Table of Contents

Computer Graphics Hardware

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.

1 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. In

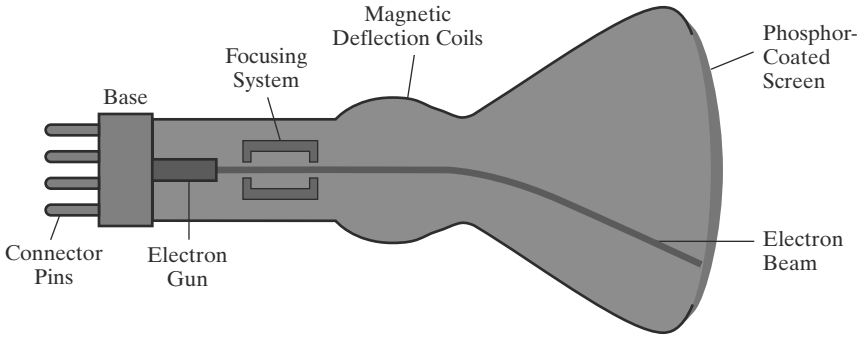


FIGURE 1
Basic design of a magnetic-deflection CRT.

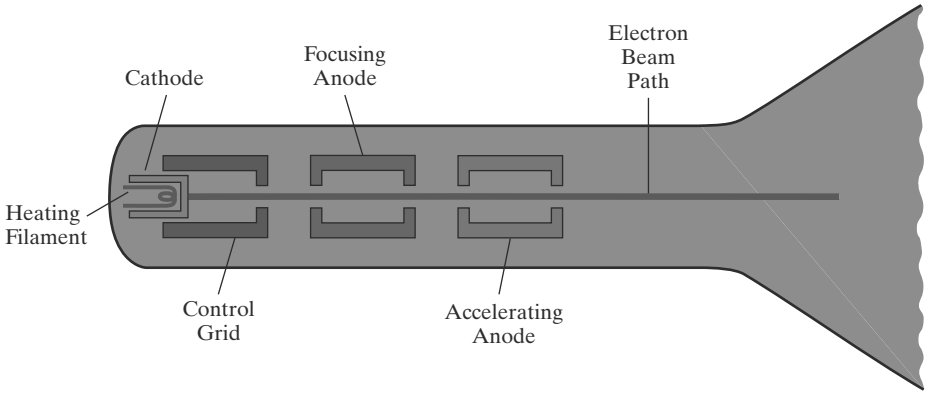


FIGURE 2
Operation of an electron gun with an accelerating anode.

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

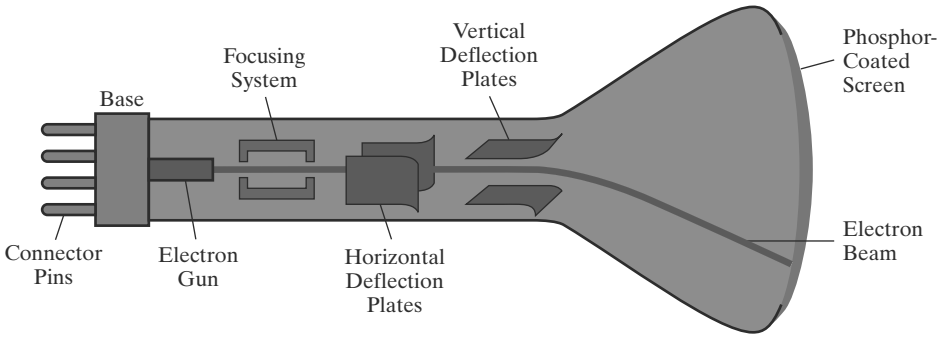


FIGURE 3
Electrostatic deflection of the electron beam in a CRT.

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 quantums 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.

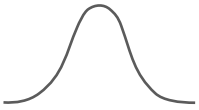


FIGURE 4
Intensity distribution of an illuminated phosphor spot on a CRT screen.

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 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*.

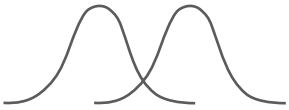


FIGURE 5
Two illuminated phosphor spots are distinguishable when their separation is greater than the diameter at which a spot intensity has fallen to 60 percent of maximum.

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.

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

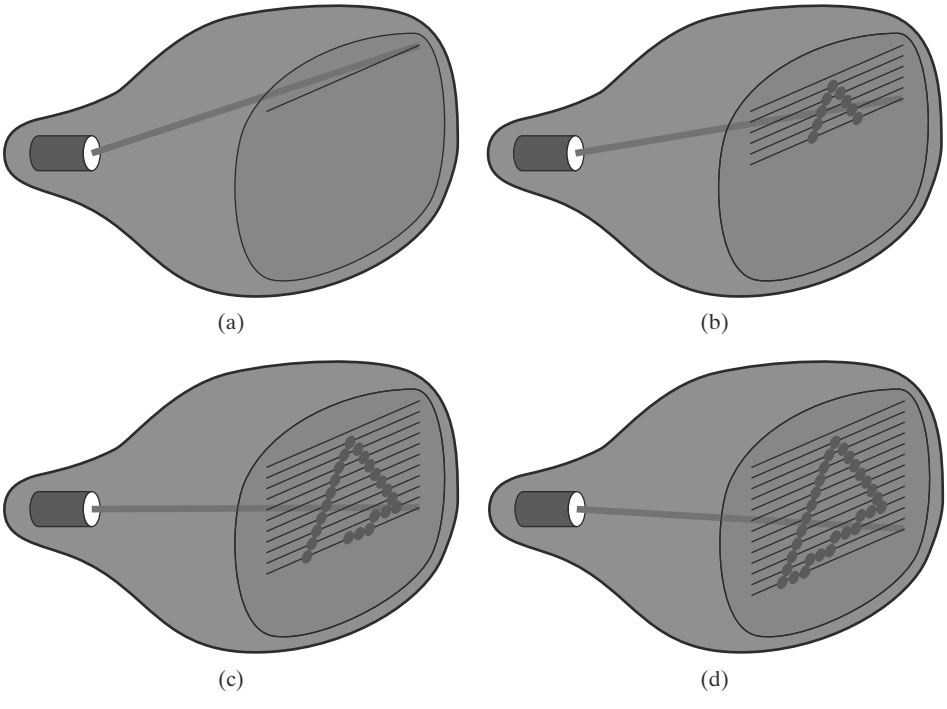


FIGURE 6
A raster-scan system displays an object as a set of discrete points across each scan line.

is **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, motion-picture 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

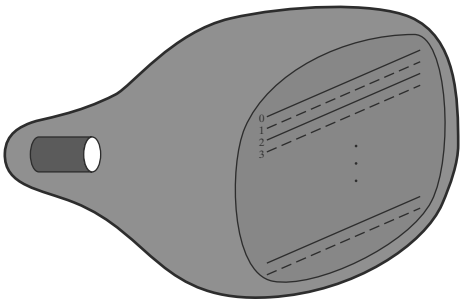


FIGURE 7
Interlacing scan lines on a raster-scan display. First, all points on the even-numbered (solid) scan lines are displayed; then all points along the odd-numbered (dashed) lines are displayed.

each scan line, is called the **horizontal retrace** of the electron beam. And at the end of each frame (displayed in $\frac{1}{80}$ to $\frac{1}{60}$ 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 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 frame-per-second, non-interlaced display, for instance, some flicker is noticeable. But with interlacing, each of the two passes can be accomplished in $\frac{1}{60}$ 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 line

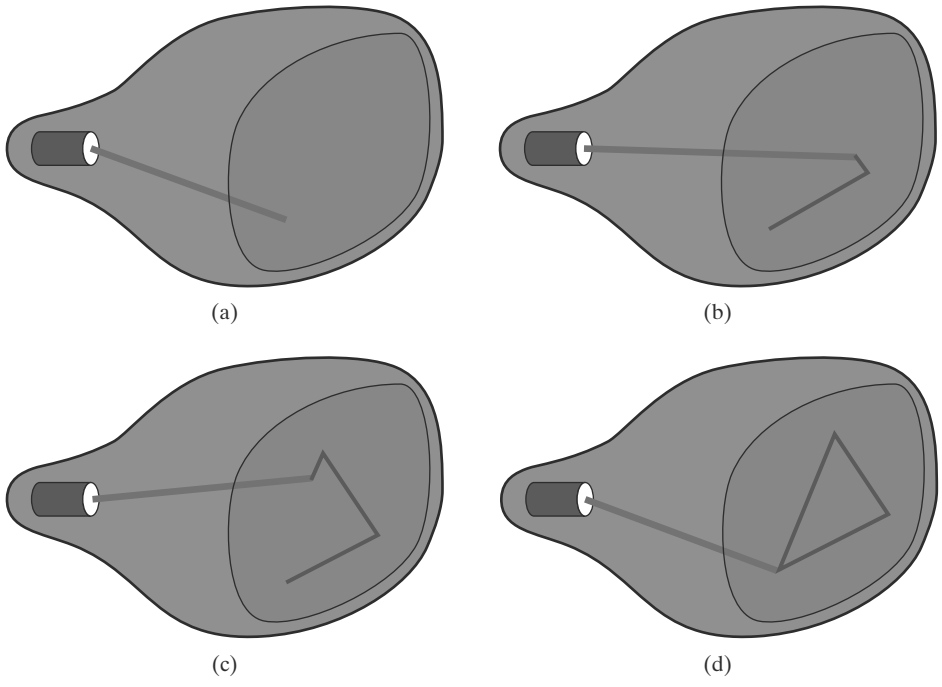


FIGURE 8
A random-scan system draws the component lines of an object in any specified order.

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.

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 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

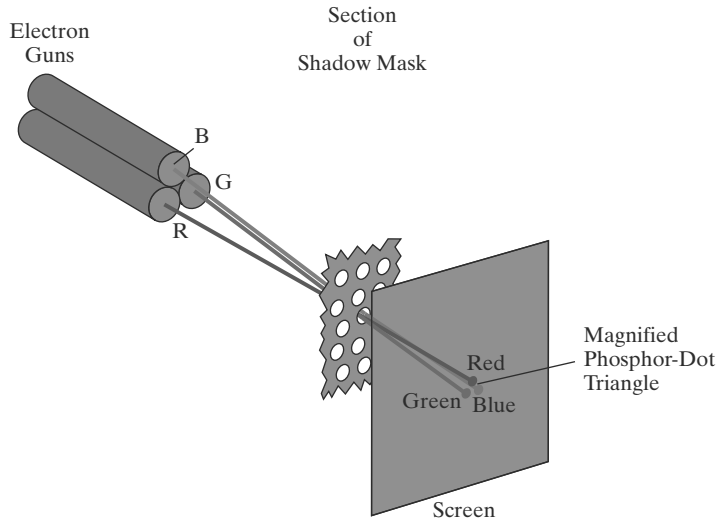


FIGURE 9
Operation of a delta-delta, shadow-mask CRT. Three electron guns, aligned with the triangular color-dot patterns on the screen, are directed to each dot triangle by a shadow mask.

light emitted from 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 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

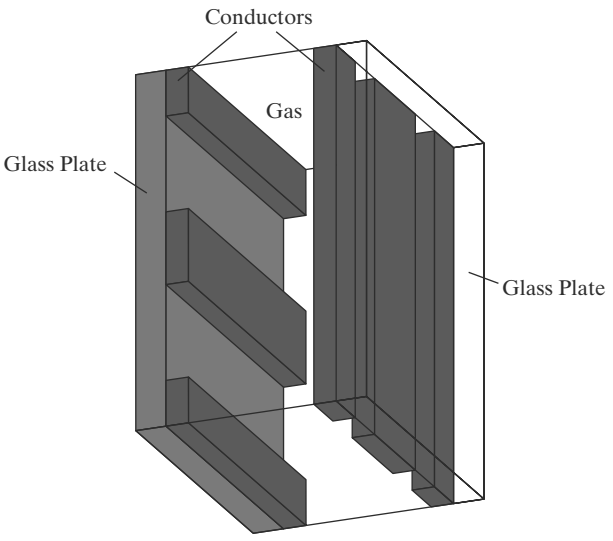


FIGURE 10
Basic design of a plasma-panel display device.

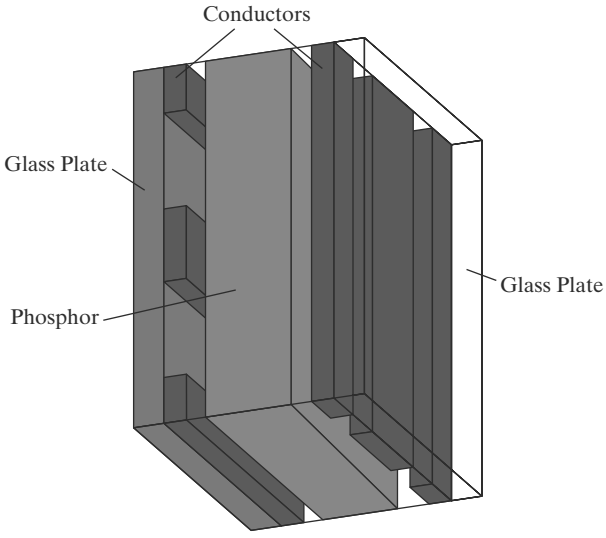


FIGURE 11
Basic design of a thin-film electroluminescent display device.

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, 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 (**LCD**s) 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, 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

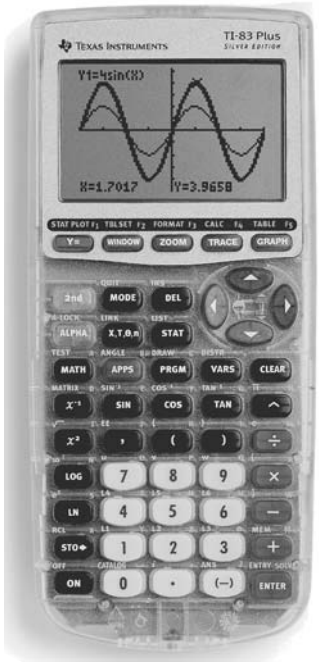


FIGURE 12
A handheld calculator with an LCD screen. (*Courtesy of Texas Instruments.*)

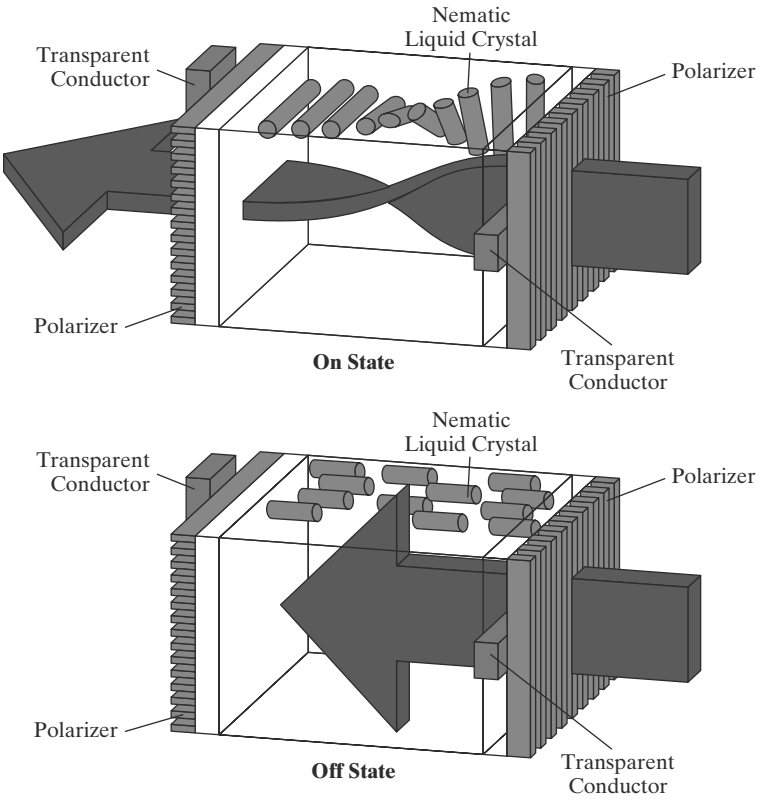


FIGURE 13
The light-twisting, shutter effect used in the design of most LCD devices.

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 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.

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.

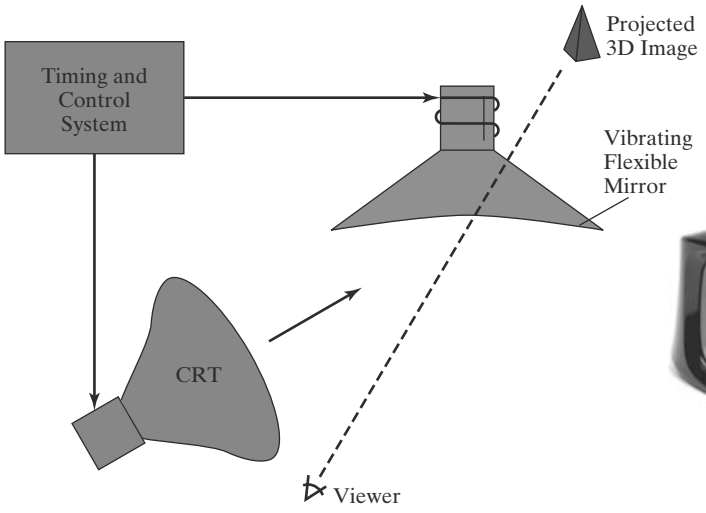


FIGURE 14
Operation of a three-dimensional display system using a vibrating mirror that changes focal length to match the depths of points in a scene.

FIGURE 15
Glasses for viewing a stereoscopic scene in 3D. (*Courtesy of XPAND, X6D USA Inc.*)

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.

2 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 Cartesian coordinates. In an application program, we use the commands

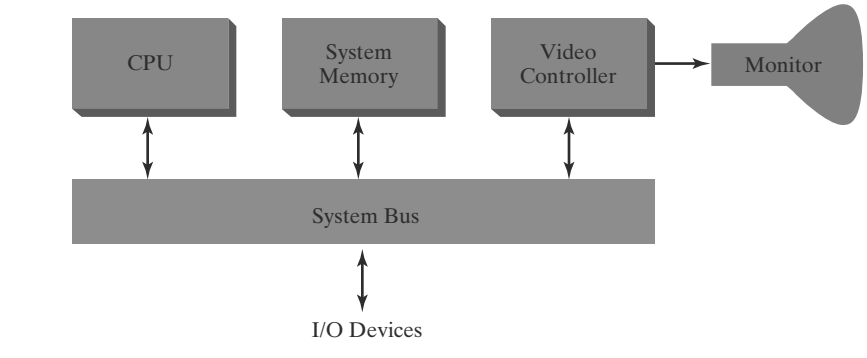


FIGURE 16
Architecture of a simple raster-graphics system.

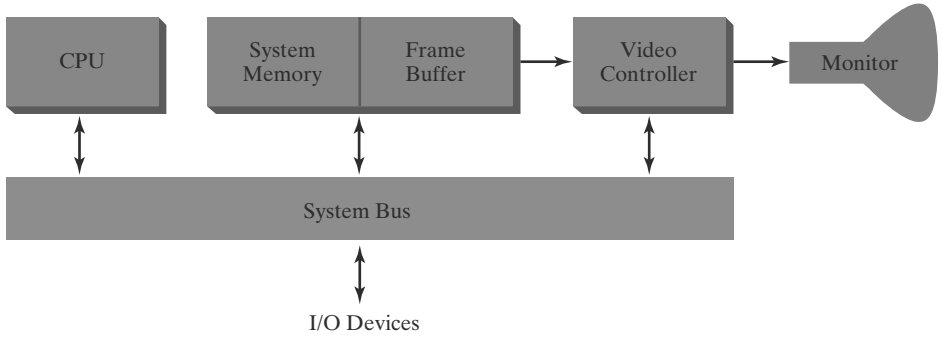


FIGURE 17
Architecture of a raster system with a fixed portion of the system memory reserved for the frame buffer.

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 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 scan line. 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,

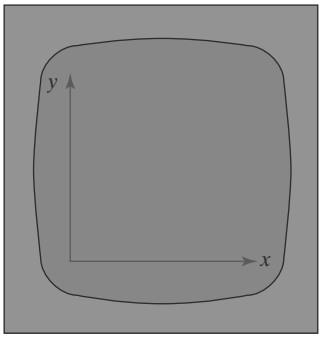


FIGURE 18
A Cartesian reference frame with origin at the lower-left corner of a video monitor.

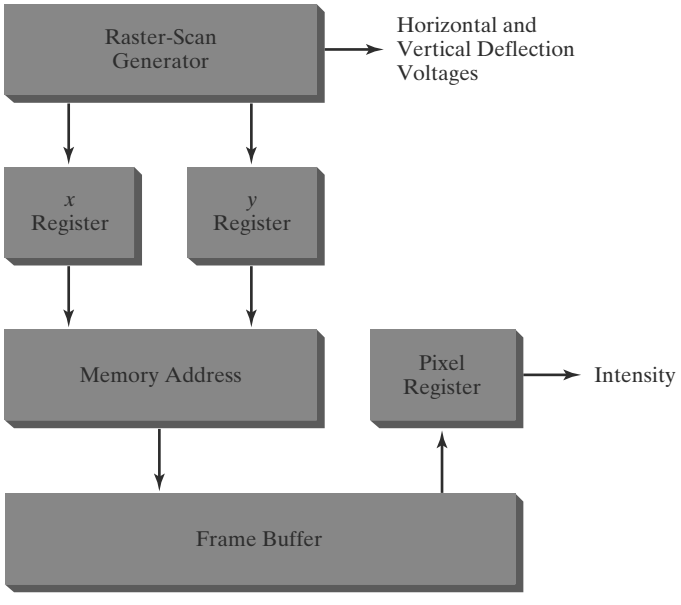


FIGURE 19
Basic video-controller refresh operations.

