small, vertical lever (called the stick) mounted on a base. We use the joystick to steer the screen cursor around. Most joysticks select screen positions with actual stick movement; others respond to pressure on the stick. Some joysticks are mounted on a keyboard, and some are designed as stand-alone units.

The distance that the stick is moved in any direction from its center position corresponds to the relative screen-cursor movement in that direction.

Potentiometers mounted at the base of the joystick measure the amount of movement, and springs return the stick to the center position when it is released. One or more buttons can be programmed to act as input switches to signal actions that are to be executed once a screen position has been selected.

In another type of movable joystick, the stick is used to activate switches that cause the screen cursor to move at a constant rate in the direction selected. Eight switches, arranged in a circle, are sometimes provided so that the stick can select any one of eight directions for cursor movement. Pressure-sensitive joysticks, also called *isometric joysticks*, have a non-movable stick. A push or pull on the stick is measured with strain gauges and converted to movement of the screen cursor in the direction of the applied pressure.

### **Data Gloves**

A **data glove** is a device that fits over the user's hand and can be used to grasp a "virtual object." The glove is constructed with a series of sensors that detect hand and finger motions. Electromagnetic coupling between transmitting antennas and receiving antennas are used to provide information about the position and orientation of the hand. The transmitting and receiving antennas can each be structured as a set of three mutually perpendicular coils, forming a three dimensional Cartesian reference system. Input from the glove is used to position or manipulate objects in a virtual scene. A two-dimensional projection of the scene can be viewed on a video monitor, or a three-dimensional projection can be viewed with a headset.

### **Digitizers**

A common device for drawing, painting, or interactively selecting positions is a **digitizer**. These devices can be designed to input coordinate values in either a two-dimensional or a three-dimensional space. In engineering or architectural applications, a digitizer is often used to scan a



drawing or object and to input a set of discrete coordinate positions. The input positions are then joined with straight-line segments to generate an approximation of a curve or surface shape.

One type of digitizer is the **graphics tablet** (also referred to as a *data tablet*), which is used to input two-dimensional coordinates by activating a hand cursor or stylus at selected positions on a flat surface. A hand cursor contains crosshairs for sighting positions, while a stylus is a pencil-shaped device that is pointed at positions on the tablet. The tablet size varies from 12 by 12 inches for desktop models to 44 by 60 inches or larger for floor models. Graphics tablets provide a highly accurate method for selecting coordinate positions, with an accuracy that varies from about 0.2 mm on desktop models to about 0.05 mm or less on larger models.

Many graphics tablets are constructed with a rectangular grid of wires embedded in the tablet surface. Electromagnetic pulses are generated in sequence along the wires, and an electric signal is induced in a wire coil in an activated stylus or hand-cursor to record a tablet position. Depending on the technology, signal strength, coded pulses, or phase shifts can be used to determine the position on the tablet.

An *acoustic (or sonic) tablet* uses sound waves to detect a stylus position. Either strip microphones or point microphones can be employed to detect the sound emitted by an electrical spark from a stylus tip. The position of the stylus is calculated by timing the arrival of the generated sound at the different microphone positions. An advantage of two-dimensional acoustic tablets is that the microphones can be placed on any surface to form the “tablet” work area. For example, the microphones could be placed on a book page while a figure on that page is digitized.

Three-dimensional digitizers use sonic or electromagnetic transmissions to record positions. One electromagnetic transmission method is similar to that employed in the data glove: A coupling between the transmitter and receiver is used to compute the location of a stylus as it moves over an object surface. As the points are selected on a nonmetallic object, a wire-frame outline of the surface is displayed on the computer screen. Once the surface outline is constructed, it can be rendered using lighting effects to produce a realistic display of the object.

### **Image Scanners**

Drawings, graphs, photographs, or text can be stored for computer processing with an **image scanner** by passing an optical scanning mechanism over the information to be stored. The gradations of grayscale or color are then recorded and stored in an array. Once we have the internal representation of a picture, we can apply transformations to rotate, scale, or crop the picture to a particular screen area. We can also apply various image-processing methods to modify the array representation of the picture. For scanned text input, various editing operations can be performed on the stored documents. Scanners are available in a variety of sizes and capabilities, including small handheld models, drum scanners, and flatbed scanners.

## Touch Panels

As the name implies, **touch panels** allow displayed objects or screen positions to be selected with the touch of a finger. A typical application of touch panels is for the selection of processing options that are represented as a menu of graphical icons. Some monitors are designed with touch screens. Other systems can be adapted for touch input by fitting a transparent device containing a touch-sensing mechanism over the video monitor screen. Touch input can be recorded using optical, electrical, or acoustical methods.

Optical touch panels employ a line of infrared light-emitting diodes (LEDs) along one vertical edge and along one horizontal edge of the frame. Light detectors are placed along the opposite vertical and horizontal edges. These detectors are used to record which beams are interrupted when the panel is touched. The two crossing beams that are interrupted identify the horizontal and vertical coordinates of the screen position selected. Positions can be selected with an accuracy of about 1/4 inch. With closely spaced LEDs, it is possible to break two horizontal or two vertical beams simultaneously. In this case, an average position between the two interrupted beams is recorded. The LEDs operate at infrared frequencies so that the light is not visible to a user.

An electrical touch panel is constructed with two transparent plates separated by a small distance. One of the plates is coated with a conducting material, and the other plate is coated with a resistive material. When the outer plate is touched, it is forced into contact with the inner plate. This contact creates a voltage drop across the resistive plate that is converted to the coordinate values of the selected screen position.

In acoustical touch panels, high-frequency sound waves are generated in horizontal and vertical directions across a glass plate. Touching the screen causes part of each wave to be reflected from the finger to the emitters. The screen position at the point of contact is calculated from a measurement of the time interval between the transmission of each wave and its reflection to the emitter.

## Light Pens

**Light pens** are pencil-shaped devices used to select screen positions by detecting the light coming from points on the CRT screen. They are sensitive to the short burst of light emitted from the phosphor coating at the instant the electron beam strikes a particular point. Other light sources, such as the background light in the room, are usually not detected by a light pen. An activated light pen, pointed at a spot on the screen as the electron beam lights up that spot, generates an electrical pulse that causes the coordinate position of the electron beam to be recorded. As with cursor-positioning devices, recorded light-pen coordinates can be used to position an object or to select a processing option.

Although light pens are still with us, they are not as popular as they once were because they have several disadvantages compared to other input devices that have been developed. For example, when a light pen is pointed at the screen, part of the screen image is obscured by the hand and pen. In addition, prolonged use of the light pen can cause arm fatigue, and light pens require special implementations for some applications because they cannot detect positions within black areas. To be able to select positions in any

screen area with a light pen, we must have some nonzero light intensity emitted from each pixel within that area. In addition, lightpens sometimes give false readings due to background lighting in a room.

### Voice Systems

Speech recognizers are used with some graphics workstations as input devices for voice commands. The **voice system** input can be used to initiate graphics operations or to enter data. These systems operate by matching an input against a predefined dictionary of words and phrases.

A dictionary is set up by speaking the command words several times. The system then analyzes each word and establishes a dictionary of word frequency patterns, along with the corresponding functions that are to be performed.

Later, when a voice command is given, the system searches the dictionary for a frequency-pattern match. A separate dictionary is needed for each operator using the system. Input for a voice system is typically spoken into a microphone mounted on a headset; the microphone is designed to minimize input of background sounds. Voice systems have some advantage over other input devices because the attention of the operator need not switch from one device to another to enter a command.

### HARD COPY DEVICES

We can obtain hard-copy output for our images in several formats.

For presentations or archiving, we can send image files to devices or service bureaus that will produce overhead



**FIGURE 24**  
A picture generated on a dot-matrix printer, illustrating how the density of dot patterns can be varied to produce light and dark areas. (Courtesy of Apple Computer, Inc.)

transparencies, 35mm slides, or film. Also, we can put our pictures on paper by directing graphics output to a printer or plotter.

The quality of the pictures obtained from an output device depends on dot size and the number of dots per inch, or lines per inch, that can be displayed.

To produce smooth patterns, higher-quality printers shift dot positions so that adjacent dots overlap.

Printers produce output by either impact or nonimpact methods. *Impact* printers press formed character faces against an inked ribbon onto the paper. A line printer is an example of an impact device, with the typefaces mounted on bands, chains, drums, or wheels. *Nonimpact* printers and plotters use laser techniques, ink-jet sprays, electrostatic methods, and electrothermal methods to get images onto paper.

Character impact printers often have a *dot-matrix* print head containing a rectangular array of protruding wire pins, with the number of pins varying depending upon the quality of the printer. Individual characters or graphics patterns are obtained by retracting certain pins so that the remaining pins form the pattern to be printed. Figure 24 shows a picture printed on a dot-matrix printer.

In a *laser* device, a laser beam creates a charge distribution on a rotating drum coated with a photoelectric material, such as selenium. Toner is applied to the drum and then transferred to paper. *Ink-jet* methods produce output by squirting ink in horizontal rows across a roll of paper wrapped on a drum. The electrically charged ink stream is deflected by an electric field to produce dot-matrix patterns.

An *electrostatic* device places a negative charge on the paper, one complete row at a time across the sheet. Then the paper is exposed to a positively charged toner. This causes the toner to be attracted to the negatively charged areas, where it adheres to produce the specified output. Another output technology is the *electrothermal* printer. With these systems, heat is applied to a dot-matrix print head to output patterns on heat-sensitive paper.

We can get limited color output on some impact printers by using different colored ribbons. Non-impact devices use various techniques to combine three different color pigments (cyan, magenta, and yellow) to produce a range of color patterns. Laser and electrostatic devices deposit the three pigments on separate passes; ink-jet methods shoot the three colors simultaneously on a single pass along each print line.

Drafting layouts and other drawings are typically generated with ink-jet or pen plotters. A pen plotter has one or more pens mounted on a carriage, or crossbar, that spans a sheet of paper. Pens with varying colors and widths are used to produce a variety of shadings and line styles. Wet-ink, ballpoint, and felt-tip pens are all possible choices for use with a pen plotter. Plotter paper can lie flat or it can be rolled onto a drum or belt. Crossbars can be either movable or stationary, while the pen moves back and forth along the bar. The paper is held in position using clamps, a vacuum, or an electrostatic charge.

## **GRAPHICS SOFTWARE**

So far, we have mainly considered graphics applications on an isolated system with a single user. However, multiuser environments and computer networks are now common elements in many graphics applications. Various resources, such as processors, printers, plotters, and data files, can be distributed on a network and shared by multiple users.

A graphics monitor on a network is generally referred to as a **graphics server**, or simply a **server**. Often, the monitor includes standard input devices such as a keyboard and a mouse or trackball. In that case, the system can provide input, as well as being an output server. The computer on the network that is executing a graphics application program is called the **client**, and the output of the program is displayed on a server. A workstation that includes processors, as well as a monitor and input devices, can function as both a server and a client.

When operating on a network, a client computer transmits the instructions for displaying a picture to the monitor (server). Typically, this is accomplished by collecting the instructions into packets before transmission instead of sending the individual graphics instructions one at a time over the network. Thus, graphics software packages often contain commands that affect packet transmission, as well as the commands for creating pictures.

## AREA FILL ATTRIBUTES

Most graphics packages limit fill areas to polygons because they are described with linear equations. A further restriction requires fill areas to be convex polygons, so that scan lines do not intersect more than two boundary edges. However, in general, we can fill any specified regions, including circles, ellipses, and other objects with curved boundaries. Also, application systems, such as paint programs, provide fill options for arbitrarily shaped regions.

### Fill Styles

A basic fill-area attribute provided by a general graphics library is the display style of the interior. We can display a region with a single color, a specified fill pattern, or in a “hollow” style by showing only the boundary of the region. These three fill styles are illustrated in Figure 5. We can also fill selected regions of a scene using various brush styles, color-blending combinations, or textures. Other options include specifications for the display of the boundaries of a fill area.

For polygons, we could show the edges in different colors, widths, and styles; and we can select different display attributes for the front and back faces of a region.

Fill patterns can be defined in rectangular color arrays that list different colors for different positions in the array. Alternatively, a fill pattern could be specified as a bit array that indicates which relative positions are to be displayed in a single selected color. An array specifying a fill pattern is a *mask* that is to be applied to the display area. Some graphics systems provide an option for selecting an arbitrary initial position for overlaying the mask. From this starting position, the mask is replicated in the horizontal and vertical directions until the display area is filled with nonoverlapping copies of the pattern. Where the pattern overlaps specified fill areas, the array pattern indicates which pixels should be displayed in a particular color. This process of filling an area with a rectangular pattern is called **tiling**, and a rectangular fill pattern is sometimes referred to as a **tiling pattern**. Sometimes, predefined fill patterns are available in a system, such as the *hatch* fill patterns shown in Figure 6.

### Color-Blended Fill Regions

It is also possible to combine a fill pattern with background colors in various ways. A pattern could be combined with background colors using a *transparency factor* that determines how much of the background should be mixed with the object color.

Some fill methods using blended colors have been referred to as **soft-fill** algorithms. One use for

these fill methods is to soften the fill colors at object borders that have been blurred to antialias the edges. Another application of a soft-fill algorithm is to allow repainting of a color area that was originally filled with a semitransparent brush, where the current color is then a mixture of the brush color and the background colors “behind” the area. In

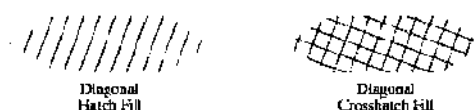


FIGURE 6  
Areas filled with hatch patterns.



either case, we want the new fill color to have the same variations over the area as the current fill color.

### CHARACTER ATTRIBUTES

We control the appearance of displayed characters with attributes such as font, size, color, and orientation. In many packages, attributes can be set both for entire character strings (text) and for individual characters that can be used for special purposes such as plotting a data graph.

There are a great many possible text-display options. First, there is the choice of font (or typeface), which is a set of characters with a particular design style such as New York, Courier, Helvetica, London, Times Roman, and various special symbol groups. The characters in a selected font can also be displayed with assorted underlining styles (solid, -d-o-t-t-e-d-, double), in **boldface**, in *italic*, and in OUTLINE or shadow styles.

Color settings for displayed text can be stored in the system attribute list and used by the procedures that generate character definitions in the frame buffer.

When a character string is to be displayed, the current color is used to set pixel values in the frame buffer corresponding to the character shapes and positions.

We could adjust text size by scaling the overall dimensions (height and width) of characters or by scaling only the height or the width. Character size (height) is specified by printers and compositors in *points*, where 1 point is about 0.035146

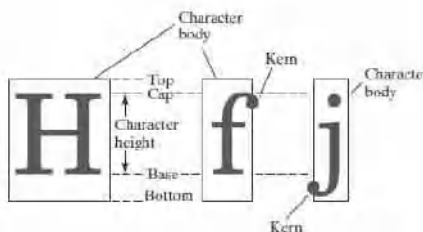


FIGURE 11  
Examples of character bodies.

centimeters (or 0.013837 inch, which is approximately 172 inch). For example, the characters in this book are set in a 10-point font. Point measurements specify the size of the *body* of a character (Figure 11), but different fonts with the same point specifications can have different character sizes, depending on the design of the typeface. The distance between the *bottomline* and the *topline* of the character body is the same for all characters in a particular size and typeface, but the body width may vary. *Proportionally spaced fonts* assign a smaller body width to narrow characters such as *i*, *j*, *l*, and *f* compared to broad characters such as *W* or *M*. *Character height* is defined as the distance between the *baseline* and the *capline* of characters. *Kerned* characters, such as *f* and *j* in Figure 11, typically extend beyond the character body limits, and letters with descenders (*g*, *j*, *p*, *q*, *y*) extend below the baseline. Each character is positioned within the character body by a font designer in such a way that suitable spacing is attained along and between print lines when text is displayed with character bodies touching.

Sometimes, text size is adjusted without changing the width-to-height ratio of characters. Figure 12 shows a character string displayed with three different character heights, while maintaining the ratio of width to height. Examples of text displayed with a constant height and varying widths are given in Figure 13.

Spacing between characters is another attribute that can often be assigned to a character string. Figure 14 shows a character string displayed with three different settings for the intercharacter spacing.

The orientation for a character string can be set according to the direction of a **character up vector**. Text is then displayed so that the orientation of characters from baseline to capline is in the direction of the up vector. For example, with the direction of the up vector at 45°, text would be displayed as shown in Figure 15.

A procedure for orienting text could rotate characters so that the sides of character bodies, from baseline to capline, are aligned with the up vector. The rotated character shapes are then scan

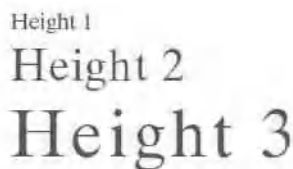


FIGURE 12 Text strings displayed with different character-height settings and a constant width-to-height ratio.

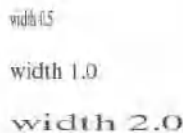


FIGURE 13 Text strings displayed with varying sizes for the character widths and a fixed height.

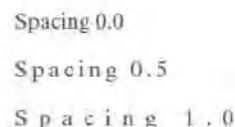


FIGURE 14 Text strings displayed with different character-spacing values.

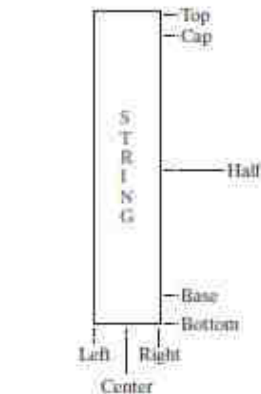
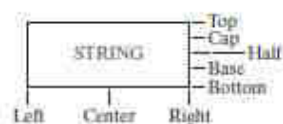


FIGURE 19 Character alignments for horizontal and vertical strings.

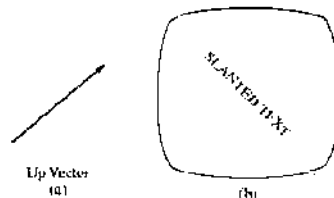


FIGURE 15 Direction of the up vector (a) controls the orientation of displayed text (b).

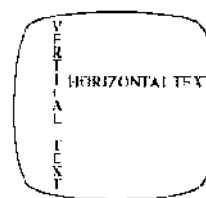


FIGURE 16 Text-path attributes can be set to produce horizontal or vertical arrangements of character strings.

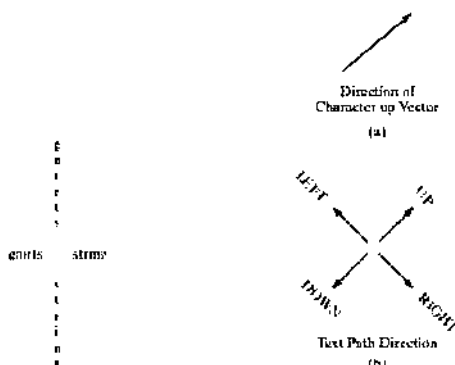


FIGURE 17 A text string displayed with the four text-path options: left, right, up, and down.

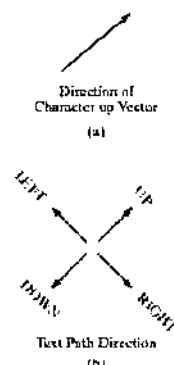


FIGURE 18 An up-vector specification (a) and associated directions for the text path (b).

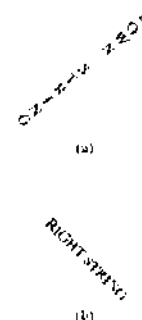


FIGURE 19 The 45° up vector (a) produces the display (a) for a down path and the display (b) for a right path.



FIGURE 21 Character-string alignments.

converted into the frame buffer.

It is useful in many applications to be able to arrange character strings vertically or horizontally. Examples of this are given in Figure 16. We could also arrange the characters in a text string so that the string is displayed forward or backward. Examples of text displayed with these options are shown in Figure 17. A procedure for implementing text-path orientation adjusts the position of the individual

characters in the frame buffer according to the option selected.

Character strings could also be oriented using a combination of up-vector and text-path specifications to produce slanted text. Figure 18 shows the directions of character strings generated by various text path settings for a 45° up vector.

Examples of character strings generated for text-path values *down* and *right* with this up vector are illustrated in Figure 19.

Another possible attribute for character strings is alignment. This attribute specifies how text is to be displayed with respect to a reference position. For example, individual characters could be aligned according to the base lines or the character centers. Figure 20 illustrates typical character positions for horizontal and vertical alignments. String alignments are also possible, and Figure 21 shows common alignment positions for horizontal and vertical text labels. In some graphics packages, a text-precision attribute is also available. This parameter specifies the amount of detail and the particular processing options that are to be used with a text string. For a low-precision text string, many attribute selections, such as text path, are ignored, and faster procedures are used for processing the characters through the viewing pipeline.

Finally, a library of text-processing routines often supplies a set of special characters, such as a small circle or cross, which are useful in various applications. Most often these characters are used as marker symbols in network layouts or in graphing data sets. The attributes for these marker symbols are typically *color* and *size*.

We have two methods for displaying characters with the OpenGL package. Either we can design a font set using the bitmap functions in the core library, or we can invoke the GLUT character-generation routines. The GLUT library contains functions for displaying predefined bitmap and stroke character sets. Therefore, the character attributes we can set are those that apply to either bitmaps or line segments.

For either bitmap or outline fonts, the display color is determined by the current color state. In general, the spacing and size of characters is determined by the font designation, such as **GLUT\_BITMAP\_9\_BY\_15** and **GLUT\_STROKE\_MONO\_ROMAN**. However, we can also set the line width and line type for the outline fonts. We specify the width for a line with the **glLineWidth** function, and we select a line type with the **glLineStipple** function. The GLUT stroke fonts will then be displayed using the current values we specified for the OpenGL line-width and line-type attributes.

We can accomplish some other text-display characteristics using transformation functions. The transformation routines allow us to scale, position, and rotate the GLUT stroke characters in either two-dimensional space or three-dimensional space. In addition, the three-dimensional viewing transformations can be used to generate other display effects.

### **INQUIRY FUNCTION**

We can retrieve current values for any of the state parameters, including attribute settings, using OpenGL **query functions**. These functions copy specified state values into an array, which we can save for later reuse or to check the current state of the system if an error occurs.

For current attribute values we use an appropriate “**glGet**” function, such as

```
glGetBooleanv ( )  
glGetFloatv ( )  
glGetIntegerv ( )  
glGetDoublev ( )
```

In each of the preceding functions, we specify two arguments. The first argument is an OpenGL symbolic constant that identifies an attribute or other state parameter.

The second argument is a pointer to an array of the data type indicated by the function name. For instance, we can retrieve the current RGBA floating-point color settings with

```
glGetFloatv (GL_CURRENT_COLOR, colorValues);
```

The current color components are then passed to the array **colorValues**. To obtain the integer values for the current color components, we invoke the **glGet-Integerv** function. In some cases, a type conversion may be necessary to return the specified data type.

Other OpenGL constants, such as **GL POINT SIZE**, **GL LINE WIDTH**, and **GL CURRENT RASTER POSITION**, can be used in these functions to return current state values. Also, we could check the range of point sizes or linewidths that are supported using the constants **GL POINT SIZE RANGE** and **GL LINE WIDTH RANGE**.

Although we can retrieve and reuse settings for a single attribute with the **glGet** functions, OpenGL provides other functions for saving groups of attributes and reusing their values. We consider the use of these functions for saving current attribute settings in the next section.

There are many other state and system parameters that are often useful to query. For instance, to determine how many bits per pixel are provided in the frame buffer on a particular system, we can ask the system how many bits are available for each individual color component, such as

```
glGetIntegerv (GL_RED_BITS, redBitSize);
```

Here, array **redBitSize** is assigned the number of red bits available in each of the buffers (frame buffer, depth buffer, accumulation buffer, and stencil buffer).

Similarly, we can make an inquiry for the other color bits using **GL GREEN BITS**, **GL BLUE BITS**, **GL ALPHA BITS**, or **GL INDEX BITS**.

We can also find out whether edge flags have been set, whether a polygon face was tagged as a front face or a back face, and whether the system supports double buffering. In addition, we can inquire whether certain routines, such as color blending, line stippling or antialiasing, have been enabled or disabled.

## **OUTPUT PRIMITIVES**

A general software package for graphics applications, sometimes referred to as a computer-graphics application programming interface (CG API), provides a library of functions that we can use within a programming language such as C++ to create pictures. The set of library functions can be subdivided into several categories. One of the first things we need to do when creating a picture is to describe the component parts of the scene to be displayed.

Picture components could be trees and terrain, furniture and walls, storefronts and street scenes, automobiles and billboards, atoms and molecules, or stars and galaxies. For each type of scene, we need to describe the structure of the individual objects and their coordinate locations within the scene. Those functions in a graphics package that we use to describe the various picture components are called the **graphics output primitives**, or simply **primitives**. The output primitives describing the geometry of objects are typically referred to as geometric primitives. Point positions and straight-line segments are the simplest geometric primitives. Additional geometric primitives that can be available in a graphics package include circles and other conic sections, quadric surfaces, spline curves and surfaces, and polygon color areas. Also, most graphics systems provide some functions for displaying character strings. After the geometry of a picture has been specified within a selected coordinate reference frame, the output primitives are projected to a two-dimensional plane, corresponding to the display area of an output device, and scan converted into integer pixel positions within the frame buffer.

In this chapter, we introduce the output primitives available in OpenGL, and discuss their use.

### Coordinate Reference Frames

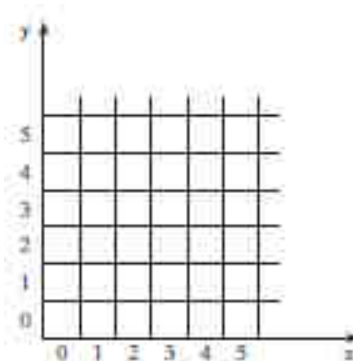
To describe a picture, we first decide upon a convenient Cartesian coordinate system, called the *world-coordinate reference frame*, which could be either two-dimensional or three-dimensional. We then describe the objects in our picture by giving their geometric specifications in terms of positions in world coordinates.

For instance, we define a straight-line segment with two endpoint positions, and a polygon is specified with a set of positions for its vertices. These coordinate positions are stored in the scene description along with other information about the objects, such as their color and their **coordinate extents**, which are the minimum and maximum  $x$ ,  $y$ , and  $z$  values for each object. A set of coordinate extents is also described as a **bounding box** for an object. For a two-dimensional figure, the coordinate extents are sometimes called an object's **bounding rectangle**.

Objects are then displayed by passing the scene information to the viewing routines, which identify visible surfaces and ultimately map the objects to positions on the video monitor. The scan-conversion process stores information about the scene, such as color values, at the appropriate locations in the frame buffer, and the objects in the scene are displayed on the output device.

### Screen Coordinates

Locations on a video monitor are referenced in integer **screen coordinates**, which correspond to the pixel positions in the frame buffer. Pixel coordinate values give the *scan line number* (the  $y$  value) and the *column number* (the  $x$  value along a scan line). Hardware processes, such as screen refreshing, typically address pixel positions with respect to the top-left corner of the screen. Scan



**FIGURE 1**

Pixel positions referenced with respect to the lower-left corner of a screen area.



lines are then referenced from 0, at the top of the screen, to some integer value,  $y_{\max}$ , at the bottom of the screen, and pixel positions along each scan line are numbered from 0 to  $x_{\max}$ , left to right. However, with software commands, we can set up any convenient reference frame for screen positions. For example, we could specify an integer range for screen positions with the coordinate origin at the lower-left of a screen area (Figure 1), or we could use noninteger Cartesian values for a picture description. The coordinate values we use to describe the geometry of a scene are then converted by the viewing routines to integer pixel positions within the frame buffer.

Scan-line algorithms for the graphics primitives use the defining coordinate descriptions to determine the locations of pixels that are to be displayed. For example, given the endpoint coordinates for a line segment, a display algorithm must calculate the positions for those pixels that lie along the line path between the endpoints. Since a pixel position occupies a finite area of the screen, the finite size of a pixel must be taken into account by the implementation algorithms.

For the present, we assume that each integer screen position references the center of a pixel area.

Once pixel positions have been identified for an object, the appropriate color values must be stored in the frame buffer. For this purpose, we will assume that we have available a low-level procedure of the form

**setPixel (x, y);**

This procedure stores the current color setting into the frame buffer at integer position  $(x, y)$ , relative to the selected position of the screen-coordinate origin. We sometimes also will want to be able to retrieve the current frame-buffer setting for a pixel location. So we will assume that we have the following low-level function for obtaining a frame-buffer color value:

**getPixel (x, y, color);**

In this function, parameter **color** receives an integer value corresponding to the combined red, green, and blue (RGB) bit codes stored for the specified pixel at position  $(x, y)$ .

Although we need only specify color values at  $(x, y)$  positions for a two-dimensional picture, additional screen-coordinate information is needed for

three-dimensional scenes. In this case, screen coordinates are stored as three-dimensional values, where the third dimension references the depth of object positions relative to a viewing position. For a two-dimensional scene, all depth values are 0.

### **Absolute and Relative Coordinate Specifications**

So far, the coordinate references that we have discussed are stated as **absolute coordinate** values. This means that the values specified are the actual positions within the coordinate system in use.

However, some graphics packages also allow positions to be specified using **relative coordinates**. This method is useful for various graphics applications, such as producing drawings with pen plotters, artist's drawing and painting systems, and graphics packages for publishing and printing applications. Taking this approach, we can specify a coordinate position as an offset from the last position that was referenced (called the

**current position**). For example, if location(3, 8) is the last position that has been referenced in an application program, a relative coordinate specification of (2, -1) corresponds to an absolute position of (5, 7). An additional function is then used to set a current position before any coordinates for primitive functions are specified. To describe an object, such as a series of connected line segments, we then need to give only a sequence of relative coordinates (offsets), once a starting position has been established. Options can be provided in a graphics system to allow the specification of locations using either relative or absolute coordinates. In the following discussions, we will assume that all coordinates are specified as absolute references unless explicitly stated otherwise.

### Specifying A Two-Dimensional World-Coordinate Reference Frame in OpenGL

The **gluOrtho2D** command is a function we can use to set up any two-dimensional Cartesian reference frame. The arguments for this function are the four values defining the  $x$  and  $y$  coordinate limits for the picture we want to display.

Since the **gluOrtho2D** function specifies an orthogonal projection, we need also to be sure that the coordinate values are placed in the OpenGL projection matrix. In addition, we could assign the identity matrix as the projection matrix before defining the world-coordinate range. This would ensure that the coordinate values were not accumulated with any values we may have previously set for the projection matrix. Thus, for our initial two-dimensional examples, we can define the coordinate frame for the screen display window with the following statements:

```
glMatrixMode (GL_PROJECTION);  
glLoadIdentity ( );  
gluOrtho2D (xmin, xmax, ymin, ymax);
```

The display window will then be referenced by coordinates (**xmin, ymin**) at the lower-left corner and by coordinates (**xmax, ymax**) at the upper-right corner, as shown in Figure 2.

We can then designate one or more graphics primitives for display using the coordinate reference specified in the **gluOrtho2D** statement. If the coordinate extents of a primitive are within the coordinate range

of the display window, all of the primitive will be displayed. Otherwise, only those parts of the primitive within the display-window coordinate limits will be shown. Also, when we set up the geometry describing a picture, all positions for the OpenGL primitives must be given in absolute coordinates, with respect to the reference frame defined in the **gluOrtho2D** function.

### OpenGL Point Functions

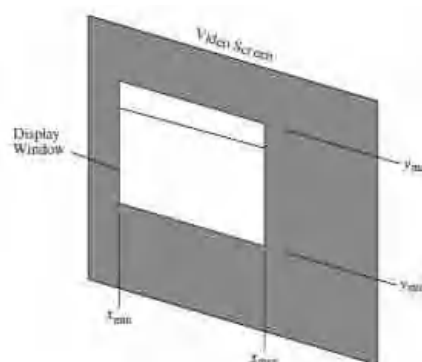


FIGURE 2  
World-coordinate limits for a display window, as specified in the **gluOrtho2D** function.

To specify the geometry of a point, we simply give a coordinate position in the world reference frame. Then this coordinate position, along with other geometric descriptions we may have in our scene, is passed to the viewing routines. Unless we specify other attribute values, OpenGL primitives are displayed with a default size and color. The default color for primitives is white, and the default point size is equal to the size of a single screen pixel.

We use the following OpenGL function to state the coordinate values for a single position:

```
glVertex* ( );
```

where the asterisk (\*) indicates that suffix codes are required for this function.

These suffix codes are used to identify the spatial dimension, the numerical datatype to be used for the coordinate values, and a possible vector form for the coordinate specification. Calls to **glVertex** functions must be placed between a **glBegin** function and a **glEnd** function. The argument of the **glBegin** function is used to identify the kind of output primitive that is to be displayed, and **glEnd** takes no arguments. For point plotting, the argument of the **glBegin** function is the symbolic constant **GL\_POINTS**. Thus, the form for an OpenGL specification of a point position is

```
glBegin (GL_POINTS);
```

```
glVertex* ( );
```

```
glEnd ( );
```

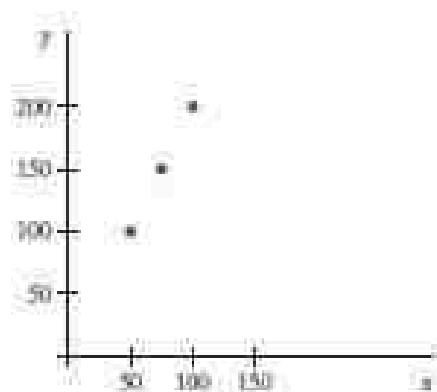
Although the term *vertex* strictly refers to a “corner” point of a polygon, the point of intersection of the sides of an angle, a point of intersection of an ellipse with its major axis, or other similar coordinate positions on geometric structures, the **glVertex** function is used in OpenGL to specify coordinates for any point position. In this way, a single function is used for point, line, and polygon specifications—and, most often, polygon patches are used to describe the objects in a scene.

Coordinate positions in OpenGL can be given in two, three, or four dimensions.

We use a suffix value of 2, 3, or 4 on the **glVertex** function to indicate the dimensionality of a coordinate position. A four-dimensional specification indicates a *homogeneous-coordinate* representation, where the *homogeneous parameter*  $h$  (the fourth coordinate) is a scaling factor for the Cartesian-coordinate values. Homogeneous-coordinate representations are useful for expressing transformation operations in matrix form. Because OpenGL treats two-dimensions as a special case of three dimensions, any  $(x, y)$  coordinate specification is equivalent to a three-dimensional specification of  $(x, y, 0)$ . Furthermore, OpenGL represents vertices internally in four dimensions, so each of these specifications are equivalent to the four-dimensional specification  $(x, y, 0, 1)$ .

We also need to state which data type is to be used for the numerical values specifications of the coordinates. This is accomplished with a second suffix code on the **glVertex** function. Suffix codes for specifying a numerical data type are **i** (integer), **s** (short), **f** (float), and **d** (double). Finally, the coordinate values can be listed explicitly in the **glVertex** function, or a single argument can be used that references a coordinate position as an

array. If we use an array specification for a coordinate position, we need to append **v** (for “vector”) as a third suffix code.



**FIGURE 3**  
Display of three point positions generated with  
`glBegin (GL_POINTS)`

In the following example, three equally spaced points are plotted along a two-dimensional, straight-line path with a slope of 2 (see Figure 3). Coordinates are given as integer pairs:

```
glBegin (GL_POINTS);
  glVertex2i (50, 100);
  glVertex2i (75, 150);
  glVertex2i (100, 200);
glEnd ( );
```

Alternatively, we could specify the coordinate values for the preceding points in arrays such as

```
int point1 [ ] = {50, 100};
int point2 [ ] = {75, 150};
int point3 [ ] = {100, 200};
```

and call the OpenGL functions for plotting the three points as

```
glBegin (GL_POINTS);
  glVertex2iv (point1);
  glVertex2iv (point2);
  glVertex2iv (point3);
glEnd ( );
```

In addition, here is an example of specifying two point positions in a three-dimensional world reference frame. In this case, we give the coordinates as explicit floating-point values:

```
glBegin (GL_POINTS);
  glVertex3f (-78.05, 909.72, 14.60);
  glVertex3f (261.91, -5200.67, 188.21);
glEnd ( );
```

We could also define a C++ class or structure (struct) for specifying point positions in various dimensions. For example,

```
class wPt3D {
public:
  GLfloat x, y;
};
```

Using this class definition, we could specify a two-dimensional, world-coordinate point position with the statements

```
wcPt2D pointPos;
pointPos.x = 120.75;
pointPos.y = 45.30;
glBegin (GL_POINTS);
glVertex2f (pointPos.x, pointPos.y);
glEnd ( );
```

Also, we can use the OpenGL point-plotting functions within a C++ procedure to implement the **setPixel** command.

### **OpenGL Line Functions**

Graphics packages typically provide a function for specifying one or more straight-line segments, where each line segment is defined by two endpoint coordinate positions. In OpenGL, we select a single endpoint coordinate position using the **glVertex** function, just as we did for a point position. And we enclose a list of **glVertex** functions between the **glBegin/glEnd** pair. But now we use a symbolic constant as the argument for the **glBegin** function that interprets a list of positions as the endpoint coordinates for line segments. There are three symbolic constants in OpenGL that we can use to specify how a list of endpoint positions should be connected to form a set of straight-line segments. By default, each symbolic constant displays solid, white lines.

A set of straight-line segments between each successive pair of endpoints in a list is generated using the primitive line constant **GL\_LINES**. In general, this will result in a set of unconnected lines unless some coordinate positions are repeated, because OpenGL considers lines to be connected only if they share a vertex; lines that cross but do not share a vertex are still considered to be unconnected. Nothing is displayed if only one endpoint is specified, and the last endpoint is not processed if the number of endpoints listed is odd. For example, if we have five coordinate positions, labeled **p1** through **p5**, and each is represented as a two-dimensional array, then the following code could generate the display shown in Figure 4(a):

```
glBegin (GL_LINES);
glVertex2iv (p1);
glVertex2iv (p2);
glVertex2iv (p3);
glVertex2iv (p4);
glVertex2iv (p5);
glEnd ( );
```

Thus, we obtain one line segment between the first and second coordinate positions and another line segment between the third and fourth positions. In this case, the number of specified endpoints is odd, so the last coordinate position is ignored.

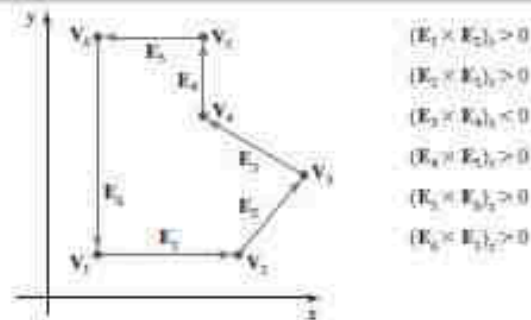
With the OpenGL primitive constant **GL\_LINE\_STRIP**, we obtain a **polyline**.

In this case, the display is a sequence of connected line segments between the first endpoint in the list and the last endpoint. The first line segment in the polyline is displayed between the first endpoint and the second endpoint; the second line segment is between the second and third



endpoints; and so forth, up to the last lineendpoint. Nothing is displayed if we do not list at least two coordinate positions.

**FIGURE 9**  
Identifying a concave polygon by calculating cross-products of successive pairs of edge vectors.



Another way to identify a concave polygon is to look at the polygon vertex positions relative to the extension line of any edge. If some vertices are on one side of the extension line and some vertices are on the other side, the polygon is concave.

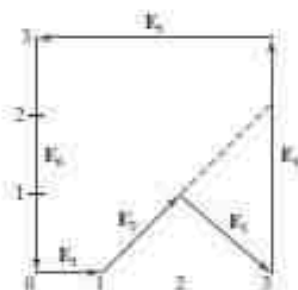
### Splitting Concave Polygons

Once we have identified a concave polygon, we can split it into a set of convex polygons. This can be accomplished using edge vectors and edge cross-products; or, we can use vertex positions relative to an edge extension line to determine which vertices are on one side of this line and which are on the other. For the following algorithms, we assume that all polygons are in the  $xy$  plane. Of course, the original position of a polygon described in world coordinates may not be in the  $xy$  plane, but we can always move it into that plane.

With the **vector method** for splitting a concave polygon, we first need to form the edge vectors. Given two consecutive vertex positions,  $V_k$  and  $V_{k+1}$ , we define the edge vector between them as

$$E_k = V_{k+1} - V_k$$

Next we calculate the cross-products of successive edge vectors in order around the polygon perimeter. If the  $z$  component of some cross-products is positive while other cross-products have a negative  $z$  component, the polygon is concave. Otherwise, the polygon is convex. This assumes that no series of three successive vertices are collinear, in which case the cross-product of the two edge vectors for these vertices would be zero. If all vertices are collinear, we have a degenerate polygon (a straight line). We can apply the vector method by processing edge vectors in counterclockwise order. If any cross-product has a negative  $z$  component (as in Figure 9), the polygon is concave and we can split it along the line of the first edge vector in the cross-product pair. The following example illustrates this method for splitting a concave polygon.

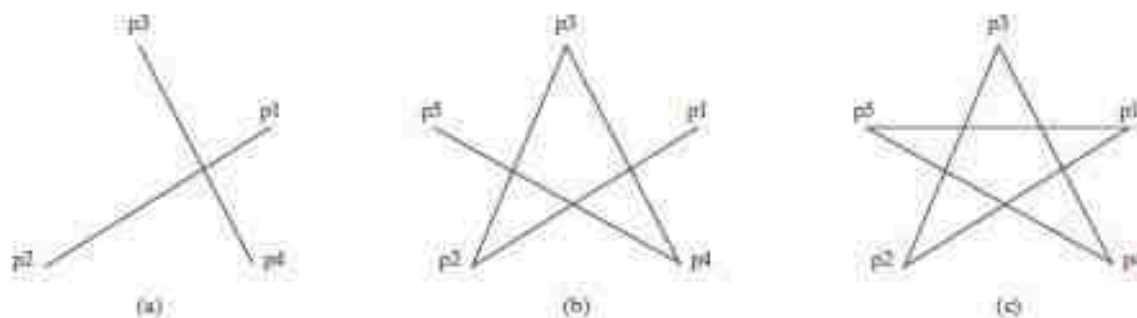


**FIGURE 10**  
Splitting a concave polygon using the vector method.

#### EXAMPLE 1 The Vector Method for Splitting Concave Polygons

Figure 10 shows a concave polygon with six edges. Edge vectors for this polygon can be expressed as

$$\begin{aligned} E_1 &= (1, 0, 0) & E_2 &= (1, 1, 0) \\ E_2 &= (1, -1, 0) & E_4 &= (0, 2, 0) \\ E_3 &= (-3, 0, 0) & E_6 &= (0, -2, 0) \end{aligned}$$



**FIGURE 4**

line segments that can be displayed in OpenGL using a list of five endpoint coordinates: (a) An unconnected set of lines generated with the primitive line constant `GL_LINES`; (b) A polyline generated with `GL_LINE_STRIP`; (c) A closed polyline generated with `GL_LINE_LOOP`.

Using the same five coordinate positions as in the previous example, we obtain the display in Figure 4(b) with the code

```
glBegin (GL_LINE_STRIP);
  glVertex2iv (p1);
  glVertex2iv (p2);
  glVertex2iv (p3);
  glVertex2iv (p4);
  glVertex2iv (p5);
glEnd ();
```

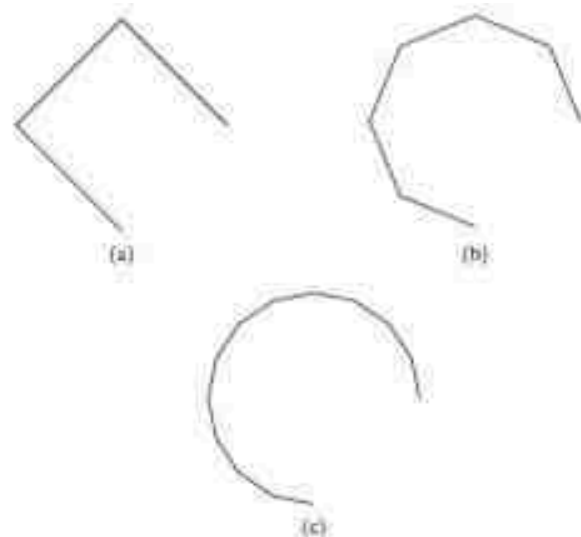
The third OpenGL line primitive is `GL_LINE_LOOP`, which produces a closed polyline. Lines are drawn as with `GL_LINE_STRIP`, but an additional line is drawn to connect the last coordinate position and the first coordinate position. Figure 4(c) shows the display of our endpoint list when we select this line option, using the code

```
glBegin (GL_LINE_LOOP);
  glVertex2iv (p1);
  glVertex2iv (p2);
  glVertex2iv (p3);
  glVertex2iv (p4);
  glVertex2iv (p5);
glEnd ();
```

As noted earlier, picture components are described in a world-coordinate reference frame that is eventually mapped to the coordinate reference for the output device. Then the geometric information about the picture is scan-converted to pixel positions.

## 5 OpenGL Curve Functions

Routines for generating basic curves, such as circles and ellipses, are not included as primitive functions in the OpenGL core library. But this library does contain functions for displaying Bézier splines, which are polynomials that are defined with a discrete point set. And the OpenGL Utility (GLU) library has routines for three-dimensional quadrics, such as spheres and cylinders, as well as



**FIGURE 5**  
A circular arc approximated with  
(a) three straight-line segments,  
(b) six line segments, and  
(c) twelve line segments.

routines for producing rational B-splines, which are a general class of splines that include the simpler Bézier curves. Using rational B-splines, we can display circles, ellipses, and other two-dimensional quadrics. In addition, there are routines in the OpenGL Utility Toolkit (GLUT) that we can use to display some three-dimensional quadrics, such as spheres and cones, and some other shapes. However, all these routines are more involved than the basic primitives we introduce in this chapter.

Another method we can use to generate a display of a simple curve is to approximate it using a polyline. We just need to locate a set of points along the curve path and connect the points with straight-line segments. The more line sections we include in the polyline, the smoother the appearance of the curve. As an example, Figure 5 illustrates various polyline displays that could be used for a circle segment.

A third alternative is to write our own curve-generation functions based on the algorithms presented in following chapters.

## 6 Fill-Area Primitives

Another useful construct, besides points, straight-line segments, and curves, for describing components of a picture is an area that is filled with some solid color or pattern. A picture component of this type is typically referred to as a **fill area** or a **filled area**. Most often, fill areas are used to describe surfaces of solid objects, but they are also useful in a variety of other applications. Also, fill regions are usually planar surfaces, mainly polygons. But, in general, there are many possible shapes for a region in a picture that we might wish to fill with a color option. Figure 6 illustrates a few possible fill-area shapes. For the present, we assume that all fill areas are to be displayed with a specified solid color.

Although any fill-area shape is possible, graphics libraries generally do not support specifications for arbitrary fill shapes. Most library routines require that



**FIGURE 6**  
Solid-color fill areas specified with various boundaries. (a) A circular fill region. (b) A fill area bounded by a closed polyline. (c) A filled area specified with an irregular curved boundary.



**FIGURE 7**  
Wire-frame representation for a cylinder, showing only the front (visible) faces of the polygon mesh used to approximate the surface.

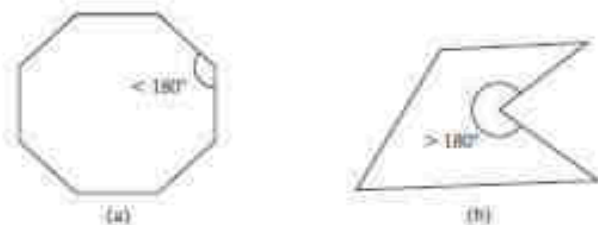
a fill area be specified as a polygon. Graphics routines can more efficiently process polygons than other kinds of fill shapes because polygon boundaries are described with linear equations. Moreover, most curved surfaces can be approximated reasonably well with a set of polygon patches, just as a curved line can be approximated with a set of straight-line segments. In addition, when lighting effects and surface-shading procedures are applied, an approximated curved surface can be displayed quite realistically. Approximating a curved surface with polygon facets is sometimes referred to as *surface tessellation*, or fitting the surface with a *polygon mesh*. Figure 7 shows the side and top surfaces of a metal cylinder approximated in an outline form as a polygon mesh. Displays of such figures can be generated quickly as *wire-frame* views, showing only the polygon edges to give a general indication of the surface structure. Then the wire-frame model could be shaded to generate a display of a natural-looking material surface. Objects described with a set of polygon surface patches are usually referred to as **standard graphics objects**, or just **graphics objects**.

In general, we can create fill areas with any boundary specification, such as a circle or connected set of spline-curve sections. And some of the polygon methods discussed in the next section can be adapted to display fill areas with a nonlinear border.

## 7 Polygon Fill Areas

Mathematically defined, a **polygon** is a plane figure specified by a set of three or more coordinate positions, called *vertices*, that are connected in sequence by straight-line segments, called the *edges* or *sides* of the polygon. Further, in basic geometry, it is required that the polygon edges have no common point other than their endpoints. Thus, by definition, a polygon must have all its vertices within a single plane and there can be no edge crossings. Examples of polygons include triangles, rectangles, octagons, and decagons. Sometimes, any plane figure with a closed-polyline boundary is alluded to as a polygon, and one with no crossing edges is referred to as a *standard polygon* or a *simple polygon*. In an effort to avoid ambiguous object references, we will use the term *polygon* to refer only to those planar shapes that have a closed-polyline boundary and no edge crossings.

For a computer-graphics application, it is possible that a designated set of polygon vertices do not all lie exactly in one plane. This can be due to round-off error in the calculation of numerical values, to errors in selecting coordinate positions for the vertices, or, more typically, to approximating a curved surface with a set of polygonal patches. One way to rectify this problem is simply to divide the specified surface mesh into triangles. But in some cases, there may be reasons



**FIGURE 8**  
A convex polygon (a), and a concave polygon (b).

to retain the original shape of the mesh patches, so methods have been devised for approximating a nonplanar polygonal shape with a plane figure. We discuss how these plane approximations are calculated in the section on plane equations.

### Polygon Classifications

An **interior angle** of a polygon is an angle inside the polygon boundary that is formed by two adjacent edges. If all interior angles of a polygon are less than or equal to  $180^\circ$ , the polygon is **convex**. An equivalent definition of a convex polygon is that its interior lies completely on one side of the infinite extension line of any one of its edges. Also, if we select any two points in the interior of a convex polygon, the line segment joining the two points is also in the interior. A polygon that is not convex is called a **concave** polygon. Figure 8 gives examples of convex and concave polygons.

The term **degenerate polygon** is often used to describe a set of vertices that are collinear or that have repeated coordinate positions. Collinear vertices generate a line segment. Repeated vertex positions can generate a polygon shape with extraneous lines, overlapping edges, or edges that have a length equal to 0. Sometimes the term degenerate polygon is also applied to a vertex list that contains fewer than three coordinate positions.

To be robust, a graphics package could reject degenerate or nonplanar vertex sets. But this requires extra processing to identify these problems, so graphics systems usually leave such considerations to the programmer.

Concave polygons also present problems. Implementations of fill algorithms and other graphics routines are more complicated for concave polygons, so it is generally more efficient to split a concave polygon into a set of convex polygons before processing. As with other polygon preprocessing algorithms, concave polygon splitting is often not included in a graphics library. Some graphics packages, including OpenGL, require all fill polygons to be convex. And some systems accept only triangular fill areas, which greatly simplifies many of the display algorithms.

### Identifying Concave Polygons

A concave polygon has at least one interior angle greater than  $180^\circ$ . Also, the extension of some edges of a concave polygon will intersect other edges, and some pair of interior points will produce a line segment that intersects the polygon boundary. Therefore, we can use any one of these characteristics of a concave polygon as a basis for constructing an identification algorithm.

If we set up a vector for each polygon edge, then we can use the cross-product of adjacent edges to test for concavity. All such vector products will be of the same sign (positive or negative) for a convex polygon. Therefore, if some cross-products yield a positive value and some a negative value, we have a concave polygon. Figure 9 illustrates the edge-vector, cross-product method for identifying concave polygons.

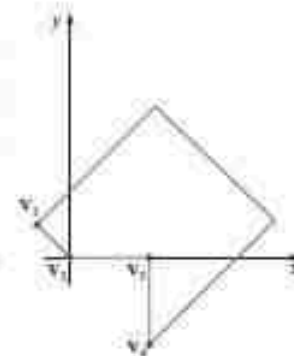


where the  $z$  component is 0, since all edges are in the  $xy$  plane. The cross-product  $\mathbf{E}_j \times \mathbf{E}_k$  for two successive edge vectors is a vector perpendicular to the  $xy$  plane with  $z$  component equal to  $E_{jx}E_{ky} - E_{kx}E_{jy}$ :

$$\begin{array}{ll} \mathbf{E}_1 \times \mathbf{E}_2 = (0, 0, 1) & \mathbf{E}_7 \times \mathbf{E}_8 = (0, 0, -2) \\ \mathbf{E}_2 \times \mathbf{E}_3 = (0, 0, 2) & \mathbf{E}_4 \times \mathbf{E}_5 = (0, 0, 6) \\ \mathbf{E}_5 \times \mathbf{E}_6 = (0, 0, 6) & \mathbf{E}_6 \times \mathbf{E}_7 = (0, 0, 2) \end{array}$$

Since the cross-product  $\mathbf{E}_2 \times \mathbf{E}_3$  has a negative  $z$  component, we split the polygon along the line of vector  $\mathbf{E}_2$ . The line equation for this edge has a slope of 1 and a  $y$  intercept of  $-1$ . We then determine the intersection of this line with the other polygon edges to split the polygon into two pieces. No other edge cross-products are negative, so the two new polygons are both convex.

We can also split a concave polygon using a **rotational method**. Proceeding counterclockwise around the polygon edges, we shift the position of the polygon so that each vertex  $\mathbf{V}_k$  in turn is at the coordinate origin. Then, we rotate the polygon about the origin in a clockwise direction so that the next vertex  $\mathbf{V}_{k+1}$  is on the  $x$  axis. If the following vertex,  $\mathbf{V}_{k+2}$ , is below the  $x$  axis, the polygon is concave. We then split the polygon along the  $x$  axis to form two new polygons, and we repeat the concave test for each of the two new polygons. These steps are repeated until we have tested all vertices in the polygon list. Figure 11 illustrates the rotational method for splitting a concave polygon.



**FIGURE 11** Splitting a concave polygon using the rotational method. After moving  $\mathbf{V}_2$  to the coordinate origin and rotating  $\mathbf{V}_3$  onto the  $x$  axis, we find that  $\mathbf{V}_4$  is below the  $x$  axis. So we split the polygon along the line of  $\mathbf{V}_2/\mathbf{V}_3$ , which is the  $x$  axis.

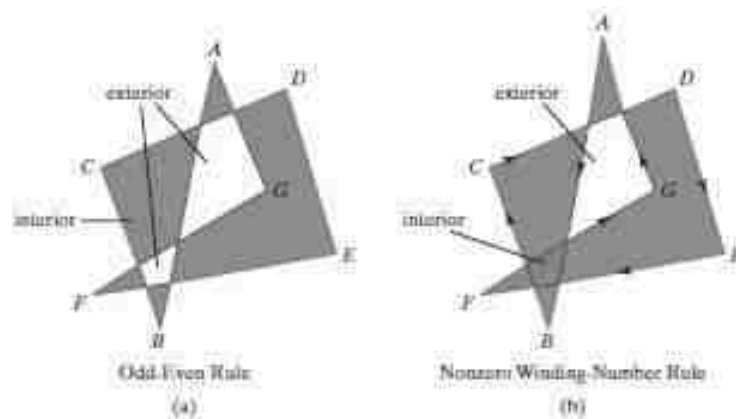
### Splitting a Convex Polygon into a Set of Triangles

Once we have a vertex list for a convex polygon, we could transform it into a set of triangles. This can be accomplished by first defining any sequence of three consecutive vertices to be a new polygon (a triangle). The middle triangle vertex is then deleted from the original vertex list. Then the same procedure is applied to this modified vertex list to strip off another triangle. We continue forming triangles in this manner until the original polygon is reduced to just three vertices, which define the last triangle in the set. A concave polygon can also be divided into a set of triangles using this approach, although care must be taken that the new diagonal edge formed by joining the first and third selected vertices does not cross the concave portion of the polygon, and that the three selected vertices at each step form an interior angle that is less than  $180^\circ$  (a "convex" angle).

### Inside-Outside Tests

Various graphics processes often need to identify interior regions of objects. Identifying the interior of a simple object, such as a convex polygon, a circle, or a sphere, is generally a straightforward process. But sometimes we must deal with more complex objects. For example, we may want to specify a complex fill region with intersecting edges, as in Figure 12. For such shapes, it is not always clear which regions of the  $xy$  plane we should call "interior" and which regions we should designate as "exterior" to the object boundaries. Two commonly used algorithms for identifying interior areas of a plane figure are the odd-even rule and the nonzero winding-number rule.

We apply the **odd-even rule**, also called the *odd-parity rule* or the *even-odd rule*, by first conceptually drawing a line from any position  $\mathbf{P}$  to a distant point



**FIGURE 12**  
Identifying interior and exterior regions  
of a closed polyline that contains  
self-intersecting segments.

outside the coordinate extents of the closed polyline. Then we count the number of line-segment crossings along this line. If the number of segments crossed by this line is odd, then  $P$  is considered to be an *interior* point. Otherwise,  $P$  is an *exterior* point. To obtain an accurate count of the segment crossings, we must be sure that the line path we choose does not intersect any line-segment endpoints. Figure 12(a) shows the interior and exterior regions obtained using the odd-even rule for a self-intersecting closed polyline. We can use this procedure, for example, to fill the interior region between two concentric circles or two concentric polygons with a specified color.

Another method for defining interior regions is the **nonzero winding-number rule**, which counts the number of times that the boundary of an object “winds” around a particular point in the counterclockwise direction. This count is called the **winding number**, and the interior points of a two-dimensional object can be defined to be those that have a nonzero value for the winding number. We apply the nonzero winding number rule by initializing the winding number to 0 and again imagining a line drawn from any position  $P$  to a distant point beyond the coordinate extents of the object. The line we choose must not pass through any endpoint coordinates. As we move along the line from position  $P$  to the distant point, we count the number of object line segments that cross the reference line in each direction. We add 1 to the winding number every time we intersect a segment that crosses the line in the direction from right to left, and we subtract 1 every time we intersect a segment that crosses from left to right. The final value of the winding number, after all boundary crossings have been counted, determines the relative position of  $P$ . If the winding number is nonzero,  $P$  is considered to be an interior point. Otherwise,  $P$  is taken to be an exterior point. Figure 12(b) shows the interior and exterior regions defined by the nonzero winding-number rule for a self-intersecting, closed polyline. For simple objects, such as polygons and circles, the nonzero winding-number rule and the odd-even rule give the same results. But for more complex shapes, the two methods may yield different interior and exterior regions, as in the example of Figure 12.

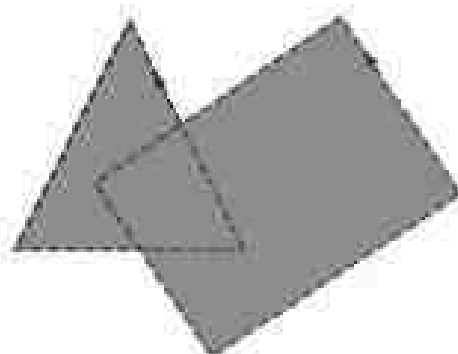
One way to determine directional boundary crossings is to set up vectors along the object edges (or boundary lines) and along the reference line. Then we compute the vector cross-product of the vector  $\mathbf{u}$ , along the line from  $P$  to a distant point, with an object edge vector  $\mathbf{E}$  for each edge that crosses the line. Assuming that we have a two-dimensional object in the  $xy$  plane, the direction of each vector cross-product will be either in the  $+z$  direction or in the  $-z$  direction. If the  $z$  component of a cross-product  $\mathbf{u} \times \mathbf{E}$  for a particular crossing is positive, that segment crosses from right to left and we add 1 to the winding number.

Otherwise, the segment crosses from left to right and we subtract 1 from the winding number.

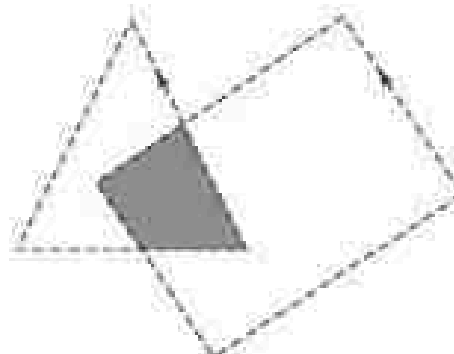
A somewhat simpler way to compute directional boundary crossings is to use vector dot products instead of cross-products. To do this, we set up a vector that is perpendicular to vector  $\mathbf{u}$  and that has a right-to-left direction as we look along the line from  $\mathbf{P}$  in the direction of  $\mathbf{u}$ . If the components of  $\mathbf{u}$  are denoted as  $(u_x, u_y)$ , then the vector that is perpendicular to  $\mathbf{u}$  has components  $(-u_y, u_x)$ . Now, if the dot product of this perpendicular vector and a boundary-line vector is positive, that crossing is from right to left and we add 1 to the winding number. Otherwise, the boundary crosses our reference line from left to right, and we subtract 1 from the winding number.

The nonzero winding-number rule tends to classify as interior some areas that the odd-even rule deems to be exterior, and it can be more versatile in some applications. In general, plane figures can be defined with multiple, disjoint components, and the direction specified for each set of disjoint boundaries can be used to designate the interior and exterior regions. Examples include characters (such as letters of the alphabet and punctuation symbols), nested polygons, and concentric circles or ellipses. For curved lines, the odd-even rule is applied by calculating intersections with the curve paths. Similarly, with the nonzero winding-number rule, we need to calculate tangent vectors to the curves at the crossover intersection points with the reference line from position  $\mathbf{P}$ .

Variations of the nonzero winding-number rule can be used to define interior regions in other ways. For example, we could define a point to be interior if its winding number is positive or if it is negative; or we could use any other rule to generate a variety of fill shapes. Sometimes, Boolean operations are used to specify a fill area as a combination of two regions. One way to implement Boolean operations is by using a variation of the basic winding-number rule. With this scheme, we first define a simple, nonintersecting boundary for each of two regions. Then if we consider the direction for each boundary to be counterclockwise, the union of two regions would consist of those points whose winding number is positive (Figure 13). Similarly, the intersection of two regions with counterclockwise boundaries would contain those points whose winding number is greater than 1, as illustrated in Figure 14. To set up a fill area that is the difference of two regions (say,  $A - B$ ), we can enclose region  $A$  with a counterclockwise border and  $B$  with a clockwise border. Then the difference region (Figure 15) is the set of all points whose winding number is positive.

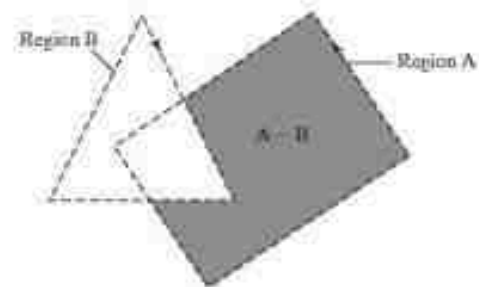


**FIGURE 13**  
A fill area defined as a region that has a positive value for the winding number. This fill area is the union of two regions, each with a counterclockwise border direction.



**FIGURE 14**  
A fill area defined as a region with a winding number greater than 1. This fill area is the intersection of two regions, each with a counterclockwise border direction.



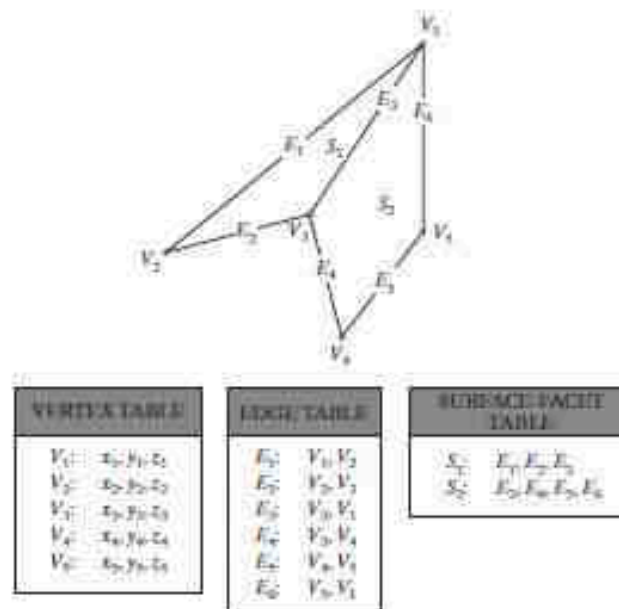


**FIGURE 15**  
A fill area defined as a region with a positive value for the winding number. This fill area is the difference,  $A - B$ , of two regions, where region A has a positive border direction (counterclockwise) and region B has a negative border direction (clockwise).

### Polygon Tables

Typically, the objects in a scene are described as sets of polygon surface facets. In fact, graphics packages often provide functions for defining a surface shape as a mesh of polygon patches. The description for each object includes coordinate information specifying the geometry for the polygon facets and other surface parameters such as color, transparency, and light-reflection properties. As information for each polygon is input, the data are placed into tables that are to be used in the subsequent processing, display, and manipulation of the objects in the scene. These polygon data tables can be organized into two groups: geometric tables and attribute tables. Geometric data tables contain vertex coordinates and parameters to identify the spatial orientation of the polygon surfaces. Attribute information for an object includes parameters specifying the degree of transparency of the object and its surface reflectivity and texture characteristics.

Geometric data for the objects in a scene are arranged conveniently in three lists: a vertex table, an edge table, and a surface-facet table. Coordinate values for each vertex in the object are stored in the vertex table. The edge table contains pointers back into the vertex table to identify the vertices for each polygon edge. And the surface-facet table contains pointers back into the edge table to identify the edges for each polygon. This scheme is illustrated in Figure 16 for two



**FIGURE 16**  
Geometric data-table representation for two adjacent polygon surface facets, formed with six edges and five vertices.



adjacent polygon facets on an object surface. In addition, individual objects and their component polygon faces can be assigned object and facet identifiers for easy reference.

Listing the geometric data in three tables, as in Figure 16, provides a convenient reference to the individual components (vertices, edges, and surface facets) for each object. Also, the object can be displayed efficiently by using data from the edge table to identify polygon boundaries. An alternative arrangement is to use just two tables: a vertex table and a surface-facet table. But this scheme is less convenient, and some edges could get drawn twice in a wire-frame display. Another possibility is to use only a surface-facet table, but this duplicates coordinate information, since explicit coordinate values are listed for each vertex in each polygon facet. Also the relationship between edges and facets would have to be reconstructed from the vertex listings in the surface-facet table.

We can add extra information to the data tables of Figure 16 for faster information extraction. For instance, we could expand the edge table to include forward pointers into the surface-facet table so that a common edge between polygons could be identified more rapidly (Figure 17). This is particularly useful for rendering procedures that must vary surface shading smoothly across the edges from one polygon to the next. Similarly, the vertex table could be expanded to reference corresponding edges, for faster information retrieval.

Additional geometric information that is usually stored in the data tables includes the slope for each edge and the coordinate extents for polygon edges, polygon facets, and each object in a scene. As vertices are input, we can calculate edge slopes, and we can scan the coordinate values to identify the minimum and maximum  $x$ ,  $y$ , and  $z$  values for individual lines and polygons. Edge slopes and bounding-box information are needed in subsequent processing, such as surface rendering and visible-surface identification algorithms.

Because the geometric data tables may contain extensive listings of vertices and edges for complex objects and scenes, it is important that the data be checked for consistency and completeness. When vertex, edge, and polygon definitions are specified, it is possible, particularly in interactive applications, that certain input errors could be made that would distort the display of the objects. The more information included in the data tables, the easier it is to check for errors. Therefore, error checking is easier when three data tables (vertex, edge, and surface facet) are used, since this scheme provides the most information. Some of the tests that could be performed by a graphics package are (1) that every vertex is listed as an endpoint for at least two edges, (2) that every edge is part of at least one polygon, (3) that every polygon is closed, (4) that each polygon has at least one shared edge, and (5) that if the edge table contains pointers to polygons, every edge referenced by a polygon pointer has a reciprocal pointer back to the polygon.

### Plane Equations

To produce a display of a three-dimensional scene, a graphics system processes the input data through several procedures. These procedures include transformation of the modeling and world-coordinate descriptions through the viewing pipeline, identification of visible surfaces, and the application of rendering routines to the individual surface facets. For some of these processes, information about the spatial orientation of the surface components of objects is needed. This information is obtained from the vertex coordinate values and the equations that describe the polygon surfaces.

$E_1$	$V_1, V_2, S_1$
$E_2$	$V_2, V_3, S_1$
$E_3$	$V_3, V_1, S_1, S_2$
$E_4$	$V_3, V_4, S_2$
$E_5$	$V_4, V_5, S_2$
$E_6$	$V_5, V_1, S_2$

FIGURE 17  
Edge table for the surfaces of Figure 16 expanded to include pointers into the surface-facet table.

Each polygon in a scene is contained within a plane of infinite extent. The general equation of a plane is

$$Ax + By + Cz + D = 0 \quad (1)$$

where  $(x, y, z)$  is any point on the plane, and the coefficients  $A$ ,  $B$ ,  $C$ , and  $D$  (called *plane parameters*) are constants describing the spatial properties of the plane. We can obtain the values of  $A$ ,  $B$ ,  $C$ , and  $D$  by solving a set of three plane equations using the coordinate values for three noncollinear points in the plane. For this purpose, we can select three successive convex-polygon vertices,  $(x_1, y_1, z_1)$ ,  $(x_2, y_2, z_2)$ , and  $(x_3, y_3, z_3)$ , in a counterclockwise order and solve the following set of simultaneous linear plane equations for the ratios  $A/D$ ,  $B/D$ , and  $C/D$ :

$$(A/D)x_k + (B/D)y_k + (C/D)z_k = -1, \quad k = 1, 2, 3 \quad (2)$$

The solution to this set of equations can be obtained in determinant form, using Cramer's rule, as

$$\begin{aligned} A &= \begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{vmatrix} & B &= \begin{vmatrix} x_1 & 1 & z_1 \\ x_2 & 1 & z_2 \\ x_3 & 1 & z_3 \end{vmatrix} \\ C &= \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} & D &= - \begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix} \end{aligned} \quad (3)$$

Expanding the determinants, we can write the calculations for the plane coefficients in the form

$$\begin{aligned} A &= y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2) \\ B &= z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2) \\ C &= x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2) \\ D &= -x_1(y_2z_3 - y_3z_2) - x_2(y_3z_1 - y_1z_3) - x_3(y_1z_2 - y_2z_1) \end{aligned} \quad (4)$$

These calculations are valid for any three coordinate positions, including those for which  $D = 0$ . When vertex coordinates and other information are entered into the polygon data structure, values for  $A$ ,  $B$ ,  $C$ , and  $D$  can be computed for each polygon facet and stored with the other polygon data.

It is possible that the coordinates defining a polygon facet may not be contained within a single plane. We can solve this problem by dividing the facet into a set of triangles; or we could find an approximating plane for the vertex list. One method for obtaining an approximating plane is to divide the vertex list into subsets, where each subset contains three vertices, and calculate plane parameters  $A$ ,  $B$ ,  $C$ ,  $D$  for each subset. The approximating plane parameters are then obtained as the average value for each of the calculated plane parameters. Another approach is to project the vertex list onto the coordinate planes. Then we take parameter  $A$  proportional to the area of the polygon projection on the  $yz$  plane, parameter  $B$  proportional to the projection area on the  $xz$  plane, and parameter  $C$  proportional to the projection area on the  $xy$  plane. The projection method is often used in ray-tracing applications.

### Front and Back Polygon Faces

Because we are usually dealing with polygon surfaces that enclose an object interior, we need to distinguish between the two sides of each surface. The side of a polygon that faces into the object interior is called the *back face*, and the visible, or outward, side is the *front face*. Identifying the position of points in space

relative to the front and back faces of a polygon is a basic task in many graphics algorithms, as, for example, in determining object visibility. Every polygon is contained within an infinite plane that partitions space into two regions. Any point that is not on the plane and that is visible to the front face of a polygon surface section is said to be *in front of* (or *outside*) the plane, and, thus, outside the object. And any point that is visible to the back face of the polygon is *behind* (or *inside*) the plane. A point that is behind (inside) all polygon surface planes is inside the object. We need to keep in mind that this inside/outside classification is relative to the plane containing the polygon, whereas our previous inside/outside tests using the winding-number or odd-even rule were in reference to the interior of some two-dimensional boundary.

Plane equations can be used to identify the position of spatial points relative to the polygon facets of an object. For any point  $(x, y, z)$  not on a plane with parameters  $A, B, C, D$ , we have:

$$Ax + By + Cz + D \neq 0$$

Thus, we can identify the point as either behind or in front of a polygon surface contained within that plane according to the sign (negative or positive) of  $Ax + By + Cz + D$ :

- if  $Ax + By + Cz + D < 0$ , the point  $(x, y, z)$  is behind the plane  
 if  $Ax + By + Cz + D > 0$ , the point  $(x, y, z)$  is in front of the plane

These inequality tests are valid in a right-handed Cartesian system, provided the plane parameters  $A, B, C$ , and  $D$  were calculated using coordinate positions selected in a strictly counterclockwise order when viewing the surface along a front-to-back direction. For example, in Figure 18, any point outside (in front of) the plane of the shaded polygon satisfies the inequality  $x - 1 > 0$ , while any point inside (in back of) the plane has an  $x$ -coordinate value less than 1.

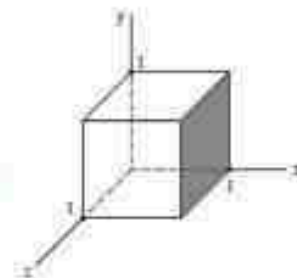
Orientation of a polygon surface in space can be described with the **normal vector** for the plane containing that polygon, as shown in Figure 19. This surface normal vector is perpendicular to the plane and has Cartesian components  $(A, B, C)$ , where parameters  $A, B$ , and  $C$  are the plane coefficients calculated in Equations 4. The normal vector points in a direction from inside the plane to the outside; that is, from the back face of the polygon to the front face.

As an example of calculating the components of the normal vector for a polygon, which also gives us the plane parameters, we choose three of the vertices of the shaded face of the unit cube in Figure 18. These points are selected in a counterclockwise ordering as we view the cube from outside looking toward the origin. Coordinates for these vertices, in the order selected, are then used in Equations 4 to obtain the plane coefficients:  $A = 1, B = 0, C = 0, D = -1$ . Thus, the normal vector for this plane is  $\mathbf{N} = (1, 0, 0)$ , which is in the direction of the positive  $x$  axis. That is, the normal vector is pointing from inside the cube to the outside and is perpendicular to the plane  $x = 1$ .

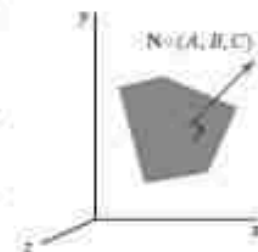
The elements of a normal vector can also be obtained using a vector cross-product calculation. Assuming we have a convex-polygon surface facet and a right-handed Cartesian system, we again select any three vertex positions,  $\mathbf{V}_1, \mathbf{V}_2$ , and  $\mathbf{V}_3$ , taken in counterclockwise order when viewing from outside the object toward the inside. Forming two vectors, one from  $\mathbf{V}_1$  to  $\mathbf{V}_2$  and the second from  $\mathbf{V}_1$  to  $\mathbf{V}_3$ , we calculate  $\mathbf{N}$  as the vector cross-product:

$$\mathbf{N} = (\mathbf{V}_2 - \mathbf{V}_1) \times (\mathbf{V}_3 - \mathbf{V}_1) \quad (5)$$

This generates values for the plane parameters  $A, B$ , and  $C$ . We can then obtain the value for parameter  $D$  by substituting these values and the coordinates for one of



**FIGURE 18**  
The shaded polygon surface of the unit cube has the plane equation  $x - 1 = 0$ .



**FIGURE 19**  
The normal vector  $\mathbf{N}$  for a plane described with the equation  $Ax + By + Cz + D = 0$  is perpendicular to the plane and has Cartesian components  $(A, B, C)$ .



the polygon vertices into Equation 1 and solving for  $D$ . The plane equation can be expressed in vector form using the normal  $\mathbf{N}$  and the position  $\mathbf{P}$  of any point in the plane as

$$\mathbf{N} \cdot \mathbf{P} = -D \quad (6)$$

For a convex polygon, we could also obtain the plane parameters using the cross-product of two successive edge vectors. And with a concave polygon, we can select the three vertices so that the two vectors for the cross-product form an angle less than  $180^\circ$ . Otherwise, we can take the negative of their cross-product to get the correct normal vector direction for the polygon surface.

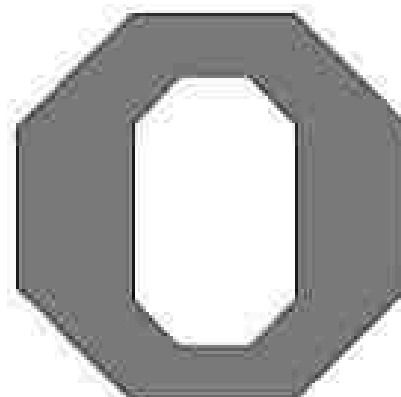
## 8 OpenGL Polygon Fill-Area Functions

With one exception, the OpenGL procedures for specifying fill polygons are similar to those for describing a point or a polyline. A `glVertex` function is used to input the coordinates for a single polygon vertex, and a complete polygon is described with a list of vertices placed between a `glBegin`/`glEnd` pair. However, there is one additional function that we can use for displaying a rectangle that has an entirely different format.

By default, a polygon interior is displayed in a solid color, determined by the current color settings. As options, we can fill a polygon with a pattern and we can display polygon edges as line borders around the interior fill. There are six different symbolic constants that we can use as the argument in the `glBegin` function to describe polygon fill areas. These six primitive constants allow us to display a single fill polygon, a set of unconnected fill polygons, or a set of connected fill polygons.

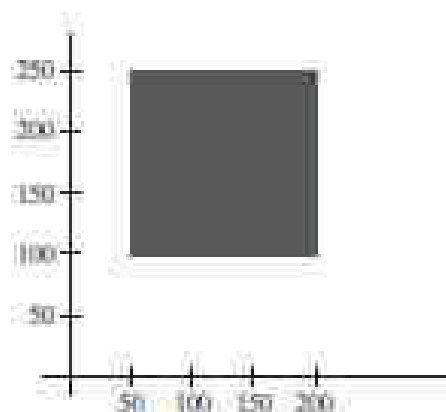
In OpenGL, a fill area must be specified as a convex polygon. Thus, a vertex list for a fill polygon must contain at least three vertices, there can be no crossing edges, and all interior angles for the polygon must be less than  $180^\circ$ . And a single polygon fill area can be defined with only one vertex list, which precludes any specifications that contain holes in the polygon interior, such as that shown in Figure 20. We could describe such a figure using two overlapping convex polygons.

Each polygon that we specify has two faces: a back face and a front face. In OpenGL, fill color and other attributes can be set for each face separately, and back/front identification is needed in both two-dimensional and three-dimensional viewing routines. Therefore, polygon vertices should be specified in a counterclockwise order as we view the polygon from "outside." This identifies the front face of that polygon.



**FIGURE 20**  
A polygon with a complex interior that cannot be specified with a single vertex list.





**FIGURE 21**  
The display of a square fill area using the `glRect` function.

Because graphics displays often include rectangular fill areas, OpenGL provides a special rectangle function that directly accepts vertex specifications in the  $xy$  plane. In some implementations of OpenGL, the following routine can be more efficient than generating a fill rectangle using `glVertex` specifications:

```
glRectf (x1, y1, x2, y2);
```

One corner of this rectangle is at coordinate position  $(x1, y1)$ , and the opposite corner of the rectangle is at position  $(x2, y2)$ . Suffix codes for `glRect` specify the coordinate data type and whether coordinates are to be expressed as array elements. These codes are `i` (for integer), `s` (for short), `f` (for float), `d` (for double), and `v` (for vector). The rectangle is displayed with edges parallel to the  $xy$  coordinate axes. As an example, the following statement defines the square shown in Figure 21:

```
glRecti (200, 100, 50, 250);
```

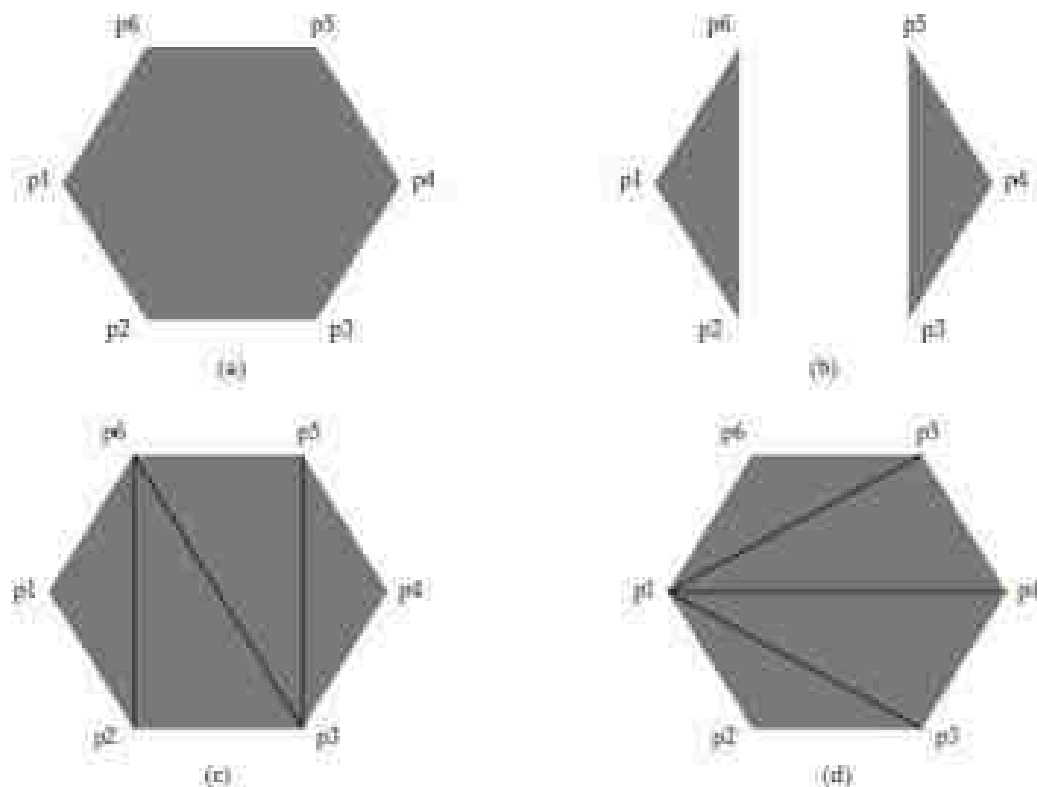
If we put the coordinate values for this rectangle into arrays, we can generate the same square with the following code:

```
int vertex1 [ ] = {200, 100};
int vertex2 [ ] = {50, 250};

glRectiv (vertex1, vertex2);
```

When a rectangle is generated with function `glRect`, the polygon edges are formed between the vertices in the order  $(x1, y1)$ ,  $(x2, y1)$ ,  $(x2, y2)$ ,  $(x1, y2)$ , and then back to  $(x1, y1)$ . Thus, in our example, we produced a vertex list with a clockwise ordering. In many two-dimensional applications, the determination of front and back faces is unimportant. But if we do want to assign different properties to the front and back faces of the rectangle, then we should reverse the order of the two vertices in this example so that we obtain a counterclockwise ordering of the vertices.

Each of the other six OpenGL polygon fill primitives is specified with a symbolic constant in the `glBegin` function, along with a list of `glVertex` commands. With the OpenGL primitive constant `GL_POLYGON`, we can display a single polygon fill area such as that shown in Figure 22(a). For this example, we assume that we have a list of six points, labeled `p1` through `p6`, specifying



**FIGURE 22**  
 Displaying polygon fill areas using a list of six vertex positions. (a) A single convex polygon fill area generated with the primitive constant `GL_POLYGON`. (b) Two unconnected triangles generated with `GL_TRIANGLES`. (c) Four connected triangles generated with `GL_TRIANGLE_STRIP`. (d) Four connected triangles generated with `GL_TRIANGLE_FAN`.

two-dimensional polygon vertex positions in a counterclockwise ordering. Each of the points is represented as an array of  $(x, y)$  coordinate values:

```
glBegin (GL_POLYGON);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p3);
    glVertex2iv (p4);
    glVertex2iv (p5);
    glVertex2iv (p6);
glEnd ( );
```

A polygon vertex list must contain at least three vertices. Otherwise, nothing is displayed.

If we reorder the vertex list and change the primitive constant in the previous code example to `GL_TRIANGLES`, we obtain the two separated triangle fill areas in Figure 22(b):

```
glBegin (GL_TRIANGLES);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p6);
    glVertex2iv (p3);
    glVertex2iv (p4);
    glVertex2iv (p5);
glEnd ( );
```

In this case, the first three coordinate points define the vertices for one triangle, the next three points define the next triangle, and so forth. For each triangle fill area, we specify the vertex positions in a counterclockwise order. A set of unconnected triangles is displayed with this primitive constant unless some vertex coordinates are repeated. Nothing is displayed if we do not list at least three vertices, and if the number of vertices specified is not a multiple of 3, the final one or two vertex positions are not used.

By reordering the vertex list once more and changing the primitive constant to `GL_TRIANGLE_STRIP`, we can display the set of connected triangles shown in Figure 22(c).

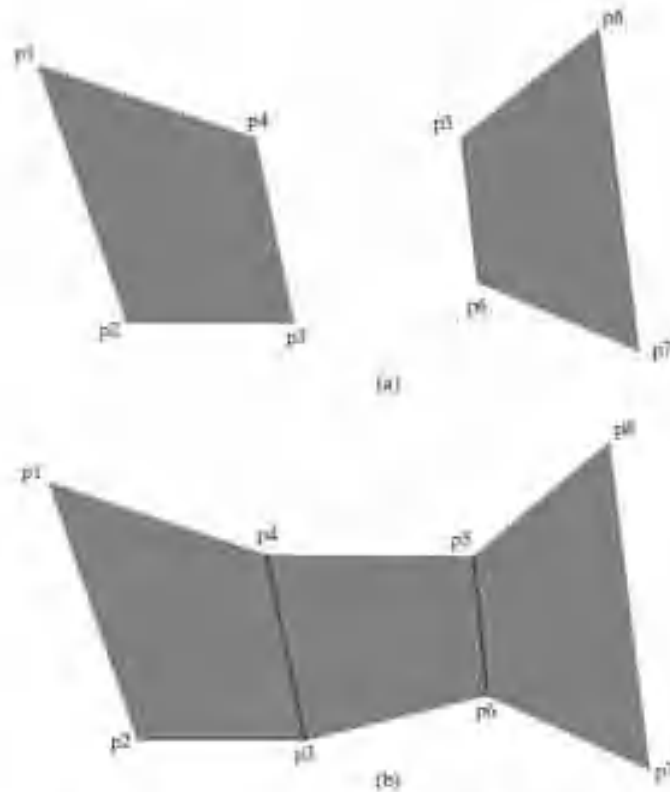
```
glBegin (GL_TRIANGLE_STRIP);
    glVertex3iv (p1);
    glVertex3iv (p2);
    glVertex3iv (p6);
    glVertex3iv (p3);
    glVertex3iv (p5);
    glVertex3iv (p4);
glEnd ();
```

Assuming that no coordinate positions are repeated in a list of  $N$  vertices, we obtain  $N - 2$  triangles in the strip. Clearly, we must have  $N \geq 3$  or nothing is displayed. In this example,  $N = 6$  and we obtain four triangles. Each successive triangle shares an edge with the previously defined display triangle, so the ordering of the vertex list must be set up to ensure a consistent display. One triangle is defined for each vertex position listed after the first two vertices. Thus, the first three vertices should be listed in counterclockwise order, when viewing the front (outside) surface of the triangle. After that, the set of three vertices for each subsequent triangle is arranged in a counterclockwise order within the polygon table. This is accomplished by processing each position  $n$  in the vertex list in the order  $n = 1, n = 2, \dots, n = N - 2$  and arranging the order of the corresponding set of three vertices according to whether  $n$  is an odd number or an even number. If  $n$  is odd, the polygon table listing for the triangle vertices is in the order  $n, n + 1, n + 2$ . If  $n$  is even, the triangle vertices are listed in the order  $n + 1, n, n + 2$ . In the preceding example, our first triangle ( $n = 1$ ) would be listed as having vertices  $(p1, p2, p6)$ . The second triangle ( $n = 2$ ) would have the vertex ordering  $(p6, p2, p3)$ . Vertex ordering for the third triangle ( $n = 3$ ) would be  $(p6, p3, p5)$ . And the fourth triangle ( $n = 4$ ) would be listed in the polygon tables with vertex ordering  $(p5, p3, p4)$ .

Another way to generate a set of connected triangles is to use the "fan" approach illustrated in Figure 22(d), where all triangles share a common vertex. We obtain this arrangement of triangles using the primitive constant `GL_TRIANGLE_FAN` and the original ordering of our six vertices:

```
glBegin (GL_TRIANGLE_FAN);
    glVertex3iv (p1);
    glVertex3iv (p2);
    glVertex3iv (p3);
    glVertex3iv (p4);
    glVertex3iv (p5);
    glVertex3iv (p6);
glEnd ();
```

For  $N$  vertices, we again obtain  $N - 2$  triangles, providing no vertex positions are repeated, and we must list at least three vertices. In addition, the vertices must



**FIGURE 23**  
 Displaying quadrilateral meshes using a list of eight vertex positions. (a) Two unconnected quadrilaterals generated with `GL_QUADS`. (b) Three connected quadrilaterals generated with `GL_QUAD_STRIP`.

be specified in the proper order to define front and back faces for each triangle correctly. The first coordinate position listed (in this case,  $p_1$ ) is a vertex for each triangle in the fan. If we again enumerate the triangles and the coordinate positions listed as  $n = 1, n = 2, \dots, n = N - 2$ , then vertices for triangle  $n$  are listed in the polygon tables in the order  $1, n + 1, n + 2$ . Therefore, triangle 1 is defined with the vertex list  $(p_1, p_2, p_3)$ ; triangle 2 has the vertex ordering  $(p_1, p_3, p_4)$ ; triangle 3 has its vertices specified in the order  $(p_1, p_4, p_5)$ ; and triangle 4 is listed with vertices  $(p_1, p_5, p_6)$ .