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

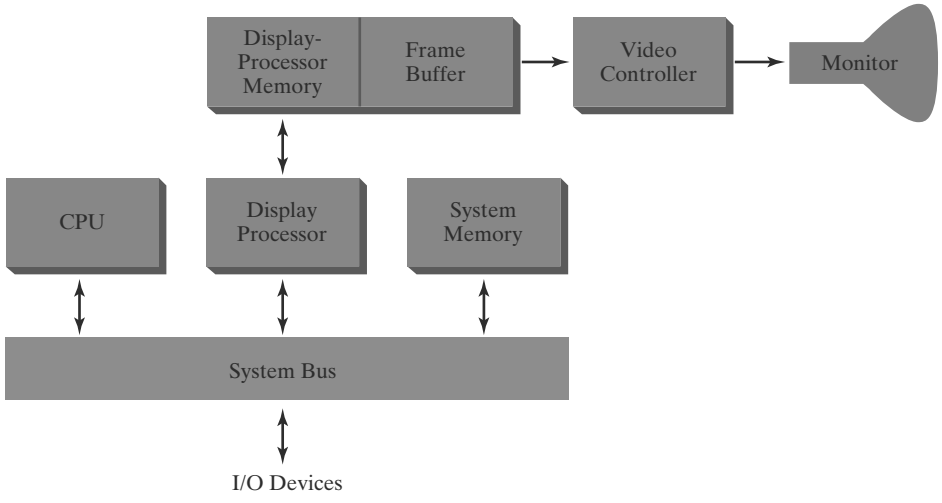


FIGURE 20
Architecture of a raster-graphics system with a display processor.

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 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.

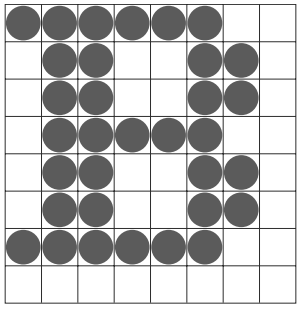


FIGURE 21
A character defined as a rectangular grid of pixel positions.

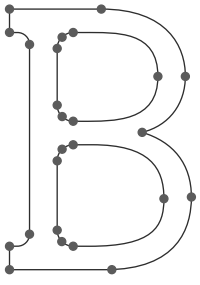


FIGURE 22
A character defined as an outline shape.

3 Graphics Workstations and Viewing Systems

Most graphics monitors today operate as raster-scan displays, and both CRT and flat-panel systems are in common use. Graphics workstations range from small general-purpose computer systems to multi-monitor facilities, often with ultra-large viewing screens. For a personal computer, screen resolutions vary from about 640 by 480 to 1280 by 1024, and diagonal screen lengths measure from 12 inches to over 21 inches. Most general-purpose systems now have considerable color capabilities, and many are full-color systems. For a desktop workstation specifically designed for graphics applications, the screen resolution can vary from 1280 by 1024 to about 1600 by 1200, with a typical screen diagonal of 18 inches or more. Commercial workstations can also be obtained with a variety of devices for specific applications.

High-definition graphics systems, with resolutions up to 2560 by 2048, are commonly used in medical imaging, air-traffic control, simulation, and CAD. Many high-end graphics workstations also include large viewing screens, often with specialized features.

Multi-panel display screens are used in a variety of applications that require "wall-sized" viewing areas. These systems are designed for presenting graphics displays at meetings, conferences, conventions, trade shows, retail stores, museums, and passenger terminals. A multi-panel display can be used to show a large

view of a single scene or several individual images. Each panel in the system displays one section of the overall picture. Color Plate 7 shows a 360° paneled viewing system in the NASA control-tower simulator, which is used for training and for testing ways to solve air-traffic and runway problems at airports. Large graphics displays can also be presented on curved viewing screens. A large, curved-screen system can be useful for viewing by a group of people studying a particular graphics application, such as the example in Color Plate 8. A control center, featuring a battery of standard monitors, allows an operator to view sections of the large display and to control the audio, video, lighting, and projection systems using a touch-screen menu. The system projectors provide a seamless, multichannel display that includes edge blending, distortion correction, and color balancing. And a surround-sound system is used to provide the audio environment.

4 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, button boxes, 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 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 mouse pad that has a grid of horizontal and vertical lines. The optical sensor detects

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

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 keystrongs, 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 Spaceballs

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 are 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, light pens 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.

5 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 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.

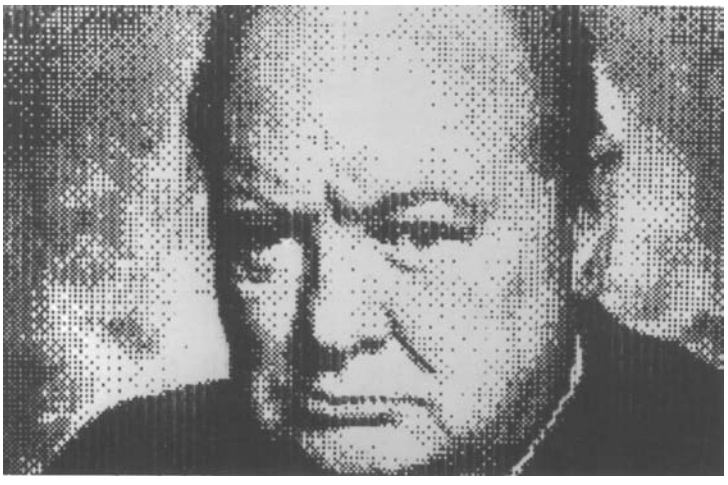


FIGURE 24
A picture generated on a dot-matrix printer, illustrating how the density of dot patterns can be varied to produce light and dark areas. (*Courtesy of Apple Computer, Inc.*)

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. Nonimpact 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.

6 Graphics Networks

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.

7 Graphics on the Internet

A great deal of graphics development is now done on the global collection of computer networks known as the **Internet.** Computers on the Internet communicate using *transmission control protocol/internet protocol* (TCP/IP). In addition, the **World Wide Web** provides a hypertext system that allows users to locate and view documents that can contain text, graphics, and audio. Resources, such as graphics files, are identified by a *uniform* (or *universal*) *resource locator* (*URL*). Each URL contains two parts: (1) the protocol for transferring the document, and (2) the server that contains the document and, optionally, the location (directory) on the server. For example, the URL *http://www.siggraph.org/* indicates a document that is to be transferred with the *hypertext transfer protocol* (*http*) and that the server is www.siggraph.org, which is the home page of the Special Interest Group in Graphics (SIGGRAPH) of the Association for Computing Machinery. Another common type of URL begins with *ftp://*. This identifies a site that accepts *file transfer protocol* (FTP) connections, through which programs or other files can be downloaded.

Documents on the Internet can be constructed with the *Hypertext Markup Language* (*HTML*). The development of HTML provided a simple method for describing a document containing text, graphics, and references (hyperlinks) to other documents. Although resources could be made available using HTML and URL addressing, it was difficult originally to find information on the Internet. Subsequently, the National Center for Supercomputing Applications (NCSA) developed a "browser" called Mosaic, which made it easier for users to search for Web resources. The Mosaic browser later evolved into the browser called Netscape Navigator. In turn, Netscape Navigator inspired the creation of the Mozilla family of browsers, whose most well-known member is, perhaps, Firefox.

HTML provides a simple method for developing graphics on the Internet, but it has limited capabilities. Therefore, other languages have been developed for Internet graphics applications.

8 Summary

In this chapter, we surveyed the major hardware and software features of computer-graphics systems. Hardware components include video monitors, hardcopy output devices, various kinds of input devices, and components for interacting with virtual environments.

The predominant graphics display device is the raster refresh monitor, based on television technology. A raster system uses a frame buffer to store the color value for each screen position (pixel). Pictures are then painted onto the screen by retrieving this information from the frame buffer (also called a *refresh buffer*) as the electron beam in the CRT sweeps across each scan line from top to bottom. Older vector displays construct pictures by drawing straight-line segments between specified endpoint positions. Picture information is then stored as a set of line-drawing instructions.

Many other video display devices are available. In particular, flat-panel display technology is developing at a rapid rate, and these devices are now used in a variety of systems, including both desktop and laptop computers. Plasma panels and liquid-crystal devices are two examples of flat-panel displays. Other display technologies include three-dimensional and stereoscopic-viewing systems. Virtual-reality systems can include either a stereoscopic headset or a standard video monitor.

For graphical input, we have a range of devices to choose from. Keyboards, button boxes, and dials are used to input text, data values, or programming options. The most popular "pointing" device is the mouse, but trackballs, spaceballs, joysticks, cursor-control keys, and thumbwheels are also used to position the screen cursor. In virtual-reality environments, data gloves are commonly used. Other input devices are image scanners, digitizers, touch panels, light pens, and voice systems.

Hardcopy devices for graphics workstations include standard printers and plotters, in addition to devices for producing slides, transparencies, and film output. Printers produce hardcopy output using dot-matrix, laser, ink-jet, electrostatic, or electrothermal methods. Graphs and charts can be produced with an ink-pen plotter or with a combination printer-plotter device.

REFERENCES

A general treatment of electronic displays is available in Tannas (1985) and in Sherr (1993). Flat-panel devices are discussed in Depp and Howard (1993). Additional information on raster-graphics architecture can be found in Foley et al. (1990). Three-dimensional and stereoscopic displays are discussed in Johnson (1982) and in Grotch (1983). Head-mounted displays and virtual-reality environments are discussed in Chung et al. (1989).

EXERCISES

1 List the operating characteristics for the following display technologies: raster refresh systems, vector refresh systems, plasma panels, and LCDs.

2 List some applications appropriate for each of the display technologies in the previous question.

3 Determine the resolution (pixels per centimeter) in the x and y directions for the video monitor in use on your system. Determine the aspect ratio, and explain how relative proportions of objects can be maintained on your system.

4 Consider three different raster systems with resolutions of 800 by 600, 1280 by 960, and 1680 by 1050. What size frame buffer (in bytes) is needed for each of these systems to store 16 bits per pixel? How much storage is required for each system if 32 bits per pixel are to be stored?

5 Suppose an RGB raster system is to be designed using an 8 inch by 10 inch screen with a resolution of 100 pixels per inch in each direction. If we want to store 6 bits per pixel in the frame buffer, how much storage (in bytes) do we need for the frame buffer?

6 How long would it take to load an 800 by 600 frame buffer with 16 bits per pixel, if 105 bits can be transferred per second? How long would it take to

load a 32-bit-per-pixel frame buffer with a resolution of 1680 by 1050 using this same transfer rate?

7 Suppose we have a computer with 32 bits per word and a transfer rate of 1 mip (one million instructions per second). How long would it take to fill the frame buffer of a 300 dpi (dot per inch) laser printer with a page size of 8.5 inches by 11 inches?

8 Consider two raster systems with resolutions of 800 by 600 and 1680 by 1050. How many pixels could be accessed per second in each of these systems by a display controller that refreshes the screen at a rate of 60 frames per second? What is the access time per pixel in each system?

9 Suppose we have a video monitor with a display area that measures 12 inches across and 9.6 inches high. If the resolution is 1280 by 1024 and the aspect ratio is 1, what is the diameter of each screen point?

10 How much time is spent scanning across each row of pixels during screen refresh on a raster system with a resolution of 1680 by 1050 and a refresh rate of 30 frames per second?

11 Consider a noninterlaced raster monitor with a resolution of n by m (m scan lines and n pixels per scan line), a refresh rate of r frames per second, a horizontal retrace time of t_{horiz}, and a vertical retrace time of t_{vert}. What is the fraction of the total refresh time per frame spent in retrace of the electron beam?

12 What is the fraction of the total refresh time per frame spent in retrace of the electron beam for a non-interlaced raster system with a resolution of 1680 by 1050, a refresh rate of 65 Hz, a horizontal retrace time of 4 microseconds, and a vertical retrace time of 400 microseconds?

13 Assuming that a certain full-color (24 bits per pixel) RGB raster system has a 1024 by 1024 frame buffer, how many distinct color choices (intensity levels) would we have available? How many different colors could we display at any one time?

14 Compare the advantages and disadvantages of a three-dimensional monitor using a varifocal mirror to those of a stereoscopic system.

15 List the different input and output components that are typically used with virtual-reality systems. Also, explain how users interact with a virtual scene displayed with different output devices, such as two-dimensional and stereoscopic monitors.

16 Explain how virtual-reality systems can be used in design applications. What are some other applications for virtual-reality systems?

17 List some applications for large-screen displays.

18 Explain the differences between a general graphics system designed for a programmer and one designed for a specific application, such as architectural design.

IN MORE DEPTH

1 In this course, you will design and build a graphics application incrementally. You should have a basic understanding of the types of applications for which computer graphics are used. Try to formulate a few ideas about one or more particular applications you may be interested in developing over the course of your studies. Keep in mind that you will be asked to incorporate techniques covered in this text, as well as to show your understanding of alternative methods for implementing those concepts. As such, the application should be simple enough that you can realistically implement it in a reasonable amount of time, but complex enough to afford the inclusion of each of the relevant concepts in the text. One obvious example is a video game of some sort in which the user interacts with a virtual environment that is initially displayed in two dimensions and later in three dimensions. Some concepts to consider would be two- and three-dimensional objects of different forms (some simple, some moderately complex with curved surfaces, etc.), complex shading of object surfaces, various lighting techniques, and animation of some sort. Write a report with at least three to four ideas that you will choose to implement you acquire more knowledge of the course material. Note that one type of application may be more suited to demonstrate a given concept than another.

2 Find details about the graphical capabilities of the graphics controller and the display device in your system by looking up their specifications. Record the following information:

- What is the maximum resolution your graphics controller is capable of rendering?
- What is the maximum resolution of your display device?
- What type of hardware does your graphics controller contain?
- What is the GPU's clock speed?
- How much of its own graphics memory does it have?

If you have a relatively new system, it is unlikely that you will be pushing the envelope of your graphics hardware in your application development for this text. However, knowing the capabilities of your graphics system will provide you with a sense of how much it will be able to handle.

Computer Graphics Hardware Color Plates

Color Plate 7
The 360° viewing screen in the NASA airport control-tower simulator, called the FutureFlight Central Facility. (*Courtesy of Silicon Graphics, Inc. and NASA. © 2003 SGI. All rights reserved.*)

Color Plate 8
A geophysical visualization presented on a 25-foot semicircular screen, which provides a 160° horizontal and 40° vertical field of view. (*Courtesy of Silicon Graphics, Inc., the Landmark Graphics Corporation, and Trimension Systems. © 2003 SGI. All rights reserved.*)

This page intentionally left blank

Computer Graphics Software

There are two broad classifications for computer-graphics software: special-purpose packages and general programming packages. Special-purpose packages are designed for nonprogrammers who want to generate pictures, graphs, or charts in some application area without worrying about the graphics procedures that might be needed to produce such displays. The interface to a special-purpose package is typically a set of menus that allow users to communicate with the programs in their own terms. Examples of such applications include artists' painting programs and various architectural, business, medical, and engineering CAD systems. By contrast, a general programming package provides a library of graphics functions that can be used in a programming language such as C, C++, Java, or Fortran. Basic functions in a typical graphics library include those for specifying picture components (straight lines, polygons, spheres, and other objects), setting color values, selecting views of a scene, and applying rotations or other transformations. Some examples of general graphics programming packages are

GL (Graphics Library), OpenGL, VRML (Virtual-Reality Modeling Language), Java 2D, and Java 3D. A set of graphics functions is often called a **computer-graphics application programming interface** (**CG API**) because the library provides a software interface between a programming language (such as C++) and the hardware. So when we write an application program in C++, the graphics routines allow us to construct and display a picture on an output device.

1 Coordinate Representations

To generate a picture using a programming package, we first need to give the geometric descriptions of the objects that are to be displayed. These descriptions determine the locations and shapes of the objects. For example, a box is specified by the positions of its corners (vertices), and a sphere is defined by its center position and radius. With few exceptions, general graphics packages require geometric descriptions to be specified in a standard, right-handed, Cartesian-coordinate reference frame. If coordinate values for a picture are given in some other reference frame (spherical, hyperbolic, etc.), they must be converted to Cartesian coordinates before they can be input to the graphics package. Some packages that are designed for specialized applications may allow use of other coordinate frames that are appropriate for those applications.

In general, several different Cartesian reference frames are used in the process of constructing and displaying a scene. First, we can define the shapes of individual objects, such as trees or furniture, within a separate reference frame for each object. These reference frames are called **modeling coordinates,** or sometimes **local coordinates** or **master coordinates.** Once the individual object shapes have been specified, we can construct ("model") a scene by placing the objects into appropriate locations within a scene reference frame called **world coordinates.** This step involves the transformation of the individual modeling-coordinate frames to specified positions and orientations within the world-coordinate frame. As an example, we could construct a bicycle by defining each of its parts (wheels, frame, seat, handlebars, gears, chain, pedals) in a separate modeling-coordinate frame. Then, the component parts are fitted together in world coordinates. If both bicycle wheels are the same size, we need to describe only one wheel in a local-coordinate frame. Then the wheel description is fitted into the world-coordinate bicycle description in two places. For scenes that are not too complicated, object components can be set up directly within the overall world-coordinate object structure, bypassing the modeling-coordinate and modeling-transformation steps. Geometric descriptions in modeling coordinates and world coordinates can be given in any convenient floating-point or integer values, without regard for the constraints of a particular output device. For some scenes, we might want to specify object geometries in fractions of a foot, while for other applications we might want to use millimeters, or kilometers, or light-years.

After all parts of a scene have been specified, the overall world-coordinate description is processed through various routines onto one or more output-device reference frames for display. This process is called the **viewing pipeline.** World-coordinate positions are first converted to *viewing coordinates* corresponding to the view we want of a scene, based on the position and orientation of a hypothetical camera. Then object locations are transformed to a two-dimensional (2D) projection of the scene, which corresponds to what we will see on the output device. The scene is then stored in **normalized coordinates,** where each coordinate value is in the range from -1 to 1 or in the range from 0 to 1, depending on the system.

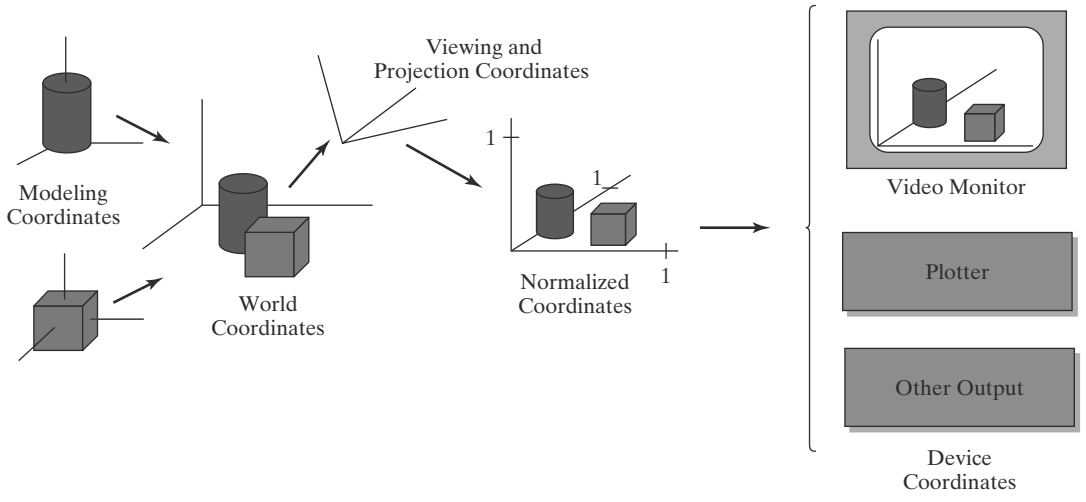


FIGURE 1
The transformation sequence from modeling coordinates to device coordinates for a three-dimensional scene. Object shapes can be individually defined in modeling-coordinate reference systems. Then the shapes are positioned within the world-coordinate scene. Next, world-coordinate specifications are transformed through the viewing pipeline to viewing and projection coordinates and then to normalized coordinates. At the final step, individual device drivers transfer the normalized-coordinate representation of the scene to the output devices for display.

Normalized coordinates are also referred to as *normalized device coordinates*, since using this representation makes a graphics package independent of the coordinate range for any specific output device. We also need to identify visible surfaces and eliminate picture parts outside the bounds for the view we want to show on the display device. Finally, the picture is scan-converted into the refresh buffer of a raster system for display. The coordinate systems for display devices are generally called **device coordinates,** or **screen coordinates** in the case of a video monitor. Often, both normalized coordinates and screen coordinates are specified in a left-handed coordinate reference frame so that increasing positive distances from the xy plane (the screen, or viewing plane) can be interpreted as being farther from the viewing position.

Figure 1 briefly illustrates the sequence of coordinate transformations from modeling coordinates to device coordinates for a display that is to contain a view of two three-dimensional (3D) objects. An initial modeling-coordinate position (x_{mc}, y_{mc}, z_{mc}) in this illustration is transferred to world coordinates, then to viewing and projection coordinates, then to left-handed normalized coordinates, and finally to a device-coordinate position (x_{dc}, y_{dc}) with the sequence:

$$(x_{mc}, y_{mc}, z_{mc}) \rightarrow (x_{wc}, y_{wc}, z_{wc}) \rightarrow (x_{vc}, y_{vc}, z_{vc}) \rightarrow (x_{pc}, y_{pc}, z_{pc})$$
$$\rightarrow (x_{nc}, y_{nc}, z_{nc}) \rightarrow (x_{dc}, y_{dc})$$

Device coordinates (x_{dc}, y_{dc}) are integers within the range $(0, 0)$ to $(x_{\max}, y_{\max})$ for a particular output device. In addition to the two-dimensional positions (x_{dc}, y_{dc}) on the viewing surface, depth information for each device-coordinate position is stored for use in various visibility and surface-processing algorithms.

2 Graphics Functions

A general-purpose graphics package provides users with a variety of functions for creating and manipulating pictures. These routines can be broadly classified

according to whether they deal with graphics output, input, attributes, transformations, viewing, subdividing pictures, or general control.

The basic building blocks for pictures are referred to as **graphics output primitives.** They include character strings and geometric entities, such as points, straight lines, curved lines, filled color areas (usually polygons), and shapes defined with arrays of color points. In addition, some graphics packages provide functions for displaying more complex shapes such as spheres, cones, and cylinders. Routines for generating output primitives provide the basic tools for constructing pictures.

Attributes are properties of the output primitives; that is, an attribute describes how a particular primitive is to be displayed. This includes color specifications, line styles, text styles, and area-filling patterns.

We can change the size, position, or orientation of an object within a scene using **geometric transformations.** Some graphics packages provide an additional set of functions for performing **modeling transformations,** which are used to construct a scene where individual object descriptions are given in local coordinates. Such packages usually provide a mechanism for describing complex objects (such as an electrical circuit or a bicycle) with a tree (hierarchical) structure. Other packages simply provide the geometric-transformation routines and leave modeling details to the programmer.

After a scene has been constructed, using the routines for specifying the object shapes and their attributes, a graphics package projects a view of the picture onto an output device. **Viewing transformations** are used to select a view of the scene, the type of projection to be used, and the location on a video monitor where the view is to be displayed. Other routines are available for managing the screen display area by specifying its position, size, and structure. For three-dimensional scenes, visible objects are identified and the lighting conditions are applied.

Interactive graphics applications use various kinds of input devices, including a mouse, a tablet, and a joystick. **Input functions** are used to control and process the data flow from these interactive devices.

Some graphics packages also provide routines for subdividing a picture description into a named set of component parts. And other routines may be available for manipulating these picture components in various ways.

Finally, a graphics package contains a number of housekeeping tasks, such as clearing a screen display area to a selected color and initializing parameters. We can lump the functions for carrying out these chores under the heading **control operations.**

3 Software Standards

The primary goal of standardized graphics software is portability. When packages are designed with standard graphics functions, software can be moved easily from one hardware system to another and used in different implementations and applications. Without standards, programs designed for one hardware system often cannot be transferred to another system without extensive rewriting of the programs.

International and national standards-planning organizations in many countries have cooperated in an effort to develop a generally accepted standard for computer graphics. After considerable effort, this work on standards led to the development of the **Graphical Kernel System** (**GKS**) in 1984. This system was adopted as the first graphics software standard by the International Standards Organization (ISO) and by various national standards organizations, including

the American National Standards Institute (ANSI). Although GKS was originally designed as a two-dimensional graphics package, a three-dimensional GKS extension was soon developed. The second software standard to be developed and approved by the standards organizations was **Programmer's Hierarchical Interactive Graphics System** (**PHIGS**), which is an extension of GKS. Increased capabilities for hierarchical object modeling, color specifications, surface rendering, and picture manipulations are provided in PHIGS. Subsequently, an extension of PHIGS, called PHIGS+, was developed to provide three-dimensional surface-rendering capabilities not available in PHIGS.

As the GKS and PHIGS packages were being developed, the graphics workstations from Silicon Graphics, Inc. (SGI), became increasingly popular. These workstations came with a set of routines called **GL** (**Graphics Library**), which very soon became a widely used package in the graphics community. Thus, GL became a de facto graphics standard. The GL routines were designed for fast, real-time rendering, and soon this package was being extended to other hardware systems. As a result, OpenGL was developed as a hardware-independent version of GL in the early 1990s. This graphics package is now maintained and updated by the **OpenGL Architecture Review Board,** which is a consortium of representatives from many graphics companies and organizations. The OpenGL library is specifically designed for efficient processing of three-dimensional applications, but it can also handle two-dimensional scene descriptions as a special case of three dimensions where all the z coordinate values are 0.

Graphics functions in any package are typically defined as a set of specifications independent of any programming language. A **language binding** is then defined for a particular high-level programming language. This binding gives the syntax for accessing the various graphics functions from that language. Each language binding is defined to make best use of the corresponding language capabilities and to handle various syntax issues, such as data types, parameter passing, and errors. Specifications for implementing a graphics package in a particular language are set by the ISO. The OpenGL bindings for the C and C++ languages are the same. Other OpenGL bindings are also available, such as those for Java and Python.

Later in this book, we use the C/C++ binding for OpenGL as a framework for discussing basic graphics concepts and the design and application of graphics packages. Example programs in C++ illustrate applications of OpenGL and the general algorithms for implementing graphics functions.

4 Other Graphics Packages

Many other computer-graphics programming libraries have been developed. Some provide general graphics routines, and some are aimed at specific applications or particular aspects of computer graphics, such as animation, virtual reality, or graphics on the Internet.

A package called *Open Inventor* furnishes a set of object-oriented routines for describing a scene that is to be displayed with calls to OpenGL. The *Virtual-Reality Modeling Language* (VRML), which began as a subset of Open Inventor, allows us to set up three-dimensional models of virtual worlds on the Internet. We can also construct pictures on the Web using graphics libraries developed for the Java language. With *Java 2D,* we can create two-dimensional scenes within Java applets, for example; or we can produce three-dimensional web displays with *Java 3D*. With the *RenderMan Interface* from the Pixar Corporation, we can generate scenes

using a variety of lighting models. Finally, graphics libraries are often provided in other types of systems, such as Mathematica, MatLab, and Maple.

5 Introduction to OpenGL

A basic library of functions is provided in OpenGL for specifying graphics primitives, attributes, geometric transformations, viewing transformations, and many other operations. As we noted in the last section, OpenGL is designed to be hardware independent, so many operations, such as input and output routines, are not included in the basic library. However, input and output routines and many additional functions are available in auxiliary libraries that have been developed for OpenGL programs.

Basic OpenGL Syntax

Function names in the **OpenGL basic library** (also called the **OpenGL core library**) are prefixed with `gl`, and each component word within a function name has its first letter capitalized. The following examples illustrate this naming convention:

```
glBegin,   glClear,   glCopyPixels,   glPolygonMode
```

Certain functions require that one (or more) of their arguments be assigned a symbolic constant specifying, for instance, a parameter name, a value for a parameter, or a particular mode. All such constants begin with the uppercase letters `GL`. In addition, component words within a constant name are written in capital letters, and the underscore (_) is used as a separator between all component words in the name. The following are a few examples of the several hundred symbolic constants available for use with OpenGL functions:

```
GL_2D,  GL_RGB,  GL_CCW,  GL_POLYGON,  GL_AMBIENT_AND_DIFFUSE
```

The OpenGL functions also expect specific data types. For example, an OpenGL function parameter might expect a value that is specified as a 32-bit integer. But the size of an integer specification can be different on different machines. To indicate a specific data type, OpenGL uses special built-in, data-type names, such as

```
GLbyte,  GLshort,  GLint,  GLfloat,  GLdouble,  GLboolean
```

Each data-type name begins with the capital letters `GL`, and the remainder of the name is a standard data-type designation written in lowercase letters.

Some arguments of OpenGL functions can be assigned values using an array that lists a set of data values. This is an option for specifying a list of values as a pointer to an array, rather than specifying each element of the list explicitly as a parameter argument. A typical example of the use of this option is in specifying xyz coordinate values.

Related Libraries

In addition to the OpenGL basic (core) library, there are a number of associated libraries for handling special operations. The **OpenGL Utility** (**GLU**) provides routines for setting up viewing and projection matrices, describing complex objects with line and polygon approximations, displaying quadrics and B-splines

using linear approximations, processing the surface-rendering operations, and other complex tasks. Every OpenGL implementation includes the GLU library, and all GLU function names start with the prefix `glu`. There is also an object-oriented toolkit based on OpenGL, called **Open Inventor,** which provides routines and predefined object shapes for interactive three-dimensional applications. This toolkit is written in C++.

To create a graphics display using OpenGL, we first need to set up a **display window** on our video screen. This is simply the rectangular area of the screen in which our picture will be displayed. We cannot create the display window directly with the basic OpenGL functions since this library contains only device-independent graphics functions, and window-management operations depend on the computer we are using. However, there are several window-system libraries that support OpenGL functions for a variety of machines. The **OpenGL Extension to the X Window System** (**GLX**) provides a set of routines that are prefixed with the letters `glX`. Apple systems can use the **Apple GL** (**AGL**) interface for window-management operations. Function names for this library are prefixed with `agl`. For Microsoft Windows systems, the **WGL** routines provide a **Windows-to-OpenGL** interface. These routines are prefixed with the letters `wgl`. The **Presentation Manager to OpenGL** (**PGL**) is an interface for the IBM OS/2, which uses the prefix `pgl` for the library routines. The **OpenGL Utility Toolkit** (**GLUT**) provides a library of functions for interacting with any screen-windowing system. The GLUT library functions are prefixed with `glut`, and this library also contains methods for describing and rendering quadric curves and surfaces.

Since GLUT is an interface to other device-specific window systems, we can use it so that our programs will be device-independent. Information regarding the latest version of GLUT and download procedures for the source code are available at the following web site:

```
http://www.opengl.org/resources/libraries/glut/
```

Header Files

In all of our graphics programs, we will need to include the header file for the OpenGL core library. For most applications we will also need GLU, and on many systems we will need to include the header file for the window system. For instance, with Microsoft Windows, the header file that accesses the WGL routines is `windows.h`. This header file must be listed before the OpenGL and GLU header files because it contains macros needed by the Microsoft Windows version of the OpenGL libraries. So the source file in this case would begin with

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
```

However, if we use GLUT to handle the window-managing operations, we do not need to include `gl.h` and `glu.h` because GLUT ensures that these will be included correctly. Thus, we can replace the header files for OpenGL and GLU with

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

(We could include `gl.h` and `glu.h` as well, but doing so would be redundant and could affect program portability.) On some systems, the header files for OpenGL and GLUT routines are found in different places in the filesystem. For instance, on Apple OS X systems, the header file inclusion statement would be

```
#include <GLUT/glut.h>
```

In addition, we will often need to include header files that are required by the C++ code. For example,

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

With the ISO/ANSI standard for C++, these header files are called `cstdio`, `cstdlib`, and `cmath`.

Display-Window Management Using GLUT

To get started, we can consider a simplified, minimal number of operations for displaying a picture. Since we are using the OpenGL Utility Toolkit, our first step is to initialize GLUT. This initialization function could also process any command-line arguments, but we will not need to use these parameters for our first example programs. We perform the GLUT initialization with the statement

```
glutInit (&argc, argv);
```

Next, we can state that a display window is to be created on the screen with a given caption for the title bar. This is accomplished with the function

```
glutCreateWindow ("An Example OpenGL Program");
```

where the single argument for this function can be any character string that we want to use for the display-window title.

Then we need to specify what the display window is to contain. For this, we create a picture using OpenGL functions and pass the picture definition to the GLUT routine `glutDisplayFunc`, which assigns our picture to the display window. As an example, suppose we have the OpenGL code for describing a line segment in a procedure called `lineSegment`. Then the following function call passes the line-segment description to the display window:

```
glutDisplayFunc (lineSegment);
```

But the display window is not yet on the screen. We need one more GLUT function to complete the window-processing operations. After execution of the following statement, all display windows that we have created, including their graphic content, are now activated:

```
glutMainLoop ( );
```

This function must be the last one in our program. It displays the initial graphics and puts the program into an infinite loop that checks for input from devices such as a mouse or keyboard. Our first example will not be interactive, so the program will just continue to display our picture until we close the display window. In later chapters, we consider how we can modify our OpenGL programs to handle interactive input.

Although the display window that we created will be in some default location and size, we can set these parameters using additional GLUT functions. We use the `glutInitWindowPosition` function to give an initial location for the upper-left corner of the display window. This position is specified in integer screen

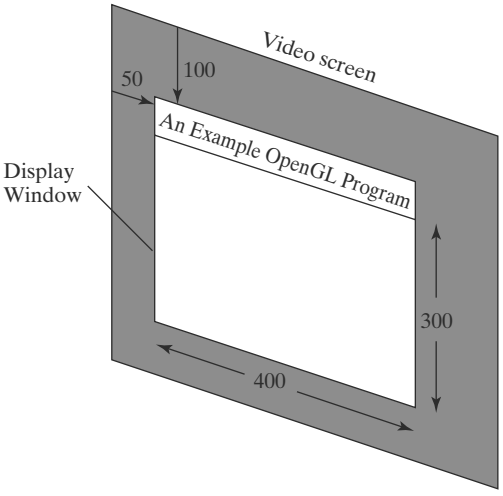


FIGURE 2
A 400 by 300 display window at position (50, 100) relative to the top-left corner of the video display.

coordinates, whose origin is at the upper-left corner of the screen. For instance, the following statement specifies that the upper-left corner of the display window should be placed 50 pixels to the right of the left edge of the screen and 100 pixels down from the top edge of the screen:

```
glutInitWindowPosition (50, 100);
```

Similarly, the `glutInitWindowSize` function is used to set the initial pixel width and height of the display window. Thus, we specify a display window with an initial width of 400 pixels and a height of 300 pixels (Fig. 2) with the statement

```
glutInitWindowSize (400, 300);
```

After the display window is on the screen, we can reposition and resize it.

We can also set a number of other options for the display window, such as buffering and a choice of color modes, with the `glutInitDisplayMode` function. Arguments for this routine are assigned symbolic GLUT constants. For example, the following command specifies that a single refresh buffer is to be used for the display window and that we want to use the color mode which uses red, green, and blue (RGB) components to select color values:

```
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
```

The values of the constants passed to this function are combined using a logical *or* operation. Actually, single buffering and RGB color mode are the default options. But we will use the function now as a reminder that these are the options that are set for our display. Later, we discuss color modes in more detail, as well as other display options, such as double buffering for animation applications and selecting parameters for viewing three-dimensional scenes.

A Complete OpenGL Program

There are still a few more tasks to perform before we have all the parts that we need for a complete program. For the display window, we can choose a background color. And we need to construct a procedure that contains the appropriate OpenGL functions for the picture that we want to display.

Using RGB color values, we set the background color for the display window to be white, as in Figure 2, with the OpenGL function:

```
glClearColor (1.0, 1.0, 1.0, 0.0);
```

The first three arguments in this function set the red, green, and blue component colors to the value 1.0, giving us a white background color for the display window. If, instead of 1.0, we set each of the component colors to 0.0, we would get a black background. And if all three of these components were set to the same intermediate value between 0.0 and 1.0, we would get some shade of gray. The fourth parameter in the `glClearColor` function is called the *alpha value* for the specified color. One use for the alpha value is as a "blending" parameter. When we activate the OpenGL blending operations, alpha values can be used to determine the resulting color for two overlapping objects. An alpha value of 0.0 indicates a totally transparent object, and an alpha value of 1.0 indicates an opaque object. Blending operations will not be used for a while, so the value of alpha is irrelevant to our early example programs. For now, we will simply set alpha to 0.0.

Although the `glClearColor` command assigns a color to the display window, it does not put the display window on the screen. To get the assigned window color displayed, we need to invoke the following OpenGL function:

```
glClear (GL_COLOR_BUFFER_BIT);
```

The argument `GL_COLOR_BUFFER_BIT` is an OpenGL symbolic constant specifying that it is the bit values in the color buffer (refresh buffer) that are to be set to the values indicated in the `glClearColor` function. (OpenGL has several different kinds of buffers that can be manipulated.

In addition to setting the background color for the display window, we can choose a variety of color schemes for the objects we want to display in a scene. For our initial programming example, we will simply set the object color to be a dark green:

```
glColor3f (0.0, 0.4, 0.2);
```

The suffix `3f` on the `glColor` function indicates that we are specifying the three RGB color components using floating-point (`f`) values. This function requires that the values be in the range from 0.0 to 1.0, and we have set red = 0.0, green = 0.4, and blue = 0.2.

For our first program, we simply display a two-dimensional line segment. To do this, we need to tell OpenGL how we want to "project" our picture onto the display window because generating a two-dimensional picture is treated by OpenGL as a special case of three-dimensional viewing. So, although we only want to produce a very simple two-dimensional line, OpenGL processes our picture through the full three-dimensional viewing operations. We can set the projection type (mode) and other viewing parameters that we need with the following two functions:

```
glMatrixMode (GL_PROJECTION);
gluOrtho2D (0.0, 200.0, 0.0, 150.0);
```

This specifies that an orthogonal projection is to be used to map the contents of a two-dimensional rectangular area of world coordinates to the screen, and that the x-coordinate values within this rectangle range from 0.0 to 200.0 with y-coordinate values ranging from 0.0 to 150.0. Whatever objects we define

within this world-coordinate rectangle will be shown within the display window. Anything outside this coordinate range will not be displayed. Therefore, the GLU function `gluOrtho2D` defines the coordinate reference frame within the display window to be (0.0, 0.0) at the lower-left corner of the display window and (200.0, 150.0) at the upper-right window corner. Since we are only describing a two-dimensional object, the orthogonal projection has no other effect than to "paste" our picture into the display window that we defined earlier. For now, we will use a world-coordinate rectangle with the same aspect ratio as the display window, so that there is no distortion of our picture. Later, we will consider how we can maintain an aspect ratio that does not depend upon the display-window specification.

Finally, we need to call the appropriate OpenGL routines to create our line segment. The following code defines a two-dimensional, straight-line segment with integer, Cartesian endpoint coordinates (180, 15) and (10, 145).

```
glBegin (GL_LINES);
    glVertex2i (180, 15);
    glVertex2i (10, 145);
glEnd ( );
```

Now we are ready to put all the pieces together. The following OpenGL program is organized into three functions. We place all initializations and related one-time parameter settings in function `init`. Our geometric description of the "picture" that we want to display is in function `lineSegment`, which is the function that will be referenced by the GLUT function `glutDisplayFunc`. And the `main` function contains the GLUT functions for setting up the display window and getting our line segment onto the screen. Figure 3 shows the display window and line segment generated by this program.

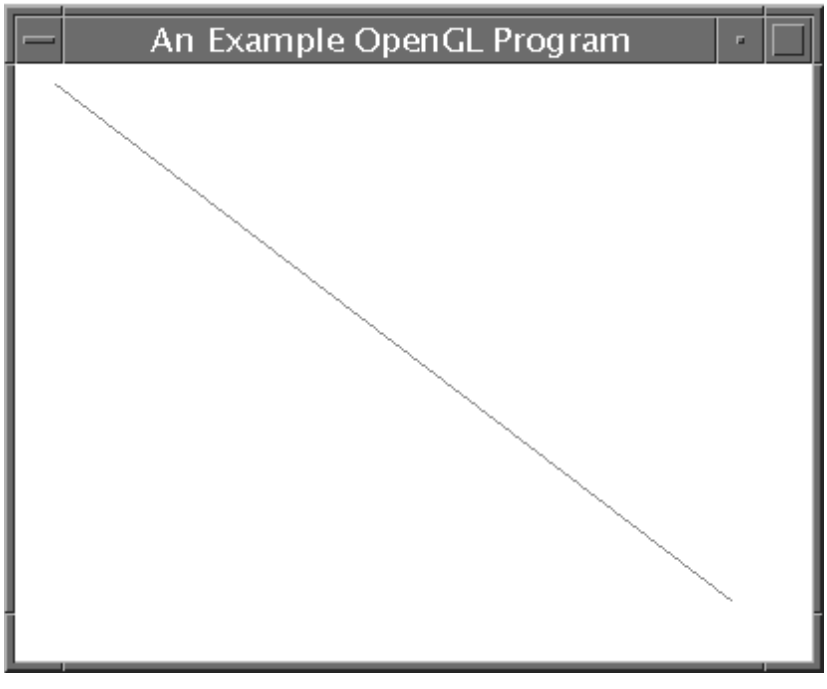


FIGURE 3
The display window and line segment produced by the example program.