Besides the primitive functions for triangles and a general polygon, OpenGL provides for the specifications of two types of quadrilaterals (four-sided polygons). With the `GL_QUADS` primitive constant and the following list of eight vertices, specified as two-dimensional coordinate arrays, we can generate the display shown in Figure 23(a):

```
glBegin (GL_QUADS);
  glVertex2iv (p1);
  glVertex2iv (p2);
  glVertex2iv (p3);
  glVertex2iv (p4);
  glVertex2iv (p5);
  glVertex2iv (p6);
  glVertex2iv (p7);
  glVertex2iv (p8);
glEnd ( );
```



The first four coordinate points define the vertices for one quadrilateral, the next four points define the next quadrilateral, and so on. For each quadrilateral fill area, we specify the vertex positions in a counterclockwise order. If no vertex coordinates are repeated, we display a set of unconnected four-sided fill areas. We must list at least four vertices with this primitive. Otherwise, nothing is displayed. And if the number of vertices specified is not a multiple of 4, the extra vertex positions are ignored.

Rearranging the vertex list in the previous quadrilateral code example and changing the primitive constant to `GL_QUAD_STRIP`, we can obtain the set of connected quadrilaterals shown in Figure 23(b):

```
glBegin (GL_QUAD_STRIP);
    glVertex3f (p1);
    glVertex3f (p2);
    glVertex3f (p4);
    glVertex3f (p3);
    glVertex3f (p5);
    glVertex3f (p6);
    glVertex3f (p8);
    glVertex3f (p7);
glEnd ( );
```

A quadrilateral is set up for each pair of vertices specified after the first two vertices in the list, and we need to list the vertices so that we generate a correct counterclockwise vertex ordering for each polygon. For a list of  $N$  vertices, we obtain  $\frac{N}{2} - 1$  quadrilaterals, providing that  $N \geq 4$ . If  $N$  is not a multiple of 4, any extra coordinate positions in the list are not used. We can enumerate these fill polygons and the vertices listed as  $n = 1, n = 2, \dots, n = \frac{N}{2} - 1$ . Then polygon tables will list the vertices for quadrilateral  $n$  in the vertex order number  $2n - 1, 2n, 2n + 2, 2n + 1$ . For this example,  $N = 8$  and we have 3 quadrilaterals in the strip. Thus, our first quadrilateral ( $n = 1$ ) is listed as having a vertex ordering of (p1, p2, p3, p4). The second quadrilateral ( $n = 2$ ) has the vertex ordering (p4, p3, p6, p5), and the vertex ordering for the third quadrilateral ( $n = 3$ ) is (p5, p6, p7, p8).

Most graphics packages display curved surfaces as a set of approximating plane facets. This is because plane equations are linear, and processing the linear equations is much quicker than processing quadric or other types of curve equations. So OpenGL and other packages provide polygon primitives to facilitate the approximation of a curved surface. Objects are modeled with polygon meshes, and a database of geometric and attribute information is set up to facilitate the processing of the polygon facets. In OpenGL, primitives that we can use for this purpose are the *triangle strip*, the *triangle fan*, and the *quad strip*. Fast hardware-implemented polygon renderers are incorporated into high-quality graphics systems with the capability for displaying millions of shaded polygons per second (usually triangles), including the application of surface texture and special lighting effects.

Although the OpenGL core library allows only convex polygons, the GLU library provides functions for dealing with concave polygons and other nonconvex objects with linear boundaries. A set of GLU *polygon tessellation* routines is available for converting such shapes into a set of triangles, triangle meshes, triangle fans, and straight-line segments. Once such objects have been decomposed, they can be processed with basic OpenGL functions.

```

void quad (GLint n1, GLint n2, GLint n3, GLint n4)
{
    glBegin (GL_QUADS);
        glVertex3iv (pt [n1]);
        glVertex3iv (pt [n2]);
        glVertex3iv (pt [n3]);
        glVertex3iv (pt [n4]);
    glEnd ( );
}
void cube ( )
{
    quad (6, 2, 3, 7);
    quad (5, 1, 0, 4);
    quad (7, 3, 1, 5);
    quad (4, 0, 2, 6);
    quad (2, 0, 1, 3);
    quad (7, 5, 4, 6);
}

```

Thus, the specification for each face requires six OpenGL functions, and we have six faces to specify. When we add color specifications and other parameters, our display program for the cube could easily contain 100 or more OpenGL function calls. And scenes with many complex objects can require much more.

As we can see from the preceding cube example, a complete scene description could require hundreds or thousands of coordinate specifications. In addition, there are various attribute and viewing parameters that must be set for individual objects. Thus, object and scene descriptions could require an enormous number of function calls, which puts a demand on system resources and can slow execution of the graphics programs. A further problem with complex displays is that object surfaces (such as the cube in Figure 24) usually have shared vertex coordinates. Using the methods we have discussed up to now, these shared positions may need to be specified multiple times.

To alleviate these problems, OpenGL provides a mechanism for reducing the number of function calls needed in processing coordinate information. Using a **vertex array**, we can arrange the information for describing a scene so that we need only a very few function calls. The steps involved are

1. Invoke the function `glEnableClientState (GL_VERTEX_ARRAY)` to activate the vertex-array feature of OpenGL.
2. Use the function `glVertexPointer` to specify the location and data format for the vertex coordinates.
3. Display the scene using a routine such as `glDrawElements`, which can process multiple primitives with very few function calls.

Using the `pt` array previously defined for the cube, we implement these three steps in the following code example:

```

glEnableClientState (GL_VERTEX_ARRAY);
glVertexPointer (3, GL_INT, 0, pt);

GLubyte vertindex [] = (6, 2, 3, 7, 5, 1, 0, 4, 7, 3, 1, 5,
                        4, 0, 2, 6, 2, 0, 1, 3, 7, 5, 4, 6);

glDrawElements (GL_QUADS, 14, GL_UNSIGNED_BYTE, vertindex);

```

## 9 OpenGL Vertex Arrays

Although our examples so far have contained relatively few coordinate positions, describing a scene containing several objects can get much more complicated. To illustrate, we first consider describing a single, very basic object: the unit cube shown in Figure 24, with coordinates given in integers to simplify our discussion. A straightforward method for defining the vertex coordinates is to use a double-subscripted array, such as

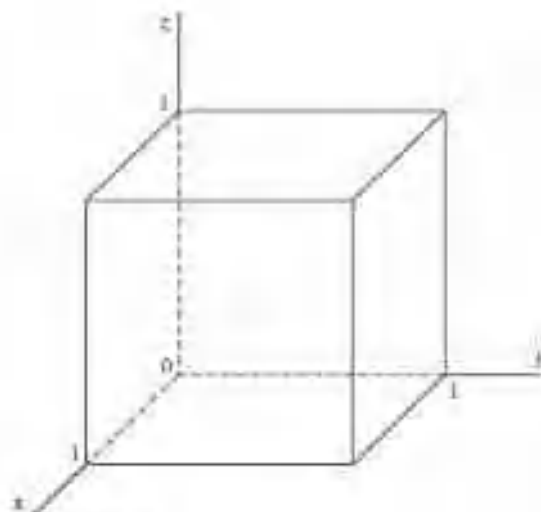
```
GLfloat points [8][3] = { {0, 0, 0}, {0, 1, 0}, {1, 0, 0}, {1, 1, 0},
                          {0, 0, 1}, {0, 1, 1}, {1, 0, 1}, {1, 1, 1} };
```

Alternatively, we could first define a data type for a three-dimensional vertex position and then give the coordinates for each vertex position as an element of a single-subscripted array as, for example,

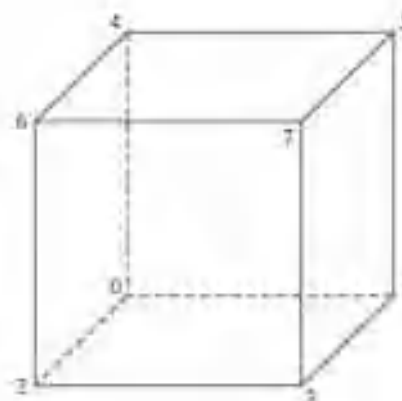
```
typedef GLfloat vertex3 [3];

vertex3 pt [8] = { {0, 0, 0}, {0, 1, 0}, {1, 0, 0}, {1, 1, 0},
                  {0, 0, 1}, {0, 1, 1}, {1, 0, 1}, {1, 1, 1} };
```

Next, we need to define each of the six faces of this object. For this, we could make six calls either to `glBegin (GL_POLYGON)` or to `glBegin (GL_QUADS)`. In either case, we must be sure to list the vertices for each face in a counterclockwise order when viewing that surface from the outside of the cube. In the following code segment, we specify each cube face as a quadrilateral and use a function call to pass array subscript values to the OpenGL primitive routines. Figure 25 shows the subscript values for array `pt` corresponding to the cube vertex positions.



**FIGURE 24**  
A cube with an edge length of 1



**FIGURE 25**  
Subscript values for array `pt` corresponding to the vertex coordinates for the cube shown in Figure 24.

With the first command, `glEnableClientState (GL_VERTEX_ARRAY)`, we activate a capability (in this case, a vertex array) on the client side of a client-server system. Because the client (the machine that is running the main program) retains the data for a picture, the vertex array must be there also. The server (our workstation, for example) generates commands and displays the picture. Of course, a single machine can be both client and server. The vertex-array feature of OpenGL is deactivated with the command

```
glDisableClientState (GL_VERTEX_ARRAY);
```

We next give the location and format of the coordinates for the object vertices in the function `glVertexPointer`. The first parameter in `glVertexPointer` (3 in this example) specifies the number of coordinates used in each vertex description. Data type for the vertex coordinates is designated using an OpenGL symbolic constant as the second parameter in this function. For our example, the data type is `GL_INT`. Other data types are specified with the symbolic constants `GL_BYTE`, `GL_SHORT`, `GL_FLOAT`, and `GL_DOUBLE`. With the third parameter, we give the byte offset between consecutive vertices. The purpose of this argument is to allow various kinds of data, such as coordinates and colors, to be packed together in one array. Because we are giving only the coordinate data, we assign a value of 0 to the offset parameter. The last parameter in the `glVertexPointer` function references the vertex array, which contains the coordinate values.

All the indices for the cube vertices are stored in array `vertIndex`. Each of these indices is the subscript for array `pt` corresponding to the coordinate values for that vertex. This index list is referenced as the last parameter value in function `glDrawElements` and is then used by the primitive `GL_QUADS`, which is the first parameter, to display the set of quadrilateral surfaces for the cube. The second parameter specifies the number of elements in array `vertIndex`. Because a quadrilateral requires just 4 vertices and we specified 24, the `glDrawElements` function continues to display another cube face after each successive set of 4 vertices until all 24 have been processed. Thus, we accomplish the final display of all faces of the cube with this single function call. The third parameter in function `glDrawElements` gives the type for the index values. Because our indices are small integers, we specified a type of `GL_UNSIGNED_BYTE`. The two other index types that can be used are `GL_UNSIGNED_SHORT` and `GL_UNSIGNED_INT`.

Additional information can be combined with the coordinate values in the vertex arrays to facilitate the processing of a scene description. We can specify color values and other attributes for objects in arrays that can be referenced by the `glDrawElements` function. Also, we can interlace the various arrays for greater efficiency.

---

## 10 Pixel-Array Primitives

In addition to straight lines, polygons, circles, and other primitives, graphics packages often supply routines to display shapes that are defined with a rectangular array of color values. We can obtain the rectangular grid pattern by digitizing (scanning) a photograph or other picture or by generating a shape with a graphics program. Each color value in the array is then mapped to one or more screen pixel positions. A pixel array of color values is typically referred to as a *pixmap*.



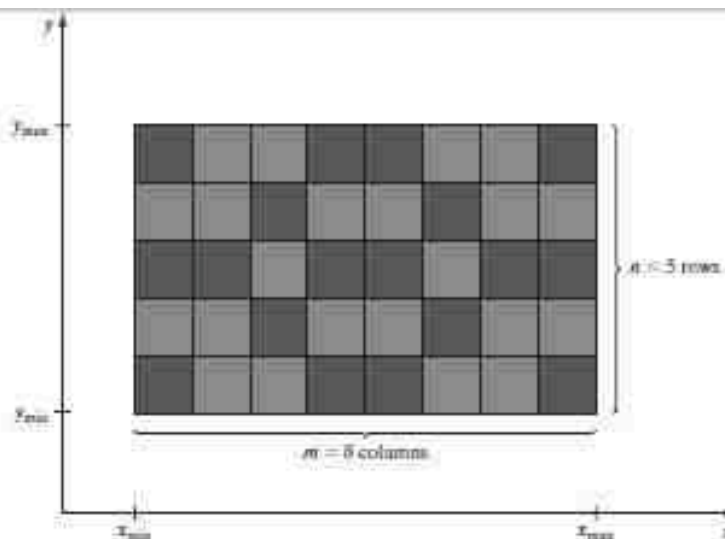


FIGURE 26  
Mapping an  $n$  by  $m$  color array onto a  
region of the screen coordinates.

Parameters for a pixel array can include a pointer to the color matrix, the size of the matrix, and the position and size of the screen area to be affected by the color values. Figure 26 gives an example of mapping a pixel-color array onto a screen area.

Another method for implementing a pixel array is to assign either the bit value 0 or the bit value 1 to each element of the matrix. In this case, the array is simply a *bitmap*, which is sometimes called a *mask*, that indicates whether a pixel is to be assigned (or combined with) a preset color.

## 11 OpenGL Pixel-Array Functions

There are two functions in OpenGL that we can use to define a shape or pattern specified with a rectangular array. One function defines a bitmap pattern, and the other a pixmap pattern. Also, OpenGL provides several routines for saving, copying, and manipulating arrays of pixel values.

### OpenGL Bitmap Function

A binary array pattern is defined with the function

```
glBitmap (width, height, x0, y0, xoffset, yoffset, bitShape);
```

Parameters `width` and `height` in this function give the number of columns and number of rows, respectively, in the array `bitShape`. Each element of `bitShape` is assigned either a 1 or a 0. A value of 1 indicates that the corresponding pixel is to be displayed in a previously defined color. Otherwise, the pixel is unaffected by the bitmap. (As an option, we could use a value of 1 to indicate that a specified color is to be combined with the color value stored in the refresh buffer at that position.) Parameters `x0` and `y0` define the position that is to be considered the "origin" of the rectangular array. This origin position is specified relative to the lower-left corner of `bitShape`, and values for `x0` and `y0` can be positive or negative. In addition, we need to designate a location in the frame buffer where the pattern is to be applied. This location is called the **current raster position**, and the bitmap is displayed by positioning its origin,  $(x_0, y_0)$ , at the current

raster position. Values assigned to parameters `xOffset` and `yOffset` are used as coordinate offsets to update the frame-buffer current raster position after the bitmap is displayed.

Coordinate values for `x0`, `y0`, `xOffset`, and `yOffset`, as well as the current raster position, are maintained as floating-point values. Of course, bitmaps will be applied at integer pixel positions. But floating-point coordinates allow a set of bitmaps to be spaced at arbitrary intervals, which is useful in some applications, such as forming character strings with bitmap patterns.

We use the following routine to set the coordinates for the current raster position:

```
glRasterPos* ( )
```

Parameters and suffix codes are the same as those for the `glVertex` function. Thus, a current raster position is given in world coordinates, and it is transformed to screen coordinates by the viewing transformations. For our two-dimensional examples, we can specify coordinates for the current raster position directly in integer screen coordinates. The default value for the current raster position is the world-coordinate origin (0, 0, 0).

The color for a bitmap is the color that is in effect at the time that the `glRasterPos` command is invoked. Any subsequent color changes do not affect the bitmap.

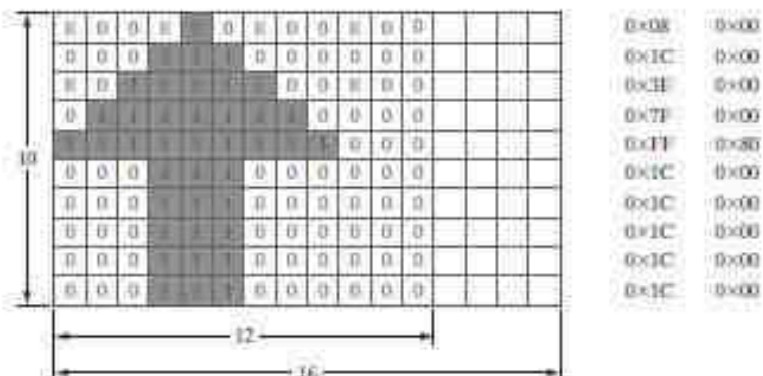
Each row of a rectangular bit array is stored in multiples of 8 bits, where the binary data is arranged as a set of 8-bit unsigned characters. But we can describe a shape using any convenient grid size. For example, Figure 27 shows a bit pattern defined on a 10-row by 9-column grid, where the binary data is specified with 16 bits for each row. When this pattern is applied to the pixels in the frame buffer, all bit values beyond the ninth column are ignored.

We apply the bit pattern of Figure 27 to a frame-buffer location with the following code section:

```
GLubyte bitShape [20] = {
    0x1c, 0x00, 0x1c, 0x00, 0x1c, 0x00, 0x1c, 0x00, 0x1c, 0x00,
    0xff, 0x50, 0x7f, 0x00, 0x3e, 0x00, 0x1c, 0x00, 0x08, 0x00};

glPixelStorei (GL_UNPACK_ALIGNMENT, 1); // Set pixel storage mode.

glRasterPos2i (30, 40);
glBitmap (9, 10, 0.0, 0.0, 20.0, 15.0, bitShape);
```



**FIGURE 27**  
A bit pattern, specified in an array with 10 rows and 9 columns, is stored in 8-bit blocks of 10 rows with 16 bit values per row.

Array values for `bitShape` are specified row by row, starting at the bottom of the rectangular-grid pattern. Next we set the storage mode for the bitmap with the OpenGL routine `glPixelStorei`. The parameter value of 1 in this function indicates that the data values are to be aligned on byte boundaries. With `glRasterPos`, we set the current raster position to (30, 40). Finally, function `glBitmap` specifies that the bit pattern is given in array `bitShape`, and that this array has 9 columns and 10 rows. The coordinates for the origin of this pattern are (0.0, 0.0), which is the lower-left corner of the grid. We illustrate a coordinate offset with the values (20.0, 15.0), although we do not use the offset in this example.

## OpenGL Pixmap Function

A pattern defined as an array of color values is applied to a block of frame-buffer pixel positions with the function

```
glDrawPixels (width, height, dataFormat, dataType, pixmap):
```

Again, parameters `width` and `height` give the column and row dimensions, respectively, of the array `pixmap`. Parameter `dataFormat` is assigned an OpenGL constant that indicates how the values are specified for the array. For example, we could specify a single blue color for all pixels with the constant `GL_BLUE`, or we could specify three color components in the order blue, green, red with the constant `GL_BGR`. A number of other color specifications are possible. An OpenGL constant, such as `GL_BYTE`, `GL_INT`, or `GL_FLOAT`, is assigned to parameter `dataType` to designate the data type for the color values in the array. The lower-left corner of this color array is mapped to the current raster position, as set by the `glRasterPos` function. As an example, the following statement displays a pixmap defined in a 128 × 128 array of RGB color values:

```
glDrawPixels (128, 128, GL_RGB, GL_UNSIGNED_BYTE, colorShape):
```

Because OpenGL provides several buffers, we can paste an array of values into a particular buffer by selecting that buffer as the target of the `glDrawPixels` routine. Some buffers store color values and some store other kinds of pixel data. A *depth buffer*, for instance, is used to store object distances (depths) from the viewing position, and a *stencil buffer* is used to store boundary patterns for a scene. We select one of these two buffers by setting parameter `dataFormat` in the `glDrawPixels` routine to either `GL_DEPTH_COMPONENT` or `GL_STENCIL_INDEX`. For these buffers, we would need to set up the pixel array using either depth values or stencil information.

There are four *color buffers* available in OpenGL that can be used for screen refreshing. Two of the color buffers constitute a left-right scene pair for displaying stereoscopic views. For each of the stereoscopic buffers, there is a front-back pair for double-buffered animation displays. In a particular implementation of OpenGL, either stereoscopic viewing or double buffering, or both, might not be supported. If neither stereoscopic effects nor double buffering is supported, then there is only a single refresh buffer, which is designated as the *front-left color buffer*. This is the default refresh buffer when double buffering is not available or not in effect. If double buffering is in effect, the default is either the back-left and back-right buffers or only the back-left buffer, depending on the current state of stereoscopic viewing. Also, a number of user-defined, auxiliary color buffers are supported that can be used for any nonrefresh purpose, such as saving a picture that is to be copied later into a refresh buffer for display.

We select a single color or auxiliary buffer or a combination of color buffers for storing a pixmap with the following command:

```
glDrawBuffer (buffer);
```

A variety of OpenGL symbolic constants can be assigned to parameter *buffer* to designate one or more "draw" buffers. For instance, we can pick a single buffer with either `GL_FRONT_LEFT`, `GL_FRONT_RIGHT`, `GL_BACK_LEFT`, or `GL_BACK_RIGHT`. We can select both front buffers with `GL_FRONT`, and we can select both back buffers with `GL_BACK`. This is assuming that stereoscopic viewing is in effect. Otherwise, the previous two symbolic constants designate a single buffer. Similarly, we can designate either the left or right buffer pairs with `GL_LEFT` or `GL_RIGHT`, and we can select all the available color buffers with `GL_FRONT_AND_BACK`. An auxiliary buffer is chosen with the constant `GL_AUXk`, where *k* is an integer value from 0 to 3, although more than four auxiliary buffers may be available in some implementations of OpenGL.

### OpenGL Raster Operations

In addition to storing an array of pixel values in a buffer, we can retrieve a block of values from a buffer or copy the block into another buffer area, and we can perform a variety of other operations on a pixel array. In general, the term *raster operation* or *raster op* is used to describe any function that processes a pixel array in some way. A raster operation that moves an array of pixel values from one place to another is also referred to as a *block transfer* of pixel values. On a bilevel system, these operations are called *bitblt transfers* (*bit-block transfers*), particularly when the functions are hardware-implemented. On a multilevel system, the term *pixblt* is sometimes used for block transfers.

We use the following function to select a rectangular block of pixel values in a designated set of buffers:

```
glReadPixels (xmin, ymin, width, height,
             dataFormat, dataType, array);
```

The lower-left corner of the rectangular block to be retrieved is at screen-coordinate position (*xmin*, *ymin*). Parameters *width*, *height*, *dataFormat*, and *dataType* are the same as in the `glDrawPixels` routine. The type of data to be saved in parameter *array* depends on the selected buffer. We can choose either the depth buffer or the stencil buffer by assigning either the value `GL_DEPTH_COMPONENT` or the value `GL_STENCIL_INDEX` to parameter *dataFormat*.

A particular combination of color buffers or an auxiliary buffer is selected for the application of the `glReadPixels` routine with the function:

```
glReadBuffer (buffer);
```

Symbolic constants for specifying one or more buffers are the same as in the `glDrawBuffer` routine except that we cannot select all four of the color buffers. The default buffer selection is the front left-right pair or just the front-left buffer, depending on the status of stereoscopic viewing.

We can also copy a block of pixel data from one location to another within the set of OpenGL buffers using the following routine:

```
glCopyPixels (xmin, ymin, width, height, pixelValues);
```

The lower-left corner of the block is at screen-coordinate location (`xmin`, `ymin`), and parameters `width` and `height` are assigned positive integer values to designate the number of columns and rows, respectively, that are to be copied. Parameter `pixelvalues` is assigned either `GL_COLOR`, `GL_DEPTH`, or `GL_STENCIL` to indicate the kind of data we want to copy: color values, depth values, or stencil values. In addition, the block of pixel values is copied from a *source buffer* to a *destination buffer*, with its lower-left corner mapped to the current raster position. We select the source buffer with the `glReadBuffer` command, and we select the destination buffer with the `glDrawBuffer` command. Both the region to be copied and the destination area should lie completely within the bounds of the screen coordinates.

To achieve different effects as a block of pixel values is placed into a buffer with `glDrawPixels` or `glCopyPixels`, we can combine the incoming values with the old buffer values in various ways. As an example, we could apply logical operations, such as *and*, *or*, and *exclusive or*, to combine the two blocks of pixel values. In OpenGL, we select a bitwise, logical operation for combining incoming and destination pixel color values with the functions

```
glEnable (GL_COLOR_LOGIC_OP);

glLogicOp (logicOp);
```

A variety of symbolic constants can be assigned to parameter `logicOp`, including `GL_AND`, `GL_OR`, and `GL_XOR`. In addition, either the incoming bit values or the destination bit values can be inverted (interchanging 0 and 1 values). We use the constant `GL_COPY_INVERTED` to invert the incoming color bit values and then replace the destination values with the inverted incoming values; and we could simply invert the destination bit values without replacing them with the incoming values using `GL_INVERT`. The various invert operations can also be combined with the logical *and*, *or*, and *exclusive or* operations. Other options include clearing all the destination bits to the value 0 (`GL_CLEAR`), or setting all the destination bits to the value 1 (`GL_SET`). The default value for the `glLogicOp` routine is `GL_COPY`, which simply replaces the destination values with the incoming values.

Additional OpenGL routines are available for manipulating pixel arrays processed by the `glDrawPixels`, `glReadPixels`, and `glCopyPixels` functions. For example, the `glPixelTransfer` and `glPixelMap` routines can be used to shift or adjust color values, depth values, or stencil values. We return to pixel operations in later chapters as we explore other facets of computer-graphics packages.

---

## 12 Character Primitives

Graphics displays often include textural information, such as labels on graphs and charts, signs on buildings or vehicles, and general identifying information for simulation and visualization applications. Routines for generating character primitives are available in most graphics packages. Some systems provide an extensive set of character functions, while other systems offer only minimal support for character generation.

Letters, numbers, and other characters can be displayed in a variety of sizes and styles. The overall design style for a set (or family) of characters is called a **typeface**. Today, there are thousands of typefaces available for computer applications. Examples of a few common typefaces are Courier, Helvetica, New York,



Computer Graphics with OpenGL (4th ed.) [Hearn, Baker & Carithers 2013].pdf - Adobe Reader

File Edit View Document Tools Window Help

78 (85 of 819) 65% OUTPUT PRIMITIV

Palatino and Zapf Chancery. Originally, the term *font* referred to a set of cast metal character forms in a particular size and format, such as 10-point Courier Italic or 12-point Palatino Bold. A 14-point font has a total character height of about 0.5 centimeter. In other words, 72 points is about the equivalent of 2.54 centimeters (1 inch). The terms *font* and *typeface* are now often used interchangeably, since most printing is no longer done with cast metal forms.

Fonts can be divided into two broad groups: *serif* and *sans serif*. Serif type has small lines or accents at the ends of the main character strokes, while sans-serif type does not have such accents. For example, this text is set in a serif font (Palatino), but this sentence is printed in a sans-serif font (Univers). Serif type is generally more *readable*; that is, it is easier to read in longer blocks of text. On the other hand, the individual characters in sans-serif type are easier to recognize. For this reason, sans-serif type is said to be more *legible*. Since sans-serif characters can be recognized quickly, this font is good for labeling and short headings.

Fonts are also classified according to whether they are monospace or proportional. Characters in a monospace font all have the same width. In a proportional font, character widths varies.

Two different representations are used for storing computer fonts. A simple method for representing the character shapes in a particular typeface is to set up a pattern of binary values on a rectangular grid. The set of characters is then referred to as a **bitmap font** (or **bitmapped font**). A bitmapped character set is also sometimes referred to as a **raster font**. Another, more flexible, scheme is to describe character shapes using straight-line and curve sections, as in **FontScript**, for example. In this case, the set of characters is called an **outline font** or a **stroke font**. Figure 28 illustrates the two methods for character representation. When the pattern in Figure 28(a) is applied to an area of the frame buffer, the 1 bits designate which pixel positions are to be displayed in a specified color. To display the character shape in Figure 28(b), the interior of the character outline is treated as a fill area.

Bitmap fonts are the simplest to define and display. We just need to map the character grids to a frame-buffer position. In general, however, bitmap fonts require more storage space because each variation (size and format) must be saved in a *font cache*. It is possible to generate different sizes and other variations, such as bold and italic, from one bitmap font set, but this often does not produce good results. We can increase or decrease the size of a character bitmap only in integer multiples of the pixel size. To double the size of a character, we need to double the number of pixels in the bitmap. This just increases the ragged appearance of its edges.

In contrast to bitmap fonts, outline fonts can be increased in size without distorting the character shapes. And outline fonts require less storage because each variation does not require a distinct font cache. We can produce boldface,

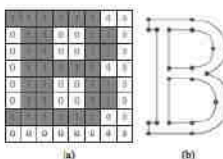


FIGURE 28  
The letter "B" represented with an 8 × 8  
bitmap pattern (a) and with an outline  
shape defined with straight-line and curve  
segments (b).

Start Computer Graphics wi... A JEROME ROBINSON E... FIELD @RESEARCH My Computer Local Disk (C:) 3:52 PM

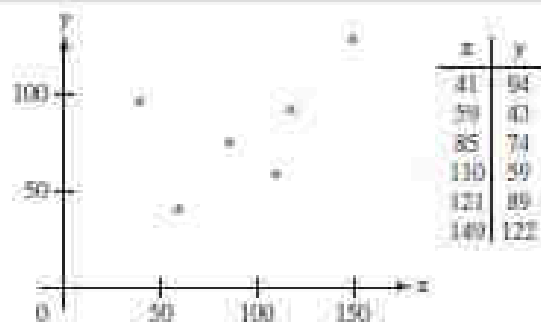


FIGURE 29  
A polymarker graph of a set of data values.

italic, or different sizes by manipulating the curve definitions for the character outlines. But it does take more time to process the outline fonts because they must be scan-converted into the frame buffer.

There is a variety of possible functions for implementing character displays. Some graphics packages provide a function that accepts any character string and a frame-buffer starting position for the string. Another type of function displays a single character at one or more selected positions. Since this character routine is useful for showing markers in a network layout or in displaying a point plot of a discrete data set, the character displayed by this routine is sometimes referred to as a **marker symbol** or **polymarker**, in analogy with a polyline primitive. In addition to standard characters, special shapes such as dots, circles, and crosses are often available as marker symbols. Figure 29 shows a plot of a discrete data set using an asterisk as a marker symbol.

Geometric descriptions for characters are given in world coordinates, just as they are for other primitives, and this information is mapped to screen coordinates by the viewing transformations. A bitmap character is described with a rectangular grid of binary values and a grid reference position. This reference position is then mapped to a specified location in the frame buffer. An outline character is defined by a set of coordinate positions that are to be connected with a series of curves and straight-line segments and a reference position that is to be mapped to a given frame-buffer location. The reference position can be specified either for a single outline character or for a string of characters. In general, character routines can allow the construction of both two-dimensional and three-dimensional character displays.

## 13 OpenGL Character Functions

Only low-level support is provided by the basic OpenGL library for displaying individual characters and text strings. We can explicitly define any character as a bitmap, as in the example shape shown in Figure 27, and we can store a set of bitmap characters as a font list. A text string is then displayed by mapping a selected sequence of bitmaps from the font list into adjacent positions in the frame buffer.

However, some predefined character sets are available in the GLUT library, so we do not need to create our own fonts as bitmap shapes unless we want to display a font that is not available in GLUT. The GLUT library contains routines for displaying both bitmapped and outline fonts. Bitmapped GLUT fonts are rendered using the OpenGL `glBitmap` function, and the outline fonts are generated with polyline (`GL_LINE_STRIP`) boundaries.

We can display a bitmap GLUT character with

```
glutBitmapCharacter (font, character);
```

where parameter `font` is assigned a symbolic GLUT constant identifying a particular set of typefaces, and parameter `character` is assigned either the ASCII code or the specific character we wish to display. Thus, to display the uppercase letter "A," we can either use the ASCII value 65 or the designation 'A'. Similarly, a code value of 66 is equivalent to 'B', code 97 corresponds to the lowercase letter 'a', code 98 corresponds to 'b', and so forth. Both fixed-width fonts and proportionally spaced fonts are available. We can select a fixed-width font by assigning either `GLUT_BITMAP_8_BY_13` or `GLUT_BITMAP_9_BY_15` to parameter `font`. And we can select a 10-point, proportionally spaced font with either `GLUT_BITMAP_TIMES_ROMAN_10` or `GLUT_BITMAP_HELVETICA_10`. A 12-point Times-Roman font is also available, as well as 12-point and 18-point Helvetica fonts.

Each character generated by `glutBitmapCharacter` is displayed so that the origin (lower-left corner) of the bitmap is at the current raster position. After the character bitmap is loaded into the refresh buffer, an offset equal to the width of the character is added to the `x` coordinate for the current raster position. As an example, we could display a text string containing 36 bitmap characters with the following code:

```
glRasterPosition2i (x, y);
for (k = 0; k < 36; k++)
    glutBitmapCharacter (GLUT_BITMAP_9_BY_15, text [k]);
```

Characters are displayed in the color that was specified before the execution of the `glutBitmapCharacter` function.

An outline character is displayed with the following function call:

```
glutStrokeCharacter (font, character);
```

For this function, we can assign parameter `font` either the value `GLUT_STROKE_ROMAN`, which displays a proportionally spaced font, or the value `GLUT_STROKE_MONO_ROMAN`, which displays a font with constant spacing. We control the size and position of these characters by specifying transformation operations before executing the `glutStrokeCharacter` routine. After each character is displayed, a coordinate offset is applied automatically so that the position for displaying the next character is to the right of the current character. Text strings generated with outline fonts are part of the geometric description for a two-dimensional or three-dimensional scene because they are constructed with line segments. Thus, they can be viewed from various directions, and we can shrink or expand them without distortion, or transform them in other ways. But they are slower to render, compared to bitmapped fonts.

---

## 14 Picture Partitioning

Some graphics libraries include routines for describing a picture as a collection of named sections and for manipulating the individual sections of a picture. Using these functions, we can create, edit, delete, or move a part of a picture independently of the other picture components. In addition, we can use this feature of a graphics package for hierarchical modeling, in which an object description is given as a tree structure composed of a number of levels specifying the object subparts.

Various names are used for the subsections of a picture. Some graphics packages refer to them as `structures`, while other packages call them `segments`.

or objects. Also, the allowable subsection operations vary greatly from one package to another. Modeling packages, for example, provide a wide range of operations that can be used to describe and manipulate picture elements. On the other hand, for any graphics library, we can always structure and manage the components of a picture using procedural elements available in a high-level language such as C++.

---

## 15 OpenGL Display Lists

Often it can be convenient or more efficient to store an object description (or any other set of OpenGL commands) as a named sequence of statements. We can do this in OpenGL using a structure called a **display list**. Once a display list has been created, we can reference the list multiple times with different display operations. On a network, a display list describing a scene is stored on the server machine, which eliminates the need to transmit the commands in the list each time the scene is to be displayed. We can also set up a display list so that it is saved for later execution, or we can specify that the commands in the list be executed immediately. And display lists are particularly useful for hierarchical modeling, where a complex object can be described with a set of simpler subparts.

### Creating and Naming an OpenGL Display List

A set of OpenGL commands is formed into a display list by enclosing the commands within the `glNewList/glEndList` pair of functions. For example,

```
glNewList (listID, listMode);
.
.
.
glEndList ( );
```

This structure forms a display list with a positive integer value assigned to parameter `listID` as the name for the list. Parameter `listMode` is assigned an OpenGL symbolic constant that can be either `GL_COMPILE` or `GL_COMPILE_AND_EXECUTE`. If we want to save the list for later execution, we use `GL_COMPILE`. Otherwise, the commands are executed as they are placed into the list, in addition to allowing us to execute the list again at a later time.

As a display list is created, expressions involving parameters such as coordinate positions and color components are evaluated so that only the parameter values are stored in the list. Any subsequent changes to these parameters have no effect on the list. Because display-list values cannot be changed, we cannot include certain OpenGL commands, such as vertex-list pointers, in a display list.

We can create any number of display lists, and we execute a particular list of commands with a call to its identifier. Further, one display list can be embedded within another display list. But if a list is assigned an identifier that has already been used, the new list replaces the previous list that had been assigned that identifier. Therefore, to avoid losing a list by accidentally reusing its identifier, we can let OpenGL generate an identifier for us, as follows:

```
listID = glGenLists (1);
```

This statement returns one (1) unused positive integer identifier to the variable `listID`. A range of unused integer list identifiers is obtained if we change the argument of `glGenLists` from the value 1 to some other positive integer. For

instance, if we invoke `glGenLists (6)`, then a sequence of six contiguous positive integer values is reserved and the first value in this list of identifiers is returned to the variable `listID`. A value of 0 is returned by the `glGenLists` function if an error occurs or if the system cannot generate the range of contiguous integers requested. Therefore, before using an identifier obtained from the `glGenLists` routine, we could check to be sure that it is not 0.

Although unused list identifiers can be generated with the `glGenList` function, we can independently query the system to determine whether a specific integer value has been used as a list name. The function to accomplish this is

```
glIsList (listID);
```

A value of `GL_TRUE` is returned if the value of `listID` is an integer that has already been used as a display-list name. If the integer value has not been used as a list name, the `glIsList` function returns the value `GL_FALSE`.

### Executing OpenGL Display Lists

We execute a single display list with the statement

```
glCallList (listID);
```

The following code segment illustrates the creation and execution of a display list. We first set up a display list that contains the description for a regular hexagon, defined in the  $xy$  plane using a set of six equally spaced vertices around the circumference of a circle, whose center coordinates are (200, 200) and whose radius is 150. Then we issue a call to function `glCallList`, which displays the hexagon.

```
const double TWO_PI = 6.2831853;

GLuint regHex;

double theta;
int x, y, k;

/* Set up a display list for a regular hexagon.
 * Vertices for the hexagon are six equally spaced
 * points around the circumference of a circle.
 */
regHex = glGenLists (1); // Get an identifier for the display list.
glNewList (regHex, GL_COMPILE);
glBegin (GL_POLYGON);
  for (k = 0; k < 6; k++) {
    theta = TWO_PI * k / 6.0;
    x = 200 + 150 * cos (theta);
    y = 200 + 150 * sin (theta);
    glVertex2i (x, y);
  }
glEnd ();
glEndList ();

glCallList (regHex);
```



Several display lists can be executed using the following two statements:

```
glListBase (offsetValue);

glCallLists (nLists, arrayDataType, listIDArray);
```

The integer number of lists that we want to execute is assigned to parameter `nLists`, and parameter `listIDArray` is an array of display-list identifiers. In general, `listIDArray` can contain any number of elements, and invalid display-list identifiers are ignored. Also, the elements in `listIDArray` can be specified in a variety of data formats, and parameter `arrayDataType` is used to indicate a data type, such as `GL_BYTE`, `GL_INT`, `GL_FLOAT`, `GL_3_BYTES`, or `GL_4_BYTES`. A display-list identifier is calculated by adding the value in an element of `listIDArray` to the integer value of `offsetValue` that is given in the `glListBase` function. The default value for `offsetValue` is 0.

This mechanism for specifying a sequence of display lists that are to be executed allows us to set up groups of related display lists, whose identifiers are formed from symbolic names or codes. A typical example is a font set where each display-list identifier is the ASCII value of a character. When several font sets are defined, we use parameter `offsetValue` in the `glListBase` function to obtain a particular font described within the array `listIDArray`.

## Deleting OpenGL Display Lists

We eliminate a contiguous set of display lists with the function call

```
glDeleteLists (startID, nLists);
```

Parameter `startID` gives the initial display-list identifier, and parameter `nLists` specifies the number of lists that are to be deleted. For example, the statement

```
glDeleteLists (5, 4);
```

eliminates the four display lists with identifiers 5, 6, 7, and 8. An identifier value that references a nonexistent display list is ignored.

---

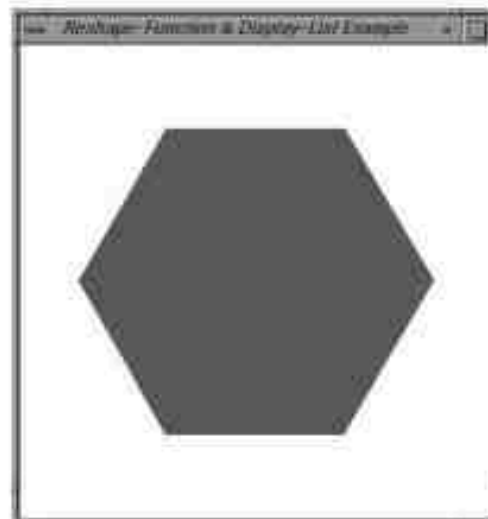
## 16 OpenGL Display-Window Reshape Function

After the generation of our picture, we often want to use the mouse pointer to drag the display window to another screen location or to change its size. Changing the size of a display window could change its aspect ratio and cause objects to be distorted from their original shapes.

To allow us to compensate for a change in display-window dimensions, the GLUT library provides the following routine:

```
glutReshapeFunc (winReshapeFcn);
```

We can include this function in the `main` procedure in our program, along with the other GLUT routines, and it will be activated whenever the display-window size is altered. The argument for this GLUT function is the name of a procedure that



**FIGURE 30**  
The display window generated by the example program illustrating the use of the reshape function.

is to receive the new display-window width and height. We can then use the new dimensions to reset the projection parameters and perform any other operations, such as changing the display-window color. In addition, we could save the new width and height values so that they could be used by other procedures in our program.

As an example, the following program illustrates how we might structure the `winReshapeFcn` procedure. The `glLoadIdentity` command is included in the `reshape` function so that any previous values for the projection parameters will not affect the new projection settings. This program displays the regular hexagon discussed in Section 15. Although the hexagon center (at the position of the circle center) in this example is specified in terms of the display-window parameters, the position of the hexagon is unaffected by any changes in the size of the display window. This is because the hexagon is defined within a display list, and only the original center coordinates are stored in the list. If we want the position of the hexagon to change when the display window is resized, we need to define the hexagon in another way or alter the coordinate reference for the display window. The output from this program is shown in Figure 30.

```
#include <GL/glut.h>
#include <math.h>
#include <stdlib.h>

const double TWO_PI = 6.2831853;

/* Initial display window size. */
#define winWidth = 400, winHeight = 400;
GLuint regHex;

class screenPt
{
private:
    GLint x, y;
```

```

public:
    /* Default Constructor: initializes coordinate position to (0, 0). */
    coordPt ( ) {
        x = y = 0;
    }

    void setCoords (GLint xCoord, GLint yCoord) {
        x = xCoord;
        y = yCoord;
    }

    GLint getX ( ) const {
        return x;
    }

    GLint getY ( ) const {
        return y;
    }
};

static void init (void)
{
    screenPt hasVertex, circCtr;
    GLfloat theta;
    GLint k;

    /* Set circle center coordinates. */
    circCtr.setCoords (winWidth / 2, winHeight / 2);

    glClearColor (1.0, 1.0, 1.0, 0.0); // Display-window color = white.

    /* Set up a display list for a red regular hexagon.
     * Vertices for the hexagon are six equally spaced
     * points around the circumference of a circle.
     */
    regHex = glGenLists (1); // Get an identifier for the display list.
    glNewList (regHex, GL_COMPILE);
    glColor3f (1.0, 0.0, 0.0); // Set fill color for hexagon to red.
    glBegin (GL_POLYGON);
        for (k = 0; k < 6; k++) {
            theta = TWO_PI * k / 6.0;
            hasVertex.setCoords (circCtr.getX ( ) + 150 * cos (theta),
                                circCtr.getY ( ) + 150 * sin (theta));
            glVertex3f (hasVertex.getX ( ), hasVertex.getY ( ));
        }
    glEnd ( );
    glEndList ( );
}

void regHexagon (void)
{
    glClear (GL_COLOR_BUFFER_BIT);

    glCallList (regHex);

    glFlush ( );
}

```

```

void reshapeFcn (int newWidth, int newHeight)
{
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluOrtho2D (0.0, (GLdouble) newWidth, 0.0, (GLdouble) newHeight);

    glClearColor (GL_COLOR_BUFFER_BIT);
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Reshape-Function & Display-List Example");

    init ();
    glutDisplayFunc (ragHexagon);
    glutReshapeFunc (winReshapeFcn);

    glutMainLoop ();
}

```

FIGURE 1  
Star-saw effect (jaggies) produced  
when a line is generated as a series of  
pixel positions.



## LINE FUNCTION

Graphics packages typically provide a function for specifying one or more straight-line segments, where each line segment is defined by two endpoint coordinate positions. In OpenGL, we select a single endpoint coordinate position using the **glVertex** function, just as we did for a point position. And we enclose a list of **glVertex** functions between the **glBegin/glEnd** pair. But now we use a symbolic constant as the argument for the **glBegin** function that interprets a list of positions as the endpoint coordinates for line segments. There are three symbolic constants in OpenGL that we can use to specify how a list of endpoint positions should be connected to form a set of straight-line segments. By default, each symbolic constant displays solid, white lines.

A set of straight-line segments between each successive pair of endpoints in a list is generated using the primitive line constant **GL\_LINES**. In general, this will result in a set of unconnected lines unless some coordinate positions are repeated, because OpenGL considers lines to be connected only if they share a vertex; lines that cross but do not share a vertex are still considered to be unconnected. Nothing is displayed if only one endpoint is specified, and the last endpoint is not processed if the number of endpoints listed is odd. For example, if we have five coordinate positions, labeled **p1** through **p5**, and each is represented as a two-dimensional array, then the following code could generate the display shown in Figure 4(a):

```

glBegin (GL_LINES);
glVertex2iv (p1);
glVertex2iv (p2);
glVertex2iv (p3);
glVertex2iv (p4);
glVertex2iv (p5);
glEnd ( );

```

Thus, we obtain one line segment between the first and second coordinate positions and another line segment between the third and fourth positions. In this case, the number of specified endpoints is odd, so the last coordinate position is ignored.

With the OpenGL primitive constant **GL LINE STRIP**, we obtain a **polyline**. In this case, the display is a sequence of connected line segments between the first endpoint in the list and the last endpoint. The first line segment in the polyline is displayed between the first endpoint and the second endpoint; the second line segment is between the second and third endpoints; and so forth, up to the last line endpoint. Nothing is displayed if we do not list at least two coordinate positions.

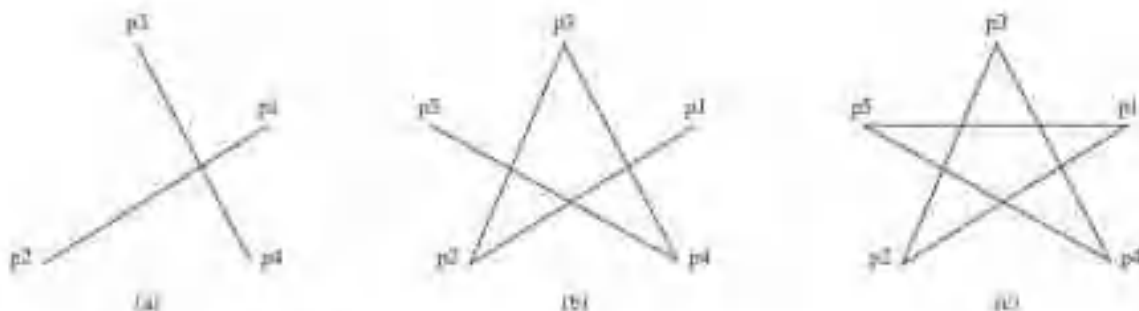


FIGURE 4

Line segments that can be displayed in OpenGL using a list of five endpoint coordinates. (a) An unordered set of lines generated with the primitive line constant `GL_LINES`. (b) A polyline generated with `GL_LINE_STRIP`. (c) A closed polyline generated with `GL_LINE_LOOP`.

Using the same five coordinate positions as in the previous example, we obtain the display in Figure 4(b) with the code

```

glBegin (GL_LINE_STRIP);
glVertex2iv (p1);
glVertex2iv (p2);
glVertex2iv (p3);
glVertex2iv (p4);
glVertex2iv (p5);
glEnd ( );

```

The third OpenGL line primitive is `GL_LINE_LOOP`, which produces a **closed** polyline. Lines are drawn as with `GL_LINE_STRIP`, but an additional line is drawn to connect the last coordinate position and the first coordinate position. Figure 4(c) shows the display of our endpoint list when we select this line option, using the code



```

glBegin (GL_LINE_LOOP);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p3);
    glVertex2iv (p4);
    glVertex2iv (p5);
glEnd ( );

```

As noted earlier, picture components are described in a world-coordinate reference frame that is eventually mapped to the coordinate reference for the output device. Then the geometric information about the picture is scan-converted to pixel positions.

### LINE DRAWING ALGORITHMS

A straight-line segment in a scene is defined by the coordinate positions for the endpoints of the segment. To display the line on a raster monitor, the graphics system must first project the endpoints to integer screen coordinates and determine the nearest pixel positions along the line path between the two endpoints. Then the line color is loaded into the frame buffer at the corresponding pixel coordinates. Reading from the frame buffer, the video controller plots the screen pixels. This process digitizes the line into a set of discrete integer positions that, in general, only approximates the actual line path. A computed line position of (10.48, 20.51), for example, is converted to pixel position (10, 21). This rounding of coordinate values to integers causes all but horizontal and vertical lines to be displayed with a stair-step appearance (known as “the jaggies”), as represented in Figure 1. The characteristic stair-step shape of raster lines is particularly noticeable on systems with low resolution, and we can improve their appearance somewhat by displaying them on high-resolution systems. More effective techniques for smoothing a raster line are based on adjusting pixel intensities along the line path.

#### INITIALIZING LINES

We determine pixel positions along a straight-line path from the geometric properties of the line. The Cartesian *slope-intercept equation* for a straight line is

$$y = m \cdot x + b \quad (1)$$

with  $m$  as the slope of the line and  $b$  as the  $y$  intercept. Given that the two endpoints of a line segment are specified at positions  $(x_0, y_0)$  and  $(x_{end}, y_{end})$ , as shown in Figure 2, we can determine values for the slope  $m$  and  $y$  intercept  $b$  with the following calculations:

$$m = \frac{y_{end} - y_0}{x_{end} - x_0} \quad (2)$$

$$b = y_0 - m \cdot x_0 \quad (3)$$

Algorithms for displaying straight lines are based on Equation 1 and the calculations given in Equations 2 and 3.

For any given  $x$  interval  $\delta x$  along a line, we can compute the corresponding  $y$  interval,  $\delta y$ , from Equation 2 as

$$\delta y = m \cdot \delta x \quad (4)$$

Similarly, we can obtain the  $x$  interval  $\delta x$  corresponding to a specified  $\delta y$  as  $\delta x = \delta y m$  (5)

These equations form the basis for determining deflection voltages in analog displays, such as a vector-scan system, where arbitrarily small changes in deflection voltage are possible. For lines with slope magnitudes  $|m| < 1$ ,  $\delta x$  can be set proportional to a small horizontal deflection voltage, and the corresponding vertical deflection is then set proportional to  $\delta y$  as calculated from Equation 4. For lines whose slopes have magnitudes  $|m| > 1$ ,  $\delta y$  can be set proportional to a small vertical deflection voltage with the corresponding horizontal deflection voltage set proportional to  $\delta x$ , calculated from Equation 5. For lines with  $m = 1$ ,  $\delta x = \delta y$  and the horizontal and vertical deflections voltages are equal. In each case, a smooth line with slope  $m$  is generated between the specified endpoints.

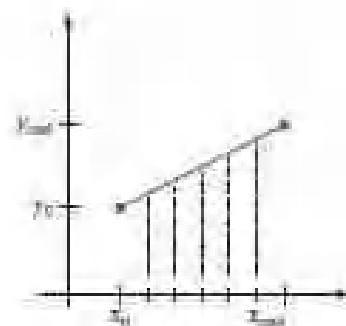


FIGURE 3  
Straight-line segment with five sampling positions along the  $x$  axis between  $x_0$  and  $x_{end}$ .

On raster systems, lines are plotted with pixels, and step sizes in the horizontal and vertical directions are constrained by pixel separations. That is, we must "sample" a line at discrete positions and determine the nearest pixel to the line at each sampled position. This scan-conversion process for straight lines is illustrated in Figure 3 with discrete sample positions along the  $x$  axis.

### DDA Algorithm

The *digital differential analyzer (DDA)* is a scan-conversion line algorithm based on calculating either  $\delta y$  or  $\delta x$ , using Equation 4 or Equation 5. A line is sampled at unit intervals in one coordinate and the corresponding integer values nearest the line path are determined for the other coordinate.

We consider first a line with positive slope, as shown in Figure 2. If the slope is less than or equal to 1, we sample at unit  $x$  intervals ( $\delta x = 1$ ) and compute successive  $y$  values as

$$y_{k+1} = y_k + m \quad (6)$$

Subscript  $k$  takes integer values starting from 0, for the first point, and increases by 1 until the final endpoint is reached. Because  $m$  can be any real number between 0.0 and 1.0, each calculated  $y$  value must be rounded to the nearest integer corresponding to a screen pixel position in the  $x$  column that we are processing.

For lines with a positive slope greater than 1.0, we reverse the roles of  $x$  and  $y$ .

That is, we sample at unit  $y$  intervals ( $\delta y = 1$ ) and calculate consecutive  $x$  values as

$$x_{k+1} = x_k + 1/m \quad (7)$$

In this case, each computed  $x$  value is rounded to the nearest pixel position along the current  $y$  scan line.

Equations 6 and 7 are based on the assumption that lines are to be processed from the left endpoint to the right endpoint (Figure 2). If this processing is reversed, so that the starting endpoint is at the right, then either we have  $\delta x = -1$  and

$$y_{k+1} = y_k - m \quad (8)$$

or

(when the slope is greater than 1) we have  $\delta y = -1$  with

$$x_{k+1} = x_k - 1/m \quad (9)$$

Similar calculations are carried out using Equations 6 through 9 to determine pixel positions along a line with negative slope. Thus, if the absolute value of the slope is less than 1 and the starting endpoint is at the left, we set  $\delta x = 1$  and calculate  $y$  values with Equation 6. When the starting endpoint is at the right (for the same slope), we set  $\delta x = -1$  and obtain  $y$  positions using Equation 8. For a negative slope with absolute value greater than 1, we use  $\delta y = -1$  and Equation 9, or we use  $\delta y = 1$  and Equation 7. This algorithm is summarized in the following procedure, which accepts as input two integer screen positions for the endpoints of a line segment. Horizontal and vertical differences between the endpoint positions are assigned to parameters **dx** and **dy**. The difference with the greater magnitude determines the value of parameter **steps**. This value is the number of pixels that must be drawn beyond the starting pixel; from it, we calculate the  $x$  and  $y$  increments needed to generate the next pixel position at each step along the line path. We draw the starting pixel at position (**x0**, **y0**), and then draw the remaining pixels iteratively, adjusting  $x$  and  $y$  at each step to obtain the next pixel's position before drawing it. If the magnitude of **dx** is greater than the magnitude of **dy** and **x0** is less than **xEnd**, the values for the increments in the  $x$  and  $y$  directions are 1 and  $m$ , respectively. If the greater change is in the  $x$  direction, but **x0** is greater than **xEnd**, then the decrements  $-1$  and  $-m$  are used to generate each new point on the line. Otherwise, we use a unit increment (or decrement) in the  $y$  direction and an  $x$  increment (or decrement) of  $1/m$ .

straight-line segments are to be drawn. The vertical axes show scan-line positions, and the horizontal axes identify pixel columns. Sampling at unit  $x$  intervals in these examples, we need to decide which of two possible pixel positions is closer to the line path at each sample step. Starting from the left endpoint shown in Figure 4, we need to determine at the next sample position whether to plot the pixel at position  $(11, 11)$  or the one at  $(11, 12)$ . Similarly, Figure 5 shows a negative-slope line path starting from the left endpoint at pixel position  $(50, 50)$ . In this one, do we select the next pixel position as  $(51, 50)$  or as  $(51, 49)$ ? These questions are answered with Bresenham's line algorithm by testing the sign of an integer parameter whose value is proportional to the difference between the vertical separations of the two pixel positions from the actual line path.

To illustrate Bresenham's approach, we first consider the scan-conversion process for lines with positive slope less than 1.0. Pixel positions along a line path are then determined by sampling at unit  $x$  intervals. Starting from the left endpoint  $(x_k, y_k)$  of a given line, we step to each successive column ( $x$  position) and plot the pixel whose scan-line  $y$  value is closest to the line path. Figure 6 demonstrates the  $k$ th step in this process. Assuming that we have determined that the pixel at  $(x_k, y_k)$  is to be displayed, we next need to decide which pixel to plot in column  $x_{k+1} = x_k + 1$ . Our choices are the pixels at positions  $(x_k + 1, y_k)$  and  $(x_k + 1, y_k + 1)$ .

At sampling position  $x_k + 1$ , we label vertical pixel separations from the mathematical line path as  $d_{lower}$  and  $d_{upper}$  (Figure 7). The  $y$  coordinate on the mathematical line at pixel column position  $x_k + 1$  is calculated as

$$y = m(x_k + 1) + b \quad (10)$$

Then

$$\begin{aligned} d_{lower} &= y - y_k \\ &= m(x_k + 1) + b - y_k \end{aligned} \quad (11)$$

and

$$\begin{aligned} d_{upper} &= (y_k + 1) - y \\ &= y_k + 1 - m(x_k + 1) - b \end{aligned} \quad (12)$$

To determine which of the two pixels is closest to the line path, we can set up an efficient test that is based on the difference between the two pixel separations as follows:

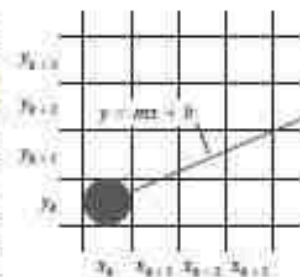
$$d_{lower} - d_{upper} = 2m(x_k + 1) - 2y_k + 2b - 1 \quad (13)$$

A decision parameter  $p_k$  for the  $k$ th step in the line algorithm can be obtained by rearranging Equation 13 so that it involves only integer calculations. We accomplish this by substituting  $m = \Delta y / \Delta x$ , where  $\Delta y$  and  $\Delta x$  are the vertical and horizontal separations of the endpoint positions, and defining the decision parameter as

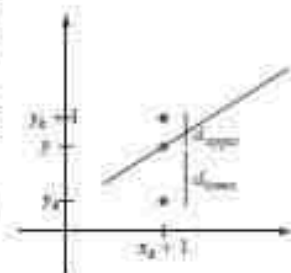
$$\begin{aligned} p_k &= \Delta x(d_{lower} - d_{upper}) \\ &= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c \end{aligned} \quad (14)$$

The sign of  $p_k$  is the same as the sign of  $d_{lower} - d_{upper}$ , because  $\Delta x > 0$  for our example. Parameter  $c$  is constant and has the value  $2\Delta y + \Delta x(2b - 1)$ , which is independent of the pixel position and will be eliminated in the recursive calculations for  $p_k$ . If the pixel at  $y_k$  is "closer" to the line path than the pixel at  $y_k + 1$  (that is,  $d_{lower} < d_{upper}$ ), then decision parameter  $p_k$  is negative. In that case, we plot the lower pixel; otherwise, we plot the upper pixel.

A section of a display screen where a negative slope line segment is to be plotted, starting from the pixel at column 50 on scan line 50.



**FIGURE 6**  
A section of the screen showing a pixel in column  $x_k$  on scan line  $y_k$  that is to be plotted along the path of a line segment with slope  $0 < m < 1$ .



**FIGURE 7**  
Vertical distances between pixel positions and the line  $y$  coordinate at sampling position  $x_k + 1$ .

algorithm, developed by Bresenham, that uses only incremental integer calculations. In addition, Bresenham's line algorithm can be adapted to display circles and other curves. Figures 4 and 5 illustrate sections of a display screen where

Coordinate changes along the line occur in unit steps in either the  $x$  or  $y$  direction. Therefore, we can obtain the values of successive decision parameters using incremental integer calculations. At step  $k + 1$ , the decision parameter is evaluated from Equation 14 as

$$p_{k+1} = 2\Delta y \cdot x_{k+1} - 2\Delta x \cdot y_{k+1} + c$$

Subtracting Equation 14 from the preceding equation, we have

$$p_{k+1} - p_k = 2\Delta y(x_{k+1} - x_k) - 2\Delta x(y_{k+1} - y_k)$$

However,  $x_{k+1} = x_k + 1$ , so that

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k) \quad (15)$$

where the term  $y_{k+1} - y_k$  is either 0 or 1, depending on the sign of parameter  $p_k$ .

This recursive calculation of decision parameters is performed at each integer  $x$  position, starting at the left coordinate endpoint of the line. The first parameter,  $p_0$ , is evaluated from Equation 14 at the starting pixel position  $(x_0, y_0)$  and with  $m$  evaluated as  $\Delta y/\Delta x$  as follows:

$$p_0 = 2\Delta y - \Delta x \quad (16)$$

We summarize Bresenham line drawing for a line with a positive slope less than 1 in the following outline of the algorithm. The constants  $2\Delta y$  and  $2\Delta y - 2\Delta x$  are calculated once for each line to be scan-converted; so the arithmetic involves only integer addition and subtraction of these two constants. Step 4 of the algorithm will be performed a total of  $\Delta x$  times.

#### Bresenham's Line-Drawing Algorithm for $|m| < 1.0$

1. Input the two line endpoints and select the left endpoint as  $(x_0, y_0)$ .
2. Set the color for frame-buffer position  $(x_0, y_0)$ ; i.e., plot the first point.
3. Calculate the constants  $\Delta x$ ,  $\Delta y$ ,  $2\Delta y$ , and  $2\Delta y - 2\Delta x$ , and obtain the starting value for the decision parameter as

$$p_0 = 2\Delta y - \Delta x$$

4. At each  $x_k$  along the line, starting at  $k = 0$ , perform the following test: If  $p_k < 0$ , the next point to plot is  $(x_k + 1, y_k)$  and

$$p_{k+1} = p_k + 2\Delta y$$

Otherwise, the next point to plot is  $(x_k + 1, y_k + 1)$  and

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

5. Repeat step 4  $\Delta x - 1$  more times.

#### EXAMPLE 1 Bresenham Line Drawing

To illustrate the algorithm, we digitize the line with endpoints  $(20, 10)$  and  $(30, 18)$ . This line has a slope of 0.8, with

$$\Delta x = 10, \quad \Delta y = 8$$

The initial decision parameter has the value

$$\begin{aligned} p_0 &= 2\Delta y - \Delta x \\ &= 6 \end{aligned}$$



and the increments for calculating successive decision parameters are:

$$2\Delta y = 16, \quad 2\Delta y - 2\Delta x = -4$$

We plot the initial point  $(x_0, y_0) = (20, 10)$ , and determine successive pixel positions along the line path from the decision parameter as follows:

$k$	$p_k$	$(X_{k+1}, Y_{k+1})$	$k$	$p_k$	$(X_{k+1}, Y_{k+1})$
0	0	(21, 11)	5	6	(26, 15)
1	-2	(22, 12)	6	2	(27, 16)
2	-2	(23, 12)	7	-2	(28, 16)
3	14	(24, 13)	8	14	(29, 17)
4	10	(25, 14)	9	10	(30, 18)

A plot of the pixels generated along this line path is shown in Figure 8.

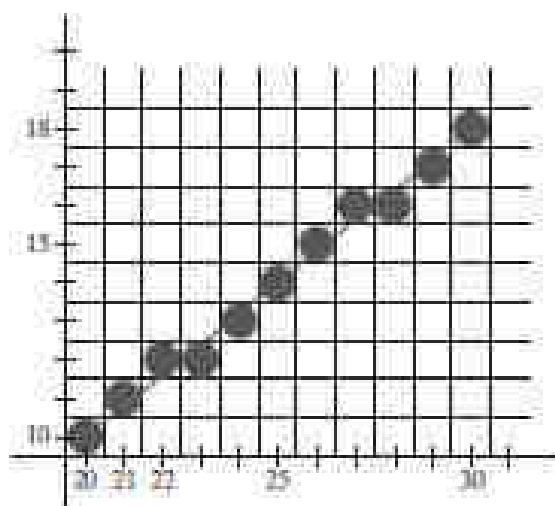


FIGURE 8

Pixel positions along the line path between endpoints  $(20, 10)$  and  $(30, 18)$ , plotted with Bresenham's line algorithm.

An implementation of Bresenham line drawing for slopes in the range  $0 < m < 1.0$  is given in the following procedure. Endpoint pixel positions for the line are passed to this procedure, and pixels are plotted from the left endpoint to the right endpoint.

```

#include <stdlib.h>
#include <math.h>

/* Bresenham line drawing procedure for |m| < 1.0. */
void lineDraw (int x0, int y0, int x1d, int y1d)
{
    int dx = fabs(x1d - x0), dy = fabs(y1d - y0);
    int p = 2 * dy - dx;
    int twody = 2 * dy, twodyminusdx = 2 * (dy - dx);
    int x, y;

    /* Determine which endpoint to use as start position. */
    if (x0 > x1d) {
        x = x1d;
        y = y1d;
        x1d = x0;
    }
}

```

```

else {
    x = x0;
    y = y0;
}
setPixel (x, y);

while (x < xend) {
    x++;
    if (p < 0)
        p += twofy;
    else {
        y++;
        p += twofy - fourdx;
    }
    setPixel (x, y);
}

```

Bresenham's algorithm is generalized to lines with arbitrary slope by considering the symmetry between the various octants and quadrants of the  $xy$  plane. For a line with positive slope greater than 1.0, we interchange the roles of the  $x$  and  $y$  directions. That is, we step along the  $y$  direction in unit steps and calculate successive  $x$  values nearest the line path. Also, we could revise the program to plot pixels starting from either endpoint. If the initial position for a line with positive slope is the right endpoint, both  $x$  and  $y$  decrease as we step from right to left. To ensure that the same pixels are plotted regardless of the starting endpoint, we always choose the upper (or the lower) of the two candidate pixels whenever the two vertical separations from the line path are equal ( $d_{lower} = d_{upper}$ ). For negative slopes, the procedures are similar, except that now one coordinate decreases as the other increases. Finally, special cases can be handled separately: Horizontal lines ( $\Delta y = 0$ ), vertical lines ( $\Delta x = 0$ ), and diagonal lines ( $|\Delta x| = |\Delta y|$ ) can each be loaded directly into the frame buffer without processing them through the line-plotting algorithm.

### Displaying Polylines

Implementation of a polyline procedure is accomplished by invoking a line-drawing routine  $n - 1$  times to display the lines connecting the  $n$  endpoints. Each successive call passes the coordinate pair needed to plot the next line section, where the first endpoint of each coordinate pair is the last endpoint of the previous section. Once the color values for pixel positions along the first line segment have been set in the frame buffer, we process subsequent line segments starting with the next pixel position following the first endpoint for that segment. In this way, we can avoid setting the color of some endpoints twice. We discuss methods for avoiding the overlap of displayed objects in more detail in Section 8.

---

## 2 Parallel Line Algorithms

The line-generating algorithms we have discussed so far determine pixel positions sequentially. Using parallel processing, we can calculate multiple pixel positions along a line path simultaneously by partitioning the computations

among the various processors available. One approach to the partitioning problem is to adapt an existing sequential algorithm to take advantage of multiple processors. Alternatively, we can look for other ways to set up the processing so that pixel positions can be calculated efficiently in parallel. An important consideration in devising a parallel algorithm is to balance the processing load among the available processors.

Given  $n_p$  processors, we can set up a parallel Bresenham line algorithm by subdividing the line path into  $n_p$  partitions and simultaneously generating line segments in each of the subintervals. For a line with slope  $0 < m < 1.0$  and left endpoint coordinate position  $(x_0, y_0)$ , we partition the line along the positive  $x$  direction. The distance between beginning  $x$  positions of adjacent partitions can be calculated as

$$\Delta x_p = \frac{\Delta x + n_p - 1}{n_p} \quad (17)$$

where  $\Delta x$  is the width of the line, and the value for partition width  $\Delta x_p$  is computed using integer division. Numbering the partitions, and the processors, as 0, 1, 2, up to  $n_p - 1$ , we calculate the starting  $x$  coordinate for the  $k$ th partition as

$$x_k = x_0 + k\Delta x_p \quad (18)$$

For example, if we have  $n_p = 4$  processors, with  $\Delta x = 15$ , the width of the partitions is 4 and the starting  $x$  values for the partitions are  $x_0$ ,  $x_0 + 4$ ,  $x_0 + 8$ , and  $x_0 + 12$ . With this partitioning scheme, the width of the last (rightmost) subinterval will be smaller than the others in some cases. In addition, if the line endpoints are not integers, truncation errors can result in variable-width partitions along the length of the line.

To apply Bresenham's algorithm over the partitions, we need the initial value for the  $y$  coordinate and the initial value for the decision parameter in each partition. The change  $\Delta y_p$  in the  $y$  direction over each partition is calculated from the line slope  $m$  and partition width  $\Delta x_p$ :

$$\Delta y_p = m\Delta x_p \quad (19)$$

At the  $k$ th partition, the starting  $y$  coordinate is then

$$y_k = y_0 + \text{round}(k\Delta y_p) \quad (20)$$

The initial decision parameter for Bresenham's algorithm at the start of the  $k$ th subinterval is obtained from Equation 14:

$$p_k = (k\Delta x_p)(2\Delta y) - \text{round}(k\Delta y_p)(2\Delta x) + 2\Delta y - \Delta x \quad (21)$$

Each processor then calculates pixel positions over its assigned subinterval using the preceding starting decision parameter value and the starting coordinates  $(x_k, y_k)$ . Floating-point calculations can be reduced to integer arithmetic in the computations for starting values  $y_k$  and  $p_k$  by substituting  $m = \Delta y/\Delta x$  and rearranging terms. We can extend the parallel Bresenham algorithm to a line with slope greater than 1.0 by partitioning the line in the  $y$  direction and calculating beginning  $x$  values for the partitions. For negative slopes, we increment coordinate values in one direction and decrement in the other.

Another way to set up parallel algorithms on raster systems is to assign each processor to a particular group of screen pixels. With a sufficient number of processors, we can assign each processor to one pixel within some screen region. This

approach can be adapted to a line display by assigning one processor to each of the pixels within the limits of the coordinate extents of the line and calculating pixel distances from the line path. The number of pixels within the bounding box of a line is  $\Delta x \cdot \Delta y$  (as illustrated in Figure 9). Perpendicular distance  $d$  from the line in Figure 9 to a pixel with coordinates  $(x, y)$  is obtained with the calculation

$$d = Ax + By + C \quad (22)$$

where

$$A = \frac{-\Delta y}{\text{linelength}}$$

$$B = \frac{\Delta x}{\text{linelength}}$$

$$C = \frac{x_0 \Delta y - y_0 \Delta x}{\text{linelength}}$$

with

$$\text{linelength} = \sqrt{\Delta x^2 + \Delta y^2}$$

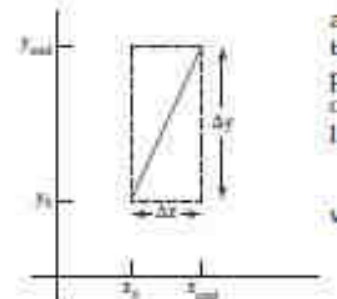


FIGURE 9  
Bounding box for a line with endpoint  
separations  $\Delta x$  and  $\Delta y$ .

Once the constants  $A$ ,  $B$ , and  $C$  have been evaluated for the line, each processor must perform two multiplications and two additions to compute the pixel distance  $d$ . A pixel is plotted if  $d$  is less than a specified line thickness parameter.

Instead of partitioning the screen into single pixels, we can assign to each processor either a scan line or a column of pixels depending on the line slope. Each processor then calculates the intersection of the line with the horizontal row or vertical column of pixels assigned to that processor. For a line with slope  $|m| < 1.0$ , each processor simply solves the line equation for  $y$ , given an  $x$  column value. For a line with slope magnitude greater than 1.0, the line equation is solved for  $x$  by each processor, given a scan line  $y$  value. Such direct methods, although slow on sequential machines, can be performed efficiently using multiple processors.

## CIRCLE GENERATING ALGORITHMS

Because the circle is a frequently used component in pictures and graphs, a procedure for generating either full circles or circular arcs is included in many graphics packages. In addition, sometimes a general function is available in a graphics library for displaying various kinds of curves, including circles and ellipses.

### Properties of Circles

A circle (Figure 11) is defined as the set of points that are all at a given distance  $r$  from a center position  $(x_c, y_c)$ . For any circle point  $(x, y)$ , this distance relationship is expressed by the Pythagorean theorem in Cartesian coordinates as

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \quad (26)$$

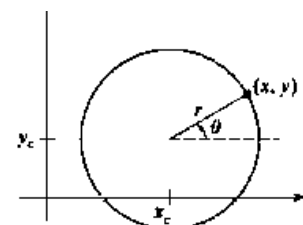


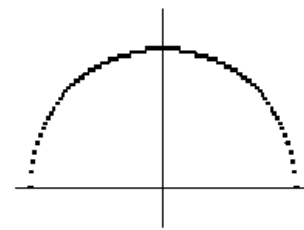
FIGURE 11  
Circle with center coordinates  $(x_c, y_c)$   
and radius  $r$ .

We could use this equation to calculate the position of points on a circle circumference by stepping along the  $x$  axis in unit steps from  $x_c - r$  to  $x_c + r$  and calculating the corresponding  $y$  values at each position as

$$y = y_c \pm \sqrt{r^2 - (x_c - x)^2} \quad (27)$$

However, this is not the best method for generating a circle. One problem with this approach is that it involves considerable computation at each step. Moreover, the spacing between plotted pixel positions is not uniform, as demonstrated in Figure 12. We could adjust the spacing by interchanging  $x$  and  $y$  (stepping through  $y$  values and calculating  $x$  values) whenever the absolute value of the slope of the circle is greater than 1; but this simply increases the computation and processing required by the algorithm.

Another way to eliminate the unequal spacing shown in Figure 12 is to calculate points along the circular boundary using polar coordinates  $r$  and  $\theta$



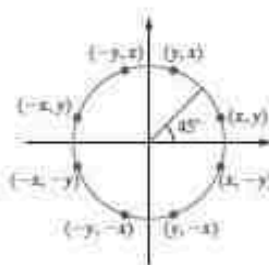
**FIGURE 12**  
Upper half of a circle plotted with Equation 27 and with  $(x_c, y_c) = (0, 0)$ .

(Figure 11). Expressing the circle equation in parametric polar form yields the pair of equations

$$\begin{aligned} x &= x_c + r \cos \theta \\ y &= y_c + r \sin \theta \end{aligned} \quad (28)$$

When a display is generated with these equations using a fixed angular step size, a circle is plotted with equally spaced points along the circumference. To reduce calculations, we can use a large angular separation between points along the circumference and connect the points with straight-line segments to approximate the circular path. For a more continuous boundary on a raster display, we can set the angular step size at  $\frac{1}{7}$ . This plots pixel positions that are approximately one unit apart. Although polar coordinates provide equal point spacing, the trigonometric calculations are still time-consuming.

For any of the previous circle-generating methods, we can reduce computations by considering the symmetry of circles. The shape of the circle is similar in each quadrant. Therefore, if we determine the curve positions in the first quadrant, we can generate the circle section in the second quadrant of the  $xy$  plane by noting that the two circle sections are symmetric with respect to the  $y$  axis. Also, circle sections in the third and fourth quadrants can be obtained from sections in the first and second quadrants by considering symmetry about the  $x$  axis. We can take this one step further and note that there is also symmetry between octants. Circle sections in adjacent octants within one quadrant are symmetric with respect to the  $45^\circ$  line dividing the two octants. These symmetry conditions are illustrated in Figure 13, where a point at position  $(x, y)$  on a one-eighth circle sector is mapped into the seven circle points in the other octants of the  $xy$  plane. Taking advantage of the circle symmetry in this way, we can generate all pixel positions around a circle by calculating only the points within the sector from  $x=0$  to  $x=y$ . The slope of the curve in this octant has a magnitude less than or equal to 1.0. At  $x=0$ , the circle slope is 0, and at  $x=y$ , the slope is  $-1.0$ .



**FIGURE 13**  
Symmetry of a circle. Calculation of a circle point  $(x, y)$  in one octant yields the circle points shown for the other seven octants.

Determining pixel positions along a circle circumference using symmetry and either Equation 26 or Equation 28 still requires a good deal of computation. The Cartesian equation 26 involves multiplications and square-root calculations, while the parametric equations contain multiplications and trigonometric calculations. More efficient circle algorithms are based on incremental calculation of decision parameters, as in the Bresenham line algorithm, which involves only simple integer operations.

Bresenham's line algorithm for raster displays is adapted to circle generation by setting up decision parameters for finding the closest pixel to the circumference at each sampling step. The circle equation 26, however, is nonlinear, so that square-root evaluations would be required to compute pixel distances from a circular path. Bresenham's circle algorithm avoids these square-root calculations by comparing the squares of the pixel separation distances.

However, it is possible to perform a direct distance comparison without a squaring operation. The basic idea in this approach is to test the halfway position between two pixels to determine if this midpoint is inside or outside the circle boundary. This method is applied more easily to other conics, and for an integer circle radius, the midpoint approach generates the same pixel positions as the Bresenham circle algorithm. For a straight-line segment, the midpoint method is equivalent to the Bresenham line algorithm. Also, the error involved in locating pixel positions along any conic section using the midpoint test is limited to half the pixel separation.



### Midpoint Circle Algorithm

As in the raster line algorithm, we sample at unit intervals and determine the closest pixel position to the specified circle path at each step. For a given radius  $r$  and screen center position  $(x_c, y_c)$ , we can first set up our algorithm to calculate pixel positions around a circle path centered at the coordinate origin  $(0, 0)$ . Then each calculated position  $(x, y)$  is moved to its proper screen position by adding  $x_c$  to  $x$  and  $y_c$  to  $y$ . Along the circle section from  $x = 0$  to  $x = y$  in the first quadrant, the slope of the curve varies from 0 to  $-1.0$ . Therefore, we can take unit steps in the positive  $x$  direction over this octant and use a decision parameter to determine which of the two possible pixel positions in any column is vertically closer to the circle path. Positions in the other seven octants are then obtained by symmetry.

To apply the midpoint method, we define a circle function as

$$f_{\text{circ}}(x, y) = x^2 + y^2 - r^2 \quad (29)$$

Any point  $(x, y)$  on the boundary of the circle with radius  $r$  satisfies the equation  $f_{\text{circ}}(x, y) = 0$ . If the point is in the interior of the circle, the circle function is negative, and if the point is outside the circle, the circle function is positive. To summarize, the relative position of any point  $(x, y)$  can be determined by checking the sign of the circle function as follows:

$$f_{\text{circ}}(x, y) \begin{cases} < 0, & \text{if } (x, y) \text{ is inside the circle boundary} \\ = 0, & \text{if } (x, y) \text{ is on the circle boundary} \\ > 0, & \text{if } (x, y) \text{ is outside the circle boundary} \end{cases} \quad (30)$$

The tests in 30 are performed for the midpositions between pixels near the circle path at each sampling step. Thus, the circle function is the decision parameter in the midpoint algorithm, and we can set up incremental calculations for this function as we did in the line algorithm.

Figure 14 shows the midpoint between the two candidate pixels at sampling position  $x_k + 1$ . Assuming that we have just plotted the pixel at  $(x_k, y_k)$ , we next need to determine whether the pixel at position  $(x_k + 1, y_k)$  or the one at position  $(x_k + 1, y_k - 1)$  is closer to the circle. Our decision parameter is the circle function 29 evaluated at the midpoint between these two pixels:

$$\begin{aligned} p_k &= f_{\text{circ}}\left(x_k + 1, y_k - \frac{1}{2}\right) \\ &= (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2 \end{aligned} \quad (31)$$

If  $p_k < 0$ , this midpoint is inside the circle and the pixel on scan line  $y_k$  is closer to the circle boundary. Otherwise, the midpoint is outside or on the circle boundary, and we select the pixel on scan line  $y_k - 1$ .

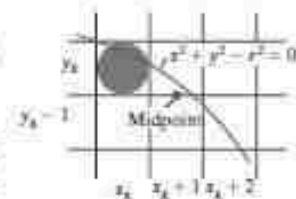
Successive decision parameters are obtained using incremental calculations. We obtain a recursive expression for the next decision parameter by evaluating the circle function at sampling position  $x_{k+1} + 1 = x_k + 2$ :

$$\begin{aligned} p_{k+1} &= f_{\text{circ}}\left(x_{k+1} + 1, y_{k+1} - \frac{1}{2}\right) \\ &= [(x_k + 1) + 1]^2 + \left(y_{k+1} - \frac{1}{2}\right)^2 - r^2 \end{aligned}$$

or

$$p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1 \quad (32)$$

where  $y_{k+1}$  is either  $y_k$  or  $y_k - 1$ , depending on the sign of  $p_k$ .



**FIGURE 14**  
Midpoint between candidate pixels at sampling position  $x_k + 1$  along a circular path.

Increments for obtaining  $p_{k+1}$  are either  $2x_{k+1} + 1$  (if  $p_k$  is negative) or  $2x_{k+1} + 1 - 2y_{k+1}$ . Evaluation of the terms  $2x_{k+1}$  and  $2y_{k+1}$  can also be done incrementally as

$$2x_{k+1} = 2x_k + 2$$

$$2y_{k+1} = 2y_k - 2$$

At the start position  $(0, r)$ , these two terms have the values 0 and  $2r$ , respectively. Each successive value for the  $2x_{k+1}$  term is obtained by adding 2 to the previous value, and each successive value for the  $2y_{k+1}$  term is obtained by subtracting 2 from the previous value.

The initial decision parameter is obtained by evaluating the circle function at the start position  $(x_0, y_0) = (0, r)$ :

$$\begin{aligned} p_0 &= f_{\text{in}}\left(1, r - \frac{1}{2}\right) \\ &= 1 + \left(r - \frac{1}{2}\right)^2 - r^2 \end{aligned}$$

or

$$p_0 = \frac{5}{4} - r \quad (33)$$

If the radius  $r$  is specified as an integer, we can simply round  $p_0$  to

$$p_0 = 1 - r \quad (\text{for } r \text{ an integer})$$

because all increments are integers.

As in Bresenham's line algorithm, the midpoint method calculates pixel positions along the circumference of a circle using integer additions and subtractions, assuming that the circle parameters are specified in integer screen coordinates. We can summarize the steps in the midpoint circle algorithm as follows:

### Midpoint Circle Algorithm

1. Input radius  $r$  and circle center  $(x_c, y_c)$ , then set the coordinates for the first point on the circumference of a circle centered on the origin as

$$(x_0, y_0) = (0, r)$$

2. Calculate the initial value of the decision parameter as

$$p_0 = \frac{5}{4} - r$$

3. At each  $x_k$  position, starting at  $k = 0$ , perform the following test: If  $p_k < 0$ , the next point along the circle centered on  $(0, 0)$  is  $(x_{k+1}, y_k)$  and

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

Otherwise, the next point along the circle is  $(x_k + 1, y_k - 1)$  and

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$

where  $2x_{k+1} = 2x_k + 2$  and  $2y_{k+1} = 2y_k - 2$ .

4. Determine symmetry points in the other seven octants.
5. Move each calculated pixel position  $(x, y)$  onto the circular path centered at  $(x_c, y_c)$  and plot the coordinate values as follows:

$$x = x + x_c \quad y = y + y_c$$

6. Repeat steps 3 through 5 until  $x \geq y$ .

### EXAMPLE 2 Midpoint Circle Drawing

Given a circle radius  $r = 10$ , we demonstrate the midpoint circle algorithm by determining positions along the circle octant in the first quadrant from  $x = 0$  to  $x = y$ . The initial value of the decision parameter is

$$p_0 = 1 - r = -9$$

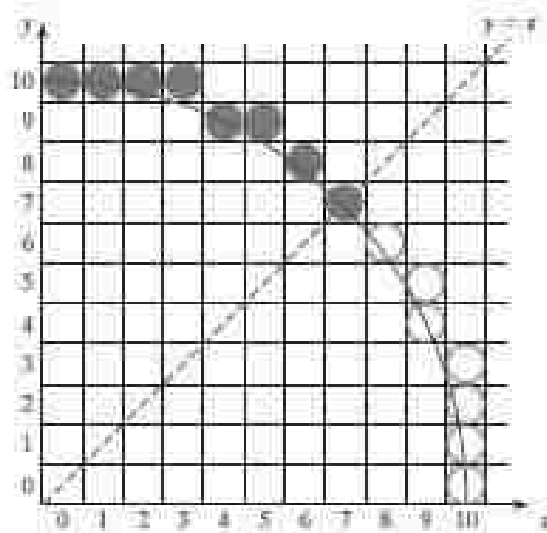
For the circle centered on the coordinate origin, the initial point is  $(x_0, y_0) = (0, 10)$ , and initial increment terms for calculating the decision parameters are

$$2x_0 = 0, \quad 2y_0 = 20$$

Successive midpoint decision parameter values and the corresponding coordinate positions along the circle path are listed in the following table:

$k$	$p_k$	$(x_{k+1}, y_{k+1})$	$2x_{k+1}$	$2y_{k+1}$
0	-9	(1, 10)	2	20
1	-6	(2, 10)	4	20
2	-1	(3, 10)	6	20
3	6	(4, 9)	8	18
4	3	(5, 9)	10	18
5	8	(6, 8)	12	16
6	5	(7, 7)	14	14

A plot of the generated pixel positions in the first quadrant is shown in Figure 15.



**FIGURE 15**  
Pixel positions (solid circles) along a circle path centered on the origin and with radius  $r = 10$ , as calculated by the midpoint circle algorithm. Open ("hollow") circles show the symmetry positions in the first quadrant.

The following code segment illustrates procedures that could be used to implement the midpoint circle algorithm. Values for a circle radius and for the center coordinates of the circle are passed to procedure `circleMidpoint`. A pixel position along the circular path in the first octant is then computed and passed to procedure `circlePlotPoints`. This procedure sets the circle color in the frame buffer for all circle symmetry positions with repeated calls to the `setPixel` routine, which is implemented with the OpenGL point-plotting functions.

```

#include <GL/glut.h>

class screenPt
{
private:
    GLint x, y;

public:
    /* Default Constructor: initializes coordinate position to (0, 0). */
    screenPt ( ) {
        x = y = 0;
    }
    void setCoords (GLint xCoordValue, GLint yCoordValue) {
        x = xCoordValue;
        y = yCoordValue;
    }

    GLint getX ( ) const {
        return x;
    }

    GLint getY ( ) const {
        return y;
    }
    void incrementx ( ) {
        x++;
    }
    void decrementy ( ) {
        y--;
    }
};

void setPixel (GLint xCoord, GLint yCoord)
{
    glBegin (GL_POINTS);
    glVertex2i (xCoord, yCoord);
    glEnd ( );
}

void circleMidpoint (GLint xc, GLint yc, GLint radius)
{
    screenPt circPt;

    GLint p = 1 - radius; // Initial value for midpoint parameter.

    circPt.setCoords (0, radius); // Set coordinates for top point of circle.

    void circlePlotPoints (GLint, GLint, screenPt);
    /* Plot the initial point in each circle quadrant. */
    circlePlotPoints (xc, yc, circPt);
    /* Calculate next point and plot in each octant. */
}

```

```

while (circPt.getx ( ) < circPt.gety ( )) {
    circPt.incrementx ( );
    if (p < 0)
        p += 1 * circPt.getx ( ) + 1;
    else {
        circPt.decrementy ( );
        p += 1 * (circPt.getx ( ) - circPt.gety ( )) + 1;
    }
    circlePlotPoints (xc, yc, circPt);
}

void circlePlotPoints (GLint xc, GLint yc, screenPt circPt)
{
    setPixel (xc + circPt.getx ( ), yc + circPt.gety ( ));
    setPixel (xc - circPt.getx ( ), yc + circPt.gety ( ));
    setPixel (xc + circPt.getx ( ), yc - circPt.gety ( ));
    setPixel (xc - circPt.getx ( ), yc - circPt.gety ( ));
    setPixel (xc + circPt.getx ( ), yc + circPt.getx ( ));
    setPixel (xc - circPt.getx ( ), yc + circPt.getx ( ));
    setPixel (xc + circPt.gety ( ), yc - circPt.getx ( ));
    setPixel (xc - circPt.gety ( ), yc - circPt.getx ( ));
}

```

## ELLIPSE GENERATING ALGORITHMS

Loosely stated, an ellipse is an elongated circle. We can also describe an ellipse as a modified circle whose radius varies from a maximum value in one direction to a minimum value in the perpendicular direction. The straight-line segments through the interior of the ellipse in these two perpendicular directions are referred to as the *major* and *minor* axes of the ellipse.

### Properties of Ellipses

A precise definition of an ellipse can be given in terms of the distances from any point on the ellipse to two fixed positions, called the foci of the ellipse. The sum of these two distances is the same value for all points on the ellipse (Figure 16). If the distances to the two focus positions from any point  $P = (x, y)$  on the ellipse are labeled  $d_1$  and  $d_2$ , then the general equation of an ellipse can be stated as

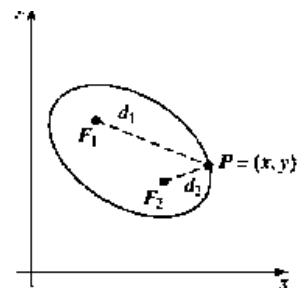
$$d_1 + d_2 = \text{constant} \quad (34)$$

Expressing distances  $d_1$  and  $d_2$  in terms of the focal coordinates  $F_1 = (x_1, y_1)$  and  $F_2 = (x_2, y_2)$ , we have

$$\sqrt{(x - x_1)^2 + (y - y_1)^2} + \sqrt{(x - x_2)^2 + (y - y_2)^2} = \text{constant} \quad (35)$$

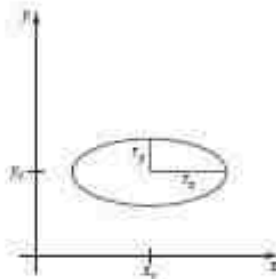
By squaring this equation, isolating the remaining radical, and squaring again, we can rewrite the general ellipse equation in the form

$$Ax^2 + By^2 + Cxy + Dx + Ey + F = 0 \quad (36)$$

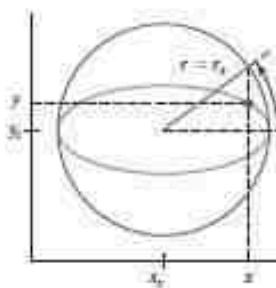


**FIGURE 16**  
Ellipse generated about foci  $F_1$  and  $F_2$ .

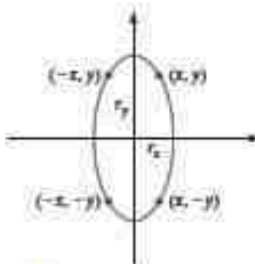




**FIGURE 17**  
Ellipse centered at  $(x_c, y_c)$  with semimajor axis  $r_x$  and semiminor axis  $r_y$ .



**FIGURE 18**  
The bounding circle and eccentric angle  $\theta$  for an ellipse with  $r_x > r_y$ .



**FIGURE 19**  
Symmetry of an ellipse. Calculation of a point  $(x, y)$  in one quadrant yields the ellipse points shown for the other three quadrants.

where the coefficients  $A$ ,  $B$ ,  $C$ ,  $D$ ,  $E$ , and  $F$  are evaluated in terms of the focal coordinates and the dimensions of the major and minor axes of the ellipse. The major axis is the straight-line segment extending from one side of the ellipse to the other through the foci. The minor axis spans the shorter dimension of the ellipse, perpendicularly bisecting the major axis at the halfway position (ellipse center) between the two foci.