```
#include <GL/glut.h>      // (or others, depending on the system in use)

void init (void)
{
    glClearColor (1.0, 1.0, 1.0, 0.0);  // Set display-window color to white.

    glMatrixMode (GL_PROJECTION);       // Set projection parameters.
    gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}

void lineSegment (void)
{
    glClear (GL_COLOR_BUFFER_BIT);  // Clear display window.

    glColor3f (0.0, 0.4, 0.2);      // Set line segment color to green.
    glBegin (GL_LINES);
        glVertex2i (180, 15);       // Specify line-segment geometry.
        glVertex2i (10, 145);
    glEnd ( );

    glFlush ( );     // Process all OpenGL routines as quickly as possible.
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);                          // Initialize GLUT.
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);   // Set display mode.
    glutInitWindowPosition (50, 100);   // Set top-left display-window position.
    glutInitWindowSize (400, 300);      // Set display-window width and height.
    glutCreateWindow ("An Example OpenGL Program"); // Create display window.

    init ( );                            // Execute initialization procedure.
    glutDisplayFunc (lineSegment);       // Send graphics to display window.
    glutMainLoop ( );                    // Display everything and wait.
}
```

At the end of procedure `lineSegment` is a function, `glFlush`, that we have not yet discussed. This is simply a routine to force execution of our OpenGL functions, which are stored by computer systems in buffers in different locations, depending on how OpenGL is implemented. On a busy network, for example, there could be delays in processing some buffers. But the call to `glFlush` forces all such buffers to be emptied and the OpenGL functions to be processed.

The procedure `lineSegment` that we set up to describe our picture is referred to as a *display callback function*. And this procedure is described as being "registered" by `glutDisplayFunc` as the routine to invoke whenever the display window might need to be redisplayed. This can occur, for example, if the display window is moved. In subsequent chapters, we will look at other types of callback functions and the associated GLUT routines that we use to register them. In general, OpenGL programs are organized as a set of callback functions that are to be invoked when certain actions occur.

TABLE 1

OpenGL Error Codes

Symbolic Constant	Meaning
GL_INVALID_ENUM	A GLenum argument was out of range
GL_INVALID_VALUE	A numeric argument was out of range
GL_INVALID_OPERATION	An operation was illegal in the current OpenGL state
GL_STACK_OVERFLOW	The command would cause a stack overflow
GL_STACK_UNDERFLOW	The command would cause a stack underflow
GL_OUT_OF_MEMORY	There was not enough memory to execute a command

Error Handling in OpenGL

Many of the features of the OpenGL API are extremely powerful. Unfortunately, they can also be very confusing, particularly for programmers who are just learning to use them. Although we would like to believe otherwise, it's quite possible (if not likely) that our OpenGL programs may contain errors. Accordingly, it is worth spending a few moments discussing error handling in OpenGL programs.

The OpenGL and GLU libraries have a relatively simple method of recording errors. When OpenGL detects an error in a call to a base library routine or a GLU routine, it records an error code internally, and the routine which caused the error is ignored (so that it has no effect on either the internal OpenGL state or the contents of the frame buffer). However, OpenGL only records one error code at a time. Once an error occurs, no other error code will be recorded until your program explicitly queries the OpenGL error state:

```
GLenum code;

code = glGetError ();
```

This call returns the current error code and clears the internal error flag. If the returned value is equal to the OpenGL symbolic constant GL_NO_ERROR, everything is fine. Any other return value indicates that a problem has occurred.

The base OpenGL library defines a number of symbolic constants which represent different error conditions; the ones which occur most often are listed in Table 1. The GLU library also defines a number of error codes, but most of them have almost meaningless names such as GLU_NURBS_ERROR1, GLU_NURBS_ERROR2, and so on. (These are not actually meaningless names, but their meaning won't become clear until we discuss more advanced concepts in later chapters.)

These symbolic constants are helpful, but printing them out directly is not particularly informative. Fortunately, the GLU library contains a function that returns a descriptive string for each of the GLU and GL errors. To use it, we first retrieve the current error code, and then pass it as a parameter to this function. The return value can be printed out using, for example, the C standard library fprintf function:

```
#include <stdio.h>
GLenum code;
```

```
const GLubyte *string;

code = glGetError ();
string = gluErrorString (code);
fprintf( stderr, "OpenGL error: %s\n", string );
```

The value returned by `gluErrorString` points to a string located inside the GLU library. It is not a dynamically-allocated string, so it must not be deallocated by our program. It also must not be modified by our program (hence the `const` modifier on the declaration of `string`).

We can easily encapsulate these function calls into a general error-reporting function in our program. The following function will retrieve the current error code, print the descriptive error string, and return the code to the calling routine:

```
#include <stdio.h>

GLenum errorCheck ()
{
   GLenum code;
   const GLubyte *string;

   code = glGetError ();
   if (code != GL_NO_ERROR)
   {
      string = gluErrorString (code);
      fprintf( stderr, "OpenGL error: %s\n", string );
   }

   return code;
}
```

We encourage you to use a function like this in the OpenGL programs you develop. It is usually a good idea to check for errors at least once in your display callback routine, and more frequently if you are using OpenGL features you have never used before, or if you are seeing unusual or unexpected results in the image your program produces.

6 Summary

In this chapter, we surveyed the major features of graphics software systems. Some software systems, such as CAD packages and paint programs, are designed for particular applications. Other software systems provide a library of general graphics routines that can be used within a programming language such as C++ to generate pictures for any application.

Standard graphics-programming packages developed and approved through ISO and ANSI are GKS, 3D GKS, PHIGS, and PHIGS+. Other packages that have evolved into standards are GL and OpenGL. Many other graphics libraries are available for use in a programming language, including Open Inventor, VRML, RenderMan, Java 2D, and Java 3D. Other systems, such as Mathematica, MatLab, and Maple, often provide a set of graphics-programming functions.

Normally, graphics-programming packages require coordinate specifications to be given in Cartesian reference frames. Each object for a scene can be defined

in a separate modeling Cartesian-coordinate system, which is then mapped to a world-coordinate location to construct the scene. From world coordinates, three-dimensional objects are projected to a two-dimensional plane, converted to normalized device coordinates, and then transformed to the final display-device coordinates. The transformations from modeling coordinates to normalized device coordinates are independent of particular output devices that might be used in an application. Device drivers are then used to convert normalized coordinates to integer device coordinates.

Functions that are available in graphics programming packages can be divided into the following categories: graphics output primitives, attributes, geometric and modeling transformations, viewing transformations, input functions, picture-structuring operations, and control operations.

The OpenGL system consists of a device-independent set of routines (called the core library), the utility library (GLU), and the utility toolkit (GLUT). In the auxiliary set of routines provided by GLU, functions are available for generating complex objects, for parameter specifications in two-dimensional viewing applications, for dealing with surface-rendering operations, and for performing some other supporting tasks. In GLUT, we have an extensive set of functions for managing display windows, interacting with screen-window systems, and for generating some three-dimensional shapes. We can use GLUT to interface with any computer system, or we can use GLX, Apple GL, WGL, or another system-specific software package.

REFERENCES

Standard sources for information on OpenGL are Woo, et al. (1999), Shreiner (2000), and Shreiner (2010). Open Inventor is explored in Wernecke (1994). McCarthy and Descartes (1998) can be consulted for discussions of VRML. Presentations on RenderMan can be found in Upstill (1989) and Apodaca and Gritz (2000). Examples of graphics programming in Java 2D are given in Knudsen (1999), Hardy (2000), and Horstmann and Cornell (2001). Graphics programming using Java 3D is explored in Sowizral, et al. (2000); Palmer (2001); Selman (2002); and Walsh and Gehringer (2002).

For information on PHIGS and PHIGS+, see Howard, et al. (1991); Hopgood and Duce (1991); Gaskins (1992); and Blake (1993). Information on the two-dimensional GKS standard and on the evolution of graphics standards is available in Hopgood, et al. (1983). An additional reference for GKS is Enderle, et al. (1984).

EXERCISES

1 What command could we use to set the color of an OpenGL display window to dark gray? What command would we use to set the color of the display window to white?

2 List the statements needed to set up an OpenGL display window whose lower-left corner is at pixel position (75, 200) and has a window width of 200 pixels and a height of 150 pixels.

3 List the OpenGL statements needed to draw a line segment from the upper-right corner of a display window of width 150 and height 250 to the lower-left corner of the window.

4 Suppose we have written a function called `rectangle` whose purpose is to draw a rectangle in the middle of a given display window. What OpenGL statement would be needed to make sure the rectangle is drawn correctly each time the display window needs to be repainted?

5 Explain what is meant by the term "OpenGL display callback function".

6 Explain the difference between modeling coordinates and world coordinates.

7 Explain what normalized coordinates are and why they are useful for graphics software packages.

IN MORE DEPTH

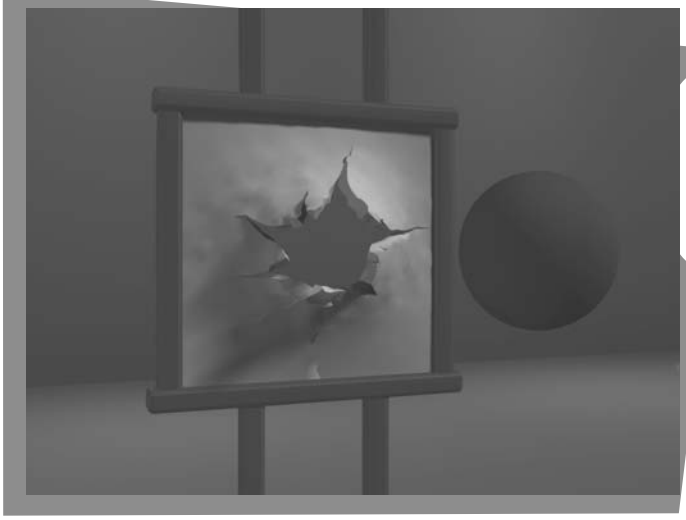
1 Develope a few application ideas and identify and describe the objects that you will be manipulating graphically in the application. Explain desirable physical and visual properties of these objects in detail so that you can concretely define what attributes you will develop for them in later exercises. Consider the following:

- Are the objects complex in shape or texture?
- Can the objects be approximated fairly well by simpler shapes?
- Are some of the objects comprised of fairly complex curved surfaces?
- Do the objects lend themselves to being represented two-dimensionally at first, even if unrealistically?
- Can the objects be represented as a hierarchically organized group of smaller objects, or parts?
- Will the objects change position and orientation dynamically in response to user input?
- Will the lighting conditions change in the application, and do the object appearances change under these varying conditions?

If all of the objects in your application yield a "no" answer to any of these questions, consider modifying the application or your design and implementation approach so that such properties are present in at least one of the objects. Alternatively, come up with two or more applications so that at least one of them contains objects that satisfy each of these properties. Draw a rough specification, using visual diagrams and/or text, listing the properties of the objects in your application. You will use and modify this specification as you continue to develop your application.

2 You will be developing your application using the OpenGL API. In order to do this, you will need to have a working programming environment in which you can edit, compile and run OpenGL programs. For this exercise, you should go through the necessary steps to get this environment up and running on your machine. Once you have the environment set up, create a new project with the source code given in the example in this chapter that draws a single line in the display window. Make sure you can compile and run this program. Information about getting an OpenGL environment up and running on your system can be found at http://www.opengl.org/ and http://www.opengl.org/wiki/Getting_started. Make sure that you obtain all of the necessary libraries, including the GLU and GLUT libraries. The examples in this text are all written in C++. Follow your instructor's guidelines on the use of other languages (provided OpenGL bindings can be obtained for them).

Graphics 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.

1 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 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.

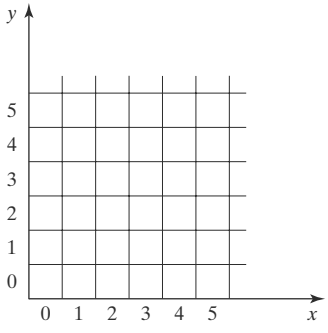


FIGURE 1
Pixel positions referenced with respect to the lower-left corner of a screen area.

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.

2 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.

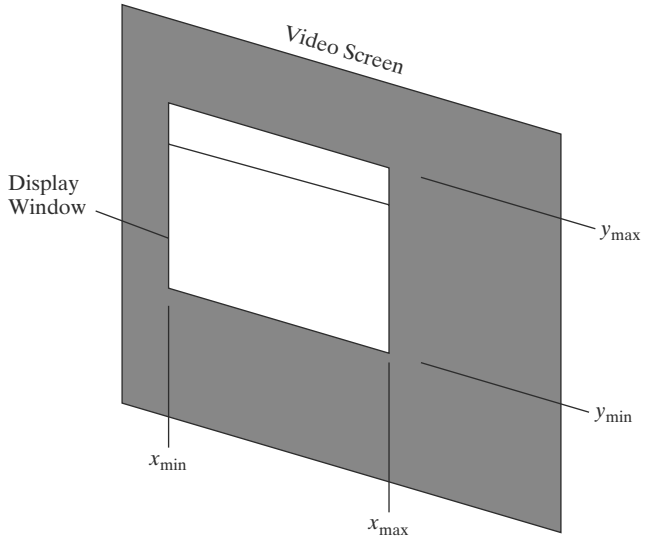


FIGURE 2
World-coordinate limits for a display window, as specified in the `glOrtho2D` function.

3 OpenGL Point Functions

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 data type 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-value 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.

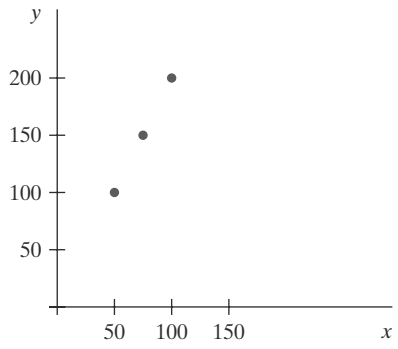


FIGURE 3
Display of three point positions generated with `glBegin (GL_POINTS)`.

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, 188.33);
glEnd ( );
```