An interactive method for specifying an ellipse in an arbitrary orientation is to input the two foci and a point on the ellipse boundary. With these three coordinate positions, we can evaluate the constant in Equation 35. Then, the values for the coefficients in Equation 36 can be computed and used to generate pixels along the elliptical path.

Ellipse equations are greatly simplified if the major and minor axes are oriented to align with the coordinate axes. In Figure 17, we show an ellipse in "standard position," with major and minor axes oriented parallel to the  $x$  and  $y$  axes. Parameter  $r_x$  for this example labels the semimajor axis, and parameter  $r_y$  labels the semiminor axis. The equation for the ellipse shown in Figure 17 can be written in terms of the ellipse center coordinates and parameters  $r_x$  and  $r_y$  as

$$\left(\frac{x-x_c}{r_x}\right)^2 + \left(\frac{y-y_c}{r_y}\right)^2 = 1 \quad (37)$$

Using polar coordinates  $r$  and  $\theta$ , we can also describe the ellipse in standard position with the parametric equations

$$\begin{aligned} x &= x_c + r_x \cos \theta \\ y &= y_c + r_y \sin \theta \end{aligned} \quad (38)$$

Angle  $\theta$ , called the *eccentric angle* of the ellipse, is measured around the perimeter of a bounding circle. If  $r_x > r_y$ , the radius of the bounding circle is  $r = r_x$  (Figure 18). Otherwise, the bounding circle has radius  $r = r_y$ .

As with the circle algorithm, symmetry considerations can be used to reduce computations. An ellipse in standard position is symmetric between quadrants, but, unlike a circle, it is not symmetric between the two octants of a quadrant. Thus, we must calculate pixel positions along the elliptical arc throughout one quadrant, then use symmetry to obtain curve positions in the remaining three quadrants (Figure 19).

### Midpoint Ellipse Algorithm

Our approach here is similar to that used in displaying a raster circle. Given parameters  $r_x$ ,  $r_y$ , and  $(x_c, y_c)$ , we determine curve positions  $(x, y)$  for an ellipse in standard position centered on the origin, then we shift all the points using a fixed offset so that the ellipse is centered at  $(x_c, y_c)$ . If we wish also to display the ellipse in nonstandard position, we could rotate the ellipse about its center coordinates to reorient the major and minor axes in the desired directions. For the present, we consider only the display of ellipses in standard position.

The midpoint ellipse method is applied throughout the first quadrant in two parts. Figure 20 shows the division of the first quadrant according to the slope of an ellipse with  $r_x < r_y$ . We process this quadrant by taking unit steps in the  $x$  direction where the slope of the curve has a magnitude less than 1.0, and then we take unit steps in the  $y$  direction where the slope has a magnitude greater than 1.0.

At the next sampling position ( $x_{k+1} + 1 = x_k + 2$ ), the decision parameter for region 1 is evaluated as

$$\begin{aligned} p1_{k+1} &= f_{\text{ellipse}}\left(x_{k+1} + 1, y_{k+1} - \frac{1}{2}\right) \\ &= r_y^2[(x_k + 1) + 1]^2 + r_x^2\left(y_{k+1} - \frac{1}{2}\right)^2 - r_x^2 r_y^2 \end{aligned}$$

or

$$p1_{k+1} = p1_k + 2r_y^2(x_k + 1) + r_y^2 + r_x^2 \left[ \left(y_{k+1} - \frac{1}{2}\right)^2 - \left(y_k - \frac{1}{2}\right)^2 \right] \quad (44)$$

where  $y_{k+1}$  is either  $y_k$  or  $y_k - 1$ , depending on the sign of  $p1_k$ .

Decision parameters are incremented by the following amounts:

$$\text{increment} = \begin{cases} 2r_y^2 x_{k+1} + r_y^2, & \text{if } p1_k < 0 \\ 2r_y^2 x_{k+1} + r_y^2 - 2r_x^2 y_{k+1}, & \text{if } p1_k \geq 0 \end{cases}$$

Increments for the decision parameters can be calculated using only addition and subtraction, as in the circle algorithm, because values for the terms  $2r_y^2 x$  and  $2r_x^2 y$  can be obtained incrementally. At the initial position  $(0, r_y)$ , these two terms evaluate to

$$2r_y^2 x = 0 \quad (45)$$

$$2r_x^2 y = 2r_x^2 r_y \quad (46)$$

As  $x$  and  $y$  are incremented, updated values are obtained by adding  $2r_y^2$  to the current value of the increment term in Equation 45 and subtracting  $2r_x^2$  from the current value of the increment term in Equation 46. The updated increment values are compared at each step, and we move from region 1 to region 2 when condition 42 is satisfied.

In region 1, the initial value of the decision parameter is obtained by evaluating the ellipse function at the start position  $(x_0, y_0) = (0, r_y)$ :

$$\begin{aligned} p1_0 &= f_{\text{ellipse}}\left(0, r_y - \frac{1}{2}\right) \\ &= r_y^2 + r_x^2\left(r_y - \frac{1}{2}\right)^2 - r_x^2 r_y^2 \end{aligned}$$

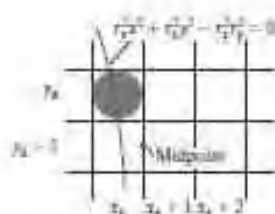
or

$$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2 \quad (47)$$

Over region 2, we sample at unit intervals in the negative  $y$  direction, and the midpoint is now taken between horizontal pixels at each step (Figure 22). For this region, the decision parameter is evaluated as

$$\begin{aligned} p2_k &= f_{\text{ellipse}}\left(x_k + \frac{1}{2}, y_k - 1\right) \\ &= r_y^2\left(x_k + \frac{1}{2}\right)^2 + r_x^2(y_k - 1)^2 - r_x^2 r_y^2 \end{aligned} \quad (48)$$

If  $p2_k > 0$ , the midpoint is outside the ellipse boundary, and we select the pixel at  $x_k$ . If  $p2_k \leq 0$ , the midpoint is inside or on the ellipse boundary, and we select pixel position  $x_{k+1}$ .



**FIGURE 22**  
Midpoint between candidate pixels at sampling position  $p_k - 1$  along an elliptical path.

Regions 1 and 2 (Figure 20) can be processed in various ways. We can start at position  $(0, r_y)$  and step clockwise along the elliptical path in the first quadrant, shifting from unit steps in  $x$  to unit steps in  $y$  when the slope becomes less than  $-1.0$ . Alternatively, we could start at  $(r_x, 0)$  and select points in a counterclockwise order, shifting from unit steps in  $y$  to unit steps in  $x$  when the slope becomes greater than  $-1.0$ . With parallel processors, we could calculate pixel positions in the two regions simultaneously. As an example of a sequential implementation of the midpoint algorithm, we take the start position at  $(0, r_y)$  and step along the ellipse path in clockwise order throughout the first quadrant.

We define an ellipse function from Equation 37 with  $(x, y) \neq (0, 0)$  as:

$$f_{\text{ellipse}}(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2 \quad (39)$$

which has the following properties:

$$f_{\text{ellipse}}(x, y) \begin{cases} < 0, & \text{if } (x, y) \text{ is inside the ellipse boundary} \\ = 0, & \text{if } (x, y) \text{ is on the ellipse boundary} \\ > 0, & \text{if } (x, y) \text{ is outside the ellipse boundary} \end{cases} \quad (40)$$

Thus, the ellipse function,  $f_{\text{ellipse}}(x, y)$  serves as the decision parameter in the midpoint algorithm. At each sampling position, we select the next pixel along the ellipse path according to the sign of the ellipse function evaluated at the midpoint between the two candidate pixels.

Starting at  $(0, r_y)$ , we take unit steps in the  $x$  direction until we reach the boundary between region 1 and region 2 (Figure 20). Then we switch to unit steps in the  $y$  direction over the remainder of the curve in the first quadrant. At each step we need to test the value of the slope of the curve. The ellipse slope is calculated from Equation 39 as:

$$\frac{dy}{dx} = -\frac{2r_y^2 x}{2r_x^2 y} \quad (41)$$

At the boundary between region 1 and region 2,  $dy/dx = -1.0$  and

$$2r_y^2 x = 2r_x^2 y$$

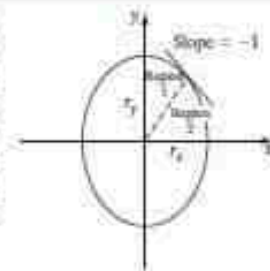
Therefore, we move out of region 1 whenever

$$2r_y^2 x \geq 2r_x^2 y \quad (42)$$

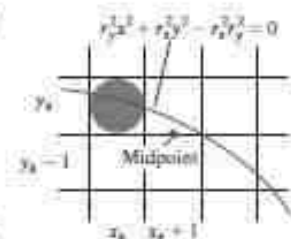
Figure 21 shows the midpoint between the two candidate pixels at sampling position  $x_k + 1$  in the first region. Assuming position  $(x_k, y_k)$  has been selected in the previous step, we determine the next position along the ellipse path by evaluating the decision parameter (that is, the ellipse function 39) at this midpoint:

$$\begin{aligned} p1_k &= f_{\text{ellipse}}\left(x_k + 1, y_k - \frac{1}{2}\right) \\ &= r_y^2 (x_k + 1)^2 + r_x^2 \left(y_k - \frac{1}{2}\right)^2 - r_x^2 r_y^2 \end{aligned} \quad (43)$$

If  $p1_k < 0$ , the midpoint is inside the ellipse and the pixel on scan line  $y_k$  is closer to the ellipse boundary. Otherwise, the midpoint is outside or on the ellipse boundary, and we select the pixel on scan line  $y_k - 1$ .



**FIGURE 20**  
Ellipse processing regions. Over region 1, the magnitude of the ellipse slope is less than 1.0; over region 2, the magnitude of the slope is greater than 1.0.



**FIGURE 21**  
Midpoint between candidate pixels at sampling position  $x_k + 1$  along an elliptical path.



To determine the relationship between successive decision parameters in region 2, we evaluate the ellipse function at the next sampling step  $y_{k+1} - 1 = y_k - 2$ :

$$\begin{aligned} p2_{k+1} &= f_{\text{ellipse}}\left(x_{k+1} + \frac{1}{2}, y_{k+1} - 1\right) \\ &= r_y^2\left(x_{k+1} + \frac{1}{2}\right)^2 + r_x^2[(y_k - 1) - 1]^2 - r_x^2 r_y^2 \end{aligned} \quad (49)$$

or

$$p2_{k+1} = p2_k - 2r_x^2(y_k - 1) + r_x^2 + r_y^2\left[\left(x_{k+1} + \frac{1}{2}\right)^2 - \left(x_k + \frac{1}{2}\right)^2\right] \quad (50)$$

with  $x_{k+1}$  set either to  $x_k$  or to  $x_k + 1$ , depending on the sign of  $p2_k$ .

When we enter region 2, the initial position  $(x_0, y_0)$  is taken as the last position selected in region 1 and the initial decision parameter in region 2 is then

$$\begin{aligned} p2_0 &= f_{\text{ellipse}}\left(x_0 + \frac{1}{2}, y_0 - 1\right) \\ &= r_y^2\left(x_0 + \frac{1}{2}\right)^2 + r_x^2(y_0 - 1)^2 - r_x^2 r_y^2 \end{aligned} \quad (51)$$

To simplify the calculation of  $p2_0$ , we could select pixel positions in counterclockwise order starting at  $(r_x, 0)$ . Unit steps would then be taken in the positive  $y$  direction up to the last position selected in region 1.

This midpoint algorithm can be adapted to generate an ellipse in nonstandard position using the ellipse function Equation 36 and calculating pixel positions over the entire elliptical path. Alternatively, we could reorient the ellipse axes to standard position, apply the midpoint ellipse algorithm to determine curve positions, and then convert calculated pixel positions to path positions along the original ellipse orientation.

Assuming  $r_x, r_y$ , and the ellipse center are given in integer screen coordinates, we need only incremental integer calculations to determine values for the decision parameters in the midpoint ellipse algorithm. The increments  $r_x^2, r_y^2, 2r_x^2$ , and  $2r_y^2$  are evaluated once at the beginning of the procedure. In the following summary, we list the steps for displaying an ellipse using the midpoint algorithm:

### Midpoint Ellipse Algorithm

1. Input  $r_x, r_y$ , and ellipse center  $(x_c, y_c)$ , and obtain the first point on an ellipse centered on the origin as

$$(x_0, y_0) = (0, r_y)$$

2. Calculate the initial value of the decision parameter in region 1 as

$$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2$$

3. At each  $x_k$  position in region 1, starting at  $k = 0$ , perform the following test: If  $p1_k < 0$ , the next point along the ellipse centered on  $(0, 0)$  is  $(x_{k+1}, y_k)$  and

$$p1_{k+1} = p1_k + 2r_x^2 x_{k+1} + r_x^2$$

Otherwise, the next point along the ellipse is  $(x_k + 1, y_k - 1)$  and

$$p1_{k+1} = p1_k + 2r_x^2 x_{k+1} - 2r_y^2 y_{k+1} + r_x^2$$

$$2r_y^2x_{k+1} = 2r_y^2x_k + 2r_y^2 \quad 2r_x^2y_{k+1} = 2r_x^2y_k - 2r_x^2$$

and continue until  $2r_y^2x \geq 2r_x^2y$

4. Calculate the initial value of the decision parameter in region 2 as

$$p2_0 = r_x^2 \left( y_0 + \frac{1}{2} \right)^2 + r_y^2 (x_0 - 1)^2 - r_x^2 r_y^2$$

where  $(x_0, y_0)$  is the last position calculated in region 1.

5. At each  $y_k$  position in region 2, starting at  $k = 0$ , perform the following test: If  $p2_k > 0$ , the next point along the ellipse centered on  $(0, 0)$  is  $(x_k, y_k - 1)$  and

$$p2_{k+1} = p2_k - 2r_x^2y_{k+1} + r_x^2$$

Otherwise, the next point along the ellipse is  $(x_k + 1, y_k - 1)$  and

$$p2_{k+1} = p2_k + 2r_y^2x_{k+1} - 2r_x^2y_{k+1} + r_x^2$$

using the same incremental calculations for  $x$  and  $y$  as in region 1. Continue until  $y = 0$ .

6. For both regions, determine symmetry points in the other three quadrants.

7. Move each calculated pixel position  $(x, y)$  onto the elliptical path centered on  $(x_0, y_0)$  and plot these coordinate values

$$x = x + x_0 \quad y = y + y_0$$

### EXAMPLE 3 Midpoint Ellipse Drawing

Given input ellipse parameters  $r_x = 8$  and  $r_y = 6$ , we illustrate the steps in the midpoint ellipse algorithm by determining raster positions along the ellipse path in the first quadrant. Initial values and increments for the decision parameter calculations are

$$2r_y^2x = 0 \quad (\text{with increment } 2r_y^2 = 72)$$

$$2r_x^2y = 2r_x^2r_y \quad (\text{with increment } -2r_x^2 = -128)$$

For region 1, the initial point for the ellipse centered on the origin is  $(x_0, y_0) = (0, 6)$ , and the initial decision parameter value is

$$p1_0 = r_y^2 - r_x^2r_y + \frac{1}{4}r_x^2 = -332$$

Successive midpoint decision-parameter values and the pixel positions along the ellipse are listed in the following table:

$k$	$p1_k$	$(x_{k+1}, y_{k+1})$	$2r_y^2x_{k+1}$	$2r_x^2y_{k+1}$
0	-332	(1, 6)	72	768
1	-224	(2, 6)	144	768
2	-44	(3, 6)	216	768
3	208	(4, 5)	288	640
4	-108	(5, 5)	360	640
5	288	(6, 4)	432	512
6	244	(7, 3)	504	384



We now move out of region 1 because  $2r_y^2x > 2r_x^2y$ .

For region 2, the initial point is  $(x_0, y_0) = (7, 3)$  and the initial decision parameter is

$$p2_0 = f_{\text{ellipse}}\left(7 + \frac{1}{2}, 2\right) = -151$$

The remaining positions along the ellipse path in the first quadrant are then calculated as

k	$p1_k$	$(x_{k+1}, y_{k+1})$	$2r_y^2x_{k+1}$	$2r_x^2y_{k+1}$
0	-151	(8, 2)	576	256
1	233	(8, 1)	576	128
2	745	(8, 0)	—	—

A plot of the calculated positions for the ellipse within the first quadrant is shown in Figure 23.

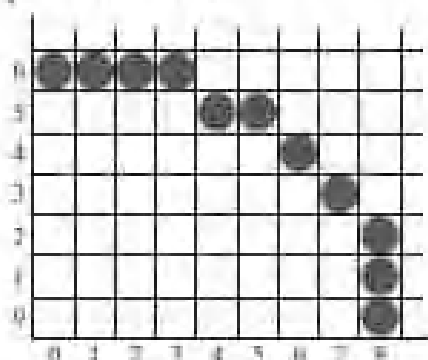


FIGURE 23

Pixel positions along an elliptical path centered on the origin with  $r_x = 8$  and  $r_y = 6$ , using the midpoint algorithm to calculate iterations within the first quadrant.

In the following code segment, example procedures are given for implementing the midpoint ellipse algorithm. Values for the ellipse parameters  $R_x$ ,  $R_y$ ,  $xCenter$ , and  $yCenter$  are input to procedure `ellipseMidpoint`. Positions along the curve in the first quadrant are then calculated and passed to procedure `ellipsePlotPoints`. Symmetry is used to obtain ellipse positions in the other three quadrants, and the `setPixel` routine sets the ellipse color in the frame-buffer locations corresponding to these positions.

```

inline int round (const float a) { return int (a + 0.5); }

/* The following procedure accepts values for an ellipse
 * center position and its semimajor and semiminor axes. Then
 * calculates ellipse positions using the midpoint algorithm.
 */
void ellipseMidpoint (int xCenter, int yCenter, int Rx, int Ry)
{
    int Rx2 = Rx * Rx;
    int Ry2 = Ry * Ry;
    int roundX = 1 * Rx2;
    int roundY = 1 * Ry2;
    int p1;
    int x = 0;
    int y = Ry;
    int pa = 0;
    int py = roundX * y;
    void ellipsePlotPoints (int, int, int, int);
}

```

```

/* Plot the initial point in each quadrant. */
ellipsePlotPoints (xCenter, yCenter, x, y);

/* Region 1 */
p = count (Ry2 - (Ra2 * Ry) + (0.25 * Ra2));
while (p < py) {
  x++;
  px += twofly2;
  if (p < 0)
    p += Ry2 + px;
  else {
    y--;
    py -= twofly2;
    p += Ry2 + px - py;
  }
  ellipsePlotPoints (xCenter, yCenter, x, y);
}

/* Region 2 */
p = count (Ry2 * (x+0.5) * (x+0.5) + Ra2 * (y-1) * (y-1) - Ra2 * (Ry2));
while (y > 0) {
  y--;
  py -= twofly2;
  if (p > 0)
    p += Ra2 - py;
  else {
    x++;
    px += twofly2;
    p += Ra2 - py + px;
  }
  ellipsePlotPoints (xCenter, yCenter, x, y);
}
}

void ellipsePlotPoints (int xCenter, int yCenter, int x, int y) {
  setPixel (xCenter + x, yCenter + y);
  setPixel (xCenter - x, yCenter + y);
  setPixel (xCenter + x, yCenter - y);
  setPixel (xCenter - x, yCenter - y);
}

```

## ATTRIBUTES OF OUTPUT PRIMITIVES

In general, a parameter that affects the way a primitive is to be displayed is referred to as an **attribute parameter**. Some attribute parameters, such as color and size, determine the fundamental characteristics of a primitive. Other attributes specify how the primitive is to be displayed under special conditions.

Examples of special-condition attributes are the options such as visibility or detectability within an interactive object-selection program. These special-condition attributes are explored in later chapters. Here, we treat only those attributes that control the basic display properties of graphics primitives, without regard for special situations. For example, lines can be dotted or dashed, fat or thin, and blue or orange. Areas might be filled with one color or with a multicolor pattern. Text can appear reading from left to

right, slanted diagonally across the screen, or in vertical columns. Individual characters can be displayed in different fonts, colors, and sizes. And we can apply intensity variations at the edges of objects to smooth out the raster stair-step effect. One way to incorporate attribute options into a graphics package is to extend the parameter list associated with each graphics-primitive function to include the appropriate attribute values. A line-drawing function, for example, could contain additional parameters to set the color, width, and other properties of a line. Another approach is to maintain a system list of current attribute values. Separate functions are then included in the graphics package for setting the current values in the attribute list. To generate a primitive, the system checks the relevant attributes and invokes the display routine for that primitive using the current attribute settings. Some graphics packages use a combination of methods for setting attribute values, and other libraries, including OpenGL, assign attributes using separate functions that update a system attribute list.

A graphics system that maintains a list for the current values of attributes and other parameters is referred to as a **state system** or **state machine**. Attributes of output primitives and some other parameters, such as the current frame-buffer position, are referred to as **state variables** or **state parameters**. When we assign a value to one or more state parameters, we put the system into a particular state, and that state remains in effect until we change the value of a state parameter.

#### **LINE ATTRIBUTES**

A straight-line segment can be displayed with three basic attributes: color, width, and style. Line color is typically set with the same function for all graphics primitives, while line width and line style are selected with separate line functions. In addition, lines may be generated with other effects, such as pen and brush strokes.

##### **Line Width**

Implementation of line-width options depends on the capabilities of the output device. A heavy line could be displayed on a video monitor as adjacent parallel lines, while a pen plotter might require pen changes to draw a thick line.

For raster implementations, a standard-width line is generated with single pixels at each sample position, as in the Bresenham algorithm. Thicker lines are displayed as positive integer multiples of the standard line by plotting additional pixels along adjacent parallel line paths.

##### **Line Style**

Possible selections for the line-style attribute include solid lines, dashed lines, and dotted lines. We modify a line-drawing algorithm to generate such lines by setting the length and spacing of displayed solid sections along the line path. With many graphics packages, we can select the length of both the dashes and the inter-dash spacing.

## Pen and Brush Options

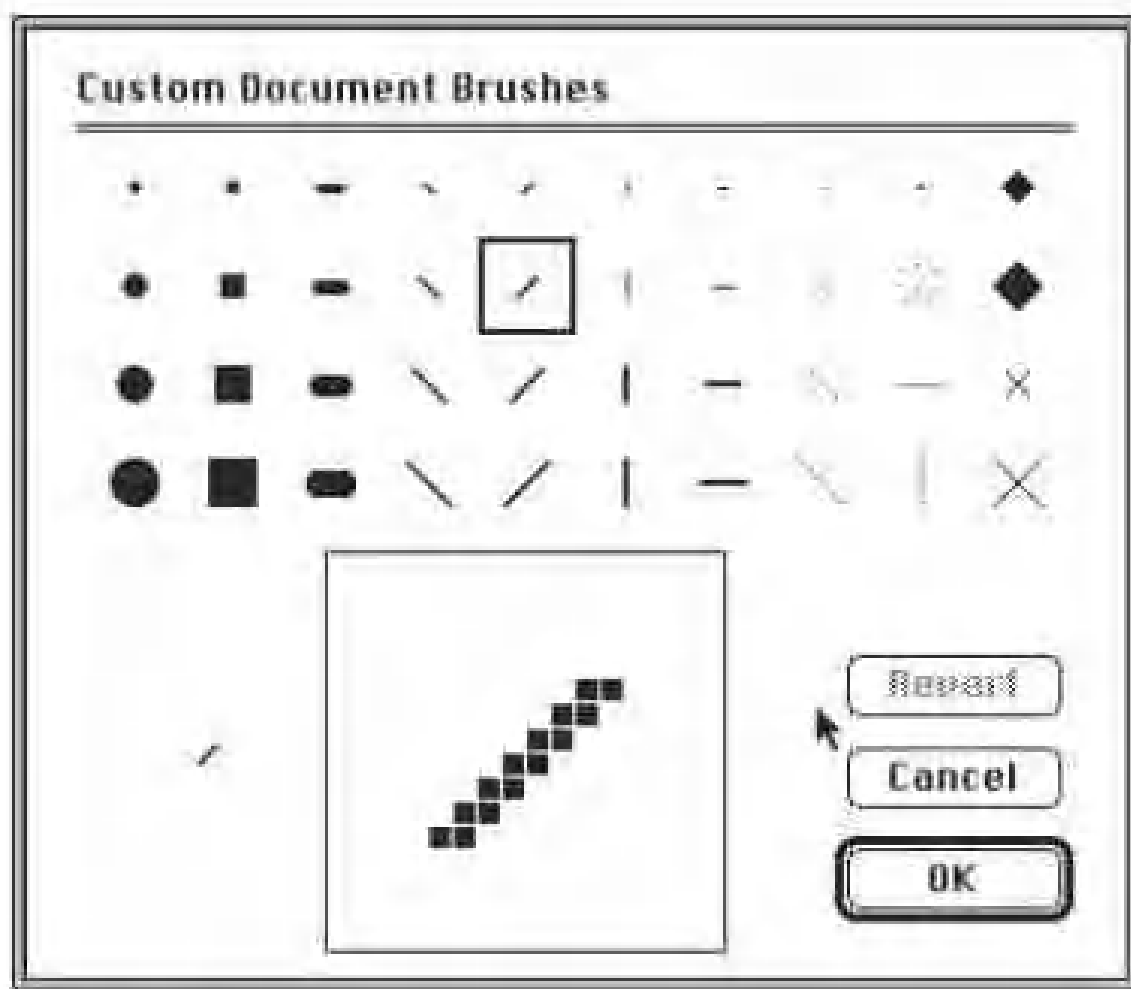


FIGURE 2  
Pen and brush shapes for free display.

With some packages, particularly painting and drawing systems, we can select different pen and brush styles directly. Options in this category include shape, size, and pattern for the pen or brush. Some example pen and brush shapes are given in Figure 2.

### **COLOR AND GRAYSCALE STYLE.**

A basic attribute for all primitives is color. Various color options can be made available to a user, depending on the capabilities and design objectives of a particular system. Color options can be specified numerically or selected from menus or displayed slider scales. For a video monitor, these color codes are then converted to intensity-level settings for the electron beams. With color plotters, the codes might control ink-jet deposits or pen selections.

### **RGB Color Components**

In a color raster system, the number of color choices available depends on the amount of storage provided per pixel in the frame buffer. Also, color information can be stored in the frame buffer in two ways: We can store red, green, and blue (RGB) color codes directly in the frame buffer, or we can

TABLE 1

The eight RGB color codes for a 3-bit-per-pixel frame buffer

Color Code	Stored Color Values in Frame Buffer			Displayed Color
	RED	GREEN	BLUE	
0	0	0	0	Black
1	0	0	1	Blue
2	0	1	0	Green
3	0	1	1	Cyan
4	1	0	0	Red
5	1	0	1	Magenta
6	1	1	0	Yellow
7	1	1	1	White

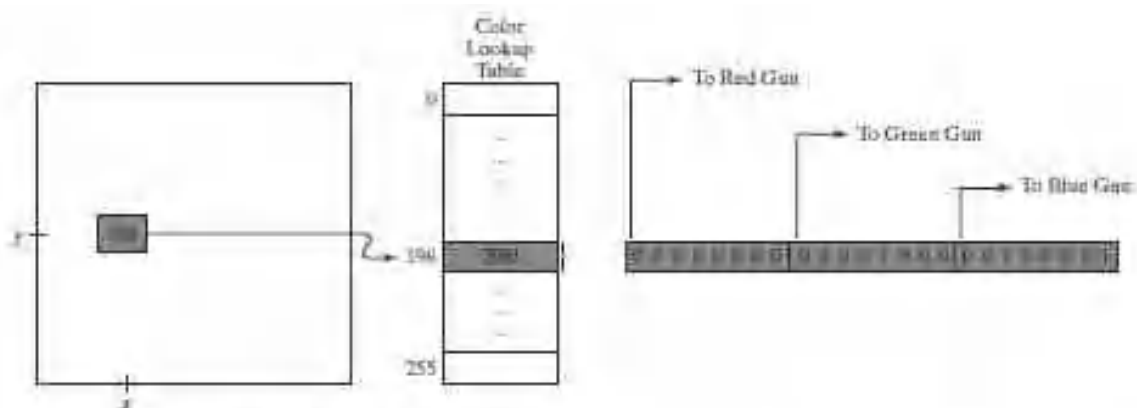
put the color codes into a separate table and use the pixel locations to store index values referencing the color-table entries. With the direct storage scheme, whenever a particular color code is specified in an application program, that color information is placed in the frame buffer at the location of each component pixel in the output primitives to be displayed in that color. A minimum number of colors can be provided in this scheme with 3 bits of storage per pixel, as shown in Table 1. Each of the three-bit positions is used to control the intensity level (either on or off, in this case) of the corresponding electron gun in an RGB monitor. The leftmost bit controls the red gun, the middle bit controls the green gun, and the rightmost bit controls the blue gun. Adding more bits per pixel to the frame buffer increases the number of color choices that we have. With 6 bits per pixel, 2 bits can be used for each gun. This allows four different intensity settings for each of the three color guns, and a total of 64 color options are available for each screen pixel. As more color options are provided, the storage required for the frame buffer also increases.

With a resolution of  $1024 \times 1024$ , a full-color (24-bit per pixel) RGB system needs 3 MB of storage for the frame buffer.

Color tables are an alternate means for providing extended color capabilities to a user without requiring large frame buffers. At one time, this was an important consideration; but today, hardware costs have decreased dramatically and extended color capabilities are fairly common, even in low-end personal computer systems. So most of our examples will simply assume that RGB color codes are stored directly in the frame buffer.

### Color Tables





**FIGURE 1**

A color lookup table with 24 bits per entry that is accessed from a frame buffer with 6 bits per pixel. A value of 196 stored at pixel position  $(x, y)$  references the location in this table containing the hexadecimal value (0x0821) (a decimal value of 2081). Each 8-bit segment of this entry controls the intensity level of one of the three electron guns in an RGB monitor.

Figure 1 illustrates a possible scheme for storing color values in a **color lookup table** (or **color map**). Sometimes a color table is referred to as a **video lookup table**. Values stored in the frame buffer are now used as indices into the colortable. In this example, each pixel can reference any of the 256 table positions, and each entry in the table uses 24 bits to specify an RGB color. For the hexadecimal color code 0x0821, a combination green-blue color is displayed for pixel location  $(x, y)$ . Systems employing this particular lookup table allow a user to select any 256 colors for simultaneous display from a palette of nearly 17 million colors.

Compared to a full-color system, this scheme reduces the number of simultaneous colors that can be displayed, but it also reduces the frame-buffer storage requirement to 1 MB. Multiple color tables are sometimes available for handling specialized rendering applications, such as antialiasing, and they are used with systems that contain more than one color output device.

A color table can be useful in a number of applications, and it can provide a "reasonable" number of simultaneous colors without requiring large frame buffers. For most applications, 256 or 512 different colors are sufficient for a single picture. Also, table entries can be changed at any time, allowing a user to be able to experiment easily with different color combinations in a design, scene, or graph without changing the attribute settings for the graphics data structure.

When a color value is changed in the color table, all pixels with that color index immediately change to the new color. Without a color table, we can change the color of a pixel only by storing the new color at that frame-buffer location. Similarly, data-visualization applications can store values for some physical quantity, such as energy, in the frame buffer and use a lookup table to experiment with various color combinations without changing the pixel values. Also, in visualization and image-processing applications, color tables are a convenient means for setting color thresholds so that all pixel values above or below a specified threshold can be set to the same color. For these reasons, some systems provide both capabilities for storing color

information. A user can then elect either to use color tables or to store color codes directly in the frame buffer.

### **Grayscale**

Because color capabilities are now common in computer-graphics systems, we use RGB color functions to set shades of gray, or **grayscale**, in an application program. When an RGB color setting specifies an equal amount of red, green, and blue, the result is some shade of gray. Values close to 0 for the color components produce dark gray, and higher values near 1.0 produce light gray. Applications for grayscale display methods include enhancing black-and-white photographs and generating visualization effects.

### **Other Color Parameters**

In addition to an RGB specification, other three-component color representations are useful in computer-graphics applications. For example, color output on printers is described with cyan, magenta, and yellow color components, and color interfaces sometimes use parameters such as lightness and darkness to choose a color. Also, color, and light in general, are complex subjects, and many terms and concepts have been devised in the fields of optics, radiometry, and psychology to describe the various aspects of light sources and lighting effects. Physically, we can describe a color as electromagnetic radiation with a particular frequency range and energy distribution, but then there are also the characteristics of our perception of the color. Thus, we use the physical term *intensity* to quantify the amount of light energy radiating in a particular direction over a period of time, and we use the psychological term *luminance* to characterize the perceived brightness of the light. We discuss these terms and other color concepts in greater detail when we consider methods for modeling lighting effects and the various models for describing color.

## **UNIT 2:**

### **TWO DIMENSIONAL TRANSFORMATION**

So far, we have seen how we can describe a scene in terms of graphics primitives, such as line segments and fill areas, and the attributes associated with these primitives.

Also, we have explored the scan-line algorithms for displaying output primitives on a raster device. Now, we take a look at transformation operations that we can apply to objects to reposition or resize them. These operations are also used in the viewing routines that convert a world-coordinate scene description to a display for an output device.

In addition, they are used in a variety of other applications, such as computer-aided design (CAD) and computer animation. An architect, for example, creates a layout by arranging the orientation and size of the component parts of a design, and a computer animator develops a video sequence by moving the “camera” position or the objects in a scene along specified paths. Operations that are applied to the geometric description of an object to change its position, orientation, or size are called **geometric transformations**.

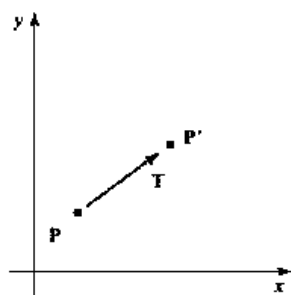
Sometimes geometric transformations are also referred to as *modeling transformations*, but some graphics packages make a distinction between

the two. In general, modeling transformations are used to construct a scene or to give the hierarchical description of a complex object that is composed of several parts, which in turn could be composed of simpler parts, and so forth. For example, an aircraft consists of wings, tail, fuselage, engine, and other components, each of which can be specified in terms of second-level components, and so on, down the hierarchy of component parts. Thus, the aircraft can be described in terms of these components and an associated “modeling” transformation for each one that describes how that component is to be fitted into the overall aircraft design.

Geometric transformations, on the other hand, can be used to describe how objects might move around in a scene during an animation sequence or simply to view them from another angle. Therefore, some graphics packages provide two sets of transformation routines, while other packages have a single set of functions that can be used for both geometric transformations and modeling transformations.

### BASIC TRANSFORMATION

The geometric-transformation functions that are available in all graphics packages are those for translation, rotation, and scaling. Other useful transformation routines that are sometimes included in a package are reflection and shearing operations. To introduce the general concepts associated with geometric transformations, we first consider operations in two dimensions. Once we understand the basic concepts, we can easily write routines to perform geometric transformations on objects in a two-dimensional scene.



**FIGURE 1**  
Translating a point from position  $P$  to position  $P'$  using a translation vector  $T$

#### Two-Dimensional Translation

We perform a translation on a single coordinate point by adding offsets to its coordinates so as to generate a new coordinate position. In effect, we are moving the original point position along a straight-line path to its new location. Similarly, a translation is applied to an object that is defined with multiple coordinate positions, such as a quadrilateral, by relocating all the coordinate positions by the same displacement along parallel paths. Then the complete object is displayed at the new location.

To translate a two-dimensional position, we add translation distances  $t_x$  and  $t_y$  to the original coordinates  $(x, y)$  to obtain the new coordinate position  $(x', y')$  as shown in Figure 1.

$$x' = x + t_x, \quad y' = y + t_y \quad (1)$$

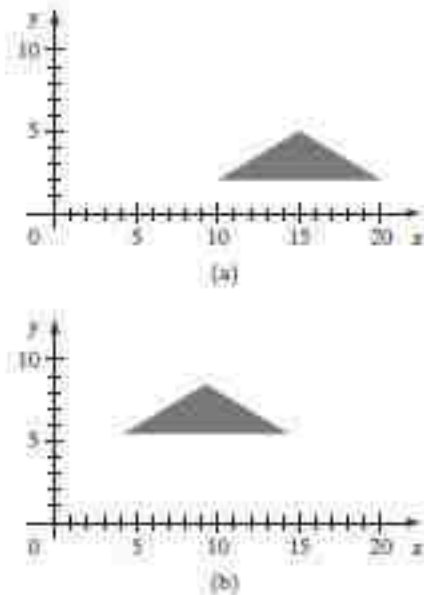
The translation distance pair  $(t_x, t_y)$  is called a **translation vector** or **shift vector**.

We can express Equations 1 as a single matrix equation by using the following column vectors to represent coordinate positions and the translation vector:

$$P = \begin{bmatrix} x \\ y \end{bmatrix}, \quad P' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \quad T = \begin{bmatrix} t_x \\ t_y \end{bmatrix} \quad (2)$$

This allows us to write the two-dimensional translation equations in the matrix form

$$P' = P + T \quad (3)$$



**FIGURE 2**  
Moving a polygon from position (a) to position (b) with the translation vector  $(-5.50, 3.75)$ .

Translation is a *rigid-body transformation* that moves objects without deformation. That is, every point on the object is translated by the same amount. A straight-line segment is translated by applying Equation 3 to each of the two line endpoints and redrawing the line between the new endpoint positions. A polygon is translated similarly. We add a translation vector to the coordinate position of each vertex and then regenerate the polygon using the new set of vertex coordinates. Figure 2 illustrates the application of a specified translation vector to move an object from one position to another.

The following routine illustrates the translation operations. An input translation vector is used to move the  $n$  vertices of a polygon from one world-coordinate position to another, and OpenGL routines are used to regenerate the translated polygon.

```
class wPt2D {
public:
    GLfloat x, y;
};

void translatePolygon (wPt2D * verts, GLint nVerts, GLfloat tx, GLfloat ty)
{
    GLint k;

    for (k = 0; k < nVerts; k++) {
        verts [k].x = verts [k].x + tx;
        verts [k].y = verts [k].y + ty;
    }
    glBegin (GL_POLYGON);
    for (k = 0; k < nVerts; k++)
        glVertex2f (verts [k].x, verts [k].y);
    glEnd ();
}
```

If we want to delete the original polygon, we could display it in the background color before translating it. Other methods for deleting picture components



are available in some graphics packages. Also, if we want to save the original polygon position, we can store the translated positions in a different array.

Similar methods are used to translate other objects. To change the position of a circle or ellipse, we translate the center coordinates and redraw the figure in the new location. For a spline curve, we translate the points that define the curve path and then reconstruct the curve sections between the new coordinate positions.

## Two-Dimensional Rotation

We generate a rotation transformation of an object by specifying a **rotation axis** and a **rotation angle**. All points of the object are then transformed to new positions by rotating the points through the specified angle about the rotation axis.

A two-dimensional rotation of an object is obtained by repositioning the object along a circular path in the  $xy$  plane. In this case, we are rotating the object about a rotation axis that is perpendicular to the  $xy$  plane (parallel to the coordinate  $z$  axis). Parameters for the two-dimensional rotation are the rotation angle  $\theta$  and a position  $(x, y)$ , called the **rotation point** (or **pivot point**), about which the object is to be rotated (Figure 3). The pivot point is the intersection position of the rotation axis with the  $xy$  plane. A positive value for the angle  $\theta$  defines a counterclockwise rotation about the pivot point, as in Figure 3, and a negative value rotates objects in the clockwise direction.

To simplify the explanation of the basic method, we first determine the transformation equations for rotation of a point position  $P$  when the pivot point is at the coordinate origin. The angular and coordinate relationships of the original and transformed point positions are shown in Figure 4. In this figure,  $r$  is the constant distance of the point from the origin, angle  $\phi$  is the original angular position of the point from the horizontal, and  $\theta$  is the rotation angle. Using standard trigonometric identities, we can express the transformed coordinates in terms of angles  $\theta$  and  $\phi$  as

$$\begin{aligned}x' &= r \cos(\phi + \theta) = r \cos \phi \cos \theta - r \sin \phi \sin \theta \\y' &= r \sin(\phi + \theta) = r \cos \phi \sin \theta + r \sin \phi \cos \theta\end{aligned}\quad (4)$$

The original coordinates of the point in polar coordinates are

$$x = r \cos \phi, \quad y = r \sin \phi \quad (5)$$

Substituting expressions 5 into 4, we obtain the transformation equations for rotating a point at position  $(x, y)$  through an angle  $\theta$  about the origin:

$$\begin{aligned}x' &= x \cos \theta - y \sin \theta \\y' &= x \sin \theta + y \cos \theta\end{aligned}\quad (6)$$

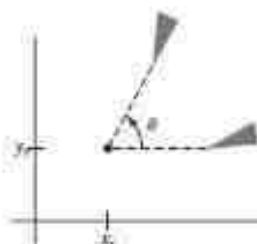
With the column-vector representations 2 for coordinate positions, we can write the rotation equations in the matrix form

$$\mathbf{P}' = \mathbf{R} \cdot \mathbf{P} \quad (7)$$

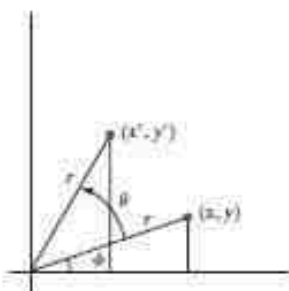
where the rotation matrix is

$$\mathbf{R} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad (8)$$

A column-vector representation for a coordinate position  $P$ , as in Equations 2, is standard mathematical notation. However, early graphics systems sometimes used a row-vector representation for point positions. This changes the order in which the matrix multiplication for a rotation would be performed. But now, graphics packages such as OpenGL, Java, PHICS, and GKS all follow the standard column-vector convention.



**FIGURE 3**  
Rotation of an object through angle  $\theta$  about the pivot point  $(x_p, y_p)$ .



**FIGURE 4**  
Rotation of a point from position  $(x, y)$  to position  $(x', y')$  through an angle  $\theta$  relative to the coordinate origin. The original angular displacement of the point from the  $x$  axis is  $\phi$ .



Rotation of a point about an arbitrary pivot position is illustrated in Figure 5. Using the trigonometric relationships indicated by the two right triangles in this figure, we can generalize Equations 6 to obtain the transformation equations for rotation of a point about any specified rotation position  $(x_r, y_r)$ :

$$\begin{aligned}x' &= x_r + (x - x_r) \cos \theta - (y - y_r) \sin \theta \\y' &= y_r + (x - x_r) \sin \theta + (y - y_r) \cos \theta\end{aligned}\quad (9)$$

These general rotation equations differ from Equations 6 by the inclusion of additive terms, as well as the multiplicative factors on the coordinate values. The matrix expression 7 could be modified to include pivot coordinates by including the matrix addition of a column vector whose elements contain the additive (translational) terms in Equations 9. There are better ways, however, to formulate such matrix equations, and in Section 2, we discuss a more consistent scheme for representing the transformation equations.

As with translations, rotations are rigid-body transformations that move objects without deformation. Every point on an object is rotated through the same angle. A straight-line segment is rotated by applying Equations 9 to each of the two line endpoints and redrawing the line between the new endpoint positions. A polygon is rotated by displacing each vertex using the specified rotation angle and then regenerating the polygon using the new vertices. We rotate a curve by repositioning the defining points for the curve and then redrawing it. A circle or an ellipse, for instance, can be rotated about a noncentral pivot point by moving the center position through the arc that subtends the specified rotation angle. In addition, we could rotate an ellipse about its center coordinates simply by rotating the major and minor axes.

In the following code example, a polygon is rotated about a specified world-coordinate pivot point. Parameters input to the rotation procedure are the original vertices of the polygon, the pivot-point coordinates, and the rotation angle  $\theta$  specified in radians. Following the transformation of the vertex positions, the polygon is regenerated using OpenGL routines.

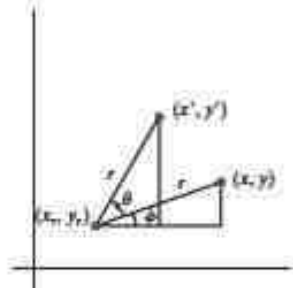


FIGURE 5  
Rotating a point from position  $(x, y)$  to position  $(x', y')$  through an angle  $\theta$  about rotation point  $(x_r, y_r)$ .

```
class wcPt2D {
public:
    GLfloat x, y;
};

void rotatePolygon (wcPt2D * verts, GLint nVerts, wcPt2D pivPt,
                  GLdouble theta)
{
    wcPt2D * vertsRot;
    GLint x;

    for (k = 0; k < nVerts; k++) {
        vertsRot [k].x = pivPt.x + (verts [k].x - pivPt.x) * cos (theta)
            - (verts [k].y - pivPt.y) * sin (theta);
        vertsRot [k].y = pivPt.y + (verts [k].x - pivPt.x) * sin (theta)
            + (verts [k].y - pivPt.y) * cos (theta);
    }
    glBegin (GL_POLYGON);
    for (k = 0; k < nVerts; k++)
        glVertex2f (vertsRot [k].x, vertsRot [k].y);
    glEnd ();
}
```

### Two-Dimensional Scaling

To alter the size of an object, we apply a **scaling** transformation. A simple two-dimensional scaling operation is performed by multiplying object positions  $(x, y)$  by **scaling factors**  $s_x$  and  $s_y$  to produce the transformed coordinates  $(x', y')$ :

$$x' = x \cdot s_x, \quad y' = y \cdot s_y \quad (10)$$

Scaling factor  $s_x$  scales an object in the  $x$  direction, while  $s_y$  scales in the  $y$  direction. The basic two-dimensional scaling equations 10 can also be written in the following matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \quad (11)$$

or

$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P} \quad (12)$$

where  $\mathbf{S}$  is the  $2 \times 2$  scaling matrix in Equation 11.

Any positive values can be assigned to the scaling factors  $s_x$  and  $s_y$ . Values less than 1 reduce the size of objects, values greater than 1 produce enlargements. Specifying a value of 1 for both  $s_x$  and  $s_y$  leaves the size of objects unchanged. When  $s_x$  and  $s_y$  are assigned the same value, a **uniform scaling** is produced, which maintains relative object proportions. Unequal values for  $s_x$  and  $s_y$  result in a **differential scaling** that is often used in design applications, where pictures are constructed from a few basic shapes that can be adjusted by scaling and positioning transformations (Figure 6). In some systems, negative values can also be specified for the scaling parameters. This not only resizes an object, it reflects it about one or more of the coordinate axes.

Objects transformed with Equation 11 are both scaled and repositioned. Scaling factors with absolute values less than 1 move objects closer to the coordinate origin, while absolute values greater than 1 move coordinate positions farther from the origin. Figure 7 illustrates scaling of a line by assigning the value 0.5 to both  $s_x$  and  $s_y$  in Equation 11. Both the line length and the distance from the origin are reduced by a factor of  $\frac{1}{2}$ .

We can control the location of a scaled object by choosing a position, called the **fixed point**, that is to remain unchanged after the scaling transformation. Coordinates for the fixed point,  $(x_f, y_f)$ , are often chosen at some object position, such as its centroid (see Appendix A), but any other spatial position can be selected. Objects are now resized by scaling the distances between object points and the fixed point (Figure 8). For a coordinate position  $(x, y)$ , the scaled coordinates  $(x', y')$  are then calculated from the following relationships:

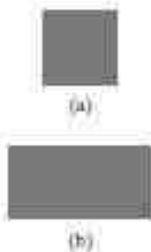
$$x' - x_f = (x - x_f)s_x, \quad y' - y_f = (y - y_f)s_y \quad (13)$$

We can rewrite Equations 13 to separate the multiplicative and additive terms as:

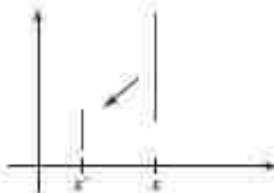
$$\begin{aligned} x' &= x \cdot s_x + x_f(1 - s_x) \\ y' &= y \cdot s_y + y_f(1 - s_y) \end{aligned} \quad (14)$$

where the additive terms  $x_f(1 - s_x)$  and  $y_f(1 - s_y)$  are constants for all points in the object.

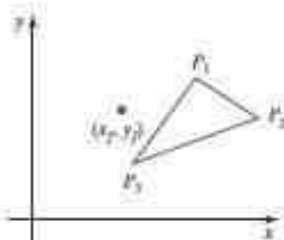
Including coordinates for a fixed point in the scaling equations is similar to including coordinates for a pivot point in the rotation equations. We can set up



**FIGURE 6**  
Turning a square (a) into a rectangle (b) with scaling factors  $s_x = 2$  and  $s_y = 1$ .



**FIGURE 7**  
A line scaled with Equation 12 using  $s_x = s_y = 0.5$  is reduced in size and moved closer to the coordinate origin.



**FIGURE 8**  
Scaling relative to a chosen fixed point  $(x_f, y_f)$ . The distance from each polygon vertex to the fixed point is scaled by Equations 13.

a column vector whose elements are the constant terms in Equations 14, then add this column vector to the product  $\mathbf{S} \cdot \mathbf{P}$  in Equation 12. In the next section, we discuss a matrix formulation for the transformation equations that involves only matrix multiplication.

Polygons are scaled by applying transformations 14 to each vertex, then regenerating the polygon using the transformed vertices. For other objects, we apply the scaling transformation equations to the parameters defining the objects. To change the size of a circle, we can scale its radius and calculate the new coordinate positions around the circumference. And to change the size of an ellipse, we apply scaling parameters to its two axes and then plot the new ellipse positions about its center coordinates.

The following procedure illustrates an application of the scaling calculations for a polygon. Coordinates for the polygon vertices and for the fixed point are input parameters, along with the scaling factors. After the coordinate transformations,

OpenGL routines are used to generate the scaled polygon.

```
class wcPt2D
{public:
    GLfloat x, y;
};
void scalePolygon (wcPt2D * verts, GLint nVerts, wcPt2D fixedPt,
GLfloat sx, GLfloat sy)
{    wcPt2D vertsNew;
    GLint k;
    for (k = 0; k < nVerts; k++)
    {    vertsNew [k].x = verts [k].x * sx + fixedPt.x * (1 - sx);
        vertsNew [k].y = verts [k].y * sy + fixedPt.y * (1 - sy);
    }
    glBegin {GL_POLYGON};
    for (k = 0; k < nVerts; k++)
    glVertex2f (vertsNew [k].x, vertsNew [k].y);
    glEnd ( );
}
```

### **MATRIX REPRESENTATION AND HOMOGENEOUS CO-ORDINATES**

Many graphics applications involve sequences of geometric transformations. An animation might require an object to be translated and rotated at each increment of the motion. In design and picture construction applications, we perform translations, rotations, and scalings to fit the picture components into their proper positions. The viewing transformations involve sequences of translations and rotations to take us from the original scene specification to the display on an output device. Here, we consider how the matrix representations discussed in the previous sections can be reformulated so that such transformation sequences can be processed efficiently.

We have seen in Section 1 that each of the three basic two-dimensional transformations (translation, rotation, and scaling) can be expressed in the general matrix form

$$\mathbf{P}' = \mathbf{M1} \cdot \mathbf{P} + \mathbf{M2}$$

with coordinate positions  $\mathbf{P}$  and  $\mathbf{P}'$  represented as column vectors. Matrix  $\mathbf{M1}$  is a  $2 \times 2$  array containing multiplicative factors, and  $\mathbf{M2}$  is a two-element column matrix containing translational terms. For translation,  $\mathbf{M1}$  is the identity matrix.

For rotation or scaling,  $\mathbf{M}_2$  contains the translational terms associated with the pivot point or scaling fixed point. To produce a sequence of transformations with these equations, such as scaling followed by rotation and then translation, we could calculate the transformed coordinates one step at a time. First, coordinate positions are scaled, then these scaled coordinates are rotated, and finally, the rotated coordinates are translated. A more efficient approach, however, is to combine the transformations so that the final coordinate positions are obtained directly from the initial coordinates, without calculating intermediate coordinate values.

We can do this by reformulating Equation 15 to eliminate the matrix addition operation.

### Homogeneous Coordinates

Multiplicative and translational terms for a two-dimensional geometric transformation can be combined into a single matrix if we expand the representation to  $3 \times 3$  matrices. Then we can use the third column of a transformation matrix for the translation terms, and all transformation equations can be expressed as matrix multiplications. But to do so, we also need to expand the matrix representation for a two-dimensional coordinate position to a three-element column matrix. A standard technique for accomplishing this is to expand each two-dimensional coordinate-position representation  $(x, y)$  to a three-element representation  $(xh, yh, h)$ , called **homogeneous coordinates**, where the **homogeneous parameter**  $h$  is a nonzero value such that  $x = xh$ ,  $y = yh$

Therefore, a general two-dimensional homogeneous coordinate representation could also be written as  $(h \cdot x, h \cdot y, h)$ . For geometric transformations, we can choose the homogeneous parameter  $h$  to be any nonzero value. Thus, each coordinate point  $(x, y)$  has an infinite number of equivalent homogeneous representations.

A convenient choice is simply to set  $h = 1$ . Each two-dimensional position is then represented with homogeneous coordinates  $(x, y, 1)$ . Other values for parameter  $h$  are needed, for example, in matrix formulations of three-dimensional viewing transformations.

The term *homogeneous coordinates* is used in mathematics to refer to the effect of this representation on Cartesian equations. When a Cartesian point  $(x, y)$  is converted to a homogeneous representation  $(xh, yh, h)$ , equations containing  $x$  and  $y$ , such as  $f(x, y) = 0$ , become homogeneous equations in the three parameters  $xh$ ,  $yh$ , and  $h$ . This just means that if each of the three parameters is replaced by any value  $v$  times that parameter, the value  $v$  can be factored out of the equations.

Expressing positions in homogeneous coordinates allows us to represent all geometric transformation equations as matrix multiplications, which is the standard method used in graphics systems. Two-dimensional coordinate positions are represented with three-element column vectors, and two-dimensional transformation operations are expressed as  $3 \times 3$  matrices.



## Two-Dimensional Translation Matrix

Using a homogeneous-coordinate approach, we can represent the equations for a two-dimensional translation of a coordinate position using the following matrix multiplication:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (17)$$

This translation operation can be written in the abbreviated form

$$\mathbf{P}' = \mathbf{T}(t_x, t_y) \cdot \mathbf{P} \quad (18)$$

with  $\mathbf{T}(t_x, t_y)$  as the  $3 \times 3$  translation matrix in Equation 17. In situations where there is no ambiguity about the translation parameters, we can simply represent the translation matrix as  $\mathbf{T}$ .

## Two-Dimensional Rotation Matrix

Similarly, two-dimensional rotation transformation equations about the coordinate origin can be expressed in the matrix form

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (19)$$

or as

$$\mathbf{P}' = \mathbf{R}(\theta) \cdot \mathbf{P} \quad (20)$$

The rotation transformation operator  $\mathbf{R}(\theta)$  is the  $3 \times 3$  matrix in Equation 19 with rotation parameter  $\theta$ . We can also write this rotation matrix simply as  $\mathbf{R}$ .

In some graphics libraries, a two-dimensional rotation function generates only rotations about the coordinate origin, as in Equation 19. A rotation about any other pivot point must then be performed as a sequence of transformation operations. An alternative approach in a graphics package is to provide additional parameters in the rotation routine for the pivot-point coordinates. A rotation routine that includes a pivot-point parameter then sets up a general rotation matrix without the need to invoke a succession of transformation functions.

## Two-Dimensional Scaling Matrix

Finally, a scaling transformation relative to the coordinate origin can now be expressed as the matrix multiplication

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (21)$$

or

$$\mathbf{P}' = \mathbf{S}(s_x, s_y) \cdot \mathbf{P} \quad (22)$$

The scaling operator  $\mathbf{S}(s_x, s_y)$  is the  $3 \times 3$  matrix in Equation 21 with parameters  $s_x$  and  $s_y$ . And, in most cases, we can represent the scaling matrix simply as  $\mathbf{S}$ .

Some libraries provide a scaling function that can generate only scaling with respect to the coordinate origin, as in Equation 21. In this case, a scaling transformation relative to another reference position is handled as a succession of transformation operations. However, other systems do include a general scaling routine that can construct the homogeneous matrix for scaling with respect to a designated fixed point.



## COMPOSITE TRANSFORMATION-MATRIX REPRESENTATION

Using matrix representations, we can set up a sequence of transformations as a **composite transformation matrix** by calculating the product of the individual transformations. Forming products of transformation matrices is often referred to as a **concatenation**, or **composition**, of matrices. Because a coordinate position is represented with a homogeneous column matrix, we must premultiply the column matrix by the matrices representing any transformation sequence.

Also, because many positions in a scene are typically transformed by the same sequence, it is more efficient to first multiply the transformation matrices to form a single composite matrix. Thus, if we want to apply two transformations to point position  $\mathbf{P}$ , the transformed location would be calculated as

$$\begin{aligned}\mathbf{P}' &= \mathbf{M}_2 \cdot \mathbf{M}_1 \cdot \mathbf{P} \\ &= \mathbf{M} \cdot \mathbf{P}\end{aligned}$$

The coordinate position is transformed using the composite matrix  $\mathbf{M}$ , rather than applying the individual transformations  $\mathbf{M}_1$  and then  $\mathbf{M}_2$ .

### Composite Two-Dimensional Translations

If two successive translation vectors  $(t1x, t1y)$  and  $(t2x, t2y)$  are applied to a two-dimensional coordinate position  $\mathbf{P}$ , the final transformed location  $\mathbf{P}'$  is calculated as

$$\begin{aligned}\mathbf{P}' &= \mathbf{T}(t2x, t2y) \cdot \{\mathbf{T}(t1x, t1y) \cdot \mathbf{P}\} \\ &= \{\mathbf{T}(t2x, t2y) \cdot \mathbf{T}(t1x, t1y)\} \cdot \mathbf{P}\end{aligned}$$

where  $\mathbf{P}$  and  $\mathbf{P}'$  are represented as three-element, homogeneous-coordinate column vectors. We can verify this result by calculating the matrix product for the two associative groupings. Also, the composite transformation matrix for this sequence of translations is

$$\begin{bmatrix} 1 & 0 & t_{1x} \\ 0 & 1 & t_{1y} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_{2x} \\ 0 & 1 & t_{2y} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{1x} + t_{2x} \\ 0 & 1 & t_{1y} + t_{2y} \\ 0 & 0 & 1 \end{bmatrix} \quad (28)$$

or

$$\mathbf{T}(t_{2x}, t_{2y}) \cdot \mathbf{T}(t_{1x}, t_{1y}) = \mathbf{T}(t_{1x} + t_{2x}, t_{1y} + t_{2y}) \quad (29)$$

which demonstrates that two successive translations are additive.

### Composite Two-Dimensional Rotations

Two successive rotations applied to a point  $\mathbf{P}$  produce the transformed position

$$\begin{aligned}\mathbf{P}' &= \mathbf{R}(\theta_2) \cdot \{\mathbf{R}(\theta_1) \cdot \mathbf{P}\} \\ &= \{\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1)\} \cdot \mathbf{P}\end{aligned} \quad (30)$$

By multiplying the two rotation matrices, we can verify that two successive rotations are additive:

$$\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1) = \mathbf{R}(\theta_1 + \theta_2) \quad (31)$$

so that the final rotated coordinates of a point can be calculated with the composite rotation matrix as

$$\mathbf{P}' = \mathbf{R}(\theta_1 + \theta_2) \cdot \mathbf{P} \quad (32)$$

### Composite Two-Dimensional Scalings

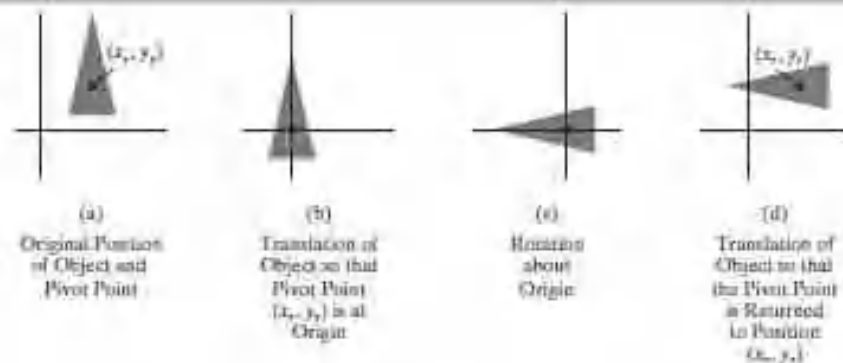
Concatenating transformation matrices for two successive scaling operations in two dimensions produces the following composite scaling matrix:

$$\begin{bmatrix} s_{2x} & 0 & 0 \\ 0 & s_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{1x} & 0 & 0 \\ 0 & s_{1y} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{1x} \cdot s_{2x} & 0 & 0 \\ 0 & s_{1y} \cdot s_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (33)$$

or

$$\mathbf{S}(s_{2x}, s_{2y}) \cdot \mathbf{S}(s_{1x}, s_{1y}) = \mathbf{S}(s_{1x} \cdot s_{2x}, s_{1y} \cdot s_{2y}) \quad (34)$$

The resulting matrix in this case indicates that successive scaling operations are multiplicative. That is, if we were to triple the size of an object twice in succession, the final size would be nine times that of the original.



**FIGURE 9**  
A transformation sequence for rotating an object about a specified pivot point using the rotation matrix  $\mathbf{R}(\theta)$  of transformation 19.

### General Two-Dimensional Pivot-Point Rotation

When a graphics package provides only a rotate function with respect to the coordinate origin, we can generate a two-dimensional rotation about any other pivot point  $(x_p, y_p)$  by performing the following sequence of translate-rotate-translate operations:

1. Translate the object so that the pivot-point position is moved to the coordinate origin.
2. Rotate the object about the coordinate origin.
3. Translate the object so that the pivot point is returned to its original position.

This transformation sequence is illustrated in Figure 9. The composite transformation matrix for this sequence is obtained with the concatenation

$$\begin{aligned} & \begin{bmatrix} 1 & 0 & x_p \\ 0 & 1 & y_p \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_p \\ 0 & 1 & -y_p \\ 0 & 0 & 1 \end{bmatrix} \\ & = \begin{bmatrix} \cos \theta & -\sin \theta & x_p(1 - \cos \theta) + y_p \sin \theta \\ \sin \theta & \cos \theta & y_p(1 - \cos \theta) - x_p \sin \theta \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \quad (35)$$

which can be expressed in the form

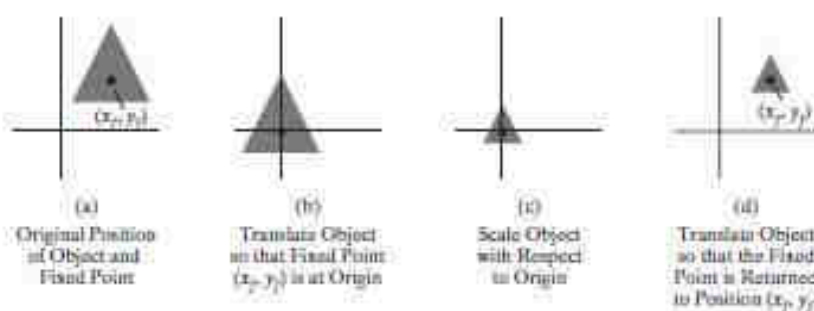
$$\mathbf{T}(x_p, y_p) \cdot \mathbf{R}(\theta) \cdot \mathbf{T}(-x_p, -y_p) = \mathbf{R}(x_p, y_p, \theta) \quad (36)$$

where  $T(-x_f, -y_f) = T^{-1}(x_f, y_f)$ . In general, a rotate function in a graphics library could be structured to accept parameters for pivot-point coordinates, as well as the rotation angle, and to generate automatically the rotation matrix of Equation 35.

## General Two-Dimensional Fixed-Point Scaling

Figure 10 illustrates a transformation sequence to produce a two-dimensional scaling with respect to a selected fixed position  $(x_f, y_f)$ , when we have a function that can scale relative to the coordinate origin only. This sequence is

1. Translate the object so that the fixed point coincides with the coordinate origin.



**FIGURE 10**  
A transformation sequence for scaling an object with respect to a specified fixed position using the scaling matrix  $S(s_x, s_y)$  of transformation 21.

2. Scale the object with respect to the coordinate origin.
3. Use the inverse of the translation in step (1) to return the object to its original position.

Concatenating the matrices for these three operations produces the required scaling matrix:

$$\begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & x_f(1-s_x) \\ 0 & s_y & y_f(1-s_y) \\ 0 & 0 & 1 \end{bmatrix} \quad (37)$$

or

$$T(x_f, y_f) \cdot S(s_x, s_y) \cdot T(-x_f, -y_f) = S(x_f, y_f, s_x, s_y) \quad (38)$$

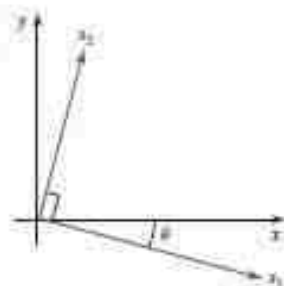
This transformation is generated automatically in systems that provide a scale function that accepts coordinates for the fixed point.

## General Two-Dimensional Scaling Directions

Parameters  $s_x$  and  $s_y$  scale objects along the  $x$  and  $y$  directions. We can scale an object in other directions by rotating the object to align the desired scaling directions with the coordinate axes before applying the scaling transformation.

Suppose we want to apply scaling factors with values specified by parameters  $s_1$  and  $s_2$  in the directions shown in Figure 11. To accomplish the scaling without changing the orientation of the object, we first perform a rotation so that the directions for  $s_1$  and  $s_2$  coincide with the  $x$  and  $y$  axes, respectively. Then the scaling transformation  $S(s_1, s_2)$  is applied, followed by an opposite rotation to return points to their original orientations. The composite matrix resulting from the product of these three transformations is

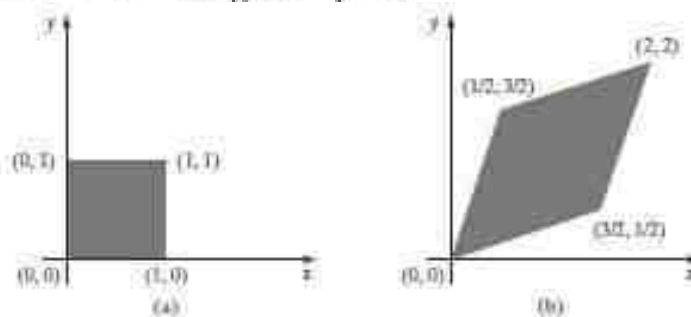
$$R^{-1}(\theta) \cdot S(s_1, s_2) \cdot R(\theta) = \begin{bmatrix} s_1 \cos^2 \theta + s_2 \sin^2 \theta & (s_2 - s_1) \cos \theta \sin \theta & 0 \\ (s_2 - s_1) \cos \theta \sin \theta & s_1 \sin^2 \theta + s_2 \cos^2 \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (39)$$



**FIGURE 11**  
Scaling parameters  $s_1$  and  $s_2$  along orthogonal directions defined by the angular displacement  $\theta$ .

As an example of this scaling transformation, we turn a unit square into a parallelogram (Figure 12) by stretching it along the diagonal from  $(0, 0)$  to  $(1, 1)$ . We first rotate the diagonal onto the  $y$  axis using  $\theta = 45^\circ$ , then we double its length with the scaling values  $s_1 = 1$  and  $s_2 = 2$ , and then we rotate again to return the diagonal to its original orientation.

In Equation 39, we assumed that scaling was to be performed relative to the origin. We could take this scaling operation one step further and concatenate the matrix with translation operators, so that the composite matrix would include parameters for the specification of a scaling fixed position.



**FIGURE 12**  
A square (a) is converted to a parallelogram (b) using the composite transformation matrix 39, with  $s_1 = 1$ ,  $s_2 = 2$ , and  $\theta = 45^\circ$ .

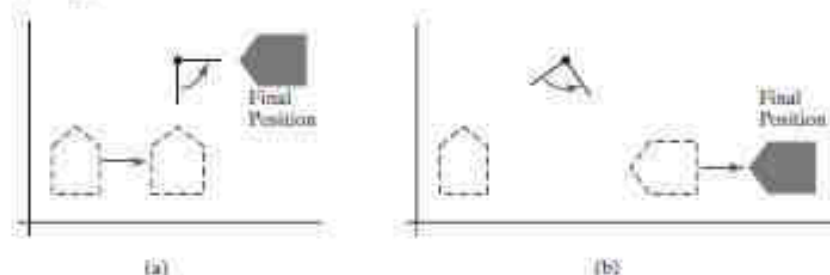
### Matrix Concatenation Properties

Multiplication of matrices is associative. For any three matrices,  $M_1$ ,  $M_2$ , and  $M_3$ , the matrix product  $M_3 \cdot M_2 \cdot M_1$  can be performed by first multiplying  $M_3$  and  $M_2$  or by first multiplying  $M_2$  and  $M_1$ :

$$M_3 \cdot M_2 \cdot M_1 = (M_3 \cdot M_2) \cdot M_1 = M_3 \cdot (M_2 \cdot M_1) \quad (40)$$

Therefore, depending upon the order in which the transformations are specified, we can construct a composite matrix either by multiplying from left to right (premultiplying) or by multiplying from right to left (postmultiplying). Some graphics packages require that transformations be specified in the order in which they are to be applied. In that case, we would first invoke transformation  $M_1$ , then  $M_2$ , then  $M_3$ . As each successive transformation routine is called, its matrix is concatenated on the left of the previous matrix product. Other graphics systems, however, postmultiply matrices, so that this transformation sequence would have to be invoked in the reverse order: the last transformation invoked (which is  $M_1$  for this example) is the first to be applied, and the first transformation that is called ( $M_3$  in this case) is the last to be applied.

Transformation products, on the other hand, may not be commutative. The matrix product  $M_2 \cdot M_1$  is not equal to  $M_1 \cdot M_2$ , in general. This means that if we want to translate and rotate an object, we must be careful about the order in which the composite matrix is evaluated (Figure 13). For some special cases—such as a sequence of transformations that are all of the same kind—the multiplication of transformation matrices is commutative. As an example, two successive rotations could be performed in either order and the final position would be the same. This commutative property holds also for two successive translations or two successive scalings. Another commutative pair of operations is rotation and uniform scaling ( $s_x = s_y$ ).



**FIGURE 13**  
Reversing the order in which a sequence of transformations is performed may affect the transformed position of an object. In (a), an object is first translated in the  $x$  direction, then rotated counterclockwise through an angle of  $45^\circ$ . In (b), the object is first rotated  $45^\circ$  counterclockwise, then translated in the  $x$  direction.

## General Two-Dimensional Composite Transformations and Computational Efficiency

A two-dimensional transformation, representing any combination of translations, rotations, and scalings, can be expressed as

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} rs_{xx} & rs_{xy} & trs_x \\ rs_{yx} & rs_{yy} & trs_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (41)$$

The four elements  $rs_{jk}$  are the multiplicative rotation-scaling terms in the transformation, which involve only rotation angles and scaling factors. Elements  $trs_x$  and  $trs_y$  are the translational terms, containing combinations of translation distances, pivot-point and fixed-point coordinates, rotation angles, and scaling parameters. For example, if an object is to be scaled and rotated about its centroid coordinates  $(x_c, y_c)$  and then translated, the values for the elements of the composite transformation matrix are

$$\begin{aligned} & T(t_x, t_y) \cdot R(x_c, y_c, \theta) \cdot S(s_x, s_y) \\ &= \begin{bmatrix} s_x \cos \theta & -s_y \sin \theta & x_c(1 - s_x \cos \theta) + y_c s_y \sin \theta + t_x \\ s_x \sin \theta & s_y \cos \theta & y_c(1 - s_y \cos \theta) - x_c s_x \sin \theta + t_y \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \quad (42)$$

Although Equation 41 requires nine multiplications and six additions, the explicit calculations for the transformed coordinates are

$$x' = x \cdot rs_{xx} + y \cdot rs_{xy} + trs_x, \quad y' = x \cdot rs_{yx} + y \cdot rs_{yy} + trs_y \quad (43)$$

Thus, we need actually perform only four multiplications and four additions to transform coordinate positions. This is the maximum number of computations required for any transformation sequence, once the individual matrices have been concatenated and the elements of the composite matrix evaluated. Without concatenation, the individual transformations would be applied one at a time, and the number of calculations could be increased significantly. An efficient implementation for the transformation operations, therefore, is to formulate transformation matrices, concatenate any transformation sequence, and calculate transformed coordinates using Equations 43. On parallel systems, direct matrix multiplications with the composite transformation matrix of Equation 41 can be equally efficient.

Because rotation calculations require trigonometric evaluations and several multiplications for each transformed point, computational efficiency can become an important consideration in rotation transformations. In animations and other applications that involve many repeated transformations and small rotation angles, we can use approximations and iterative calculations to reduce computations in the composite transformation equations. When the rotation angle is small, the trigonometric functions can be replaced with approximation values based on the first few terms of their power series expansions. For small-enough angles (less than  $10^\circ$ ),  $\cos \theta$  is approximately 1.0 and  $\sin \theta$  has a value very close to the value of  $\theta$  in radians. If we are rotating in small angular steps about the origin, for instance, we can set  $\cos \theta$  to 1.0 and reduce transformation calculations at each step to two multiplications and two additions for each set of coordinates to be rotated. These rotation calculations are



error over many steps can become quite large. We can control the accumulated error by estimating the error in  $x'$  and  $y'$  at each step and resetting object positions when the error accumulation becomes too great. Some animation applications automatically reset object positions at fixed intervals, such as every  $360^\circ$  or every  $180^\circ$ .

Composite transformations often involve inverse matrices. For example, transformation sequences for general scaling directions and for some reflections and shears (Section 5) require inverse rotations. As we have noted, the inverse matrix representations for the basic geometric transformations can be generated with simple procedures. An inverse translation matrix is obtained by changing the signs of the translation distances, and an inverse rotation matrix is obtained by performing a matrix transpose (or changing the sign of the sine terms). These operations are much simpler than direct inverse matrix calculations.

## Two-Dimensional Rigid-Body Transformation

If a transformation matrix includes only translation and rotation parameters, it is a **rigid-body transformation matrix**. The general form for a two-dimensional rigid-body transformation matrix is

$$\begin{bmatrix} r_{xx} & r_{xy} & tr_x \\ r_{yx} & r_{yy} & tr_y \\ 0 & 0 & 1 \end{bmatrix} \quad (45)$$

where the four elements  $r_{ij}$  are the multiplicative rotation terms, and the elements  $tr_x$  and  $tr_y$  are the translational terms. A rigid-body change in coordinate position is also sometimes referred to as a **rigid-motion transformation**. All angles and distances between coordinate positions are unchanged by the transformation. In addition, matrix 45 has the property that its upper-left  $2 \times 2$  submatrix is an **orthogonal matrix**. This means that if we consider each row (or each column) of the submatrix as a vector, then the two-row vectors  $(r_{xx}, r_{xy})$  and  $(r_{yx}, r_{yy})$  (or the two column vectors) form an orthogonal set of unit vectors. Such a set of vectors is also referred to as an **orthonormal vector set**. Each vector has unit length as follows:

$$r_{xx}^2 + r_{xy}^2 = r_{yx}^2 + r_{yy}^2 = 1 \quad (46)$$

and the vectors are perpendicular (their dot product is 0):

$$r_{xx}r_{yx} + r_{xy}r_{yy} = 0 \quad (47)$$

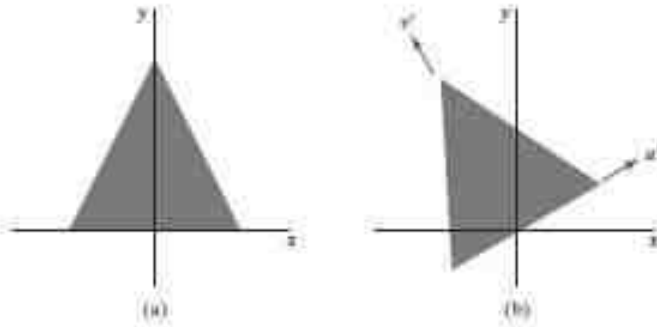
Therefore, if these unit vectors are transformed by the rotation submatrix, then the vector  $(r_{xx}, r_{xy})$  is converted to a unit vector along the  $x$  axis and the vector  $(r_{yx}, r_{yy})$  is transformed into a unit vector along the  $y$  axis of the coordinate system:

$$\begin{bmatrix} r_{xx} & r_{xy} & 0 \\ r_{yx} & r_{yy} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{xx} \\ r_{xy} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \quad (48)$$

$$\begin{bmatrix} r_{xx} & r_{xy} & 0 \\ r_{yx} & r_{yy} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{yx} \\ r_{yy} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \quad (49)$$

For example, the following rigid-body transformation first rotates an object through an angle  $\theta$  about a pivot point  $(x_p, y_p)$  and then translates the object:

$$T(t_x, t_y) \cdot R(x_p, y_p, \theta) = \begin{bmatrix} \cos \theta & -\sin \theta & x_p(1 - \cos \theta) + y_p \sin \theta + t_x \\ \sin \theta & \cos \theta & y_p(1 - \cos \theta) - x_p \sin \theta + t_y \\ 0 & 0 & 1 \end{bmatrix} \quad (50)$$



**FIGURE 14**  
The rotation matrix for revolving an object from position (a) to position (b) can be constructed with the values of the unit orientation vectors  $\mathbf{u}$  and  $\mathbf{v}$  relative to the original orientation.

Here, orthogonal unit vectors in the upper-left  $2 \times 2$  submatrix are  $(\cos \theta, -\sin \theta)$  and  $(\sin \theta, \cos \theta)$ , and

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos \theta \\ -\sin \theta \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \quad (51)$$

Similarly, unit vector  $(\sin \theta, \cos \theta)$  is converted by the preceding transformation matrix to the unit vector  $(0, 1)$  in the  $y$  direction.

### Constructing Two-Dimensional Rotation Matrices

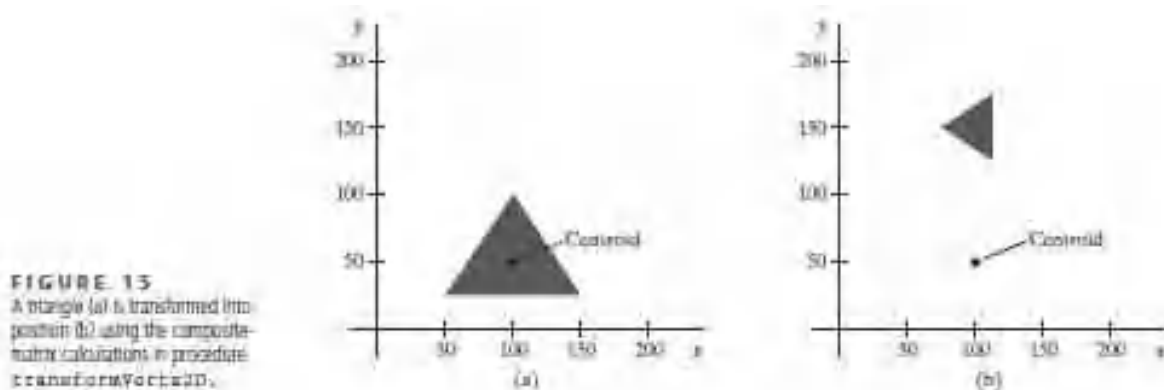
The orthogonal property of rotation matrices is useful for constructing the matrix when we know the final orientation of an object, rather than the amount of angular rotation necessary to put the object into that position. This orientation information could be determined by the alignment of certain objects in a scene or by reference positions within the coordinate system. For example, we might want to rotate an object to align its axis of symmetry with the viewing (camera) direction, or we might want to rotate one object so that it is above another object. Figure 14 shows an object that is to be aligned with the unit direction vectors  $\mathbf{u}$  and  $\mathbf{v}$ . Assuming that the original object orientation, as shown in Figure 14(a), is aligned with the coordinate axes, we construct the desired transformation by assigning the elements of  $\mathbf{u}$  to the first row of the rotation matrix and the elements of  $\mathbf{v}$  to the second row. In a modeling application, for instance, we can use this method to obtain the transformation matrix within an object's local coordinate system when we know what its orientation is to be within the overall world-coordinate scene. A similar transformation is the conversion of object descriptions from one coordinate system to another, and we take up these methods in more detail in Section 8.

### Two-Dimensional Composite-Matrix Programming Example

An implementation example for a sequence of geometric transformations is given in the following program. Initially, the composite matrix, **compMatrix**, is constructed as the identity matrix. For this example, a left-to-right concatenation order is used to construct the composite transformation matrix, and we invoke the transformation routines in the order that they are to be executed. As each of the basic transformation

routines (scale, rotate, and translate) is invoked, a matrix is set up for that transformation and left-concatenated with the composite matrix.

When all transformations have been specified, the composite transformation is applied to transform a triangle. The triangle is first scaled with respect to its centroid position, then the triangle is rotated about its centroid, and, lastly, it is translated. Figure 15 shows the original and final positions of the triangle that is transformed by this sequence. Routines in OpenGL are used to display the initial and final position of the triangle.



**FIGURE 15**  
A triangle (a) is transformed into position (b) using the composite-matrix calculations in procedure `transformVertex2D`.

```
#include <GL/glut.h>
#include <stdlib.h>
#include <math.h>
/* Set initial display-window size. */
GLsizei winWidth = 600, winHeight = 600;
/* Set range for world coordinates. */
GLfloat xwcMin = 0.0, xwcMax = 225.0;
GLfloat ywcMin = 0.0, ywcMax = 225.0;
class wcPt2D
{
    public:
        GLfloat x, y;
};
typedef GLfloat Matrix3x3 [3][3];
Matrix3x3 matComposite;
const GLdouble pi = 3.14159;
void init (void)
{
    /* Set color of display window to white. */
    glClearColor (1.0, 1.0, 1.0, 0.0);
}
/* Construct the 3 x 3 identity matrix. */
void matrix3x3SetIdentity (Matrix3x3 matIdent3x3)
{
    GLint row, col;
    for (row = 0; row < 3; row++)
        for (col = 0; col < 3; col++)
            matIdent3x3 [row][col] = (row == col);
}
```

```

/* Premultiply matrix m1 times matrix m2, store result in m2. */
void matrix3x3PreMultiply (Matrix3x3 m1, Matrix3x3 m2)
{
    GLint row, col;
    Matrix3x3 matTemp;
    for (row = 0; row < 3; row++)
        for (col = 0; col < 3 ; col++)
            matTemp [row][col] = m1 [row][0] * m2 [0][col]
                + m1 [row][1] *m2 [1][col] + m1 [row][2] *
                    m2 [2][col];

    for (row = 0; row < 3; row++)
        for (col = 0; col < 3; col++)
            m2 [row][col] = matTemp [row][col];
}

void translate2D (GLfloat tx, GLfloat ty)
{
    Matrix3x3 matTransl;
    /* Initialize translation matrix to identity. */
    matrix3x3SetIdentity (matTransl);
    matTransl [0][2] = tx;
    matTransl [1][2] = ty;
    /* Concatenate matTransl with the composite matrix. */
    matrix3x3PreMultiply (matTransl, matComposite);
}

void rotate2D (wcPt2D pivotPt, GLfloat theta)
{
    Matrix3x3 matRot;
    /* Initialize rotation matrix to identity. */
    matrix3x3SetIdentity (matRot);
    matRot [0][0] = cos (theta);
    matRot [0][1] = -sin (theta);
    matRot [0][2] = pivotPt.x * (1 - cos (theta)) +
        pivotPt.y * sin (theta);
    matRot [1][0] = sin (theta);
    matRot [1][1] = cos (theta);
    matRot [1][2] = pivotPt.y * (1 - cos (theta)) -
        pivotPt.x * sin (theta);
    /* Concatenate matRot with the composite matrix. */
    matrix3x3PreMultiply (matRot, matComposite);
}

void scale2D (GLfloat sx, GLfloat sy, wcPt2D fixedPt)
{
    Matrix3x3 matScale;
    /* Set geometric transformation parameters. */
    wcPt2D pivPt, fixedPt;
    pivPt = centroidPt;
    fixedPt = centroidPt;
    GLfloat tx = 0.0, ty = 100.0;
    GLfloat sx = 0.5, sy = 0.5;
    GLdouble theta = pi/2.0;
    glClear (GL_COLOR_BUFFER_BIT); // Clear display window.
    glColor3f (0.0, 0.0, 1.0); // Set initial fill color to blue.
    triangle (verts); // Display blue triangle.
}

```

```

    /* Initialize composite matrix to identity. */
    matrix3x3SetIdentity (matComposite);
    /* Construct composite matrix for transformation sequence. */
    scale2D (sx, sy, fixedPt); // First transformation: Scale.
    rotate2D (pivPt, theta); // Second transformation: Rotate
    translate2D (tx, ty); // Final transformation: Translate.
    /* Apply composite matrix to triangle vertices. */
    transformVerts2D (nVerts, verts);
    glColor3f (1.0, 0.0, 0.0); // Set color for transformed triangle.
    triangle (verts); // Display red transformed triangle.
    glFlush ( );
}
void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ( );
    gluOrtho2D (xwcMin, xwcMax, ywcMin, ywcMax);
    glClear (GL_COLOR_BUFFER_BIT);
}
void main (int argc, char ** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (50, 50);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Geometric Transformation Sequence");
    init ( );
    glutDisplayFunc (displayFcn);
    glutReshapeFunc (winReshapeFcn);
    glutMainLoop ( );
}
/* Initialize scaling matrix to identity. */
matrix3x3SetIdentity (matScale);
matScale [0][0] = sx;
matScale [0][2] = (1 - sx) * fixedPt.x;
matScale [1][1] = sy;
matScale [1][2] = (1 - sy) * fixedPt.y;
/* Concatenate matScale with the composite matrix. */
matrix3x3PreMultiply (matScale, matComposite);
}
/* Using the composite matrix, calculate transformed coordinates. */
void transformVerts2D (GLint nVerts, wcPt2D * verts)
{
    GLint k;
    GLfloat temp;
    for (k = 0; k < nVerts; k++) {
        temp = matComposite [0][0] * verts [k].x + matComposite [0][1] *
                verts [k].y + matComposite [0][2];
        verts [k].y = matComposite [1][0] * verts [k].x +
                matComposite [1][1] *verts [k].y + matComposite [1][2];
        verts [k].x = temp;
    }
}

```



```

}
void triangle (wcPt2D *verts)
{
    GLint k;
    glBegin (GL_TRIANGLES);
    for (k = 0; k < 3; k++)
        glVertex2f (verts [k].x, verts [k].y);
    glEnd ( );
}void displayFcn (void)
{
    /* Define initial position for triangle. */
    GLint nVerts = 3;
    wcPt2D verts [3] = { {50.0, 25.0}, {150.0, 25.0}, {100.0, 100.0} };
    /* Calculate position of triangle centroid. */
    wcPt2D centroidPt;
    GLint k, xSum = 0, ySum = 0;
    for (k = 0; k < nVerts; k++) {
        xSum += verts [k].x;
        ySum += verts [k].y;}
    centroidPt.x = GLfloat (xSum) / GLfloat (nVerts);
    centroidPt.y = GLfloat (ySum) / GLfloat (nVerts);
}

```

## OTHER TRANSFORMATIONS

### 5 Other Two-Dimensional Transformations

Basic transformations such as translation, rotation, and scaling are standard components of graphics libraries. Some packages provide a few additional transformations that are useful in certain applications. Two such transformations are reflection and shear.

#### Reflection

A transformation that produces a mirror image of an object is called a **reflection**. For a two-dimensional reflection, this image is generated relative to an axis of reflection by rotating the object  $180^\circ$  about the reflection axis. We can choose an axis of reflection in the  $xy$  plane or perpendicular to the  $xy$  plane. When the reflection axis is a line in the  $xy$  plane, the rotation path about this axis is in a plane perpendicular to the  $xy$  plane. For reflection axes that are perpendicular to the  $xy$  plane, the rotation path is in the  $xy$  plane. Some examples of common reflections follow.

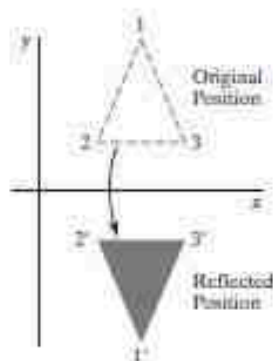
Reflection about the line  $y = 0$  (the  $x$  axis) is accomplished with the transformation matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (52)$$

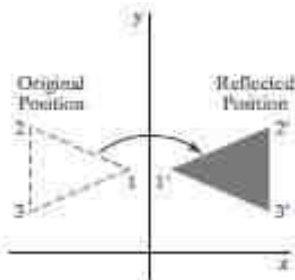
This transformation retains  $x$  values, but “flips” the  $y$  values of coordinate positions. The resulting orientation of an object after it has been reflected about the  $x$  axis is shown in Figure 16. To envision the rotation transformation path for this reflection, we can think of the flat object moving out of the  $xy$  plane and rotating  $180^\circ$  through three-dimensional space about the  $x$  axis and back into the  $xy$  plane on the other side of the  $x$  axis.

A reflection about the line  $x = 0$  (the  $y$  axis) flips  $x$  coordinates while keeping  $y$  coordinates the same. The matrix for this transformation is

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (53)$$



**FIGURE 16**  
Reflection of an object about the  $x$  axis.



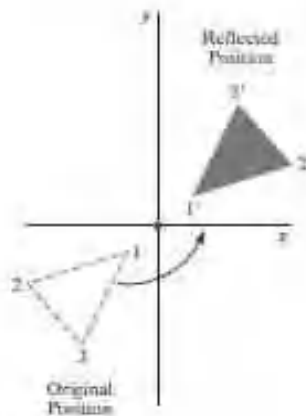
**FIGURE 17**  
Reflection of an object about the  $y$  axis.

Figure 17 illustrates the change in position of an object that has been reflected about the line  $x = 0$ . The equivalent rotation in this case is  $180^\circ$  through three-dimensional space about the  $y$  axis.

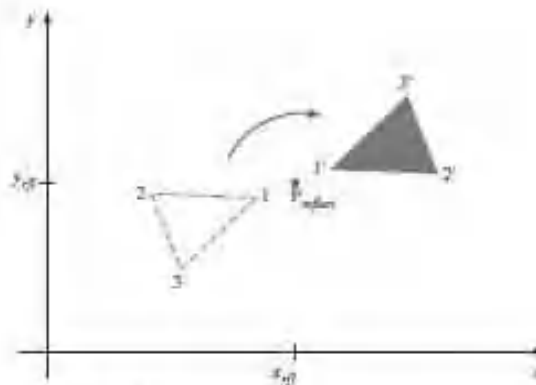
We flip both the  $x$  and  $y$  coordinates of a point by reflecting relative to an axis that is perpendicular to the  $xy$  plane and that passes through the coordinate origin. This reflection is sometimes referred to as a reflection relative to the coordinate origin, and it is equivalent to reflecting with respect to both coordinate axes. The matrix representation for this reflection is

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (54)$$

An example of reflection about the origin is shown in Figure 18. The reflection matrix 54 is the same as the rotation matrix  $\mathbf{R}(\theta)$  with  $\theta = 180^\circ$ . We are simply rotating the object in the  $xy$  plane half a revolution about the origin.