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

```
class wcPt2D {
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.

4 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 line endpoint. Nothing is displayed if we do not list at least two coordinate positions.

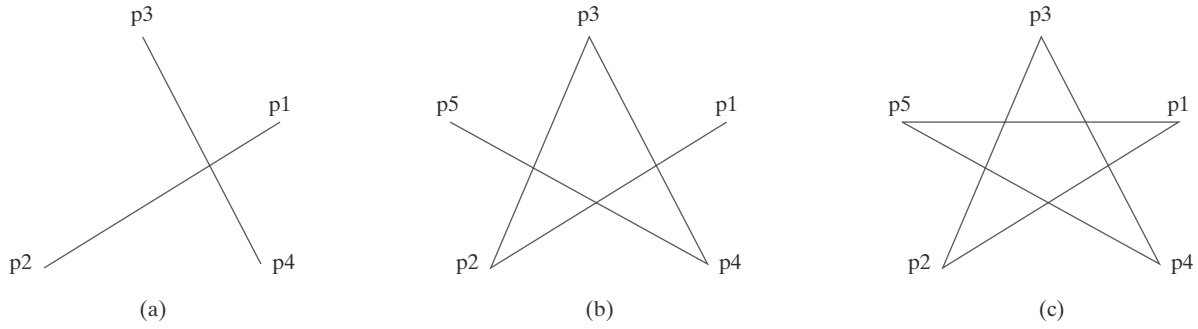


FIGURE 4
Line segments that can be displayed in OpenGL using a list of five endpoint coordinates. (a) An unconnected set of lines generated with the primitive line constant `GL_LINES`. (b) A polyline generated with `GL_LINE_STRIP`. (c) A closed polyline generated with `GL_LINE_LOOP`.

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

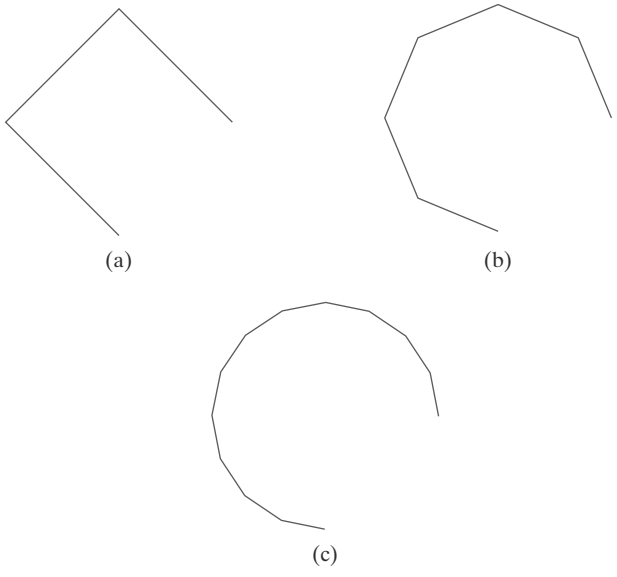


FIGURE 5
A circular arc approximated with (a) three straight-line segments, (b) six line segments, and (c) twelve line segments.

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

(a) (b) (c)

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.

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.**

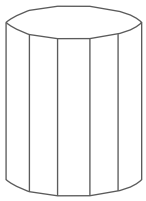


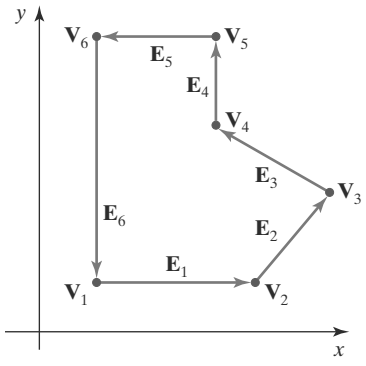
FIGURE 7
Wire-frame representation for a cylinder, showing only the front (visible) faces of the polygon mesh used to approximate the surfaces.

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



$(\mathbf{E}_1 \times \mathbf{E}_2)_z > 0$
$(\mathbf{E}_2 \times \mathbf{E}_3)_z > 0$
$(\mathbf{E}_3 \times \mathbf{E}_4)_z < 0$
$(\mathbf{E}_4 \times \mathbf{E}_5)_z > 0$
$(\mathbf{E}_5 \times \mathbf{E}_6)_z > 0$
$(\mathbf{E}_6 \times \mathbf{E}_1)_z > 0$

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, $\mathbf{V}_k$ and $\mathbf{V}_{k+1}$, we define the edge vector between them as

$$\mathbf{E}_k = \mathbf{V}_{k+1} - \mathbf{V}_k$$

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.

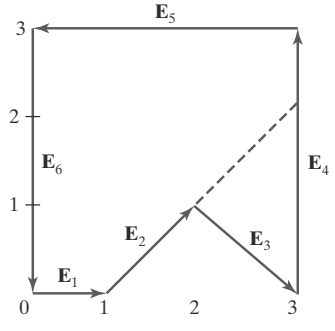


FIGURE 10
Splitting a concave polygon using the vector method.

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

$$\mathbf{E}_1 = (1, 0, 0) \qquad \mathbf{E}_2 = (1, 1, 0)$$
$$\mathbf{E}_3 = (1, -1, 0) \qquad \mathbf{E}_4 = (0, 2, 0)$$
$$\mathbf{E}_5 = (-3, 0, 0) \qquad \mathbf{E}_6 = (0, -2, 0)$$

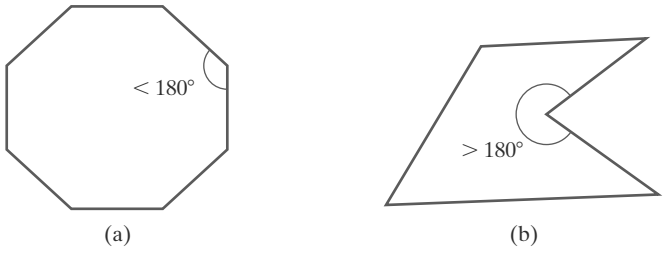


FIGURE 8
A convex polygon (a), and a concave polygon (b).

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°, 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°. 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 $E_{jx}E_{ky} - E_{kx}E_{jy}$:

$$\mathbf{E}_1 \times \mathbf{E}_2 = (0, 0, 1) \qquad \mathbf{E}_2 \times \mathbf{E}_3 = (0, 0, -2)$$
$$\mathbf{E}_3 \times \mathbf{E}_4 = (0, 0, 2) \qquad \mathbf{E}_4 \times \mathbf{E}_5 = (0, 0, 6)$$
$$\mathbf{E}_5 \times \mathbf{E}_6 = (0, 0, 6) \qquad \mathbf{E}_6 \times \mathbf{E}_1 = (0, 0, 2)$$

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.

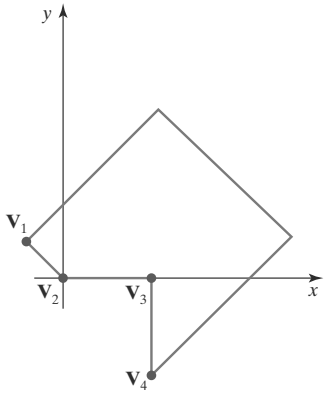


FIGURE 11
Splitting a concave polygon using the rotational method. After moving $\mathbf{V}_2$ to the coordinate origin and rotating $\mathbf{V}_3$ onto the x axis, we find that $\mathbf{V}_4$ is below the x axis. So we split the polygon along the line of $\overline{\mathbf{V}_2\mathbf{V}_3}$, which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

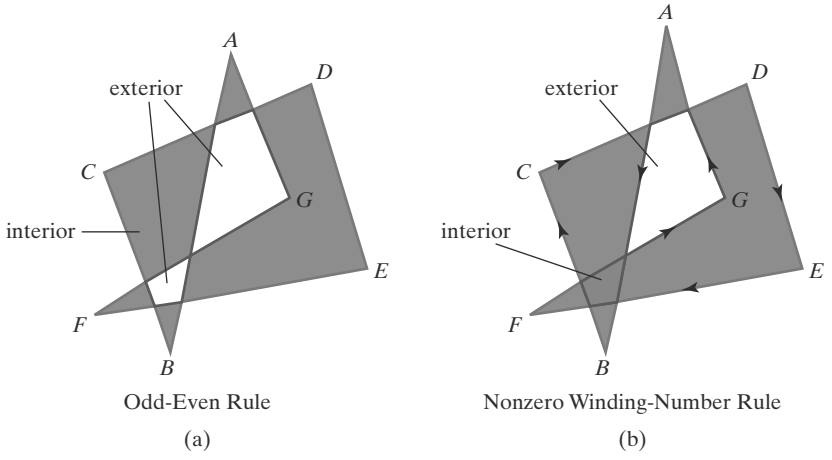


FIGURE 12
Identifying interior and exterior regions of a closed polyline that contains self-intersecting segments.

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 **u**, along the line from **P** to a distant point, with an object edge vector **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 **u** and that has a right-to-left direction as we look along the line from **P** in the direction of **u.** If the components of **u** are denoted as (u_x, u_y), then the vector that is perpendicular to **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 **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.

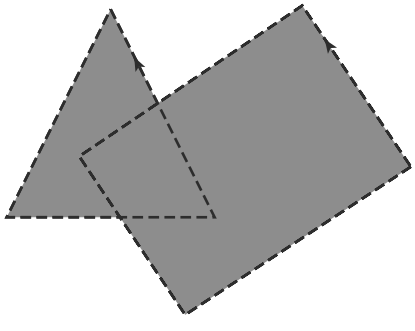


FIGURE 13
A fill area defined as a region that has a positive value for the winding number. This fill area is the union of two regions, each with a counterclockwise border direction.

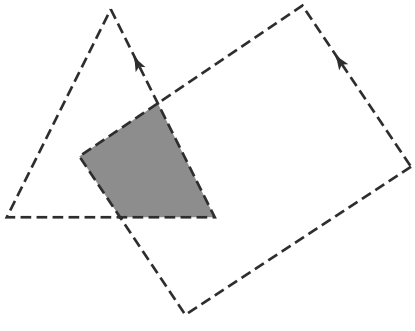


FIGURE 14
A fill area defined as a region with a winding number greater than 1. This fill area is the intersection of two regions, each with a counterclockwise border direction.

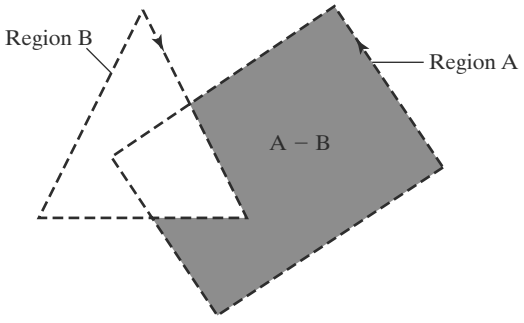
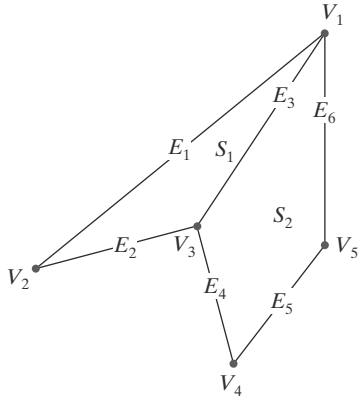


FIGURE 15
A fill area defined as a region with a positive value for the winding number. This fill area is the difference, A — B, of two regions, where region A has a positive border direction (counterclockwise) and region B has a negative border direction (clockwise).

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



VERTEX TABLE	
V_1:	x_1, y_1, z_1
V_2:	x_2, y_2, z_2
V_3:	x_3, y_3, z_3
V_4:	x_4, y_4, z_4
V_5:	x_5, y_5, z_5

EDGE TABLE	
E_1:	V_1, V_2
E_2:	V_2, V_3
E_3:	V_3, V_1
E_4:	V_3, V_4
E_5:	V_4, V_5
E_6:	V_5, V_1

SURFACE-FACET TABLE	
S_1:	E_1, E_2, E_3
S_2:	E_3, E_4, E_5, E_6

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.

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.

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.

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

$$A\,x + B\,y + C\,z + D = 0 \tag{1}$$

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:

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

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

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

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

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

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

$$A x + B y + C z + D \neq 0$$

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 $A x + B y + C z + D < 0$, the point (x, y, z) is behind the plane

if $A x + B y + C z + D > 0$, 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.

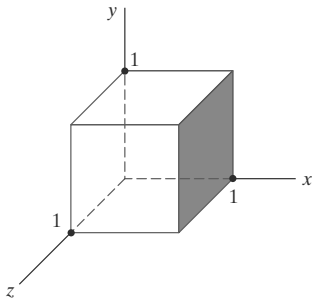


FIGURE 18
The shaded polygon surface of the unit cube has the plane equation $x - 1 = 0$.

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.

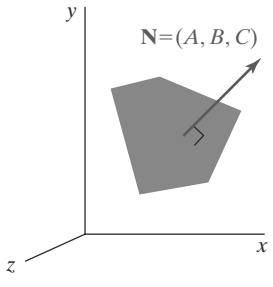


FIGURE 19
The normal vector **N** for a plane described with the equation $Ax + By + Cz + D = 0$ is perpendicular to the plane and has Cartesian components (A, B, C).

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:

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

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

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

$$\mathbf{N} \cdot \mathbf{P} = -D \tag{6}$$

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°. 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°. 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.

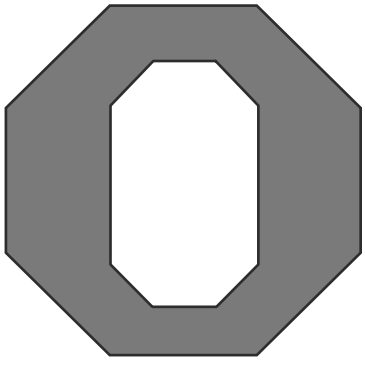


FIGURE 20
A polygon with a complex interior that cannot be specified with a single vertex list.

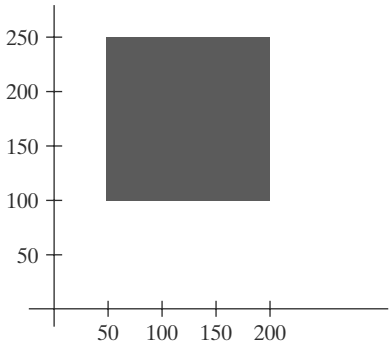


FIGURE 21
The display of a square fill area using the `glRect` function.

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:

```
glRect* (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 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

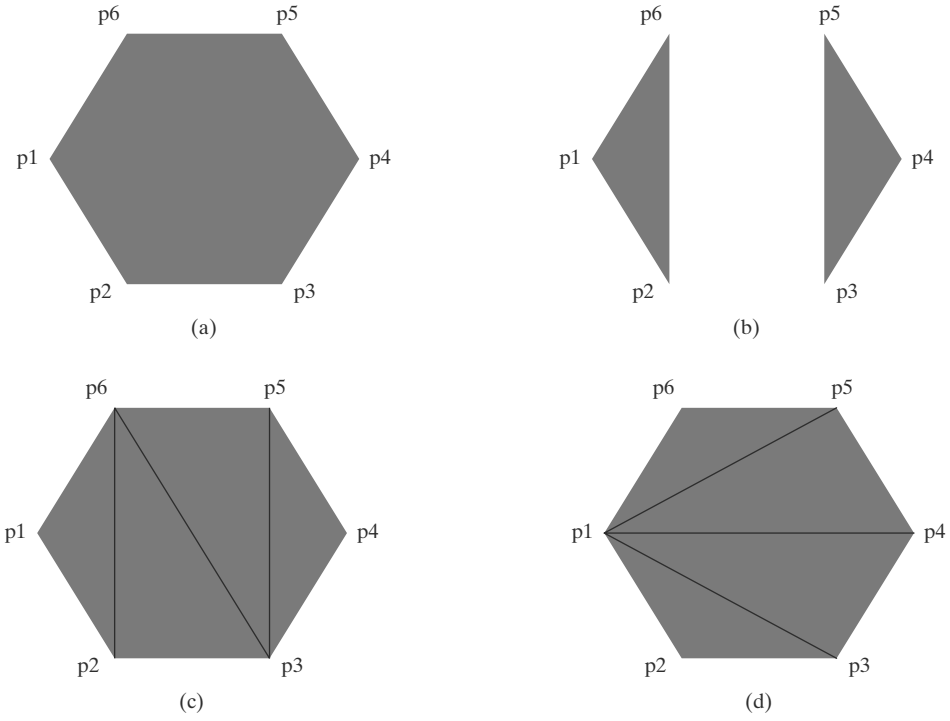


FIGURE 22
Displaying polygon fill areas using a list of six vertex positions. (a) A single convex polygon fill area generated with the primitive constant `GL_POLYGON`. (b) Two unconnected triangles generated with `GL_TRIANGLES`. (c) Four connected triangles generated with `GL_TRIANGLE_STRIP`. (d) Four 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);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p6);
    glVertex2iv (p3);
    glVertex2iv (p5);
    glVertex2iv (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 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 tables. This is accomplished by processing each position n in the vertex list in the order $n = 1$, $n = 2, \ldots, n = N - 2$ 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);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p3);
    glVertex2iv (p4);
    glVertex2iv (p5);
    glVertex2iv (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

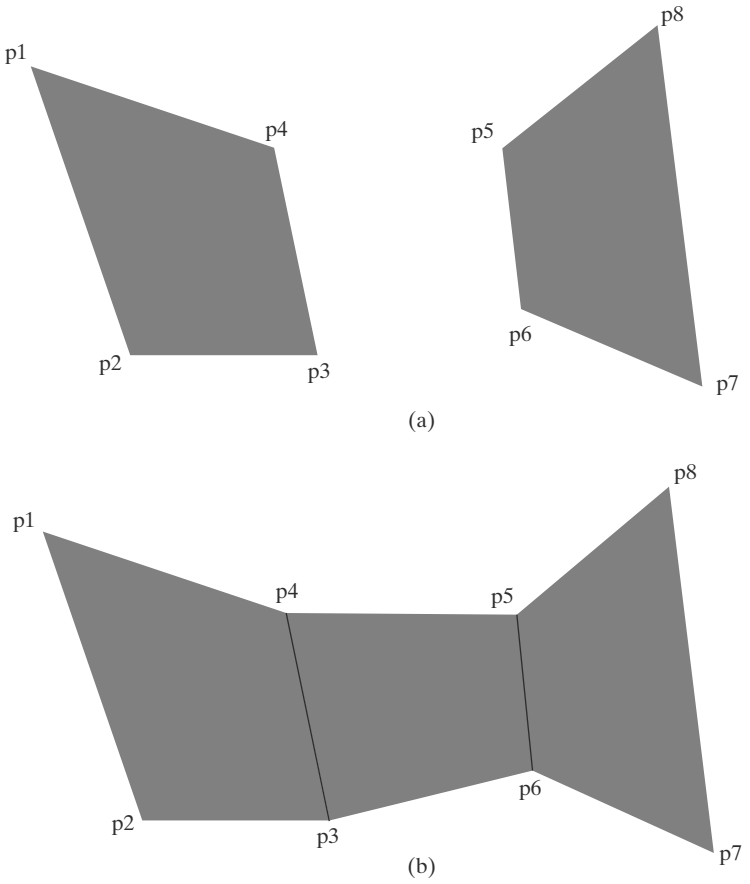


FIGURE 23
Displaying quadrilateral fill areas using a list of eight vertex positions. (a) Two unconnected quadrilaterals generated with `GL_QUADS`. (b) Three connected quadrilaterals generated with `GL_QUAD_STRIP`.

be specified in the proper order to define front and back faces for each triangle correctly. The first coordinate position listed (in this case, p1) is a vertex for each triangle in the fan. If we again enumerate the triangles and the coordinate positions listed as $n = 1, n = 2, \ldots, n = N - 2$, 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 (p1, p2, p3); triangle 2 has the vertex ordering (p1, p3, p4); triangle 3 has its vertices specified in the order (p1, p4, p5); and triangle 4 is listed with vertices (p1, p5, p6).

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);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p4);
    glVertex2iv (p3);
    glVertex2iv (p5);
    glVertex2iv (p6);
    glVertex2iv (p8);
    glVertex2iv (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 $n = 1$, $n = 2$, ..., $n = \frac{N}{2} - 1$. 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.

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

```
GLint 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 GLint 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.

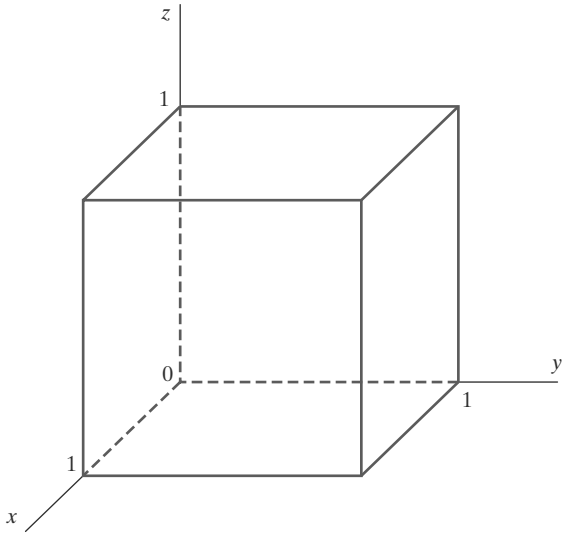


FIGURE 24
A cube with an edge length of 1.

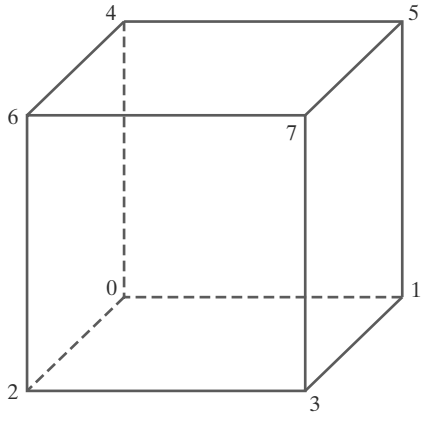


FIGURE 25
Subscript values for array `pt` corresponding to the vertex coordinates for the cube shown in Figure 24.

```
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, 24, GL_UNSIGNED_BYTE, vertIndex);
```

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.*

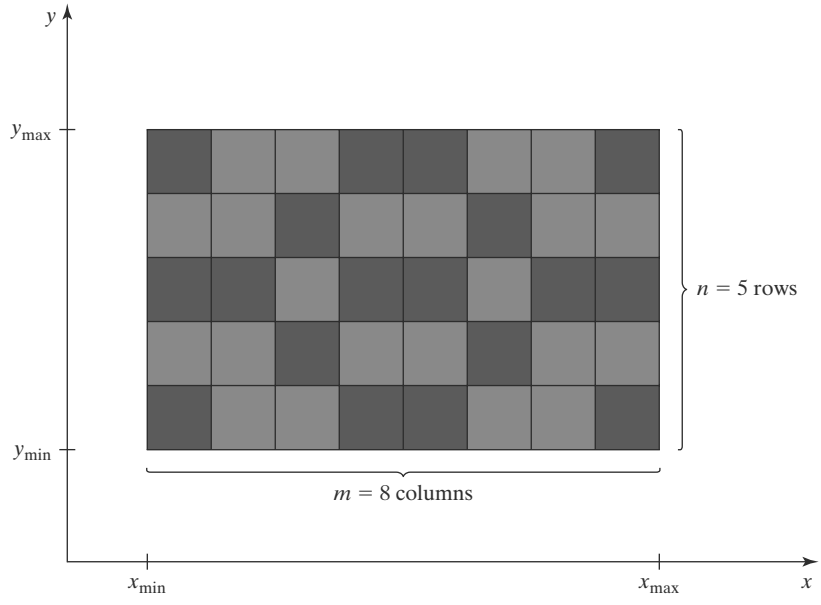


FIGURE 26
Mapping an *n* by *m* color array onto a region of the screen coordinates.

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, (`x0`, `y0`), 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, 0x80, 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);
```

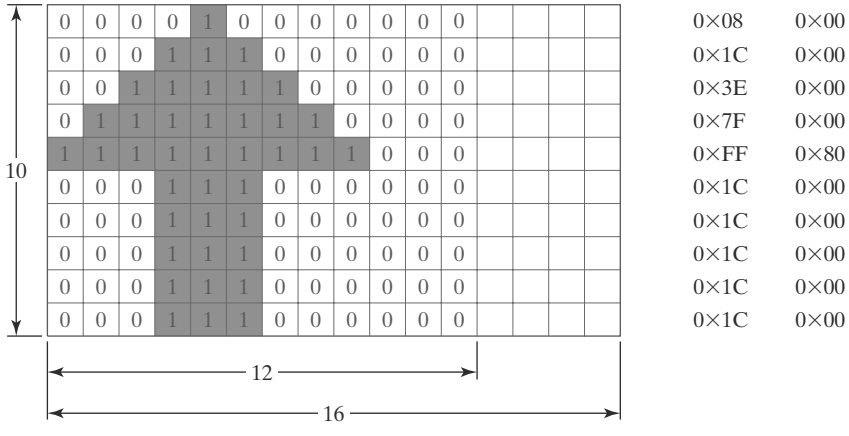


FIGURE 27
A bit pattern, specified in an array with 10 rows and 9 columns, is stored in 8-bit blocks of 10 rows with 16 bit values per row.

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,

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 width 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 PostScript, 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,

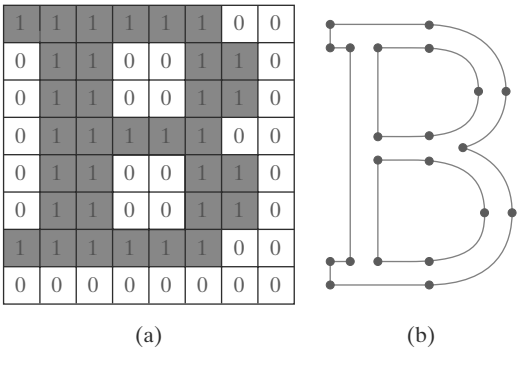
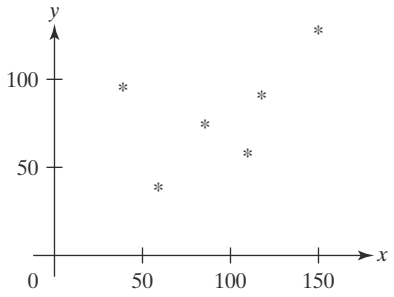


FIGURE 28
The letter "B" represented with an 8 × 8 bitmap pattern (a) and with an outline shape defined with straight-line and curve segments (b).



x	y
41	94
59	43
85	74
110	59
121	89
149	122

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 fromvarious 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;

GLdouble theta;
GLint 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

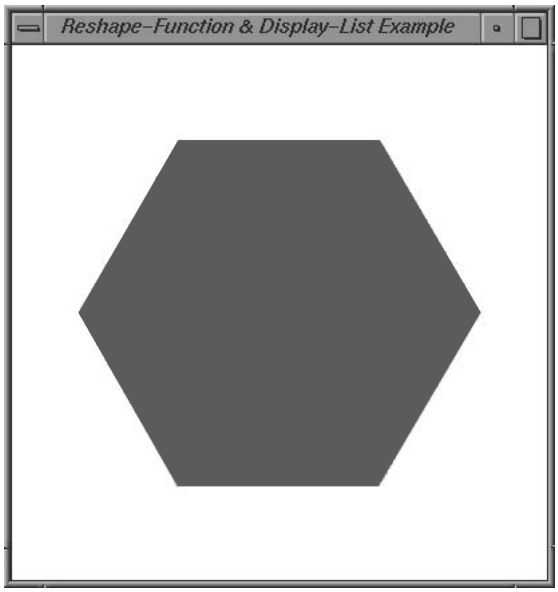


FIGURE 30
The display window generated by the example program illustrating the use of the reshape function.