**FIGURE 18**  
Reflection of an object relative to the coordinate origin. This transformation can be accomplished with a rotation in the  $xy$  plane about the coordinate origin.



**FIGURE 19**  
Reflection of an object relative to an axis perpendicular to the  $xy$  plane and passing through point  $P_{\text{reflect}}$ .

Reflection 54 can be generalized to any reflection point in the  $xy$  plane (Figure 19). This reflection is the same as a  $180^\circ$  rotation in the  $xy$  plane about the reflection point.

If we choose the reflection axis as the diagonal line  $y = x$  (Figure 20), the reflection matrix is

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (55)$$

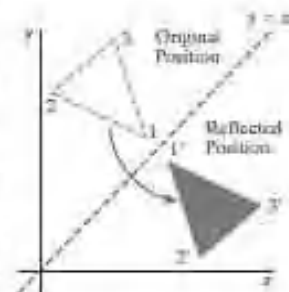
We can derive this matrix by concatenating a sequence of rotation and coordinate-axis reflection matrices. One possible sequence is shown in Figure 21. Here, we first perform a clockwise rotation with respect to the origin through a  $45^\circ$  angle, which rotates the line  $y = x$  onto the  $x$  axis. Next, we perform a reflection with respect to the  $x$  axis. The final step is to rotate the line  $y = x$  back to its original position with a counterclockwise rotation through  $45^\circ$ . Another equivalent sequence of transformations is to first reflect the object about the  $x$  axis, then rotate it counterclockwise  $90^\circ$ .

To obtain a transformation matrix for reflection about the diagonal  $y = -x$ , we could concatenate matrices for the transformation sequence: (1) clockwise rotation by  $45^\circ$ , (2) reflection about the  $y$  axis, and (3) counterclockwise rotation by  $45^\circ$ . The resulting transformation matrix is

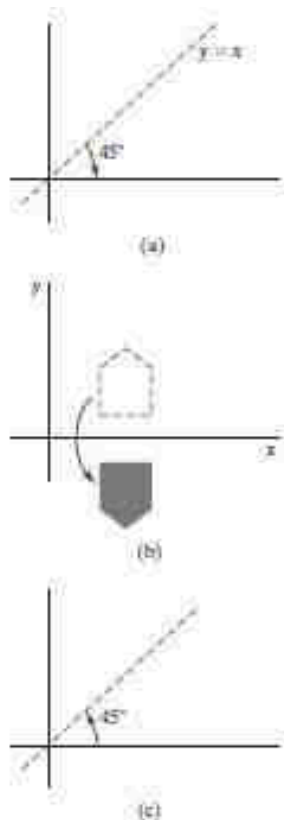
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (56)$$

Figure 22 shows the original and final positions for an object transformed with this reflection matrix.

Reflections about any line  $y = mx + b$  in the  $xy$  plane can be accomplished with a combination of translate-rotate-reflect transformations. In general, we



**FIGURE 20**  
Reflection of an object with respect to the line  $y = x$ .



**FIGURE 21**  
Sequence of transformations to produce a reflection about the line  $y = x$ : (a) A clockwise rotation of  $45^\circ$ , (b) a reflection about the  $x$  axis, and (c) a counter-clockwise rotation by  $45^\circ$ .

first translate the line so that it passes through the origin. Then we can rotate the line onto one of the coordinate axes and reflect about that axis. Finally, we restore the line to its original position with the inverse rotation and translation transformations.

We can implement reflections with respect to the coordinate axes or coordinate origin as scaling transformations with negative scaling factors. Also, elements of the reflection matrix can be set to values other than  $\pm 1$ . A reflection parameter with a magnitude greater than 1 shifts the mirror image of a point farther from the reflection axis, and a parameter with magnitude less than 1 brings the mirror image of a point closer to the reflection axis. Thus, a reflected object can also be enlarged, reduced, or distorted.

### Shear

A transformation that distorts the shape of an object such that the transformed shape appears as if the object were composed of internal layers that had been caused to slide over each other is called a **shear**. Two common shearing transformations are those that shift coordinate  $x$  values and those that shift  $y$  values.

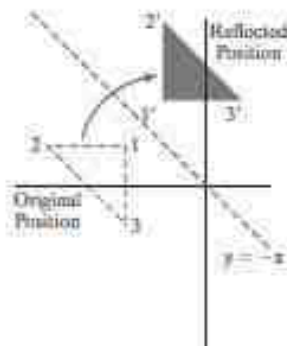
An  $x$ -direction shear relative to the  $x$  axis is produced with the transformation matrix

$$\begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (57)$$

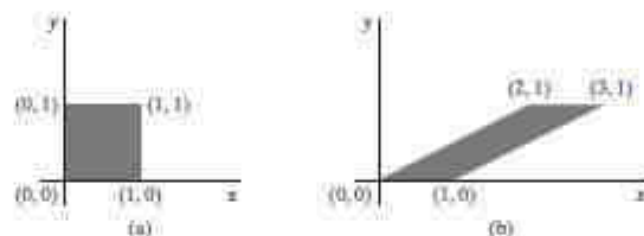
which transforms coordinate positions as

$$x' = x + sh_x \cdot y, \quad y' = y \quad (58)$$

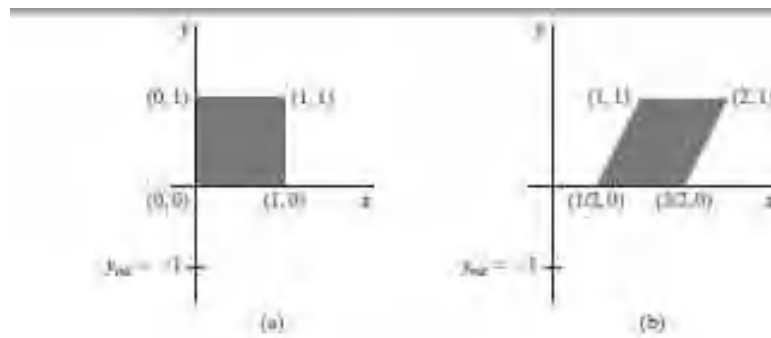
Any real number can be assigned to the shear parameter  $sh_x$ . A coordinate position  $(x, y)$  is then shifted horizontally by an amount proportional to its perpendicular distance ( $y$  value) from the  $x$  axis. Setting parameter  $sh_x$  to the value 2, for example, changes the square in Figure 23 into a parallelogram. Negative values for  $sh_x$  shift coordinate positions to the left.



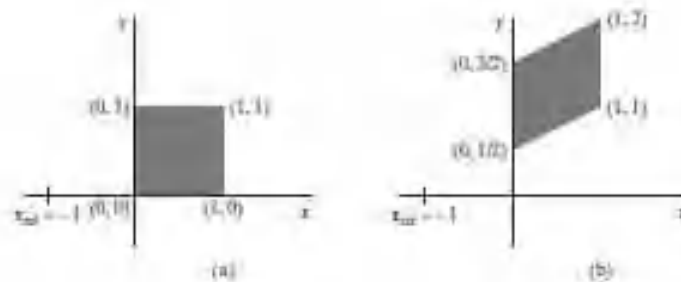
**FIGURE 22**  
Reflection with respect to the line  $y = -x$ .



**FIGURE 23**  
A unit square (a) is converted to a parallelogram (b) using the  $x$ -direction shear matrix 57 with  $sh_x = 2$ .



**FIGURE 24**  
A unit square (a) is transformed to a shifted parallelogram (b) with parameter values  $sh_x = 0.5$  and  $y_{ref} = -1$  in the shear matrix 56.



**FIGURE 25**  
A unit square (a) is transformed to a shifted parallelogram (b) with parameter values  $sh_y = 0.5$  and  $x_{ref} = -1$  in the  $y$ -direction shearing transformation 61.

We can generate  $x$ -direction shears relative to other reference lines with

$$\begin{bmatrix} 1 & sh_x & -sh_x \cdot y_{ref} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (59)$$

Now, coordinate positions are transformed as

$$x' = x + sh_x(y - y_{ref}), \quad y' = y \quad (60)$$

An example of this shearing transformation is given in Figure 24 for a shear parameter value of  $\frac{1}{2}$  relative to the line  $y_{ref} = -1$ .

A  $y$ -direction shear relative to the line  $x = x_{ref}$  is generated with the transformation matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & -sh_y \cdot x_{ref} \\ 0 & 0 & 1 \end{bmatrix} \quad (61)$$

which generates the transformed coordinate values

$$x' = x, \quad y' = y + sh_y(x - x_{ref}) \quad (62)$$

This transformation shifts a coordinate position vertically by an amount proportional to its distance from the reference line  $x = x_{ref}$ . Figure 25 illustrates the conversion of a square into a parallelogram with  $sh_y = 0.5$  and  $x_{ref} = -1$ .

Shearing operations can be expressed as sequences of basic transformations. The  $x$ -direction shear matrix 57, for example, can be represented as a composite transformation involving a series of rotation and scaling matrices. This composite transformation scales the unit square of Figure 23 along its diagonal, while maintaining the original lengths and orientations of edges parallel to the  $x$  axis. Shifts in the positions of objects relative to shearing reference lines are equivalent to translations.



## TWO DIMENSIONAL VIEWING

We now examine in more detail the procedures for displaying views of a two-dimensional picture on an output device. Typically, a graphics package allows a user to specify which part of a defined picture is to be displayed and where that part is to be placed on the display device. Any convenient Cartesian coordinate system, referred to as the world-coordinate reference frame, can be used to define the picture. For a two-dimensional picture, a view is selected by specifying a region of the  $xy$  plane that contains the total picture or any part of it. A user can select a single area for display, or several areas could be selected for simultaneous display or for an animated panning sequence across a scene.

The picture parts within the selected areas are then mapped onto specified areas of the device coordinates. When multiple view areas are selected, these areas can be placed in separate display locations, or some areas could be inserted into other, larger display areas. Two-dimensional viewing transformations from world to device coordinates involve translation, rotation, and scaling operations, as well as procedures for deleting those parts of the picture that are outside the limits of a selected scene area.

### WINDOW - TO- VIEWPORT CO-ORDINATE TRANSFORMATION.

#### Normalization and Viewport Transformations

With some graphics packages, the normalization and window-to-viewport transformations are combined into one operation. In this case, the viewport coordinates are often given in the range from 0 to 1 so that the viewport is positioned within a unit square. After clipping, the unit square containing the viewport is mapped to the output display device. In other systems, the normalization and clipping routines are applied before the viewport transformation. For these systems, the viewport boundaries are specified in screen coordinates relative to the display window position.

#### Mapping the Clipping Window into a Normalized Viewport

To illustrate the general procedures for the normalization and viewport transformations, we first consider a viewport defined with normalized coordinate values between 0 and 1. Object descriptions are transferred to this normalized space using a transformation that maintains the same relative placement of a point in

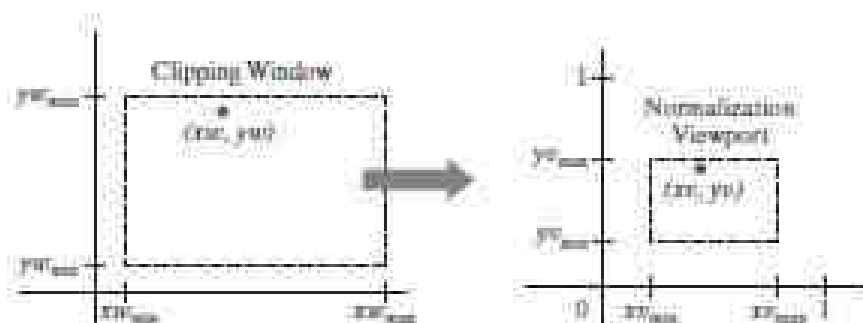


FIGURE 6

A point  $(x_w, y_w)$  in a world-coordinate clipping window  $b$  is mapped to viewport coordinates  $(x_v, y_v)$  within a unit square, so that the relative positions of the two points in their respective rectangles are the same.



the viewport as it had in the clipping window. If a coordinate position is at the center of the clipping window, for instance, it would be mapped to the center of the viewport. Figure 6 illustrates this window-to-viewport mapping. Position  $(x_w, y_w)$  in the clipping window is mapped to position  $(x_v, y_v)$  in the associated viewport.

To transform the world-coordinate point into the same relative position within the viewport, we require that

$$\frac{x_v - x_{vmin}}{x_{vmax} - x_{vmin}} = \frac{x_w - x_{wmin}}{x_{wmax} - x_{wmin}} \quad (2)$$

$$\frac{y_v - y_{vmin}}{y_{vmax} - y_{vmin}} = \frac{y_w - y_{wmin}}{y_{wmax} - y_{wmin}}$$

Solving these expressions for the viewport position  $(x_v, y_v)$ , we have

$$x_v = s_x x_w + t_x \quad (3)$$

$$y_v = s_y y_w + t_y$$

where the scaling factors are

$$s_x = \frac{x_{vmax} - x_{vmin}}{x_{wmax} - x_{wmin}} \quad (4)$$

$$s_y = \frac{y_{vmax} - y_{vmin}}{y_{wmax} - y_{wmin}}$$

and the translation factors are

$$t_x = \frac{x_{vmax} y_{vmin} - x_{vmin} y_{vmax}}{x_{wmax} - x_{wmin}} \quad (5)$$

$$t_y = \frac{y_{vmax} y_{vmin} - y_{vmin} y_{vmax}}{y_{wmax} - y_{wmin}}$$

Because we are simply mapping world-coordinate positions into a viewport that is positioned near the world origin, we can also derive Equations 3 using any transformation sequence that converts the rectangle for the clipping window into the viewport rectangle. For example, we could obtain the transformation from world coordinates to viewport coordinates with the following sequence:

1. Scale the clipping window to the size of the viewport using a fixed-point position of  $(x_{wmin}, y_{wmin})$ .
2. Translate  $(x_{wmin}, y_{wmin})$  to  $(x_{vmin}, y_{vmin})$ .

The scaling transformation in step (1) can be represented with the two-dimensional matrix

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & x_{wmin}(1-s_x) \\ 0 & s_y & y_{wmin}(1-s_y) \\ 0 & 0 & 1 \end{bmatrix} \quad (6)$$

where  $s_x$  and  $s_y$  are the same as in Equations 4. The two-dimensional matrix representation for the translation of the lower-left corner of the clipping window to the lower-left viewport corner is

$$T = \begin{bmatrix} 1 & 0 & x_{\text{view}} - x_{\text{win}} \\ 0 & 1 & y_{\text{view}} - y_{\text{win}} \\ 0 & 0 & 1 \end{bmatrix} \quad (7)$$

And the composite matrix representation for the transformation to the normalized viewport is

$$M_{\text{window to viewport}} = T \cdot S = \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix} \quad (8)$$

which gives us the same result as in Equations 3. Any other clipping-window reference point, such as the top-right corner or the window center, could be used for the scale-translate operations. Alternatively, we could first translate any clipping-window position to the corresponding location in the viewport, and then scale relative to that viewport location.

The window-to-viewport transformation maintains the relative placement of object descriptions. An object inside the clipping window is mapped to a corresponding position inside the viewport. Similarly, an object outside the clipping window is outside the viewport.

Relative proportions of objects, on the other hand, are maintained only if the aspect ratio of the viewport is the same as the aspect ratio of the clipping window.

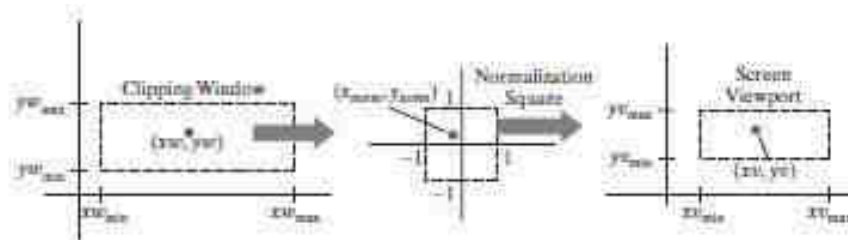
In other words, we keep the same object proportions if the scaling factors  $s_x$  and  $s_y$  are the same. Otherwise, world objects will be stretched or contracted in either the  $x$  or  $y$  directions (or both) when displayed on the output device.

The clipping routines can be applied using either the clipping-window boundaries or the viewport boundaries. After clipping, the normalized coordinates are transformed into device coordinates. And the unit square can be mapped onto the output device using the same procedures as in the window-to-viewport transformation, with the area inside the unit square transferred to the total display area of the output device.

### Mapping the Clipping Window into a Normalized Square

Another approach to two-dimensional viewing is to transform the clipping window into a normalized square, clip in normalized coordinates, and then transfer the scene description to a viewport specified in screen coordinates. This transformation is illustrated in Figure 7 with normalized coordinates in the range from  $-1$  to  $1$ . The clipping algorithms in this transformation sequence are now standardized so that objects outside the boundaries  $x = \pm 1$  and  $y = \pm 1$  are detected and removed from the scene description. At the final step of the viewing transformation, the objects in the viewport are positioned within the display window.

We transfer the contents of the clipping window into the normalization square using the same procedures as in the window-to-viewport transformation. The matrix for the normalization transformation is obtained from Equation 8 by substituting  $-1$  for  $x_{\text{min}}$  and  $y_{\text{min}}$  and substituting  $+1$  for  $x_{\text{max}}$  and  $y_{\text{max}}$ .



**FIGURE 7**

A point  $(x_w, y_w)$  in a clipping window is mapped to a normalized coordinate position  $(x_{norm}, y_{norm})$ , then to a screen-coordinate position  $(x_v, y_v)$  in a viewport. Objects are clipped against the normalization square before the transformation to viewport coordinates occurs.

Making these substitutions in the expressions for  $t_x$ ,  $t_y$ ,  $s_x$ , and  $s_y$ , we have:

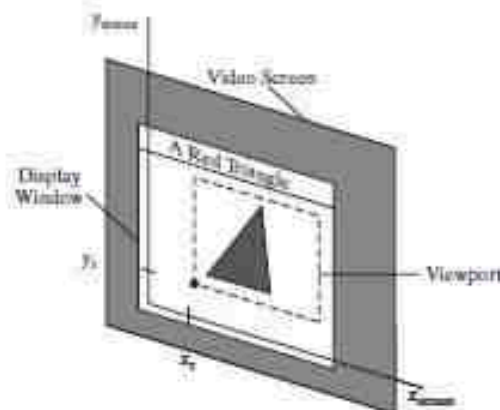
$$M_{\text{window, normsquare}} = \begin{bmatrix} \frac{2}{xw_{max} - xw_{min}} & 0 & \frac{xw_{max} + xw_{min}}{xw_{max} - xw_{min}} \\ 0 & \frac{2}{yw_{max} - yw_{min}} & \frac{yw_{max} + yw_{min}}{yw_{max} - yw_{min}} \\ 0 & 0 & 1 \end{bmatrix} \quad (9)$$

Similarly, after the clipping algorithms have been applied, the normalized square with edge length equal to 2 is transformed into a specified viewport. This time, we get the transformation matrix from Equation 8 by substituting  $-1$  for  $xw_{min}$  and  $yw_{min}$  and substituting  $+1$  for  $xw_{max}$  and  $yw_{max}$ :

$$M_{\text{normsquare, viewport}} = \begin{bmatrix} \frac{xv_{max} - xv_{min}}{2} & 0 & \frac{xv_{max} + xv_{min}}{2} \\ 0 & \frac{yv_{max} - yv_{min}}{2} & \frac{yv_{max} + yv_{min}}{2} \\ 0 & 0 & 1 \end{bmatrix} \quad (10)$$

The last step in the viewing process is to position the viewport area in the display window. Typically, the lower-left corner of the viewport is placed at a coordinate position specified relative to the lower-left corner of the display window. Figure 8 demonstrates the positioning of a viewport within a display window.

As before, we maintain the initial proportions of objects by choosing the aspect ratio of the viewport to be the same as the clipping window. Otherwise, objects



**FIGURE 8**  
A viewport at coordinate position  $(x_v, y_v)$  within a display window.

will be stretched or contracted in the  $x$  or  $y$  directions. Also, the aspect ratio of the display window can affect the proportions of objects. If the viewport is mapped to the entire area of the display window and the size of the display window is changed, objects may be distorted unless the aspect ratio of the viewport is also adjusted.

### **Display of Character Strings**

Character strings can be handled in one of two ways when they are mapped through the viewing pipeline to a viewport. The simplest mapping maintains a constant character size. This method could be employed with bitmap character patterns. But outline fonts could be transformed the same as other primitives; we just need to transform the defining positions for the line segments in the outline character shapes. Algorithms for determining the pixel patterns for the transformed characters are then applied when the other primitives in the scene are processed.

### **Split-Screen Effects and Multiple Output Devices**

By selecting different clipping windows and associated viewports for a scene, we can provide simultaneous display of two or more objects, multiple picture parts, or different views of a single scene. And we can position these views in different parts of a single display window or in multiple display windows on the screen.

In a design application, for example, we can display a wire-frame view of an object in one viewport while also displaying a fully rendered view of the object in another viewport. In addition, we could list other information or menus in a third viewport.

It is also possible that two or more output devices could be operating concurrently on a particular system, and we can set up a clipping-window/viewport pair for each output device. A mapping to a selected output device is sometimes referred to as a **workstation transformation**. In this case, viewports could be specified in the coordinates of a particular display device, or each viewport could be specified within a unit square, which is then mapped to a chosen output device. Some graphics systems provide a pair of workstation functions for this purpose. One function is used to designate a clipping window for a selected output device, identified by a *workstation number*, and the other function is used to set the associated viewport for that device.

## **UNIT 3: CLIPPING ALGORITHMS**

Generally, any procedure that eliminates those portions of a picture that are either inside or outside a specified region of space is referred to as a **clipping algorithm** or simply **clipping**. Usually a clipping region is a rectangle in standard position, although we could use any shape for a clipping application.

The most common application of clipping is in the viewing pipeline, where clipping is applied to extract a designated portion of a scene (either two-dimensional or three-dimensional) for display on an output device. Clipping methods are also used to antialias object boundaries, to construct objects using solid-modeling methods, to manage a multiwindow

environment, and to allow parts of a picture to be moved, copied, or erased in drawing and painting programs.

Clipping algorithms are applied in two-dimensional viewing procedures to identify those parts of a picture that are within the clipping window. Everything outside the clipping window is then eliminated from the scene description that is transferred to the output device for display. An efficient implementation of clipping in the viewing pipeline is to apply the algorithms to the normalized boundaries of the clipping window. This reduces calculations, because all geometric and viewing transformation matrices can be concatenated and applied to a scene description before clipping is carried out. The clipped scene can then be transferred to screen coordinates for final processing.

In the following sections, we explore two-dimensional algorithms for

- Point clipping
- Line clipping (straight-line segments)
- Fill-area clipping (polygons)
- Curve clipping
- Text clipping

Point, line, and polygon clipping are standard components of graphics packages. But similar methods can be applied to other objects, particularly conics, such as circles, ellipses, and spheres, in addition to spline curves and surfaces. Usually, however, objects with nonlinear boundaries are approximated with straight-line segments or polygon surfaces to reduce computations.

Unless otherwise stated, we assume that the clipping region is a rectangular window in standard position, with boundary edges at coordinate positions  $x_{\min}$ ,  $x_{\max}$ ,  $y_{\min}$ , and  $y_{\max}$ . These boundary edges typically correspond to a normalized square, in which the  $x$  and  $y$  values range either from 0 to 1 or from  $-1$  to 1.

### **POINT CLIPPING**

For a clipping rectangle in standard position, we save a two-dimensional point  $\mathbf{P} = (x, y)$  for display if the following inequalities are satisfied:

$$\begin{aligned} x_{\min} \leq x \leq x_{\max} \\ y_{\min} \leq y \leq y_{\max} \end{aligned} \quad (12)$$

If any of these four inequalities is not satisfied, the point is clipped (not saved for display).

Although point clipping is applied less often than line or polygon clipping, it is useful in various situations, particularly when pictures are modeled with particle systems. For example, point clipping can be applied to scenes involving clouds, sea foam, smoke, or explosions that are modeled with “particles,” such as the center coordinates for small circles or spheres.

### **LINE CLIPPING**

Figure 9 illustrates possible positions for straight-line segments in relationship to a standard clipping window. A line-clipping algorithm processes each line in a scene through a series of tests and intersection calculations to determine whether the entire line or any part of it is to be saved. The expensive part of a line-clipping procedure is in calculating the intersection positions of a line with the window edges. Therefore, a major goal for any line-clipping algorithm is to minimize the intersection



calculations. To do this, we can first perform tests to determine whether a line segment is completely inside the clipping window or completely outside. It is easy to determine whether a line is completely inside a clipping window, but it is more difficult to identify all lines that are entirely outside the window. If we are unable to identify a line as completely inside or completely

outside a clipping rectangle, we must then perform intersection calculations to determine whether any part of the line crosses the window interior.

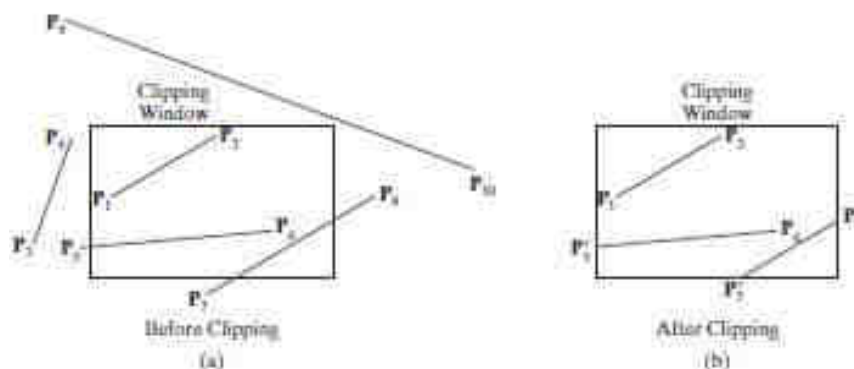
We test a line segment to determine if it is completely inside or outside a selected clipping-window edge by applying the point-clipping tests of the previous section. When both endpoints of a line segment are inside all four clipping boundaries, such as the line from **P1** to **P2** in Figure 9, the line is completely inside the clipping window and we save it. And when both endpoints of a line segment are outside any one of the four boundaries (as with line **P3P4** in Figure 9), that line is completely outside the window and it is eliminated from the scene description.

But if both these tests fail, the line segment intersects at least one clipping boundary and it may or may not cross into the interior of the clipping window.

One way to formulate the equation for a straight-line segment is to use the following parametric representation, where the coordinate positions  $(x_0, y_0)$  and  $(x_{end}, y_{end})$  designate the two line endpoints:

$$\begin{aligned} x &= x_0 + u(x_{end} - x_0) \\ y &= y_0 + u(y_{end} - y_0) \quad 0 \leq u \leq 1 \quad (13) \end{aligned}$$

We can use this parametric representation to determine where a line segment crosses each clipping-window edge by assigning the coordinate value for that edge to either  $x$  or  $y$  and solving for parameter  $u$ . For example, the left window boundary is at position  $x_{min}$ , so we substitute this value for  $x$ , solve for  $u$ , and calculate the corresponding  $y$ -intersection value. If this value of  $u$  is outside the range from 0 to 1, the line segment does not intersect that window border line.



**FIGURE 9**  
Clipping straight-line segments using a standard rectangular clipping window.

However, if the value of  $u$  is within the range from 0 to 1, part of the line is inside that border. We can then process this inside portion of the line segment against the other clipping boundaries until either we have clipped the entire line or we find a section that is inside the window.

Processing line segments in a scene using the simple clipping approach described in the preceding paragraph is straightforward, but not very efficient.

It is possible to reformulate the initial testing and the intersection calculations to reduce processing time for a set of line segments, and a number of faster lineclippers have been developed. Some of the algorithms are designed explicitly for two-dimensional pictures and some are easily adapted to sets of three-dimensional line segments.

### Cohen-Sutherland Line Clipping

This is one of the earliest algorithms to be developed for fast line clipping, and variations of this method are widely used. Processing time is reduced in the Cohen-Sutherland method by performing more tests before proceeding to the intersection calculations. Initially, every line endpoint in a picture is assigned a four-digit binary value, called a **region code**, and each bit position is used to indicate whether the point is inside or outside one of the clipping-window boundaries.

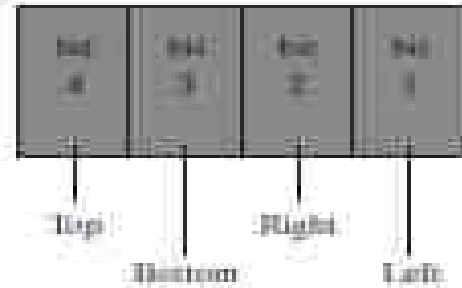


FIGURE 10

A possible ordering for the clipping-window boundaries corresponding to the bit positions in the Cohen-Sutherland endpoint region code.

We can reference the window edges in any order, and Figure 10 illustrates one possible ordering with the bit positions numbered 1 through 4 from right to left. Thus, for this ordering, the rightmost position (bit 1) references the left clipping-window boundary, and the leftmost position (bit 4) references the top window boundary. A value of 1 (or *true*) in any bit position indicates that the endpoint is outside that window border. Similarly, a value of 0 (or *false*) in any bit position indicates that the endpoint is not outside (it is inside or on) the corresponding window edge. Sometimes, a region code is referred to as an “**out**” code because a value of 1 in any bit position indicates that the spatial point is outside the corresponding clipping boundary.

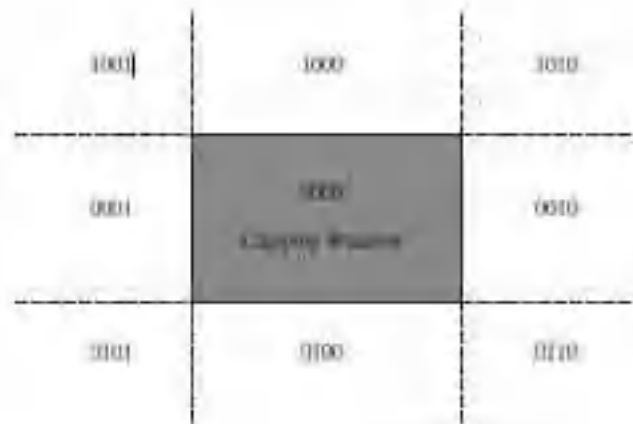


FIGURE 11

The nine binary region codes for identifying the position of a line endpoint, relative to the clipping-window boundaries.

Each clipping-window edge divides two-dimensional space into an inside half space and an outside half space. Together, the four window borders create nine regions, and Figure 11 lists the value for the binary code in each of these regions. Thus, an endpoint that is below and to the left of the clipping window is assigned the region code 0101, and the region-code value for any endpoint inside the clipping window is 0000.

Bit values in a region code are determined by comparing the coordinate values  $(x, y)$  of an endpoint to the clipping boundaries. Bit 1 is set to 1 if  $x < x_{\min}$ , and the other three bit values are determined similarly. Instead of using inequality testing, we can determine the values for a region-code more efficiently using bit-processing operations and the following two steps: (1) Calculate differences between endpoint coordinates and clipping boundaries. (2) Use the resultant sign bit of each difference calculation to set the corresponding value in the region code.

For the ordering scheme shown in Figure 10, bit 1 is the sign bit of  $x - x_{\min}$ ; bit 2 is the sign bit of  $x_{\max} - x$ ; bit 3 is the sign bit of  $y - y_{\min}$ ; and bit 4 is the sign bit of  $y_{\max} - y$ .

Once we have established region codes for all line endpoints, we can quickly determine which lines are completely inside the clip window and which are completely outside. Any lines that are completely contained within the window edges have a region code of 0000 for both endpoints, and we save these line segments.

Any line that has a region-code value of 1 in the same bit position for each endpoint is completely outside the clipping rectangle, and we eliminate that line segment.

As an example, a line that has a region code of 1001 for one endpoint and a code of 0101 for the other endpoint is completely to the left of the clipping window, as indicated by the value of 1 in the first bit position of each region code.

We can perform the inside-outside tests for line segments using logical operators.

When the *or* operation between two endpoint region codes for a line segment is *false* (0000), the line is inside the clipping window. Therefore, we save the line and proceed to test the next line in the scene description. When the *and* operation between the two endpoint region codes for a line is *true* (not 0000), the line is completely outside the clipping window, and we can eliminate it from the scene description.

Lines that cannot be identified as being completely inside or completely

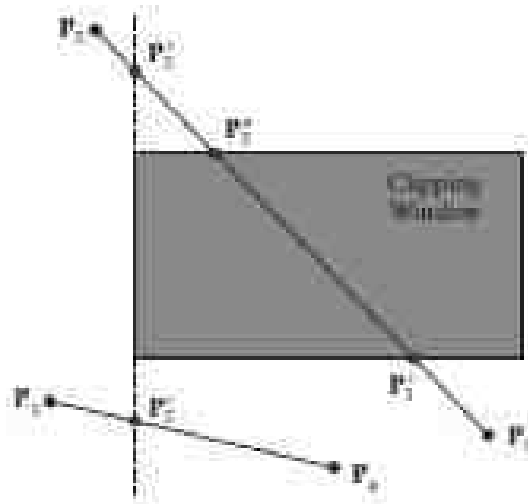


FIGURE 12

Lines extending from one clipping-window region to another may cross into the clipping window, or they could intersect one or more clipping boundaries without entering the window.

outside a clipping window by the region-code tests are next checked for intersection with the window border lines. As shown in Figure 12, line

segments can intersect clipping boundary lines without entering the interior of the window.

Therefore, several intersection calculations might be necessary to clip a line segment, depending on the order in which we process the clipping boundaries. As we process each clipping-window edge, a section of the line is clipped, and the remaining part of the line is checked against the other window borders. We continue eliminating sections until either the line is totally clipped or the remaining part of the line is inside the clipping window. For the following discussion, we assume that the window edges are processed in the following order: left, right, bottom, top. To determine whether a line crosses a selected clipping boundary, we can check corresponding bit values in the two endpoint region codes. If one of these bit values is 1 and the other is 0, the line segment crosses that boundary.

Figure 12 illustrates two line segments that cannot be identified immediately as completely inside or completely outside the clipping window. The region codes for the line from  $P_1$  to  $P_2$  are 0100 and 1001. Thus,  $P_1$  is inside the left clipping boundary and  $P_2$  is outside that boundary. We then calculate the intersection position  $P'_2$ , and we clip off the line section from  $P_2$  to  $P'_2$ . The remaining portion of the line is inside the right border line, and so we next check the bottom border. Endpoint  $P_1$  is below the bottom clipping edge and  $P'_2$  is above it, so we determine the intersection position at this boundary ( $P'_1$ ). We eliminate the line section from  $P_1$  to  $P'_1$  and proceed to the top window edge. There we determine the intersection position to be  $P'_2$ . The final step is to clip off the section above the top boundary and save the interior segment from  $P'_1$  to  $P'_2$ . For the second line, we find that point  $P_3$  is outside the left boundary and  $P_4$  is inside. Thus, we calculate the intersection position  $P'_3$  and eliminate the line section from  $P_3$  to  $P'_3$ . By checking region codes for the endpoints  $P'_3$  and  $P_4$ , we find that the remainder of the line is below the clipping window and can be eliminated as well.

It is possible, when clipping a line segment using this approach, to calculate an intersection position at all four clipping boundaries, depending on how the line endpoints are processed and what ordering we use for the boundaries. Figure 13 shows the four intersection positions that could be calculated for a line segment that is processed against the clipping-window edges in the order left, right, bottom, top. Therefore, variations of this basic approach have been developed in an effort to reduce the intersection calculations.

To determine a boundary

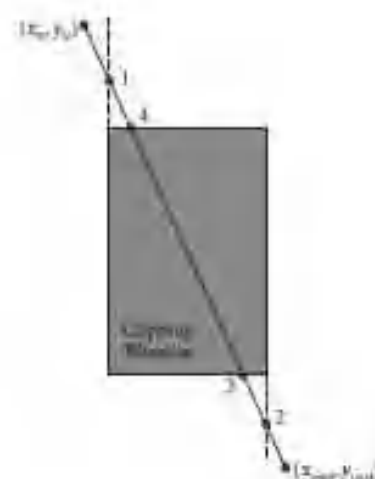


FIGURE 13  
Four intersection positions (labeled from 1 to 4) for a line segment that is clipped against the window boundaries in the order left, right, bottom, top.

intersection for a line segment, we can use the slope-intercept form of the line equation. For a line with endpoint coordinates  $(x_0, y_0)$  and  $(x_{end}, y_{end})$ , the  $y$  coordinate of the intersection point with a vertical clipping border line can be obtained with the calculation

$$y = y_0 + m(x - x_0) \quad (14)$$

where the  $x$  value is set to either  $x_{wmin}$  or  $x_{wmax}$ , and the slope of the line is calculated as

$$m = (y_{end} - y_0) / (x_{end} - x_0).$$

Similarly, if we are looking for the intersection with a horizontal border, the  $x$  coordinate can be calculated as

$$x = x_0 + (y - y_0) / m \quad (15)$$

with  $y$  set either to  $y_{wmin}$  or to  $y_{wmax}$ .

An implementation of the two-dimensional, Cohen-Sutherland line-clipping algorithm is given in the following procedures.

```
class wcPt2D
{
    public:
    GLfloat x, y;
};
inline GLint round (const GLfloat a) { return GLint (a + 0.5); }
/* Define a four-bit code for each of the outside regions of a
* rectangular clipping window.
*/
const GLint winLeftBitCode = 0x1;
const GLint winRightBitCode = 0x2;
const GLint winBottomBitCode = 0x4;
const GLint winTopBitCode = 0x8;
/* A bit-mask region code is also assigned to each endpoint of an input
* line segment, according to its position relative to the four edges of
* an input rectangular clip window.
**
An endpoint with a region-code value of 0000 is inside the clipping
* window, otherwise it is outside at least one clipping boundary. If
* the 'or' operation for the two endpoint codes produces a value of
* false, the entire line defined by these two endpoints is saved
* (accepted). If the 'and' operation between two endpoint codes is
* true, the line is completely outside the clipping window, and it is
* eliminated (rejected) from further processing.
*/
inline GLint inside (GLint code) { return GLint (!code); }
inline GLint reject (GLint code1, GLint code2)
{ return GLint (code1 & code2); }
inline GLint accept (GLint code1, GLint code2)
{ return GLint (!(code1 | code2)); }
GLubyte encode (wcPt2D pt, wcPt2D winMin, wcPt2D winMax)
{
    GLubyte code = 0x00;
    if (pt.x < winMin.x)
        code = code | winLeftBitCode;
    if (pt.x > winMax.x)
```



```

        code = code | winRightBitCode;
    if (pt.y < winMin.y)
        code = code | winBottomBitCode;
    if (pt.y > winMax.y)
        code = code | winTopBitCode;
    return (code);
}
void swapPts (wcPt2D * p1, wcPt2D * p2)
{
    wcPt2D tmp;
    tmp = *p1; *p1 = *p2; *p2 = tmp;
}
void swapCodes (GLubyte * c1, GLubyte * c2)
{
    GLubyte tmp;
    tmp = *c1; *c1 = *c2; *c2 = tmp;
}
void lineClipCohSuth (wcPt2D winMin, wcPt2D winMax, wcPt2D p1,
wcPt2D p2)
{
    GLubyte code1, code2;
    GLint done = false, plotLine = false;
    GLfloat m;
    while (!done) {
        code1 = encode (p1, winMin, winMax);
        code2 = encode (p2, winMin, winMax);
        if (accept (code1, code2))
        {
            done = true;
            plotLine = true;
        }
        else
            if (reject (code1, code2))
                done = true;
            else
            {
                /* Label the endpoint outside the display window as
                p1. */
                if (inside (code1))
                {
                    swapPts (&p1, &p2);
                    swapCodes (&code1, &code2);
                }
                /* Use slope m to find line-clipEdge intersection. */
                if (p2.x != p1.x)
                    m = (p2.y - p1.y) / (p2.x - p1.x);
                if (code1 & winLeftBitCode)
                {
                    p1.y += (winMin.x - p1.x) * m;
                    p1.x = winMin.x;
                }
                else
                    if (code1 & winRightBitCode)
                    {
                        p1.y += (winMax.x - p1.x) * m;
                        p1.x = winMax.x;
                    }
            }
        }
}

```

```

    }
    else
        if (code1 & winBottomBitCode)
        {
            /* Need to update p1.x for nonvertical
            lines only. */
            if (p2.x != p1.x)
                p1.x += (winMin.y - p1.y) / m;
            p1.y = winMin.y;
        }
        else
            if (code1 & winTopBitCode)
            {
                if (p2.x != p1.x)
                    p1.x += (winMax.y - p1.y) / m;
                p1.y = winMax.y;
            }
        }
    }
}
if (plotLine)
    lineBres (round (p1.x), round (p1.y), round (p2.x),
              round (p2.y));
}

```

### Liang-Barsky Line Clipping

Faster line-clipping algorithms have been developed that do more line testing before proceeding to the intersection calculations. One of the earliest efforts in this direction is an algorithm developed by Cyrus and Beck, which is based on an analysis of the parametric line equations. Later, Liang and Barsky independently devised an even faster form of the parametric line-clipping algorithm.

For a line segment with endpoints  $(x_0, y_0)$  and  $(x_{end}, y_{end})$ , we can describe the line with the parametric form

$$\begin{aligned} x &= x_0 + u_x \\ y &= y_0 + u_y \quad 0 \leq u \leq 1 \end{aligned} \quad (16)$$

where  $_x = x_{end} - x_0$  and  $_y = y_{end} - y_0$ . In the Liang-Barsky algorithm, the parametric line equations are combined with the point-clipping conditions 12 to obtain the inequalities

$$\begin{aligned} x_{wmin} &\leq x_0 + u_x \leq x_{wmax} \\ y_{wmin} &\leq y_0 + u_y \leq y_{wmax} \end{aligned} \quad (17)$$

which can be expressed as

$$u p_k \leq q_k, \quad k = 1, 2, 3, 4 \quad (18)$$

where parameters  $p$  and  $q$  are defined as

$$\begin{aligned} p_1 &= -_x, \quad q_1 = x_0 - x_{wmin} \\ p_2 &= _x, \quad q_2 = x_{wmax} - x_0 \\ p_3 &= -_y, \quad q_3 = y_0 - y_{wmin} \\ p_4 &= _y, \quad q_4 = y_{wmax} - y_0 \end{aligned} \quad (19)$$

Any line that is parallel to one of the clipping-window edges has  $p_k = 0$  for the value of  $k$  corresponding to that boundary, where  $k = 1, 2, 3,$  and  $4$  correspond to the left, right, bottom, and top boundaries, respectively. If, for that value of  $k$ , we also find  $q_k < 0$ , then the line is completely outside the

boundary and can be eliminated from further consideration. If  $qk \geq 0$ , the line is inside the parallel clipping border.

When  $pk < 0$ , the infinite extension of the line proceeds from the outside to the inside of the infinite extension of this particular clipping-window edge. If  $pk > 0$ , the line proceeds from the inside to the outside. For a nonzero value of  $pk$ , we can calculate the value of  $u$  that corresponds to the point where the infinitely extended line intersects the extension of window edge  $k$  as

$$u = qk/pk \quad (20)$$

For each line, we can calculate values for parameters  $u1$  and  $u2$  that define that part of the line that lies within the clip rectangle. The value of  $u1$  is determined by looking at the rectangle edges for which the line proceeds from the outside to the inside ( $p < 0$ ). For these edges, we calculate  $r_k = qk/pk$ . The value of  $u1$  is taken as the largest of the set consisting of 0 and the various values of  $r$ . Conversely, the value of  $u2$  is determined by examining the boundaries for which the line proceeds from inside to outside ( $p > 0$ ). A value of  $r_k$  is calculated for each of these boundaries, and the value of  $u2$  is the minimum of the set consisting of 1 and the calculated  $r$  values. If  $u1 > u2$ , the line is completely outside the clip window and it can be rejected. Otherwise, the endpoints of the clipped line are calculated from the two values of parameter  $u$ .

This algorithm is implemented in the following code sections. Line intersection parameters are initialized to the values  $u1 = 0$  and  $u2 = 1$ . For each clipping boundary, the appropriate values for  $p$  and  $q$  are calculated and used by the function **clipTest** to determine whether the line can be rejected or whether the intersection parameters are to be adjusted. When  $p < 0$ , parameter  $r$  is used to update  $u1$ ; when  $p > 0$ , parameter  $r$  is used to update  $u2$ . If updating  $u1$  or  $u2$  results in  $u1 > u2$ , we reject the line. Otherwise, we update the appropriate  $u$  parameter only if the new value results in a shortening of the line. When  $p = 0$  and  $q < 0$ , we can eliminate the line because it is parallel to and outside this boundary.

If the line has not been rejected after all four values of  $p$  and  $q$  have been tested, the endpoints of the clipped line are determined from values of  $u1$  and  $u2$ .

```
class wcPt2D
{
  private:
  GLfloat x, y;
  public:
  /* Default Constructor: initialize position as (0.0, 0.0). */
  wcPt3D ( )
  {
    x = y = 0.0;
  }
  setCoords (GLfloat xCoord, GLfloat yCoord)
  {
    x = xCoord;
    y = yCoord;
  }
  GLfloat getX ( ) const
  {
    return x;
  }
}
```

```

    GLfloat gety ( ) const
    {
        return y;
    }
};
inline GLint round (const GLfloat a)
{ return GLint (a + 0.5);
}
GLint clipTest (GLfloat p, GLfloat q, GLfloat * u1, GLfloat * u2)
{
    GLfloat r;
    GLint returnValue = true;
    if (p < 0.0)
    {
        r = q / p;
        if (r > *u2)
            returnValue = false;
        else
            if (r > *u1)
                *u1 = r;
    }
    else
        if (p > 0.0)
        {
            r = q / p;
            if (r < *u1)
                returnValue = false;
            else if (r < *u2)
                *u2 = r;
        }
    else
        /* Thus p = 0 and line is parallel to clipping boundary. */
        if (q < 0.0)
            /* Line is outside clipping boundary. */
            returnValue = false;
        return (returnValue);
}
void lineClipLiangBarsk (wcPt2D winMin, wcPt2D winMax, wcPt2D p1,
wcPt2D p2)
{
    GLfloat u1 = 0.0, u2 = 1.0, dx = p2.getx ( ) - p1.getx ( ), dy;
    if (clipTest (-dx, p1.getx ( ) - winMin.getx ( ), &u1, &u2))
    if (clipTest (dx, winMax.getx ( ) - p1.getx ( ), &u1, &u2))
    {
        dy = p2.gety ( ) - p1.gety ( );
        if (clipTest (-dy, p1.gety ( ) - winMin.gety ( ), &u1, &u2))
        if (clipTest (dy, winMax.gety ( ) - p1.gety ( ), &u1, &u2)) {
            if (u2 < 1.0)
                {
                    p2.setCoords (p1.getx ( ) + u2 * dx, p1.gety
                        ( ) + u2 * dy);
                }
            if (u1 > 0.0)
                {
                    p1.setCoords (p1.getx ( ) + u1 * dx, p1.gety ( ) +
                        u1 * dy);
                }
        }
    }
}

```

```

        lineBres (round (p1.getx ( )), round (p1.gety ( )),
                round (p2.getx ( )), round (p2.gety ( )));
    }
}
}

```

In general, the Liang-Barsky algorithm is more efficient than the Cohen-Sutherland line-clipping algorithm. Each update of parameters  $u_1$  and  $u_2$  requires only one division; and window intersections of the line are computed only once, when the final values of  $u_1$  and  $u_2$  have been computed. In contrast, the Cohen and Sutherland algorithm can calculate intersections repeatedly along a line path, even though the line may be completely outside the clip window. In addition, each Cohen-Sutherland intersection calculation requires both a division and a multiplication.

The two-dimensional Liang-Barsky algorithm can be extended to clip three-dimensional lines.

### Nicholl-Lee-Nicholl Line Clipping

By creating more regions around the clipping window, the Nicholl-Lee-Nicholl (NLN) algorithm avoids multiple line-intersection calculations. In the Cohen-Sutherland method, for example, multiple intersections could be calculated along the path of a line segment before an intersection on the clipping rectangle is located or the line is completely rejected. These extra intersection calculations are eliminated in the NLN algorithm by carrying out more region testing before intersection positions are calculated. Compared to both the Cohen-Sutherland and the Liang-Barsky algorithms, the Nicholl-Lee-Nicholl algorithm performs fewer comparisons and divisions. The trade-off is that the NLN algorithm can be applied only to two-dimensional clipping, whereas both the Liang-Barsky and the Cohen-Sutherland methods are easily extended to three-dimensional scenes.

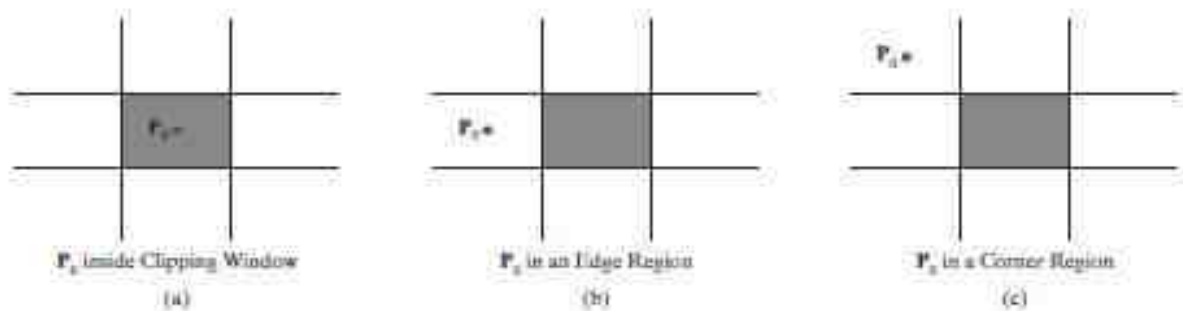
Initial testing to determine whether a line segment is completely inside the clipping window or outside the window limits can be accomplished with region code tests, as in the previous two algorithms. If a trivial acceptance or rejection of the line is not possible, the NLN algorithm proceeds to set up additional clipping regions.

For a line with endpoints  $\mathbf{P}_0$  and  $\mathbf{P}_{end}$ , we first determine the position of point  $\mathbf{P}_0$  for the nine possible regions relative to the clipping window. Only the three regions shown in Figure 14 need be considered. If  $\mathbf{P}_0$  lies in any one of the other six regions, we can move it to one of the three regions in Figure 14 using a symmetry transformation. For example, the region directly above the clip window can be transformed to the region left of the window using a reflection about the line  $y = -x$ , or we could use a 90° counterclockwise rotation.

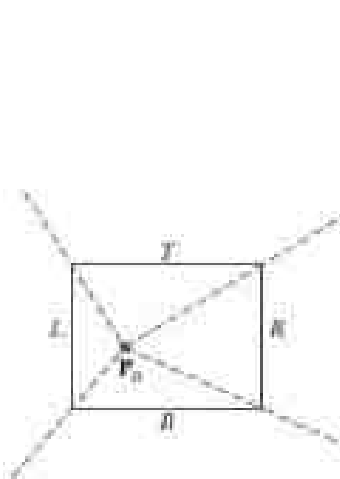
Assuming that  $\mathbf{P}_0$  and  $\mathbf{P}_{end}$  are not both inside the clipping window, we next determine the position of  $\mathbf{P}_{end}$  relative to  $\mathbf{P}_0$ . To do this, we create some new regions in the plane, depending on the location of  $\mathbf{P}_0$ . Boundaries of the new regions are semi-infinite line segments that start at the position of  $\mathbf{P}_0$  and pass through the clipping-window corners. If  $\mathbf{P}_0$  is inside the clipping window, we set up the four regions shown in Figure 15. Then, depending on which one of the four regions (L, T, R, or B) contains  $\mathbf{P}_{end}$ , we compute the line-intersection position with the corresponding window boundary.



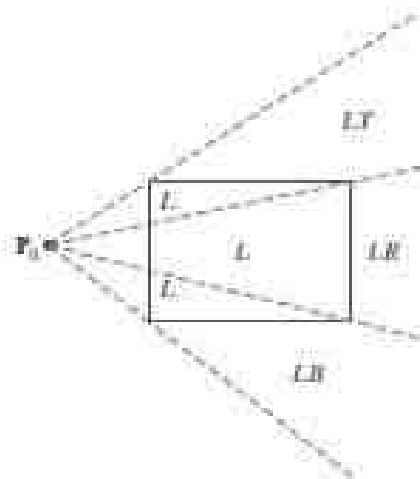
If  $P_0$  is in the region to the left of the window, we set up the four regions labeled L, LT, LR, and LB in Figure 16. These four regions again determine a unique  $P_{end}$



**FIGURE 14**  
Three possible positions for a line endpoint  $P_0$  in the NLN line-clipping algorithm.



**FIGURE 15**  
The four regions used in the NLN algorithm when  $P_0$  is inside the clipping window and  $P_{end}$  is outside.



**FIGURE 16**  
The four clipping regions used in the NLN algorithm when  $P_0$  is directly to the left of the clip window.

clipping-window edge for the line segment, relative to the position of  $P_{end}$ . For instance, if  $P_{end}$  is in any one of the three regions labeled L, we clip the line at the left window border and save the line segment from this intersection point to  $P_{end}$ . If  $P_{end}$  is in region LT, we save the line segment from the left window boundary to the top boundary. Similar processing is carried out for regions LR and LB. However, if  $P_{end}$  is not in any of the four regions L, LT, LR, or LB, the entire line is clipped.

For the third case, when  $P_0$  is to the left and above the clipping window, we use the regions in Figure 17. In this case, we have the two possibilities shown, depending on the position of  $P_0$  within the top-left corner of the clipping window. When  $P_0$  is closer to the left clipping boundary of the window, we use the regions in (a) of this figure. Otherwise, when  $P_0$  is closer to the top clipping boundary of the window, we use the regions in (b). If  $P_{end}$  is in one of the regions T, L, TR, TB, LR, or LB, this determines a unique clipping-window border for the intersection calculations. Otherwise, the entire line is rejected.

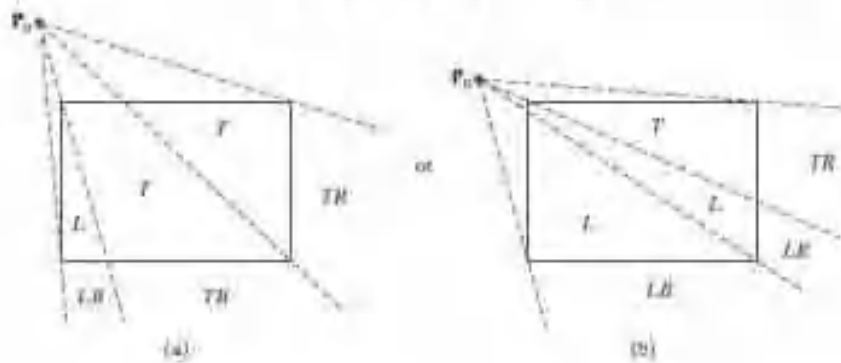
To determine the region in which  $P_{end}$  is located, we compare the slope of the line segment to the slopes of the boundaries of the NLN regions. For example, if  $P_0$  is left of the clipping window (Figure 16), then  $P_{end}$  is in region LT if

$P_0$  is left of the clipping window (Figure 16), then  $P_{\text{end}}$  is in region LT if

$$\text{slope}P_0P_{TR} < \text{slope}P_0P_{\text{end}} < \text{slope}P_0P_{TL} \quad (21)$$

or

$$\frac{y_r - y_0}{x_r - x_0} < \frac{y_{\text{end}} - y_0}{x_{\text{end}} - x_0} < \frac{y_t - y_0}{x_t - x_0} \quad (22)$$



**FIGURE 17**  
The two possible sets of clipping regions used in the SLM algorithm when  $P_0$  is above and to the left of the clipping window.

We clip the entire line if

$$(y_r - y_0)(x_{\text{end}} - x_0) < (x_t - x_0)(y_{\text{end}} - y_0) \quad (23)$$

The coordinate-difference calculations and product calculations used in the slope tests are saved and also used in the intersection calculations. From the parametric equations

$$x = x_0 + (x_{\text{end}} - x_0)u$$

$$y = y_0 + (y_{\text{end}} - y_0)u$$

we calculate an  $x$ -intersection position on the left window boundary as  $x = x_t$ , with  $u = (x_t - x_0)/(x_{\text{end}} - x_0)$ , so that the  $y$ -intersection position is

$$y = y_0 + \frac{y_{\text{end}} - y_0}{x_{\text{end}} - x_0}(x_t - x_0) \quad (24)$$

An intersection position on the top boundary has  $y = y_r$  and  $u = (y_r - y_0)/(y_{\text{end}} - y_0)$ , with

$$x = x_0 + \frac{x_{\text{end}} - x_0}{y_{\text{end}} - y_0}(y_r - y_0) \quad (25)$$

### Line Clipping Using Nonrectangular Polygon Clip Windows

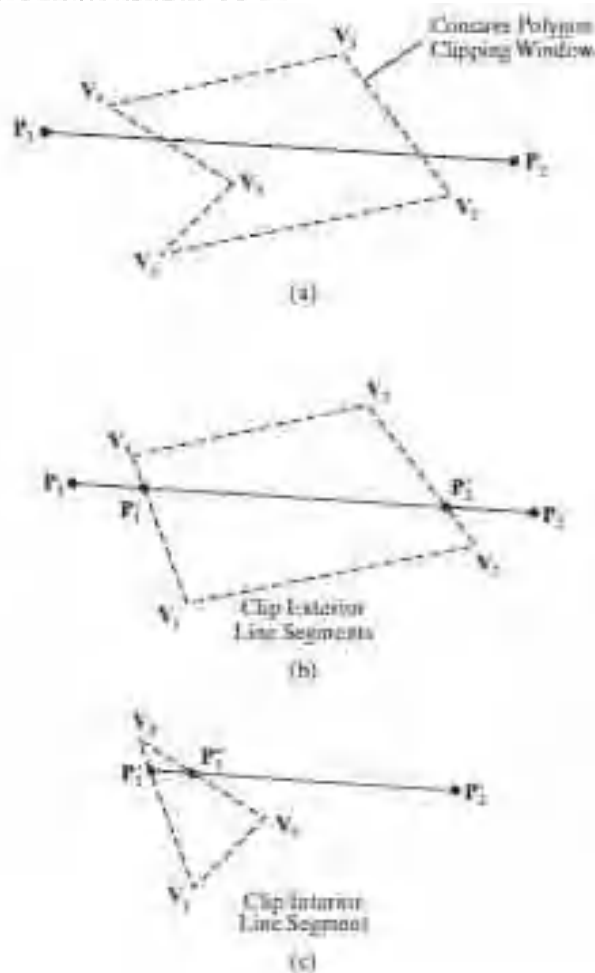
In some applications, it may be desirable to clip lines against arbitrarily shaped polygons. Methods based on parametric line equations, such as either the Cyrus-Beck algorithm or the Liang-Barsky algorithm, can be readily extended to clip lines against convex polygon windows. We do this by modifying the algorithm to include the parametric equations for the boundaries of the clipping region. Preliminary screening of line segments can be accomplished by processing lines against the coordinate extents of the clipping polygon.

For concave-polygon clipping regions, we could still apply these parametric clipping procedures if we first split the concave polygon into a set of convex polygons. Another approach is simply to add one or more edges to the concave clipping area so that it is modified to a convex-polygon shape. Then a series of clipping operations can be applied using the modified convex polygon components, as illustrated in Figure 18. The line segment  $P_1P_2$  in (a) of this figure is to be clipped by the concave window with vertices  $V_1$ ,  $V_2$ ,  $V_3$ ,  $V_4$ , and  $V_5$ . Two convex

clipping regions are obtained, in this case, by adding a line segment from  $V_4$  to  $V_1$ . Then the line is clipped in two passes: (1) Line  $\overline{P_1P_2}$  is clipped by the convex polygon with vertices  $V_1$ ,  $V_2$ ,  $V_3$ , and  $V_4$  to yield the clipped segment  $\overline{P_1'P_2'}$  [Figure 18(b)]. (2) The internal line segment  $\overline{P_1'P_2'}$  is clipped off using the convex polygon with vertices  $V_1$ ,  $V_3$ , and  $V_4$  [Figure 18(c)] to yield the final clipped line segment  $\overline{P_1''P_2''}$ .

### Line Clipping Using Nonlinear Clipping-Window Boundaries

Circles or other curved-boundary clipping regions are also possible, but they require more processing because the intersection calculations involve nonlinear equations. At the first step, lines could be clipped against the bounding rectangle (coordinate extents) of the curved clipping region. Lines that are outside the coordinate extents are eliminated. To identify lines that are inside a circle, for instance, we could calculate the distance of the line endpoints from the circle center. If the square of this distance for both endpoints of a line is less than or equal to the radius squared, we can save the entire line. The remaining lines are then processed through the intersection calculations, which must solve simultaneous circle-line equations.



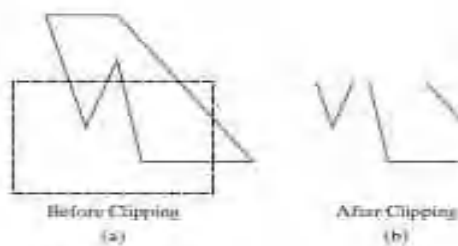
**FIGURE 18**  
A concave-polygon clipping window (a), with vertex list  $(V_1, V_2, V_3, V_4, V_4)$ , is modified to the convex polygon  $(V_1, V_2, V_3, V_4)$  in (b). The external segments of line  $\overline{P_1P_2}$  are then clipped off using this convex clipping window. The resulting line segment,  $\overline{P_1'P_2'}$ , is next processed against the triangle  $(V_1, V_3, V_4)$  (c) to clip off the internal line segment  $\overline{P_1'P_2'}$  to produce the final clipped line  $\overline{P_1''P_2''}$ .

## POLYGON CLIPPING

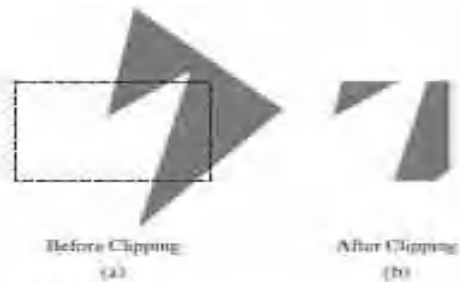
Graphics packages typically support only fill areas that are polygons, and often only convex polygons. To clip a polygon fill area, we cannot apply a line-clipping method to the individual polygon edges directly because this approach would not, in general, produce a closed polyline. Instead, a line clipper would often produce a disjoint set of lines with no complete information about how we might form a closed boundary around the clipped fill area. Figure 19 illustrates a possible output from a line-clipping procedure applied to the edges of a polygon fill area.

What we require is a procedure that will output one or more closed polylines for the boundaries of the clipped fill area, so that the polygons can be scan-converted to fill the interiors with the assigned color or pattern, as in Figure 20.

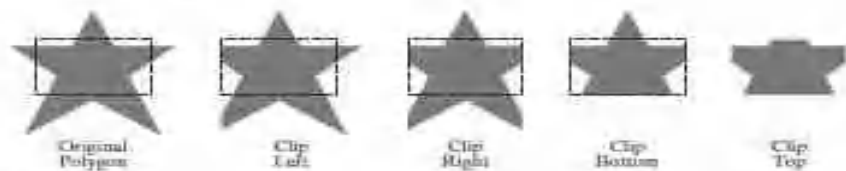
We can process a polygon fill area against the borders of a clipping window using the same general approach as in line clipping. A line segment is defined by its two endpoints, and these endpoints are processed through a line-clipping procedure by constructing a new set of clipped endpoints at each clipping-window boundary. Similarly, we need to maintain a fill area as an entity as it is processed through the clipping stages. Thus, we can clip a polygon fill area by determining the new shape for the polygon as each clipping-window edge is processed, as demonstrated in Figure 21. Of course, the interior fill for the polygon would not be applied until the final clipped border had been determined.



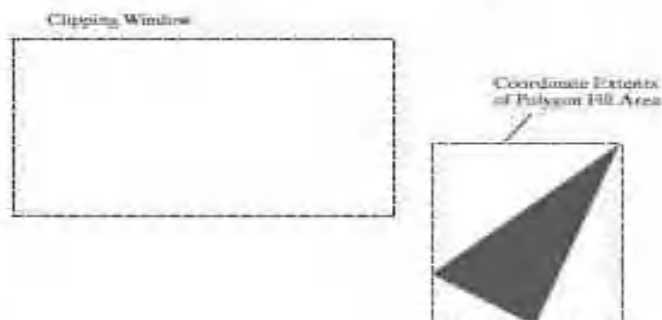
**FIGURE 19**  
A line-clipping algorithm applied to the line segments of the polygon boundary in (a) generates the unconnected set of lines in (b).



**FIGURE 20**  
Display of a correctly clipped polygon fill area.



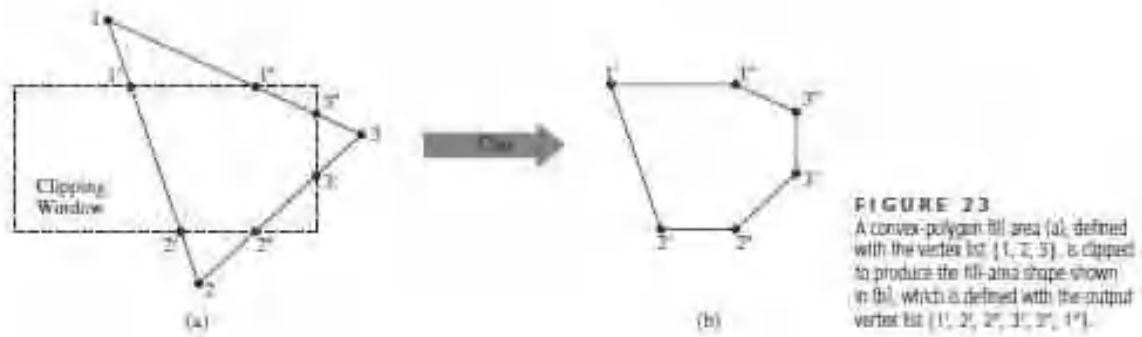
**FIGURE 21**  
Processing a polygon fill area against successive clipping-window boundaries.



**FIGURE 22**  
A polygon fill area with coordinate system outside the right clipping boundary.

Just as we first tested a line segment to determine whether it could be completely saved or completely clipped, we can do the same with a polygon fill area by checking its coordinate extents. If the minimum and maximum coordinate values for the fill area are inside all four clipping boundaries, the fill area is saved for further processing. If these coordinate extents are all outside any of the clipping-window borders, we eliminate the polygon from the scene description (Figure 22).

When we cannot identify a fill area as being completely inside or completely outside the clipping window, we then need to locate the polygon



**FIGURE 23**  
A convex-polygon fill area (a), defined with the vertex list  $\{1, 2, 3\}$ , is clipped to produce the fill-area shape shown in (b), which is defined with the output vertex list  $\{1, 2, 2', 3', 1'\}$ .

intersection positions with the clipping boundaries. One way to implement convex-polygon clipping is to create a new vertex list at each clipping boundary, and then pass this new vertex list to the next boundary clipper. The output of the final clipping stage is the vertex list for the clipped polygon (Figure 23). For concave-polygon clipping, we would need to modify this basic approach so that multiple vertex lists could be generated.

### **Sutherland-Hodgman Polygon Clipping**

An efficient method for clipping a convex-polygon fill area, developed by Sutherland and Hodgman, is to send the polygon vertices through each clipping stage so that a single clipped vertex can be immediately passed to the next stage. This eliminates the need for an output set of vertices at each clipping stage, and it allows the boundary-clipping routines to be implemented in parallel. The final output is a list of vertices that describe the edges of the clipped polygon fill area.

Because the Sutherland-Hodgman algorithm produces only one list of output vertices, it cannot correctly generate the two output polygons in Figure 20(b) that are the result of clipping the concave polygon shown in Figure 20(a). However, more processing steps can be added to the algorithm to allow it to produce multiple output vertex lists, so that general concave-polygon clipping could be accommodated.

And the basic Sutherland-Hodgman algorithm is able to process concave polygons when the clipped fill area can be described with a single vertex list.

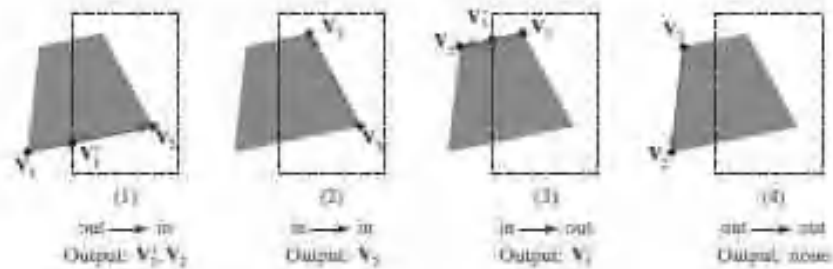
The general strategy in this algorithm is to send the pair of endpoints for each successive polygon line segment through the series of clippers (left, right, bottom, and top). As soon as a clipper completes the processing of one pair of vertices, the clipped coordinate values, if any, for that edge are sent to the next clipper. Then the first clipper processes the next pair of endpoints. In this way, the individual boundary clippers can be operating in parallel.



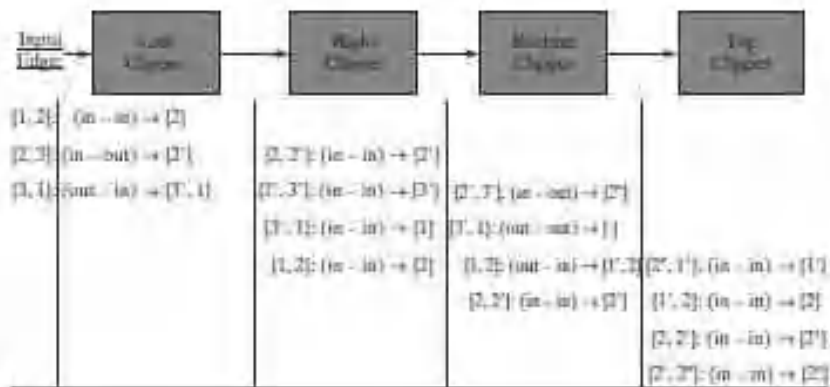
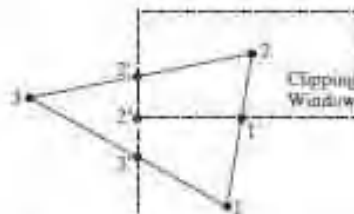
There are four possible cases that need to be considered when processing a polygon edge against one of the clipping boundaries. One possibility is that the first edge endpoint is outside the clipping boundary and the second endpoint is inside. Or, both endpoints could be inside this clipping boundary. Another possibility is that the first endpoint is inside the clipping boundary and the second endpoint is outside. And, finally, both endpoints could be outside the clipping boundary.

To facilitate the passing of vertices from one clipping stage to the next, the output from each clipper can be formulated as shown in Figure 24. As each successive pair of endpoints is passed to one of the four clippers, an output is generated for the next clipper according to the results of the following tests:

1. If the first input vertex is outside this clipping-window border and



**FIGURE 24**  
The four possible outputs generated by the left clipper, depending on the position of a pair of endpoints relative to the left boundary of the clipping window.



**FIGURE 25**  
Processing a set of polygon vertices, {1, 2, 3}, through the boundary clippers using the Sutherland-Hodgman algorithm. The final set of clipped vertices is {1', 2', 2'', 2'}.

thesecond vertex is inside, both the intersection point of the polygon edge with the window border and the second vertex are sent to the next clipper.

2. If both input vertices are inside this clipping-window border, only the second vertex is sent to the next clipper.

3. If the first vertex is inside this clipping-window border and the second vertex is outside, only the polygon edge-intersection position with theclipping-window border is sent to the next clipper.

4. If both input vertices are outside this clipping-window border, no vertices are sent to the next clipper.

The last clipper in this series generates a vertex list that describes the final clipped fill area.

Figure 25 provides an example of the Sutherland-Hodgman polygonclippingalgorithm for a fill area defined with the vertex set {1, 2, 3}. As soon

as a clipper receives a pair of endpoints, it determines the appropriate output using the tests illustrated in Figure 24. These outputs are passed in succession from the left clipper to the right, bottom, and top clippers. The output from the **return (iPt);**

```

}
void clipPoint (wcPt2D p, Boundary winEdge, wcPt2D wMin, wcPt2D
wMax,
wcPt2D * pOut, int * cnt, wcPt2D * first[], wcPt2D * s)
{
    wcPt2D iPt;
    /* If no previous point exists for this clipping boundary,
    * save this point.
    */
    if (!first[winEdge])
        first[winEdge] = &p;
    else
        /* Previous point exists. If p and previous point cross
        * this clipping boundary, find intersection. Clip against
        * next boundary, if any. If no more clip boundaries, add
        * intersection to output list.
        */
        if (cross (p, s[winEdge], winEdge, wMin, wMax)) {
            iPt = intersect (p, s[winEdge], winEdge, wMin, wMax);
            if (winEdge < Top)
                clipPoint (iPt, b+1, wMin, wMax, pOut, cnt, first, s);
            else {
                pOut[*cnt] = iPt; (*cnt)++;
            }
        }
        /* Save p as most recent point for this clip boundary. */
        s[winEdge] = p;
        /* For all, if point inside, proceed to next boundary, if any. */
        if (inside (p, winEdge, wMin, wMax))
            if (winEdge < Top)
                clipPoint (p, winEdge + 1, wMin, wMax, pOut, cnt, first, s);
            else {
                pOut[*cnt] = p; (*cnt)++;
            }
        }
    void closeClip (wcPt2D wMin, wcPt2D wMax, wcPt2D * pOut,
    GLint * cnt, wcPt2D * first [ ], wcPt2D * s)
    {

```

```

wcPt2D pt;
Boundary winEdge;
for (winEdge = Left; winEdge <= Top; winEdge++) {
if (cross (s[winEdge], *first[winEdge], winEdge, wMin, wMax)) {
pt = intersect (s[winEdge], *first[winEdge], winEdge, wMin, wMax);
if (winEdge < Top)
clipPoint (pt, winEdge + 1, wMin, wMax, pOut, cnt, first, s);
else {
pOut[*cnt] = pt; (*cnt)++;
}
}
}

```

top clipper is the set of vertices defining the clipped fill area. For this example, the output vertex list is {1\_, 2, 2\_, 2\_}.

A sequential implementation of the Sutherland-Hodgman polygon-clipping algorithm is demonstrated in the following set of procedures. An input set of vertices is converted to an output vertex list by clipping it against the four edges of the axis-aligned rectangular clipping region.

```

typedef enum { Left, Right, Bottom, Top } Boundary;
const GLint nClip = 4;
GLint inside (wcPt2D p, Boundary b, wcPt2D wMin, wcPt2D wMax)
{
switch (b) {
case Left: if (p.x < wMin.x) return (false); break;
case Right: if (p.x > wMax.x) return (false); break;
case Bottom: if (p.y < wMin.y) return (false); break;
case Top: if (p.y > wMax.y) return (false); break;
}
return (true);
}
GLint cross (wcPt2D p1, wcPt2D p2, Boundary winEdge, wcPt2D wMin,
wcPt2D wMax)
{
if (inside (p1, winEdge, wMin, wMax) == inside (p2, winEdge, wMin,
wMax))
return (false);
else return (true);
}
wcPt2D intersect (wcPt2D p1, wcPt2D p2, Boundary winEdge,
wcPt2D wMin, wcPt2D wMax)
{
wcPt2D iPt;
GLfloat m;
if (p1.x != p2.x) m = (p1.y - p2.y) / (p1.x - p2.x);
switch (winEdge) {
case Left:
iPt.x = wMin.x;
iPt.y = p2.y + (wMin.x - p2.x) * m;

```

```

break;
case Right:
iPt.x = wMax.x;
iPt.y = p2.y + (wMax.x - p2.x) * m;
break;
case Bottom:
iPt.y = wMin.y;
if (p1.x != p2.x) iPt.x = p2.x + (wMin.y - p2.y) / m;
else iPt.x = p2.x;
break;
case Top:
iPt.y = wMax.y;
if (p1.x != p2.x) iPt.x = p2.x + (wMax.y - p2.y) / m;
else iPt.x = p2.x;
break;
}
GLint polygonClipSuthHodg (wcPt2D wMin, wcPt2D wMax, GLint n,
wcPt2D * pIn, wcPt2D * pOut)
{
/* Parameter "first" holds pointer to first point processed for
* a boundary; "s" holds most recent point processed for boundary.
*/
wcPt2D * first[nClip] = { 0, 0, 0, 0 }, s[nClip];
GLint k, cnt = 0;
for (k = 0; k < n; k++)
clipPoint (pIn[k], Left, wMin, wMax, pOut, &cnt, first, s);
closeClip (wMin, wMax, pOut, &cnt, first, s);
return (cnt);
}

```

When a concave polygon is clipped with the Sutherland-Hodgman algorithm, extraneous lines may be displayed. An example of this effect is demonstrated in Figure 26. This occurs when the clipped polygon should have two or more separate sections. But since there is only one output vertex list, the last vertex in the list is always joined to the first vertex. There are several things we can do to display clipped concave polygons correctly.

### Weiler-Atherton Polygon Clipping

This algorithm provides a general polygon-clipping approach that can be used to clip a fill area that is either a convex polygon or a concave polygon. Moreover, the method was developed as a means for identifying

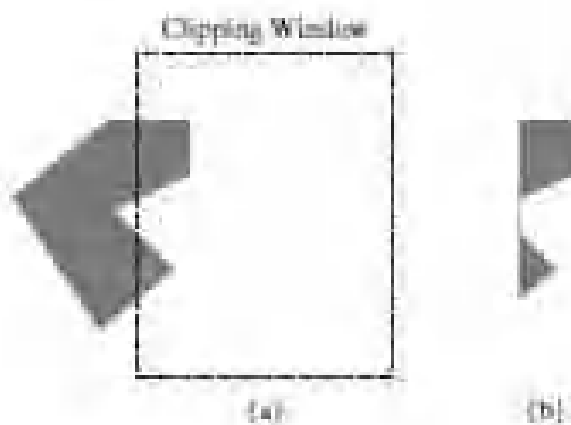


FIGURE 26  
Clipping the concave polygon in (a) using the Sutherland-Hodgman algorithm produces the two unconnected areas in (b).

visible surfaces in a threedimensionalscene. Therefore, we could also use this approach to clip any polygonfill area against a clipping window with any polygon shape.

For one, we could split a concave polygon into two or more convexpolygons and process each convex polygon separately using the Sutherland-Hodgman algorithm. Another possibility is to modify the Sutherland- Hodgmanmethod so that the final vertex list is checked for multiple intersection pointsalong any clipping-window boundary. If we find more than two vertex positionsalong any clipping boundary, we can separate the list of vertices into two ormore lists that correctly identify the separate sections of the clipped fill area. Thismay require extensive analysis to determine whether some points along the clippingboundary should be paired or whether they represent single vertex pointsthat have been clipped.Athird possibility is to use a more general polygon clipperthat has been designed to process concave polygons correctly.

Instead of simply clipping the fill-area edges as in the Sutherland-Hodgmanmethod, the Weiler-Atherton algorithm traces around the perimeter of the fillpolygon searching for the borders that enclose a clipped fill region. In this way,multiple fill regions, as in Figure 26(b), can be identified and displayed as separate,unconnected polygons. To find the edges for a clipped fill area, we follow a path (either counterclockwise or clockwise) around the fill area that detours alonga clipping-window boundary whenever a polygon edge crosses to the outside ofthat boundary. The direction of a detour at a clipping-window border is the sameas the processing direction for the polygon edges.

We can usually determine whether the processing direction is counterclockwiseor clockwise from the ordering of the vertex list that defines a polygon fillarea. In most cases, the vertex list is specified in a counterclockwise order as ameans for defining the front face of the polygon. Thus, the cross-product of twosuccessive edge vectors that form a convex angle determines the direction for thenormal vector, which is in the direction from the back face to the front face of thepolygon. If we do not know the vertex ordering, we could calculate the normal vector, or we can locate the interior of the fill area from any reference position.

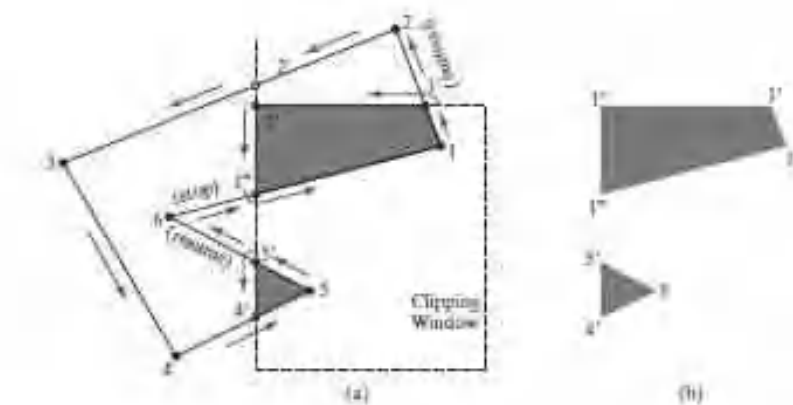
Then, if we sequentially process the edges so that the polygon interior is alwayson our left, we obtain a counterclockwise traversal. Otherwise, with the interiorto our right, we have a clockwise traversal. For a counterclockwise traversal of the polygon fill-area vertices, we applythe followingWeiler-Atherton procedures:

- 1.** Process the edges of the polygon fill area in a counterclockwise order until an inside-outside pair of vertices is encountered for one of the clippingboundaries; that is, the first vertex of the polygon edge is inside the clipregion and the second vertex is outside the clip region.
- 2.** Follow the window boundaries in a counterclockwise direction from theexit-intersection point to another intersection point with the polygon. Ifthis is a previously processed point, proceed to the next step. If this is anew intersection point, continue processing polygon edges in a counterclockwise order until a previously processed vertex is encountered.



3. Form the vertex list for this section of the clipped fill area.
4. Return to the exit-intersection point and continue processing the polygon edges in a counterclockwise order.

Figure 27 illustrates the Weiler-Atherton clipping of a concave polygon



**FIGURE 27**  
A concave polygon (a), defined with the vertex list  $\{1, 2, 3, 4, 5, 6\}$ , is clipped using the Weiler-Atherton algorithm to generate the two lists  $\{1, 1', 1'', 1'''\}$  and  $\{4, 3, 3'\}$ , which represent the separate polygon fill areas shown in (b).



**FIGURE 28**  
Clipping a polygon fill area against a concave-polygon clipping window using the Weiler-Atherton algorithm.

against a standard, rectangular clipping window for a counterclockwise traversal of the polygon edges. For a clockwise edge traversal, we would use a clockwise clipping-window traversal.

Starting from the vertex labeled 1 in Figure 27(a), the next polygon vertex to process in a counterclockwise order is labeled 2. Thus, this edge exits the clipping window at the top boundary. We calculate this intersection position (point  $1'$ ) and make a left turn there to process the window borders in a counterclockwise direction. Proceeding along the top border of the clipping window, we do not intersect a polygon edge before reaching the left window boundary. Therefore, we label this position as vertex  $1''$  and follow the left boundary to the intersection position  $1'''$ . We then follow this polygon edge in a counterclockwise direction, which returns us to vertex 1. This completes a circuit of the window boundaries and identifies the vertex list  $\{1, 1', 1'', 1'''\}$  as a clipped region of the original fill area. Processing of the polygon edges is then resumed at point  $1'$ . The edge defined by points 2 and 3 crosses to the outside of the left boundary, but points 2 and  $2'$  are above the top clipping-window border and points  $2''$  and 3 are to the left of the clipping region. Also, the edge with endpoints 3 and 4 is outside the left clipping boundary, but the next edge (from endpoint 4 to endpoint 5) reenters the clipping region and we pick up intersection point  $4'$ . The edge with endpoints 5 and 6 exits the window at intersection position  $5'$ , so we detour down the left clipping boundary to obtain the closed vertex list  $\{4', 5, 5'\}$ . We resume the polygon edge processing at position  $5'$ , which returns us to the previously processed point  $1''$ .

At this point, all polygon vertices and edges have been processed, so the fill area is completely clipped.

### **Polygon Clipping Using Nonrectangular Polygon Clip Windows**

The Liang-Barsky algorithm and other parametric line-clipping methods are particularly well suited for processing polygon fill areas against convex-polygon clipping windows. In this approach, we use a parametric representation for the edges of both the fill area and the clipping window, and both polygons are represented with a vertex list. We first compare the positions of the bounding rectangles for the fill area and the clipping polygon. If we cannot identify the fill area as completely outside the clipping polygon, we can use inside-outside tests to process the parametric edge equations. After completing all the region tests, we solve pairs of simultaneous parametric line equations to determine the window intersection positions.

We can also process any polygon fill area against any polygon-shaped clipping window (convex or concave), as in Figure 28, using the edge-traversal approach of the Weiler-Atherton algorithm. In this case, we need to maintain a vertex list for the clipping window as well as for the fill area, with both lists arranged in a counterclockwise (or clockwise) order. In addition, we need to apply inside-outside tests to determine whether a fill-area vertex is inside or outside a particular clipping-window boundary. As in the previous examples, we follow the window boundaries whenever a fill-area edge exits a clipping boundary. This clipping method can also be used when either the fill area or the clipping window contains holes that are defined with polygon borders. In addition, we can use this basic approach in constructive solid-geometry applications to identify the result of a union, intersection, or difference operation on two polygons. In fact, locating the clipped region of a fill area is equivalent to determining the intersection of two planar areas.

### **Polygon Clipping Using Nonlinear Clipping-Window Boundaries**

One method for processing a clipping window with curved boundaries is to approximate the boundaries with straight-line sections and use one of the algorithms for clipping against a general polygon-shaped clipping window. Alternatively, we could use the same general procedures that we discussed for line segments. First, we can compare the coordinate extents of the fill area to the coordinate extents of the clipping window. Depending on the shape of the clipping window, we may also be able to perform some other region tests based on symmetric considerations. For fill areas that cannot be identified as completely inside or completely outside the clipping window, we ultimately need to calculate the window intersection positions with the fill area.

## **CURVE CLIPPING**

Areas with curved boundaries can be clipped with methods similar to those discussed in the previous sections. If the objects are approximated with straight-line boundary sections, we use a polygon-clipping method. Otherwise, the clipping procedures involve nonlinear equations, and this requires more processing than for objects with linear boundaries.

We can first test the coordinate extents of an object against the clipping boundaries to determine whether it is possible to accept or reject the entire object trivially.

If not, we could check for object symmetries that we might be able to exploit in the initial accept/reject tests. For example, circles have symmetries between quadrants and octants, so we could check the coordinate extents of these individual circle regions. We cannot reject the complete circular fill area in Figure 29 just by checking its overall coordinate extents. But half of the circle is outside the right clipping border (or outside the top border), the upper-left quadrant is above the top clipping border, and the remaining two octants can be similarly eliminated.

An intersection calculation involves substituting a clipping-boundary position ( $x_{u\min}$ ,  $x_{u\max}$ ,  $y_{u\min}$ , or  $y_{u\max}$ ) in the nonlinear equation for the object boundary and solving for the other coordinate value. Once all intersection positions have been evaluated, the defining positions for the object can be stored for later use by the scan-line fill procedures. Figure 30 illustrates circle clipping against a rectangular window. For this example, the circle radius and the endpoints of the clipped arc can be used to fill the clipped region, by invoking the circle algorithm to locate positions along the arc between the intersection endpoints.

Similar procedures can be applied when clipping a curved object against a general polygon clipping region. On the first pass, we could compare the bounding rectangle of the object with the bounding rectangle of the clipping region. If this does not save or eliminate the entire object, we next solve the simultaneous line-curve equations to determine the clipping intersection points.

**TEXT CLIPPING EXTERIOR CLIPPING**

Several techniques can be used to provide text clipping in a graphics package.

In a particular application, the choice of clipping



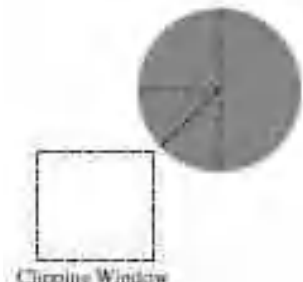
**FIGURE 31**  
Text clipping using the coordinate extents for an entire string.



**FIGURE 32**  
Text clipping using the bounding rectangle for individual characters in a string.



**FIGURE 33**  
Text clipping performed on the components of individual characters.



**FIGURE 29**  
A circle fill area, showing the quadrant and octant sections that are outside the clipping-window boundaries.



**FIGURE 30**  
Clipping a circle fill area.

method depends on how characters are generated and what requirements we have for displaying character strings.

The simplest method for processing character strings relative to the limits of a clipping window is to use the *all-or-none string-clipping* strategy shown in Figure 31. If all of the string is inside the clipping window, we display the entire string. Otherwise, the entire string is eliminated. This procedure is implemented by examining the coordinate extents of the text string. If the coordinate limits of this bounding rectangle are not completely within the clipping window, the string is rejected.

An alternative is to use the *all-or-none character-clipping* strategy. Here we eliminate only those characters that are not completely inside the clipping window (Figure 32). In this case, the coordinate extents of individual characters are compared to the window boundaries. Any character that is not completely within the clipping-window boundary is eliminated.

A third approach to text clipping is to clip the components of individual characters. This provides the most accurate display of clipped character strings, but it requires the most processing. We now treat characters in much the same way that we treated lines or polygons. If an individual character overlaps a clipping window, we clip off only the parts of the character that are outside the window (Figure 33). Outline character fonts defined with line segments are processed in this way using a polygon-clipping algorithm. Characters defined with bit maps are clipped by comparing the relative position of the individual pixels in the character grid patterns to the borders of the clipping region.

### **THREE DIMENSIONAL TRANSFORMATIONS**

Methods for geometric transformations in three dimensions are extended from two-dimensional methods by including considerations for the  $z$  coordinate. In most cases, this extension is relatively straightforward. However, in some cases particularly, rotation—the extension to three dimensions is less obvious.

When we discussed two-dimensional rotations in the  $xy$  plane, we needed to consider only rotations about axes that were perpendicular to the  $xy$  plane. In three-dimensional space, we can now select any spatial orientation for the rotation axis. Some graphics packages handle three-dimensional rotation as a composite of three rotations, one for each of the three Cartesian axes. Alternatively, we can set up general rotation equations, given the orientation of a rotation axis and the required rotation angle.

A three-dimensional position, expressed in homogeneous coordinates, is represented as a four-element column vector. Thus, each geometric transformation operator is now a  $4 \times 4$  matrix, which premultiplies a coordinate column vector. In addition, as in two dimensions, any sequence of transformations is represented as a single matrix, formed by concatenating the matrices for the individual transformations in the sequence. Each successive matrix in a transformation sequence is concatenated to the left of previous transformation matrices.

### **TRANSLATION**

A position  $P = (x, y, z)$  in three-dimensional space is translated to a location  $P' = (x', y', z')$  by adding translation distances  $t_x$ ,  $t_y$ , and  $t_z$  to the Cartesian coordinates of  $P$ :

$$x' = x + t_x \quad y' = y + t_y \quad z' = z + t_z \quad (1)$$

Figure 1 illustrates three-dimensional point translation.

We can express these three-dimensional translation operations in matrix form. But now the coordinate positions,  $P$  and  $P'$ , are represented in homogeneous coordinates with four-element column matrices, and the translation operator  $T$  is a  $4 \times 4$  matrix:

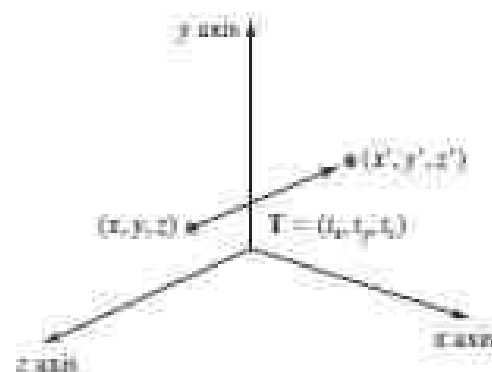
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (2)$$

or

$$P' = T \cdot P \quad (3)$$

An object is translated in three dimensions by transforming each of the defining coordinate positions for the object, then reconstructing the object at the new location. For an object represented as a set of polygon surfaces, we translate each vertex for each surface (Figure 2) and redisplay the polygon facets at the translated positions.

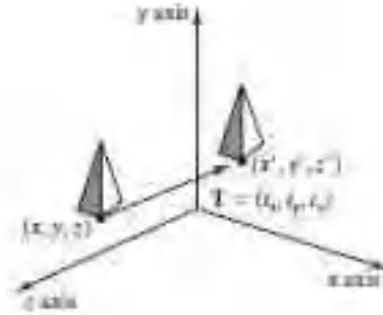
The following program segment illustrates construction of a translation matrix, given an input set of translation parameters.



**FIGURE 1**  
Moving a coordinate position with translation vector  $T = (t_x, t_y, t_z)$ .



An inverse of a three-dimensional translation matrix is obtained using the same procedures that we applied in a two-dimensional



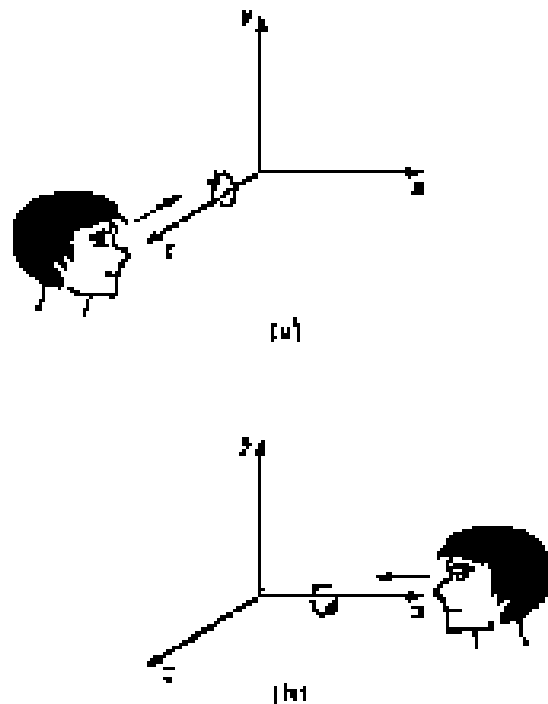
**FIGURE 2**  
Shifting the position of a three-dimensional object, using translation vector  $T$ .

translation. That is, we negate the translation distances  $tx$ ,  $ty$ , and  $tz$ . This produces a translation in the opposite direction, and the product of a translation matrix and its inverse is the identity matrix.

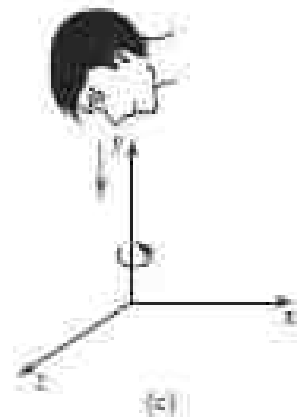
```
typedef GLfloat Matrix4x4 [4][4];
/* Construct the 4 x 4 identity matrix. */
void matrix4x4SetIdentity (Matrix4x4 matIdent4x4)
{
  GLint row, col;
  for (row = 0; row < 4; row++)
  for (col = 0; col < 4 ; col++)
    matIdent4x4 [row][col] = (row == col);
}
void translate3D (GLfloat tx, GLfloat ty, GLfloat tz)
{
  Matrix4x4 matTransl3D;
  /* Initialize translation matrix to identity. */
  matrix4x4SetIdentity (matTransl3D);
  matTransl3D [0][3] = tx;
  matTransl3D [1][3] = ty;
  matTransl3D [2][3] = tz;
}
```

### **ROTATION**

We can rotate an object about any axis in space, but the easiest rotation axes to handle are those that are parallel to the Cartesian-coordinate axes. Also, we can use combinations of coordinate-axis rotations (along with appropriate translations) to specify a rotation about any other line in space. Therefore, we first consider the operations involved in coordinate-axis rotations, then we discuss the calculations needed for other rotation axes. By convention, positive rotation angles produce counterclockwise rotations about a coordinate axis, assuming that we



are looking in the negative direction along that coordinate axis (Figure 3). This agrees with our earlier discussion of



**FIGURE 3**

Positive rotations about a coordinate axis are counterclockwise, when looking along the positive half of the axis toward the origin.

rotations in two dimensions, where positive rotations in the  $xy$  plane are counterclockwise about a pivot point (an axis that is parallel to the  $z$  axis).

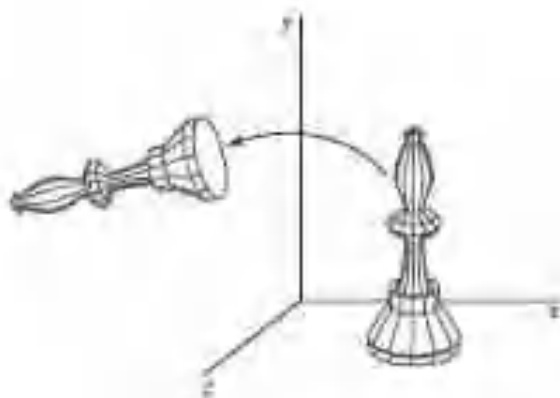
### Three-Dimensional Coordinate-Axis Rotations

The two-dimensional  $z$ -axis rotation equations are easily extended to three dimensions, as follows:

$$\begin{aligned}x' &= x \cos \theta - y \sin \theta \\y' &= x \sin \theta + y \cos \theta \\z' &= z\end{aligned}\quad (4)$$

Parameter  $\theta$  specifies the rotation angle about the  $z$  axis, and  $z$ -coordinate values are unchanged by this transformation. In homogeneous-coordinate form, the three-dimensional  $z$ -axis rotation equations are

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}\quad (5)$$



**FIGURE 4**

Rotation of an object about the  $z$  axis.

which we can write more compactly as

$$\mathbf{P}' = \mathbf{R}_z(\theta) \cdot \mathbf{P}\quad (6)$$

Figure 4 illustrates rotation of an object about the  $z$  axis.

Transformation equations for rotations about the other two coordinate axes can be obtained with a cyclic permutation of the coordinate parameters  $x$ ,  $y$ , and  $z$  in Equations 4:

$$x \rightarrow y \rightarrow z \rightarrow x \quad (7)$$

Thus, to obtain the  $x$ -axis and  $y$ -axis rotation transformations, we cyclically replace  $x$  with  $y$ ,  $y$  with  $z$ , and  $z$  with  $x$ , as illustrated in Figure 5.

Substituting permutations 7 into Equations 4, we get the equations for an  $x$ -axis rotation:

$$\begin{aligned} y' &= y \cos \theta - z \sin \theta \\ z' &= y \sin \theta + z \cos \theta \\ x' &= x \end{aligned} \quad (8)$$

Rotation of an object around the  $x$  axis is demonstrated in Figure 6.

A cyclic permutation of coordinates in Equations 8 gives us the transformation equations for a  $y$ -axis rotation:

$$\begin{aligned} z' &= z \cos \theta - x \sin \theta \\ x' &= z \sin \theta + x \cos \theta \\ y' &= y \end{aligned} \quad (9)$$

An example of  $y$ -axis rotation is shown in Figure 7.

An inverse three-dimensional rotation matrix is obtained in the same way as the inverse rotations in two dimensions. We just replace the angle  $\theta$  with  $-\theta$ .

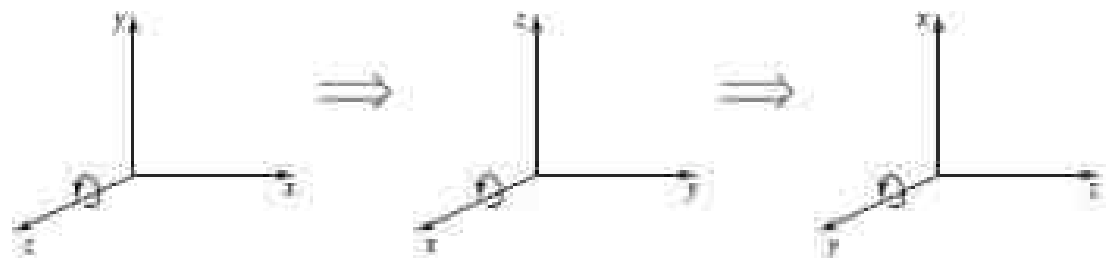


FIGURE 5

Cyclic permutation of the Cartesian-coordinate axes to produce the three sets of coordinate-axis rotation equations.

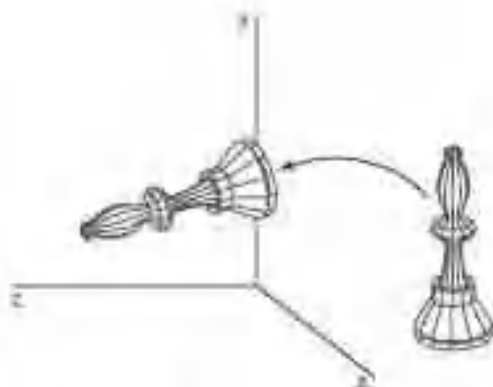


FIGURE 6

Rotation of an object about the  $x$  axis.

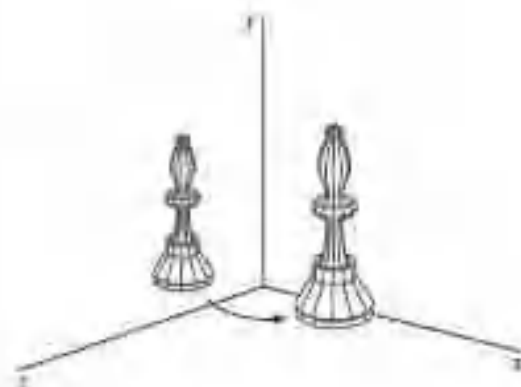


FIGURE 7

Rotation of an object about the  $y$  axis.

Negative values for rotation angles generate rotations in a clockwise direction, and the identity matrix is produced when we multiply any rotation matrix by its inverse. Because only the sine function is affected by the change in sign of the rotation angle, the inverse matrix can also be obtained by interchanging rows and columns. That is, we can calculate the inverse of any rotation matrix  $\mathbf{R}$  by forming its transpose ( $\mathbf{R}^{-1} = \mathbf{R}^T$ ).

### General Three-Dimensional Rotations

A rotation matrix for any axis that does not coincide with a coordinate axis can be set up as a composite transformation involving combinations of translations and the coordinate-axis rotations. We first move the designated rotation axis onto one of the coordinate axes. Then we apply the appropriate rotation matrix for that coordinate axis. The last step in the transformation sequence is to return the rotation axis to its original position.

In the special case where an object is to be rotated about an axis that is parallel to one of the coordinate axes, we attain the desired rotation with the following transformation sequence:

1. Translate the object so that the rotation axis coincides with the parallel coordinate axis.
2. Perform the specified rotation about that axis.
3. Translate the object so that the rotation axis is moved back to its original position.

The steps in this sequence are illustrated in Figure 5. A coordinate position  $\mathbf{P}$  is transformed with the sequence shown in this figure as

$$\mathbf{P}' = \mathbf{T}^{-1} \cdot \mathbf{R}_x(\theta) \cdot \mathbf{T} \cdot \mathbf{P} \quad (10)$$

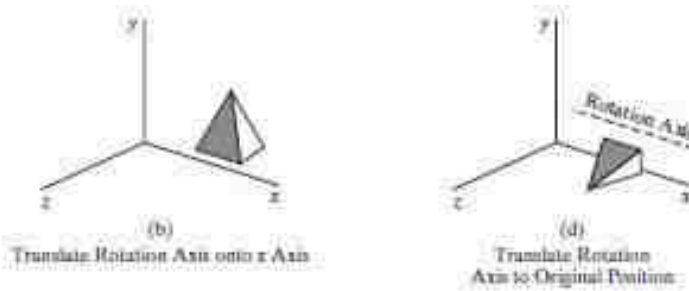
where the composite rotation matrix for the transformation is

$$\mathbf{R}(\theta) = \mathbf{T}^{-1} \cdot \mathbf{R}_x(\theta) \cdot \mathbf{T} \quad (11)$$

This composite matrix is of the same form as the two-dimensional transformation sequence for rotation about an axis that is parallel to the  $z$  axis (a pivot point that is not at the coordinate origin).

When an object is to be rotated about an axis that is not parallel to one of the coordinate axes, we must perform some additional transformations. In this





**FIGURE 8**  
Sequence of transformations for rotating an object about an axis that is parallel to the  $x$  axis.

case, we also need rotations to align the rotation axis with a selected coordinate axis and then to bring the rotation axis back to its original orientation. Given the specifications for the rotation axis and the rotation angle, we can accomplish the required rotation in five steps:

1. Translate the object so that the rotation axis passes through the coordinate origin.
2. Rotate the object so that the axis of rotation coincides with one of the coordinate axes.
3. Perform the specified rotation about the selected coordinate axis.
4. Apply inverse rotations to bring the rotation axis back to its original orientation.
5. Apply the inverse translation to bring the rotation axis back to its original spatial position.

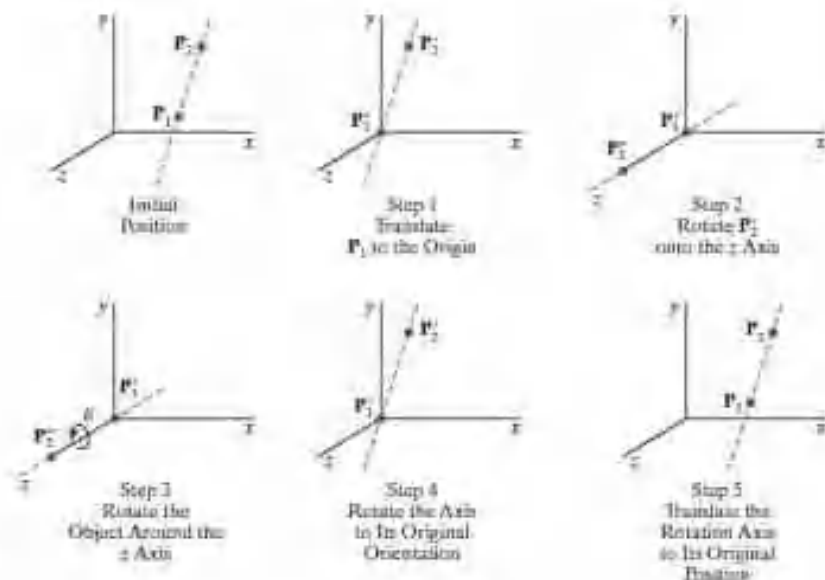
We can transform the rotation axis onto any one of the three coordinate axes. The  $z$  axis is often a convenient choice, and we next consider a transformation sequence using the  $z$ -axis rotation matrix (Figure 9).

A rotation axis can be defined with two coordinate positions, as in Figure 10, or with one coordinate point and direction angles (or direction cosines) between the rotation axis and two of the coordinate axes. We assume that the rotation axis is defined by two points, as illustrated, and that the direction of rotation is to be counterclockwise when looking along the axis from  $P_2$  to  $P_1$ . The components of the rotation-axis vector are then computed as

$$\begin{aligned} \mathbf{V} &= \mathbf{P}_2 - \mathbf{P}_1 \\ &= (x_2 - x_1, y_2 - y_1, z_2 - z_1) \end{aligned} \quad (12)$$

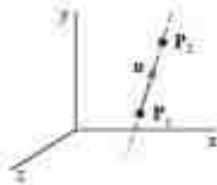
The unit rotation-axis vector  $\mathbf{u}$  is

$$\mathbf{u} = \frac{\mathbf{V}}{|\mathbf{V}|} = (a, b, c) \quad (13)$$



**FIGURE 9**  
Five transformation steps for obtaining a composite matrix for rotation about an arbitrary axis, with the rotation axis projected onto the  $z$  axis.





**FIGURE 10**  
An axis of rotation (dashed line) defined with points  $P_1$  and  $P_2$ . The direction for the unit axis vector  $\mathbf{u}$  is determined by the specified rotation direction.



**FIGURE 11**  
Translation of the rotation axis to the coordinate origin.

where the components  $a$ ,  $b$ , and  $c$  are the direction cosines for the rotation axis:

$$a = \frac{x_2 - x_1}{|\mathbf{V}|}, \quad b = \frac{y_2 - y_1}{|\mathbf{V}|}, \quad c = \frac{z_2 - z_1}{|\mathbf{V}|} \quad (14)$$

If the rotation is to be in the opposite direction (clockwise when viewing from  $P_2$  to  $P_1$ ), then we would reverse axis vector  $\mathbf{V}$  and unit vector  $\mathbf{u}$  so that they point in the direction from  $P_2$  to  $P_1$ .

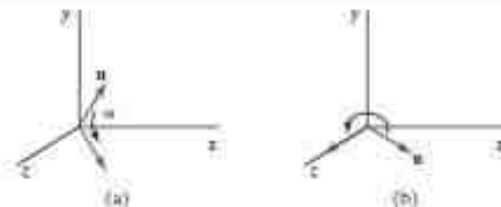
The first step in the rotation sequence is to set up the translation matrix that repositions the rotation axis so that it passes through the coordinate origin. Because we want a counterclockwise rotation when viewing along the axis from  $P_2$  to  $P_1$  (Figure 10), we move the point  $P_1$  to the origin. (If the rotation had been specified in the opposite direction, we would move  $P_2$  to the origin.) This translation matrix is

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (15)$$

which repositions the rotation axis and the object as shown in Figure 11.

Next, we formulate the transformations that will put the rotation axis onto the  $z$  axis. We can use the coordinate-axis rotations to accomplish this alignment in two steps, and there are a number of ways to perform these two steps. For this example, we first rotate about the  $x$  axis, then rotate about the  $y$  axis. The  $x$ -axis rotation gets vector  $\mathbf{u}$  into the  $xz$  plane, and the  $y$ -axis rotation swings  $\mathbf{u}$  around to the  $z$  axis. These two rotations are illustrated in Figure 12 for one possible orientation of vector  $\mathbf{u}$ .

Because rotation calculations involve sine and cosine functions, we can use standard vector operations to obtain elements of the two rotation matrices. A vector dot product can be used to determine the cosine term, and a vector cross product can be used to calculate the sine term.



**FIGURE 12**  
Unit vector  $\mathbf{u}$  is rotated about the  $x$  axis to bring it into the  $xz$  plane (a), then it is rotated around the  $y$  axis to align it with the  $z$  axis (b).

We establish the transformation matrix for rotation around the  $x$  axis by determining the values for the sine and cosine of the rotation angle necessary to get  $\mathbf{u}$  into the  $xz$  plane. This rotation angle is the angle between the projection of  $\mathbf{u}$  in the  $yz$  plane and the positive  $z$  axis (Figure 13). If we represent the projection of  $\mathbf{u}$  in the  $yz$  plane as the vector  $\mathbf{u}' = (0, b, c)$ , then the cosine of the rotation angle  $\alpha$  can be determined from the dot product of  $\mathbf{u}'$  and the unit vector  $\mathbf{u}_z$  along the  $z$  axis:

$$\cos \alpha = \frac{\mathbf{u}' \cdot \mathbf{u}_z}{|\mathbf{u}'| |\mathbf{u}_z|} = \frac{c}{d} \quad (16)$$

where  $d$  is the magnitude of  $\mathbf{u}'$ :

$$d = \sqrt{b^2 + c^2} \quad (17)$$

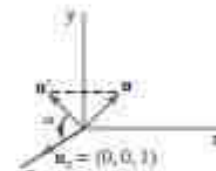
Similarly, we can determine the sine of  $\alpha$  from the cross-product of  $\mathbf{u}'$  and  $\mathbf{u}_z$ . The coordinate-independent form of this cross-product is

$$\mathbf{u}' \times \mathbf{u}_z = \mathbf{u}_z |\mathbf{u}'| \sin \alpha \quad (18)$$

and the Cartesian form for the cross-product gives us

$$\mathbf{u}' \times \mathbf{u}_z = \mathbf{u}_z \cdot b \quad (19)$$

Equating the right sides of Equations 18 and 19, and noting that  $|\mathbf{u}| = 1$  and  $|\mathbf{u}'| = d$ , we have



**FIGURE 13**  
Rotation of  $\mathbf{u}$  around the  $x$  axis into the  $xz$  plane is accomplished by rotating  $\mathbf{u}'$  (the projection of  $\mathbf{u}$  in the  $yz$  plane) through angle  $\alpha$  onto the  $z$  axis.

$$d \sin \alpha = b$$

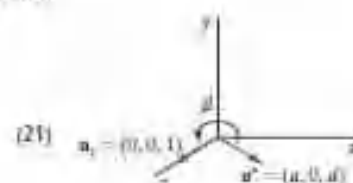
or

$$\sin \alpha = \frac{b}{d} \quad (20)$$

Now that we have determined the values for  $\cos \alpha$  and  $\sin \alpha$  in terms of the components of vector  $\mathbf{u}$ , we can set up the matrix elements for rotation of this vector about the  $x$  axis and into the  $xz$  plane:

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{c}{d} & \frac{b}{d} & 0 \\ 0 & \frac{b}{d} & \frac{c}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (21)$$

The next step in the formulation of the transformation sequence is to determine the matrix that will swing the unit vector in the  $xz$  plane counterclockwise around the  $y$  axis onto the positive  $z$  axis. Figure 14 shows the orientation of



**FIGURE 14**  
Rotation of unit vector  $\mathbf{u}'$  (vector  $\mathbf{u}$  after rotation into the  $xz$  plane) about the  $y$  axis. Positive rotation angle  $\beta$  aligns  $\mathbf{u}'$  with vector  $\mathbf{u}$ .

the unit vector in the  $xz$  plane, resulting from the rotation about the  $x$  axis. This vector, labeled  $\mathbf{u}'$ , has the value  $a$  for its  $x$  component, because rotation about the  $x$  axis leaves the  $x$  component unchanged. Its  $z$  component is  $d$  (the magnitude of  $\mathbf{u}'$ ), because vector  $\mathbf{u}'$  has been rotated onto the  $z$  axis. Also, the  $y$  component of  $\mathbf{u}'$  is 0 because it now lies in the  $xz$  plane. Again, we can determine the cosine of rotation angle  $\beta$  from the dot product of unit vectors  $\mathbf{u}'$  and  $\mathbf{u}_z$ . Thus,

$$\cos \beta = \frac{\mathbf{u}' \cdot \mathbf{u}_z}{|\mathbf{u}'| |\mathbf{u}_z|} = d \quad (22)$$

because  $|\mathbf{u}_z| = |\mathbf{u}'| = 1$ . Comparing the coordinate-independent form of the cross-product

$$\mathbf{u}' \times \mathbf{u}_z = \mathbf{u}_y |\mathbf{u}'| |\mathbf{u}_z| \sin \beta \quad (23)$$

with the Cartesian form

$$\mathbf{u}' \times \mathbf{u}_z = \mathbf{u}_y \cdot (-a) \quad (24)$$

we find that

$$\sin \beta = -a \quad (25)$$

Therefore, the transformation matrix for rotation of  $\mathbf{u}'$  about the  $y$  axis is

$$\mathbf{R}_y(\beta) = \begin{bmatrix} d & 0 & -a & 0 \\ 0 & 1 & 0 & 0 \\ a & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (26)$$

With transformation matrices 15, 21, and 26, we have aligned the rotation axis with the positive  $z$  axis. The specified rotation angle  $\theta$  can now be applied as a rotation about the  $z$  axis as follows:

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (27)$$

To complete the required rotation about the given axis, we need to transform the rotation axis back to its original position. This is done by applying the inverse of transformations 15, 21, and 26. The transformation matrix for rotation about an arbitrary axis can then be expressed as the composition of these seven individual transformations:

$$\mathbf{R}(\theta) = \mathbf{T}^{-1} \cdot \mathbf{R}_z^{-1}(\alpha) \cdot \mathbf{R}_y^{-1}(\beta) \cdot \mathbf{R}_x(\theta) \cdot \mathbf{R}_y(\beta) \cdot \mathbf{R}_z(\alpha) \cdot \mathbf{T} \quad (28)$$

A somewhat quicker, but perhaps less intuitive, method for obtaining the composite rotation matrix  $\mathbf{R}_y(\beta) \cdot \mathbf{R}_z(\alpha)$  is to use the fact that the composite matrix for any sequence of three-dimensional rotations is of the form

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (29)$$

The upper-left  $3 \times 3$  submatrix of this matrix is orthogonal. This means that the rows (or the columns) of this submatrix form a set of orthogonal unit vectors that

are rotated by matrix  $\mathbf{R}$  onto the  $x$ ,  $y$ , and  $z$  axes, respectively:

$$\mathbf{R} \cdot \begin{bmatrix} r_{11} \\ r_{12} \\ r_{13} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{R} \cdot \begin{bmatrix} r_{21} \\ r_{22} \\ r_{23} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{R} \cdot \begin{bmatrix} r_{31} \\ r_{32} \\ r_{33} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad (30)$$

Therefore, we can set up a local coordinate system with one of its axes aligned on the rotation axis. Then the unit vectors for the three coordinate axes are used to construct the columns of the rotation matrix. Assuming that the rotation axis is not parallel to any coordinate axis, we could form the following set of local unit vectors (Figure 15).

$$\begin{aligned} \mathbf{u}'_z &= \mathbf{u} \\ \mathbf{u}'_y &= \frac{\mathbf{u} \times \mathbf{u}_z}{|\mathbf{u} \times \mathbf{u}_z|} \\ \mathbf{u}'_x &= \mathbf{u}'_y \times \mathbf{u}'_z \end{aligned} \quad (31)$$

If we express the elements of the unit local vectors for the rotation axis as

$$\begin{aligned} \mathbf{u}'_z &= (u'_{z1}, u'_{z2}, u'_{z3}) \\ \mathbf{u}'_y &= (u'_{y1}, u'_{y2}, u'_{y3}) \\ \mathbf{u}'_x &= (u'_{x1}, u'_{x2}, u'_{x3}) \end{aligned} \quad (32)$$

then the required composite matrix, which is equal to the product  $\mathbf{R}_y(\beta) \cdot \mathbf{R}_z(\alpha)$ , is

$$\mathbf{R} = \begin{bmatrix} u'_{z1} & u'_{z2} & u'_{z3} & 0 \\ u'_{y1} & u'_{y2} & u'_{y3} & 0 \\ u'_{x1} & u'_{x2} & u'_{x3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (33)$$

This matrix transforms the unit vectors  $\mathbf{u}'_z$ ,  $\mathbf{u}'_y$ , and  $\mathbf{u}'_x$  onto the  $x$ ,  $y$ , and  $z$  axes, respectively. This aligns the rotation axis with the  $z$  axis, because  $\mathbf{u}'_z = \mathbf{u}$ .

### Quaternion Methods for Three-Dimensional Rotations

A more efficient method for generating a rotation about an arbitrarily selected axis is to use a quaternion representation for the rotation transformation. Quaternions, which are extensions of two-dimensional complex numbers, are useful in a number of computer-graphics procedures, including the generation of fractal objects.

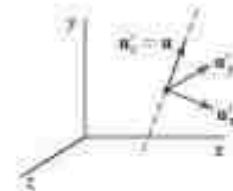


FIGURE 15  
Local coordinate system for a rotation axis defined by unit vector  $\mathbf{u}$ .

They require less storage space than  $4 \times 4$  matrices, and it is simpler to write quaternion procedures for transformation sequences. This is particularly important in animations, which often require complicated motion sequences and motion interpolations between two given positions of an object.

One way to characterize a quaternion is as an ordered pair, consisting of a *scalar part* and a *vector part*:

$$q = (s, \mathbf{v})$$

We can also think of a quaternion as a higher-order complex number with one real part (the scalar part) and three complex parts (the elements of vector  $\mathbf{v}$ ). A rotation about any axis passing through the coordinate origin is accomplished by first setting up a unit quaternion with the scalar and vector parts as follows:

$$s = \cos \frac{\theta}{2}, \quad \mathbf{v} = \mathbf{u} \sin \frac{\theta}{2} \quad (34)$$

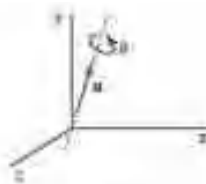


FIGURE 16  
Unit quaternion parameters  $s$  and  $\mathbf{u}$   
for rotation about a specified axis.

where  $\mathbf{u}$  is a unit vector along the selected rotation axis and  $\theta$  is the specified rotation angle about this axis (Figure 16). Any point position  $P$  that is to be rotated by this quaternion can be represented in quaternion notation as

$$P = (0, \mathbf{p})$$

with the coordinates of the point as the vector part  $\mathbf{p} = (x, y, z)$ . The rotation of the point is then carried out with the quaternion operation

$$P' = qPq^{-1} \quad (35)$$

where  $q^{-1} = (s, -\mathbf{v})$  is the inverse of the unit quaternion  $q$  with the scalar and vector parts given in Equations 34. This transformation produces the following

$$P' = (0, \mathbf{p}') \quad (36)$$

The second term in this ordered pair is the rotated point position  $\mathbf{p}'$ , which is evaluated with vector dot and cross-products as

$$\mathbf{p}' = s^2\mathbf{p} + \mathbf{v}(\mathbf{p} \cdot \mathbf{v}) + 2s(\mathbf{v} \times \mathbf{p}) + \mathbf{v} \times (\mathbf{v} \times \mathbf{p}) \quad (37)$$

Values for parameters  $s$  and  $\mathbf{v}$  are obtained from the expressions in 34. Many computer graphics systems use efficient hardware implementations of these vector calculations to perform rapid three-dimensional object rotations.

Transformation 35 is equivalent to rotation about an axis that passes through the coordinate origin. This is the same as the sequence of rotation transformations in Equation 28 that aligns the rotation axis with the  $z$  axis, rotates about  $z$ , and then returns the rotation axis to its original orientation at the coordinate origin.

We can evaluate the terms in Equation 37 using the definition for quaternion multiplication. Also, designating the components of the vector part of  $q$  as  $\mathbf{v} = (a, b, c)$ , we obtain the elements for the composite rotation matrix  $R_z^{-1}(\alpha)R_y^{-1}(\beta)R_x(\theta)R_y(\beta)R_z(\alpha)$  in a  $3 \times 3$  form as

$$\mathbf{M}_R(\theta) = \begin{bmatrix} 1 - 2b^2 - 2c^2 & 2ab - 2ac & 2ac + 2ab \\ 2ab + 2ac & 1 - 2a^2 - 2c^2 & 2bc - 2ca \\ 2ac - 2ab & 2bc + 2ca & 1 - 2a^2 - 2b^2 \end{bmatrix} \quad (38)$$

The calculations involved in this matrix can be greatly reduced by substituting explicit values for parameters  $a$ ,  $b$ ,  $c$ , and  $s$ , and then using the following trigonometric identities to simplify the terms:

$$\cos^2 \frac{\theta}{2} - \sin^2 \frac{\theta}{2} = 1 - 2 \sin^2 \frac{\theta}{2} = \cos \theta, \quad 2 \cos \frac{\theta}{2} \sin \frac{\theta}{2} = \sin \theta$$

Thus, we can rewrite Matrix 38 as

$$\mathbf{M}_R(\theta) = \begin{bmatrix} u_x^2(1 - \cos \theta) + \cos \theta & u_x u_y(1 - \cos \theta) - u_z \sin \theta & u_x u_z(1 - \cos \theta) + u_y \sin \theta \\ u_y u_x(1 - \cos \theta) + u_z \sin \theta & u_y^2(1 - \cos \theta) + \cos \theta & u_y u_z(1 - \cos \theta) - u_x \sin \theta \\ u_z u_x(1 - \cos \theta) - u_y \sin \theta & u_z u_y(1 - \cos \theta) + u_x \sin \theta & u_z^2(1 - \cos \theta) + \cos \theta \end{bmatrix} \quad (39)$$

where  $u_x$ ,  $u_y$ , and  $u_z$  are the components of the unit axis vector  $\mathbf{u}$ .

To complete the transformation sequence for rotating about an arbitrarily placed rotation axis, we need to include the translations that move the rotation axis to the coordinate axis and return it to its original position. Thus, the complete quaternion rotation expression, corresponding to Equation 28, is

$$\mathbf{R}(\theta) = \mathbf{T}^{-1} \cdot \mathbf{M}_R \cdot \mathbf{T} \quad (40)$$

For example, we can perform a rotation about the  $z$  axis by setting rotation axis vector  $\mathbf{u}$  to the unit  $z$ -axis vector  $(0, 0, 1)$ . Substituting the components of this vector into Matrix 39, we get the  $3 \times 3$  version of the  $z$ -axis rotation matrix  $\mathbf{R}_z(\theta)$  in Equation . Similarly, substituting the unit-quaternion rotation values into Equation 35 produces the rotated coordinate values in Equations 4.

In the following code, we give examples of procedures that could be used to construct a three-dimensional rotation matrix. The quaternion representation in Equation 40 is used to set up the matrix elements for a general three-dimensional rotation.

```
class wcPt3D
{public:
  GLfloat x, y, z;
};
typedef float Matrix4x4 [4][4];
Matrix4x4 matRot;
/* Construct the 4 x 4 identity matrix. */
void matrix4x4SetIdentity (Matrix4x4 matIdent4x4)
{
  GLint row, col;
  for (row = 0; row < 4; row++)
  for (col = 0; col < 4 ; col++)
    matIdent4x4 [row][col] = (row == col);
}
/* Premultiply matrix m1 by matrix m2, store result in m2. */
void matrix4x4PreMultiply (Matrix4x4 m1, Matrix4x4 m2)
{
  GLint row, col;
  Matrix4x4 matTemp;
  for (row = 0; row < 4; row++)
  for (col = 0; col < 4 ; col++)
```



```

matTemp [row][col] = m1 [row][0] * m2 [0][col] + m1 [row][1] *
m2 [1][col] + m1 [row][2] * m2 [2][col] +
m1 [row][3] * m2 [3][col];
for (row = 0; row < 4; row++)
for (col = 0; col < 4; col++)
m2 [row][col] = matTemp [row][col];
}
void translate3D (GLfloat tx, GLfloat ty, GLfloat tz)
{
Matrix4x4 matTransl3D;
/* Initialize translation matrix to identity. */
matrix4x4SetIdentity (matTransl3D);
matTransl3D [0][3] = tx;
matTransl3D [1][3] = ty;
matTransl3D [2][3] = tz;
/* Concatenate translation matrix with matRot. */
matrix4x4PreMultiply (matTransl3D, matRot);
}
void rotate3D (wcPt3D p1, wcPt3D p2, GLfloat radianAngle)
{
Matrix4x4 matQuaternionRot;
GLfloat axisVectLength = sqrt ((p2.x - p1.x) * (p2.x - p1.x) +
(p2.y - p1.y) * (p2.y - p1.y) +
(p2.z - p1.z) * (p2.z - p1.z));
GLfloat cosA = cos (radianAngle);
GLfloat oneC = 1 - cosA;
GLfloat sinA = sin (radianAngle);
GLfloat ux = (p2.x - p1.x) / axisVectLength;
GLfloat uy = (p2.y - p1.y) / axisVectLength;
GLfloat uz = (p2.z - p1.z) / axisVectLength;
/* Set up translation matrix for moving p1 to origin. */
translate3D (-p1.x, -p1.y, -p1.z);
/* Initialize matQuaternionRot to identity matrix. */
matrix4x4SetIdentity (matQuaternionRot);
matQuaternionRot [0][0] = ux*ux*oneC + cosA;
matQuaternionRot [0][1] = ux*uy*oneC - uz*sinA;
matQuaternionRot [0][2] = ux*uz*oneC + uy*sinA;
matQuaternionRot [1][0] = uy*ux*oneC + uz*sinA;
matQuaternionRot [1][1] = uy*uy*oneC + cosA;
matQuaternionRot [1][2] = uy*uz*oneC - ux*sinA;
matQuaternionRot [2][0] = uz*ux*oneC - uy*sinA;
matQuaternionRot [2][1] = uz*uy*oneC + ux*sinA;
matQuaternionRot [2][2] = uz*uz*oneC + cosA;
/* Combine matQuaternionRot with translation matrix. */
matrix4x4PreMultiply (matQuaternionRot, matRot);
/* Set up inverse matTransl3D and concatenate with
* product of previous two matrices.
*/
translate3D (p1.x, p1.y, p1.z);

```

```

}void displayFcn (void)
/* Input rotation parameters. */
/* Initialize matRot to identity matrix: */
matrix4x4SetIdentity (matRot);
/* Pass rotation parameters to procedure rotate3D. */
/* Display rotated object. */
}
SCALING

```

The matrix expression for the three-dimensional scaling transformation of a position  $\mathbf{P} = (x, y, z)$  relative to the coordinate origin is a simple extension of two-dimensional scaling. We just include the parameter for  $z$ -coordinate scaling in the transformation matrix:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (41)$$

The three-dimensional scaling transformation for a point position can be represented as

$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P} \quad (42)$$

where scaling parameters  $s_x$ ,  $s_y$ , and  $s_z$  are assigned any positive values. Explicit expressions for the scaling transformation relative to the origin are

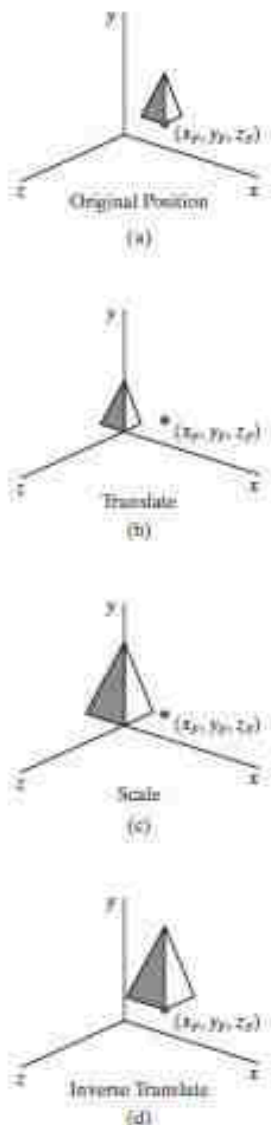
$$x' = x \cdot s_x, \quad y' = y \cdot s_y, \quad z' = z \cdot s_z \quad (43)$$

Scaling an object with transformation 41 changes the position of the object relative to the coordinate origin. A parameter value greater than 1 moves a point farther from the origin in the corresponding coordinate direction. Similarly, a parameter value less than 1 moves a point closer to the origin in that coordinate direction. Also, if the scaling parameters are not all equal, relative dimensions of a transformed object are changed. We preserve the original shape of an object with a *uniform scaling*:  $s_x = s_y = s_z$ . The result of scaling an object uniformly, with each scaling parameter set to 2, is illustrated in Figure 17.

Because some graphics packages provide only a routine that scales relative to the coordinate origin, we can always construct a scaling transformation with respect to any selected *fixed position*  $(x_f, y_f, z_f)$  using the following transformation sequence:

1. Translate the fixed point to the origin.
2. Apply the scaling transformation relative to the coordinate origin using Equation 41.
3. Translate the fixed point back to its original position.

This sequence of transformations is demonstrated in Figure 18. The matrix representation for an arbitrary fixed-point scaling can then be expressed as the



**FIGURE 18**  
A sequence of transformations for scaling an object relative to a selected fixed point, using Equation 44.

concatenation of these translate-scale-translate transformations:

$$T(x_f, y_f, z_f) \cdot S(s_x, s_y, s_z) \cdot T(-x_f, -y_f, -z_f) = \begin{bmatrix} s_x & 0 & 0 & (1-s_x)x_f \\ 0 & s_y & 0 & (1-s_y)y_f \\ 0 & 0 & s_z & (1-s_z)z_f \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (44)$$

We can set up programming procedures for constructing a three-dimensional scaling matrix using either a translate-scale-translate sequence or a direct incorporation of the fixed-point coordinates. In the following code example, we demonstrate a direct construction of a three-dimensional scaling matrix relative to a selected fixed point using the calculations in Equation 44:

```
class wPt3D
{
private:
    GLfloat x, y, z;

public:
    /* Default Constructor:
     * Initialize position as (0.0, 0.0, 0.0).
     */
    wPt3D ( ) {
        x = y = z = 0.0;
    }

    setCoords (GLfloat xCoord, GLfloat yCoord, GLfloat zCoord) {
        x = xCoord;
        y = yCoord;
        z = zCoord;
    }

    GLfloat getX ( ) const {
        return x;
    }

    GLfloat getY ( ) const {
        return y;
    }

    GLfloat getz ( ) const {
        return z;
    }
};

typedef float Matrix3x4 [4][3];

void scale3D (GLfloat sx, GLfloat sy, GLfloat sz, wPt3D fixedPt)
{
    Matrix3x4 matScale3D;

    /* Initialize scaling matrix to identity. */
    matrix3x4Identity (matScale3D);
}
```

```
matScale3D [0][0] = sx;
matScale3D [0][3] = (1 - sx) * fixedPt.getX ( );
matScale3D [1][1] = sy;
matScale3D [1][3] = (1 - sy) * fixedPt.getY ( );
matScale3D [2][2] = sz;
matScale3D [2][3] = (1 - sz) * fixedPt.getz ( );
}
```

An inverse, three-dimensional scaling matrix is set up for either Equation 41 or Equation 44 by replacing each scaling parameter ( $s_x$ ,  $s_y$ , and  $s_z$ ) with its reciprocal. However, this inverse transformation is undefined if any scaling parameter is assigned the value 0. The inverse matrix generates an opposite scaling transformation, and the concatenation of a three-dimensional scaling matrix with its inverse yields the identity matrix.

### COMPOSITE

As with two-dimensional transformations, we form a composite three-dimensional transformation by multiplying the matrix representations for

the individual operations in the transformation sequence. Any of the two-dimensional transformation sequences, such as scaling in noncoordinate directions, can be carried out in three-dimensional space.

We can implement a transformation sequence by concatenating the individual matrices from right to left or from left to right, depending on the order in which the matrix representations are specified. Of course, the rightmost term in a matrix product is always the first transformation to be applied to an object and the leftmost term is always the last transformation. We need to use this ordering for the matrix product because coordinate positions are represented as four-element column vectors, which are premultiplied by the composite  $4 \times 4$  transformation matrix.

The following program provides example routines for constructing a three-dimensional composite transformation matrix. The three basic geometric transformations are combined in a selected order to produce a single composite matrix, which is initialized to the identity matrix. For this example, we first rotate, then scale, then translate. We choose a left-to-right evaluation of the composite matrix so that the transformations are called in the order that they are to be applied.

Thus, as each matrix is constructed, it is concatenated on the left of the current composite matrix to form the updated product matrix.

```
class wcPt3D {
public:
  GLfloat x, y, z;
};
typedef GLfloat Matrix4x4 [4][4];
Matrix4x4 matComposite;
/* Construct the 4 x 4 identity matrix. */
void matrix4x4SetIdentity (Matrix4x4 matIdent4x4)
{
  GLint row, col;
  for (row = 0; row < 4; row++)
    for (col = 0; col < 4 ; col++)
      matIdent4x4 [row][col] = (row == col);
}
/* Premultiply matrix m1 by matrix m2, store result in m2. */
void matrix4x4PreMultiply (Matrix4x4 m1, Matrix4x4 m2)
{
  GLint row, col;
  Matrix4x4 matTemp;
```

```

for (row = 0; row < 4; row++)
for (col = 0; col < 4 ; col++)
matTemp [row][col] = m1 [row][0] * m2 [0][col] + m1 [row][1] *
m2 [1][col] + m1 [row][2] * m2 [2][col] +
m1 [row][3] * m2 [3][col];
for (row = 0; row < 4; row++)
for (col = 0; col < 4; col++)
m2 [row][col] = matTemp [row][col];
}
/* Procedure for generating 3-D translation matrix. */
void translate3D (GLfloat tx, GLfloat ty, GLfloat tz)
{
Matrix4x4 matTransl3D;
/* Initialize translation matrix to identity. */
matrix4x4SetIdentity (matTransl3D);
matTransl3D [0][3] = tx;
matTransl3D [1][3] = ty;
matTransl3D [2][3] = tz;
/* Concatenate matTransl3D with composite matrix. */
matrix4x4PreMultiply (matTransl3D, matComposite);
}
/* Procedure for generating a quaternion rotation matrix. */
void rotate3D (wcPt3D p1, wcPt3D p2, GLfloat radianAngle)
{
Matrix4x4 matQuatRot;
float axisVectLength = sqrt ((p2.x - p1.x) * (p2.x - p1.x) +
(p2.y - p1.y) * (p2.y - p1.y) +
(p2.z - p1.z) * (p2.z - p1.z));
float cosA = cosf (radianAngle);
float oneC = 1 - cosA;
float sinA = sinf (radianAngle);
float ux = (p2.x - p1.x) / axisVectLength;
float uy = (p2.y - p1.y) / axisVectLength;
float uz = (p2.z - p1.z) / axisVectLength;
/* Set up translation matrix for moving p1 to origin,
* and concatenate translation matrix with matComposite.
*/
translate3D (-p1.x, -p1.y, -p1.z);
/* Initialize matQuatRot to identity matrix. */
matrix4x4SetIdentity (matQuatRot);
matQuatRot [0][0] = ux*ux*oneC + cosA;
matQuatRot [0][1] = ux*uy*oneC - uz*sinA;
matQuatRot [0][2] = ux*uz*oneC + uy*sinA;
matQuatRot [1][0] = uy*ux*oneC + uz*sinA;
matQuatRot [1][1] = uy*uy*oneC + cosA;
matQuatRot [1][2] = uy*uz*oneC - ux*sinA;
matQuatRot [2][0] = uz*ux*oneC - uy*sinA;
matQuatRot [2][1] = uz*uy*oneC + ux*sinA;
matQuatRot [2][2] = uz*uz*oneC + cosA;

```



```

/* Concatenate matQuatRot with composite matrix. */
matrix4x4PreMultiply (matQuatRot, matComposite);
/* Construct inverse translation matrix for p1 and
* concatenate with composite matrix.
*/
translate3D (p1.x, p1.y, p1.z);
}
/* Procedure for generating a 3-D scaling matrix. */
void scale3D (Gfloat sx, GLfloat sy, GLfloat sz, wcPt3D fixedPt)
{
Matrix4x4 matScale3D;
/* Initialize scaling matrix to identity. */
matrix4x4SetIdentity (matScale3D);
matScale3D [0][0] = sx;
matScale3D [0][3] = (1 - sx) * fixedPt.x;
matScale3D [1][1] = sy;
matScale3D [1][3] = (1 - sy) * fixedPt.y;
matScale3D [2][2] = sz;
matScale3D [2][3] = (1 - sz) * fixedPt.z;
/* Concatenate matScale3D with composite matrix. */
matrix4x4PreMultiply (matScale3D, matComposite);
}
void displayFcn (void)
{
/* Input object description. */
/* Input translation, rotation, and scaling parameters. */
/* Set up 3-D viewing-transformation routines. */
/* Initialize matComposite to identity matrix: */
matrix4x4SetIdentity (matComposite);
/* Invoke transformation routines in the order they
* are to be applied:
*/
rotate3D (p1, p2, radianAngle); // First transformation: Rotate.
scale3D (sx, sy, sz, fixedPt); // Second transformation: Scale.
translate3D (tx, ty, tz); // Final transformation: Translate.
/* Call routines for displaying transformed objects. */
}

```

### **SHEARS AND REFLECTIONS**

In addition to translation, rotation, and scaling, the other transformations discussed for two-dimensional applications are also useful in many three-dimensional situations. These additional transformations include reflection, shear, and transformations between coordinate-reference frames.

#### **Three-Dimensional Reflections**

A reflection in a three-dimensional space can be performed relative to a selected *reflection axis* or with respect to a *reflection plane*. In general, three-dimensional reflection matrices are set up similarly to those for two dimensions. Reflections relative to a given axis are equivalent to 180° rotations about that axis. Reflections with respect to a plane are similar;

when the reflection plane is a coordinate plane ( $xy$ ,  $xz$ , or  $yz$ ), we can think of the transformation as a  $180^\circ$  rotation in four-dimensional space with a conversion between a left-handed frame and a right-handed frame.

An example of a reflection that converts coordinate specifications from a right-handed system to a left-handed system (or vice versa) is shown in Figure 19. This transformation changes the sign of  $z$  coordinates, leaving the values for the  $x$  and  $y$  coordinates unchanged. The matrix representation for this reflection relative to the  $xy$  plane is

$$M_{\text{reflex}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (45)$$

Transformation matrices for inverting  $x$  coordinates or  $y$  coordinates are defined similarly, as reflections relative to the  $yz$  plane or to the  $xz$  plane, respectively. Reflections about other planes can be obtained as a combination of rotations and coordinate-plane reflections:

**FIGURE 19**  
Conversion of coordinate specifications between a right-handed and a left-handed system can be carried out with the reflection transformation 45.



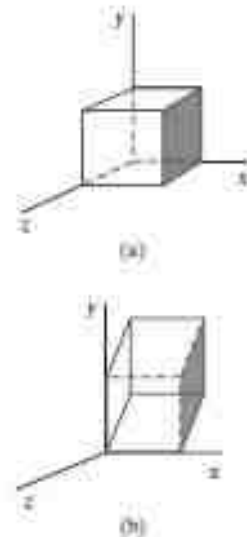
### Three-Dimensional Shears

These transformations can be used to modify object shapes, just as in two-dimensional applications. They are also applied in three-dimensional viewing transformations for perspective projections. For three-dimensional applications, we can also generate shears relative to the  $z$  axis.

A general  $z$ -axis shearing transformation relative to a selected reference position is produced with the following matrix:

$$M_{\text{shear}} = \begin{bmatrix} 1 & 0 & sh_{zx} & -sh_{zx} \cdot z_{ref} \\ 0 & 1 & sh_{zy} & -sh_{zy} \cdot z_{ref} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (46)$$

Shearing parameters  $sh_{zx}$  and  $sh_{zy}$  can be assigned any real values. The effect of this transformation matrix is to alter the values for the  $x$  and  $y$  coordinates by an amount that is proportional to the distance from  $z_{ref}$ , while leaving the  $z$  coordinate unchanged. Plane areas that are perpendicular to the  $z$  axis are thus shifted by an amount equal to  $z - z_{ref}$ . An example of the effect of this shearing matrix on a unit cube is shown in Figure 20 for shearing values  $sh_{zx} = sh_{zy} = 1$  and a reference position  $z_{ref} = 0$ . Three-dimensional transformation matrices for an  $x$ -axis shear and a  $y$ -axis shear are similar to the two-dimensional matrices. We just need to add a row and a column for the  $z$ -coordinate shearing parameters.



**FIGURE 20**  
A unit cube (a) is sheared relative to the origin (b) by Matrix 46, with  $sh_{zx} = sh_{zy} = 1$ .

## THREE DIMENSIONAL VIEWING

For two-dimensional graphics applications, viewing operation transfer positions from the world-coordinate plane to pixel positions in the plane of the output device. Using the rectangular boundaries for the clipping window and the viewport, a two-dimensional package clips a scene and maps it to device coordinates. Three-dimensional viewing operations, however, are more

involved, because we now have many more choices as to how we can construct a scene and how we can generate views of the scene on an output device.

When we model a three-dimensional scene, each object in the scene is typically defined with a set of surfaces that form a closed boundary around the object interior. And, for some applications, we may need also to specify information about the interior structure of an object. In addition to procedures that generate views of the surface features of an object, graphics packages sometimes provide routines for displaying internal components or cross-sectional views of a solid object. Viewing functions process the object descriptions through a set of procedures that ultimately project a specified view of the objects onto the surface of a display device. Many processes in three-dimensional viewing, such as the clipping routines, are similar to those in the two-dimensional viewing pipeline.

But three-dimensional viewing involves some tasks that are not present in two-dimensional viewing. For example, projection routines are needed to transfer the scene to a view on a planar surface, visible parts of a scene must be identified, and, for a realistic display, lighting effects and surface characteristics must be taken into account.

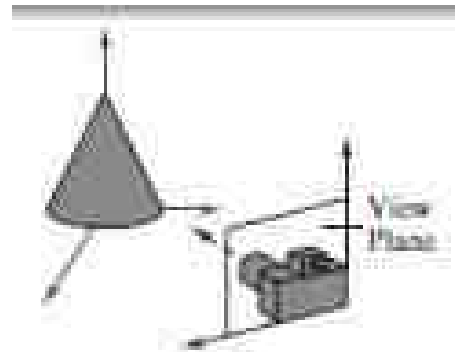
### Viewing a Three-Dimensional Scene

To obtain a display of a three-dimensional world-coordinate scene, we first setup a coordinate reference for the viewing, or “camera,” parameters. This coordinate reference defines the position and orientation for a *view plane* (or *projection plane*) that corresponds to a camera film plane (Figure 1). Object descriptions are then transferred to the viewing reference coordinates and projected onto the view plane. We can generate a view of an object on the output device in wireframe (outline) form, or we can apply lighting and surface-rendering techniques to obtain a realistic shading of the visible surfaces.

### PROJECTION

Unlike a camera picture, we can choose different methods for projecting a scene onto the view plane. One method for getting the description of a solid object onto a view plane is to project points on the object surface along parallel lines.

This technique, called *parallel projection*, is used in engineering and architectural drawings to represent an object with a set of views that show accurate dimensions of the object, as in Figure 2.



**FIGURE 1**  
Coordinate reference for obtaining a selected view of a three-dimensional scene.

**FIGURE 2**  
Three parallel projection views of an object, showing relative proportions from different viewing positions.



Another method for generating a view of a three-dimensional scene is to project points to the view plane along converging paths. This process, called *perspective projection*, causes objects farther from the viewing position to be displayed smaller than objects of the same size that are nearer to the viewing position.

A scene that is generated using a perspective projection appears more realistic, because this is the way that our eyes and a camera lens form images. Parallel lines along the viewing direction appear to converge to a distant point in the background, and objects in the background appear to be smaller than objects in the foreground.

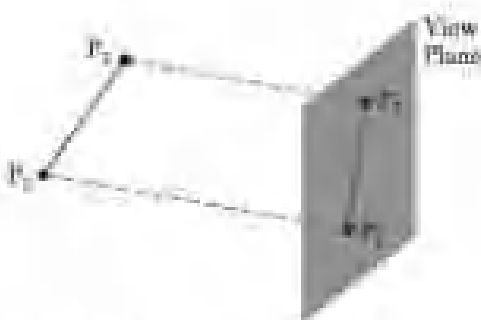
In the next phase of the three-dimensional viewing pipeline, after the transformation to viewing coordinates, object descriptions are projected to the view plane.

Graphics packages generally support both parallel and perspective projections.

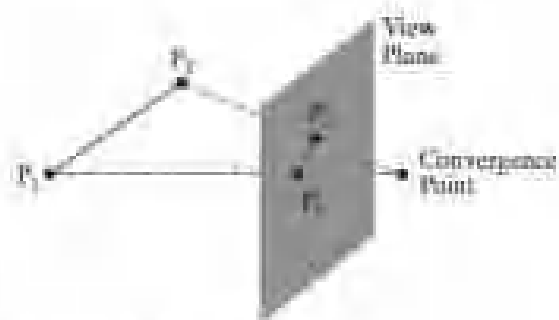
In a **parallel projection**, coordinate positions are transferred to the view plane along parallel lines. Figure 15 illustrates a parallel projection for a straight line segment defined with endpoint coordinates  $P_1$  and  $P_2$ . A parallel projection preserves relative proportions of objects, and this is the method used in computer-aided drafting and design to produce scale drawings of three-dimensional objects.

All parallel lines in a scene are displayed as parallel when viewed with a parallel projection. There are two general methods for obtaining a parallel-projection view of an object: We can project along lines that are perpendicular to the view plane, or we can project at an oblique angle to the view plane.

For a **perspective projection**, object positions are transformed to projection coordinates along lines that converge to a point behind the view plane. An example of a perspective projection for a straight-line segment,



**FIGURE 15**  
Parallel projection of a line segment onto a view plane.



**FIGURE 16**  
Perspective projection of a line segment onto a view plane.

defined with endpoint coordinates  $P1$  and  $P2$ , is given in Figure 16. Unlike a parallel projection, a perspective projection does not preserve relative proportions of objects. But perspective views of a scene are more realistic because distant objects in the projected display are reduced in size.

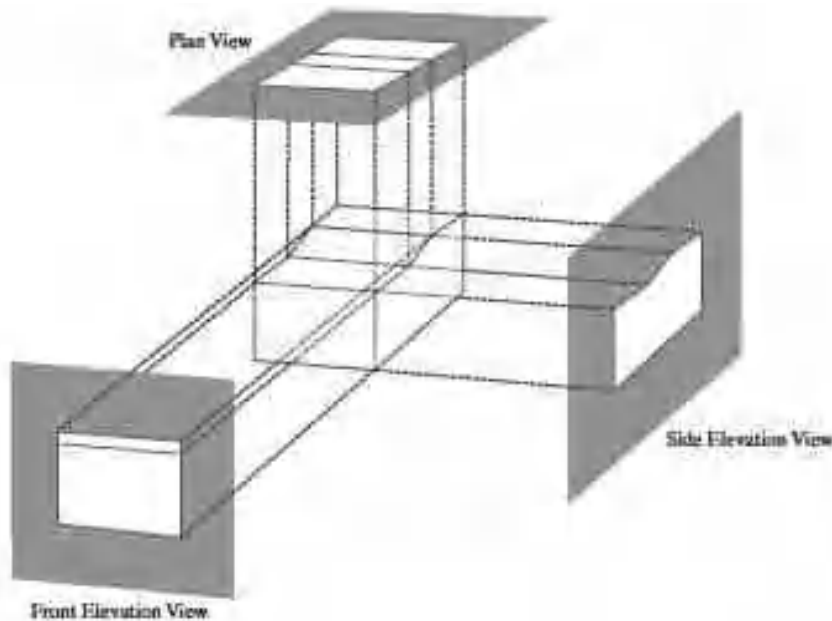


FIGURE 17  
Orthogonal projections of an object,  
displaying plan and elevation views.

## ORTHOGONAL PROJECTIONS

A transformation of object descriptions to a view plane along lines that are all parallel to the view-plane normal vector  $\mathbf{N}$  is called an **orthogonal projection** (or, equivalently, an **orthographic projection**). This produces a parallel-projection transformation in which the projection lines are perpendicular to the view plane.

Orthogonal projections are most often used to produce the front, side, and top views of an object, as shown in Figure 17. Front, side, and rear orthogonal projections of an object are called *elevations*; and a top orthogonal projection is called a *plan view*. Engineering and architectural drawings commonly employ these orthographic projections, because lengths and angles are accurately depicted and can be measured from the drawings.

### Axonometric and Isometric Orthogonal Projections

We can also form orthogonal projections that display more than one face of an object. Such views are called **axonometric** orthogonal projections. The most commonly used axonometric projection is the **isometric** projection, which is generated by aligning the projection plane (or the object) so that the plane intersects each coordinate axis in which the object is defined, called the *principal axes*, at the same distance from the origin. Figure 18 shows an isometric projection for a cube. We can obtain the isometric projection shown in this figure by aligning the view plane normal vector along a cube diagonal. There are eight positions, one in each octant, for obtaining an isometric view. All three principal axes are foreshortened equally in an isometric projection, so that relative proportions are maintained.



This is not the case in a general axonometric projection, where scaling factors may be different for the three principal directions.

### Orthogonal Projection Coordinates

With the projection direction parallel to the  $z$ view axis, the



FIGURE 18  
An isometric projection of a cube.

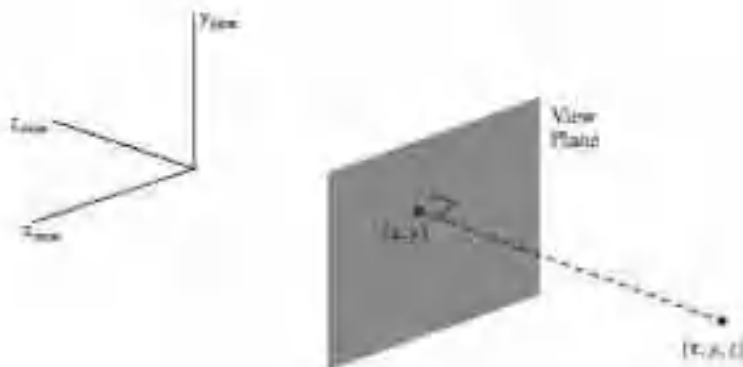


FIGURE 19  
An orthogonal projection of a spatial position onto a view plane.

transformation equations for an orthogonal projection are trivial. For any position  $(x, y, z)$  in viewing coordinates, as in Figure 19, the projection coordinates are  $x_p = x$ ,  $y_p = y$  (6)

The  $z$ -coordinate value for any projection transformation is preserved for use in the visibility determination procedures. And each three-dimensional coordinate point in a scene is converted to a position in normalized space.

### Clipping Window and Orthogonal-Projection View Volume

In the camera analogy, the type of lens is one factor that determines how much of the scene is transferred to the film plane. A wide-angle lens takes in more of the scene than a regular lens. For computer-graphics applications, we use the rectangular *clipping window* for this purpose. As in two-dimensional viewing, graphics packages typically require that clipping rectangles be placed in specific positions.

In OpenGL, we set up a clipping window for three-dimensional viewing just as we did for two-dimensional viewing, by choosing two-dimensional coordinate positions for its lower-left and upper-right corners. For three-dimensional viewing, the clipping window is positioned on the view plane with its edges parallel to the  $x$ view and  $y$ view axes, as shown in Figure 20. If we want to use some other shape or orientation for the clipping window, we must develop our own viewing procedures.

The edges of the clipping window specify the  $x$  and  $y$  limits for the part of the scene that we want to display. These limits are used to form the top, bottom, and two sides of a clipping region called the **orthogonal-projection view volume**. Because projection lines are perpendicular to the view plane, these four boundaries are planes that are also perpendicular to the view plane and that pass

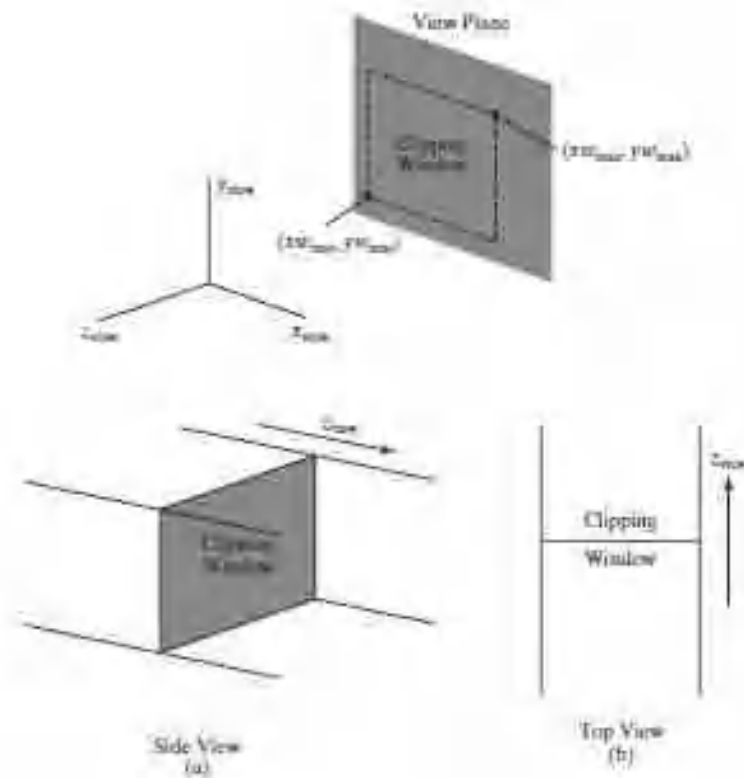


FIGURE 20  
A clipping window on the view plane,  
with minimum and maximum  
coordinates given to the viewing  
reference system.

FIGURE 21  
Infinite orthogonal-projection view  
volume.

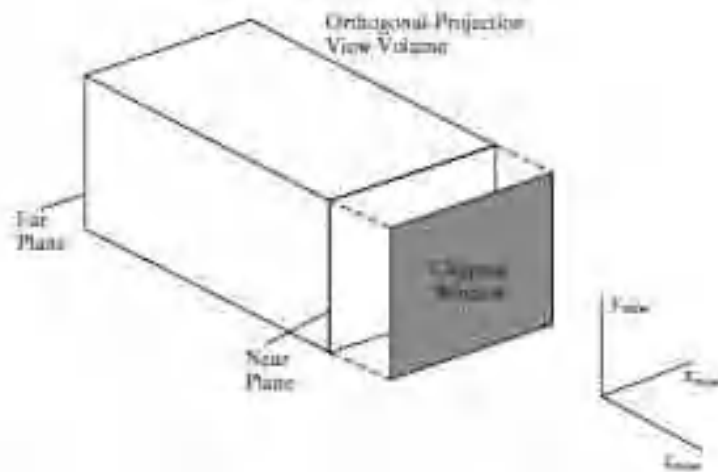
through the edges of the clipping window to form an infinite clipping region, as in Figure 21.

We can limit the extent of the orthogonal view volume in the  $z$  view direction by selecting positions for one or two additional boundary planes that are parallel to the view plane. These two planes are called the **near-far clipping planes**, or the **front-back clipping planes**. The near and far planes allow us to exclude objects that are in front of or behind the part of the scene that we want to display. With the viewing direction along the negative  $z$  view axis, we usually have  $z_{far} < z_{near}$ , so that the far plane is farther out along the negative  $z$  view axis. Some graphics libraries provide these two planes as options, and other libraries require them.

When the near and far planes are specified, we obtain a finite orthogonal view volume that is a *rectangular parallelepiped*, as shown in Figure 22 along with one possible placement for the view plane. Our view of the scene will then contain only those objects within the view volume, with all parts of the scene outside the view volume eliminated by the clipping algorithms.

Graphics packages provide varying degrees of flexibility in the positioning of the near and far clipping planes, including options for specifying additional clipping planes at other positions in the scene. In general, the near and far planes can be in any relative position to each other to achieve various viewing effects, including positions that are on opposite sides of the view point. Similarly, the viewplane can sometimes be placed in any position relative to the near and far clipping planes, although it is often taken to be coincident with the near clipping plane.

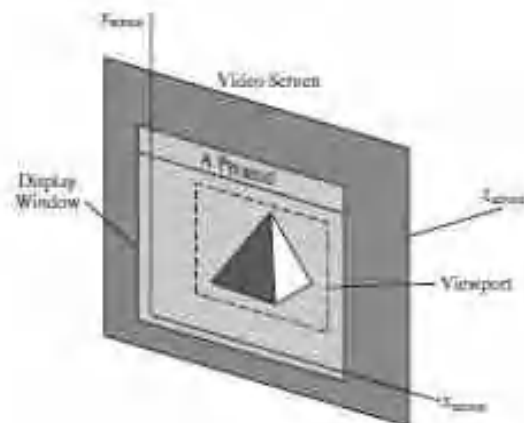
However, providing numerous positioning options for the clipping and viewplanes usually results in less efficient processing of a three-dimensional scene



**FIGURE 22**  
A left-handed orthogonal view volume with the view plane "in front" of the near plane.

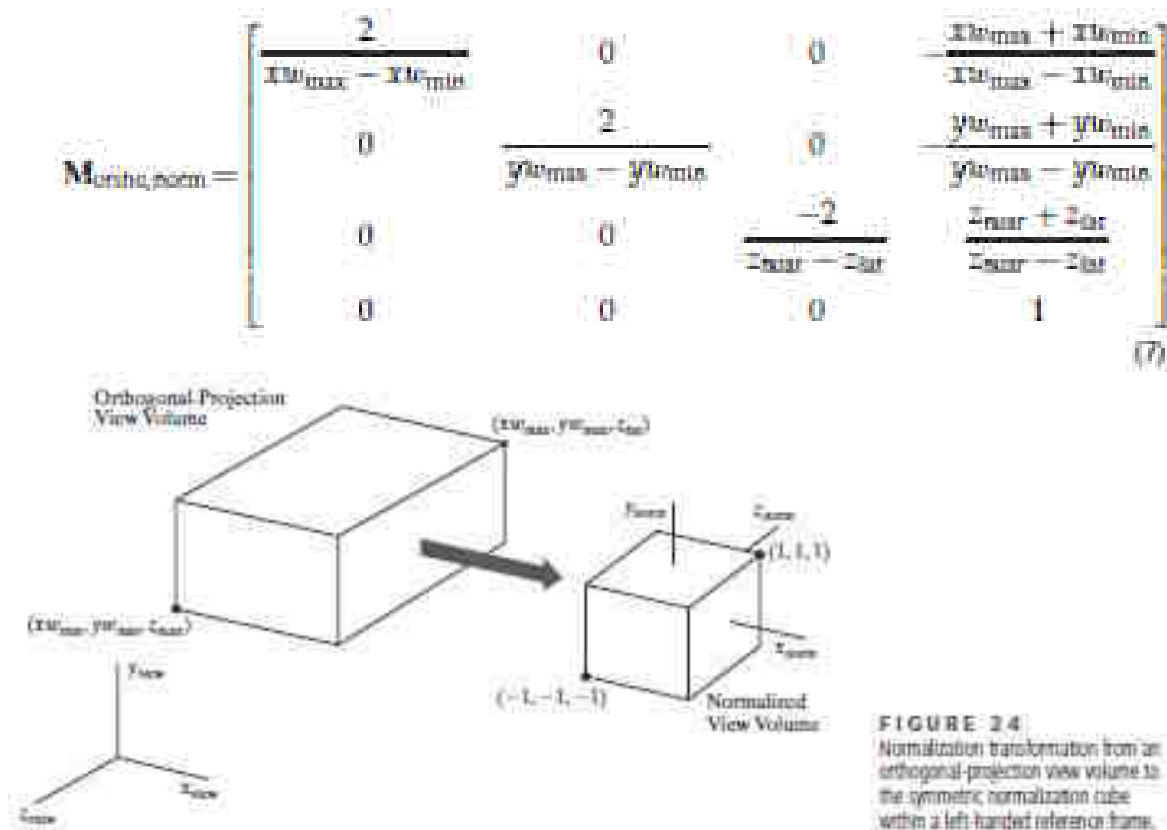
### Normalization Transformation for an Orthogonal Projection

Using an orthogonal transfer of coordinate positions onto the view plane, we obtain the projected position of any spatial point  $(x, y, z)$  as simply  $(x, y)$ . Thus, once we have established the limits for the view volume, coordinate descriptions inside this rectangular parallelepiped are the projection coordinates, and they can be mapped into a **normalized view volume** without any further projection processing. Some graphics packages use a unit cube for this normalized view volume, with each of the  $x$ ,  $y$ , and  $z$  coordinates normalized in the range from 0 to 1. Another normalization-transformation approach is to use a symmetric cube, with coordinates in the range from -1 to 1.



**FIGURE 23**  
A left-handed screen-coordinate reference frame.

Because screen coordinates are often specified in a left-handed reference frame (Figure 23), normalized coordinates also are often specified in a left-handed system. This allows positive distances in the viewing direction to be directly interpreted as distances from the screen (the viewing plane). Thus, we can convert projection coordinates into positions within a left-handed normalized-coordinate reference frame, and these coordinate positions will then be transferred to left-handed screen coordinates by the viewport transformation. To illustrate the normalization transformation, we assume that the orthogonal-projection view volume is to be mapped into the symmetric normalization cube within a left-handed reference frame. Also,  $z$ -coordinate positions for the near and far planes are denoted as  $z_{\text{near}}$  and  $z_{\text{far}}$ , respectively. Figure 24 illustrates this normalization transformation. Position  $(x_{\text{min}}, y_{\text{min}}, z_{\text{near}})$  is mapped to the normalized position  $(-1, -1, -1)$ , and position  $(x_{\text{max}}, y_{\text{max}}, z_{\text{far}})$  is mapped to  $(1, 1, 1)$ .



Transforming the rectangular-parallelepiped view volume to a normalized cube is similar to the methods for converting the clipping window into the normalized symmetric square. The normalization transformation for the orthogonal view volume is multiplied on the right by the composite viewing transformation  $\mathbf{R} \cdot \mathbf{T}$  (Section 4) to produce the complete transformation from world coordinates to normalized orthogonal-projection coordinates.

At this stage of the viewing pipeline, all device-independent coordinate transformations are completed and can be concatenated into a single composite matrix.

Thus, the clipping procedures are most efficiently performed following the normalization transformation. After clipping, procedures for visibility

testing, surfacerendering, and the viewport transformation can be applied to generate thefinal screen display of the scene.

### **OBLIQUE PARALLEL PROJECTIONS.**

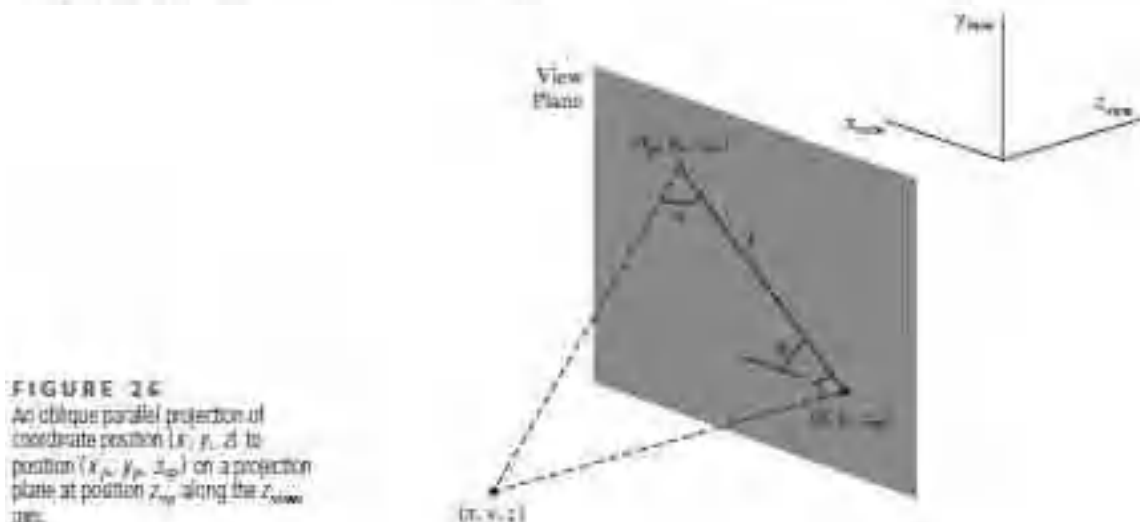
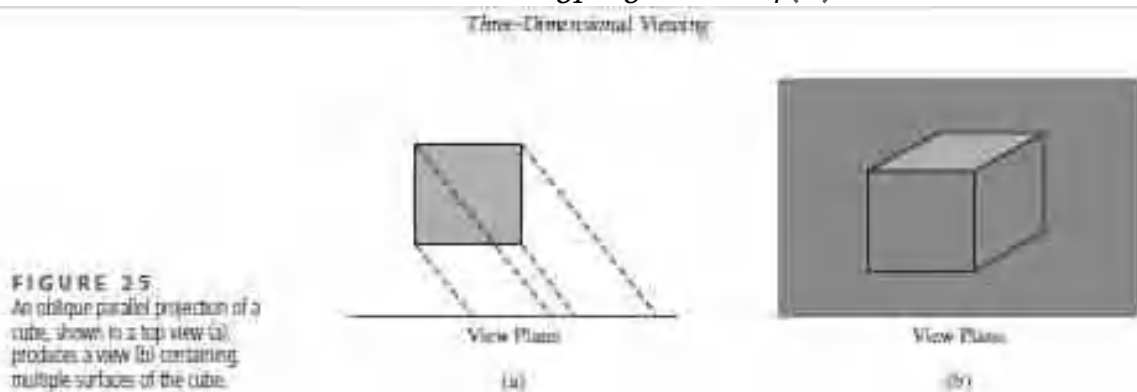
In general, a parallel-projection view of a scene is obtained by transferring objectdescriptions to the view plane along projection paths that can be in any selecteddirection relative to the view-plane normal vector. When the projection path isnot perpendicular to the view plane, this mapping is called an **oblique parallelprojection**. Using this projection, we can produce combinations such as a front,side, and top view of an object, as in Figure 25. Oblique parallel projectionsare defined by a vector direction for the projection lines, and this direction can bespecified in various ways.

#### **Oblique Parallel Projections in Drafting and Design**

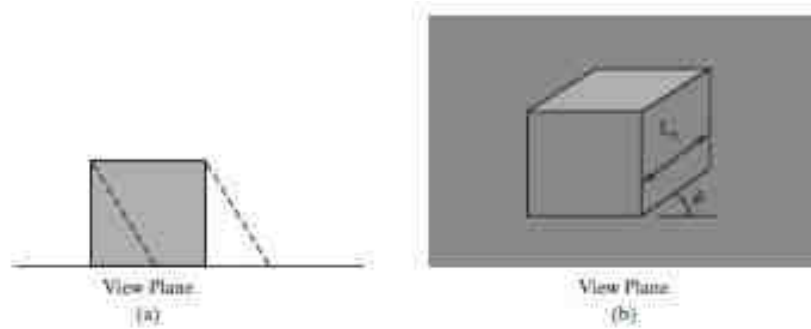
For applications in engineering and architectural design, an oblique parallel projectionis often specified with two angles,  $\alpha$  and  $\varphi$ , as shown in Figure 26. A spatial position  $(x, y, z)$ , in this illustration, is projected to  $(xp, yp, zvp)$  on a viewplane, which is at location  $zvp$  along the viewing  $z$  axis. Position  $(x, y, zvp)$  is thecorresponding orthogonal-projection point. The oblique parallel projection linefrom  $(x, y, z)$  to  $(xp, yp, zvp)$  has an intersection angle  $\alpha$  with the line on the projectionplane that joins  $(xp, yp, zvp)$  and  $(x, y, zvp)$ . This view-plane line, with length $L$ , is at an angle  $\varphi$  with the horizontal direction in the projection plane. Angle  $\alpha$  can be assigned a value between 0 and 90°, and angle  $\varphi$  can vary from 0 to 360°.

We can express the projection coordinates in terms of  $x, y, L$ , and  $\varphi$  as

$$\begin{aligned} xp &= x + L \cos \varphi \\ yp &= y + L \sin \varphi \end{aligned} \quad (8)$$







**FIGURE 27**  
An oblique parallel projection (a) of a cube (top view) onto a view plane that is coincident with the front face of the cube produces the combination front, side, and top view shown in (b).

Length  $L$  depends on the angle  $a$  and the perpendicular distance of the point  $(x, y, z)$  from the view plane:

$$\tan a = \frac{z_p - z}{L} \quad (9)$$

Thus

$$\begin{aligned} L &= \frac{z_p - z}{\tan a} \\ &= L_1(z_p - z) \end{aligned} \quad (10)$$

where  $L_1 = \cot a$ , which is also the value of  $L$  when  $z_p - z = 1$ . We can then write the oblique parallel projection equations 8 as

$$\begin{aligned} x_p &= x + L_1(z_p - z) \cos \varphi \\ y_p &= y + L_1(z_p - z) \sin \varphi \end{aligned} \quad (11)$$

An orthogonal projection is obtained when  $L_1 = 0$  (which occurs at the projection angle  $a = 90^\circ$ ). Equations 11 represent a  $z$ -axis shearing transformation. In fact, the effect of an oblique parallel projection is to shear planes of constant  $z$  and project them onto the view plane. The  $(x, y)$  positions on each plane of constant  $z$  are shifted by an amount proportional to the distance of the plane from the view plane, so that angles, distances, and parallel lines in the plane are projected accurately.

This effect is shown in Figure 27, where the view plane is positioned at the front face of a cube. The back plane of the cube is sheared and overlapped with the front plane in the projection to the viewing surface. A side edge of the cube connecting the front and back planes is projected into a line of length  $L_1$  that makes an angle  $\varphi$  with a horizontal line in the projection plane.

### Cavalier and Cabinet Oblique Parallel Projections

Typical choices for angle  $\varphi$  are  $30^\circ$  and  $45^\circ$ , which display a combination view of the front, side, and top (or front, side, and bottom) of an object. Two commonly used values for  $a$  are those for which  $\tan a = 1$  and  $\tan a = 2$ . For the first case,  $a = 45^\circ$  and the views obtained are called **cavalier** projections. All lines perpendicular to the projection plane are projected with no change in length. Examples of cavalier projections for a cube are given in Figure 28.

When the projection angle  $a$  is chosen so that  $\tan a = 2$ , the resulting views are called a **cabinet** projection. For this angle ( $\approx 63.4^\circ$ ), lines perpendicular to the viewing surface are projected at half their length. Cabinet projections appear more realistic than cavalier projections because of this reduction in the length of perpendiculars. Figure 29 shows examples of cabinet projections for a cube.

FIGURE 28

Cavalier projections of a cube onto a view plane for two values of angle  $\phi$ . The depth of the cube is projected with a length equal to that of the width and height.

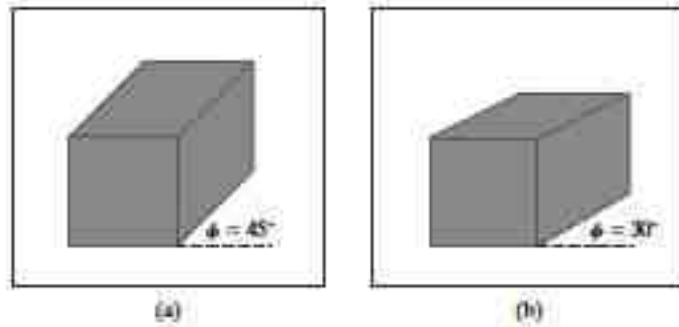
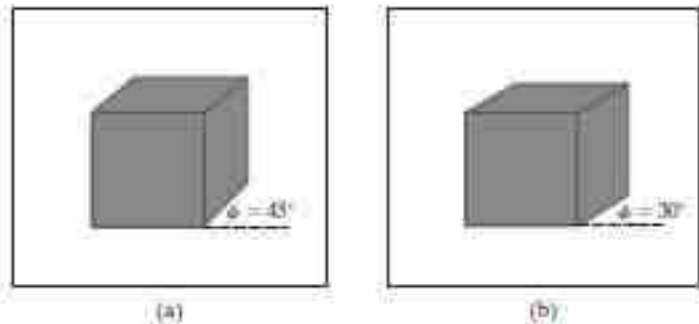


FIGURE 29

Cabinet projections of a cube onto a view plane for two values of angle  $\phi$ . The depth is projected with a length that is one-half that of the width and height of the cube.



### Oblique Parallel-Projection Vector

In graphics programming libraries that support oblique parallel projections, the direction of projection to the view plane is specified with a **parallel-projection vector,  $\mathbf{V}_p$** . This direction vector can be designated with a reference position relative to the view point, as well as with the view-plane normal vector, or with any other two points. Some packages use a reference point relative to the center of the clipping window to define the direction for a parallel projection. If the projection vector is specified in world coordinates, it must first be transformed to viewing coordinates using the rotation matrix discussed in Section 4. (The projection vector is unaffected by the translation, because it is simply a direction with no fixed position.)

Once the projection vector  $\mathbf{V}_p$  is established in viewing coordinates, all points in the scene are transferred to the view plane along lines that are parallel to this vector. Figure 30 illustrates an oblique parallel projection of a spatial point to the view plane. We can denote the components of the projection vector relative to the viewing-coordinate frame as  $\mathbf{V}_p = (V_{px}, V_{py}, V_{pz})$ , where  $V_{py}/V_{px} = \tan \phi$ .

Then, comparing similar triangles in Figure 30, we have

$$\frac{x_p - x}{z_p - z} = \frac{V_{px}}{V_{pz}}$$

$$\frac{y_p - y}{z_p - z} = \frac{V_{py}}{V_{pz}}$$

And we can write the equivalent of the oblique parallel-projection equations 11 in terms of the projection vector as

$$x_p = x + (z_p - z) \frac{V_{px}}{V_{pz}}$$

$$y_p = y + (z_p - z) \frac{V_{py}}{V_{pz}} \quad (12)$$

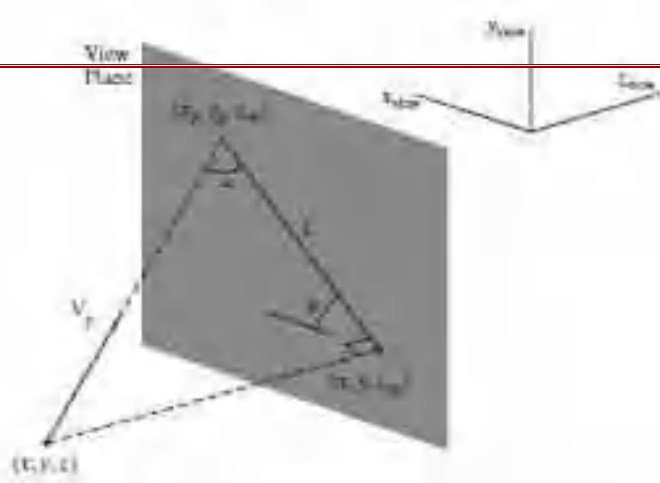


FIGURE 30  
Oblique-parallel projection of position  $(x, y, z)$  to a view plane along a projection line defined with vector  $V_p$ .

The oblique parallel-projection coordinates in 12 reduce to the orthogonal-projection coordinates 6 when  $V_{px} = V_{py} = 0$ .

**Clipping Window and Oblique Parallel-Projection View Volume**

A view volume for an oblique parallel projection is set up using the same procedures as in an orthogonal projection. We select a clipping window on the viewplane with coordinate positions  $(x_{wmin}, y_{wmin})$  and  $(x_{wmax}, y_{wmax})$ , for the lower-left and upper-right corners of the clipping rectangle. The top, bottom, and sides of the view volume are then defined by the direction of projection and the edges of the clipping window. In addition, we can limit the extent of the view volume by adding a near plane and a far plane, as in Figure 31. The finite oblique-parallel-projection view volume is an oblique parallelepiped.

Oblique parallel projections may be affected by changes in the position of the view plane, depending on how the projection direction is to be specified. In some systems, the oblique parallel-projection direction is parallel to the line connecting a reference point to the center of the clipping window. Therefore, moving the position of the view plane or clipping window without adjusting the reference point changes the shape of the view volume.

**Oblique Parallel-Projection Transformation Matrix**

Using the projection-vector parameters from the equations in 12, we can express the elements of the transformation matrix for an oblique parallel projection as

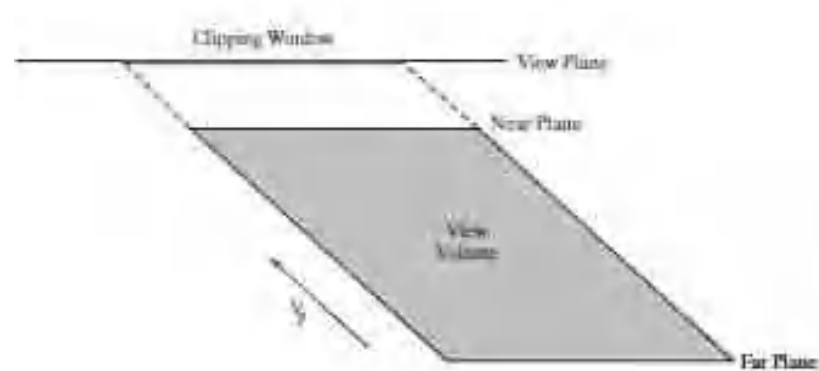


FIGURE 31  
Top view of a finite view volume for an oblique parallel projection in the direction of vector  $V_p$ .

This matrix shifts the values of the  $x$  and  $y$  coordinates by an amount proportional to the distance from the view plane, which is at position  $z_{vp}$  on the  $z$  view axis. The  $z$  values of spatial

$$M_{oblique} = \begin{bmatrix} 1 & 0 & -\frac{V_{pz}}{V_{px}} & \frac{V_{pz}}{V_{px}} \\ 0 & 1 & -\frac{V_{pz}}{V_{py}} & \frac{V_{pz}}{V_{py}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (13)$$

positions are unchanged. If  $V_{px} = V_{py} = 0$ , we have an orthogonal projection and matrix 13 is reduced to the identity matrix.

For a general oblique parallel projection, matrix 13 represents a  $z$ -axis shearing transformation. All coordinate positions within the oblique view volume are sheared by an amount proportional to their distance from the view plane. The effect is to shear the oblique view volume into a rectangular parallelepiped, as illustrated in Figure 32. Thus, positions inside the view volume are sheared into orthogonal-projection coordinates by the oblique parallel-projection transformation.

### Normalization Transformation for an Oblique Parallel Projection

Because the oblique parallel-projection equations convert object descriptions to orthogonal-coordinate positions, we can apply the normalization procedures following this transformation. The oblique view volume has been converted to a rectangular parallelepiped, so we use the same procedures as in Section 6.