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.  */
GLsizei winWidth = 400, winHeight = 400;
GLuint regHex;

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

```
      public:
         /*  Default Constructor: initializes coordinate position to (0, 0).  */
         screenPt ( )  {
             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 hexVertex, circCtr;
   GLdouble 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;
               hexVertex.setCoords (circCtr.getx ( ) + 150 * cos (theta),
                                    circCtr.gety ( ) + 150 * sin (theta));
               glVertex2i (hexVertex.getx ( ), hexVertex.gety ( ));
         }
      glEnd ( );
   glEndList ( );
}

void regHexagon (void)
{
   glClear (GL_COLOR_BUFFER_BIT);

   glCallList (regHex);

   glFlush ( );
}
```

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

    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 ("Reshape-Function & Display-List Example");

    init ( );
    glutDisplayFunc (regHexagon);
    glutReshapeFunc (winReshapeFcn);

    glutMainLoop ( );
}
```

17 Summary

The output primitives discussed in this chapter provide the basic tools for constructing pictures with individual points, straight lines, curves, filled color areas, array patterns, and text. We specify primitives by giving their geometric descriptions in a Cartesian, world-coordinate reference system.

A fill area is a planar region that is to be displayed in a solid color or color pattern. Fill-area primitives in most graphics packages are polygons. But, in general, we could specify a fill region with any boundary. Often, graphics systems allow only convex polygon fill areas. In that case, a concave-polygon fill area can be displayed by dividing it into a set of convex polygons. Triangles are the easiest polygons to fill, because each scan line crossing a triangle intersects exactly two polygon edges (assuming that the scan line does not pass through any vertices).

The odd-even rule can be used to locate the interior points of a planar region. Other methods for defining object interiors are also useful, particularly with irregular, self-intersecting objects. A common example is the nonzero winding-number rule. This rule is more flexible than the odd-even rule for handling objects defined with multiple boundaries. We can also use variations of the winding-number rule to combine plane areas using Boolean operations.

Each polygon has a front face and a back face, which determines the spatial orientation of the polygon plane. This spatial orientation can be determined from the normal vector, which is perpendicular to the polygon plane and points

in the direction from the back face to the front face. We can determine the components of the normal vector from the polygon plane equation or by forming a vector cross-product using three points in the plane, where the three points are taken in a counterclockwise order and the angle formed by the three points is less than 180°. All coordinate values, spatial orientations, and other geometric data for a scene are entered into three tables: vertex, edge, and surface-facet tables.

Additional primitives available in graphics packages include pattern arrays and character strings. Pattern arrays can be used to specify two-dimensional shapes, including a character set, using either a rectangular set of binary values or a set of color values. Character strings are used to provide picture and graph labeling.

Using the primitive functions available in the basic OpenGL library, we can generate points, straight-line segments, convex polygon fill areas, and either bitmap or pixmap pattern arrays. Routines for displaying character strings are available in GLUT. Other types of primitives, such as circles, ellipses, and concave-polygon fill areas, can be constructed or approximated with these functions, or they can be generated using routines in GLU and GLUT. All coordinate values are expressed in absolute coordinates within a right-handed Cartesian-coordinate reference system. Coordinate positions describing a scene can be given in either a two-dimensional or a three-dimensional reference frame. We can use integer or floating-point values to give a coordinate position, and we can also reference a position with a pointer to an array of coordinate values. A scene description is then transformed by viewing functions into a two-dimensional display on an output device, such as a video monitor. Except for the `glRect` function, each coordinate position for a set of points, lines, or polygons is specfied in a `glVertex` function. And the set of `glVertex` functions defining each primitive is included between a `glBegin/glEnd` pair of statements, where the primitive type is identified with a symbolic constant as the argument for the `glBegin` function. When describing a scene containing many polygon fill surfaces, we can generate the display efficiently using OpenGL vertex arrays to specify geometric and other data.

In Table 1, we list the basic functions for generating output primitives in OpenGL. Some related routines are also listed in this table.

Example Programs

Here, we present a few example OpenGL programs illustrating the use of output primitives. Each program uses one or more of the functions listed in Table 1. A display window is set up for the output from each program.

The first program illustrates the use of a polyline, a set of polymarkers, and bit-mapped character labels to generate a line graph for monthly data over a period of one year. A proportionally spaced font is demonstrated, although a fixed-width font is usually easier to align with graph positions. Because the bitmaps are referenced at the lower-left corner by the raster-position function, we must shift the reference position to align the center of a text string with a plotted data position. Figure 31 shows the output of the line-graph program.

TABLE 1

Summary of OpenGL Output Primitive Functions and Related Routines

Function	Description
`gluOrtho2D`	Specifies a two-dimensional world-coordinate reference.
`glVertex*`	Selects a coordinate position. This function must be placed within a `glBegin/glEnd` pair.
`glBegin (GL_POINTS);`	Plots one or more point positions, each specified in a `glVertex` function. The list of positions is then closed with a `glEnd` statement.
`glBegin (GL_LINES);`	Displays a set of straight-line segments, whose endpoint coordinates are specified in `glVertex` functions. The list of endpoints is then closed with a `glEnd` statement.
`glBegin (GL_LINE_STRIP);`	Displays a polyline, specified using the same structure as `GL_LINES`.
`glBegin (GL_LINE_LOOP);`	Displays a closed polyline, specified using the same structure as `GL_LINES`.
`glRect*`	Displays a fill rectangle in the xy plane.
`glBegin (GL_POLYGON);`	Displays a fill polygon, whose vertices are given in `glVertex` functions and terminated with a `glEnd` statement.
`glBegin (GL_TRIANGLES);`	Displays a set of fill triangles using the same structure as `GL_POLYGON`.
`glBegin (GL_TRIANGLE_STRIP);`	Displays a fill-triangle mesh, specified using the same structure as `GL_POLYGON`.
`glBegin (GL_TRIANGLE_FAN);`	Displays a fill-triangle mesh in a fan shape with all triangles connected to the first vertex, specified with same structure as `GL_POLYGON`.
`glBegin (GL_QUADS);`	Displays a set of fill quadrilaterals, specified with the same structure as `GL_POLYGON`.
`glBegin (GL_QUAD_STRIP);`	Displays a fill-quadrilateral mesh, specified with the same structure as `GL_POLYGON`.
`glEnableClientState (GL_VERTEX_ARRAY);`	Activates vertex-array features of OpenGL.
`glVertexPointer (size, type, stride, array);`	Specifies an array of coordinate values.
`glDrawElements (prim, num, type, array);`	Displays a specified primitive type from array data.

TABLE 1

(*continued*)

Function	Description
`glNewList (listID, listMode)`	Defines a set of commands as a display list, terminated with a `glEndList` statement.
`glGenLists`	Generates one or more display-list identifiers.
`glIsList`	Queries OpenGL to determine whether a display-list identifier is in use.
`glCallList`	Executes a single display list.
`glListBase`	Specifies an offset value for an array of display-list identifiers.
`glCallLists`	Executes multiple display lists.
`glDeleteLists`	Eliminates a specified sequence of display lists.
`glRasterPos*`	Specifies a two-dimensional or three-dimensional current position for the frame buffer. This position is used as a reference for bitmap and pixmap patterns.
`glBitmap (w, h, x0, y0, xShift, yShift, pattern);`	Specifies a binary pattern that is to be mapped to pixel positions relative to the current position.
`glDrawPixels (w, h, type, format, pattern);`	Specifies a color pattern that is to be mapped to pixel positions relative to the current position.
`glDrawBuffer`	Selects one or more buffers for storing a pixmap.
`glReadPixels`	Saves a block of pixels in a selected array.
`glCopyPixels`	Copies a block of pixels from one buffer position to another.
`glLogicOp`	Selects a logical operation for combining two pixel arrays, after enabling with the constant `GL_COLOR_LOGIC_OP`.
`glutBitmapCharacter (font, char);`	Specifies a font and a bitmap character for display.
`glutStrokeCharacter (font, char);`	Specifies a font and an outline character for display.
`glutReshapeFunc`	Specifies actions to be taken when display-window dimensions are changed.

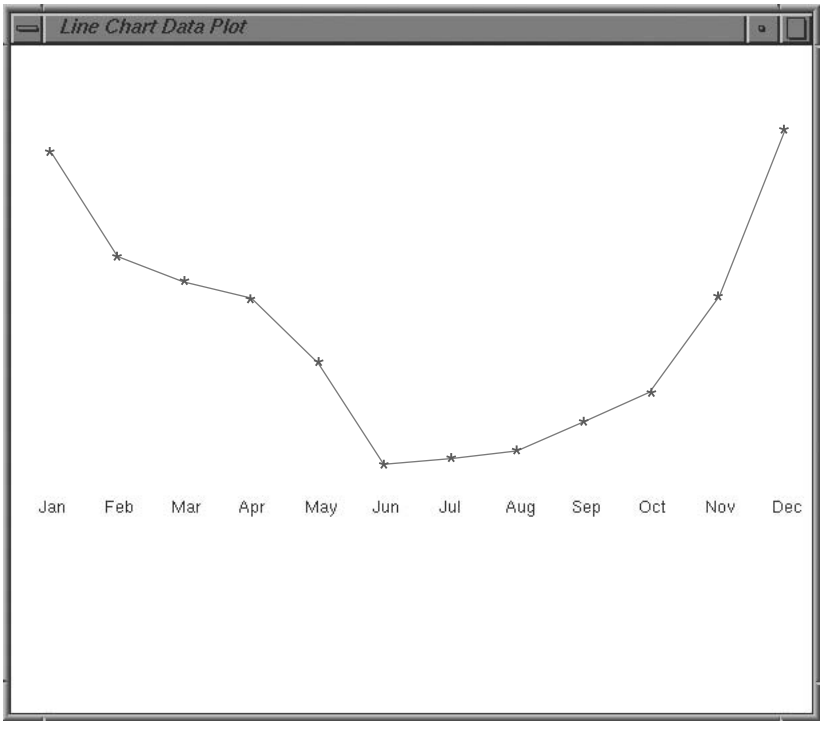


FIGURE 31
A polyline and polymarker plot of data points output by the `lineGraph` routine.

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

GLsizei winWidth = 600, winHeight = 500;    // Initial display window size.
GLint xRaster = 25, yRaster = 150;          // Initialize raster position.

GLubyte label [36] = {'J', 'a', 'n',   'F', 'e', 'b',   'M', 'a', 'r',
                      'A', 'p', 'r',   'M', 'a', 'y',   'J', 'u', 'n',
                      'J', 'u', 'l',   'A', 'u', 'g',   'S', 'e', 'p',
                      'O', 'c', 't',   'N', 'o', 'v',   'D', 'e', 'c'};

GLint dataValue [12] = {420, 342, 324, 310, 262, 185,
                        190, 196, 217, 240, 312, 438};

void init (void)
{
    glClearColor (1.0, 1.0, 1.0, 1.0);    // White display window.
    glMatrixMode (GL_PROJECTION);
    gluOrtho2D (0.0, 600.0, 0.0, 500.0);
}

void lineGraph (void)
{
    GLint month, k;
    GLint x = 30;                         // Initialize x position for chart.

    glClear (GL_COLOR_BUFFER_BIT);        //  Clear display window.
    glColor3f (0.0, 0.0, 1.0);            //  Set line color to blue.
```

```
    glBegin (GL_LINE_STRIP);             //  Plot data as a polyline.
        for (k = 0; k < 12; k++)
            glVertex2i (x + k*50, dataValue [k]);
    glEnd ( );


    glColor3f (1.0, 0.0, 0.0);           //  Set marker color to red.
    for (k = 0; k < 12; k++) {           //  Plot data as asterisk polymarkers.
        glRasterPos2i (xRaster + k*50, dataValue [k] - 4);
        glutBitmapCharacter (GLUT_BITMAP_9_BY_15, '*');
    }


    glColor3f (0.0, 0.0, 0.0);           //  Set text color to black.
    xRaster = 20;                        //  Display chart labels.
    for (month = 0; month < 12; month++) {
        glRasterPos2i (xRaster, yRaster);
        for (k = 3*month; k < 3*month + 3; k++)
          glutBitmapCharacter (GLUT_BITMAP_HELVETICA_12, label [k]);
        xRaster += 50;
    }
    glFlush ( );
}


void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ( );
    gluOrtho2D (0.0, GLdouble (newWidth), 0.0, GLdouble (newHeight));

    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 ("Line Chart Data Plot");

    init ( );
    glutDisplayFunc (lineGraph);
    glutReshapeFunc (winReshapeFcn);

    glutMainLoop ( );
}
```

We use the same data set in the second program to produce the bar chart in Figure 32. This program illustrates an application of rectangular fill areas, as well as bitmapped character labels.

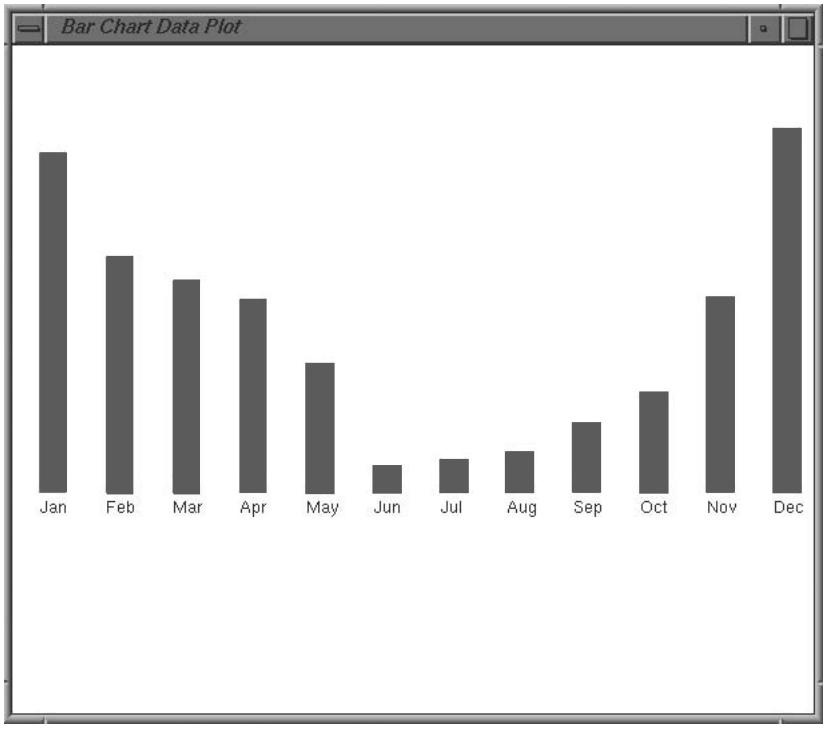


FIGURE 32
A bar chart generated by the `barChart` procedure.

```
void barChart (void)
{
   GLint month, k;

   glClear (GL_COLOR_BUFFER_BIT); //  Clear display window.

   glColor3f (1.0, 0.0, 0.0);     //  Set bar color to red.
   for (k = 0; k < 12; k++)
      glRecti (20 + k*50, 165, 40 + k*50, dataValue [k]);

  glColor3f (0.0, 0.0, 0.0);       //  Set text color to black.
  xRaster = 20;                    //  Display chart labels.
  for (month = 0; month < 12; month++) {
      glRasterPos2i (xRaster, yRaster);
      for (k = 3*month; k < 3*month + 3; k++)
         glutBitmapCharacter (GLUT_BITMAP_HELVETICA_12,
                                              label [h]);
      xRaster += 50;
   }
   glFlush ( );
}
```

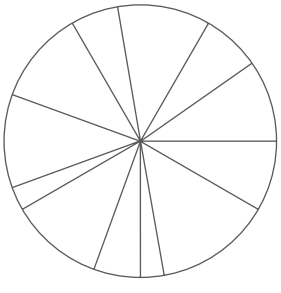


FIGURE 33
Output produced with the `pieChart` procedure.

Pie charts are used to show the percentage contribution of individual parts to the whole. The next program constructs a pie chart, using the midpoint routine for generating a circle. Example values are used for the number and relative sizes of the slices, and the output from this program appears in Figure 33.

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

const GLdouble twoPi = 6.283185;

class scrPt {
public:
    GLint x, y;
};

GLsizei winWidth = 400, winHeight = 300;    // Initial display window size.

void init (void)
{
    glClearColor (1.0, 1.0, 1.0, 1.0);

    glMatrixMode (GL_PROJECTION);
    gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}

    .                        //  Midpoint routines for displaying a circle.
    .
    .

void pieChart (void)
{
    scrPt circCtr, piePt;
    GLint radius = winWidth / 4;               // Circle radius.

    GLdouble sliceAngle, previousSliceAngle = 0.0;

    GLint k, nSlices = 12;                     // Number of slices.
    GLfloat dataValues[12] = {10.0, 7.0, 13.0, 5.0, 13.0, 14.0,
                              3.0, 16.0, 5.0, 3.0, 17.0, 8.0};
    GLfloat dataSum = 0.0;

    circCtr.x = winWidth / 2;                  // Circle center position.
    circCtr.y = winHeight / 2;
    circleMidpoint (circCtr, radius);  // Call a midpoint circle-plot routine.

    for (k = 0; k < nSlices; k++)
        dataSum += dataValues[k];

    for (k = 0; k < nSlices; k++) {
        sliceAngle = twoPi * dataValues[k] / dataSum + previousSliceAngle;
        piePt.x = circCtr.x + radius * cos (sliceAngle);
        piePt.y = circCtr.y + radius * sin (sliceAngle);
        glBegin (GL_LINES);
            glVertex2i (circCtr.x, circCtr.y);
            glVertex2i (piePt.x, piePt.y);
        glEnd ( );
        previousSliceAngle = sliceAngle;
    }
}
```

```
void displayFcn (void)
{
    glClear (GL_COLOR_BUFFER_BIT);   //  Clear display window.

    glColor3f (0.0, 0.0, 1.0);       //  Set circle color to blue.

    pieChart ( );
    glFlush ( );
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ( );
    gluOrtho2D (0.0, GLdouble (newWidth), 0.0, GLdouble (newHeight));

    glClear (GL_COLOR_BUFFER_BIT);

    /*  Reset display-window size parameters.  */
    winWidth = newWidth;
    winHeight = newHeight;
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Pie Chart");

    init ( );
    glutDisplayFunc (displayFcn);
    glutReshapeFunc (winReshapeFcn);

    glutMainLoop ( );
}
```

Some variations on the circle equations are displayed by our last example program, which uses the parametric polar equations (6-28) to compute points along the curve paths. These points are then used as the endpoint positions for straight-line sections, displaying the curves as approximating polylines. The curves shown in Figure 34 are generated by varying the radius r of a circle. Depending on how we vary r, we can produce a limaçon, cardioid, spiral, or other similar figure.

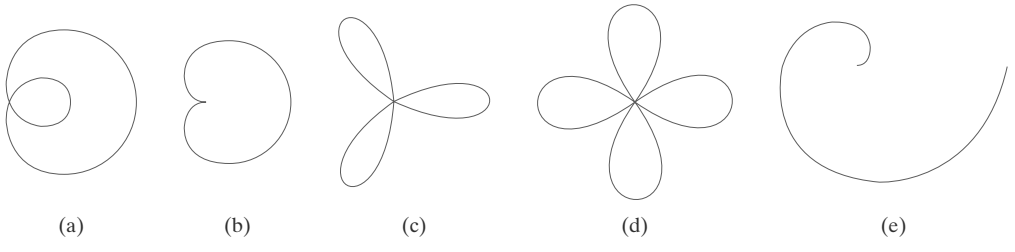


FIGURE 34
Curved figures displayed by the `drawCurve` procedure: (a) limaçon, (b) cardioid, (c) three-leaf curve, (d) four-leaf curve, and (e) spiral.

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

#include <iostream.h>

struct screenPt
{
    GLint x;
    GLint y;
};

typedef enum { limacon = 1, cardioid, threeLeaf, fourLeaf, spiral } curveName;

GLsizei winWidth = 600, winHeight = 500;    // Initial display window size.

void init (void)
{
    glClearColor (1.0, 1.0, 1.0, 1.0);

    glMatrixMode (GL_PROJECTION);
    gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}

void lineSegment (screenPt pt1, screenPt pt2)
{
    glBegin (GL_LINES);
        glVertex2i (pt1.x, pt1.y);
        glVertex2i (pt2.x, pt2.y);
    glEnd ( );
}

void drawCurve (GLint curveNum)
{
    /*  The limacon of Pascal is a modification of the circle equation
     *  with the radius varying as r = a * cos (theta) + b, where a
     *  and b are constants.  A cardioid is a limacon with a = b.
     *  Three-leaf and four-leaf curves are generated when
     *  r = a * cos (n * theta), with n = 3 and n = 2, respectively.
     *  A spiral is displayed when r is a multiple of theta.
     */

    const GLdouble twoPi = 6.283185;
    const GLint a = 175, b = 60;

    GLfloat r, theta, dtheta = 1.0 / float (a);
    GLint x0 = 200, y0 = 250;   // Set an initial screen position.
    screenPt curvePt[2];

    glColor3f (0.0, 0.0, 0.0);       //  Set curve color to black.

    curvePt[0].x = x0;      // Initialize curve position.
    curvePt[0].y = y0;
```

```
    switch (curveNum) {
        case limacon:    curvePt[0].x += a + b;  break;
        case cardioid:   curvePt[0].x += a + a;  break;
        case threeLeaf:  curvePt[0].x += a;      break;
        case fourLeaf:   curvePt[0].x += a;      break;
        case spiral:     break;
        default:         break;
    }

    theta = dtheta;
    while (theta < two_Pi) {
        switch (curveNum) {
            case limacon:
                r = a * cos (theta) + b;    break;
            case cardioid:
                r = a * (1 + cos (theta));  break;
            case threeLeaf:
                r = a * cos (3 * theta);    break;
            case fourLeaf:
                r = a * cos (2 * theta);    break;
            case spiral:
                r = (a / 4.0) * theta;      break;
            default:                        break;
        }

        curvePt[1].x = x0 + r * cos (theta);
        curvePt[1].y = y0 + r * sin (theta);
        lineSegment (curvePt[0], curvePt[1]);

        curvePt[0].x = curvePt[1].x;
        curvePt[0].y = curvePt[1].y;
        theta += dtheta;
    }
}

void displayFcn (void)
{
    GLint curveNum;

    glClear (GL_COLOR_BUFFER_BIT);   //  Clear display window.

    cout << "\nEnter the integer value corresponding to\n";
    cout << "one of the following curve names.\n";
    cout << "Press any other key to exit.\n";
    cout << "\n1-limacon, 2-cardioid, 3-threeLeaf, 4-fourLeaf, 5-spiral:  ";
    cin  >> curveNum;

    if (curveNum == 1 || curveNum == 2 || curveNum == 3 || curveNum == 4
         || curveNum == 5)
        drawCurve (curveNum);
    else
        exit (0);

    glFlush ( );
}
```

```
void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ( );
    gluOrtho2D (0.0, (GLdouble) newWidth, 0.0, (GLdouble) newHeight);

    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 ("Draw Curves");

    init ( );
    glutDisplayFunc (displayFcn);
    glutReshapeFunc (winReshapeFcn);

    glutMainLoop ( );
}
```

REFERENCES

Basic information on Bresenham's algorithms can be found in Bresenham (1965 and 1977). For midpoint methods, see Kappel (1985). Parallel methods for generating lines and circles are discussed in Pang (1990) and in Wright (1990). Many other methods for generating and processing graphics primitives are discussed in Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994), and Paeth (1995).