Following the normalization example in Section 6, we again map to the symmetric normalized cube within a left-handed coordinate frame. Thus, the complete transformation, from viewing coordinates to normalized coordinates, for an oblique parallel projection is

$$\mathbf{M}_{\text{oblique, norm}} = \mathbf{M}_{\text{ortho, norm}} \cdot \mathbf{M}_{\text{oblique}} \quad (14)$$

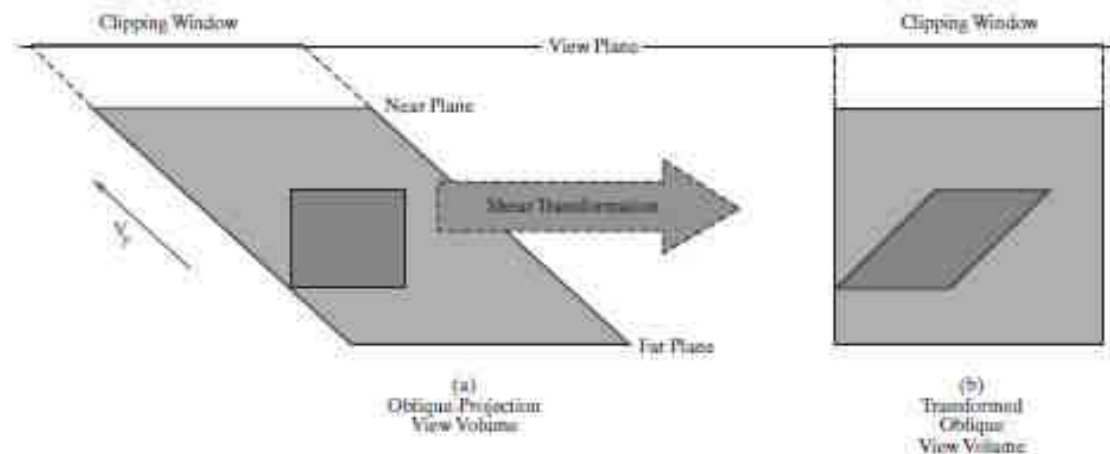


FIGURE 32

Top view of an oblique parallel-projection transformation. The oblique view volume is converted into a rectangular parallelepiped, and objects in the view volume, such as the green block, are mapped to orthogonal-projection coordinates.

Transformation  $\mathbf{M}_{\text{oblique}}$  is matrix 13, which converts the scene description to orthogonal-projection coordinates; and transformation  $\mathbf{M}_{\text{ortho, norm}}$  is matrix 7, which maps the contents of the orthogonal view volume to the symmetric normalization cube.

To complete the viewing transformations (with the exception of the mapping to viewport screen coordinates), we concatenate matrix 14 to the left of the transformation  $\mathbf{M}_{WC, VC}$  from Section 4. Clipping routines can then be applied to the normalized view volume, followed by the determination of visible objects, the surface-rendering procedures, and the viewport transformation.

## UNIT 4: VIEWING

Although a parallel-projection view of a scene is easy to generate and preserves relative proportions of objects, it does not provide a realistic representation. To simulate a camera picture, we need to consider that reflected light rays from the objects in a scene follow converging paths to the camera film plane. We can approximate this geometric-optics effect by projecting objects to the view plane along converging paths to a position called the **projection reference point** (or **center of projection**). Objects are then displayed with foreshortening effects, and projections of distant objects are smaller than the projections of objects of the same size that are closer to the view plane (Figure 33).

## PERSPECTIVE PROJECTION

### Perspective-Projection Transformation Coordinates

We can sometimes select the projection reference point as another viewing parameter in a graphics package, but some systems place this

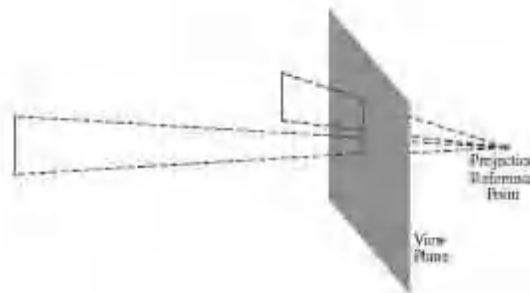


FIGURE 33  
A perspective projection of two equal-length line segments at different distances from the view plane.

convergence point at a fixed position, such as at the view point. Figure 34 shows the projection path of a spatial position  $(x, y, z)$  to a general projection reference point at  $(x_{prp}, y_{prp}, z_{prp})$ . The projection line intersects the view plane at the coordinate position  $(x_p, y_p, z_p)$ , where  $z_p$  is some selected position for the view plane on the  $z$  view axis. We can write equations describing coordinate positions along this perspective-projection line in parametric form as

$$\begin{aligned} X' &= x - (x - x_{prp})u \\ Y' &= y - (y - y_{prp})u \\ Z' &= z - (z - z_{prp})u \end{aligned} \quad 0 \leq u \leq 1 \quad (15)$$

Coordinate position  $(x', y', z')$  represents any point along the projection line. When  $u = 0$ , we are at position  $\mathbf{P} = (x, y, z)$ . At the other end of the line,  $u = 1$  and

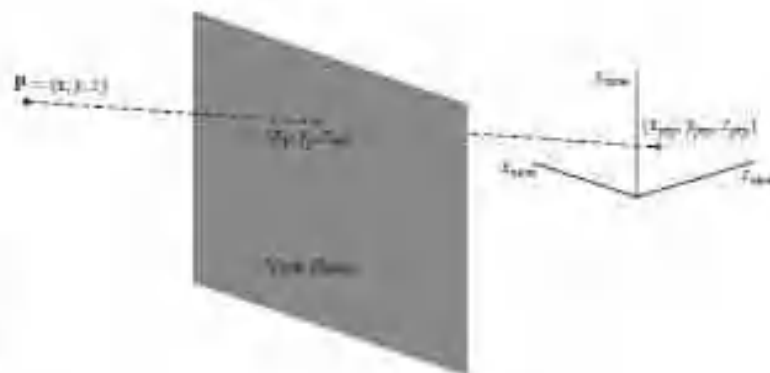


FIGURE 34  
A perspective projection of a point  $\mathbf{P}$  with coordinates  $(x, y, z)$  to a selected projection reference point. The intersection position on the view plane is  $(x_p, y_p, z_p)$ .



we have the projection reference-point coordinates  $(x_{prp}, y_{prp}, z_{prp})$ . On the view plane  $z = z_{vp}$  and we can solve the  $z'$  equation for parameter  $u$  at this position along the projection line:

$$u = \frac{z_{vp} - z}{z_{prp} - z} \quad (16)$$

Substituting this value of  $u$  into the equations for  $x'$  and  $y'$ , we obtain the general perspective-transformation equations:

$$\begin{aligned} x_p &= x \left( \frac{z_{prp} - z_{vp}}{z_{prp} - z} \right) + x_{prp} \left( \frac{z_{vp} - z}{z_{prp} - z} \right) \\ y_p &= y \left( \frac{z_{prp} - z_{vp}}{z_{prp} - z} \right) + y_{prp} \left( \frac{z_{vp} - z}{z_{prp} - z} \right) \end{aligned} \quad (17)$$

Calculations for a perspective mapping are more complex than the parallel-projection equations, because the denominators in the perspective calculations 17 are functions of the  $z$  coordinate of the spatial position. Therefore, we now need to formulate the perspective-transformation procedures a little differently so that this mapping can be concatenated with the other viewing transformations.

But first we take a look at some of the properties of Equations 17.

### Perspective-Projection Equations: Special Cases

Various restrictions are often placed on the parameters for a perspective projection. Depending on a particular graphics package, positioning for either the projection reference point or the view plane may not be completely optional.

To simplify the perspective calculations, the projection reference point could be limited to positions along the  $z_{view}$  axis, then

1.  $x_{prp} = y_{prp} = 0$ :

$$x_p = x \left( \frac{z_{prp} - z_{vp}}{z_{prp} - z} \right), \quad y_p = y \left( \frac{z_{prp} - z_{vp}}{z_{prp} - z} \right) \quad (18)$$

Sometimes the projection reference point is fixed at the coordinate origin, and

2.  $(x_{prp}, y_{prp}, z_{prp}) = (0, 0, 0)$ :

$$x_p = x \left( \frac{z_{vp}}{z} \right), \quad y_p = y \left( \frac{z_{vp}}{z} \right) \quad (19)$$

If the view plane is the  $uv$  plane and there are no restrictions on the placement of the projection reference point, then we have

3.  $z_{vp} = 0$ :

$$\begin{aligned} x_p &= x \left( \frac{z_{prp}}{z_{prp} - z} \right) - x_{prp} \left( \frac{z}{z_{prp} - z} \right) \\ y_p &= y \left( \frac{z_{prp}}{z_{prp} - z} \right) - y_{prp} \left( \frac{z}{z_{prp} - z} \right) \end{aligned} \quad (20)$$

With the  $uv$  plane as the view plane and the projection reference point on the  $z_{view}$  axis, the perspective equations are

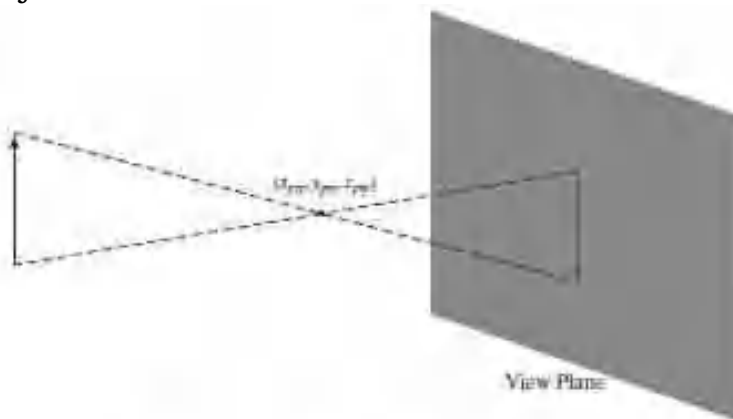
4.  $x_{prp} = y_{prp} = z_{prp} = 0$ :

$$x_p = x \left( \frac{z_{prp}}{z_{prp} - z} \right), \quad y_p = y \left( \frac{z_{prp}}{z_{prp} - z} \right) \quad (21)$$

Of course, we cannot have the projection reference point on the view plane. In that case, the entire scene would project to a single point. The view plane is usually placed between the projection reference point and the scene, but, in general, the view plane could be placed anywhere except at the projection point.

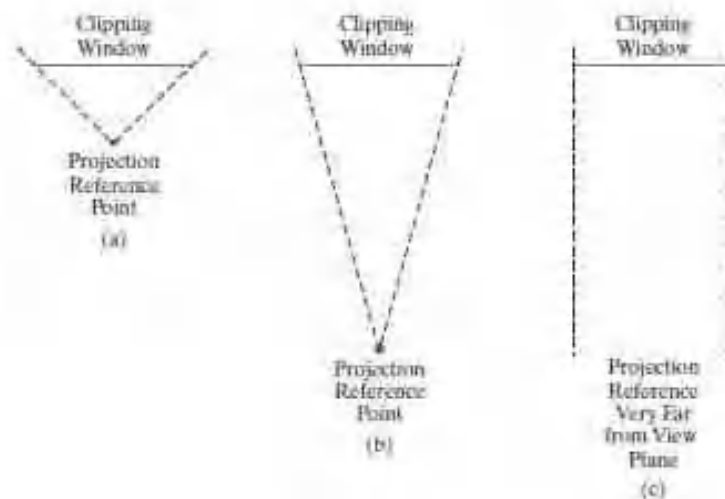
If the projection reference point is between the view plane and the scene, objects are inverted on the view plane (Figure 35). With the scene between the view plane and the projection point, objects are simply enlarged as they are projected away from the viewing position onto the view plane.

Perspective effects also depend on the distance between the projection reference point and the view plane, as illustrated in Figure 36. If the projection



**FIGURE 35**

A perspective-projection view of an object is upside down when the projection reference point is between the object and the view plane.



**FIGURE 36**

Changing perspective effects by moving the projection reference point away from the view plane.

reference point is close to the view plane, perspective effects are emphasized; that is, closer objects will appear much larger than more distant objects of the same size.

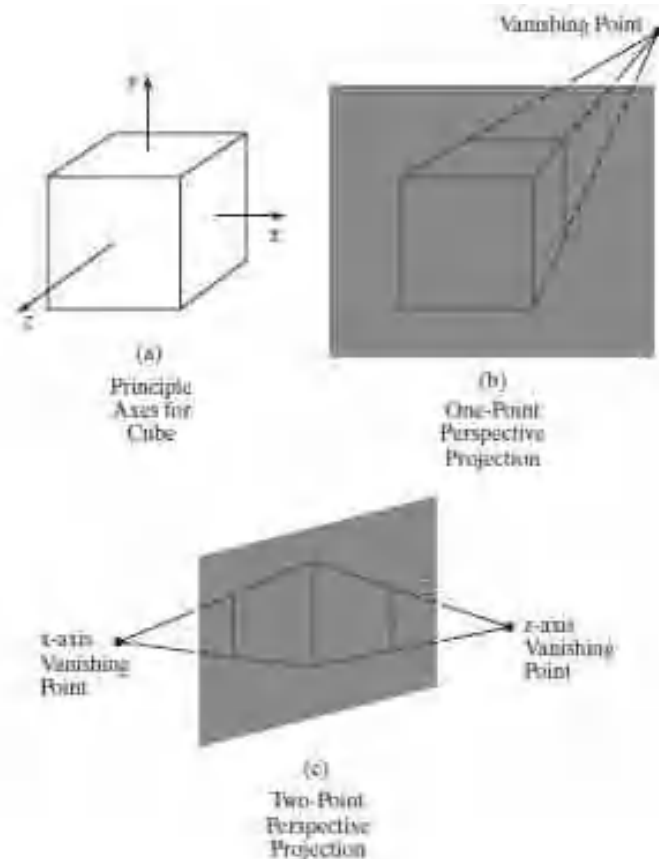
Similarly, as the projection reference point moves farther from the view plane, the difference in the size of near and far objects decreases. When the projection reference point is very far from the view plane, a perspective projection approaches a parallel projection.

### **Vanishing Points for Perspective Projections**

When a scene is projected onto a view plane using a perspective mapping, lines that are parallel to the view plane are projected as parallel

lines. But any parallel lines in the scene that are not parallel to the view plane are projected into converging lines. The point at which a set of projected parallel lines appears to converge is called a **vanishing point**. Each set of projected parallel lines has a separate vanishing point.

For a set of lines that are parallel to one of the principal axes of an object, the vanishing point is referred to as a **principal vanishing point**. We control the number of principal vanishing points (one, two, or three) with the orientation of the projection plane, and perspective projections are accordingly classified as one-point, two-point, or three-point projections. The number of principal vanishing points in a projection is equal to the number of principal axes that intersect the view plane. Figure 37 illustrates the appearance of one-point and two-point perspective projections for a cube. In the projected view (b), the view plane is aligned parallel to the  $xy$  object plane so that only the object  $z$  axis is intersected.



**FIGURE 37**

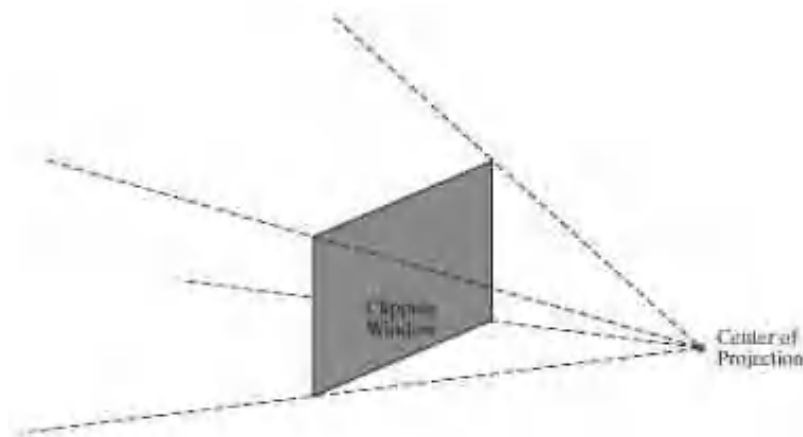
Principal vanishing points for perspective projection views of a cube. When the cube in (a) is projected to a view plane that intersects only the  $z$  axis, a single vanishing point in the  $z$  direction (b) is generated. When the cube is projected to a view plane that intersects both the  $x$  and  $z$  axes, two vanishing points (c) are produced.

This orientation produces a one-point perspective projection with a  $z$ -axis vanishing point. For the view shown in (c), the projection plane intersects both the  $x$  and  $z$  axes but not the  $y$  axis. The resulting two-point perspective projection contains both  $x$ -axis and  $z$ -axis vanishing points. There is not much increase in the realism of a three-point perspective projection compared to a two-point projection, so three-point projections are not used as often in architectural and engineering drawings.

### Perspective-Projection View Volume

We again create a view volume by specifying the position of a rectangular clipping window on the view plane. But now the bounding planes for the view volume are not parallel, because the projection lines are not

parallel. The bottom, top, and sides of the view volume are planes through the window edges that all intersect at the projection reference point. This forms a view volume that is an infinite rectangular pyramid with its apex at the



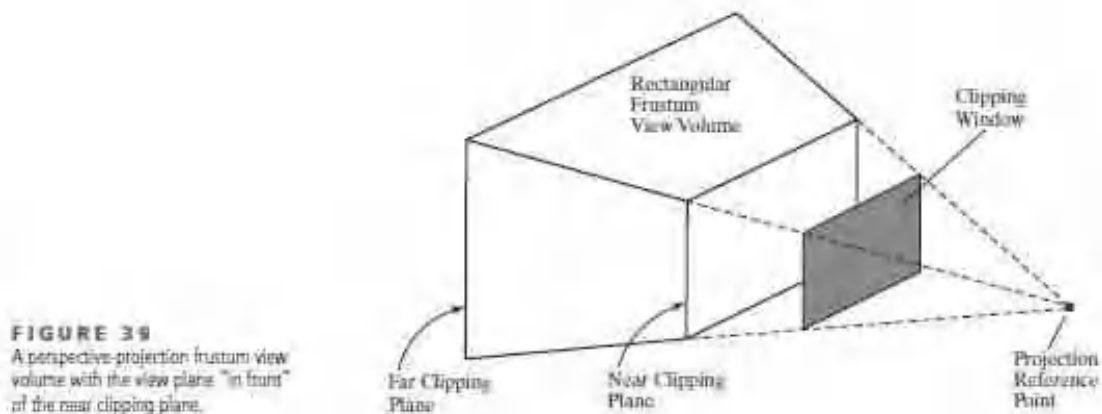
**FIGURE 38**  
An infinite, pyramid view volume for a perspective projection.

center of projection (Figure 38). All objects outside this pyramid are eliminated by the clipping routines. A perspective-projection view volume is often referred to as a **pyramid of vision** because it approximates the *cone of vision* of our eyes or a camera. The displayed view of a scene includes only those objects within the pyramid, just as we cannot see objects beyond our peripheral vision, which are outside the cone of vision.

By adding near and far clipping planes that are perpendicular to the z-view axis (and parallel to the view plane), we chop off parts of the infinite, perspective projection view volume to form a truncated pyramid, or **frustum**, view volume.

Figure 39 illustrates the shape of a finite, perspective-projection view volume with a view plane that is placed between the near clipping plane and the projection reference point. Sometimes the near and far planes are required in a graphics package, and sometimes they are optional.

Usually, both the near and far clipping planes are on the same side of the projection reference point, with the far plane farther from the projection point than the near plane along the viewing direction. And, as in a parallel projection, we can use the near and far planes simply to enclose the scene to be viewed. But with a perspective projection, we could also use the near



**FIGURE 39**  
A perspective-projection frustum view volume with the view plane "in front" of the near clipping plane.

clipping plane to take out large objects close to the view plane that could project into unrecognizable shapes within the clipping window. Similarly, the

far clipping plane could be used to cut out objects far from the projection reference point that might project to small blotson the view plane. Some systems restrict the placement of the view plane relative to the near and far planes, and other systems allow it to be placed anywhere except at the position of the projection reference point. If the view plane is “behind” the projection reference point, objects are inverted, as shown in Figure 35.

### Perspective-Projection Transformation Matrix

Unlike a parallel projection, we cannot directly use the coefficients of the  $x$  and  $y$  coordinates in equations 17 to form the perspective-projection matrix elements, because the denominators of the coefficients are functions of the  $z$  coordinate. But we can use a three-dimensional, homogeneous-coordinate representation to express the perspective-projection equations in the form

$$x_p = \frac{x_h}{h}, \quad y_p = \frac{y_h}{h} \quad (22)$$

where the homogeneous parameter has the value

$$h = z_{pp} - z \quad (23)$$

The numerators in 22 are the same as in equations 17:

$$\begin{aligned} x_h &= x(z_{pp} - z_{sp}) + x_{pp}(z_{sp} - z) \\ y_h &= y(z_{pp} - z_{sp}) + y_{pp}(z_{sp} - z) \end{aligned} \quad (24)$$

Thus, we can set up a transformation matrix to convert a spatial position to homogeneous coordinates so that the matrix contains only the perspective parameters and not coordinate values. The perspective-projection transformation of a viewing-coordinate position is then accomplished in two steps. First, we calculate the homogeneous coordinates using the perspective-transformation matrix:

$$\mathbf{P}h = \mathbf{M}_{\text{pers}} \cdot \mathbf{P} \quad (25)$$

where  $\mathbf{P}h$  is the column-matrix representation of the homogeneous point  $(xh, yh, zh, h)$  and  $\mathbf{P}$  is the column-matrix representation of the coordinate position  $(x, y, z, 1)$ . (Actually, the perspective matrix would be concatenated with the other viewing-transformation matrices, and then the composite matrix would be applied to the world-coordinate description of a scene to produce homogeneous coordinates.) Second, after other processes have been applied, such as the normalization transformation and clipping routines, homogeneous coordinates are divided by parameter  $h$  to obtain the true transformation-coordinate positions. Setting up matrix elements for obtaining the homogeneous-coordinate  $xh$  and  $yh$  values in 24 is straightforward, but we must also structure the matrix to preserve depth ( $z$ -value) information. Otherwise, the  $z$  coordinates are distorted by the homogeneous-division parameter  $h$ . We can do this by setting up the matrix elements for the  $z$  transformation so as to normalize the perspective-projection  $z_p$  coordinates. There are various ways that we could choose the matrix elements to produce the homogeneous coordinates 24 and the



normalized  $z_p$  value for a spatial position  $(x, y, z)$ . The following matrix gives one possible way to formulate a perspective-projection matrix.

$$M_{\text{persp}} = \begin{bmatrix} \frac{z_{\text{prp}} - z_{\text{np}}}{z_{\text{prp}}} & 0 & -x_{\text{prp}} & x_{\text{prp}} z_{\text{prp}} \\ 0 & \frac{z_{\text{prp}} - z_{\text{np}}}{z_{\text{prp}}} & -y_{\text{prp}} & y_{\text{prp}} z_{\text{prp}} \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & z_{\text{prp}} \end{bmatrix} \quad (26)$$

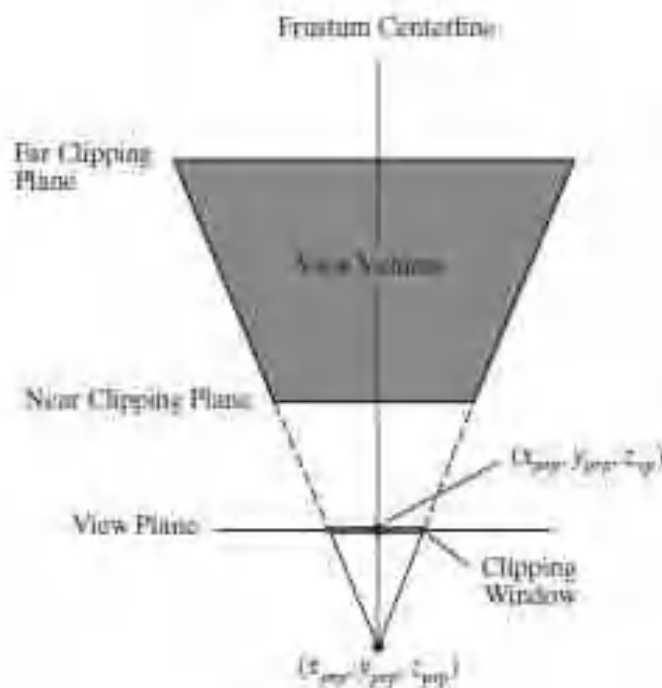
Parameters  $s_z$  and  $t_z$  are the scaling and translation factors for normalizing the projected values of  $z$ -coordinates. Specific values for  $s_z$  and  $t_z$  depend on the normalization range we select.

Matrix 26 converts the description of a scene into homogeneous parallel projection coordinates. However, the frustum view volume can have any orientation, so that these transformed coordinates could correspond to an oblique parallel projection. This occurs if the frustum view volume is not symmetric. If the frustum view volume for the perspective projection is symmetric, the resulting parallel-projection coordinates correspond to an orthogonal projection. We next consider these two possibilities.

#### Symmetric Perspective-Projection Frustum

The line from the projection reference point through the center of the clipping window and on through the view volume is the centerline for a perspective projection frustum. If this centerline is perpendicular to the view plane, we have a **symmetric frustum** (with respect to its centerline) as in Figure 40.

Because the frustum centerline intersects the view plane at the coordinate location  $(x_{\text{prp}}, y_{\text{prp}}, z_{\text{prp}})$ , we can express the corner positions for the clipping



**FIGURE 40**

A symmetric perspective-projection frustum view volume, with the view plane between the projection reference point and the near clipping plane. This frustum is symmetric about its centerline when viewed from above, below, or either side.

window in terms of the window dimensions:

$$\begin{aligned}xW_{\min} &= x_{prp} - \frac{\text{width}}{2}, & xW_{\max} &= x_{prp} + \frac{\text{width}}{2} \\yW_{\min} &= y_{prp} - \frac{\text{height}}{2}, & yW_{\max} &= y_{prp} + \frac{\text{height}}{2}\end{aligned}$$

Therefore, we could specify a symmetric perspective-projection view of a scene using the width and height of the clipping window instead of the window coordinates.

This uniquely establishes the position of the clipping window, because it is symmetric about the  $x$  and  $y$  coordinates of the projection reference point.

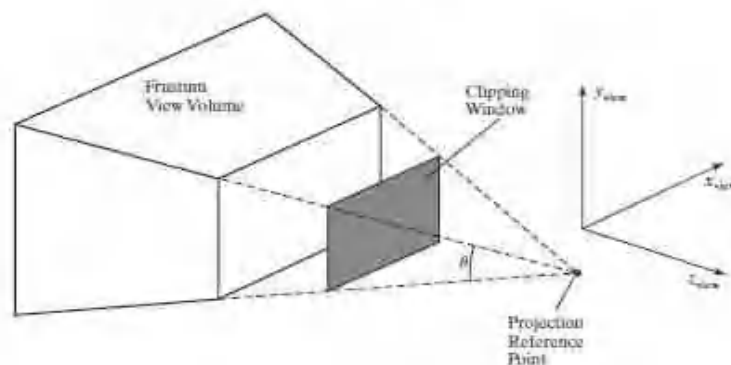
Another way to specify a symmetric perspective projection is to use parameters that approximate the properties of a camera lens. A photograph is produced with a symmetric perspective projection of a scene onto the film plane. Reflected light rays from the objects in a scene are collected on the film plane from within the “cone of vision” of the camera. This cone of vision can be referenced with a **field-of-view angle**, which is a measure of the size of the camera lens. A large field-of-view angle, for example, corresponds to a wide-angle lens. In computer graphics, the cone of vision is approximated with a symmetric frustum, and we can use a field-of-view angle to specify an angular size for the frustum. Typically, the field-of-view angle is the angle between the top clipping plane and the bottom clipping plane of the frustum, as shown in Figure 41.

For a given projection reference point and view-plane position, the field-of-view angle determines the height of the clipping window (Figure 42), but not the width. We need an additional parameter to define completely the clipping window dimensions, and this second parameter could be either the window width or the aspect ratio (width/height) of the clipping window. From the right triangles in the diagram of Figure 42, we see that

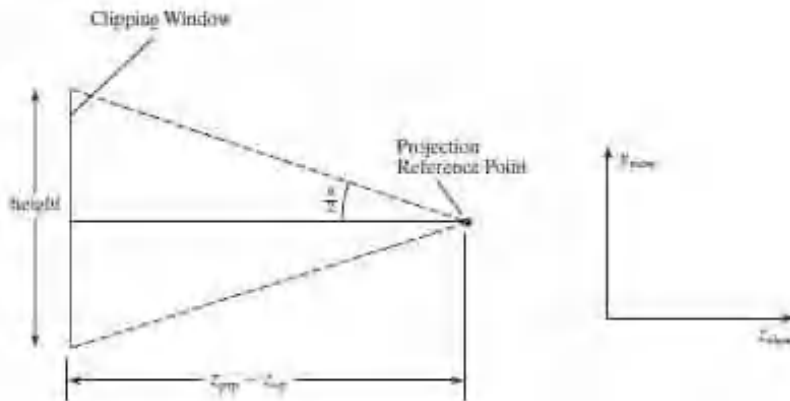
$$\tan\left(\frac{\theta}{2}\right) = \frac{\text{height}/2}{z_{prp} - z_{cp}} \quad (27)$$

so that the clipping-window height can be calculated as

$$\text{height} = 2(z_{prp} - z_{cp}) \tan\left(\frac{\theta}{2}\right) \quad (28)$$



**FIGURE 41**  
Field-of-view angle  $\theta$  for a symmetric perspective-projection view volume, with the clipping plane between the near clipping plane and the projection reference point.



**FIGURE 42**  
Relationship between the field-of-view angle  $\theta$ , the height of the clipping window, and the distance between the projection reference point and the view plane.

Therefore, the diagonal elements with the value  $z_{pp} - z_{vp}$  in matrix 26) could be replaced by either of the following two expressions,

$$\begin{aligned} z_{pp} - z_{vp} &= \frac{\text{height}}{2} \cot\left(\frac{\theta}{2}\right) \\ &= \frac{\text{width} \cdot \cot(\theta/2)}{2 \cdot \text{aspect}} \end{aligned} \quad (29)$$

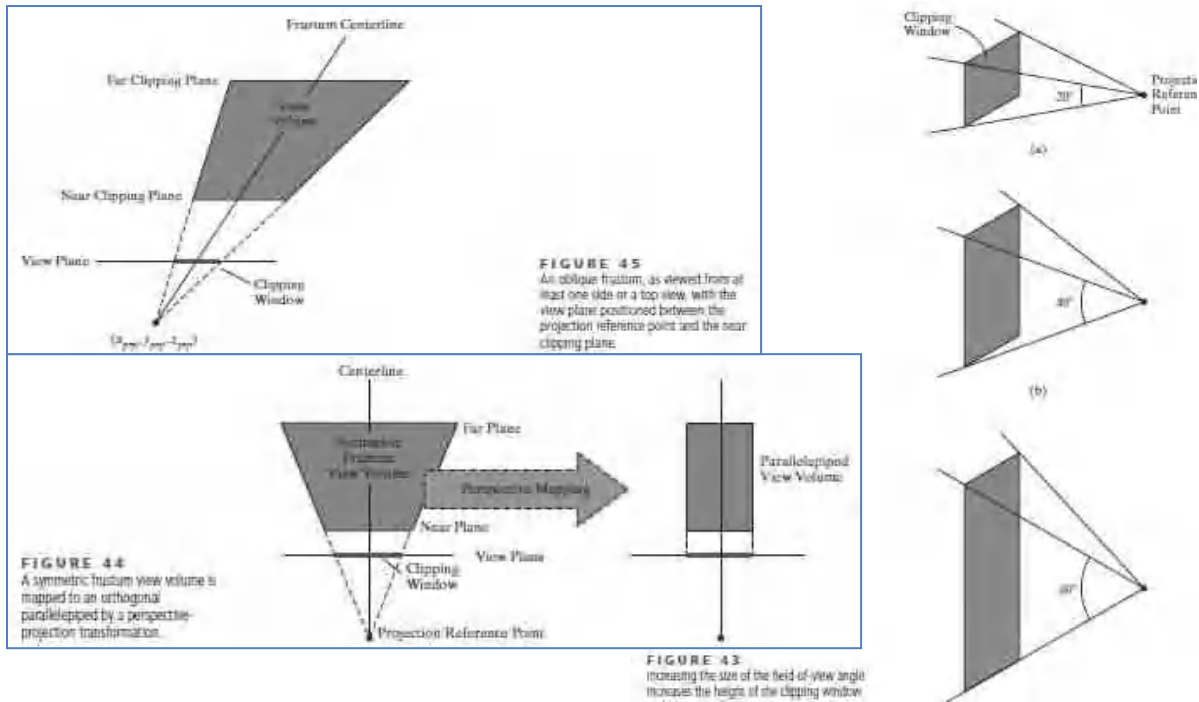
In some graphics libraries, fixed positions are used for the view plane and the projection reference point, so that a symmetric perspective projection is completely specified by the field-of-view angle, the aspect ratio of the clipping window, and the distances from the viewing position to the near and far clipping planes. The same aspect ratio is usually applied to the specification of the viewport.

If the field-of-view angle is decreased in a particular application, the foreshortening effects of a perspective projection are also decreased. This is comparable to moving the projection reference point farther from the view plane. Also, decreasing the field-of-view angle decreases the height of the clipping window, and this provides a method for zooming in on small regions of a scene. Similarly, a large field-of-view angle results in a large clipping-window height (a zoom out), and it increases perspective effects, which is what we achieve when we set the projection reference point close to the view plane. Figure 43 illustrates the effects of various field-of-view angles for a fixed-width clipping window.

When the perspective-projection view volume is a symmetric frustum, the perspective transformation maps locations inside the frustum to orthogonal projection coordinates within a rectangular parallelepiped. The centerline of the parallelepiped is the frustum centerline, because this line is already perpendicular to the view plane (Figure 44). This is a consequence of the fact that all positions along a projection line within the frustum map to the same point  $(x_p, y_p)$  on the view plane. Thus, each projection line is converted by the perspective transformation to a line that is perpendicular to the view plane and, thus, parallel to the frustum centerline. With the symmetric frustum converted to an orthogonal projection view volume, we can next apply the normalization transformation.

### **Oblique Perspective-Projection Frustum**

If the centerline of a perspective-projection view volume is not perpendicular to the view plane, we have an **oblique frustum**. Figure 45 illustrates the general



appearance of an oblique perspective-projection view volume. In this case, we can first transform the view volume to a symmetric frustum and then to a normalized view volume.

An oblique perspective-projection view volume can be converted to a symmetric frustum by applying a  $z$ -axis shearing-transformation matrix. This transformation shifts all positions on any plane that is perpendicular to the  $z$  axis by an amount that is proportional to the distance of the plane from a specified  $z$ -axis reference position. In this case, the reference position is  $z_{prp}$ , which is the  $z$  coordinate of the projection reference point. And we need to shift by an amount that will move the center of the clipping window to position  $(x_{prp}, y_{prp})$  on the view plane. Because the frustum centerline passes through the center of the clipping window, this shift adjusts the centerline so that it is perpendicular to the view plane, as in Figure 40.

The computations for the shearing transformation, as well as for the perspective and normalization transformations, are greatly reduced if we take the projection reference point to be the viewing-coordinate origin. We could do this with no loss in generality by translating all coordinate positions in a scene so that our selected projection reference point is shifted to the coordinate origin. Or we could have initially set up the viewing-coordinate reference frame so that its origin is at the projection point that we want for a scene. And, in fact, some graphics libraries do fix the projection reference point at the coordinate origin.

Taking the projection reference point as  $(x_{prp}, y_{prp}, z_{prp}) = (0, 0, 0)$ , we obtain the elements of the required shearing matrix as

$$\mathbf{M}_{z\text{-shear}} = \begin{bmatrix} 1 & 0 & \text{sh}_{xz} & 0 \\ 0 & 1 & \text{sh}_{yz} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (30)$$

We can also simplify the elements of the perspective-projection matrix a bit more if we place the view plane at the position of the near clipping plane. And, because we now want to move the center of the clipping window to coordinates (0, 0)

on the view plane, we need to choose values for the shearing parameters such that

$$\begin{bmatrix} 0 \\ 0 \\ z_{\text{near}} \\ 1 \end{bmatrix} = \mathbf{M}_{z\text{-shear}} \cdot \begin{bmatrix} \frac{xw_{\text{min}} + xw_{\text{max}}}{2} \\ \frac{yw_{\text{min}} + yw_{\text{max}}}{2} \\ z_{\text{near}} \\ 1 \end{bmatrix} \quad (31)$$

Therefore, the parameters for this shearing transformation are

$$\begin{aligned} \text{sh}_{xz} &= -\frac{xw_{\text{min}} + xw_{\text{max}}}{2 z_{\text{near}}} \\ \text{sh}_{yz} &= -\frac{yw_{\text{min}} + yw_{\text{max}}}{2 z_{\text{near}}} \end{aligned} \quad (32)$$

Similarly, with the projection reference point at the viewing-coordinate origin and with the near clipping plane as the view plane, the perspective-projection matrix 26 is simplified to

$$\mathbf{M}_{\text{pers}} = \begin{bmatrix} -z_{\text{near}} & 0 & 0 & 0 \\ 0 & -z_{\text{near}} & 0 & 0 \\ 0 & 0 & s_x & t_x \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (33)$$

Expressions for the z-coordinate scaling and translation parameters will be determined by the normalization requirements. Concatenating the simplified perspective-projection matrix 33 with the shear matrix 30, we obtain the following oblique perspective-projection matrix for converting coordinate positions in a scene to homogeneous orthogonal projection coordinates. The projection reference point for this transformation is the viewing-coordinate origin, and the near clipping plane is the view plane.

**M**oblquepers = **M**pers · **M**z shear

$$= \begin{bmatrix} -z_{\text{near}} & 0 & \frac{xw_{\text{min}} + xw_{\text{max}}}{2} & 0 \\ 0 & -z_{\text{near}} & \frac{yw_{\text{min}} + yw_{\text{max}}}{2} & 0 \\ 0 & 0 & s_x & t_x \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (34)$$



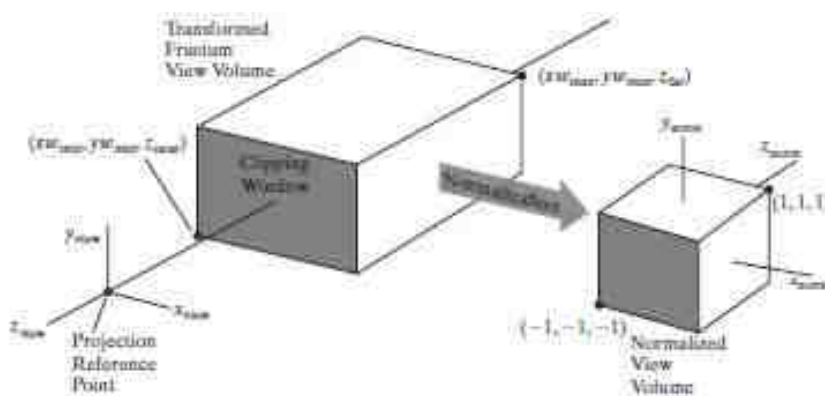
Although we no longer have options for the placement of the projection reference point and the view plane, this matrix provides an efficient method for generating a perspective-projection view of a scene without sacrificing a great deal of flexibility.

If we choose the clipping-window coordinates so that  $x_{wmax} = -x_{wmin}$  and  $y_{wmax} = -y_{wmin}$ , the frustum view volume is symmetric and matrix 34 reduces to matrix 33. This is because the projection reference point is now at the origin of the viewing-coordinate frame. We could also use Equations 29, with  $z_{prp} = 0$  and  $z_{vp} = z_{near}$ , to express the first two diagonal elements of this matrix in terms of the field-of-view angle and the clipping-window dimensions.

### Normalized Perspective-Projection Transformation Coordinates

Matrix 34 transforms object positions in viewing coordinates to perspective projection homogeneous coordinates. When we divide the homogeneous coordinates by the homogeneous parameter  $h$ , we obtain the actual projection coordinates, which are orthogonal-projection coordinates. Thus, this perspective projection transforms all points within the frustum view volume to positions within a rectangular parallelepiped view volume. The final step in the perspective transformation process is to map this parallelepiped to a *normalized view volume*.

We follow the same normalization procedure that we used for a parallel projection. The transformed frustum view volume, which is a rectangular parallelepiped, is mapped to a symmetric normalized cube within a left-handed reference frame (Figure 46). We have already included the normalization parameters for  $z$  coordinates in the perspective-projection matrix 34, but we still need to determine the values for these parameters when we transform to the symmetric normalization cube. Also, we need to determine the normalization transformation parameters for  $x$  and  $y$  coordinates. Because the centerline of the rectangular parallelepiped view volume is now the  $z_{view}$  axis, no translation is needed in the  $x$  and  $y$  normalization transformations: We require only the  $x$  and  $y$  scaling parameters relative to the coordinate origin. The scaling matrix for accomplishing the  $xy$  normalization is



**FIGURE 46**  
Normalization transformation from a transformed perspective-projection view volume (rectangular parallelepiped) to the symmetric normalization cube within a left-handed reference frame, with the near clipping plane as the view plane and the projection reference point at the viewing-coordinate origin.

$$\mathbf{M}_{xy\text{-scale}} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (35)$$

Concatenating the  $xy$ -scaling matrix with matrix 34 produces the following normalization matrix for a perspective-projection transformation:

$$\begin{aligned} \mathbf{M}_{\text{normalize}} &= \mathbf{M}_{xy\text{-scale}} \cdot \mathbf{M}_{\text{obliquepersp}} \\ &= \begin{bmatrix} -2z_{\text{near}}s_x & 0 & s_x \frac{x_{\text{Umin}} + x_{\text{Umax}}}{2} & 0 \\ 0 & -2z_{\text{near}}s_y & s_y \frac{y_{\text{Umin}} + y_{\text{Umax}}}{2} & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix} \end{aligned} \quad (36)$$

From this transformation, we obtain the homogeneous coordinates:

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix} = \mathbf{M}_{\text{normalize}} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (37)$$

And the projection coordinates are

$$\begin{aligned} x_p &= \frac{x_h}{h} = \frac{-2z_{\text{near}}s_x x + s_x(x_{\text{Umin}} + x_{\text{Umax}})/2}{-z} \\ y_p &= \frac{y_h}{h} = \frac{-2z_{\text{near}}s_y y + s_y(y_{\text{Umin}} + y_{\text{Umax}})/2}{-z} \\ z_p &= \frac{z_h}{h} = \frac{s_z z + t_z}{-z} \end{aligned} \quad (38)$$

To normalize this perspective transformation, we want the projection coordinates to be  $(x_p, y_p, z_p) = (-1, -1, -1)$  when the input coordinates are  $(x, y, z) = (x_{\text{Umin}}, y_{\text{Umin}}, z_{\text{Unear}})$ , and we want the projection coordinates to be  $(x_p, y_p, z_p) = (1, 1, 1)$  when the input coordinates are  $(x, y, z) = (x_{\text{Umax}}, y_{\text{Umax}}, z_{\text{Ufar}})$ . Therefore, when we solve equations 38 for the normalization parameters using these conditions, we obtain

$$\begin{aligned} s_x &= \frac{2}{x_{\text{Umax}} - x_{\text{Umin}}} & s_y &= \frac{2}{y_{\text{Umax}} - y_{\text{Umin}}} \\ s_z &= \frac{z_{\text{Unear}} + z_{\text{Ufar}}}{z_{\text{Unear}} - z_{\text{Ufar}}} & t_z &= \frac{2z_{\text{Unear}}z_{\text{Ufar}}}{z_{\text{Unear}} - z_{\text{Ufar}}} \end{aligned} \quad (39)$$

And the elements of the normalized transformation matrix for a generalperspective-projection are

$$\mathbf{M}_{\text{normpersp}} = \begin{bmatrix} \frac{-2z_{\text{near}}}{xw_{\text{max}} - xw_{\text{min}}} & 0 & \frac{xw_{\text{min}} + xw_{\text{max}}}{xw_{\text{max}} - xw_{\text{min}}} & 0 \\ 0 & \frac{-2z_{\text{near}}}{yw_{\text{max}} - yw_{\text{min}}} & \frac{yw_{\text{min}} + yw_{\text{max}}}{yw_{\text{max}} - yw_{\text{min}}} & 0 \\ 0 & 0 & \frac{z_{\text{near}} + z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} & \frac{2z_{\text{near}}z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (40)$$

If the perspective-projection view volume was originally specified as a symmetric frustum, we can express the elements of the normalized perspective transformation in terms of the field-of-view angle and the dimensions of the clipping window. Thus, using Equations 29, with the projection reference point at the origin and the view plane at the position of the near clipping plane, we have

$$\mathbf{M}_{\text{normsymmpersp}} = \begin{bmatrix} \frac{\cot\left(\frac{\theta}{2}\right)}{\text{aspect}} & 0 & 0 & 0 \\ 0 & \cot\left(\frac{\theta}{2}\right) & 0 & 0 \\ 0 & 0 & \frac{z_{\text{near}} + z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} & \frac{2z_{\text{near}}z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (41)$$

The complete transformation from world coordinates to normalized perspective-projection coordinates is the composite matrix formed by concatenating this perspective matrix on the left of the viewing-transformation product  $\mathbf{R} \cdot \mathbf{T}$ . Next, the clipping routines can be applied to the normalized view volume. The remaining tasks are visibility determination, surface rendering, and the transformation to the viewport.

### THREE DIMENSIONAL CLIPPING ALGORITHMS

Previously, we discussed the advantages of using the normalized boundaries of the clipping window in two-dimensional clipping algorithms. Similarly, we can apply three-dimensional clipping algorithms to the normalized boundaries of the view volume. This allows the viewing pipeline and the clipping procedures to be implemented in a highly efficient way. All device-independent transformations (geometric and viewing) are concatenated and applied before executing the clipping routines. And each of the clipping boundaries for the normalized view volume is a plane that is parallel to one of the Cartesian planes, regardless of the projection type and original shape of the view volume. Depending on whether the view volume has been normalized to a unit cube or to a symmetric cube with edge length 2, the clipping planes have coordinate positions either at 0 and 1 or at -1 and 1. For the symmetric cube, the equations for the three-dimensional clipping planes are

$$\begin{aligned}
 xw_{\text{min}} &= -1, \quad xw_{\text{max}} = 1 \\
 yw_{\text{min}} &= -1, \quad yw_{\text{max}} = 1 \quad (43) \\
 zw_{\text{min}} &= -1, \quad zw_{\text{max}} = 1
 \end{aligned}$$

The  $x$  and  $y$  clipping boundaries are the normalized limits for the clipping window, and the  $z$  clipping boundaries are the normalized positions for the near and far clipping planes.

Clipping algorithms for three-dimensional viewing identify and save all object sections within the normalized view volume for display on the output device. All parts of objects that are outside the view-volume clipping planes are eliminated. And the algorithms are now extensions of two-dimensional methods, using the normalized boundary planes of the view volume instead of the straight-line boundaries of the normalized clipping window.

### Clipping in Three-Dimensional Homogeneous Coordinates

Computer-graphics libraries process spatial positions as four-dimensional homogeneous coordinates so that all transformations can be represented as 4 by 4 matrices. As each coordinate position enters the viewing pipeline, it is converted to a four-dimensional representation:

$$(x, y, z) \rightarrow (x, y, z, 1)$$

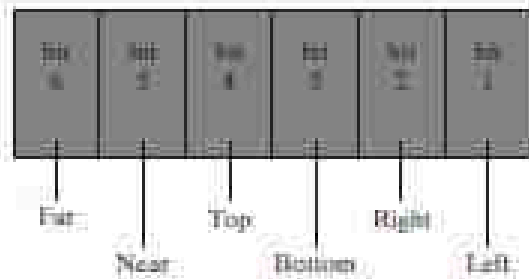
After a position has passed through the geometric, viewing, and projection transformations, it is now in the homogeneous form

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix} = \mathbf{M} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (44)$$

Where matrix  $\mathbf{M}$  represents the concatenation of all the various transformations from world coordinates to normalized, homogeneous projection coordinates, and the homogeneous parameter  $h$  may no longer have the value 1. In fact,  $h$  can have any real value, depending on how we represented objects in the scene and the type of projection we used.

If the homogeneous parameter  $h$  does have the value 1, the homogeneous coordinates are the same as the Cartesian projection coordinates. This is often the case for a parallel-projection transformation. But a perspective projection produces a homogeneous parameter that is a function of the  $z$  coordinate for any spatial position. The perspective-projection homogeneous parameter can even be negative. This occurs when coordinate positions are behind the projection reference point. Also, rational spline representations for object surfaces are often formulated in homogeneous coordinates, where the homogeneous parameter can be positive or negative. Therefore, if clipping is performed in projection coordinates after division by the homogeneous parameter  $h$ , some coordinate information can be lost and objects may not be clipped correctly.

An effective method for dealing with all possible projection transformations and object representations is to apply the clipping routines to the homogeneous coordinate representations of spatial positions. And, because all view volumes can be converted to a normalized cube, a single clipping procedure can be implemented in hardware to clip objects in homogeneous coordinates against the normalized clipping planes.



**FIGURE 49**  
A possible ordering for the view-volume clipping boundaries corresponding to the region-code bit positions.

### Three-Dimensional Region Codes

We extend the concept of a region code to three dimensions by simply adding a couple of additional bit positions to accommodate the near and far clipping planes. Thus, we now use a six-bit region code, as illustrated in Figure 49. Bit positions in this region-code example are numbered from right to left, referencing the left, right, bottom, top, near, and far clipping planes, in that order.

For a three-dimensional scene, we need to apply the clipping routines to the projection coordinates, which have been transformed to a normalized space. After the projection transformation, each point in a scene has the four-component representation  $\mathbf{P} = (x_h, y_h, z_h, h)$ . Assuming that we are clipping against the boundaries of the normalized symmetric cube (Eqs. 43), then a point is inside this normalized view volume if the projection coordinates of the point satisfy the following six inequalities:

$$-1 < \frac{x_h}{h} < 1, \quad -1 < \frac{y_h}{h} < 1, \quad -1 < \frac{z_h}{h} < 1 \quad (45)$$

Unless we have encountered an error, the value of the homogeneous parameter  $h$  is nonzero. Before implementing region-code procedures, we can first check for the possibility of a homogeneous parameter with either a zero value or an extremely small magnitude. Also, the homogeneous parameter can be either positive or negative. Therefore, assuming  $h \neq 0$ , we can write the preceding inequalities in the form

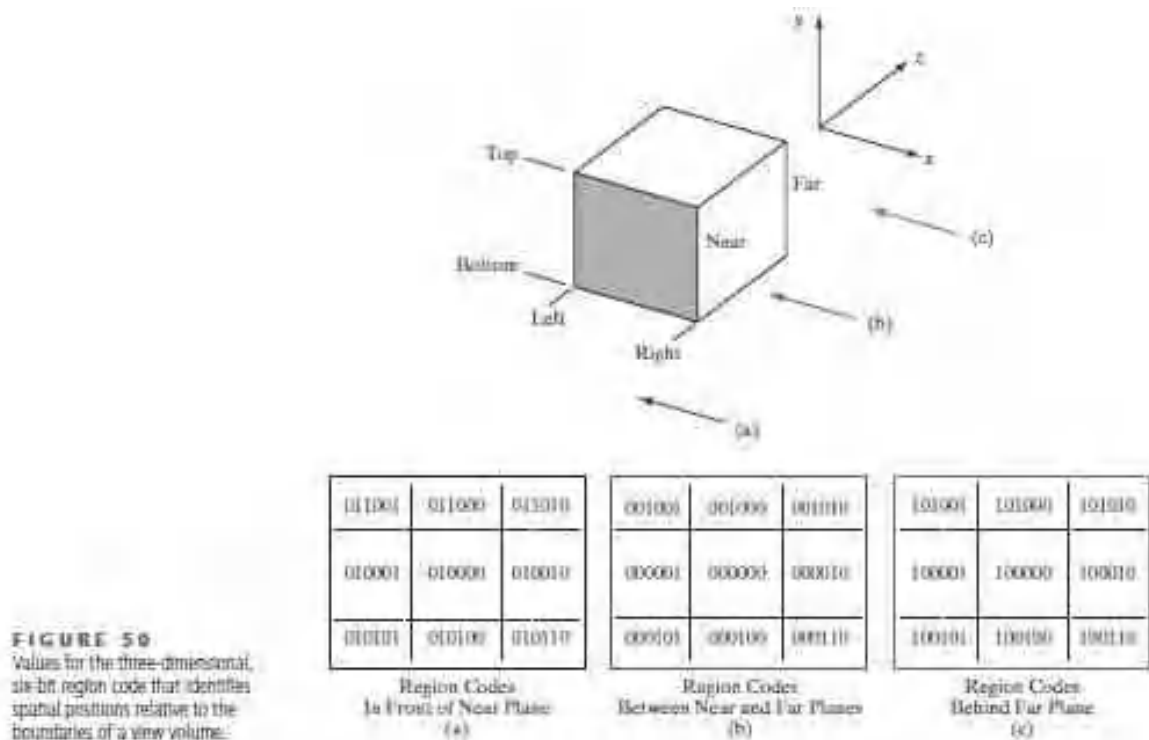
$$\begin{aligned} -h < x_h < h, & \quad -h < y_h < h, & \quad -h < z_h < h & \quad \text{if } h > 0 \\ h < x_h < -h, & \quad h < y_h < -h, & \quad h < z_h < -h & \quad \text{if } h < 0 \end{aligned} \quad (46)$$

In most cases  $h > 0$ , and we can then assign the bit values in the region code for a coordinate position according to the tests:

$$\begin{aligned} \text{bit 1} &= 1 & \text{if } h + x_h < 0 & \quad (\text{left}) \\ \text{bit 2} &= 1 & \text{if } h - x_h < 0 & \quad (\text{right}) \\ \text{bit 3} &= 1 & \text{if } h + y_h < 0 & \quad (\text{bottom}) \\ \text{bit 4} &= 1 & \text{if } h - y_h < 0 & \quad (\text{top}) \\ \text{bit 5} &= 1 & \text{if } h + z_h < 0 & \quad (\text{near}) \\ \text{bit 6} &= 1 & \text{if } h - z_h < 0 & \quad (\text{far}) \end{aligned} \quad (47)$$

These bit values can be set using the same approach as in two-dimensional clipping. That is, we simply use the sign bit of one of the calculations  $h \pm x_h$ ,  $h \pm y_h$ , or  $h \pm z_h$  to set the corresponding region-code bit value. Figure 50 lists the 27 region codes for a view volume. In those cases where  $h < 0$  for some point, we could apply clipping using the second set of inequalities in 46 or we could negate the coordinates and clip using the tests for  $h > 0$ .



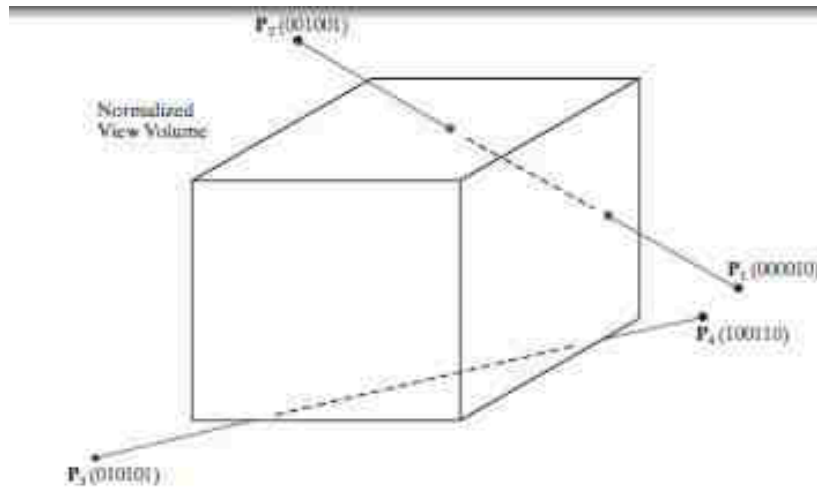


### Three-Dimensional Point and Line Clipping

For standard point positions and straight-line segments that are defined in a scene that is not behind the projection reference point, all homogeneous parameters are positive and the region codes can be established using the conditions in 47.

Then, once we have set up the region code for each position in a scene, we can easily identify a point position as outside the view volume or inside the view volume. For instance, a region code of 101000 tells us that the point is above and directly behind the view volume, while the region code 000000 indicates a point within the volume (Figure 50). Thus, for point clipping, we simply eliminate any individual point whose region code is not 000000. In other words, if any one of the tests in 47 is negative, the point is outside the view volume.

Methods for three-dimensional line clipping are essentially the same as for two-dimensional lines. We can first test the line endpoint region codes for trivial acceptance or rejection of the line. If the region code for both endpoints of a line is 000000, the line is completely inside the view volume. Equivalently, we can trivially accept the line if the logical *or* operation on the two endpoint region codes produces a value of 0. And we can trivially reject the line if the logical *and* operation on the two endpoint region codes produces a value that is not 0. This nonzero value indicates that both endpoint region codes have a 1 value in the same bit position, and hence the line is completely outside one of the clipping planes. As an example of this, the line from **P3** to **P4** in Figure 51 has the endpoint region code values of 010101 and 100110. So this line is completely below the bottom clipping plane. If a line fails these two tests, we next analyze the line equation to determine whether any part of the line should be saved.



**FIGURE 51**  
Three-dimensional region codes for two line segments. Line  $\overline{P_1P_2}$  intersects the right and top clipping boundaries of the view volume, while line  $\overline{P_3P_4}$  is completely below the bottom clipping plane.

Equations for three-dimensional line segments are conveniently expressed in parametric form, and the clipping methods of Cyrus-Bock or Liang-Barsky can be extended to three-dimensional scenes. For a line segment with endpoints  $P_1 = (x_1, y_1, z_1, h_1)$  and  $P_2 = (x_2, y_2, z_2, h_2)$ , we can write the parametric equation describing any point position along the line as

$$P = P_1 + (P_2 - P_1)u \quad 0 \leq u \leq 1 \quad (48)$$

When the line parameter has the value  $u = 0$ , we are at position  $P_1$ . And  $u = 1$  brings us to the other end of the line,  $P_2$ . Writing the parametric line equation explicitly, in terms of the homogeneous coordinates, we have

$$\begin{aligned} x &= x_1 + (x_2 - x_1)u \\ y &= y_1 + (y_2 - y_1)u \\ z &= z_1 + (z_2 - z_1)u \\ h &= h_1 + (h_2 - h_1)u \end{aligned} \quad 0 \leq u \leq 1 \quad (49)$$

Using the endpoint region codes for a line segment, we can first determine which clipping planes are intersected. If one of the endpoint region codes has a 0 value in a certain bit position while the other code has a 1 value in the same bit position, then the line crosses that clipping boundary. In other words, one of the tests in 47 generates a negative value, while the same test for the other endpoint of the line produces a nonnegative value. To find the intersection position with this clipping plane, we first use the appropriate equations in 49 to determine the corresponding value of parameter  $u$ . Then we calculate the intersection coordinates.

As an example of the intersection-calculation procedure, we consider the line segment  $\overline{P_1P_2}$  in Figure 51. This line intersects the right clipping plane, which can be described with the equation  $x_{\max} = 1$ . Therefore, we determine the intersection value for parameter  $u$  by setting the  $x$ -projection coordinate equal to 1:

$$x_p = \frac{x_1}{h} = \frac{x_1 + (x_2 - x_1)u}{h_1 + (h_2 - h_1)u} = 1 \quad (50)$$

Solving for parameter  $u$ , we obtain

$$u = \frac{x_{h1} - h_1}{(x_{h1} - h_1) - (x_{h2} - h_2)} \quad (51)$$

Next, we determine the values  $yp$  and  $zpon$  on this clipping plane, using the calculated value for  $u$ . In this case, the  $yp$  and  $zpon$  intersection values are within the  $\pm 1$  boundaries of the view volume and the line does cross into the view-volume interior.

So we next proceed to locate the intersection position with the top clipping plane. That completes the processing for this line segment, because the intersection points with the top and right clipping planes identify the part

of the line that is inside the view volume and all the line sections that are outside the view volume.

When a line intersects a clipping boundary but does not enter the view volume interior, we continue the line processing as in two-dimensional clipping.

The section of the line outside that clipping boundary is eliminated, and we update the region-code information and the values for parameter  $u$  for the part of the line inside that boundary. Then we test the remaining section of the line against the other clipping planes for possible rejection or for further intersection calculations.

Line segments in three-dimensional scenes are usually not isolated. They are most often components in the description for the solid objects in the scene, and we need to process the lines as part of the surface-clipping routines.

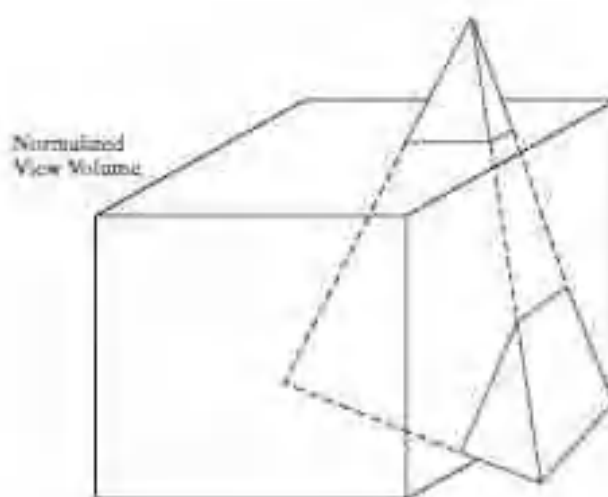
### Three-Dimensional Polygon Clipping

Graphics packages typically deal only with scenes that contain "graphics objects." These are objects whose boundaries are described with linear equations, so that each object is composed of a set of surface polygons. Therefore, to clip objects in a three-dimensional scene, we apply the clipping routines to the polygon surfaces.

Figure 52, for example, highlights the surface sections of a pyramid that are to be clipped, and the dashed lines show sections of the polygon surfaces that are inside the view volume.

We can first test a polyhedron for trivial acceptance or rejection using its coordinate extents, a bounding sphere, or some other measure of its coordinate limits. If the coordinate limits of the object are inside all clipping boundaries, we save the entire object. If the coordinate limits are all outside any one of the clipping boundaries, we eliminate the entire object.

When we cannot save or eliminate the entire object, we can next process the vertex lists for the set of polygons that define the object surfaces. Applying



**FIGURE 52**

Three-dimensional object clipping. Surface sections that are outside the view-volume clipping planes are eliminated from the object description, and new surface facets may need to be constructed.

methods similar to those in two-dimensional polygon clipping, we can clip edges to obtain new vertex lists for the object surfaces. We may also need to create some new vertex lists for additional surfaces that result from the clipping operations.

And the polygon tables are updated to add any new polygon surfaces and to revise the connectivity and shared-edge information about the surfaces.

To simplify the clipping of general polyhedra, polygon surfaces are often divided into triangular sections and described with triangle strips. We can then clip the triangle strips using the Sutherland-Hodgman approach. Each triangle strip is processed in turn against the six clipping planes to obtain the final vertex list for the strip.

For concave polygons, we can apply splitting methods to obtain a set of triangles, for example, and then clip the triangles. Alternatively, we could clip three-dimensional concave polygons using the Weiler-Atherton algorithm.

### **Three-Dimensional Curve Clipping**

As in polyhedra clipping, we first check to determine whether the coordinate extents of a curved object, such as a sphere or a spline surface, are completely inside the view volume. Then we can check to determine whether the object is completely outside any one of the six clipping planes. If the trivial rejection-acceptance tests fail, we locate the intersections with the clipping planes. To do this, we solve the simultaneous set of surface equations and the clipping-plane equation. For this reason, most graphics packages do not include clipping routines for curved objects. Instead, curved surfaces are approximated as a set of polygon patches, and the objects are then clipped using polygon-clipping routines. When surface-rendering procedures are applied to polygon patches, they can provide a highly realistic display of a curved surface.

### **Arbitrary Clipping Planes**

It is also possible, in some graphics packages, to clip a three-dimensional scene using additional planes that can be specified in any spatial orientation. This option is useful in a variety of applications. For example, we might want to isolate or clip off an irregularly shaped object, eliminate part of a scene at an oblique angle for a special effect, or slice off a section of an object along a selected axis to show a cross-sectional view of its interior.

Optional clipping planes can be specified along with the description of a scene, so that the clipping operations can be performed prior to the projection transformation.

However, this also means that the clipping routines are implemented in software.

A clipping plane can be specified with the plane parameters  $A$ ,  $B$ ,  $C$ , and  $D$ .

The plane then divides three-dimensional space into two parts, so that all parts of a scene that lie on one side of the plane are clipped off. Assuming that objects behind the plane are to be clipped, then any spatial position  $(x, y, z)$  that satisfies the following inequality is eliminated from the scene.

$$Ax + By + Cz + D < 0 \quad (52)$$

As an example, if the plane-parameter array has the values  $(A, B, C, D) = (1.0, 0.0, 0.0, 8.0)$ , then any coordinate position satisfying  $x + 8.0 < 0.0$  (or,  $x < -8.0$ ) is clipped from the scene.

To clip a line segment, we can first test its two endpoints to see if the line is completely behind the clipping plane or completely in front of the plane. We can represent inequality 52 in a vector form using the plane normal vector

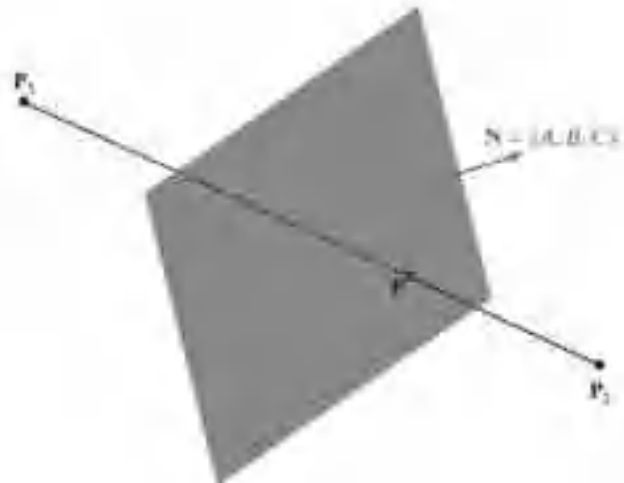


FIGURE 53  
Clipping a line segment against a plane with normal vector  $\mathbf{N}$ .

$\mathbf{N} = (A, B, C)$ .

Then, for a line segment with endpoint positions  $\mathbf{P}_1$  and  $\mathbf{P}_2$ , we clip the entire line if both endpoints satisfy

$$\mathbf{N} \cdot \mathbf{P}_k + D < 0, \quad k = 1, 2 \quad (53)$$

We save the entire line if both endpoints satisfy

$$\mathbf{N} \cdot \mathbf{P}_k + D \geq 0, \quad k = 1, 2 \quad (54)$$

Otherwise, the endpoints are on opposite sides of the clipping plane, as in Figure 53, and we calculate the line intersection point.

To calculate the line-intersection point with the clipping plane, we can use the following parametric representation for the line segment:

$$\mathbf{P} = \mathbf{P}_1 + (\mathbf{P}_2 - \mathbf{P}_1)u, \quad 0 \leq u \leq 1 \quad (55)$$

Point  $\mathbf{P}$  is on the clipping plane if it satisfies the plane equation

$$\mathbf{N} \cdot \mathbf{P} + D = 0 \quad (56)$$

Substituting the expression for  $\mathbf{P}$  from Equation 55, we have

$$\mathbf{N} \cdot [\mathbf{P}_1 + (\mathbf{P}_2 - \mathbf{P}_1)u] + D = 0 \quad (57)$$

Solving this equation for parameter  $u$ , we obtain

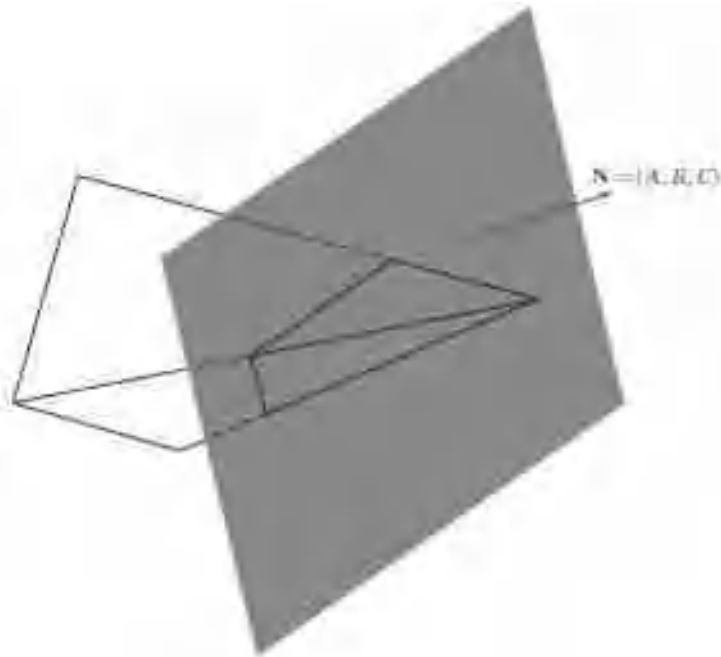
$$u = \frac{-D - \mathbf{N} \cdot \mathbf{P}_1}{\mathbf{N} \cdot (\mathbf{P}_2 - \mathbf{P}_1)} \quad (58)$$

We then substitute this value of  $u$  into the vector parametric line representation 55 to obtain values for the  $x$ ,  $y$ , and  $z$  intersection coordinates. For the example in Figure 53, the line segment from  $\mathbf{P}_1$  to  $\mathbf{P}$  is clipped and we save the section of the line from  $\mathbf{P}$  to  $\mathbf{P}_2$ .

For polyhedra, such as the pyramid in Figure 54, we apply similar clipping procedures. We first test to see if the object is completely behind or completely in front of the clipping plane. If not, we process the vertex list for each polygon surface. Line-clipping methods are applied to each polygon edge in succession, just as in view-volume clipping, to produce the surface vertex lists. But in this case, we have to deal with only one clipping plane.



Clipping a curved object against a single clipping plane is easier than clipping the object against the six planes of a view volume. However, we still need to solve a set of nonlinear equations to locate intersections, unless we approximate the curve boundaries with straight-line sections.



**FIGURE 54**  
Clipping the surfaces of a pyramid against a plane with normal vector  $\mathbf{N}$ . The surfaces in front of the plane are saved, and the surfaces of the pyramid behind the plane are eliminated.

### VISIBLE SURFACE DETECTION METHODS

A major consideration in the generation of realistic graphics displays is determining what is visible within a scene from a chosen viewing position. There are a number of approaches we can take to accomplish this, and numerous algorithms have been devised for efficient identification and display of visible objects for different types of applications. Some methods require more memory, some involve more processing time, and some apply only to special types of objects. Which method we select for a particular application can depend on such factors as the complexity of the scene, type of objects to be displayed, available equipment, and whether static or animated displays are to be generated. The various algorithms are referred to as **visible-surface detection** methods. Sometimes these methods are also referred to as **hidden-surface elimination** methods, although there can be subtle differences between identifying visible surfaces and eliminating hidden surfaces. With a wire-frame display, for example, we may not want to eliminate the hidden surfaces, but rather to display them with dashed boundaries or in some other way to retain information about their shape. They deal with the object definitions or with their projected images. These two

approaches are called **object-space** methods and **image-space** methods, respectively.

An object-space method compares objects and parts of objects to each other within the scene definition to determine which surfaces, as a whole, we should label as visible. In an image-space algorithm, visibility is decided point by point at each pixel position on the projection plane. Most visible-surface algorithms use image-space methods, although object-space methods can be used effectively to locate visible surfaces in some cases. Line-display algorithms, for instance, generally use object-space methods to identify visible lines in wire-frame displays, but many image-space visible-surface algorithms can be adapted easily to visible-line detection.

Although there are major differences in the basic approaches taken by the various visible-surface detection algorithms, most use sorting and coherence methods to improve performance. Sorting is used to facilitate depth comparisons by ordering

the individual surfaces in a scene according to their distance from the view plane. Coherence methods are used to take advantage of regularities in a scene. An individual scan line can be expected to contain intervals (runs) of constant pixel intensities, and scan-line patterns often change little from one line to the next. Animation frames contain changes only in the vicinity of moving objects. And constant relationships can often be established between the objects in a scene.

## BACKFACE DETECTION

A fast and simple object-space method for locating the **back faces** of a polyhedron is based on front-back tests. A point  $(x, y, z)$  is behind a polygon surface if

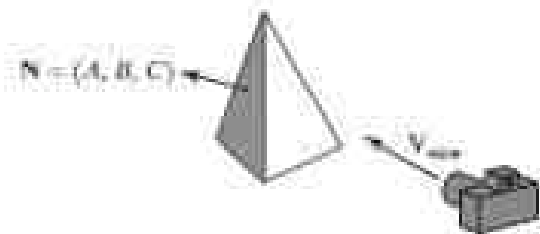
$$Ax + By + Cz + D < 0 \quad (1)$$

where  $A, B, C$ , and  $D$  are the plane parameters for the polygon. When this position is along the line of sight to the surface, we must be looking at the back of the polygon. Therefore, we could use the viewing position to test for back faces.

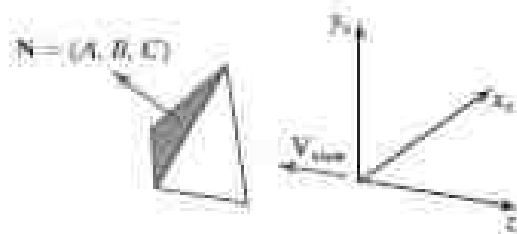
We can simplify the back-face test by considering the direction of the normal vector  $\mathbf{N}$  for a polygon surface. If  $\mathbf{V}_{\text{view}}$  is a vector in the viewing direction from our camera position, as shown in Figure 1, then a polygon is a back face if

$$\mathbf{V}_{\text{view}} \cdot \mathbf{N} > 0 \quad (2)$$

Furthermore, if object descriptions have been converted to projection coordinates and our viewing direction is parallel to the viewing  $z_v$  axis, then we need to consider only the  $z$  component of the normal vector  $\mathbf{N}$ .



**FIGURE 1**  
A surface normal vector  $\mathbf{N}$  and the viewing-direction vector  $\mathbf{V}_{\text{view}}$ .



**FIGURE 2**  
A polygon surface with plane parameter  $C < 0$  in a right-handed viewing coordinate system is identified as a back face when the viewing direction is along the negative  $z_v$  axis.

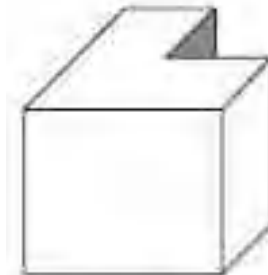
In a right-handed viewing system with the viewing direction along the negative  $z_v$  axis (Figure 2), a polygon is a back face if the  $z$  component,  $C$ , of its normal vector  $\mathbf{N}$  satisfies  $C < 0$ . Also, we cannot see any face whose normal has  $z$  component  $C = 0$ , because our viewing direction is grazing that polygon. Thus, in general, we can label any polygon as a back face if its normal vector has a  $z$  component value that satisfies the inequality  $C \leq 0$ .

Similar methods can be used in packages that employ a left-handed viewing system. In these packages, plane parameters  $A, B, C$ , and  $D$  can be calculated from polygon vertex coordinates specified in a clockwise direction (instead of the counterclockwise direction used in a right-handed system). Inequality 1 then remains a valid test for points behind the polygon. Also, back faces have normal vectors that point away from the viewing position and are identified by  $C \geq 0$  when the viewing direction is along the positive  $z_v$  axis.

By examining parameter  $C$  for the different plane surfaces describing an object, we can immediately identify all the back faces. For a single convex polyhedron, such as the pyramid in Figure 2, this test identifies all the hidden surfaces in the scene, because each surface is either completely visible or completely hidden.

Also, if a scene contains only nonoverlapping convex polyhedra, then again all hidden surfaces are identified with the back-face method.

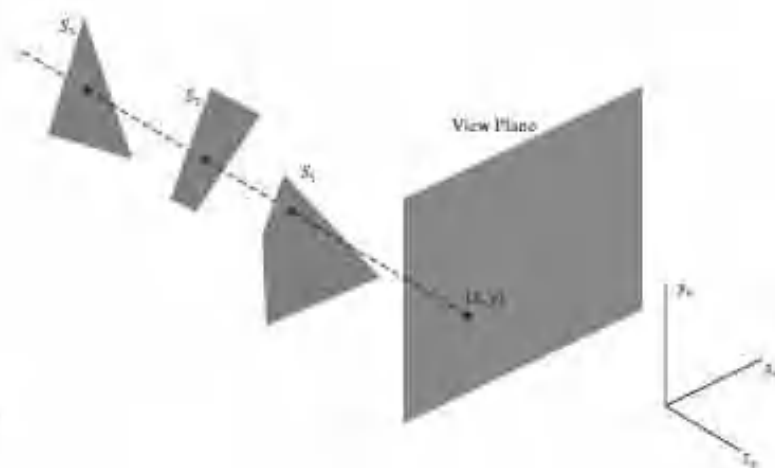
For other objects, such as the concave polyhedron in Figure 3, more tests must be carried out to determine whether there are additional faces that are totally or partially obscured by other faces. A general scene can be expected to contain overlapping objects along the line of sight, and we then need to determine where the obscured objects are partly or completely hidden by other objects. In general, back-face removal can be expected to eliminate about half of the polygon surfaces in a scene from further visibility tests.



**FIGURE 3**  
View of a concave polyhedron with one face partially hidden by other faces of the object.

### DEPTH BUFFER

A commonly used image-space approach for detecting visible surfaces is the **depth-buffer method**, which compares surface depth values throughout a scene for each pixel position on the projection plane. Each surface of a scene is processed separately, one pixel position at a time, across the surface. The algorithm is usually applied to scenes containing only polygon surfaces, because depth values can be computed very quickly and the method is easy to implement. But we could also apply the same procedures to nonplanar surfaces. This visibility-detection approach is also frequently alluded to as the *z-buffer method*, because object depth is usually measured along the  $z$  axis of a viewing system. However, rather than using actual  $z$  coordinates within the scene, depth-buffer algorithms often compute a distance from the view plane along the  $z$  axis.



**FIGURE 4**  
Three surfaces overlapping plane position  $(x, y)$  on the view plane. The visible surface,  $S_1$ , has the smallest depth value.

Figure 4 shows three surfaces at varying distances along the orthographic projection line from position  $(x, y)$  on a view plane. These surfaces can be processed in any order. As each surface is processed, its

depth from the view plane is compared to previously processed surfaces. If a surface is closer than any previously processed surfaces, its surface color is calculated and saved, along with its depth. The visible surfaces in a scene are represented by the set of surface colors that have been saved after all surface processing is completed. Implementation of the depth-buffer algorithm is typically carried out in normalized coordinates, so that depth values range from 0 at the near clipping plane (the view plane) to 1.0 at the far clipping plane.

As implied by the name of this method, two buffer areas are required. A depth buffer is used to store depth values for each  $(x, y)$  position as surfaces are processed, and the frame buffer stores the surface-color values for each pixel position. Initially, all positions in the depth buffer are set to 1.0 (maximum depth), and the frame buffer (refresh buffer) is initialized to the background color. Each surface listed in the polygon tables is then processed, one scan line at a time, by calculating the depth value at each  $(x, y)$  pixel position. This calculated depth is compared to the value previously stored in the depth buffer for that pixel position.

If the calculated depth is less than the value stored in the depth buffer, the new depth value is stored. Then the surface color at that position is computed and placed in the corresponding pixel location in the frame buffer.

The depth-buffer processing steps are summarized in the following algorithm. This algorithm assumes that depth values are normalized on the range from 0.0 to 1.0 with the view plane at depth = 0 and the farthest depth = 1. We can also apply this algorithm for any other depth range, and some graphics packages allow the user to specify the depth range over which the depth-buffer algorithm is to be applied. Within the algorithm, the variable  $z$  represents the depth of the polygon (that is, its distance from the view plane along the negative  $z$  axis).

### **Depth-Buffer Algorithm**

1. Initialize the depth buffer and frame buffer so that for all buffer positions  $(x, y)$ ,

**depthBuff  $(x, y)$  = 1.0, frameBuff  $(x, y)$  = backgndColor**

2. Process each polygon in a scene, one at a time, as follows:

- For each projected  $(x, y)$  pixel position of a polygon, calculate the depth  $z$  (if not already known).

- If  $z < \text{depthBuff } (x, y)$ , compute the surface color at that position and set

**depthBuff  $(x, y)$  =  $z$ , frameBuff  $(x, y)$  = surfColor  $(x, y)$**

After all surfaces have been processed, the depth buffer contains depth values for the visible surfaces and the frame buffer contains the corresponding color values for those surfaces.

Given the depth values for the vertex positions of any polygon in a scene, we can calculate the depth at any other point on the plane containing the polygon. At surface position  $(x, y)$ , the depth is calculated from the plane equation as

$$z = \frac{-Ax - By - D}{C} \tag{4}$$

For any scan line (Figure 5), adjacent horizontal  $x$  positions across the line differ by  $\pm 1$ , and vertical  $y$  values on adjacent scan lines differ by  $\pm 1$ . If the depth of position  $(x, y)$  has been determined to be  $z$ , then the depth  $z'$  of the next position  $(x + 1, y)$  along the scan line is obtained from Eq. 4 as

$$z' = \frac{-A(x + 1) - By - D}{C} \tag{5}$$

or

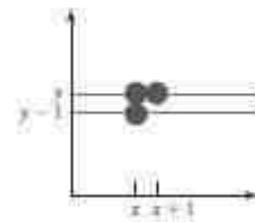
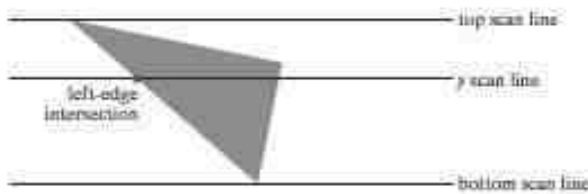
$$z' = z - \frac{A}{C} \tag{6}$$

The ratio  $-A/C$  is constant for each surface, so succeeding depth values across a scan line are obtained from preceding values with a single addition.

Processing pixel positions from left to right across each scan line, we start by calculating the depth on a left polygon edge that intersects that scan line (Figure 6). For each successive position across the scan line, we then calculate the depth value using Eq. 6.

We can implement the depth-buffer algorithm by starting at a top vertex of the polygon. Then, we could recursively calculate the  $x$ -coordinate values down a left edge of the polygon. The  $x$  value for the beginning position on each scan line can be calculated from the beginning (edge)  $x$  value of the previous scan line as

$$x' = x - \frac{1}{m}$$



**FIGURE 5**  
From position  $(x, y)$  on a scan line, the next position across the line has coordinates  $(x + 1, y)$ , and the position immediately below on the next line has coordinates  $(x, y - 1)$ .

**FIGURE 6**  
Scan lines intersecting a polygon surface.

**FIGURE 7**  
Intersection positions on successive scan lines along a left polygon edge.



where  $m$  is the slope of the edge (Figure 7). Depth values down this edge are obtained recursively as

$$z' = z + \frac{A/m + B}{C} \tag{7}$$

If we are processing down a vertical edge, the slope is infinite and the recursive calculations reduce to

$$z' = z + \frac{B}{C}$$

One slight complication with this approach is that while pixel positions are at integer  $(x, y)$  coordinates, the actual point of intersection of a scan line with the edge of a polygon may not be. As a result, it may be necessary to adjust the intersection point by rounding its fractional part up or down, as is done in scan-line polygon fill algorithms.



An alternative approach is to use a midpoint method or Bresenham-type algorithm for determining the starting  $x$  values along edges for each scan line.

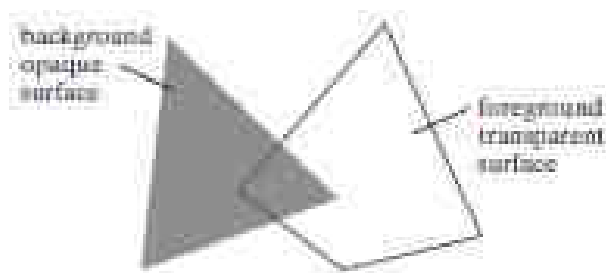
The method can be applied to curved surfaces by determining depth and color values at each surface projection point.

For polygon surfaces, the depth-buffer method is very easy to implement, and it requires no sorting of the surfaces in a scene. But it does require the availability of a second buffer in addition to the refresh buffer. A system with a resolution of  $1280 \times 1024$ , for example, would require over 1.3 million positions in the depth buffer, with each position containing enough bits to represent the number of depth increments needed. One way to reduce storage requirements is to process one section of the scene at a time, using a smaller depth buffer. After each view section is processed, the buffer is reused for the next section.

In addition, the basic depth-buffer algorithm often performs needless calculations.

Objects are processed in an arbitrary order, so that a color can be computed for a surface point that is later replaced by a closer surface. To alleviate this problem, some graphics packages provide options that allow a user to adjust the depth range for surface testing. This allows distant objects, for example, to be excluded from the depth tests. Using this option, we could even exclude objects that are very close to the projection plane. Hardware implementations of the depth-buffer algorithm are typically an integral component of sophisticated computer-graphics systems.

## A-BUFFER



**FIGURE 8**  
Viewing an opaque surface through a transparent surface requires multiple color inputs and the application of color-blending operations.

An extension of the depth-buffer ideas is the **A-buffer** procedure (at the other end of the alphabet from “z-buffer,” where  $z$  represents depth). This depth-buffer extension is an antialiasing, area-averaging, visibility-detection method developed at Lucasfilm Studios for inclusion in the surface-rendering system called REYES (an acronym for “Renders Everything You Ever Saw”). The buffer region for this procedure is referred to as the *accumulation buffer*, because it is used to store a variety of surface data, in addition to depth values.

A drawback of the depth-buffer method is that it identifies only one visible surface at each pixel position. In other words, it deals only with opaque surfaces and cannot accumulate color values for more than one surface, as is necessary if transparent surfaces are to be displayed (Figure 8). The A-buffer method expands the depth-buffer algorithm so that each position in the buffer can reference a linked list of surfaces. This allows a

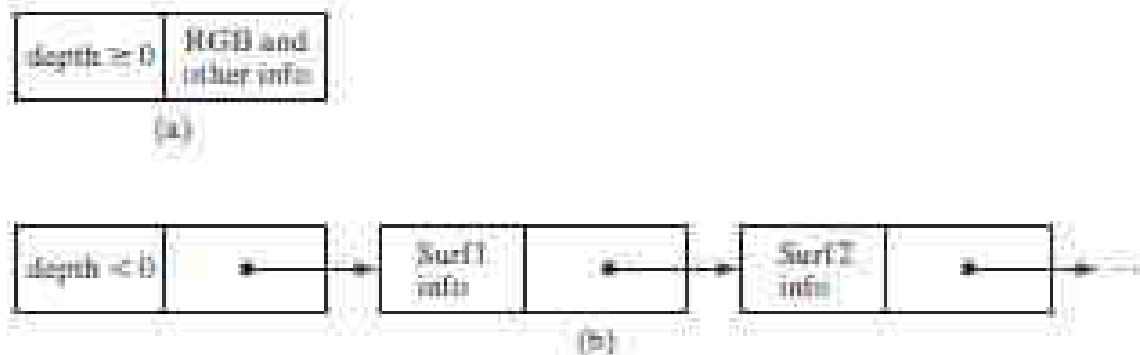
pixel color to be computed as a combination of different surface colors for transparency or antialiasing effects.

Each position in the A-buffer has two fields:

- Depth field: Stores a real-number value (positive, negative, or zero).
- Surface data field: Stores surface data or a pointer.

If the depth field is nonnegative, the number stored at that position is the depth of a surface that overlaps the corresponding pixel area. The surface data field then stores various surface information, such as the surface color for that position and the percent of pixel coverage, as illustrated in Figure 9(a). If the depth field for a position in the A-buffer is negative, this indicates multiple-surface contribution to the pixel color. The color field then stores a pointer to a linked list of surface data, as in Figure 9(b). Surface information in the A-buffer includes

- RGB intensity components
- Opacity parameter (percent of transparency)
- Depth
- Percent of area coverage
- Surface identifier
- Other surface-rendering parameters



**FIGURE 9**

Two possible organizations for surface information in an A-buffer representation for a pixel position. When a single surface overlaps the pixel, the surface depth, color, and other information are stored as in (a). When more than one surface overlaps the pixel, a linked list of surface data is stored as in (b).

The A-buffer visibility-detection scheme can be implemented using methods similar to those in the depth-buffer algorithm. Scan lines are processed to determine how much of each surface covers each pixel position across the individual scan lines. Surfaces are subdivided into a polygon mesh and clipped against the pixel boundaries. Using the opacity factors and percent of surface coverage, the rendering algorithms calculate the color for each pixel as an average of the contributions from the overlapping surfaces.

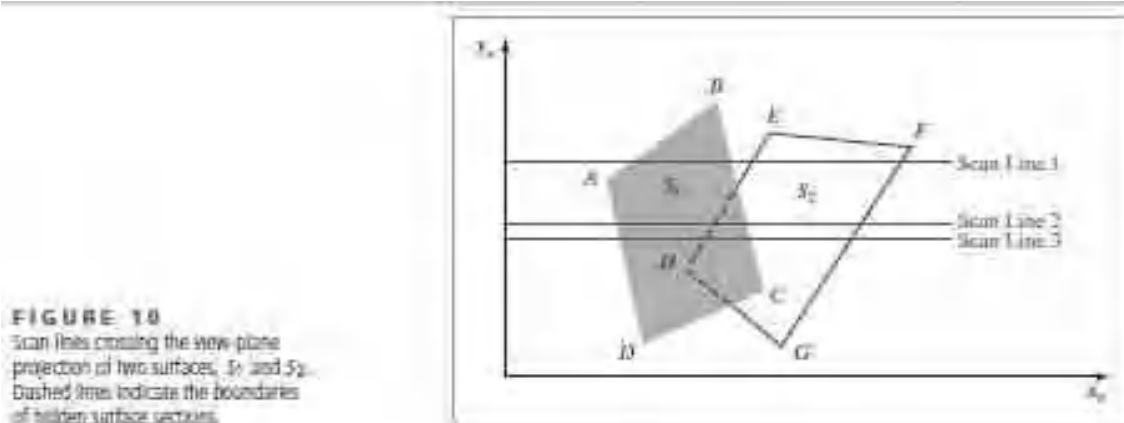
### SCAN-LINE

This image-space method for identifying visible surfaces computes and compares depth values along the various scan lines for a scene. As each scan line is processed, all polygon surface projections intersecting that line are examined to determine which are visible. Across each scan line, depth

calculations are performed to determine which surface is nearest to the view plane at each pixel position. When the visible surface has been determined for a pixel, the surface color for that position is entered into the frame buffer.

Surfaces are processed using the information stored in the polygon tables. The edge table contains coordinate endpoints for each line in the scene, the inverse slope of each line, and pointers into the surface-facet table to identify the surfaces bounded by each line. The surface-facet table contains the plane coefficients, surface material properties, other surface data, and possibly pointers into the edge table. To facilitate the search for surfaces crossing a given scan line, an active list of edges is formed for each scan line as it is processed. The active edge list contains only those edges that cross the current scan line, sorted in order of increasing  $x$ . In addition, we define a flag for each surface that is set to "on" or "off" to indicate whether a position along a scan line is inside or outside the surface. Pixel positions across each scan line are processed from left to right. At the left intersection with the surface projection of a convex polygon, the surface flag is turned on; at the right intersection point along the scan line, it is turned off. For a concave polygon, scan-line intersections can be sorted from left to right, with the surface flag set to "on" between each intersection pair.