Additional programming examples using OpenGL primitive functions are given in Woo et al. (1999). A listing of all OpenGL primitive functions is available in Shreiner (2000). For a complete reference to GLUT, see Kilgard (1996).

EXERCISES

1 Set up geometric data tables as in Figure 16 for a square pyramid (a square base with four triangular sides that meet at a pinnacle).

2 Set up geometric data tables for a square pyramid using just a vertex table and a surface-facet table, then store the same information using just the surface-facet table. Compare the two methods for representing the unit cube with a representation using the three tables in the previous exercise. Estimate the storage requirements for each.

3 Set up a procedure for establishing the geometric data tables for any input set of points defining the polygon facets for the surface of a three-dimensional object.

4 Devise routines for checking the three geometric data tables in Figure 16 to ensure consistency and completeness.

5 Calculate the plane parameters A, B, C, and D for each face of a unit cube centered at the world coordinate origin.

6 Write a program for calculating parameters A, B, C, and D for an input mesh of polygon-surface facets.

7 Write a procedure to determine whether an input coordinate position is in front of a polygon surface or behind it, given the plane parameters A, B, C, and D for the polygon.

8 Write a procedure to determine whether a given point is inside or outside of a cube with a given set of coordinates.

9 If the coordinate reference for a scene is changed from a right-handed system to a left-handed system, what changes could we make in the values of surface plane parameters A, B, C, and D to ensure that the orientation of the plane is correctly described?

10 Given that the first three vertices, V_1, V_2, and V_3, of a pentagon have been used to calculate plane parameters $A = 15$, $B = 21$, $C = 9$, $D = 0$, determine from the final two vertices $V_4 = (2, -1, -1)$ and $V_5 = (1, -2, 2)$ whether the pentagon is planar or non-planar.

11 Develop a procedure for identifying a nonplanar vertex list for a quadrilateral.

12 Extend the algorithm of the previous exercise to identify a nonplanar vertex list that contains more than four coordinate positions.

13 Write a procedure to split a set of four polygon vertex positions into a set of triangles.

14 Split the octagon given by the list of vertices V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8 into a set of triangles and give the vertices that make up each triangle.

15 Devise an algorithm for splitting a set of n polygon vertex positions, with $n > 4$, into a set of triangles.

16 Set up an algorithm for identifying a degenerate polygon vertex list that may contain repeated vertices or collinear vertices.

17 Devise an algorithm for identifying a polygon vertex list that contains intersecting edges.

18 Write a routine to identify concave polygons by calculating cross-products of pairs of edge vectors.

19 Write a routine to split a concave polygon, using the vector method.

20 Write a routine to split a concave polygon, using the rotational method.

21 Devise an algorithm for determining interior regions for any input set of vertices using the nonzero winding-number rule and cross-product calculations to identify the direction for edge crossings.

22 Devise an algorithm for determining interior regions for any input set of vertices using the nonzero winding-number rule and dot-product calculations to identify the direction for edge crossings.

23 What regions of the self-intersecting polyline shown in Figure 12 have a positive winding number? What are the regions that have a negative winding number? What regions have a winding number greater than 1?

24 Write a routine to implement a text-string function that has two parameters: one parameter specifies a world-coordinate position and the other parameter specifies a text string.

25 Write a routine to implement a polymarker function that has two parameters: one parameter is the character that is to be displayed and the other parameter is a list of world-coordinate positions.

26 Modify the example program in Section 16 so that the displayed hexagon is always at the center of the display window, regardless of how the display window may be resized.

27 Write a complete program for displaying a bar chart. Input to the program is to include the data points and the labeling required for the x and y axes. The data points are to be scaled by the program so that the graph is displayed across the full area of a display window.

28 Write a program to display a bar chart in any selected area of a display window.

29 Write a procedure to display a line graph for any input set of data points in any selected area of the screen, with the input data set scaled to fit the selected screen area. Data points are to be displayed as asterisks joined with straight-line segments, and the x and y axes are to be labeled according to input specifications. (Instead of asterisks, small circles or some other symbols could be used to plot the data points.)

30 Using a circle function, write a routine to display a pie chart with appropriate labeling. Input to the routine is to include a data set giving the distribution of the data over some set of intervals, the name of the pie chart, and the names of the intervals. Each section label is to be displayed outside the boundary of the pie chart near the corresponding pie section.

IN MORE DEPTH

1 For this exercise, draw a rough sketch of what a single "snapshot" of your application might look like and write a program to display this snapshot. Choose a background color and default window size. Make sure the snapshot includes at least a few objects. Represent each object as a polygonal approximation to the true object. Use a different shape for each object type. Represent at least one of the objects as a concave polygon. Make each object its own color distinct from the background color. It is a good idea to write a separate function for each object (or each object type) in which you define the representation. Use display lists to create and display each object. Include a display window reshape function to redraw the scene appropriately if the window is if the window is resized.

2 Choose one of the concave polygons you generated in the previous exercise and set up the vertex, edge, and surface facet tables for the shape as described in Section 7. Now split the shape it into a set of convex polygons using the vector method given in the same section. Then split each of the resulting convex polygons into a set of triangles using the method described in Section 7 as well. Finally, set up the vertex, edge, and surface facet tables for the resulting set of triangles. Compare the two table sets and the amount of memory needed to store each.

Attributes of Graphics 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.

1 OpenGL State Variables

Attribute values and other parameter settings are specified with separate functions that define the current OpenGL state. The state parameters in OpenGL include color and other primitive attributes, the current matrix mode, the elements of the model-view matrix, the current position for the frame buffer, and the parameters for the lighting effects in a scene. All OpenGL state parameters have default values, which remain in effect until new values are specified. At any time, we can query the system to determine the current value of a state parameter. In the following sections of this chapter, we discuss only the attribute settings for output primitives. Other state parameters are examined in later chapters.

All graphics primitives in OpenGL are displayed with the attributes in the current state list. Changing one or more of the attribute settings affects only those primitives that are specified after the OpenGL state is changed. Primitives that were defined before the state change retain their attributes. Thus, we can display a green line, change the current color to red, and define another line segment. Both the green line and the red line will then be displayed. Also, some OpenGL state values can be specified within `glBegin/glEnd` pairs, along with the coordinate values, so that parameter settings can vary from one coordinate position to another.

2 Color and Grayscale

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

TABLE 1

The eight RGB color codes for a 3-bit-per-pixel frame buffer

	Stored Color Values in Frame Buffer			
Color Code	**RED**	**GREEN**	**BLUE**	**Displayed Color**
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

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 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 1024×1024, 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 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 color table. 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

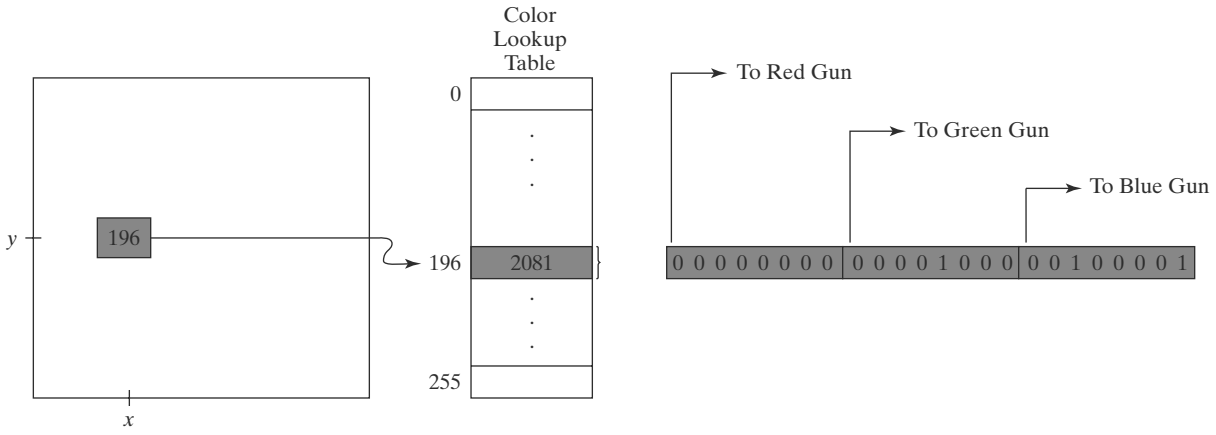


FIGURE 1
A color lookup table with 24 bits per entry that is accessed from a frame buffer with 8 bits per pixel. A value of 196 stored at pixel position (x, y) references the location in this table containing the hexadecimal value 0x0821 (a decimal value of 2081). Each 8-bit segment of this entry controls the intensity level of one of the three electron guns in an RGB monitor.

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.

3 OpenGL Color Functions

In an OpenGL color routines use one function to set the color for the display window, and use another function to specify a color for the straight-line segment. Set the **color display mode** to RGB with the statement

```
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
```

The first constant in the argument list states that we are using a single buffer for the frame buffer, and the second constant puts us into the RGB mode, which is the default color mode. If we wanted to specify colors by an index into a color table, we would replace the OpenGL constant `GLUT_RGB` with `GLUT_INDEX`. When we specify a particular set of color values for primitives, we define the *color state* of OpenGL. The current color is applied to all subsequently defined primitives until we change the color settings. A new color specification affects only the objects we define after the color change.

The OpenGL RGB and RGBA Color Modes

Most color settings for OpenGL primitives are made in the **RGB mode**. In addition to red, green, and blue color coefficients, there is a fourth component called the **alpha coefficient** which is used to control color blending. The four-dimensional color specification is called *RGBA mode*, and we can select it using the OpenGL constant `GLUT_RGBA` when we call `glutInitDisplayMode`. This fourth color parameter can be used to control color blending for overlapping primitives. An important application of color blending is in the simulation of transparency effects. For these calculations, the value of alpha corresponds to a transparency (or, opacity) setting. The alpha value is optional; the only difference between the RGB and RGBA modes is whether we are employing it for color blending.

In the RGB (or RGBA) mode, we select the current color components with the function

```
glColor* (colorComponents);
```

Suffix codes are similar to those for the `glVertex` function. We use a code of either 3 or 4 to specify the RGB or RGBA mode along with the numerical data-type code and an optional vector suffix. The suffix codes for the numerical data types are `b` (byte), `i` (integer), `s` (short), `f` (float), and `d` (double), as well as unsigned numerical values. Floating-point values for the color components are in the range from 0.0 to 1.0, and the default color components for `glColor`, including the alpha value, are (1.0, 1.0, 1.0, 1.0), which sets the RGB color to white and the alpha value to 1.0. If we select the current color using an RGB specification (i.e., we use `glColor3` instead of `glColor4`), the alpha component will be automatically set to 1.0 to indicate that we do not want color blending. As an example, the following statement uses floating-point values in RGB mode to set the current color for primitives to cyan (a combination of the highest intensities for green and blue):

```
glColor3f (0.0, 1.0, 1.0);
```

Using an array specification for the three color components, we could set the color in this example as

```
glColor3fv (colorArray);
```

An OpenGL color selection can be assigned to individual point positions within `glBegin/glEnd` pairs.

Internally, OpenGL represents color information in floating-point format. We can specify colors using integer values, but they will be converted automatically to floating-point. The conversion is based on the data type we choose and the range of values that we can specify in that type. For unsigned types, the minimum value will be converted to a floating-point 0.0, and the maximum value to 1.0; for signed values, the minimum will be converted to −1.0 and the maximum to 1.0. For example, unsigned byte values (suffix code `ub`) have a range of 0 to 255, which corresponds to the color specification system used by some windowing systems. We could specify the cyan color used in our previous example this way:

```
glColor3ub (0, 255, 255);
```

However, if we were to use unsigned 32-bit integers (suffix code `ui`), the range is 0 to 4,294,967,295! At this scale, small changes in color component values are essentially invisible; to make a one-percent change in the intensity of a single color component, for instance, we would need to change that component's value by 42,949,673. For that reason, the most commonly used data types are floating-point and small integer types.

OpenGL Color-Index Mode

Color specifications in OpenGL can also be given in the **color-index mode,** which references values in a color table. Using this mode, we set the current color by specifying an index into a color table as follows:

```
glIndex* (colorIndex);
```

Parameter `colorIndex` is assigned a nonnegative integer value. This index value is then stored in the frame-buffer positions for subsequently specified primitives. We can specify the color index in any of the following data types: unsigned byte,

integer, or floating point. The data type for parameter `colorIndex` is indicated with a suffix code of `ub`, `s`, `i`, `d`, or `f`, and the number of index positions in a color table is always a power of 2, such as 256 or 1024. The number of bits available at each table position depends on the hardware features of the system. As an example of specifying a color in index mode, the following statement sets the current color index to the value 196:

```
glIndexi (196);
```

All primitives defined after this statement will be assigned the color stored at that position in the color table until the current color is changed.

There are no functions provided in the core OpenGL library for loading values into a color-lookup table because table-processing routines are part of a window system. Also, some window systems support multiple color tables and full color, while other systems may have only one color table and limited color choices. However, we do have a GLUT routine that interacts with a window system to set color specifications into a table at a given index position as follows:

```
glutSetColor (index, red, green, blue);
```

Color parameters `red`, `green`, and `blue` are assigned floating-point values in the range from 0.0 to 1.0. This color is then loaded into the table at the position specified by the value of parameter `index`.

Routines for processing three other color tables are provided as extensions to the OpenGL core library. These routines are part of the **Imaging Subset** of OpenGL. Color values stored in these tables can be used to modify pixel values as they are processed through various buffers. Some examples of using these tables are setting camera focusing effects, filtering out certain colors from an image, enhancing certain intensities or making brightness adjustments, converting a grayscale photograph to color, and antialiasing a display. In addition, we can use these tables to change color models; that is, we can change RGB colors to another specification using three other "primary" colors (such as cyan, magenta, and yellow).

A particular color table in the Imaging Subset of OpenGL is activated with the `glEnable` function using one of the table names: `GL_COLOR_TABLE`, `GL_POST_CONVOLUTION_COLOR_TABLE`, or `GL_POST_COLOR_MATRIX_COLOR_TABLE`. We can then use routines in the Imaging Subset to select a particular color table, set color-table values, copy table values, or specify which component of a pixel's color we want to change and how we want to change it.

OpenGL Color Blending

In many applications, it is convenient to be able to combine the colors of overlapping objects or to blend an object with the background. Some examples are simulating a paintbrush effect, forming a composite image of two or more pictures, modeling transparency effects, and antialiasing the objects in a scene. Most graphics packages provide methods for producing various color-mixing effects, and these procedures are called **color-blending functions** or **image-compositing functions.** In OpenGL, the colors of two objects can be blended by first loading one object into the frame buffer, then combining the color of the second object with the frame-buffer color. The current frame-buffer color is referred to as the OpenGL *destination color* and the color of the second object is the OpenGL *source color*. Blending methods can be performed only in RGB or RGBA mode. To apply

color blending in an application, we first need to activate this OpenGL feature using the following function:

```
glEnable (GL_BLEND);
```

We turn off the color-blending routines in OpenGL with

```
glDisable (GL_BLEND);
```

If color blending is not activated, an object's color simply replaces the frame-buffer contents at the object's location.

Colors can be blended in a number of different ways, depending on the effects that we want to achieve, and we generate different color effects by specifying two sets of *blending factors*. One set of blending factors is for the current object in the frame buffer (the "destination object"), and the other set of blending factors is for the incoming ("source") object. The new, blended color that is then loaded into the frame buffer is calculated as

$$(S_r R_s + D_r R_d, S_g G_s + D_g G_d, S_b B_s + D_b B_d, S_a A_s + D_a A_d) \quad (1)$$

where the RGBA source color components are (R_s, G_s, B_s, A_s), the destination color components are (R_d, G_d, B_d, A_d), the source blending factors are (S_r, S_g, S_b, S_a), and the destination blending factors are (D_r, D_g, D_b, D_a). Computed values for the combined color components are clamped to the range from 0.0 to 1.0. That is, any sum greater than 1.0 is set to the value 1.0, and any sum less than 0.0 is set to 0.0.

We select the blending-factor values with the OpenGL function

```
glBlendFunc (sFactor, dFactor);
```

Parameters `sFactor` and `dFactor`, the source and destination factors, are each assigned an OpenGL symbolic constant specifying a predefined set of four blending coefficients. For example, the constant `GL_ZERO` yields the blending factors (0.0, 0.0, 0.0, 0.0) and `GL_ONE` gives us the set (1.0, 1.0, 1.0, 1.0). We could set all four blending factors either to the destination alpha value or to the source alpha value using `GL_DST_ALPHA` or `GL_SRC_ALPHA`. Other OpenGL constants that are available for setting the blending factors include `GL_ONE_MINUS_DST_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`, `GL_DST_COLOR`, and `GL_SRC_COLOR`. These blending factors are often used for simulating transparency, and they are discussed in greater detail in Section 18-4. The default value for parameter `sFactor` is `GL_ONE`, and the default value for parameter `dFactor` is `GL_ZERO`. Hence, the default values for the blending factors result in the incoming color values replacing the current values in the frame buffer.

OpenGL Color Arrays

We can also specify color values for a scene in combination with the coordinate values in a vertex array. This can be done either in RGB mode or in color-index mode. As with vertex arrays, we must first activate the color-array features of OpenGL as follows:

```
glEnableClientState (GL_COLOR_ARRAY);
```

Then, for RGB color mode, we specify the location and format of the color components with

```
glColorPointer (nColorComponents, dataType,
      offset, colorArray);
```

Parameter `nColorComponents` is assigned a value of either 3 or 4, depending on whether we are listing RGB or RGBA color components in the array `colorArray`. An OpenGL symbolic constant such as `GL_INT` or `GL_FLOAT` is assigned to parameter `dataType` to indicate the data type for the color values. For a separate color array, we can assign the value 0 to parameter `offset`. However, if we combine color data with vertex data in the same array, the `offset` value is the number of bytes between each set of color components in the array.

As an example of using color arrays, we can modify a vertex-array to include a color array. The following code fragment sets the color of all vertices on the front face of the cube to blue, and all vertices of the back face are assigned the color red:

```
typedef GLint vertex3 [3], color3 [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} };
color3 hue [8] = { {1, 0, 0}, {1, 0, 0}, {0, 0, 1},
    {0, 0, 1}, {1, 0, 0}, {1, 0, 0}, {0, 0, 1}, {0, 0, 1} };

glEnableClientState (GL_VERTEX_ARRAY);
glEnableClientState (GL_COLOR_ARRAY);

glVertexPointer (3, GL_INT, 0, pt);
glColorPointer (3, GL_INT, 0, hue);
```

We can even stuff both the colors and the vertex coordinates into one **interlaced array.** Each of the pointers would then reference the single interlaced array with an appropriate `offset` value. For example,

```
static GLint hueAndPt [ ] =
    {1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0,
     0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0,
     1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1,
     0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1};

glVertexPointer (3, GL_INT, 6*sizeof(GLint), hueAndPt[3]);
glColorPointer (3, GL_INT, 6*sizeof(GLint), hueAndPt[0]);
```

The first three elements of this array specify an RGB color value, the next three elements specify a set of (x, y, z) vertex coordinates, and this pattern continues to the last color-vertex specification. We set the `offset` parameter to the number of bytes between successive color, or vertex, values, which is `6*sizeof(GLint)` for both. Color values start at the first element of the interlaced array, which is `hueAndPt [0]`, and vertex values start at the fourth element, which is `hueAndPt [3]`.