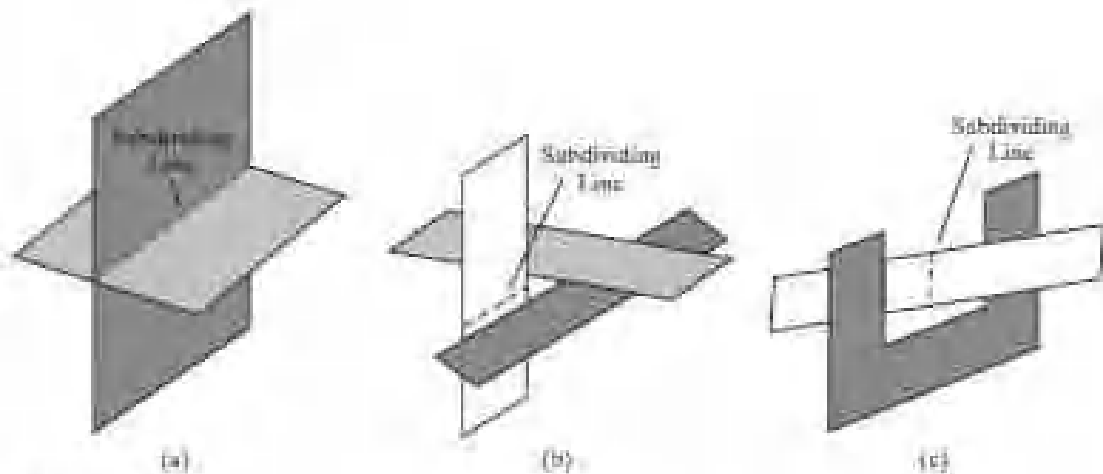
Figure 10 illustrates the scan-line method for locating visible portions of surfaces for pixel positions along a scan line. The active list for scan line 1 contains information from the edge table for edges AB, BC, EH, and FG. For positions along this scan line between edges AB and BC, only the flag for



surface  $S_1$  is on. Therefore, no depth calculations are necessary, and color values are calculated from the surface properties and lighting conditions for surface  $S_1$ . Similarly, between edges EH and FG, only the flag for surface  $S_2$  is on. No other positions along scan line 1 intersect surfaces, so the color for those pixels is the background color, which could be loaded into the frame buffer as part of the initialization routine.

For scan lines 2 and 3 in Figure 10, the active edge list contains edges AD, EH, BC, and FG. Along scan line 2 from edge AD to edge EH, only the flag for surface  $S_1$  is on. But between edges EH and BC, the flags for both surfaces are on.

Therefore, a depth calculation is necessary, using the plane coefficients for the two surfaces, when we encounter edge EH. For this example, the depth of surface  $S_1$  is assumed to be less than that of  $S_2$ , so the color values for surface  $S_1$  are assigned to the pixels across the scan line



**FIGURE 11**  
Intersecting and cyclically overlapping surfaces that alternately obscure one another.

until boundary BC is encountered. Then the surface flag for S1 goes off, and the colors for surface S2 are stored up to edge FG. No other depth calculations are necessary, because we assume that surface S2 remains behind S1 once we have determined the depth relationship at edge EH.

We can take advantage of coherence along the scan lines as we pass from one scan line to the next. In Figure 10, scan line 3 has the same active list of edges as scan line 2. No changes have occurred in line intersections, so it is again unnecessary to make depth calculations between edges EH and BC. The two surfaces must be in the same orientation as determined on scan line 2, so the colors for surface S1 can be entered without further depth calculations.

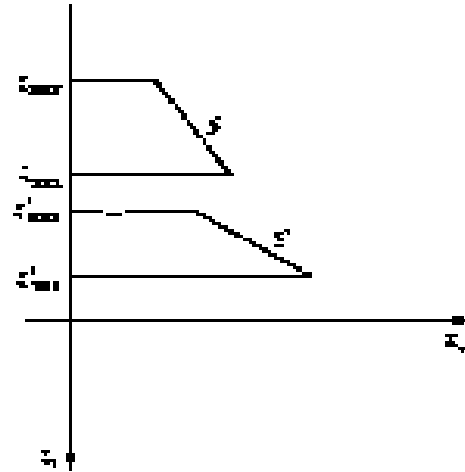
Any number of overlapping polygon surfaces can be processed with this scanline method. Flags for the surfaces are set to indicate whether a position is inside or outside, and depth calculations are performed only at the edges of overlapping surfaces. This procedure works correctly only if surfaces do not cut through or otherwise cyclically overlap each other (Figure 11). If any kind of cyclic overlap is present in a scene, we can divide the surfaces to eliminate the overlaps. The dashed lines in this figure indicate where planes could be subdivided to form two distinct surfaces, so that the cyclic overlaps are eliminated.

### **DEPTH SORTING**

Using both image-space and object-space operations, the **depth-sorting** method performs the following basic functions:

1. Surfaces are sorted in order of decreasing depth.
2. Surfaces are scan-converted in order, starting with the surface of greatest depth.

Sorting operations are carried out in both image and object space, and the scan conversion of the polygon surfaces is performed in image space. This visibility-detection method is often referred to as the **painter's algorithm**.



**FIGURE 12**  
Two surfaces with no depth overlap.

In creating an oil painting, an artist first paints the background colors. Next, the most distant objects are added, then the nearer objects, and so forth. At the final step, the foreground is painted on the canvas over the background and the more distant objects. Each color layer covers up the previous layer. Using a similar technique, we first sort surfaces according to their distance from the view plane.

The color values for the farthest surface can then be entered into the refresh buffer. Taking each succeeding surface in turn (in decreasing depth order), we “paint” the surface onto the frame buffer over the colors of the previously processed surfaces.

Painting polygon surfaces into the frame buffer according to depth is carried out in several steps. Assuming we are viewing along the  $z$  direction, surfaces are ordered on the first pass according to the smallest  $z$  value on each surface. The surface  $S$  at the end of the list (with the greatest depth) is then compared to the other surfaces in the list to determine whether there are any depth overlaps. If no depth overlaps occur,  $S$  is the most distant surface and it is scan-converted.

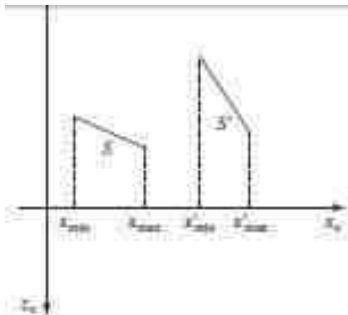
Figure 12 shows two surfaces that overlap in the  $xy$  plane but have no depth overlap. This process is then repeated for the next surface in the list. So long as no overlaps occur, each surface is processed in depth order until all have been scan-converted. If a depth overlap is detected at any point in the list, we need to make some additional comparisons to determine whether any of the surfaces should be reordered.

We make the following tests for each surface that has a depth overlap with  $S$ . If any one of these tests is true, no reordering is necessary for  $S$  and the surface being tested. The tests are listed in order of increasing difficulty:

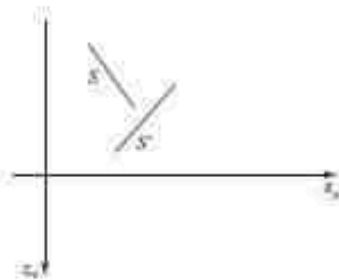
- 1.** The bounding rectangles (coordinate extents) in the  $xy$  directions for the two surfaces do not overlap.
- 2.** Surface  $S$  is completely behind the overlapping surface relative to the viewing position.
- 3.** The overlapping surface is completely in front of  $S$  relative to the viewing position.
- 4.** The boundary-edge projections of the two surfaces onto the view plane do not overlap.



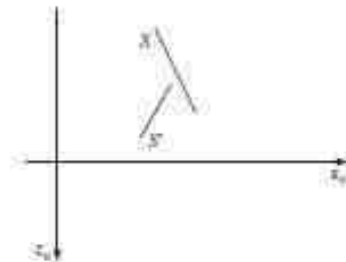
We perform these tests in the order listed and proceed to the next overlapping surface as soon as we find that one of the tests is true. If all the overlapping surfaces



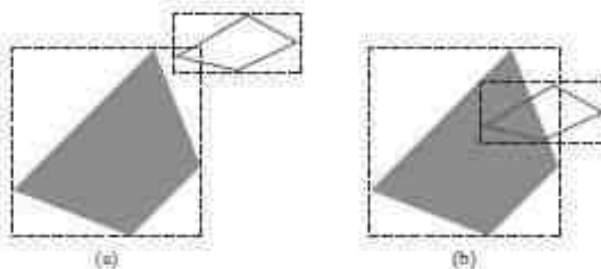
**FIGURE 13**  
Two surfaces with depth overlap but no overlap in the  $x$  direction.



**FIGURE 14**  
Surface  $S$  is completely behind the overlapping surface  $S'$ .



**FIGURE 15**  
Overlapping surface  $S'$  is completely in front of surface  $S$ , but  $S$  is not completely behind  $S'$ .



**FIGURE 16**  
Two polygon surfaces with overlapping bounding rectangles in the  $xy$  plane.

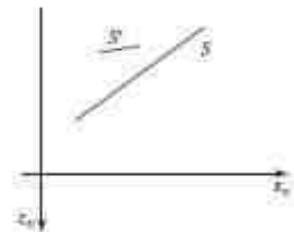
pass at least one of these tests, then  $S$  is the most distant surface. No reordering is then necessary, therefore, and  $S$  is scan-converted.

Test 1 is performed in two parts. We check for overlap first in the  $x$  direction, then in the  $y$  direction. If there is no surface overlap in either of these directions, the two planes cannot obscure one another. An example of two surfaces that overlap in the  $z$  direction but not in the  $x$  direction is shown in Figure 13.

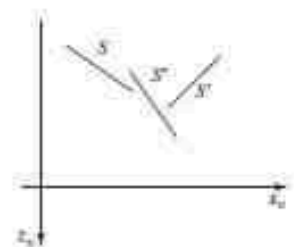
We can perform tests 2 and 3 using back-front polygon tests. That is, we substitute the coordinates for all vertices of  $S$  into the plane equation for the overlapping surface and check the sign of the result. If the plane equations are set up so that the front of the surface is toward the viewing position, then  $S$  is behind  $S'$  if all vertices of  $S$  are in back of  $S'$  (Figure 14). Similarly,  $S'$  is completely ahead of  $S$  if all vertices of  $S'$  are in front of  $S$ . Figure 15 shows an overlapping surface  $S'$  that is completely in front of  $S$ , but surface  $S$  is not completely behind  $S'$  (test 2 is not true).

If tests 1 through 3 have all failed, we perform test 4 to determine whether the two surface projections overlap. As demonstrated in Figure 16, two surfaces may or may not intersect even though their coordinate extents overlap.

Should all four tests fail for an overlapping surface  $S'$ , we interchange surfaces  $S$  and  $S'$  in the sorted list. An example of two surfaces that would be reordered with this procedure is given in Figure 17. At this point, we still do not know for certain that we have found the farthest surface from the view plane. Figure 18 illustrates a situation in which we would first interchange  $S$  and  $S'$ . However,  $S''$  obscures part of  $S'$ , so we need to interchange  $S''$  and  $S'$  to get the three surfaces



**FIGURE 17**  
Surface  $S$  extends to a greater depth, but it obscures surface  $S'$ .



**FIGURE 18**  
Three surfaces that have been entered into the sorted surface list in the order  $S, S', S''$  should be reordered as  $S', S'', S$ .

into the correct depth order. Therefore, we need to repeat the testing process for each surface that is reordered in the list.

It is possible for the algorithm just outlined to get into an infinite loop if two or more surfaces alternately obscure each other, as in Figure 11. In such situations, the algorithm would continually rearrange the ordering of

the overlapping surfaces. To avoid such loops, we can flag any surface that has been reordered to a farther depth position so that it cannot be moved again. If an attempt is made to switch the surface a second time, we divide it into two parts to eliminate the cyclic overlap. The original surface is then replaced by the two new surfaces, and we continue processing as before.

### BSP-TREE

A **binary space-partitioning** (BSP) tree is an efficient method for determining object visibility by painting surfaces into the frame buffer from back to front, as in the painter's algorithm. The BSP tree is particularly useful when the view reference point changes, but the objects in a scene are at fixed positions.

Applying a BSP tree to visibility testing involves identifying surfaces that are behind or in front of the partitioning plane at each step of the space subdivision, relative to the viewing direction. Figure 19 illustrates the basic concept in this algorithm. With plane  $P_1$ , we first partition the space into two sets of objects.

One set of objects is in back of plane  $P_1$  relative to the viewing direction, and the other set is in front of  $P_1$ . Because one object is intersected by plane  $P_1$ , we divide that object into two separate objects, labeled  $A$  and  $B$ . Objects  $A$  and  $C$  are in front of  $P_1$ , and objects  $B$  and  $D$  are behind  $P_1$ . Because each object list contains more than one object, we partition the space again with plane  $P_2$ , recursively processing the front and back object lists. This process continues until all object lists contain no more than one object. This partitioning can be easily represented using a binary tree such as the one shown in Figure 19(b). In this tree, the objects are represented as terminal nodes, with front objects occupying the left branches and back objects occupying the right branches. The location of an object in the tree exactly represents its position relative to each of the partitioning planes.

For objects described with polygon facets, we often choose the partitioning planes to coincide with polygon-surface planes. The polygon equations are then used to identify back and front polygons, and the tree is constructed with one partitioning plane for each polygon face. Any polygon intersected by a partitioning plane is split into two parts.

When the BSP tree is complete, we interpret the tree relative to the position of our viewpoint, beginning at the root node. If the viewpoint is in front of that partitioning plane, we recursively process the back subtree, then recursively process the front subtree. If the viewpoint is behind the

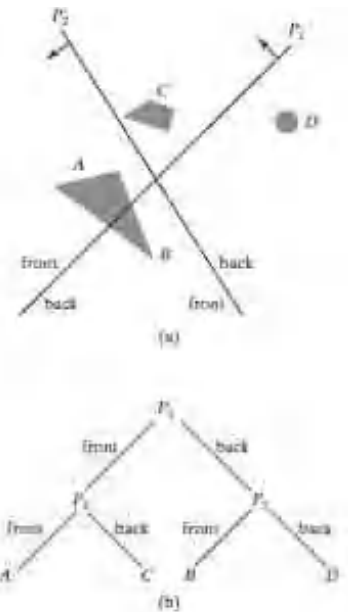


FIGURE 19  
A region of space (a) is partitioned with two planes  $P_1$  and  $P_2$  to form the BSP tree representation shown in (b).

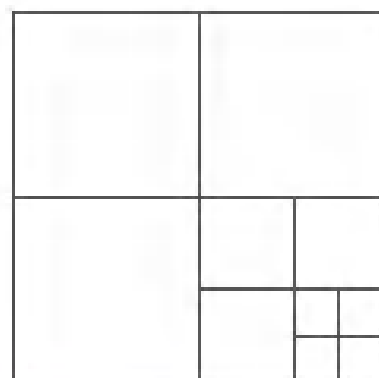
partitioning plane, wereverse this, and process the front subtree followed by the back subtree. Thus, the surfaces are generated for display in the order back to front, so that foreground objects are painted over the background objects. Fast hardware implementations for constructing and processing BSP trees are used in some systems.

### AREA SUBDIVISION

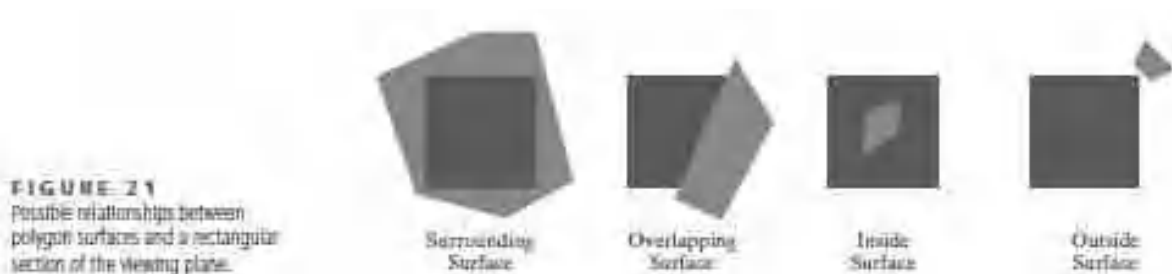
This technique for hidden-surface removal is essentially an image-space method, but object-space operations can be used to accomplish depth ordering of surfaces.

The **area-subdivision method** takes advantage of area coherence in a scene by locating those projection areas that represent part of a single surface. We apply this method by successively dividing the total view-plane area into smaller and smaller rectangles until each rectangular area contains the projection of part of a single visible surface, contains no surface projections, or the area has been reduced to the size of a pixel.

To implement this method, we need to establish tests that can quickly identify the area as part of a single surface or tell us that the area is too complex to analyze easily. Starting with the total view, we apply the tests to determine whether we should subdivide the total area into smaller rectangles. If the tests indicate that the view is sufficiently complex, we subdivide it. Next, we apply the tests to each of the smaller areas, subdividing these if the tests indicate that visibility of a single surface is still uncertain. We continue this process until the subdivisions are easily analyzed as belonging to a single surface or until we have reached the resolution limit. An easy way to do this is to successively divide the area into four equal



**FIGURE 20**  
Dividing a square area into equal-sized quadrants at each step.



**FIGURE 21**  
Possible relationships between polygon surfaces and a rectangular section of the viewing plane.

parts at each step, as shown in Figure 20. This approach is similar to that used in constructing a quadtree. A viewing area with a pixel resolution of  $1024 \times 1024$  could be subdivided ten times in this way before a subarea is reduced to the size of a single pixel.

There are four possible relationships that a surface can have with an area of the subdivided view plane. We can describe these relative surface positions using the following classifications (Figure 21).

**Surrounding Surface:** A surface that completely encloses the area.

**Overlapping Surface:** A surface that is partly inside and partly outside the area.

**Inside Surface:** A surface that is completely inside the area.

**Outside Surface:** A surface that is completely outside the area. The tests for determining surface visibility within a rectangular area can be stated in terms of the four surface classifications illustrated in Figure 21. No further subdivisions of a specified area are needed if one of the following conditions is true.

**Condition 1:** An area has no inside, overlapping, or surrounding surfaces (all surfaces are outside the area).

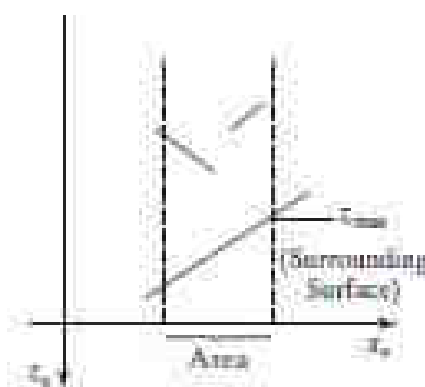
**Condition 2:** An area has only one inside, overlapping, or surrounding surface.

**Condition 3:** An area has one surrounding surface that obscures all other surfaces within the area boundaries.

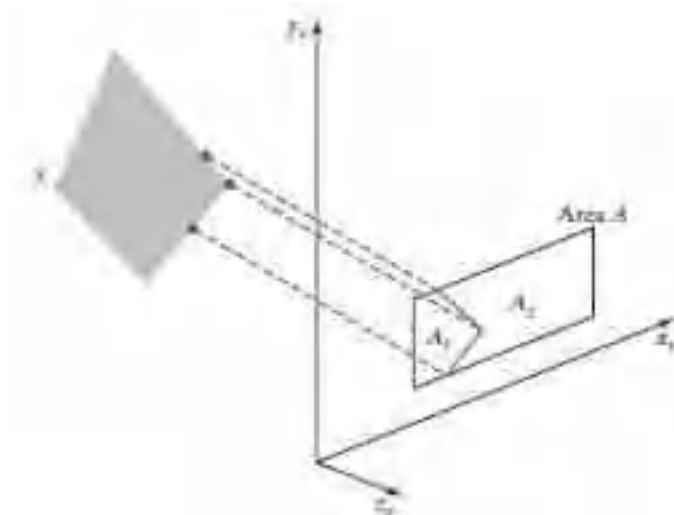
Initially, we can compare the coordinate extents of each surface with the area boundaries. This will identify the inside and surrounding surfaces, but overlapping and outside surfaces usually require intersection tests. If a single bounding rectangle intersects the area in some way, additional checks are used to determine whether the surface is surrounding, overlapping, or outside. Once a single inside, overlapping, or surrounding surface has been identified, the surface color values are stored in the frame buffer.

One method for testing condition 3 is to sort the surfaces according to minimum depth from the view plane. For each surrounding surface, we then compute the maximum depth within the area under consideration. If the maximum depth of one of these surrounding surfaces is closer to the view plane than the minimum depth of all other surfaces within the area, condition 3 is satisfied. Figure 22 illustrates this situation. Another method for testing condition 3 that does not require depth sorting is to use plane equations to calculate depth values at the four vertices of the area for all surrounding, overlapping, and inside surfaces. If all four depths for one of the surrounding surfaces are less than the calculated depths for all other surfaces, condition 3 is satisfied. Then the area can be displayed with the colors for that surrounding surface.

For some situations, the previous two testing methods may fail to identify correctly a surrounding surface that obscures all the other surfaces. Further testing could be carried out to identify the single surface that covers the area, but it is faster to subdivide the area than to continue with more complex testing. Once a surface has been identified as an outside or surrounding surface for an area, it will remain in that category for all subdivisions of the area. Furthermore, we can expect to eliminate some inside and overlapping surfaces as the subdivision process continues, so that the



**FIGURE 22**  
Within a specified area, a surrounding surface with a maximum depth of  $z_{max}$  obscures all surfaces that have a minimum depth beyond  $z_{max}$ .



**FIGURE 23**  
Area A is subdivided into  $A_1$  and  $A_2$  using the boundary of surface S on the view plane.

areas become easier to analyze. In the limiting case, when a subdivision the size of a pixel is produced, we simply calculate the depth of each relevant surface at that point and assign the color of the nearest surface to that pixel.

As a variation on the basic subdivision process, we could subdivide areas along surface boundaries instead of dividing them in half. If the surfaces have been sorted according to minimum depth, we can use the surface of smallest depth value to subdivide a given area. Figure 23 illustrates this method for subdividing areas. The projection of the boundary of surface  $S$  is used to partition the original area into the subdivisions  $A_1$  and  $A_2$ . Surface  $S$  is then a surrounding surface for  $A_1$ , and visibility conditions 2 and 3 can be tested to determine whether further subdividing is necessary. In general, fewer subdivisions are required using this approach, but more processing is needed to subdivide areas and to analyze the relation of surfaces to the subdivision boundaries.

### **OCTREE METHOD**



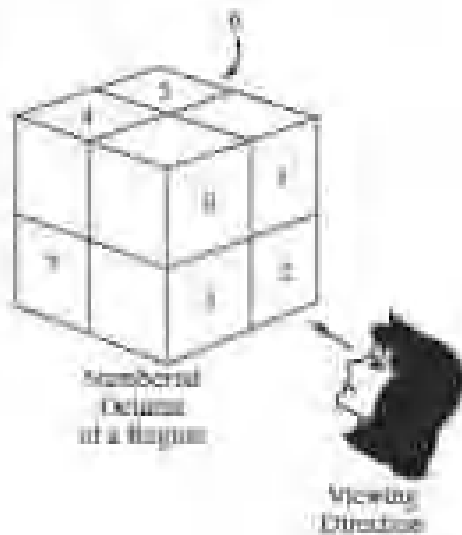


FIGURE 24  
Objects in octants 0, 1, 2, and 3 obscure objects in the back octants (4, 5, 6, 7) when the viewing direction is as shown.

When an octree representation is used for the viewing volume, visible-surface identification is accomplished by searching octree nodes in a front-to-back order.

In Figure 24, the foreground of a scene is contained in octants 0, 1, 2, and 3.

Surfaces in the front of these octants are visible to the viewer. Any surfaces toward the rear of the front octants or in the back octants (4, 5, 6, and 7) may be hidden by the front surfaces.

We can process the octree nodes of Figure 24 in the order 0, 1, 2, 3, 4, 5, 6, 7. This results in a depth-first traversal of the octree, where the nodes for the four front suboctants of octant 0 are visited before the nodes for the four back suboctants. The traversal of the octree continues in this order for each octant subdivision. When a color value is encountered in an octree node, that color is saved in the quadtree only if no values have previously been saved for the same area. In this way, only the front colors are saved. Nodes that have the value "void" are ignored.

Any node that is completely obscured is eliminated from further processing, so that its subtrees are not accessed. Figure 25 depicts the octants in a region of space and the corresponding quadrants on the view plane. Contributions to quadrant 0 come from octants 0 and 4. Color values in quadrant 1 are obtained from surfaces in octants 1 and 5, and values in each of the other two quadrants are generated from the pairs of octants aligned with each of these quadrants.

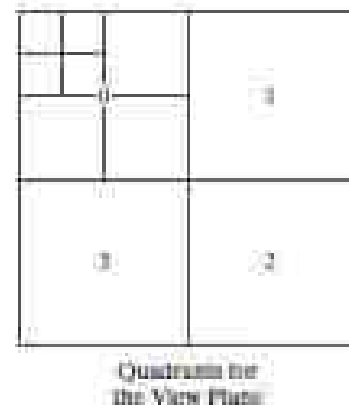
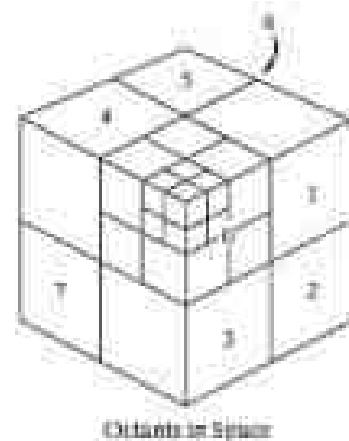


FIGURE 25  
Octant divisions for a region of space and the corresponding quadrant plane.

Effective octree visibility testing is carried out with recursive processing of octree nodes and the creation of a quadtree representation for the visible surfaces.

In most cases, both a front and a back octant must be considered in determining the correct color values for a quadrant. But if the front octant is homogeneously filled with some color, we do not process the back octant. For heterogeneous regions, a recursive procedure is called, passing as new arguments the child of the heterogeneous octant and a newly created quadtree node. If the front is empty, it is necessary only to process the child of the rear octant. Otherwise, two recursive calls are made: one for the rear octant and one for the front octant.

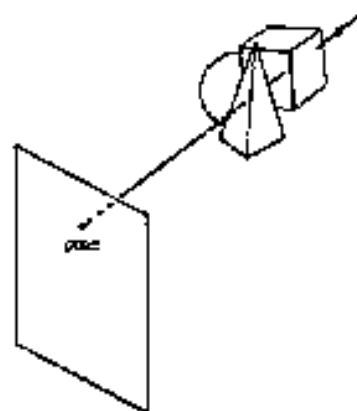
Different views of objects represented as octrees can be obtained by applying transformations to the octree representation that reorient the object according to the view selected. Octants can then be renumbered so that the octree representation is always organized with octants 0, 1, 2, and 3 as the front face.

## OTHER METHODS.

### Ray-Casting Method

If we consider the line of sight from a pixel position on the view plane through a scene, as in Figure 26, we can determine which objects in the scene (if any) intersect this line. After calculating all ray-surface intersections, we identify the visible surface as the one

FIGURE 26  
A ray along the line of sight from a pixel position through a scene.



whose intersection point is closest to the pixel. This visibility-detection scheme uses *ray casting* procedures. Ray casting, as a visibility-detection tool, is based on geometric optics methods, which trace the paths of light rays. Because there are an infinite number of light rays in a scene and we are interested only in those rays that pass through pixel positions, we can trace the light-ray paths backward from the pixels through the scene. The ray-casting approach is an effective visibility-detection method for scenes with curved surfaces, particularly spheres. We can think of ray casting as a variation on the depth-buffer method (Section 3). In the depth-buffer algorithm, we process surfaces one at a time and calculate depth values for all projection points over the surface. The calculated surface depths are then compared to previously stored depths to determine visible surfaces at each pixel. In ray casting, we process pixels one at a time and calculate depths for all surfaces along the projection path to that pixel. Ray casting is a special case of *ray-tracing* algorithms that trace multiple ray paths to pick up global reflection and refraction contributions from multiple objects in a scene. With ray casting, we only follow a ray out from each pixel to the nearest object.

Efficient ray-surface intersection calculations have been developed for common objects, particularly spheres.

## **UNIT 5 :** **COMPUTER ANIMATION**

Computer-graphics methods are now commonly used to produce animations for a variety of applications, including entertainment (motion pictures and cartoons), advertising, scientific and engineering studies, and training and education. Although we tend to think of animation as implying object motion, the term **computer animation** generally refers to any time sequence of visual changes in a picture. In addition to changing object positions using translations or rotations, a computer-generated animation could display time variations in object size, color, transparency, or surface texture. Advertising animations often transition one object shape into another: for example, transforming a can of motor oil into an automobile engine. We can also generate computer animations by varying camera parameters, such as position, orientation, or focal length, and variations in lighting effects or other parameters and procedures associated with illumination and rendering can be used to produce computer animations.

Another consideration in computer-generated animation is realism. Many applications require realistic displays. An accurate representation of the shape of a thunderstorm or other natural phenomena described with a numerical model is important for evaluating the reliability of the model. Similarly, simulators for training aircraft pilots and heavy-equipment operators must produce reasonably accurate representations of the environment. Entertainment and advertising applications, on the other hand, are sometimes more interested in visual effects. Thus, scenes may be displayed with exaggerated shapes and unrealistic motions and transformations.

However, there are many entertainment and advertising applications that do require accurate representations for computer-generated scenes. Also, in some scientific and engineering studies, realism is not a goal. For example, physical quantities are often displayed with pseudo-colors or abstract shapes that change over time to help the researcher understand the nature of the physical process.

Two basic methods for constructing a motion sequence are **real-time animation** and **frame-by-frame animation**. In a real-time computer animation, each stage of the sequence is viewed as it is created. Thus the animation must be generated at a rate that is compatible with the constraints of the refresh rate. For a frame-by-frame animation, each frame of the motion is separately generated and stored. Later, the frames can be recorded on film, or they can be displayed consecutively on a video monitor in “real-time playback” mode. Simple animation displays are generally produced in real time, while more complex animations are constructed more slowly, frame by frame. However, some applications require real-time animation, regardless of the complexity of the animation.

A flight-simulator animation, for example, is produced in real time because the video displays must be generated in immediate response to changes in the control settings.

In such cases, special hardware and software systems are often developed to allow the complex display sequences to be developed quickly.

### **Raster Methods for Computer Animation**

Most of the time, we can create simple animation sequences in our programs using real-time methods. In general, though, we can produce an animation sequence on a raster-scan system one frame at a time, so that each completed frame could be saved in a file for later viewing. The animation can then be viewed by cycling through the completed frame sequence, or the frames could be transferred to film.

If we want to generate an animation in real time, however, we need to produce the motion frames quickly enough so that a continuous motion sequence is displayed.

For a complex scene, one frame of the animation could take most of the refresh cycle time to construct. In that case, objects generated first would be displayed for most of the frame refresh time, but objects generated toward the end of the refresh cycle would disappear almost as soon as they were displayed. For very complex animations, the frame construction time could be greater than the time to refresh the screen, which can lead to erratic motion and fractured frame displays.

Because the screen display is generated from successively modified pixel values in the refresh buffer, we can take advantage of some of the characteristics of the raster screen-refresh process to produce motion sequences quickly.

### **Double Buffering**

One method for producing a real-time animation with a raster system is to employ two refresh buffers. Initially, we create a frame for the animation in one of the buffers. Then, while the screen is being refreshed from that buffer, we construct the next frame in the other buffer. When that frame is complete, we switch the roles of the two buffers so that the refresh routines use the second buffer during the process of creating the next frame in the first buffer. This alternating buffer process continues throughout the animation. Graphics libraries that permit such operations typically have one function for activating the double buffering routines and another function for interchanging the roles of the two buffers.

When a call is made to switch two refresh buffers, the interchange could be performed at various times. The most straightforward implementation is to switch the two buffers at the end of the current refresh cycle, during the vertical retrace of the electron beam. If a program can complete the construction of a frame within the time of a refresh cycle, say 1

60 of a second, each motion sequence is displayed in synchronization with the screen refresh rate. However, if the time to construct a frame is longer than the refresh time, the current frame is displayed for two or more refresh cycles while the next animation frame is being generated. For example, if the screen refresh rate is 60 frames per second and it takes  $1/50$  of a second to construct an animation frame, each frame is displayed on the screen twice and the animation rate is only 30 frames each second. Similarly, if the frame construction time is  $1/25$  of a second, the animation frame rate is reduced to 20 frames per second because each frame is displayed three times.

Irregular animation frame rates can occur with double buffering when the frame construction time is very nearly equal to an integer multiple of the screen refresh time. As an example of this, if the screen refresh rate is 60 frames per second, then an erratic animation frame rate is possible when the frame construction time is very close to 160 of a second, or 260 of a second, or 360 of a second, and so forth. Because of slight variations in the implementation time for the routines that generate the primitives and their attributes, some frames could take a little more time to construct and some a little less time. Thus, the animation frame rate can change abruptly and erratically. One way to compensate for this effect is to add a small time delay to the program. Another possibility is to alter the motion or scene description to shorten the frame construction time.

### Generating Animations Using Raster Operations

We can also generate real-time raster animations for limited applications using block transfers of a rectangular array of pixel values. This animation technique is often used in game-playing programs. A simple method for translating an object from one location to another in the  $xy$  plane is to transfer the group of pixel values that define the shape of the object to the new location.

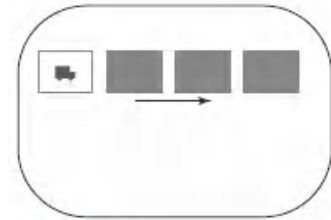


FIGURE 1  
Real-time raster color-table animation.

Two-dimensional rotations in multiples of  $90^\circ$  are also simple to perform, although we can rotate rectangular blocks of pixels through other angles using antialiasing procedures. For a rotation that is not a multiple of  $90^\circ$ , we need to estimate the percentage of area coverage for those pixels that overlap the rotated block. Sequences of raster operations can be executed to produce real-time animation for either two-dimensional or three-dimensional objects, so long as we restrict the animation to motions in the projection plane. Then no viewing or visible-surface algorithms need be invoked.

We can also animate objects along two-dimensional motion paths using **color table transformations**. Here we predefine the object at successive positions along the motion path and set the successive blocks of pixel values to color-table entries.

The pixels at the first position of the object are set to a foreground color, and the pixels at the other object positions are set to the background color. The animation is then accomplished by changing the color-table values so that the object color at successive positions along the animation path becomes the foreground color as the preceding position is set to the background color (Figure 1).

### Design of Animation Sequences

Constructing an animation sequence can be a complicated task, particularly when it involves a story line and multiple objects, each of which can move in a different way. A basic approach is to design such animation sequences using the following development stages:

- Storyboard layout
- Object definitions
- Key-frame specifications
- Generation of in-between frames



The **storyboard** is an outline of the action. It defines the motion sequence as a set of basic events that are to take place. Depending on the type of animation to be produced, the storyboard could consist of a set of rough sketches, along with a brief description of the movements, or it could just be a list of the basic ideas for the action. Originally, the set of motion sketches was attached to a large board that was used to present an overall view of the animation project. Hence, the name “storyboard.”

An **object definition** is given for each participant in the action. Objects can be defined in terms of basic shapes, such as polygons or spline surfaces. In addition, a description is often given of the movements that are to be performed by each character or object in the story.

A **key frame** is a detailed drawing of the scene at a certain time in the animation sequence. Within each key frame, each object (or character) is positioned according to the time for that frame. Some key frames are chosen at extreme positions in the action; others are spaced so that the time interval between keyframes is not too great. More key frames are specified for intricate motions than for simple, slowly varying motions. Development of the key frames is generally the responsibility of the senior animators, and often a separate animator is assigned to each character in the animation.

**In-betweens** are the intermediate frames between the key frames. The total number of frames, and hence the total number of in-betweens, needed for an animation is determined by the display media that is to be used. Film requires 24 frames per second, and graphics terminals are refreshed at the rate of 60 or more frames per second. Typically, time intervals for the motion are set up so that there are from three to five in-betweens for each pair of key frames. Depending on the speed specified for the motion, some key frames could be duplicated. As an example, a 1-minute film sequence with no duplication requires a total of 1,440 frames. If five in-betweens are required for each pair of key frames, then 288 key frames would need to be developed.

There are several other tasks that may be required, depending on the application.

These additional tasks include motion verification, editing, and the production and synchronization of a soundtrack. Many of the functions needed to produce general animations are now computer-generated. Figures 2 and 3 show examples of computer-generated frames for animation sequences.



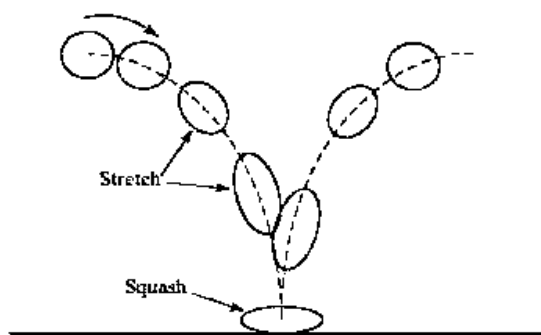
**FIGURE 2**  
One frame from the award-winning computer animated short film *Luxo Jr.* The film was designed using a key-frame animation system and cartoon animation techniques to provide lifelike actions of the lamps. Final images were rendered with multiple light sources and procedural texturing techniques. (Courtesy of Pixar. © 1986 Pixar.)



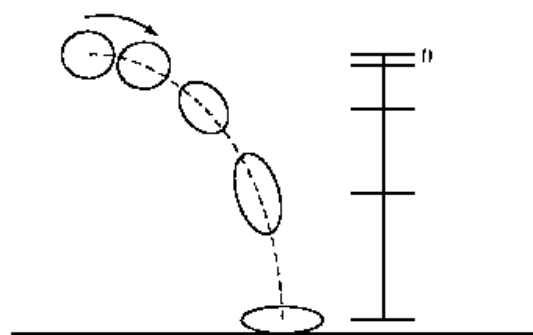
**FIGURE 3**  
One frame from the short film *Tin Toy*, the first computer-animated film to win an Oscar. Designed using a key-frame animation system, the film also required extensive facial-expression modeling. Final images were rendered using procedural shading, soft shadowing techniques, motion blur, and texture mapping. (Courtesy of Pixar. © 1988 Pixar.)

## Traditional Animation Techniques

Film animators use a variety of methods for depicting and emphasizing motion sequences. These include object deformations, spacing between animation frames, motion anticipation and follow-through, and action focusing.



**FIGURE 4**  
A bouncing-ball illustration of the "squash and stretch" technique for emphasizing object acceleration.



**FIGURE 5**  
The position changes between motion frames for a bouncing ball increase as the speed of the ball increases.

One of the most important techniques for simulating acceleration effects, particularly for nonrigid objects, is **squash and stretch**. Figure 4 shows how this technique is used to emphasize the acceleration and deceleration of a bouncing ball. As the ball accelerates, it begins to stretch. When the ball hits the floor and stops, it is first compressed (squashed) and then stretched again as it accelerates and bounces upwards.

Another technique used by film animators is **timing**, which refers to the spacing between motion frames. A slower moving object is represented with more closely spaced frames, and a faster moving object is displayed with fewer frames over the path of the motion. This effect is illustrated in Figure 5, where the position changes between frames increase as a bouncing ball moves faster.

Object movements can also be emphasized by creating preliminary actions that indicate an **anticipation** of a coming motion. For example, a cartoon character might lean forward and rotate its body before starting to run; or a character might perform a "windup" before throwing a ball. Similarly, **follow-through actions** can be used to emphasize a previous motion. After throwing a ball, a character can continue the arm swing back to its body; or a hat can fly off a character that is stopped abruptly. An action also can be emphasized with **staging**, which refers to any method for focusing on an important part of a scene, such as a character hiding something.

### **General Computer-Animation Functions**

Many software packages have been developed either for general animation design or for performing specialized animation tasks. Typical animation functions include managing object motions, generating views of objects, producing camera motions, and the generation of in-between frames. Some animation packages, such as Wavefront for example, provide special functions for both the overall animation design and the processing of individual objects. Others are special-purpose packages for particular features of an animation, such as a system for generating in-between frames or a system for figure animation.

A set of routines is often provided in a general animation package for storing and managing the object database. Object shapes and associated parameters are stored and updated in the database. Other object functions include those for generating the object motion and those for rendering the object surfaces.

Movements can be generated according to specified constraints using two-dimensional or three-dimensional transformations. Standard functions can then be applied to identify visible surfaces and apply the rendering algorithms.

Another typical function set simulates camera movements. Standard camera motions are zooming, panning, and tilting. Finally, given the specification for the key frames, the in-betweens can be generated automatically.

### Computer-Animation Languages

We can develop routines to design and control animation sequences within a general-purpose programming language, such as C, C++, Lisp, or Fortran, but several specialized animation languages have been developed. These languages typically include a graphics editor, a key-frame generator, an in-between generator, and standard graphics routines. The graphics editor allows an animator to design and modify object shapes, using spline surfaces, constructive solid geometry methods, or other representation schemes.

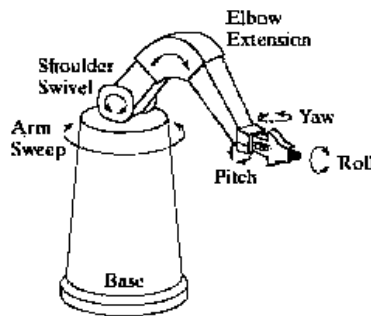


FIGURE 6  
Degrees of freedom for a stationary, single-armed robot.

An important task in an animation specification is *scene description*. This includes the positioning of objects and light sources, defining the photometric parameters (light-source intensities and surface illumination properties), and setting the camera parameters (position, orientation, and lens characteristics).

Another standard function is *action specification*, which involves the layout of motion paths for the objects and camera. We need the usual graphics routines:

viewing and perspective transformations, geometric transformations to generate object movements as a function of accelerations or kinematic path specifications, visible-surface identification, and the surface-rendering operations.

**Key-frame systems** were originally designed as a separate set of animation routines for generating the in-betweens from the user-specified key frames. Now, these routines are often a component in a more general animation package. In the simplest case, each object in a scene is defined as a set of rigid bodies connected at the joints and with a limited

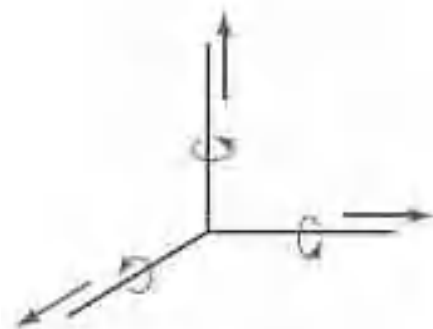


FIGURE 7  
Translational and rotational degrees of freedom for the base of the robot arm.

number of degrees of freedom. As an example, the single-armed robot in Figure 6 has 6 degrees of freedom, which are referred to as arm sweep, shoulder swivel, elbow extension, pitch, yaw, and roll. We can extend the number of degrees of freedom for this robot arm to 9 by allowing three-dimensional translations for the base (Figure 7). If we also allow base rotations, the robot arm can have a total of 12 degrees of freedom. The human body, in comparison, has more than 200 degrees of freedom.

**Parameterized systems** allow object motion characteristics to be specified as part of the object definitions. The adjustable parameters control such object characteristics as degrees of freedom, motion limitations, and allowable shape changes.

**Scripting systems** allow object specifications and animation sequences to be defined with a user-input *script*. From the script, a library of various objects and motions can be constructed.

### Key-Frame Systems

A set of in-betweens can be generated from the specification of two (or more) key frames using a key-frame system. Motion paths can be given with a *kinematic description* as a set of spline curves, or the motions can be *physically based* by specifying the forces acting on the objects to be animated.

For complex scenes, we can separate the frames into individual components or objects called **cels** (celluloid transparencies). This term developed from cartoon animation techniques where the background and each character in a scene were placed on a separate transparency. Then, with the transparencies stacked in the order from background to foreground, they were photographed to obtain the completed frame. The specified animation paths are then used to obtain the next cell for each character, where the positions are interpolated from the key-frame times. With complex object transformations, the shapes of objects may change over time. Examples are clothes, facial features, magnified detail, evolving shapes, and exploding or disintegrating objects. For surfaces described with polygon meshes, these changes can result in significant changes in polygon shape such that the number of edges in a polygon could be different from one frame to the next.

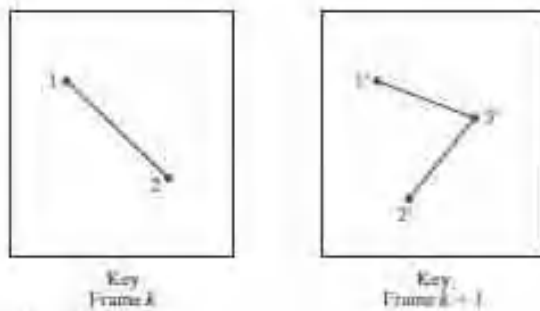
These changes are incorporated into the development of the in-between frames by adding or subtracting polygon edges according to the requirements of the defining key frames.

### Morphing

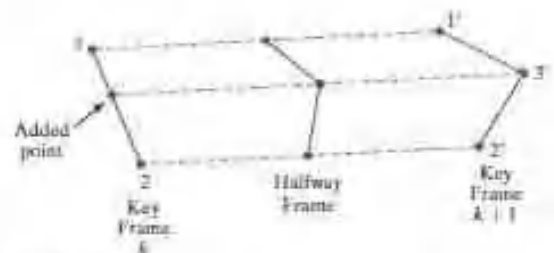
Transformation of object shapes from one form to another is termed **morphing**, which is a shortened form of “metamorphosing.” An animator can model morphing by transitioning polygon shapes through the in-betweens from one key frame to the next.

Given two key frames, each with a different number of line segments specifying an object transformation, we can first adjust the object

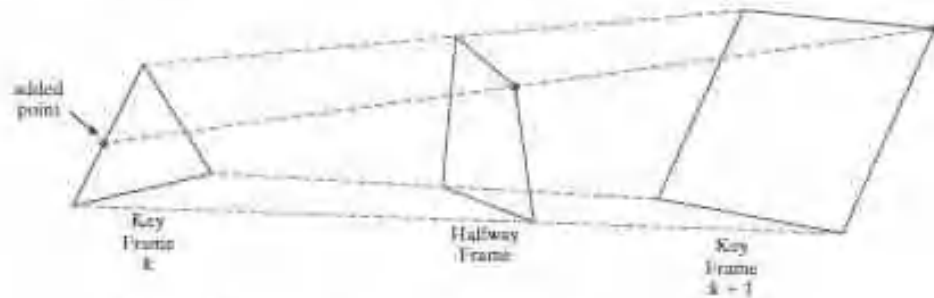
$$N_c = \text{int} \left( \frac{L_{\text{max}}}{L_{\text{min}}} \right) \quad (7)$$



**FIGURE 8**  
An edge with vertex positions 1 and 2 in key frame  $k$  evolves into two connected edges in key frame  $k + 1$ .



**FIGURE 9**  
Linear interpolation for transforming a line segment in key frame  $k$  into two connected line segments in key frame  $k + 1$ .



**FIGURE 10**  
Linear interpolation for transforming a triangle into a quadrilateral.

specification in one of the frames so that the number of polygon edges (or the number of polygon vertices) is the same for the two frames. This preprocessing step is illustrated in Figure 8. A straight-line segment in key frame  $k$  is transformed into two line segments in key frame  $k + 1$ . Because key frame  $k + 1$  has an extra vertex, we add a vertex between vertices 1 and 2 in key frame  $k$  to balance the number of vertices (and edges) in the two key frames. Using linear interpolation to generate the in-betweens, we transition the added vertex in key frame  $k$  into vertex 3\_ along the straight-line path shown in Figure 9. An example of a triangle linearly expanding into a quadrilateral is given in Figure 10.

We can state general preprocessing rules for equalizing key frames in terms of either the number of edges or the number of vertices to be added to a key frame. We first consider equalizing the edge count, where parameters  $L_k$



and  $L_{k+1}$  denote the number of line segments in two consecutive frames. The maximum and minimum number of lines to be equalized can be determined as  $L_{\max} = \max(L_k, L_{k+1})$ ,  $L_{\min} = \min(L_k, L_{k+1})$  (1)

Next we compute the following two quantities:

$$N_e = L_{\max} \bmod L_{\min}$$

The preprocessing steps for edge equalization are then accomplished with the following two procedures:

1. Divide  $N_e$  edges of  $keyframe_{\max}$  into  $N_e + 1$  sections.
2. Divide the remaining lines of  $keyframe_{\min}$  into  $N_e$  sections.

As an example, if  $L_k = 15$  and  $L_{k+1} = 11$ , we would divide four lines of  $keyframe_{k+1}$  into two sections each. The remaining lines of  $keyframe_k$  are left intact.

If we equalize the vertex count, we can use parameters  $V_k$  and  $V_{k+1}$  to denote the number of vertices in the two consecutive key frames. In this case, we determine the maximum and minimum number of vertices as

$$V_{\max} = \max(V_k, V_{k+1}), \quad V_{\min} = \min(V_k, V_{k+1}) \quad (3)$$

Then we compute the following two values:

$$\begin{aligned} N_e &= (V_{\min} - 1) \bmod (V_{\max} - 1) \\ N_p &= \text{int}\left(\frac{V_{\min} - 1}{V_{\max} - 1}\right) \end{aligned} \quad (4)$$

These two quantities are then used to perform vertex equalization with the following procedures:

1. Add  $N_p$  points to  $N_e$  line sections of  $keyframe_{\min}$ .
2. Add  $N_p - 1$  points to the remaining edges of  $keyframe_{\min}$ .

For the triangle-to-quadrilateral example,  $V_k = 3$  and  $V_{k+1} = 4$ . Both  $N_e$  and  $N_p$  are 1, so we would add one point to one edge of  $keyframe_k$ . No points would be added to the remaining lines of  $keyframe_k$ .

## Simulating Accelerations

Curve-fitting techniques are often used to specify the animation paths between key frames. Given the vertex positions at the key frames, we can fit the positions with linear or nonlinear paths. Figure 11 illustrates a nonlinear fit of key-frame positions. To simulate accelerations, we can adjust the time spacing for the

in-betweens.

If the motion is to occur at constant speed (zero acceleration), we use equal-interval time spacing for the in-betweens. For instance, with  $n$  in-betweens and

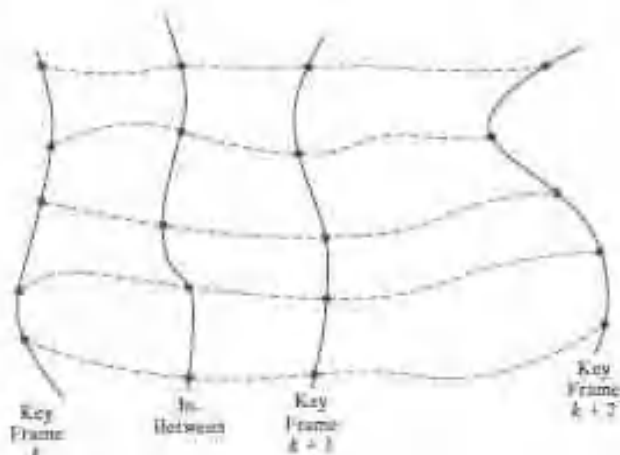
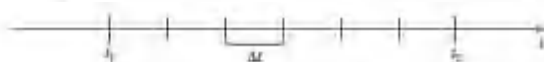


FIGURE 11  
Fitting key frame vertex positions with nonlinear splines.

FIGURE 12  
In-between positions for motion at constant speed.



key-frame times of  $t_1$  and  $t_2$  (Figure 12), the time interval between the key frames is divided into  $n + 1$  equal subintervals, yielding an in-between spacing of

$$\Delta t = \frac{t_2 - t_1}{n + 1} \tag{5}$$

The time for the  $j$ th in-between is

$$tB_j = t_1 + j\Delta t, \quad j = 1, 2, \dots, n \tag{6}$$

and this time value is used to calculate coordinate positions, color, and other physical parameters for that frame of the motion.

Speed changes (nonzero accelerations) are usually necessary at some point in an animation film or cartoon, particularly at the beginning and end of a motion sequence. The startup and slowdown portions of an animation path are often modeled with spline or trigonometric functions, but parabolic and cubic time functions have been applied to acceleration modeling. Animation packages commonly furnish trigonometric functions for simulating accelerations.

To model increasing speed (positive acceleration), we want the time spacing between frames to increase so that greater changes in position occur as the object moves faster. We can obtain an increasing size for the time interval with the function

$$1 - \cos \theta, \quad 0 < \theta < \pi/2$$

For  $n$  in-betweens, the time for the  $j$ th in-between would then be calculated as

$$tB_j = t_1 + \Delta t \left[ 1 - \cos \frac{j\pi}{2(n+1)} \right], \quad j = 1, 2, \dots, n \tag{7}$$

where  $\Delta t$  is the time difference between the two key frames. Figure 13 gives a plot of the trigonometric acceleration function and the in-between spacing for  $n = 5$ .

We can model decreasing speed (deceleration) using the function  $\sin \theta$ , with  $0 < \theta < \pi/2$ . The time position of an in-between is then determined as

$$tB_j = t_1 + \Delta t \sin \frac{j\pi}{2(n+1)}, \quad j = 1, 2, \dots, n \tag{8}$$

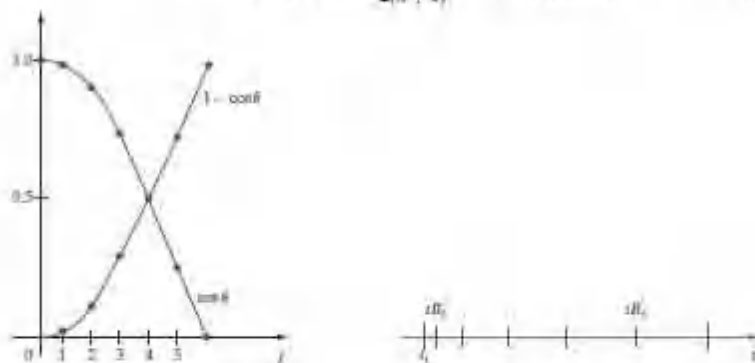
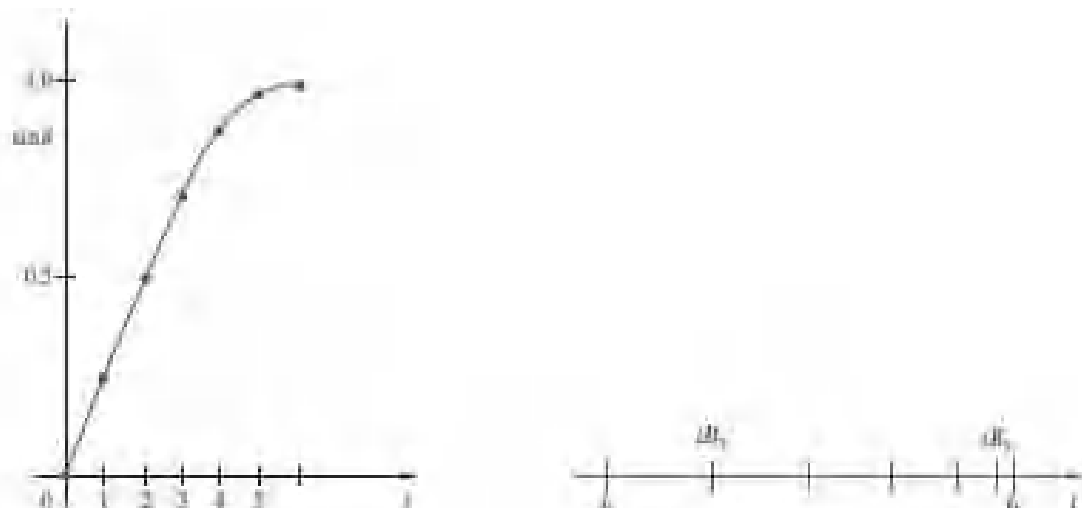


FIGURE 13  
A trigonometric acceleration function and the corresponding in-between spacing for  $n = 5$  and  $\theta = \pi/12$  in Equation 7, producing increased coordinate changes as the object moves through each time interval.



**FIGURE 14**

A trigonometric deceleration function and the corresponding in-between spacing for  $n = 5$  and  $\theta = j\pi/12$  in Equation 8, producing decreased coordinate changes as the object moves through each time interval.

A plot of this function and the decreasing size of the time intervals is shown in Figure 14 for five in-betweens.

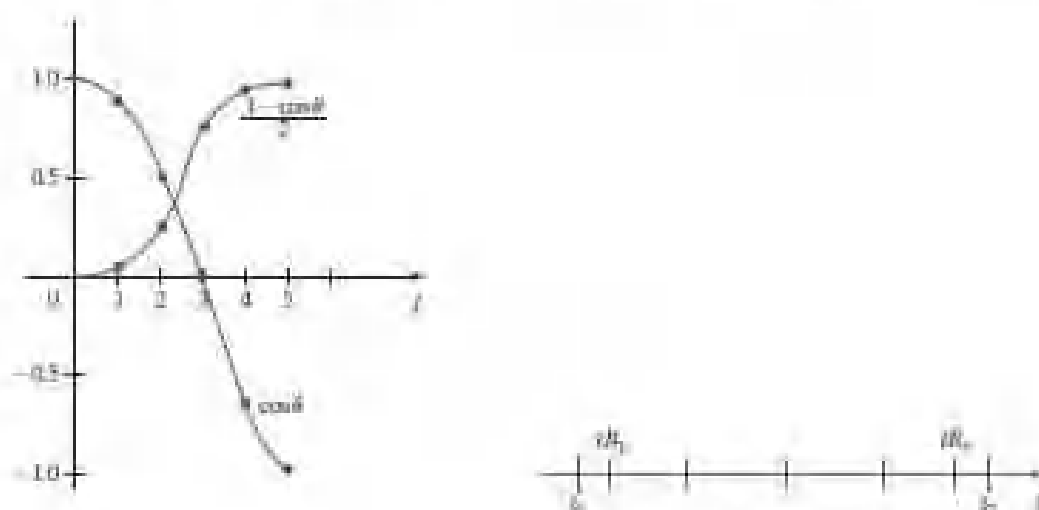
Often, motions contain both speedups and slowdowns. We can model a combination of increasing–decreasing speed by first increasing the in-between time spacing and then decreasing this spacing. A function to accomplish these time changes is

$$\frac{1}{2}(1 + \cos\theta), \quad 0 < \theta < \pi/2$$

The time for the  $j$ th in-between is now calculated as

$$tB_j = t_0 + \Delta t \left\{ \frac{1 + \cos[j\pi/(n+1)]}{2} \right\}, \quad j = 1, 2, \dots, n \quad (9)$$

with  $\Delta t$  denoting the time difference between the two key frames. Time intervals for a moving object first increase and then decrease, as shown in Figure 15.



**FIGURE 15**

The trigonometric acceleration–deceleration function  $(1 + \cos\theta)/2$  and the corresponding in-between spacing for  $n = 5$  in Equation 9.

Processing the in-betweens is simplified by initially modeling “skeleton (wire-frame) objects so that motion sequences can be interactively adjusted. After the animation sequence is completely defined, objects can be fully rendered.

### Motion Specifications

General methods for describing an animation sequence range from an explicit specification of the motion paths to a description of the interactions that produce the motions. Thus, we could define how an animation is to take place by giving the transformation parameters, the motion path parameters, the forces that are to act on objects, or the details of how objects interact to produce motion.

#### Direct Motion Specification

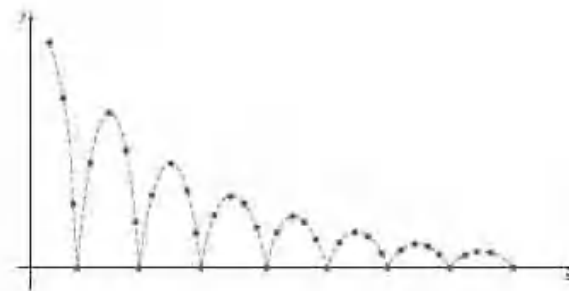
The most straightforward method for defining an animation is *direct motion specification* of the geometric-transformation parameters. Here, we explicitly set the values for the rotation angles and translation vectors. Then the geometric transformation matrices are applied to transform coordinate positions. Alternatively, we could use an approximating equation involving these parameters to specify certain kinds of motions. We can approximate the path of a bouncing ball, for instance, with a damped, rectified, sine curve (Figure 16):

$$y(x) = A | \sin(\omega x + \theta_0) | e^{-kx} \quad (10)$$

where  $A$  is the initial amplitude (height of the ball above the ground),  $\omega$  is the angular frequency,  $\theta_0$  is the phase angle, and  $k$  is the damping constant.

This method for motion specification is particularly useful for simple

FIGURE 16  
Approximating the motion of a  
bouncing ball with a damped sine  
function (Eq. 10).



user-programmed animation sequences.

### Goal-Directed Systems

At the opposite extreme, we can specify the motions that are to take place in general terms that abstractly describe the actions in terms of the final results. In other words, an animation is specified in terms of the final state of the movements. These systems are referred to as *goal-directed*, since values for the motion parameters are determined from the goals of the animation. For example, we could specify that we want an object to “walk” or to “run” to a particular destination; or we could state that we want an object to “pick up” some other specified object. The input directives are then interpreted in terms of component motions that will accomplish the described task. Human motions, for instance, can be defined as a hierarchical structure of submotions for the torso, limbs, and so forth. Thus, when a goal, such as “walk to the door” is given, the movements required of the torso and limbs to accomplish this action are calculated.

### Kinematics and Dynamics

We can also construct animation sequences using *kinematic* or *dynamic* descriptions. With a kinematic description, we specify the animation by giving motion parameters (position, velocity, and acceleration) without reference to causes or goals of the motion. For constant velocity (zero acceleration), we designate the motions of rigid bodies in a scene by giving an initial position and velocity vector for each object. For example, if a velocity is specified as (3, 0, -4) km per sec, then this vector gives the direction for the straight-line motion path and the speed (magnitude of velocity) is calculated as 5 km per sec. If we also specify accelerations (rate of change of velocity), we can generate speedups, slowdowns, and curved motion paths. Kinematic specification of a motion can also be given by simply describing the motion path. This is often accomplished using spline curves.

An alternate approach is to use *inverse kinematics*. Here, we specify the initial and final positions of objects at specified times and the motion parameters are computed by the system. For example, assuming zero acceleration, we can determine the constant velocity that will accomplish the movement of an object from the initial position to the final position. This method is often used with complex objects by giving the positions and orientations of an end node of an object, such as a hand or a foot. The system then determines the motion parameters of other nodes to accomplish the desired motion.

Dynamic descriptions, on the other hand, require the specification of the forces that produce the velocities and accelerations. The description of object behavior in terms of the influence of forces is generally referred to as *physically based modeling*.

Examples of forces affecting object motion include electromagnetic, gravitational, frictional, and other mechanical forces.

Object motions are obtained from the force equations describing physical laws, such as Newton's laws of motion for gravitational and frictional processes, Euler or Navier-Stokes equations describing fluid flow, and Maxwell's equations for electromagnetic forces. For example, the general form of Newton's second law for a particle of mass  $m$  is

$$\mathbf{F} = (d/dt)(m\mathbf{v}) \quad (11)$$

where  $\mathbf{F}$  is the force vector and  $\mathbf{v}$  is the velocity vector. If mass is constant, we solve the equation  $\mathbf{F} = m\mathbf{a}$ , with  $\mathbf{a}$  representing the acceleration vector. Otherwise, mass is a function of time, as in relativistic motions or the motions of space vehicles that consume measurable amounts of fuel per unit time. We can also use *inversedynamics* to obtain the forces, given the initial and final positions of objects and the type of motion required.

Applications of physically based modeling include complex rigid-body systems and such nonrigid systems as cloth and plastic materials. Typically, numerical methods are used to obtain the motion parameters incrementally from the dynamical equations using initial conditions or boundary values.

### Character Animation

Animation of simple objects is relatively straightforward. When we consider the animation of more complex figures such as humans or animals, however, it becomes much more difficult to create realistic animation. Consider the animation of walking or running human (or humanoid)



characters. Based upon observations in their own lives of walking or running people, viewers will expect to see animated characters move in particular ways. If an animated character's movement doesn't match this expectation, the believability of the character may suffer. Thus, much of the work involved in character animation is focused on creating believable movements.

### Articulated Figure Animation

A basic technique for animating people, animals, insects, and other critters is to model them as **articulated figures**, which are hierarchical structures composed of a set of rigid links that are connected at rotary joints (Figure 17). In less formal terms, this just means that we model animate objects as moving stick figures, or simplified skeletons, that can later be wrapped with surfaces representing skin, hair, fur, feathers, clothes, or other outer coverings.

The connecting points, or hinges, for an articulated figure are placed at the shoulders, hips, knees, and other skeletal joints, which travel along specified motion paths as the body moves. For example, when a motion is specified for an object, the



FIGURE 17  
A simple articulated figure with five joints and twelve connecting links, not counting the oval head.

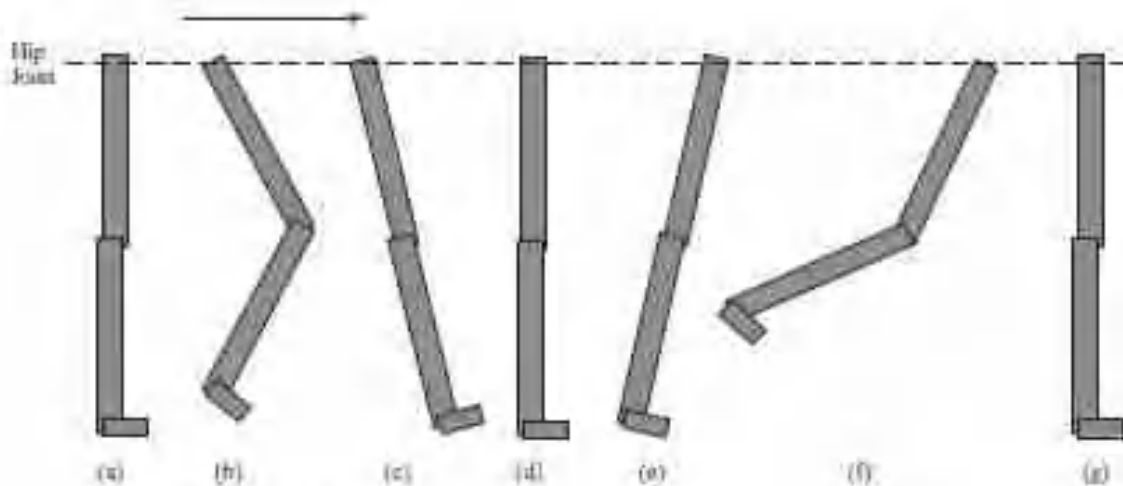


FIGURE 18  
Possible motions for a set of connected links representing a walking leg.

shoulder automatically moves in a certain way and, as the shoulder moves, the arms move. Different types of movement, such as walking, running, or jumping, are defined and associated with particular motions for the joints and connecting links.

A series of walking leg motions, for instance, might be defined as in Figure 18. The hip joint is translated forward along a horizontal line, while the connecting links perform a series of movements about the hip, knee, and

angle joints. Starting with a straight leg [Figure 18(a)], the first motion is a kneebend as the hip moves forward [Figure 18(b)]. Then the leg swings forward, returns to the vertical position, and swings back, as shown in Figures 18(c), (d), and (e). The final motions are a wide swing back and a

return to the straightvertical position, as in Figures 18(f) and (g). This motion cycle is repeated forthe duration of the animation as the figure moves over a specified distance or timeinterval.

As a figure moves, other movements are incorporated into the various joints.A sinusoidal motion, often with varying amplitude, can be applied to the hips sothat they move about on the torso. Similarly, a rolling or rocking motion can be imparted to the shoulders, and the head can bob up and down.

Both kinematic-motion descriptions and inverse kinematics are used in figureanimations. Specifying the joint motions is generally an easier task, but inversekinematics can be useful for producing simple motion over arbitrary terrain. For a complicated figure, inverse kinematics may not produce a unique animationsequence: Many different rotational motions may be possible for a given set ofinitial and final conditions. In such cases, a unique solution may be possible byadding more constraints, such as conservation of momentum, to the system.

### **Motion Capture**

An alternative to determining the motion of a character computationally is todigitally record the movement of a live actor and to base the movement of an animated character on that information. This technique, known as *motion capture* or *mo-cap*, can be used when the movement of the character is predetermined(as in a scripted scene). The animated character will perform the same series of movements as the live actor.

The classic motion capture technique involves placing a set of markers atstrategic positions on the actor's body, such as the arms, legs, hands, feet, andjoints. It is possible to place the markers directly on the actor, but more commonlythey are affixed to a special skintight body suit worn by the actor. The actor isthem filmed performing the scene. Image processing techniques are then usedto identify the positions of the markers in each frame of the film, and their positionsare translated to coordinates. These coordinates are used to determine thepositioning of the body of the animated character. The movement of each markerfrom frame to frame in the film is tracked and used to control the correspondingmovement of the animated character.

To accurately determine the positions of the markers, the scene must be filmedby multiple cameras placed at fixed positions. The digitized marker data fromeachrecording can then be used to triangulate the position of each marker in threedimensions. Typical motion capture systems will use up to two dozen cameras,but systems with several hundred cameras exist.

Optical motion capture systems rely on the reflection of light from a marker into the camera. These can be relatively simple passive systems using photorefectivemarkers that reflect illumination from special lights placed near thecameras, or more advanced active systems in which the markers are poweredand emit light. Active systems can be constructed so that the markers illuminatein a pattern or sequence, which allows each marker to be uniquely identified ineach frame of the recording, simplifying the tracking process.

Non-optical systems rely on the direct transmission of position informationfromthe markers to a recording device. Some non-optical

systems use inertial sensors that provide gyroscope-based position and orientation information. Others use magnetic sensors that measure changes in magnetic flux. A series of transmitters placed around the stage generate magnetic fields that induce current in the magnetic sensors; that information is then transmitted to receivers.

Some motion capture systems record more than just the gross movements of the parts of the actor's body. It is possible to record even the actor's facial movements. Often called *performance capture* systems, these typically use a camera trained on the actor's face and small light-emitting diode (LED) lights that illuminate the face. Small photoreflective markers attached to the face reflect the light from the LEDs and allow the camera to capture the small movements of the muscles of the face, which can then be used to create realistic facial animation on a computer-generated character.

### Periodic Motions

When we construct an animation with repeated motion patterns, such as a rotating object, we need to be sure that the motion is sampled frequently enough to represent the

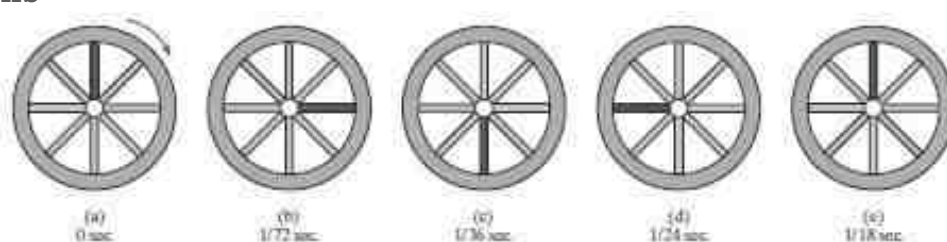


FIGURE 19 Five positions for a red spoke during one cycle of a wheel motion that is turning at the rate of 18 revolutions per second.

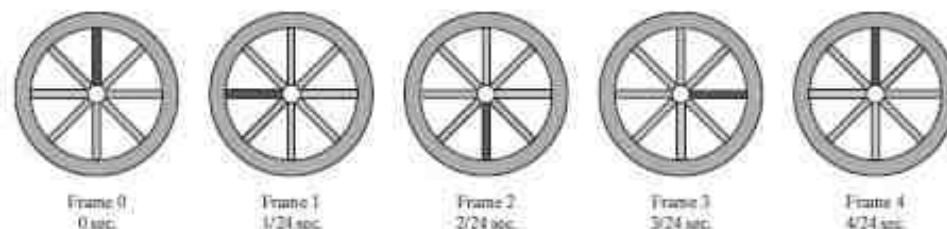


FIGURE 20 The first five film frames of the rotating wheel in Figure 19 produced at the rate of 24 frames per second.

movements correctly. In other words, the motion must be synchronized with the frame-generation rate so that we display enough frames per cycle to show the true motion. Otherwise, the animation may be displayed incorrectly. A typical example of an undersampled periodic-motion display is the wagon wheel in a Western movie that appears to be turning in the wrong direction.

Figure 19 illustrates one complete cycle in the rotation of a wagon wheel with one red spoke that makes 18 clockwise revolutions per second. If this motion is recorded on film at the standard motion-picture projection rate of 24 frames per second, then the first five frames depicting this motion would be as shown in Figure 20. Because the wheel completes 34 of a turn every 124 of a second, only one animation frame is generated per cycle, and the wheel thus appears to be rotating in the opposite (counterclockwise) direction.

In a computer-generated animation, we can control the sampling rate in a periodic motion by adjusting the motion parameters. For example, we can set the angular increment for the motion of a rotating object so that multiple frames are generated in each revolution. Thus, a  $3^\circ$  increment for a rotation

angle produces 120 motion steps during one revolution, and a  $4^\circ$  increment generates 90 steps.

For faster motion, larger rotational steps could be used, so long as the number of samples per cycle is not too small and the motion is clearly displayed. When complex objects are to be animated, we also must take into account the effect that the frame construction time might have on the refresh rate, as discussed in Section 1. The motion of a complex object can be much slower than we want it to be if it takes too long to construct each frame of the animation.

Another factor that we need to consider in the display of a repeated motion is the effect of round-off in the calculations for the motion parameters. We can reset parameter values periodically to prevent the accumulated error from producing erratic motions. For a continuous rotation, we could reset parameter values once every cycle ( $360^\circ$ ).

### THREE DIMENSIONAL OBJECT REPRESENTATIONS

Graphics scenes can contain many different kinds of objects and material surfaces: trees, flowers, clouds, rocks, water, bricks, wood paneling, rubber, paper, marble, steel, glass, plastic, and cloth, just to mention a few. So it may not be surprising that there is no single method that we can use to describe objects that will include all the characteristics of these different materials.

Polygon and quadric surfaces provide precise descriptions for simple Euclidean objects such as polyhedrons and ellipsoids. They are examples of **boundary representations (B-reps)**, which describe a three-dimensional object as a set of surfaces that separate the object interior from the environment. In this chapter, we consider the features of these types of representation schemes and how they are used in computer-graphics applications.

#### Polyhedra

The most commonly used boundary representation for a three-dimensional graphics object is a set of surface polygons that enclose the object interior. Many graphics systems store all object descriptions as sets of surface polygons. This simplifies and speeds up the surface rendering and display of objects because all surfaces are described with linear equations. For this reason, polygon descriptions are often referred to as *standard graphics objects*. In some cases, a polygonal representation is the only one available, but many packages also allow object surfaces to be described with other schemes, such

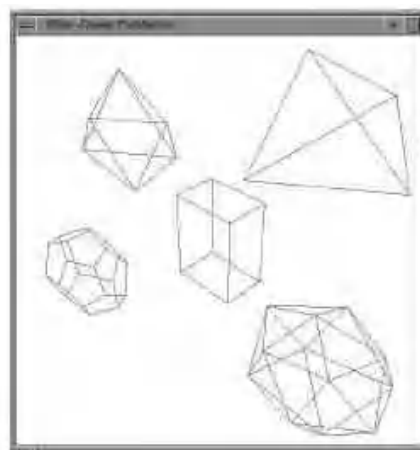


FIGURE 1  
A perspective view of the five GLUT  
polyhedra, scaled and positioned  
within a display window by procedure:  
displayWirePolyhedra.

as spline surfaces, which are usually converted to polygonal representations for processing through the viewing pipeline.

To describe an object as a set of polygon facets, we give the list of vertex coordinates for each polygon section over the object surface. The vertex coordinates and edge information for the surface sections are then stored in tables along with other information, such as the surface normal vector for each polygon. Some graphics packages provide routines for generating a polygon-surface mesh as a set of triangles or quadrilaterals. This allows us to describe a large section of an object's bounding surface, or even the entire surface, with a single command.

And some packages also provide routines for displaying common shapes, such as a cube, sphere, or cylinder, represented with polygon surfaces. Sophisticated graphics systems use fast hardware-implemented polygon renderers that have the capability for displaying a million or more shaded polygons (usually triangles) per second, including the application of surface texture and special lighting effects.

### **OpenGL Polyhedron Functions**

We have two methods for specifying polygon surfaces in an OpenGL program.

Using the polygon primitives we can generate a variety of polyhedron shapes and surface meshes. In addition, we can use GLUT functions to display the five regular polyhedra.

### **OpenGL Polygon Fill-Area Functions**

A set of polygon patches for a section of an object surface, or a completed description for a polyhedron, can be given using the OpenGL primitive constants **GL POLYGON**, **GL TRIANGLES**, **GL TRIANGLE STRIP**, **GL TRIANGLE FAN**, **GL QUADS**, and **GL QUAD STRIP**. For example, we could tessellate the lateral (axial) surface of a cylinder using a quadrilateral strip. Similarly, all faces of a parallelogram can be described with a set of rectangles, and all faces of a triangular pyramid could be specified using a set of connected triangular surfaces.

### **GLUT Regular Polyhedron Functions**

Some standard shapes—the five regular polyhedra—are predefined by routines in the GLUT library. These polyhedra, also called the *Platonic solids*, are distinguished by the fact that all the faces of any regular polyhedron are identical regular polygons. Thus, all edges in a regular polyhedron are equal, all edge angles are equal, and all angles between faces are equal. Polyhedra are named according to the number of faces in each of the solids, and the five regular polyhedra are the regular tetrahedron (or triangular pyramid, with 4 faces), the regular hexahedron (or cube, with 6 faces), the regular octahedron (8 faces), the regular dodecahedron (12 faces), and the regular icosahedron (20 faces).

Ten functions are provided in GLUT for generating these solids: five of the functions produce wire-frame objects, and five display the polyhedra facets as shaded fill areas. The displayed surface characteristics for the fill areas are determined by the material properties and the lighting conditions that we set for a scene. Each regular polyhedron is described in modeling coordinates, so that each is centered at the world-coordinate origin.



We obtain the four-sided, regular triangular pyramid using either of these two functions:

```
glutWireTetrahedron ( );  
or glutSolidTetrahedron ( );
```

This polyhedron is generated with its center at the world-coordinate origin and with a radius (distance from the center of the tetrahedron to any vertex) equal to  $\sqrt{3}$ .

The six-sided regular hexahedron (cube) is displayed with

```
glutWireCube (edgeLength);  
or  
glutSolidCube (edgeLength);
```

Parameter **edgeLength** can be assigned any positive, double-precision floating-point value, and the cube is centered on the coordinate origin. To display the eight-sided regular octahedron, we invoke either of the following commands:

```
glutWireOctahedron ( );  
or  
glutSolidOctahedron ( );
```

This polyhedron has equilateral triangular faces, and the radius (distance from the center of the octahedron at the coordinate origin to any vertex) is 1.0.

The twelve-sided regular dodecahedron, centered at the world-coordinate origin, is generated with

```
glutWireDodecahedron ( );  
or  
glutSolidDodecahedron ( );
```

Each face of this polyhedron is a pentagon. The following two functions generate the twenty-sided regular icosahedron:

```
glutWireIcosahedron ( );  
or  
glutSolidIcosahedron ( );
```

Default radius (distance from the polyhedron center at the coordinate origin to any vertex) for the icosahedron is 1.0, and each face is an equilateral triangle.

#### **Example GLUT Polyhedron Program**

Using the GLUT functions for the Platonic solids, the following program generates a transformed, wire-frame perspective display of these polyhedrons. All five solids are positioned within one display window (shown in Figure 1).

```
#include <GL/glut.h>  
GLsizei winWidth = 500, winHeight = 500; // Initial display-window  
size.  
void init (void)  
{  
glClearColor (1.0, 1.0, 1.0, 0.0); // White display window.  
}  
void displayWirePolyhedra (void)  
{  
glClear (GL_COLOR_BUFFER_BIT); // Clear display window.
```

```

glColor3f (0.0, 0.0, 1.0); // Set line color to blue.
/* Set viewing transformation. */
gluLookAt (5.0, 5.0, 5.0, 0.0, 0.0, 0.0, 1.0, 0.0);
/* Scale cube and display as wire-frame parallelepiped. */
glScalef (1.5, 2.0, 1.0);
glutWireCube (1.0);
/* Scale, translate, and display wire-frame dodecahedron. */
glScalef (0.8, 0.5, 0.8);
glTranslatef (-6.0, -5.0, 0.0);
glutWireDodecahedron ( );
/* Translate and display wire-frame tetrahedron. */
glTranslatef (8.6, 8.6, 2.0);
glutWireTetrahedron ( );
/* Translate and display wire-frame octahedron. */
glTranslatef (-3.0, -1.0, 0.0);
glutWireOctahedron ( );
/* Scale, translate, and display wire-frame icosahedron. */
glScalef (0.8, 0.8, 1.0);
glTranslatef (4.3, -2.0, 0.5);
glutWireIcosahedron ( );
glFlush ( );
}
void winReshapeFcn (GLint newWidth, GLint newHeight)
{
glViewport (0, 0, newWidth, newHeight);
glMatrixMode (GL_PROJECTION);
glFrustum (-1.0, 1.0, -1.0, 1.0, 2.0, 20.0);
glMatrixMode (GL_MODELVIEW);
glClear (GL_COLOR_BUFFER_BIT);
}
void main (int argc, char** argv)
{
glutInit (&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowPosition (100, 100);
glutInitWindowSize (winWidth, winHeight);
glutCreateWindow ("Wire-Frame Polyhedra");
init ( );
glutDisplayFunc (displayWirePolyhedra);
glutReshapeFunc (winReshapeFcn);
glutMainLoop ( );
}

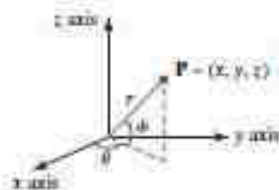
```

### **Curved Surfaces**

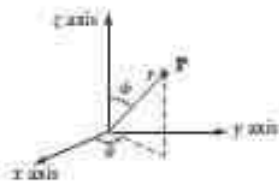
Equations for objects with curved boundaries can be expressed in either a parametric or a nonparametric form. The various objects that are often useful in graphics functions, and spline surfaces. These input object descriptions typically are tessellated to produce polygon-mesh approximations for the surfaces.

### **Quadric Surfaces**

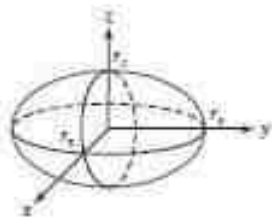
A frequently used class of objects are the *quadric surfaces*, which are described with second-degree equations (quadratics). They include spheres, ellipsoids, tori, paraboloids, and hyperboloids. Quadric surfaces, particularly spheres and ellipsoids, are common elements of graphics scenes, and routines for generating these surfaces are often available in graphics packages. Also, quadric surfaces can be reproduced with rational spline representations.



**FIGURE 2**  
Parametric coordinate position  $(r, \theta, \phi)$  on the surface of a sphere with radius  $r$ .



**FIGURE 3**  
Spherical coordinate parameters  $(r, \theta, \phi)$ , using colatitude by angle  $\phi$ .



**FIGURE 4**  
An ellipsoid with radii  $r_x$ ,  $r_y$ , and  $r_z$ , centered on the coordinate origin.

### Sphere

In Cartesian coordinates, a spherical surface with radius  $r$  centered on the coordinate origin is defined as the set of points  $(x, y, z)$  that satisfy the equation

$$x^2 + y^2 + z^2 = r^2 \quad (1)$$

We can also describe the spherical surface in parametric form, using latitude and longitude angles (Figure 2):

$$\begin{aligned} x &= r \cos \phi \cos \theta, & -\pi/2 \leq \phi \leq \pi/2 \\ y &= r \cos \phi \sin \theta, & -\pi \leq \theta \leq \pi \\ z &= r \sin \phi \end{aligned} \quad (2)$$

The parametric representation in Equations 2 provides a symmetric range for the angular parameters  $\theta$  and  $\phi$ . Alternatively, we could write the parametric equations using standard spherical coordinates, where angle  $\phi$  is specified as the colatitude (Figure 3). Then,  $\phi$  is defined over the range  $0 \leq \phi \leq \pi$ , and  $\theta$  is often taken in the range  $0 \leq \theta < 2\pi$ . We could also set up the representation using parameters  $u$  and  $v$  defined over the range from 0 to 1 by substituting  $\phi = \pi u$  and  $\theta = 2\pi v$ .

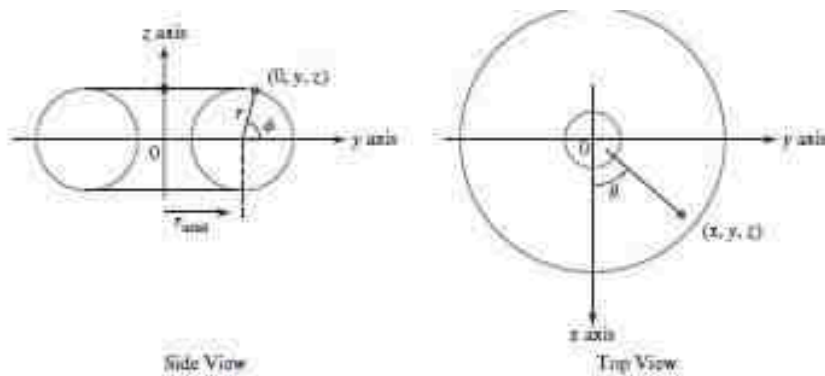
### Ellipsoid

An ellipsoidal surface can be described as an extension of a spherical surface where the radii in three mutually perpendicular directions can have different values (Figure 4). The Cartesian representation for points over the surface of an ellipsoid centered on the origin is

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1 \quad (3)$$

And a parametric representation for the ellipsoid in terms of the latitude angle  $\phi$  and the longitude angle  $\theta$  in Figure 2 is

$$\begin{aligned} x &= r_x \cos \phi \cos \theta, & -\pi/2 \leq \phi \leq \pi/2 \\ y &= r_y \cos \phi \sin \theta, & -\pi \leq \theta \leq \pi \\ z &= r_z \sin \phi \end{aligned} \quad (4)$$



**FIGURE 5**  
A torus, centered on the coordinate origin, with a circular cross-section and with the torus axis along the  $z$  axis.

### Torus

A doughnut-shaped object is called a *torus* or *anchor ring*. Most often it is described as the surface generated by rotating a circle or an ellipse about a coplanar axis line that is external to the conic. The defining parameters for a torus are then the distance of the conic center from the rotation axis and the dimensions of the conic. A torus generated by the rotation of a circle with radius  $r$  in the  $yz$  plane about the  $z$  axis is shown in Figure 5. With the circle center on the  $y$  axis, the axial radius,  $r_{\text{axial}}$ , of the resulting torus is equal to the distance along the  $y$  axis to the circle center from the  $z$  axis (the rotation axis); and the cross-sectional radius of the torus is the radius of the generating circle.

The equation for the cross-sectional circle shown in the side view of Figure 5 is

$$(y - r_{\text{axial}})^2 + z^2 = r^2$$

Rotating this circle about the  $z$  axis produces the torus whose surface positions are described with the Cartesian equation

$$(\sqrt{x^2 + y^2} - r_{\text{axial}})^2 + z^2 = r^2 \quad (5)$$

The corresponding parametric equations for the torus with a circular cross-section are

$$\begin{aligned} x &= (r_{\text{axial}} + r \cos \phi) \cos \theta, & -\pi &\leq \phi \leq \pi \\ y &= (r_{\text{axial}} + r \cos \phi) \sin \theta, & -\pi &\leq \theta \leq \pi \\ z &= r \sin \phi \end{aligned} \quad (6)$$

We could also generate a torus by rotating an ellipse, instead of a circle, about the  $z$  axis. For an ellipse in the  $yz$  plane with semimajor and semiminor axes denoted as  $r_y$  and  $r_z$ , we can write the ellipse equation as

$$\left(\frac{y - r_{\text{axial}}}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1$$

where  $r_{\text{axial}}$  is the distance along the  $y$  axis from the rotation  $z$  axis to the ellipse center. This generates a torus that can be described with the Cartesian equation

$$\left(\frac{\sqrt{x^2 + y^2} - r_{\text{axial}}}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1 \quad (7)$$

The corresponding parametric representation for the torus with an elliptical cross-section is

$$\begin{aligned} x &= (r_{\text{axial}} + r_y \cos \phi) \cos \theta, & -\pi &\leq \phi \leq \pi \\ y &= (r_{\text{axial}} + r_y \cos \phi) \sin \theta, & -\pi &\leq \theta \leq \pi \\ z &= r_z \sin \phi \end{aligned} \quad (8)$$

Other variations on the preceding torus equations are possible. For example, we could generate a torus surface by rotating either a circle or an ellipse along an elliptical path around the rotation axis.

## 5 Superquadrics

The class of objects called **Superquadrics** is a generalization of the quadric representations. Superquadrics are formed by incorporating additional parameters into the quadric equations to provide increased flexibility for adjusting object shapes. One additional parameter is added to curve equations, and two additional parameters are used in surface equations.

### Superellipse

We obtain a Cartesian representation for a superellipse from the corresponding equation for an ellipse by allowing the exponent on the  $x$  and  $y$  terms to be variable. One way to do this is to write the Cartesian superellipse equation in the form

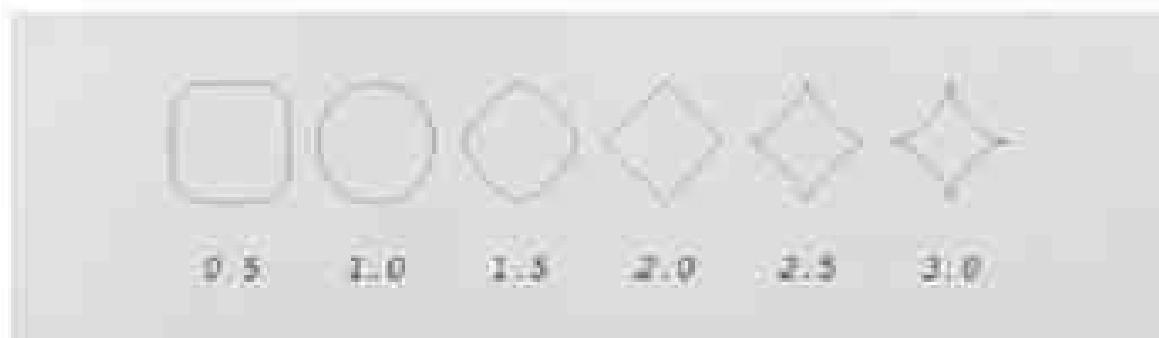
$$\left(\frac{x}{r_x}\right)^{2s} + \left(\frac{y}{r_y}\right)^{2s} = 1 \quad (9)$$

where parameter  $s$  can be assigned any real value. When  $s = 1$ , we have an ordinary ellipse.

Corresponding parametric equations for the superellipse of Equation 9 can be expressed as

$$\begin{aligned} x &= r_x \cos^s \theta, & -\pi \leq \theta \leq \pi \\ y &= r_y \sin^s \theta \end{aligned} \quad (10)$$

Figure 6 illustrates superellipse shapes that can be generated using various values for parameter  $s$ .



**FIGURE 6** Superellipses plotted with values for parameter  $s$  ranging from 0.5 to 3.0 and with  $r_x = r_y$ .



## Superellipsoid

A Cartesian representation for a superellipsoid is obtained from the equation for an ellipsoid by incorporating two exponent parameters as follows:

$$\left[ \left( \frac{x}{r_x} \right)^{2/s_1} + \left( \frac{y}{r_y} \right)^{2/s_1} \right]^{s_2/2} + \left( \frac{z}{r_z} \right)^{2/s_2} = 1 \quad (11)$$

For  $s_1 = s_2 = 1$ , we have an ordinary ellipsoid.

We can then write the corresponding parametric representation for the superellipsoid of Equation 11 as

$$\begin{aligned} x &= r_x \cos^n \phi \cos^m \theta, & -\pi/2 \leq \phi \leq \pi/2 \\ y &= r_y \cos^n \phi \sin^m \theta, & -\pi \leq \theta \leq \pi \\ z &= r_z \sin^n \phi \end{aligned} \quad (12)$$

Color Plate 10 illustrates superellipsoid shapes that can be generated using various values for parameters  $s_1$  and  $s_2$ . These and other superquadric shapes can be combined to create more complex structures, such as depictions of furniture, threaded bolts, and other hardware.

## SPLINE REPRESENTATION

Splines are another example of boundary representation modeling techniques. In drafting terminology, a spline is a flexible strip used to produce a smooth curve through a designated set of points. Several small weights are distributed along the length of the strip to hold it in position on the drafting table as the curve is drawn. The term *spline curve* originally referred to a curve drawn in this manner. We can mathematically describe such a curve with a piecewise cubic polynomial function whose first and second derivatives are continuous across the various curve sections. In computer graphics, the term **spline curve** now refers to any composite curve formed with polynomial sections satisfying any specified continuity conditions at the boundary of the pieces. A **spline surface** can be described with two sets of spline curves. There are several different kinds of spline specifications that are used in computer-graphics applications. Each individual specification simply refers to a particular type of polynomial with certain prescribed boundary conditions.

Splines are used to design curve and surface shapes, to digitize drawings, and to specify animation paths for the objects or the camera position in a scene. Typical computer-aided design (CAD) applications for splines include the design of automobile bodies, aircraft and spacecraft surfaces, ship hulls, and home appliances.

### BEZIER CURVES

This spline approximation method was developed by the French engineer Pierre Bézier for use in the design of Renault automobile bodies. **Bézier splines** have a number of properties that make them highly useful and convenient for curve and surface design. They are also easy to implement. For these reasons, Bézier splines are widely available in various CAD systems, in general graphics packages, and in assorted drawing and painting packages.

In general, a Bézier curve section can be fitted to any number of control points, although some graphic packages limit the number of control points to four. The degree of the Bézier polynomial is determined by the number of control points to be approximated and their relative position. As with the interpolation splines, we can specify the Bézier curve path in the vicinity of the control points using blending functions, a characterizing matrix, or boundary conditions. For general Bézier curves, with no restrictions on the number of control points, the blending functions specification is the most convenient representation.

### Bézier Curve Equations

We first consider the general case of  $n + 1$  control-point positions, denoted as  $\mathbf{p}_k = (x_k, y_k, z_k)$ , with  $k$  varying from 0 to  $n$ . These coordinate points are blended to produce the following position vector  $\mathbf{P}(u)$ , which describes the path of an approximating Bézier polynomial function between  $\mathbf{p}_0$  and  $\mathbf{p}_n$ :

$$\mathbf{P}(u) = \sum_{k=0}^n \mathbf{p}_k \text{BEZ}_{k,n}(u), \quad 0 \leq u \leq 1 \quad (22)$$

The Bézier blending functions  $\text{BEZ}_{k,n}(u)$  are the Bernstein polynomials

$$\text{BEZ}_{k,n}(u) = C(n, k) u^k (1 - u)^{n-k} \quad (23)$$

where parameters  $C(n, k)$  are the binomial coefficients

$$C(n, k) = \frac{n!}{k!(n-k)!} \quad (24)$$

Equation 22 represents a set of three parametric equations for the individual curve coordinates:

$$\begin{aligned} x(u) &= \sum_{k=0}^n x_k \text{BEZ}_{k,n}(u) \\ y(u) &= \sum_{k=0}^n y_k \text{BEZ}_{k,n}(u) \\ z(u) &= \sum_{k=0}^n z_k \text{BEZ}_{k,n}(u) \end{aligned} \quad (25)$$

In most cases, a Bézier curve is a polynomial of a degree that is one less than the designated number of control points: Three points generate a parabola, four points a cubic curve, and so forth. Figure 20 demonstrates the appearance of some Bézier curves for various selections of control points in the  $xy$  plane ( $z = 0$ ). With certain control-point placements, however, we obtain degenerate Bézier polynomials. For example, a Bézier curve generated with three collinear control points is a straight-line segment, and a set of control points that are all at the same coordinate position produce a Bézier "curve" that is a single point.

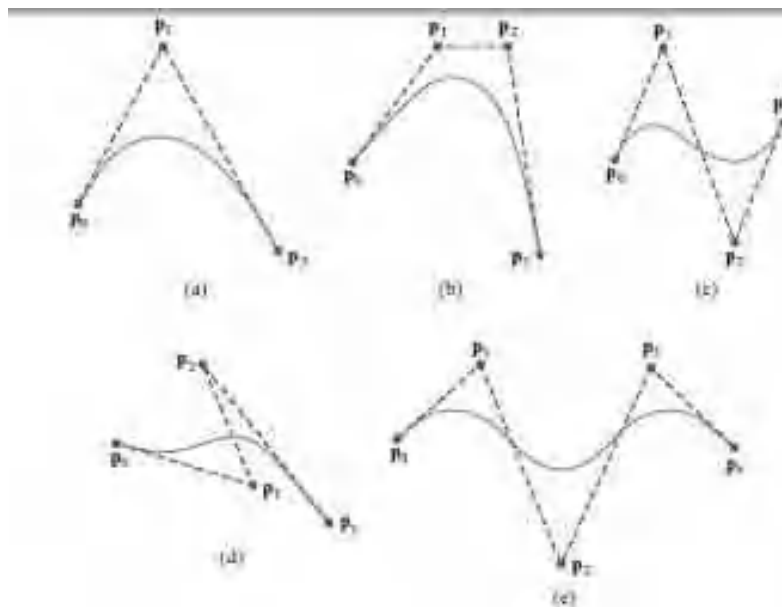
Recursive calculations can be used to obtain successive binomial-coefficient values as

$$C(n, k) = \frac{n-k+1}{k} C(n, k-1) \quad (26)$$

for  $n \geq k$ . Also, the Bézier blending functions satisfy the recursive relationship

$$\text{BEZ}_{k,n}(u) = (1-u)\text{BEZ}_{k,n-1}(u) + u\text{BEZ}_{k-1,n-1}(u), \quad n \geq k \geq 1 \quad (27)$$

with  $\text{BEZ}_{k,k} = u^k$  and  $\text{BEZ}_{0,k} = (1-u)^k$ .



**FIGURE 20**  
Examples of two-dimensional Bézier curves generated with three, four, and five control points. Dashed lines correct the control-point positions.

### Properties of Bézier Curves

A very useful property of a Bézier curve is that the curve connects the first and last control points. Thus, a basic characteristic of any Bézier curve is that

$$\begin{aligned} P(0) &= p_0 \\ P(1) &= p_n \end{aligned} \quad (28)$$

Values for the parametric first derivatives of a Bézier curve at the endpoints can be calculated from control-point coordinates as

$$\begin{aligned} P'(0) &= -np_0 + np_1 \\ P'(1) &= -np_{n-1} + np_n \end{aligned} \quad (29)$$

From these expressions, we see that the slope at the beginning of the curve is along the line joining the first two control points, and the slope at the end of the curve is along the line joining the last two endpoints. Similarly, the parametric second derivatives of a Bézier curve at the endpoints are calculated as

$$\begin{aligned} P''(0) &= n(n-1)[(p_2 - p_1) - (p_1 - p_0)] \\ P''(1) &= n(n-1)[(p_{n-1} - p_{n-2}) - (p_{n-1} - p_n)] \end{aligned} \quad (30)$$

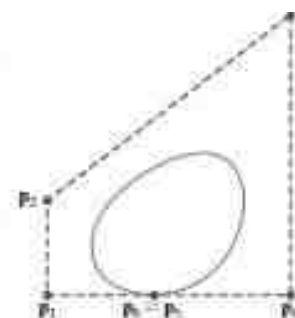
Another important property of any Bézier curve is that it lies within the convex hull (convex polygon boundary) of the control points. This follows from the fact that the Bézier blending functions are all positive and their sum is always 1:

$$\sum_{k=0}^n \text{BEZ}_{k,n}(u) = 1 \quad (31)$$

so that any curve position is simply the weighted sum of the control-point positions. The convex-hull property for a Bézier curve ensures that the polynomial smoothly follows the control points without erratic oscillations.

### Design Techniques Using Bézier Curves

A closed Bézier curve is generated when we set the last control-point position to the coordinate position of the first control point, as in the example shown in Figure 22. Also, specifying multiple control points at a single coordinate position



**FIGURE 22**  
A closed Bézier curve generated by specifying the first and last control points at the same location.

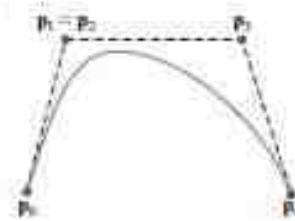
gives more weight to that position. In Figure 23, a single coordinate position is input as two control points, and the resulting curve is pulled nearer to this position.

We can fit a Bézier curve to any number of control points, but this requires the calculation of polynomial functions of higher degree. When complicated curves are to be generated, they can be formed by piecing together several Bézier sections of lower degree. Generating smaller Bézier-curve sections also gives us better local control over the shape of the curve. Because Bézier curves connect the first and last control points, it is easy to match curve sections (zero-order continuity). Also, Bézier curves have the important property that the tangent to the curve at an endpoint is along the line joining that endpoint to the adjacent control point. Therefore, to obtain first-order continuity between curve sections, we can pick control points  $p_0$  and  $p_1$  for the next curve section to be along the same straight line as control points  $p_{n-1}$  and  $p_n$  of the preceding section (Figure 24). If the first curve section has  $n$  control points and the next curve section has  $n'$  control points, then we match curve tangents by placing control point  $p_1$  at the position

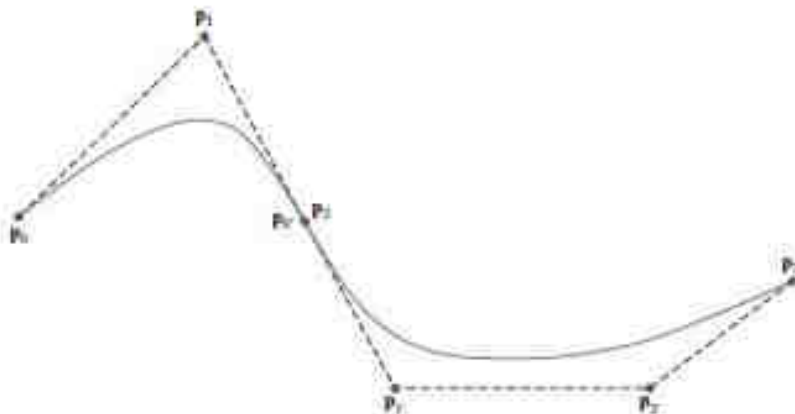
$$p_1 = p_0 + \frac{n}{n'}(p_n - p_{n-1}) \quad (32)$$

To simplify the placement of  $p_1$ , we can require only geometric continuity and place  $p_1$  anywhere along the line of  $p_{n-1}$  and  $p_n$ .

We obtain  $C^2$  continuity by using the expressions in Equations 30 to match parametric second derivatives for two adjacent Bézier sections. This establishes a coordinate position for control point  $p_2$ , in addition to the fixed positions for



**FIGURE 23**  
A Bézier curve can be made to pass closer to a given coordinate position by assigning multiple control points to that position.



**FIGURE 24**  
Piecewise approximation curve formed with two Bézier sections. Zero-order and first-order continuity is attained between the two curve sections by setting  $p_2 = p_4$  and by setting  $p_3$  along the line formed with points  $p_1$  and  $p_5$ .

$p_3$  and  $p_5$  that we need for  $C^2$  and  $C^3$  continuity. However, requiring second-order continuity for Bézier curve sections can be unnecessarily restrictive. This is particularly true with cubic curves, which have only four control points per section. In this case, second-order continuity fixes the position of the first three control points and leaves us only one point that we can use to adjust the shape of the curve segment.

### Cubic Bézier Curves

Many graphics packages provide functions for displaying only cubic splines. This allows reasonable design flexibility while avoiding the increased calculations needed with higher-order polynomials. Cubic Bézier curves are generated with four control points. The four blending functions for cubic Bézier curves, obtained by substituting  $n = 3$  into Equation 23, are

$$\begin{aligned} \text{BEZ}_{0,3} &= (1-u)^3 \\ \text{BEZ}_{1,3} &= 3u(1-u)^2 \\ \text{BEZ}_{2,3} &= 3u^2(1-u) \\ \text{BEZ}_{3,3} &= u^3 \end{aligned} \quad (33)$$

Plots of the four cubic Bézier blending functions are given in Figure 25. The form of the blending functions determine how the control points influence the shape of the curve for values of parameter  $u$  over the range from 0 to 1. At  $u = 0$ ,

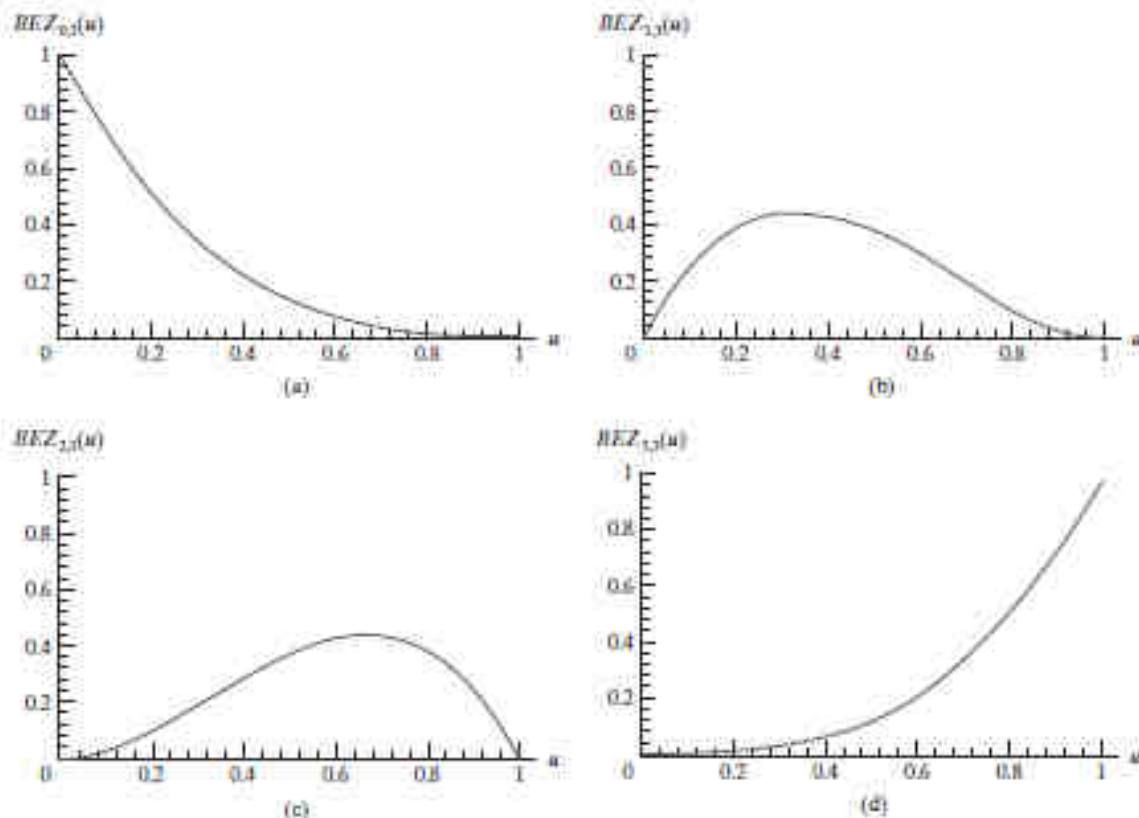


FIGURE 25  
The four Bézier blending functions for cubic curves ( $n = 3$ ).

the only nonzero blending function is  $BEZ_{0,3}$ , which has the value 1. At  $u = 1$ , the only nonzero function is  $BEZ_{3,3}(1) = 1$ . Thus, a cubic Bézier curve always begins at control point  $p_0$  and ends at the position of control point  $p_3$ . The other functions,  $BEZ_{1,3}$  and  $BEZ_{2,3}$ , influence the shape of the curve at intermediate values of the parameter  $u$  so that the resulting curve tends toward the points  $p_1$  and  $p_2$ . Blending function  $BEZ_{1,3}$  is maximized at  $u = 1/3$ , and  $BEZ_{2,3}$  is maximized at  $u = 2/3$ .

We note in Figure 25 that each of the four blending functions is nonzero over the entire range of parameter  $u$  between the endpoint positions. Thus, Bézier curves do not allow for local control of the curve shape. If we reposition any one of the control points, the entire curve is affected.

At the end positions of the cubic Bézier curve, the parametric first derivatives (slopes) are

$$\mathbf{P}'(0) = 3(\mathbf{p}_1 - \mathbf{p}_0), \quad \mathbf{P}'(1) = 3(\mathbf{p}_3 - \mathbf{p}_2)$$

and the parametric second derivatives are

$$\mathbf{P}''(0) = 6(\mathbf{p}_2 - 2\mathbf{p}_1 + \mathbf{p}_0), \quad \mathbf{P}''(1) = 6(\mathbf{p}_1 - 2\mathbf{p}_2 + \mathbf{p}_3)$$

We can construct complex spline curves using a series of cubic-Bézier sections. Using expressions for the parametric derivatives, we can equate curve tangents to attain  $C^1$  continuity between the curve sections. In addition, we could use the expressions for the second derivatives to obtain  $C^2$  continuity, although this leaves us with no options for the placement of the first three control points.

A matrix formulation for the cubic-Bézier curve function is obtained by expanding the polynomial expressions for the blending functions and restruc-



ting the equations as

$$\mathbf{P}(u) = [u^3 \quad u^2 \quad u \quad 1] \mathbf{M}_{\text{bez}} \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix} \quad (34)$$

where the Bézier matrix is

$$\mathbf{M}_{\text{bez}} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad (35)$$

We could also introduce additional parameters to allow adjustment of curve "tension" and "bias," as we did with the interpolating splines. But more versatile types of splines (such as B-splines and beta-splines, discussed later in this chapter) are often provided with this capability.

## BEZIER SURFACES

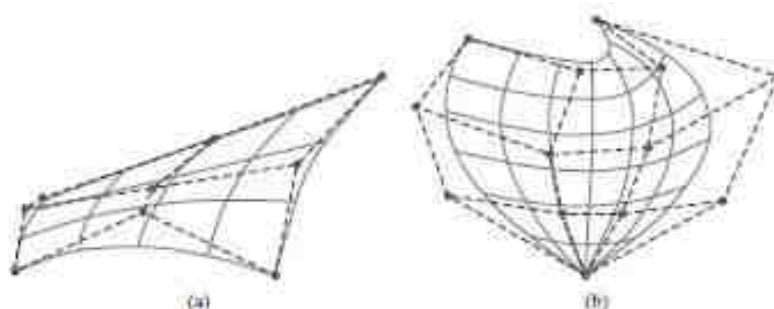
Two sets of orthogonal Bézier curves can be used to design an object surface. The parametric vector function for the Bézier surface is formed as the tensor product of Bézier blending functions:

$$\mathbf{P}(u, v) = \sum_{j=0}^m \sum_{k=0}^n \mathbf{p}_{j,k} \text{BEZ}_{j,m}(u) \text{BEZ}_{k,n}(v) \quad (36)$$

with  $\mathbf{p}_{j,k}$  specifying the location of the  $(m+1)$  by  $(n+1)$  control points.

Figure 26 illustrates two Bézier surface plots. The control points are connected by dashed lines, and the solid lines show curves of constant  $u$  and

**FIGURE 26**  
Wire-frame Bézier surfaces constructed with (a) 9 control points arranged in a  $3 \times 3$  mesh and (b) 16 control points arranged in a  $4 \times 4$  mesh. Dashed lines connect the control points.

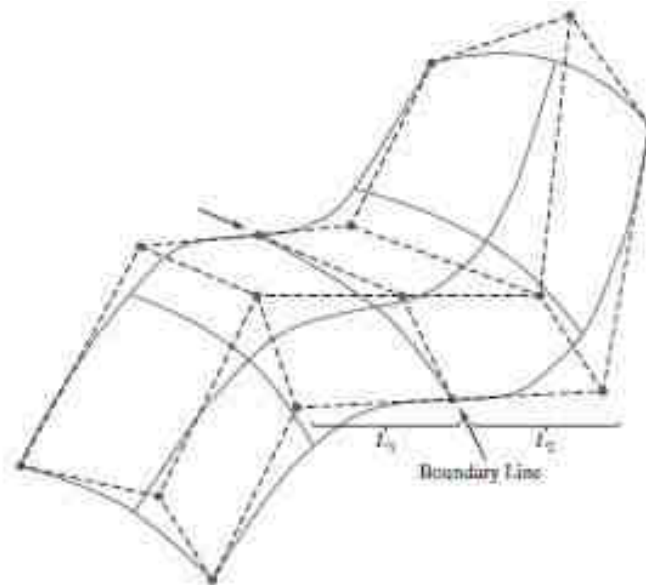


constant  $v$ . Each curve of constant  $u$  is plotted by varying  $v$  over the interval from 0 to 1, with  $u$  fixed at one of the values in this unit interval. Curves of constant  $v$  are plotted similarly.

Bézier surfaces have the same properties as Bézier curves, and they provide a convenient method for interactive design applications. To specify the three-dimensional coordinate positions for the control points, we could first construct a rectangular grid in the  $xy$  "ground" plane. We then choose elevations above the ground plane at the grid intersections as the  $z$ -coordinate values for the control points.

Figure 27 illustrates a surface formed with two Bézier sections. As with curves, a smooth transition from one section to the other is assured by establishing both zero-order and first-order continuity at the boundary line. Zero-order continuity is obtained by matching control points at the boundary. First-order continuity is obtained by choosing control points along a straight line across the boundary and by maintaining a constant ratio of collinear line segments for each set of specified control points across section boundaries.

**FIGURE 27**  
A composite Bézier surface constructed with two Bézier sections, joined at the indicated boundary line. The dashed lines connect the control points. First-order continuity is established by making the ratio of length  $l_1$  to length  $l_2$  constant for each collinear line of control points across the boundary between the surface sections.



## B-SPLINE CURVES

This spline category is the most widely used, and **B-spline** functions are commonly available in CAD systems and many graphics-programming packages. Like Bézier splines, B-splines are generated by approximating a set of control points. But **B-splines** have two advantages over Bézier splines: (1) the degree of a B-spline polynomial can be set independently of the number of control points (with certain limitations), and (2) B-splines allow local control over the shape of a spline. The tradeoff is that B-splines are more complex than Bézier splines.

### B-Spline Curve Equations

We can write a general expression for the calculation of coordinate positions along a B-spline curve using a blending-function formulation as

$$P(u) = \sum_{i=0}^n p_i B_{i,d}(u), \quad u_{\min} \leq u \leq u_{\max}, \quad 2 \leq d \leq n+1 \quad (27)$$

where  $p_i$  is an input set of  $n+1$  control points. There are several differences between this B-spline formulation and the expression for a Bézier spline curve. The range of parameter  $u$  now depends on how we choose the other B-spline parameters. And the B-spline blending functions  $B_{i,d}$  are polynomials of degree  $d-1$ , where  $d$  is the degree parameter. (Sometimes parameter  $d$  is alluded to as the "order" of the polynomial, but this can be misleading because the term order is also often used to mean simply the degree of the polynomial.) The degree parameter  $d$  can be assigned any integer value in the range from 2 up to the number of control points ( $n+1$ ). Actually, we could also set the value of the degree parameter at 1, but then our "curve" is just a point plot of the control points. Local control for B-splines is achieved by defining the blending functions over subintervals of the total range of  $u$ .

Blending functions for B-spline curves are defined by the Cox-deBoor recursion formulas:

$$B_{k,1}(u) = \begin{cases} 1 & \text{if } u_k \leq u \leq u_{k+1} \\ 0 & \text{otherwise} \end{cases} \quad (38)$$

$$B_{k,d}(u) = \frac{u - u_k}{u_{k+d-1} - u_k} B_{k,d-1}(u) + \frac{u_{k+d} - u}{u_{k+d} - u_{k+1}} B_{k+1,d-1}(u)$$

where each blending function is defined over  $d$  subintervals of the total range of  $u$ . Each subinterval endpoint  $u_j$  is referred to as a **knot**, and the entire set of selected subinterval endpoints is called a **knot vector**. We can choose any values for the subinterval endpoints, subject to the condition  $u_j \leq u_{j+1}$ . Values for  $u_{\min}$  and  $u_{\max}$  then depend on the number of control points we select, the value we choose for the degree parameter  $d$ , and how we set up the subintervals (knot vector). Because it is possible to choose the elements of the knot vector so that some denominators in the Cox-deBoor calculations evaluate to 0, this formulation assumes that any terms evaluated as 0/0 are to be assigned the value 0.

Figure 28 demonstrates the local-control characteristics of B-splines. In addition to local control, B-splines allow us to vary the number of control points used to design a curve without changing the degree of the polynomial. Also, we can increase the number of values in the knot vector to aid in curve design. When we do this, however, we must add control points because the size of the knot vector depends on parameter  $u$ .

**FIGURE 28**  
Local modification of a B-spline curve. Changing one of the control points in (a) produces curve (b), which is modified only in the neighborhood of the altered control point.



B-spline curves have the following properties:

- The polynomial curve has degree  $d - 1$  and  $C^{d-2}$  continuity over the range of  $u$ .
- For  $n + 1$  control points, the curve is described with  $n + 1$  blending functions.
- Each blending function  $B_{k,d}$  is defined over  $d$  subintervals of the total range of  $u$ , starting at knot value  $u_k$ .
- The range of parameter  $u$  is divided into  $n + d$  subintervals by the  $n + d + 1$  values specified in the knot vector.
- With knot values labeled as  $\{u_0, u_1, \dots, u_{n+d}\}$ , the resulting B-spline curve is defined only in the interval from knot value  $u_{d-1}$  up to knot value  $u_{n+1}$ . (Some blending functions are undefined outside this interval.)
- Each section of the spline curve (between two successive knot values) is influenced by  $d$  control points.
- Any one control point can affect the shape of at most  $d$  curve sections.

In addition, a B-spline curve lies within the convex hull of at most  $d + 1$  control points, so that B-splines are tightly bound to the input positions. For any value of  $u$  in the interval from knot value  $u_{d-1}$  to  $u_{n+1}$ , the sum over all basis functions is 1, as follows:

$$\sum_{k=0}^n B_{k,d}(u) = 1 \quad (39)$$

Given the control-point positions and the value of the degree parameter  $d$ , we then need to specify the knot values to obtain the blending functions using the recurrence relations 38. There are three general classifications for knot vectors: uniform, open uniform, and nonuniform. B-splines are commonly described according to the selected knot vector class.

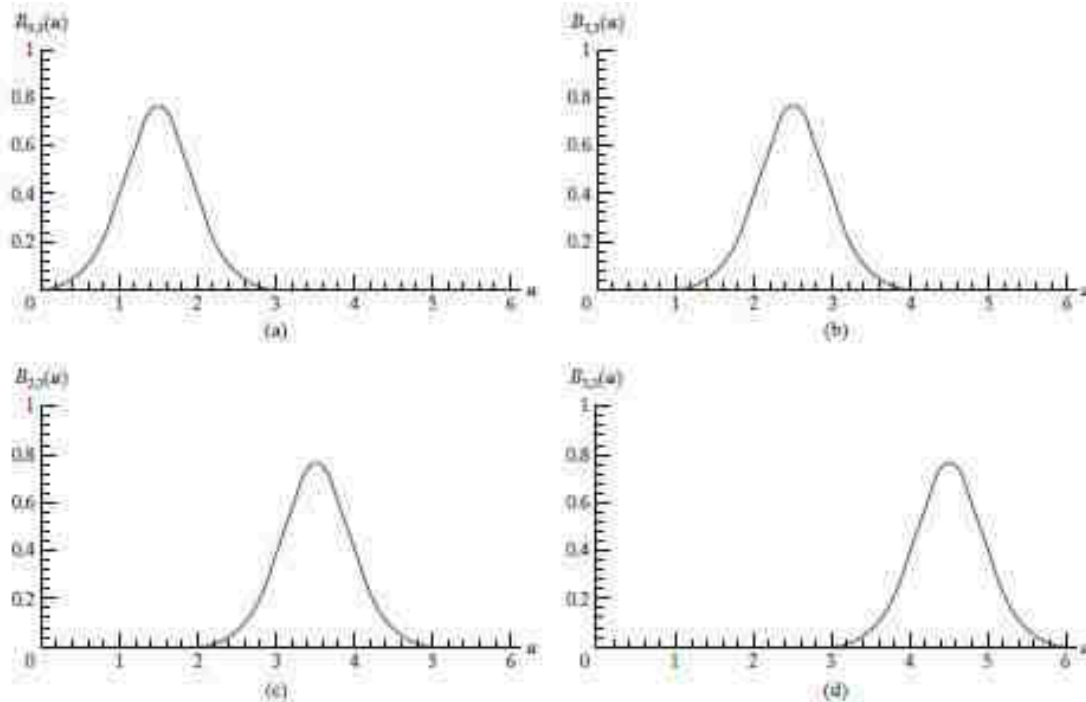
### Uniform Periodic B-Spline Curves

When the spacing between knot values is constant, the resulting curve is called a **uniform B-spline**. For example, we can set up a uniform knot vector as

$$\{-1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0\}$$

Often knot values are normalized to the range between 0 and 1, as in

$$\{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$$



**FIGURE 28**  
Periodic B-spline blending functions for  $n = d = 3$  and a uniform, integer knot vector.

It is convenient in many applications to set up uniform knot values with a separation of 1 and a starting value of 0. The following knot vector is an example of this specification scheme:

$$\{0, 1, 2, 3, 4, 5, 6, 7\}$$

Uniform B-splines have **periodic** blending functions. That is, for given values of  $n$  and  $d$ , all blending functions have the same shape. Each successive blending function is simply a shifted version of the previous function:

$$B_{k,d}(u) = B_{k+1,d}(u + \Delta u) = B_{k+2,d}(u + 2\Delta u) \quad (40)$$

where  $\Delta u$  is the interval between adjacent knot values. Figure 29 shows the quadratic, uniform B-spline blending functions generated in the following example for a curve with four control points.

#### EXAMPLE 1 Uniform, Quadratic B-Splines

To illustrate the formulation of B-spline blending functions for a uniform,

integer knot vector, we select parameter values  $d = n = 3$ . The knot vector must then contain  $n + d + 1 = 7$  knot values:

$$\{0, 1, 2, 3, 4, 5, 6\}$$

and the range for parameter  $u$  is from 0 to 6, with  $n + d = 6$  subintervals.

Each of the four blending functions spans  $d = 3$  subintervals of the total range for  $u$ . Using the recurrence relations 38, we obtain the first-blending function as:

$$B_{0,3}(u) = \begin{cases} \frac{1}{2}u^2, & \text{for } 0 \leq u < 1 \\ \frac{1}{2}u(2-u) + \frac{1}{2}(u-1)(3-u), & \text{for } 1 \leq u < 2 \\ \frac{1}{2}(3-u)^2, & \text{for } 2 \leq u < 3 \end{cases}$$

We obtain the next periodic blending function using Equation 40, substituting  $u - 1$  for  $u$  in  $B_{0,3}$ , and shifting the starting positions up by 1:

$$B_{1,3}(u) = \begin{cases} \frac{1}{2}(u-1)^2, & \text{for } 1 \leq u < 2 \\ \frac{1}{2}(u-1)(3-u) + \frac{1}{2}(u-2)(4-u), & \text{for } 2 \leq u < 3 \\ \frac{1}{2}(4-u)^2, & \text{for } 3 \leq u < 4 \end{cases}$$

Similarly, the remaining two periodic functions are obtained by successively shifting  $B_{1,3}$  to the right:

$$B_{2,3}(u) = \begin{cases} \frac{1}{2}(u-2)^2, & \text{for } 2 \leq u < 3 \\ \frac{1}{2}(u-2)(4-u) + \frac{1}{2}(u-3)(5-u), & \text{for } 3 \leq u < 4 \\ \frac{1}{2}(5-u)^2, & \text{for } 4 \leq u < 5 \end{cases}$$

$$B_{3,3}(u) = \begin{cases} \frac{1}{2}(u-3)^2, & \text{for } 3 \leq u < 4 \\ \frac{1}{2}(u-3)(5-u) + \frac{1}{2}(u-4)(6-u), & \text{for } 4 \leq u < 5 \\ \frac{1}{2}(6-u)^2, & \text{for } 5 \leq u < 6 \end{cases}$$

A plot of the four periodic, quadratic blending functions is given in Figure 29, which demonstrates the local feature of B-splines. The first control point is multiplied by blending function  $B_{0,3}(u)$ . Therefore, changing the



position of the first control point affects the shape of the curve only up to  $u = 3$ . Similarly, the last control point influences the shape of the spline curve in the interval where  $B_{2,3}$  is defined.

Figure 29 also illustrates the limits of the B-spline curve for this example. All blending functions are present in the interval from  $u_{(i-1)} = 2$  to  $u_{(i+1)} = 4$ . Below 2 and above 4, not all blending functions are present. This interval, from 2 to 4, is the range of the polynomial curve, and the interval in which Equation 39 is valid. Thus, the sum of all blending functions is 1 within this interval. Outside this interval, we cannot sum all blending functions, since they are not all defined below 2 and above 4.

Because the range of the resulting polynomial curve is from 2 to 4, we can determine the starting and ending position of the curve by evaluating the blending functions at these points to obtain

$$P_{\text{start}} = \frac{1}{2}(P_0 + P_1), \quad P_{\text{end}} = \frac{1}{2}(P_2 + P_3)$$

Thus, the curve starts at the midposition between the first two control points and ends at the midposition between the last two control points.

We can also determine the parametric derivatives at the starting and ending positions of the curve. Taking the derivatives of the blending functions and substituting the endpoint values for parameter  $u$ , we find that

$$P'_{\text{start}} = P_1 - P_0, \quad P'_{\text{end}} = P_3 - P_2$$

The parametric slope of the curve at the start position is parallel to the line joining the first two control points, and the parametric slope at the end of the curve is parallel to the line joining the last two control points.

An example plot of the quadratic periodic B-spline curve is given in Figure 30 for four control points selected in the  $xy$  plane.

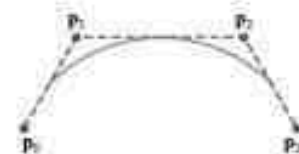


FIGURE 30  
A quadratic, periodic B-spline fitted to four control points in the  $xy$  plane.

In the preceding example, we noted that the quadratic curve starts between the first two control points and ends at a position between the last two control points. This result is valid for a quadratic periodic B-spline fitted to any number of distinct control points. In general, for higher-order polynomials, the start and end positions are each weighted averages of  $d - 1$  control points. We can pull a spline curve closer to any control-point position by specifying that position multiple times.

General expressions for the boundary conditions for periodic B-splines can be obtained by reparameterizing the blending functions so that parameter  $u$  is mapped onto the unit interval from 0 to 1. Beginning and ending conditions are then obtained at  $u = 0$  and  $u = 1$ .

### Cubic Periodic B-Spline Curves

Because cubic periodic B-splines are commonly used in graphics packages, we consider the formulation for this class of splines. Periodic splines are particularly useful for generating certain closed curves. For example, the closed curve in Figure 31 can be generated in sections by cyclically specifying four of the six control points for each section. Also, if the coordinate positions for three consecutive control points are identical, the curve passes through that point.

For cubic B-spline curves,  $d = 4$  and each blending function spans four sub-intervals of the total range of  $u$ . If we are to fit the cubic to four control points, then we could use the integer knot vector

$$(0, 1, 2, 3, 4, 5, 6, 7)$$

and recurrence relations 38 to obtain the periodic blending functions, as we did in the last section for quadratic periodic B-splines.

To derive the curve equations for a periodic, cubic B-spline, we consider an alternate formulation by starting with the boundary conditions and obtaining the blending functions normalized to the interval  $0 \leq u \leq 1$ . Using this formulation, we can also obtain the characteristic matrix easily. The boundary conditions for

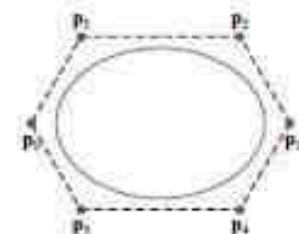


FIGURE 31  
A closed, periodic, piecewise cubic B-spline constructed using a cyclic specification of four control points for each curve section.

periodic cubic B-splines with four control points, labeled  $p_0$ ,  $p_1$ ,  $p_2$ , and  $p_3$ , are

$$\begin{aligned} P(0) &= \frac{1}{6}(p_2 + 4p_1 + p_0) \\ P(1) &= \frac{1}{6}(p_1 + 4p_2 + p_3) \\ P'(0) &= \frac{1}{2}(p_1 - p_0) \\ P'(1) &= \frac{1}{2}(p_2 - p_1) \end{aligned} \quad (41)$$

These boundary conditions are similar to those for cardinal splines: Curve sections are defined with four control points, and parametric derivatives (slopes) at the beginning and end of each curve section are parallel to the chords joining adjacent control points. The B-spline curve section starts at a position near  $p_1$  and ends at a position near  $p_2$ .

A matrix formulation for a cubic periodic B-spline with four control points can then be written as

$$P(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot M_B \cdot \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} \quad (42)$$

where the B-spline matrix for periodic cubic polynomials is

$$M_B = \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \quad (43)$$

This matrix can be obtained by solving for the coefficients in a general cubic polynomial expression, using the specified four boundary conditions.

We can also modify the B-spline equations to include a tension parameter  $t$  (as in cardinal splines). The matrix for the periodic cubic B-spline, with tension parameter  $t$ , is

$$M_{B_t} = \frac{1}{6} \begin{bmatrix} -t & 12 - 9t & 9t - 12 & t \\ 3t & 12t - 18 & 18 - 15t & 0 \\ -3t & 0 & 3t & 0 \\ t & 6 - 2t & t & 0 \end{bmatrix} \quad (44)$$

which reduces to  $M_B$  when  $t = 1$ .

We obtain the periodic cubic B-spline blending functions over the parameter range from 0 to 1 by expanding the matrix representation into polynomial form. For example, using the tension value  $t = 1$ , we have

$$\begin{aligned} B_{0,3}(u) &= \frac{1}{6}(1-u)^3, & 0 \leq u \leq 1 \\ B_{1,3}(u) &= \frac{1}{6}(3u^3 - 6u^2 + 4) \\ B_{2,3}(u) &= \frac{1}{6}(-3u^3 + 3u^2 + 3u + 1) \\ B_{3,3}(u) &= \frac{1}{6}u^3 \end{aligned} \quad (45)$$

### Open Uniform B-Spline Curves

This class of B-splines is a cross between uniform B-splines and nonuniform B-splines. Sometimes it is treated as a special type of uniform B-spline, and sometimes it is considered to be in the nonuniform B-spline classification. For the **open uniform B-splines**, or simply **open B-splines**, the knot spacing is uniform except at the ends, where knot values are repeated  $d$  times.

Here are two examples of open, uniform, integer knot vectors, each with a starting value of 0:

$$\begin{aligned} (0, 0, 1, 2, 3, 3) & \quad \text{for } d = 2 \text{ and } n = 3 \\ (0, 0, 0, 0, 1, 2, 2, 2) & \quad \text{for } d = 4 \text{ and } n = 4 \end{aligned} \quad (46)$$

We can normalize these knot vectors to the unit interval from 0 to 1 as

$$\begin{aligned} (0, 0, 0.33, 0.67, 1, 1) & \quad \text{for } d = 2 \text{ and } n = 3 \\ (0, 0, 0, 0, 0.5, 1, 1, 1) & \quad \text{for } d = 4 \text{ and } n = 4 \end{aligned} \quad (47)$$

For any values of parameters  $d$  and  $n$ , we can generate an open uniform knot vector with integer values using the calculations

$$u_j = \begin{cases} 0 & \text{for } 0 \leq j < d \\ j - d + 1 & \text{for } d \leq j \leq n \\ n - d + 2 & \text{for } j > n \end{cases} \quad (48)$$

for values of  $j$  ranging from 0 to  $n + d$ . With this assignment, the first  $d$  knots are assigned the value 0, and the last  $d$  knots have the value  $n - d + 2$ .

Open uniform B-splines have characteristics that are very similar to Bézier splines. In fact, when  $d = n + 1$  (degree of the polynomial is  $n$ ), open B-splines reduce to Bézier splines, and all knot values are either 0 or 1. For example, with a cubic open B-spline ( $d = 4$ ) and four control points, the knot vector is

$$(0, 0, 0, 0, 1, 1, 1, 1)$$

The polynomial curve for an open B-spline connects the first and last control points. Also, the parametric slope of the curve at the first control point is parallel to the straight line formed by the first two control points, and the parametric slope at the last control point is parallel to the line defined by the last two control points. Thus, the geometric constraints for matching curve sections are the same as for Bézier curves.

As with Bézier curves, specifying multiple control points at the same coordinate position pulls any B-spline curve closer to that position. Since open B-splines start at the first control point and end at the last control point, a closed curve can be generated by setting the first and last control points at the same coordinate position.

#### EXAMPLE 2 Open Uniform, Quadratic B-Splines

From conditions 48 with  $d = 3$  and  $n = 4$  (five control points), we obtain the following eight values for the knot vector:

$$\{u_0, u_1, u_2, u_3, u_4, u_5, u_6, u_7\} = \{0, 0, 0, 1, 2, 3, 3, 3\}$$

The total range of  $u$  is divided into seven subintervals, and each of the five blending functions  $B_{k,3}$  is defined over three subintervals, starting at knot position  $u_k$ . Thus  $B_{0,3}$  is defined from  $u_0 = 0$  to  $u_3 = 1$ ,  $B_{1,3}$  is defined

from  $u_1 = 0$  to  $u_4 = 2$ , and  $B_{4,3}$  is defined from  $u_4 = 2$  to  $u_7 = 3$ . Explicit polynomial expressions are obtained for the blending functions from recurrence relations 36 as

$$\begin{aligned}
 B_{2,3}(u) &= (1-u)^2 & 0 \leq u < 1 \\
 B_{1,3}(u) &= \begin{cases} \frac{1}{2}u(4-3u) & 0 \leq u < 1 \\ \frac{1}{2}(2-u)^2 & 1 \leq u < 2 \end{cases} \\
 B_{2,3}(u) &= \begin{cases} \frac{1}{2}u^2 & 0 \leq u < 1 \\ \frac{1}{2}u(2-u) + \frac{1}{2}(u-1)(3-u) & 1 \leq u < 2 \\ \frac{1}{2}(3-u)^2 & 2 \leq u < 3 \end{cases} \\
 B_{3,3}(u) &= \begin{cases} \frac{1}{2}(u-1)^2 & 1 \leq u < 2 \\ \frac{1}{2}(3-u)(3u-5) & 2 \leq u < 3 \end{cases} \\
 B_{4,3}(u) &= (u-2)^2 & 2 \leq u < 3
 \end{aligned}$$

Figure 32 shows the shape of these five blending functions. The local features of B-splines are again demonstrated. Blending function  $B_{1,3}$  is nonzero only in the subinterval from 0 to 1, so the first control point influences the curve only in this interval. Similarly, function  $B_{4,3}$  is 0 outside the interval from 2 to 3, and the position of the last control point does not affect the shape of the beginning and middle parts of the curve.

Matrix formulations for open B-splines are not generated as conveniently as they are for periodic uniform B-splines. This is due to the multiplicity of knot values at the beginning and end of the knot vector.

### Nonuniform B-Spline Curves

For this class of splines, we can specify any values and intervals for the knot vector. With nonuniform B-splines, we can choose multiple internal knot values and unequal spacing between the knot values. Some examples are

$$\begin{aligned}
 &\{0, 1, 2, 3, 3, 4\} \\
 &\{0, 2, 2, 3, 3, 6\} \\
 &\{0, 0, 0, 1, 1, 3, 3, 3\} \\
 &\{0, 0.2, 0.6, 0.9, 1.0\}
 \end{aligned}$$

Nonuniform B-splines provide increased flexibility in controlling a curve shape. With unequally spaced intervals in the knot vector, we obtain different shapes for the blending functions in different intervals, which can be used in designing the spline features. By increasing knot multiplicity, we can produce subtle variations in the curve path and introduce discontinuities. Multiple knot values also reduce the continuity by 1 for each repeat of a particular value.

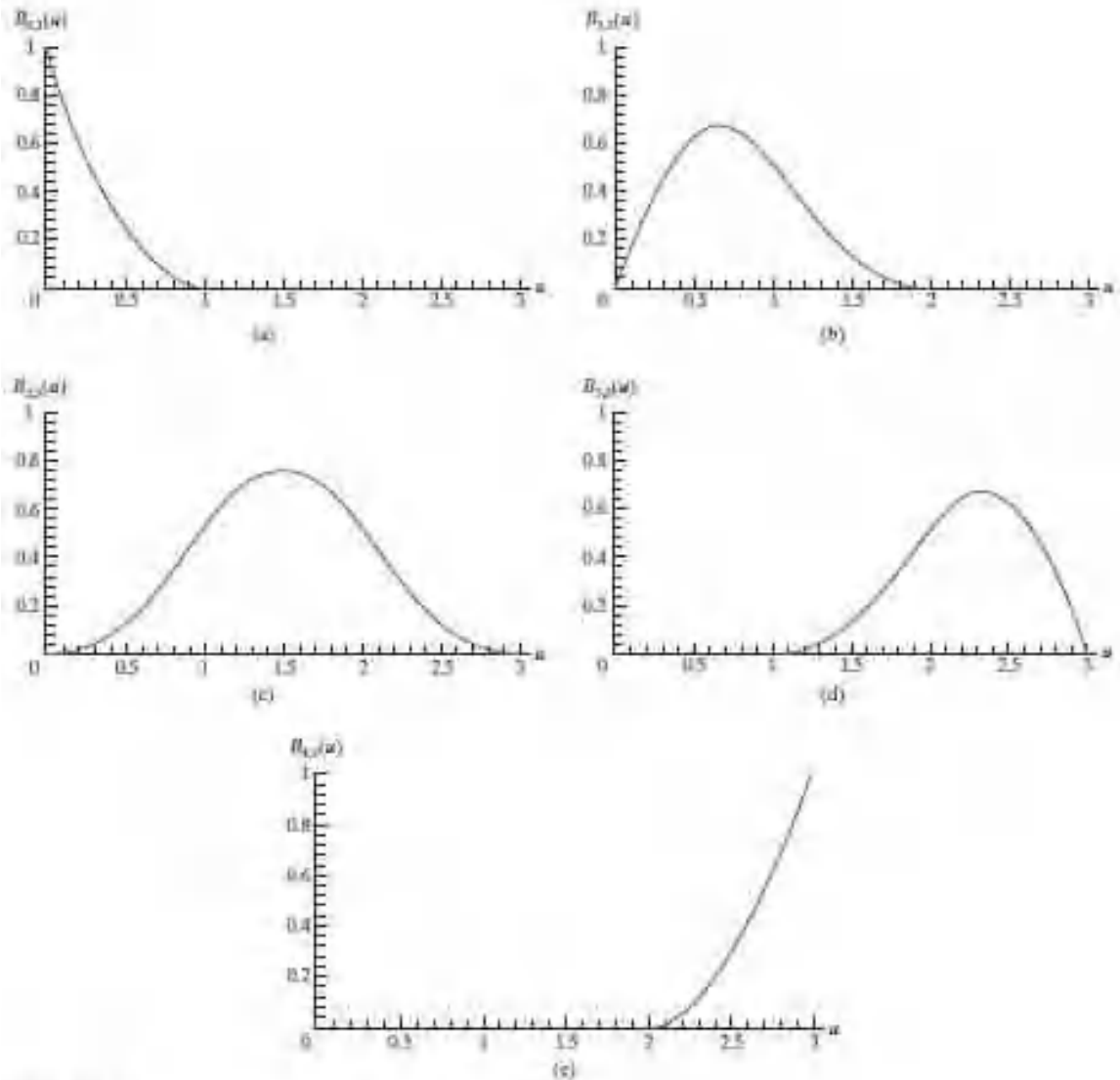


FIGURE 22.  
Open, uniform B-spline blending functions for  $n = 4$  and  $d = 3$ .

We obtain the blending functions for a nonuniform B-spline using methods similar to those discussed for uniform and open B-splines. Given a set of  $n + 1$  control points, we set the degree of the polynomial and select the knot values. Then, using the recurrence relations, we could either obtain the set of blending functions or evaluate curve positions directly for the display of the curve. Graphics packages often restrict the knot intervals to be either 0 or 1 to reduce computations. A set of characteristic matrices can then be stored and used to compute values along the spline curve without evaluating the recurrence relations for each curve point to be plotted.

## B-SPLINE SURFACES

Formulation of a B-spline surface is similar to that for Bézier splines. We can obtain a vector point function over a B-spline surface using the tensor product of B-spline blending functions in the form

$$P(u, v) = \sum_{k=0}^{n_u} \sum_{l=0}^{n_v} P_{k,l} B_{k,d_u}(u) B_{l,d_v}(v) \quad (49)$$



where the vector values for  $\mathbf{p}_k, k_v$  specify the positions of the  $(n_u + 1)$  by  $(n_v + 1)$  control points. B-spline surfaces exhibit the same properties as those of their component B-spline curves. A surface can be constructed from selected values for degree parameters  $du$  and  $dv$ , which set the degrees for the orthogonal surface polynomials at  $du - 1$  and  $dv - 1$ . For each surface parameter  $u$  and  $v$ , we also select values for the knot vectors, which determines the parameter range for the blending functions.

## COLOR MODELS AND COLOR APPLICATIONS.

Our discussions of color up to this point have concentrated on methods involving red, green, and blue (RGB) components, which we use for generating displays on video monitors. Several other color descriptions are useful as well in computer-graphics applications. Some methods are used to describe color output on printers and plotters, some are used for transmitting and storing color information, and others are used to provide a more intuitive color-parameter interface to a program.

### 1 Properties of Light

Light exhibits many different characteristics, and we describe the properties of light in different ways in different contexts. Physically, we can characterize light as radiant energy, but we also need other concepts to describe our perception of light.

#### The Electromagnetic Spectrum

In physical terms, color is electromagnetic radiation within a narrow frequency band. Some of the other frequency groups in the electromagnetic spectrum are referred to as radio waves, microwaves, infrared waves, and X-rays. Figure 1 shows the approximate frequency ranges for these various aspects of electromagnetic radiation.

Each frequency value within the visible region of the electromagnetic spectrum corresponds to a distinct spectral color. At the low-frequency end (approximately  $3.8 \times 10^{14}$  hertz) are the red colors, and at the high-frequency end (approximately  $7.9 \times 10^{14}$  hertz) are the violet colors. Actually, the human eye is sensitive to some frequencies into the infrared and ultraviolet bands. Spectral colors range from shades of red through orange and yellow, at the low-frequency end, to shades of green, blue, and violet at the high end.

In the wave model of electromagnetic radiation, light can be described as oscillating transverse electric and magnetic fields propagating through space. The electric and magnetic fields are oscillating in directions that are perpendicular to each other and to the direction of propagation. For each spectral color, the rate of oscillation of the field magnitude is given by the frequency  $f$ . Figure 2

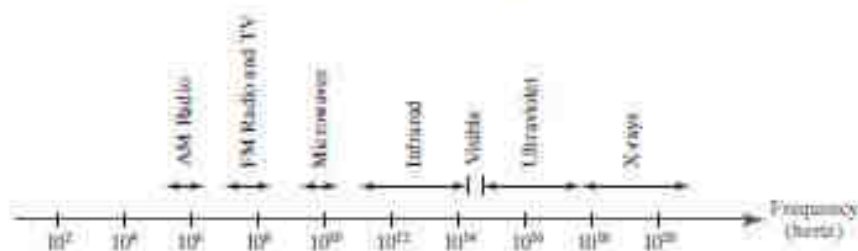
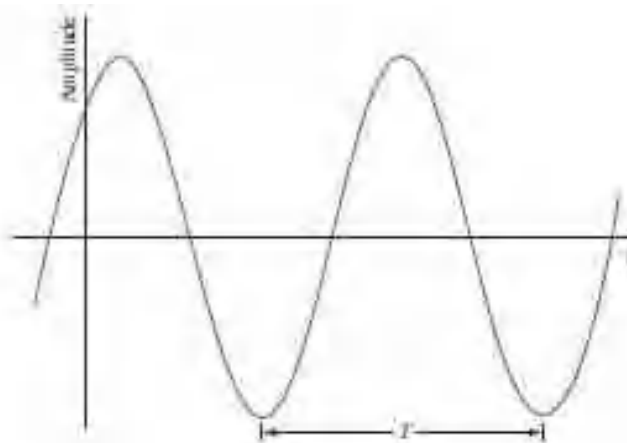


FIGURE 1  
Electromagnetic spectrum.

**FIGURE 2**  
Time variations for the amplitude of the electric field for one-frequency component of a plane-polarized electromagnetic wave. The time between two consecutive amplitude peaks or two consecutive amplitude minima is called the period of the wave.



illustrates the time-varying oscillations for the magnitude of the electric field within one plane. The time between any two consecutive positions on the wave that have the same amplitude is called the *period* ( $T$ ) of the wave, which is the inverse of the frequency (i.e.,  $T=1/f$ ). And the distance that the wave has traveled from the beginning of one oscillation to the beginning of the next oscillation is called the *wavelength* ( $\lambda$ ). For one spectral color (a monochromatic wave), the wavelength and frequency are inversely proportional to each other, with the proportionality constant as the speed of light ( $c$ ):

$$c = \lambda f \quad (1)$$

Frequency for each spectral color is a constant for all materials, but the speed of light and the wavelength are material dependent. In a vacuum, the speed of light is very nearly  $c = 3 \times 10^{10}$  cm/sec. Light wavelengths are very small, so length units for designating spectral colors are usually given in angstroms ( $1 \text{ \AA} = 10^{-8}$  cm) or in nanometers ( $1 \text{ nm} = 10^{-7}$  cm). An equivalent term for nanometer is *millimicron*. Light at the low-frequency end of the spectrum (red) has a wavelength of approximately 780 nanometers (nm), and the wavelength at the other end of the spectrum (violet) is about 380 nm. Because wavelength units are somewhat more convenient to deal with than frequency units, spectral colors are typically specified in terms of the wavelength values in a vacuum.

A light source such as the sun or a standard household light bulb emits all frequencies within the visible range to produce white light. When white light is incident upon an opaque object, some frequencies are reflected and some are absorbed. The combination of frequencies present in the reflected light determines what we perceive as the color of the object. If low frequencies are predominant in the reflected light, the object is described as red. In this case, we say that the perceived light has a **dominant frequency** (or **dominant wavelength**) at the red end of the spectrum. The dominant frequency is also called the **hue**, or simply the **color**, of the light.

### Psychological Characteristics of Color

Other properties besides frequency are needed to characterize our perception of light. When we view a source of light, our eyes respond to the color (or dominant frequency) and two other basic sensations. One of these we call the **brightness**, which corresponds to the total light energy and can be quantified as the luminance of the light. The third perceived characteristic is called the **purity**, or the **saturation**, of the light. Purity describes how close a light appears to be to a pure spectral color, such as red. Pastels and pale colors have low purity (low saturation) and they appear to be nearly white. Another term, **chromaticity**, is used to refer collectively to the two properties describing color characteristics: purity and dominant frequency (hue).

Radiation emitted by a white light source has an energy distribution that can be represented over the visible frequencies as in Figure 3. Each frequency component within the range from red to violet contributes more or less equally to the total energy, and the color of the source is described as white. When a dominant frequency

is present, the energy distribution for the source takes a form such as that in

Figure 4. We would describe this light as a red color (the dominant frequency), with a relatively high value for the purity. The energy density of the dominant

light component is labeled as  $E_D$  in this

figure, and the contributions from the

other frequencies produce white light of energy density  $E_W$ . We can calculate the

brightness of the source as the area under the curve, which gives the total energy

density emitted. Purity (saturation) depends on the difference between  $E_D$  and  $E_W$ . The larger the energy  $E_D$  of

the dominant frequency compared to the

white-light component  $E_W$ , the higher the purity of the light. We have a purity of

100 percent when  $E_W = 0$  and a purity of 0 percent when  $E_W = E_D$ .

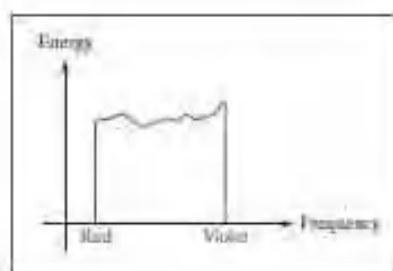


FIGURE 3  
Energy distribution for a white light source.

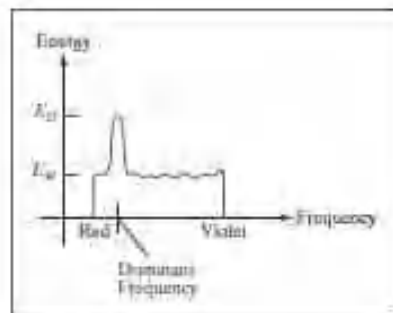


FIGURE 4  
Energy distribution for a light source with a dominant frequency near the red end of the frequency range.

## Color Models

Any method for explaining the properties or behavior of color within some particular context is called a **color model**. No single model can explain all aspects of color, so we make use of different models to help describe different color characteristics.

### Primary Colors

When we combine the light from two or more sources with different dominant frequencies, we can vary the amount (intensity) of light from each source to generate a range of additional colors. This represents one method for forming a color model. The hues that we choose for the sources are called the **primary colors**, and the **color gamut** for the model is the set of all colors that we can produce from the primary colors. Two primaries that produce white are referred to as **complementary colors**. Examples of complementary color pairs are red and cyan, green and magenta, and blue and yellow.

No finite set of real primary colors can be combined to produce all possible visible colors. Nevertheless, three primaries are sufficient for most purposes, and colors not in the color gamut for a specified set of primaries can still be described using extended methods. Given a set of three primary colors, we can characterize any fourth color using color-mixing processes. Thus, a mixture of one or two of the primaries with the fourth color can be used to match some combination of the remaining primaries. In this extended sense, a set of three primary colors can be considered to describe all colors. Figure 5 shows a set of *color-matching functions* for three primaries and the amount of each needed to produce any spectral color. The curves plotted in Figure 5 were obtained by averaging the judgments of a large number of observers. Colors in the vicinity of 500 nm can be matched only by “subtracting” an amount

of red light from a combination of blue and green

lights. This means that a color around 500 nm is described only by combining

that color with an amount of red light to produce the blue-green combination

specified in the diagram. Thus, an RGB color monitor cannot display colors in the neighborhood of 500 nm.

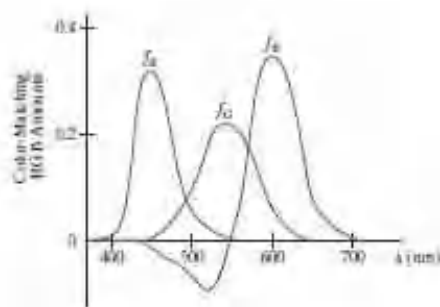


FIGURE 5  
Three color-matching functions for displaying spectral frequencies within the approximate range from 400 nm to 700 nm.

## Intuitive Color Concepts

An artist creates a color painting by mixing color pigments with white and black pigments to form the various shades, tints, and tones in the scene. Starting with the pigment for a “pure color” (“pure hue”), the artist adds a black pigment to produce different **shades** of that color. The more black pigment, the darker the shade. Similarly, different **tints** of the color are obtained by adding a white pigment to the original color, making it lighter as more white is added. **Tones** of the color are produced by adding both black and white pigments.

To many, these color concepts are more intuitive than describing a color as a set of three numbers that give the relative proportions of the primary colors. It is generally much easier to think of creating a pastel red color by adding white to pure red and producing a dark blue color by adding black to pure blue. Therefore, graphics packages providing color palettes to a user often employ two or more color models. One model provides an intuitive color interface for the user, and the others describe the color components for the output devices.

## 3 Standard Primaries and the Chromaticity Diagram

Because no finite set of light sources can be combined to display all possible colors, three standard primaries were defined in 1931 by the International Commission on Illumination, referred to as the CIE (Commission Internationale de l'Éclairage). The three standard primaries are imaginary colors. They are defined

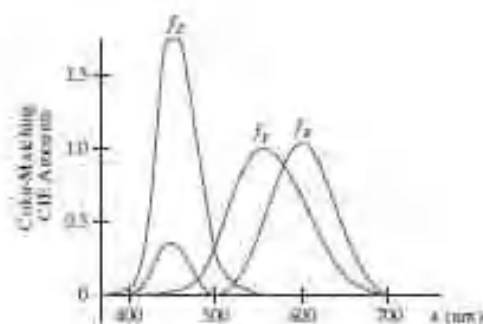


FIGURE 6  
The three color-matching functions for the CIE primaries.

mathematically with positive color-matching functions (Figure 6) that

specify the amount of each primary needed to describe any spectral color. This provides an international standard definition for all colors, and the CIE primaries eliminate negative-value color-matching and other problems associated with selecting a set of real primaries.

### The XYZ Color Model

The set of CIE primaries is generally referred to as the XYZ color model, where parameters  $X$ ,  $Y$ , and  $Z$  represent the amount of each CIE primary needed to produce a selected color. Thus, a color is described with the XYZ model in the same way that we described a color using the RGB model.

In the three-dimensional XYZ color space, we represent any color  $C(\lambda)$  as

$$C(\lambda) = (X, Y, Z) \quad (2)$$

where  $X$ ,  $Y$ , and  $Z$  are calculated from the color-matching functions (Figure 6):

$$X = k \int_{\text{visible}} f_X(\lambda) I(\lambda) d\lambda$$

$$Y = k \int_{\text{visible}} f_Y(\lambda) I(\lambda) d\lambda \quad (3)$$

$$Z = k \int_{\text{visible}} f_Z(\lambda) I(\lambda) d\lambda$$

Parameter  $k$  in these calculations has the value 683 lumens/watt, where lumen is a unit of measure for light radiation per unit solid angle from a "standard" point light source (once called a *candle*). The function  $I(\lambda)$  represents the spectral radiance, which is the selected light intensity in a particular direction, and the color-matching function  $f_Y$  is chosen so that parameter  $Y$  is the luminance for that color. Luminance values are normally adjusted to the range from 0 to 100.0, where 100.0 represents the luminance of white light.

Any color can be represented in the XYZ color space as an additive combination of the primaries using unit vectors  $X, Y, Z$ . Thus, we can write Equation 2 as

$$C(\lambda) = XX + YX + ZX \quad (4)$$

### Normalized XYZ Values

In discussing color properties, it is convenient to normalize the amounts in Equation 3 against the sum  $X + Y + Z$ , which represents the total light energy. Normalized amounts are thus calculated as

$$x = \frac{X}{X+Y+Z}, \quad y = \frac{Y}{X+Y+Z}, \quad z = \frac{Z}{X+Y+Z} \quad (5)$$

Because  $x + y + z = 1$ , any color can be represented with just the  $x$  and  $y$  amounts. Also, we have normalized against total energy, so parameters  $x$  and  $y$  depend only on hue and purity and are called the **chromaticity values**. However, the  $x$  and  $y$  values alone do not allow us to describe all properties of the color completely, and we cannot obtain the amounts  $X, Y$ , and  $Z$ . Therefore, a complete description of a color is typically given with three values:  $x, y$ , and the luminance  $Y$ . The remaining CIE amounts are then calculated as

$$X = \frac{x}{y}Y, \quad Z = \frac{z}{y}Y \quad (6)$$

where  $z = 1 - x - y$ . Using chromaticity coordinates  $(x, y)$ , we can represent all colors on a two-dimensional diagram.

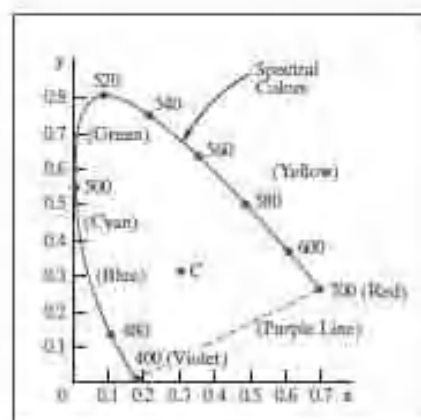


FIGURE 7  
CIE chromaticity diagram for the spectral colors  
from 400 nm to 700 nm.



## The CIE Chromaticity Diagram

When we plot the normalized amounts  $x$  and  $y$  for colors in the visible spectrum, we obtain the tongue-shaped curve shown in Figure 7. This curve is called the **CIE chromaticity diagram**. Points along the curve are the **spectral colors** (pure colors). The line joining the red and violet spectral points, referred to as the **purple line**, is not part of the spectrum. Interior points represent all possible visible color combinations. Point  $C$  in the diagram corresponds to the white-light position. Actually, this point is plotted for a white light source known as **illuminant C**, which is used as a standard approximation for average daylight.

Luminance values are not available in the chromaticity diagram because of normalization. Colors with different luminance but with the same chromaticity map to the same point. The chromaticity diagram is useful for:

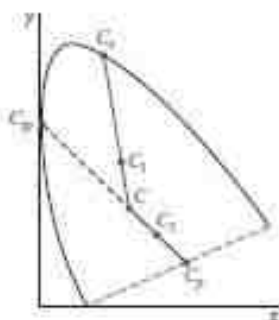
- Comparing color gamuts for different sets of primaries.
- Identifying complementary colors.
- Determining purity and dominant wavelength for a given color.

### Color Gamuts

We identify color gamuts on the chromaticity diagram as straight-line segments or polygon regions. All colors along the straight line joining points  $C_1$  and  $C_2$  in Figure 8 can be obtained by mixing appropriate amounts of the colors  $C_1$  and  $C_2$ . If a greater proportion of  $C_1$  is used, the resultant color is closer to  $C_1$  than to  $C_2$ . The color gamut for three points, such as  $C_3$ ,  $C_4$ , and  $C_5$  in Figure 8, is a triangle with vertices at the three color positions. These three primaries can generate only the colors inside or on the bounding edges of the triangle. Thus, the chromaticity diagram helps us to understand why no set of three primaries can be additively combined to generate all colors, because no triangle within the diagram can encompass all colors. Color gamuts for video monitors and hard-copy devices are compared conveniently on the chromaticity diagram.

### Complementary Colors

Because the color gamut for two points is a straight line, complementary colors must be represented on the chromaticity diagram as two points on opposite sides of  $C$  and collinear with  $C$ , as in Figure 9. The distances of the two colors  $C_1$  and  $C_2$  to  $C$  determine the amounts of each needed to produce white light.



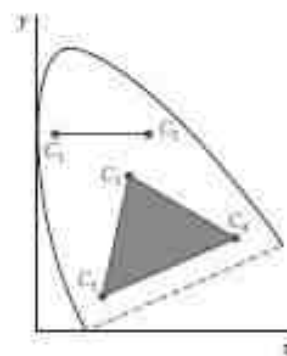
**FIGURE 10**  
Determining dominant wavelength and purity using the chromaticity diagram.

### Dominant Wavelength

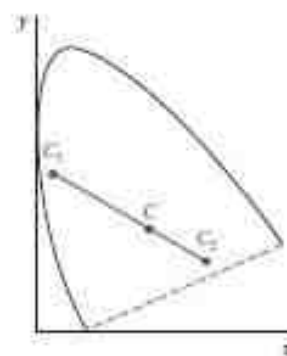
To determine the dominant wavelength of a color, we draw a straight line from  $C$  through that color point to a spectral color on the chromaticity curve. The spectral color  $C_2$  in Figure 10 is the dominant wavelength for color  $C_1$  in this diagram. Thus, color  $C_1$  can be represented as a combination of white light  $C$  and the spectral color  $C_2$ . This method for determining dominant wavelength will not work for color points that are between  $C$  and the purple line. Drawing a line from  $C$  through point  $C_2$  in Figure 10 takes us to point  $C_3$  on the purple line, which is not in the visible spectrum. In this case, we take the complement of  $C_3$  on the spectral curve, which is the point  $C_3'$ , as the dominant wavelength. Colors such as  $C_2$  in this diagram have spectral distributions with subtractive dominant wavelengths. We can describe such colors by subtracting the spectral dominant wavelength from white light.

### Purity

For a color point such as  $C_1$  in Figure 10, we determine the purity as the relative distance of  $C_1$  from  $C$  along the straight line joining  $C$  to  $C_2$ . If  $d_1$  denotes the distance from  $C$  to  $C_1$  and  $d_2$  is the distance from  $C$  to  $C_2$ , we can represent purity as the ratio  $d_1/d_2$ . Color  $C_1$  in this figure is about 25 percent pure, because it is situated at about one-fourth the total distance from  $C$  to  $C_2$ . At position  $C_2$ , the color point would be 100 percent pure.



**FIGURE 8**  
Color gamuts defined on the chromaticity diagram for a two-color and a three-color system of primaries.



**FIGURE 9**  
Representing complementary colors on the chromaticity diagram.

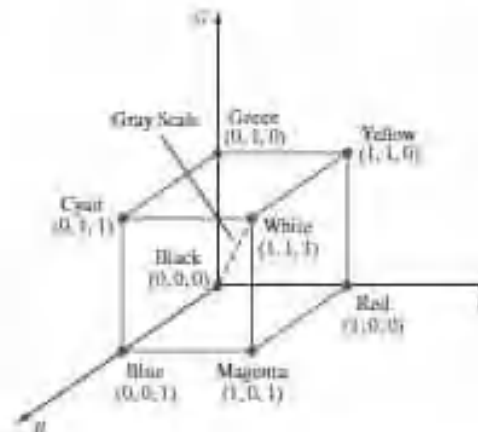
## 4 The RGB Color Model

According to the *tristimulus theory of vision*, our eyes perceive color through the stimulation of three visual pigments in the cones of the retina. One of the pigments is most sensitive to light with a wavelength of about 630 nm (red), another has its peak sensitivity at about 530 nm (green), and the third pigment is most receptive



to light with a wavelength) of about 450 nm (blue). By comparing intensities in a light source, we perceive the color of the light. This theory of vision is the basis for displaying color output on a video monitor using the three primaries red, green, and blue, which is referred to as the *RGB color model*.

We can represent this model using the unit cube defined on *R*, *G*, and *B* axes, as shown in Figure 11. The origin represents black and the diagonally opposite



**FIGURE 11**  
the RGB color model. Any color within the unit cube can be described as an additive combination of the three primary colors.

**TABLE 1**

**RGB (*x*, *y*) Chromaticity Coordinates**

	NTSC Standard	CIE Model	Approx. Color Monitor Values
R	(0.670, 0.330)	(0.735, 0.265)	(0.628, 0.346)
G	(0.210, 0.710)	(0.274, 0.717)	(0.268, 0.588)
B	(0.140, 0.080)	(0.167, 0.089)	(0.150, 0.070)

vertex, with coordinates (1, 1, 1), is white. Vertices of the cube on the axes represent the primary colors, and the remaining vertices are the complementary color points for each of the primary colors.

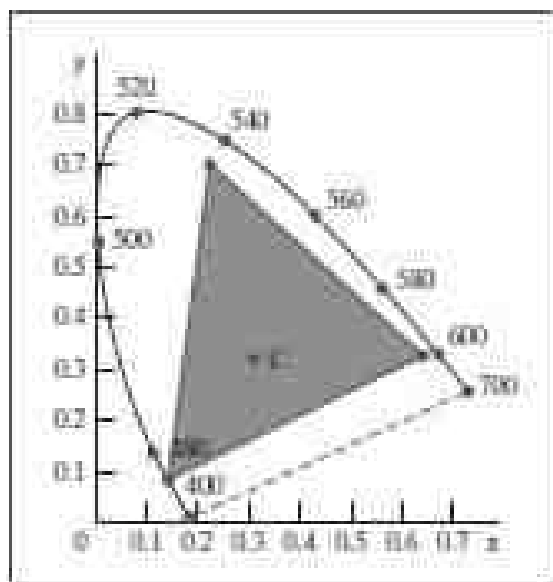
As with the XYZ color system, the RGB color scheme is an additive model. Each color point within the unit cube can be represented as a weighted vector sum of the primary colors, using unit vectors *R*, *G*, and *B*:

$$C(\lambda) = (R, G, B) = R\mathbf{R} + G\mathbf{G} + B\mathbf{B} \quad (7)$$

where parameters *R*, *G*, and *B* are assigned values in the range from 0 to 1.0. For example, the magenta vertex is obtained by adding maximum red and blue values to produce the triple (1, 0, 1), and white at (1, 1, 1) is the sum of the maximum values for red, green, and blue. Shades of gray are represented along the main diagonal of the cube from the origin (black) to the white vertex. Points along this diagonal have equal contributions from each primary color, and a gray shade halfway between black and white is represented as (0.5, 0.5, 0.5). The color graduations along the front and top planes of the RGB cube are illustrated in Color Plate 22.

Chromaticity coordinates for the National Television System Committee

(NTSC) standard RGB phosphors are listed in Table 1. Also listed are the RGB chromaticity coordinates within the CIE color model and the approximate values used for phosphors in color monitors. Figure 12 shows the approximate color gamut for the NTSC standard RGB primaries.



**FIGURE 12**  
The RGB color gamut for NTSC chromaticity coordinates. Illuminant C is at position (0.310, 0.316), with a luminance value of  $Y = 100.0$ .

### The YIQ and Related Color Models

Although an RGB graphics monitor requires separate signals for the red, green, and blue components of an image, a television monitor uses a composite signal.

NTSC color encoding for forming the composite video signal is called the *YIQ color model*.

#### The YIQ Parameters

In the YIQ color model, parameter  $Y$  is the same as the  $Y$  component in the CIE XYZ color space. Luminance (brightness) information is conveyed by the  $Y$  parameter, while chromaticity information (hue and purity) is incorporated into the  $I$  and  $Q$  parameters. A combination of red, green, and blue is chosen for the  $Y$  parameter to yield the standard luminosity curve. Because  $Y$  contains the luminance information, black-and-white television monitors use only the  $Y$  signal.

Parameter  $I$  contains orange-cyan color information that provides the flesh-tone shading, and parameter  $Q$  carries green-magenta color information.

The NTSC composite color signal is designed to provide information in a form that can be received by black-and-white television monitors, which obtain grayscale information for a picture within a 6-MHz bandwidth. Thus, the YIQ information is also encoded within a 6-MHz bandwidth, but the luminance and chromaticity values are encoded on separate analog signals. In this way, the luminance signal is unchanged for black-and-white monitors, and the color information is simply added within the same bandwidth. Luminance information, the  $Y$  value, is conveyed as an amplitude modulation on a carrier signal with a bandwidth of about 4.2 MHz.

Chromaticity information, the  $I$  and  $Q$  values, is combined on a second carrier signal that has a bandwidth of about 1.8 MHz.

The parameter names  $I$  and  $Q$  refer to the modulation methods used to encode the color information on this carrier. An amplitude-modulation encoding (the “in-phase” signal) transmits the  $I$  value, using about 1.3 MHz of the bandwidth.

And a phase-modulation encoding (the “quadrature” signal), using about 0.5 MHz, carries the  $Q$  value.

Luminance values are encoded at a higher precision in the NTSC signal (4.2 MHz bandwidth) than the chromaticity values (1.8 MHz bandwidth), because we can detect small brightness changes more easily compared to small color changes. However, the lower precision for the chromaticity encoding does result in some degradation of the color quality for an NTSC picture.

$$Y = 0.299 R + 0.587 G + 0.114 B$$

$$I = R - Y \quad (8)$$

$$Q = B - Y$$

### Transformations Between RGB and YIQ Color Spaces

An RGB color is converted to a set of YIQ values using an NTSC encoder that implements the calculations in Equation 9 and modulates the carrier signals.

The conversion from RGB space to YIQ space is accomplished using the following transformation matrix:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.701 & -0.587 & -0.114 \\ -0.299 & -0.587 & 0.587 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (9)$$

Conversely, an NTSC video signal is converted to RGB color values using an NTSC decoder, which first separates the video signal into the YIQ components, and then converts the YIQ values to RGB values. The conversion from YIQ space to RGB space is accomplished with the inverse of transformation

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 1.000 & 0.000 \\ 1.000 & -0.500 & -0.194 \\ 1.000 & 0.000 & 1.000 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix} \quad (10)$$

### The YUV and YCrCb Systems

Because of the lower bandwidth assigned to the chromaticity information in the NTSC composite analog video signal, the color quality of an NTSC picture is somewhat impaired. Therefore, variations of the YIQ encoding have been developed to improve the color quality of video transmissions. One such encoding is the YUV set of color parameters, which provides the composite color information for video transmissions by systems such as Phase Alternation Line (PAL) Broadcasting, used in most of Europe, as well as Africa, Australia, and Eurasia. Another variation of YIQ is the digital encoding called YCrCb. This color representation is used for digital

video transformations, and it is incorporated into various graphicsfile formats, such as the JPEG syste .

## 6 The CMY and CMYK Color Models

A video monitor displays color patterns by combining light that is emitted fromthe screen phosphors, which is an additive process. However, hard-copy devices,such as printers and plotters, produce a color picture by coating a paper withcolor pigments. We see the color patterns on the paper by reflected light, whichis a subtractive process.

### The CMY Parameters

A subtractive color model can be formed with the three primary colors cyan,magenta, and yellow. As we have noted, cyan can be described as a combinationof green and blue. Therefore, when white light is reflected from cyancoloredink, the reflected light contains only the green and blue components,and the red component is absorbed, or subtracted, by the ink. Similarly, magenta ink subtracts the green component from incident light, and yellow subtracts theblue component. A unit cube representation for the CMY model is illustrated inFigure 13.

In the CMY model, the spatial position (1,1,1) represents black, becauseall components of the incident light are subtracted. The origin represents whitelight. Equal amounts of each of the primary colors produce shades of gray alongthe main diagonal of the cube. A combination of cyan and magenta ink producesblue light, because the red and green components of the incident light areabsorbed. Similarly, a combination of cyan and yellow ink produces green light,and a combination of magenta and yellow ink yields red light.

The CMY printing process often uses a collection of four ink dots, which arearranged in a close pattern somewhat as an RGB monitor uses three phosphordots. Thus, in practice, the CMY color model is referred to as the CMYK model,where  $K$  is the black color parameter. One ink dot is used for each of the primarycolors (cyan, magenta, and yellow), and one ink dot is black. A black dot isincluded because reflected light fromthe cyan, magenta, and yellow inks typicallyproduce only shades of gray. Some plotters produce different color combinationsby spraying the ink for the three primary colors over each other and allowingthem to mix before they dry. For black-and-white or grayscale printing, only theblack ink is used.

### Transformations Between CMY and RGB Color Spaces

We can express the conversion from an RGB representation to a CMY representationusing the following matrix transformation:

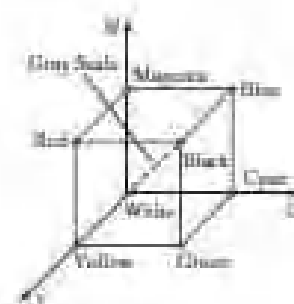


FIGURE 13  
The CMY color model. Positions within the unit cube are described by subtracting the specified amounts of the primary colors from white.

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (11)$$

where the white point in RGB space is represented as the unit column vector. And we convert from a CMY color representation to an RGB representation using the matrix transformation

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix} \quad (12)$$

In this transformation, the unit column vector represents the black point in the CMY color space.

For the conversion from RGB to the CMYK color space, we first set  $K = \max(R, G, B)$ . Then  $K$  is subtracted from each of  $C$ ,  $M$ , and  $Y$  in Equation 11. Similarly, for the transformation from CMYK to RGB, we first set

$$K = \min(R, G, B).$$

Then  $K$  is subtracted from each of  $R$ ,  $G$ , and  $B$  in Equation 12. In practice, these transformation equations are often modified to improve the printing quality for a particular system.

### The HSV Color Model

Interfaces for selecting colors often use a color model based on intuitive concepts, rather than a set of primary colors. We can give a color specification in an intuitive model by selecting a spectral color and the amounts of white and black that are to be added to that color to obtain different shades, tints, and tones (Section 2).

### The HSV Parameters

Color parameters in this model are called *hue* ( $H$ ), *saturation* ( $S$ ), and *value* ( $V$ ). We derive this three-dimensional color space by relating the HSV parameters to the directions in the RGB cube. If we imagine viewing the cube along the diagonal from the white vertex to the origin (black), we see an outline of the cube that has the hexagon shape shown in Figure 14. The boundary of the hexagon represents the various hues, and it is used as the top of the HSV hexcone (Figure 15).

In HSV space, saturation  $S$  is measured along a horizontal axis, and the

value parameter  $V$  is measured along a vertical axis through the center of the hexcone.

Hue is represented as an angle about the vertical axis, ranging from  $0^\circ$  at red through  $360^\circ$ .

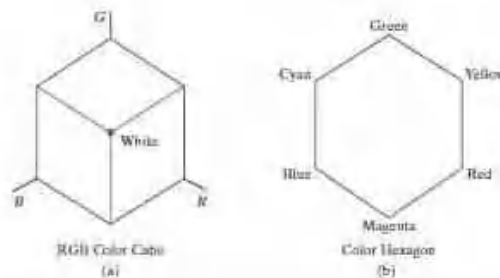


FIGURE 14 When the RGB color cube (a) is viewed along the diagonal from white to black, the color-cube outline is a hexagon (b).

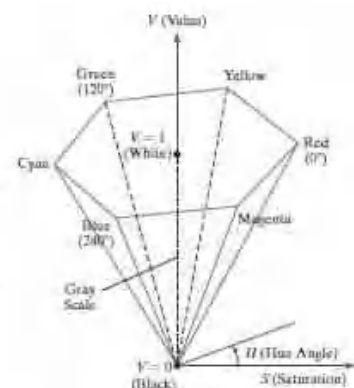


FIGURE 15 The HSV hexcone.



Vertices of the hexagon are separated by  $60^\circ$  intervals. Yellow is at  $60^\circ$ , green at  $120^\circ$ , and cyan (opposite the red point) is at  $H = 180^\circ$ . Complementary colors are  $180^\circ$  apart.

Saturation parameter  $S$  is used to designate the purity of a color. A pure color (spectral color) has the value  $S=1.0$ , and decreasing  $S$  values tend toward the grayscale line ( $S = 0$ ) at the center of the hexcone.

Value  $V$  varies from 0 at the apex of the hexcone to 1.0 at the top plane. The apex of the hexcone is the black point. At the top plane, colors have their maximum intensity. When  $V = 1.0$  and  $S = 1.0$ , we have the pure hues. Parameter values for the white point are  $V = 1.0$  and  $S = 0$ .

For most users, this is a more convenient model for selecting colors. Starting with a selection for a pure hue, which specifies the hue angle  $H$  and sets

$$V = S = 1.0,$$

we describe the color we want in terms of adding either white or black to the pure hue. Adding black decreases the setting for  $V$  while  $S$  is held constant. To get a dark blue, for instance,  $V$  could be set to 0.4 with  $S = 1.0$  and  $H = 240^\circ$ .

Similarly, when white is to be added to the selected hue, parameter  $S$  is decreased while keeping  $V$  constant. A light blue could be designated with  $S = 0.3$  while  $V = 1.0$  and  $H = 240^\circ$ . By adding some black and some white, we decrease both  $V$  and  $S$ . An interface for this model typically presents the HSV parameter choices in a color palette containing sliders and a color wheel.

### Selecting Shades, Tints, and Tones

Color regions for selecting shades, tints, and tones are represented in the cross-sectional plane of the HSV hexcone shown in Figure 16. Adding black to a spectral color decreases  $V$  along the side of the hexcone toward the black point.

Thus, various shades are represented with the values  $S=1.0$  and  $0.0 \leq V \leq 1.0$ . Adding white to spectral colors produces the tints across the top plane of the hexcone, where parameter values are  $V = 1.0$  and  $0 \leq S \leq 1.0$ . Various tones are obtained by

adding both black and white to spectral colors, which generates color points within the triangular cross-sectional area of the hexcone. The human eye can distinguish about 128 different hues and about 130 different tints (saturation levels). For each of these, a number of shades (value settings) can be detected, depending on the hue selected. About 23 shades are discernible with yellow colors, and about 16 different shades can be seen at the blue end of the spectrum. This means that we can distinguish about  $128 \times 130 \times 23 = 382,720$  different colors. For most graphics applications, 128 hues, 8 saturation levels, and 16 value settings are sufficient. With this range of parameters in the HSV color model, 16,384 colors are available to a

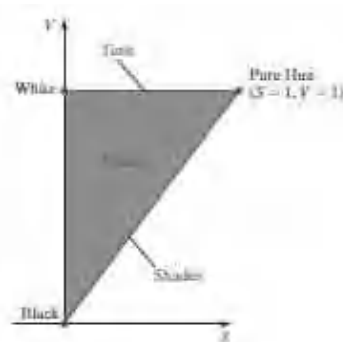


FIGURE 16  
Cross-section of the HSV hexcone, showing regions for shades, tints, and tones.

user. These color values can be stored in 14 bits per pixel, or we could use color-lookup tables and fewer bits per pixel.

### Transformations Between HSV and RGB Color Spaces

To determine the operations required for the transformations between the HSV and RGB spaces, we first consider how the HSV hexcone can be constructed from the RGB cube. The diagonal of the RGB cube from black (the origin) to white corresponds to the  $V$  axis of the hexcone. Also, each subcube of the RGB cube corresponds to a hexagonal cross-sectional area of the hexcone. At any cross section, all sides of the hexagon and all radial lines from the  $V$  axis to any vertex have the value  $V$ . Thus, for any set of RGB values,  $V$  is equal to the value of the maximum RGB component. The HSV point corresponding to this set of RGB values lies on the hexagonal cross section at value  $V$ . Parameter  $S$  is then determined as the relative distance of this point from the  $V$  axis. Parameter  $H$  is determined by calculating the relative position of the point within each sextant of the hexagon.

An algorithm for mapping any set of RGB values into the corresponding HSV values is given in the following procedure:

### The HLS Color Model

Another model based on intuitive color parameters is the *HLS system* used by the Tektronix Corporation. This color space has the double-cone representations shown in Figure 17. The three parameters in this color model are called hue ( $H$ ), lightness ( $L$ ), and saturation ( $S$ ).

Hue has the same meaning as in the HSV model. It specifies an angle about the vertical axis that locates a hue (spectral color). In this model,

$H = 0^\circ$  corresponds to blue. The remaining colors are specified around the perimeter of the cone in the same order as in the HSV model. Magenta is located at  $H = 60^\circ$ , red is at  $H = 120^\circ$ , and cyan is at  $H = 300^\circ$ . Again, complementary colors are  $180^\circ$  apart on the double cone.

The vertical axis in this model is called lightness,  $L$ . At  $L = 0$ , we have black, and at  $L = 1.0$ , we have white. Grayscale values are along the  $L$  axis, and the pure colors lie on the  $L = 0.5$  plane.

Saturation parameter  $S$  again specifies the purity of a color. This parameter varies from 0 to 1.0, and pure colors are those for which  $S = 1.0$  and  $L = 0.5$ . As  $S$  decreases, more white is added to a color. The grayscale line is at  $S = 0$ .

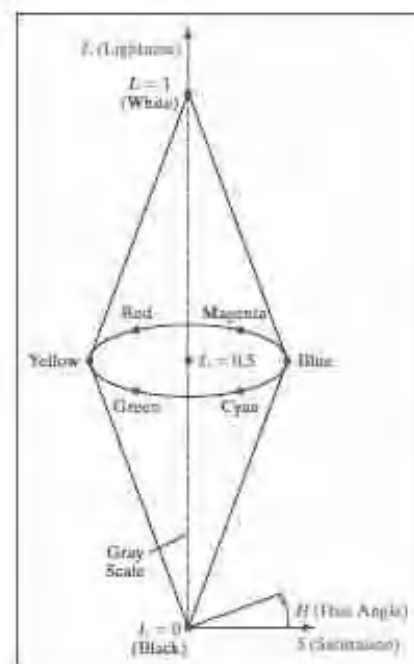


FIGURE 17  
The HLS double cone.

To specify a color, we begin by selecting hue angle  $H$ . Then a particular shade, tint, or tone for that hue is obtained by adjusting parameters  $L$  and  $S$ . We obtain a lighter color by increasing  $L$ , and we obtain a darker color by decreasing  $L$ . When  $S$  is decreased, the spatial color point moves toward the grayscale line.

### **Color Selection and Applications**

A graphics package can provide color capabilities in a way that aids us in making color selections. For example, an interface can contain sliders and color wheels instead of requiring that all color specifications be provided as numerical values for the RGB components. In addition, some aids can be provided for choosing harmonious color combinations and for basic color selection guidelines.

One method for obtaining a set of coordinating colors is to generate the color combinations from a small subspace of a color model. If colors are selected at regular intervals along any straight line within the RGB or CMY cube, for example, we can expect to obtain a set of well-matched colors. Randomly selected hues can be expected to produce harsh and clashing color combinations. Another consideration in color displays is the fact that we perceive colors at different depths.

This occurs because our eyes focus on colors according to their frequency. Blues, in particular, tend to recede. Displaying a blue pattern next to a red pattern can cause eye fatigue, because we continually need to refocus when our attention is switched from one area to the other. This problem can be reduced by separating these colors or by using colors from one-half or less of the color hexagon in the HSV model. With this technique, a display contains either blues and greens or reds and yellows.

As a general rule, the use of a smaller number of colors produces a better looking display than one with a large number of colors. Also, tints and shades tend to blend better than the pure hues. For a background, gray or the complement of one of the foreground colors is usually best.

\*\*\*\*\*