Because a scene generally contains several objects, each with multiple planar surfaces, OpenGL provides a function in which we can specify all the vertex and color arrays at once, as well as other types of information. If we change the color and vertex values in this example to floating-point, we use this function in the form

```
glInterleavedArrays (GL_C3F_V3F, 0, hueAndPt);
```

The first parameter is an OpenGL constant that indicates three-element floating-point specifications for both color (C) and vertex coordinates (V). The elements of array `hueAndPt` are to be interlaced with the color for each vertex listed before the coordinates. This function also automatically enables both vertex and color arrays.

In color-index mode, we define an array of color indices with

```
glIndexPointer (type, stride, colorIndex);
```

Color indices are listed in the array `colorIndex` and the `type` and `stride` parameters are the same as in `glColorPointer`. No `size` parameter is needed because color-table indices are specified with a single value.

Other OpenGL Color Functions

The following function selects RGB color components for a display window:

```
glClearColor (red, green, blue, alpha);
```

Each color component in the designation (red, green, and blue), as well as the alpha parameter, is assigned a floating-point value in the range from 0.0 to 1.0. The default value for all four parameters is 0.0, which produces the color black. If each color component is set to 1.0, the clear color is white. Shades of gray are obtained with identical values for the color components between 0.0 and 1.0. The fourth parameter, `alpha`, provides an option for blending the previous color with the current color. This can occur only if we activate the blending feature of OpenGL; color blending cannot be performed with values specified in a color table.

There are several *color buffers* in OpenGL that can be used as the current refresh buffer for displaying a scene, and the `glClearColor` function specifies the color for all the color buffers. We then apply the clear color to the color buffers with the command

```
glClear (GL_COLOR_BUFFER_BIT);
```

We can also use the `glClear` function to set initial values for other buffers that are available in OpenGL. These are the *accumulation buffer,* which stores blended-color information, the *depth buffer,* which stores depth values (distances from the viewing position) for objects in a scene, and the *stencil buffer,* which stores information to define the limits of a picture.

In color-index mode, we use the following function (instead of `glClearColor`) to set the display-window color:

```
glClearIndex (index);
```

The window background color is then assigned the color that is stored at position `index` in the color table; and the window is displayed in this color when we issue the `glClear (GL_COLOR_BUFFER_BIT)` function.

Many other color functions are available in the OpenGL library for dealing with a variety of tasks, such as changing color models, setting lighting effects for a scene, specifying camera effects, and rendering the surfaces of an object. We examine other color functions as we explore each of the component processes in a computer-graphics system. For now, we limit our discussion to those functions relating to color specifications for graphics primitives.

4 Point Attributes

Basically, we can set two attributes for points: color and size. In a state system, the displayed color and size of a point is determined by the current values stored in the attribute list. Color components are set with RGB values or an index into a color table. For a raster system, point size is an integer multiple of the pixel size, so that a large point is displayed as a square block of pixels.

5 OpenGL Point-Attribute Functions

The displayed color of a designated point position is controlled by the current color values in the state list. Also, a color is specified with either the `glColor` function or the `glIndex` function.

We set the size for an OpenGL point with

```
glPointSize (size);
```

and the point is then displayed as a square block of pixels. Parameter `size` is assigned a positive floating-point value, which is rounded to an integer (unless the point is to be antialiased). The number of horizontal and vertical pixels in the display of the point is determined by parameter `size`. Thus, a point size of 1.0 displays a single pixel, and a point size of 2.0 displays a 2×2 pixel array. If we activate the antialiasing features of OpenGL, the size of a displayed block of pixels will be modified to smooth the edges. The default value for point size is 1.0.

Attribute functions may be listed inside or outside of a `glBegin/glEnd` pair. For example, the following code segment plots three points in varying colors and sizes. The first is a standard-size red point, the second is a double-size green point, and the third is a triple-size blue point:

```
glColor3f (1.0, 0.0, 0.0);
glBegin (GL_POINTS);
   glVertex2i (50, 100);
   glPointSize (2.0);
   glColor3f (0.0, 1.0, 0.0);
   glVertex2i (75, 150);
   glPointSize (3.0);
   glColor3f (0.0, 0.0, 1.0);
   glVertex2i (100, 200);
glEnd ( );
```

6 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

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.

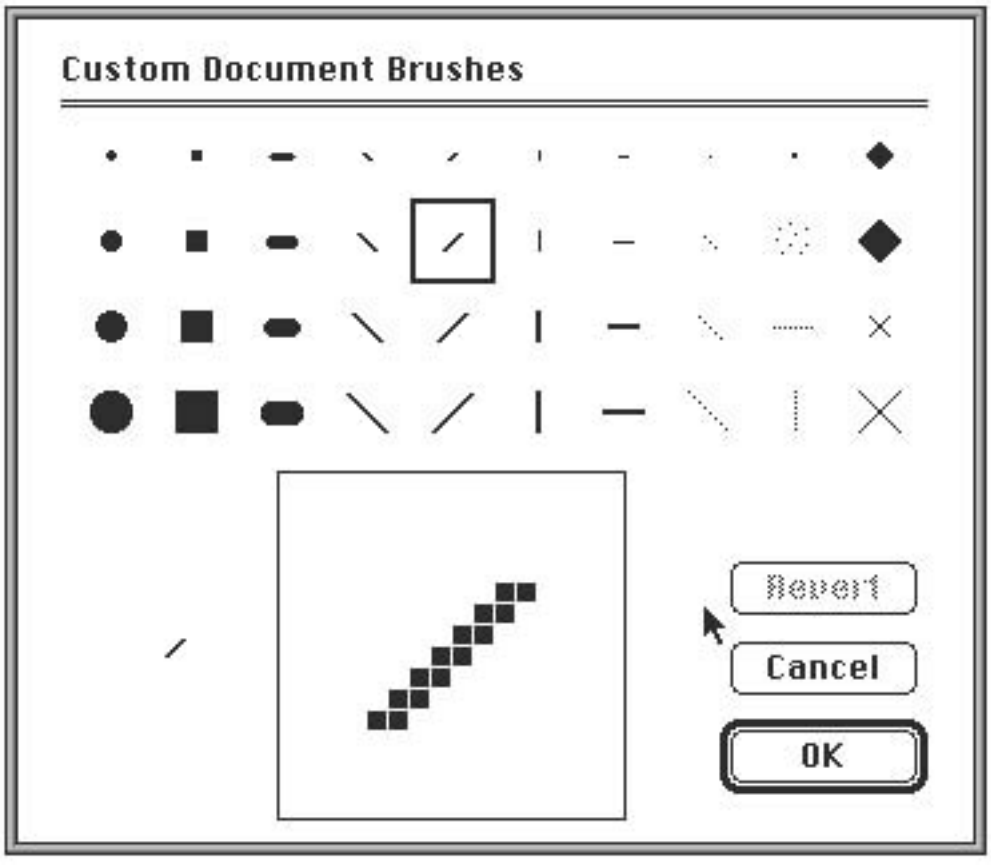


FIGURE 2
Pen and brush shapes for line display.

7 OpenGL Line-Attribute Functions

We can control the appearance of a straight-line segment in OpenGL with three attribute settings: line color, line width, and line style. We have already seen how to make a color selection, and OpenGL provides a function for setting the width of a line and another function for specifying a line style, such as a dashed or dotted line.

OpenGL Line-Width Function

Line width is set in OpenGL with the function

```
glLineWidth (width);
```

We assign a floating-point value to parameter `width`, and this value is rounded to the nearest nonnegative integer. If the input value rounds to 0.0, the line is displayed with a standard width of 1.0, which is the default width. However, when antialiasing is applied to the line, its edges are smoothed to reduce the raster stair-step appearance and fractional widths are possible. Some implementations of the line-width function might support only a limited number of widths, and some might not support widths other than 1.0.

The magnitude of the horizontal and vertical separations of the line endpoints, Δx and Δy, are compared to determine whether to generate a thick line using vertical pixel spans or horizontal pixel spans.

OpenGL Line-Style Function

By default, a straight-line segment is displayed as a solid line. However, we can also display dashed lines, dotted lines, or a line with a combination of dashes and dots, and we can vary the length of the dashes and the spacing between dashes or dots. We set a current display style for lines with the OpenGL function

```
glLineStipple (repeatFactor, pattern);
```

Parameter `pattern` is used to reference a 16-bit integer that describes how the line should be displayed. A 1 bit in the pattern denotes an "on" pixel position, and a 0 bit indicates an "off" pixel position. The pattern is applied to the pixels along the line path starting with the low-order bits in the pattern. The default pattern is 0xFFFF (each bit position has a value of 1), which produces a solid line. Integer parameter `repeatFactor` specifies how many times each bit in the pattern is to be repeated before the next bit in the pattern is applied. The default repeat value is 1.

With a polyline, a specified line-style pattern is not restarted at the beginning of each segment. It is applied continuously across all the segments, starting at the first endpoint of the polyline and ending at the final endpoint for the last segment in the series.

As an example of specifying a line style, suppose that parameter `pattern` is assigned the hexadecimal representation 0x00FF and the repeat factor is 1. This would display a dashed line with eight pixels in each dash and eight pixel positions that are "off" (an eight-pixel space) between two dashes. Also, because low-order bits are applied first, a line begins with an eight-pixel dash starting at the first endpoint. This dash is followed by an eight-pixel space, then another eight-pixel dash, and so forth, until the second endpoint position is reached.

Before a line can be displayed in the current line-style pattern, we must activate the line-style feature of OpenGL. We accomplish this with the following function:

```
glEnable (GL_LINE_STIPPLE);
```

If we forget to include this enable function, solid lines are displayed; that is, the default pattern 0xFFFF is used to display line segments. At any time, we can turn off the line-pattern feature with

```
glDisable (GL_LINE_STIPPLE);
```

This replaces the current line-style pattern with the default pattern (solid lines).

In the following program outline, we illustrate use of the OpenGL line-attribute functions by plotting three line graphs in different styles and widths. Figure 3 shows the data plots that could be generated by this program.

```
/* Define a two-dimensional world-coordinate data type. */
typedef struct { float x, y; } wcPt2D;

wcPt2D dataPts [5];

void linePlot (wcPt2D dataPts [5])
{
   int k;

   glBegin (GL_LINE_STRIP);
      for (k = 0; k < 5; k++)
         glVertex2f (dataPts [k].x, dataPts [k].y);

   glFlush ( );

   glEnd ( );
}

/* Invoke a procedure here to draw coordinate axes.  */

glEnable (GL_LINE_STIPPLE);

/* Input first set of (x, y) data values. */
glLineStipple (1, 0x1C47);    // Plot a dash-dot, standard-width polyline.
linePlot (dataPts);

/* Input second set of (x, y) data values. */
glLineStipple (1, 0x00FF);    // Plot a dashed, double-width polyline.
glLineWidth (2.0);
linePlot (dataPts);

/* Input third set of (x, y) data values. */
glLineStipple (1, 0x0101);    // Plot a dotted, triple-width polyline.
glLineWidth (3.0);
linePlot (dataPts);

glDisable (GL_LINE_STIPPLE);
```

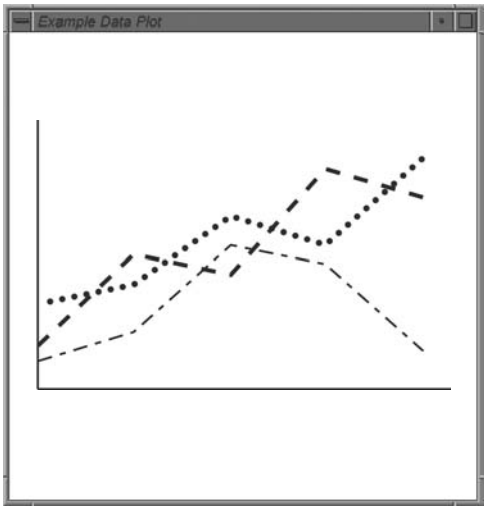


FIGURE 3
Plotting three data sets with three different OpenGL line styles and line widths: single-width dash-dot pattern, double-width dash pattern, and triple-width dot pattern.

Other OpenGL Line Effects

In addition to specifying width, style, and a solid color, we can display lines with color gradations. For example, we can vary the color along the path of a solid line by assigning a different color to each line endpoint as we define the line. In the following code segment, we illustrate this by assigning a blue color to one endpoint of a line and a red color to the other endpoint. The solid line is then displayed as a linear interpolation of the colors at the two endpoints:

```
glShadeModel (GL_SMOOTH);

glBegin (GL_LINES);
   glColor3f (0.0, 0.0, 1.0);
   glVertex2i (50, 50);
   glColor3f (1.0, 0.0, 0.0);
   glVertex2i (250, 250);
glEnd ( );
```

Function `glShadeModel` can also be given the argument `GL_FLAT`. In that case, the line segment would have been displayed in a single color: the color of the second endpoint, (250, 250). That is, we would have generated a red line. Actually, `GL_SMOOTH` is the default, so we would generate a smoothly interpolated color line segment even if we did not include this function in our code.

We can produce other effects by displaying adjacent lines that have different colors and patterns. In addition, we can use the color-blending features of OpenGL by superimposing lines or other objects with varying alpha values. A brush stroke and other painting effects can be simulated with a pixmap and color blending. The pixmap can then be moved interactively to generate line segments. Individual pixels in the pixmap can be assigned different alpha values to display lines as brush or pen strokes.

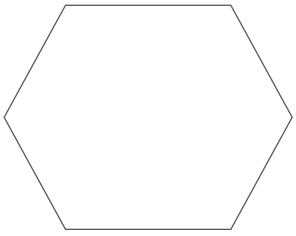
8 Curve Attributes

Parameters for curve attributes are the same as those for straight-line segments. We can display curves with varying colors, widths, dot-dash patterns, and available pen or brush options. for adapting curve-drawing algorithms to accommodate attribute selections are similar to those for line drawing.

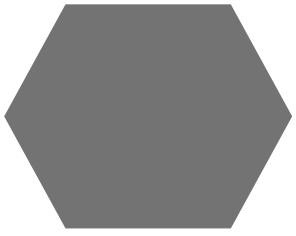
FIGURE 4
Curved lines drawn with a paint program using various shapes and patterns. From left to right, the brush shapes are square, round, diagonal line, dot pattern, and faded airbrush.

Painting and drawing programs allow pictures to be constructed interactively by using a pointing device, such as a stylus and a graphics tablet, to sketch various curve shapes. Some examples of such curve patterns are shown in Figure 4. An additional pattern option that can be provided in a paint package is the display of simulated brush strokes.

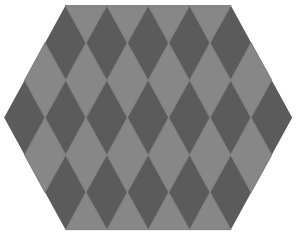
Strictly speaking, OpenGL does not consider curves to be drawing primitives in the same way that it considers points and lines to be primitives. Curves can be drawn in several ways in OpenGL. Perhaps the simplest approach is to approximate the shape of the curve using short line segments. Alternatively, curved segments can be drawn using *splines*. These can be drawn using OpenGL *evaluator* functions, or by using functions from the OpenGL Utility (GLU) library which draw splines.



Hollow
(a)



Solid
(b)



Patterned
(c)

FIGURE 5
Basic polygon fill styles.

9 Fill-Area 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.

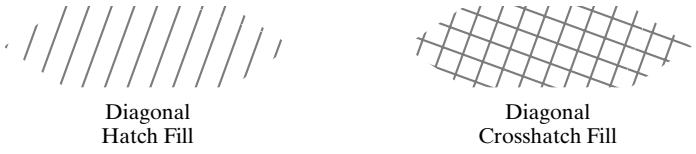


FIGURE 6
Areas filled with hatch patterns.

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

10 OpenGL Fill-Area Attribute Functions

In the OpenGL graphics package, fill-area routines are available for convex polygons only. We generate displays of filled convex polygons in four steps:

1. Define a fill pattern.
2. Invoke the polygon-fill routine.
3. Activate the polygon-fill feature of OpenGL.
4. Describe the polygons to be filled.

A polygon fill pattern is displayed up to and including the polygon edges. Thus, there are no boundary lines around the fill region unless we specifically add them to the display.

In addition to specifying a fill pattern for a polygon interior, there are a number of other options available. One option is to display a hollow polygon, where no interior color or pattern is applied and only the edges are generated. A hollow polygon is equivalent to the display of a closed polyline primitive. Another option is to show the polygon vertices, with no interior fill and no edges. Also, we designate different attributes for the front and back faces of a polygon fill area.

OpenGL Fill-Pattern Function

By default, a convex polygon is displayed as a solid-color region, using the current color setting. To fill the polygon with a pattern in OpenGL, we use a 32×32 bit mask. A value of 1 in the mask indicates that the corresponding pixel is to be set to the current color, and a 0 leaves the value of that frame-buffer position unchanged. The fill pattern is specified in unsigned bytes using the OpenGL data type `GLubyte`, just as we did with the `glBitmap` function. We define a bit pattern with hexadecimal values as, for example,

```
GLubyte fillPattern [ ] = {
    0xff, 0x00, 0xff, 0x00, ... };
```

The bits must be specified starting with the bottom row of the pattern, and continuing up to the topmost row (32) of the pattern. This pattern is replicated

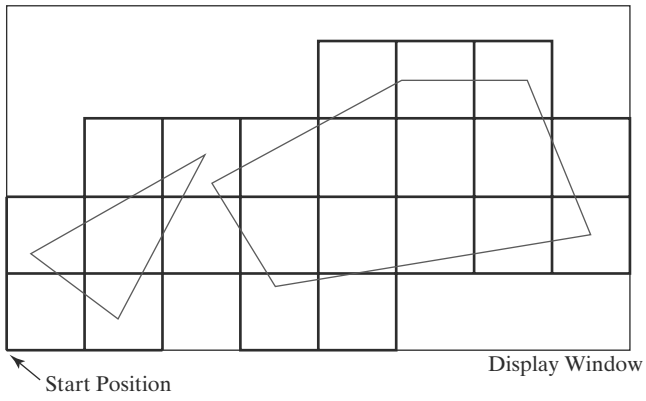


FIGURE 7
Tiling a rectangular fill pattern across a display window to fill two convex polygons.

across the entire area of the display window, starting at the lower-left window corner, and specified polygons are filled where the pattern overlaps those polygons (Figure 7).

Once we have set a mask, we can establish it as the current fill pattern with the function

```
glPolygonStipple (fillPattern);
```

Next, we need to enable the fill routines before we specify the vertices for the polygons that are to be filled with the current pattern. We do this with the statement

```
glEnable (GL_POLYGON_STIPPLE);
```

Similarly, we turn off pattern filling with

```
glDisable (GL_POLYGON_STIPPLE);
```

Figure 8 illustrates how a 3×3 bit pattern, repeated over a 32×32 bit mask, might be applied to fill a parallelogram.

OpenGL Texture and Interpolation Patterns

Another method for filling polygons is to use texture patterns. This can produce fill patterns that simulate the surface appearance of wood, brick, brushed steel, or some other material. Also, we can obtain an interpolation coloring of a polygon interior just as we did with the line primitive. To do this, we assign different colors to polygon vertices. Interpolation fill of a polygon interior is used to produce realistic displays of shaded surfaces under various lighting conditions.

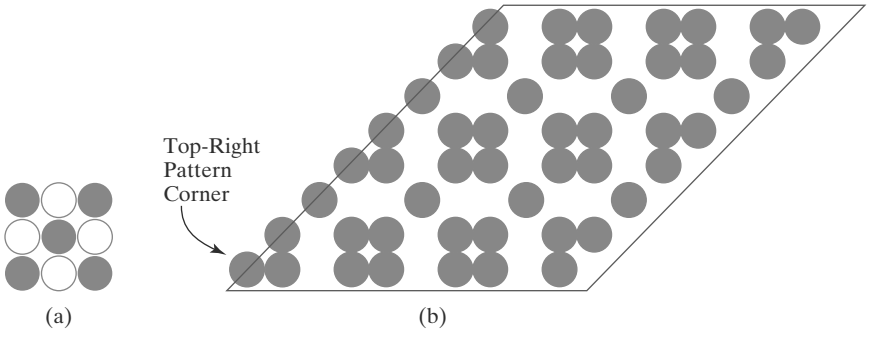


FIGURE 8
A 3×3 bit pattern (a) superimposed on a parallelogram to produce the fill area in (b), where the top-right corner of the pattern coincides with the lower-left corner of the parallelogram.

As an example of an interpolation fill, the following code segment assigns either a blue, red, or green color to each of the three vertices of a triangle. The polygon fill is then a linear interpolation of the colors at the vertices:

```
glShadeModel (GL_SMOOTH);

glBegin (GL_TRIANGLES);
   glColor3f (0.0, 0.0, 1.0);
   glVertex2i (50, 50);
   glColor3f (1.0, 0.0, 0.0);
   glVertex2i (150, 50);
   glColor3f (0.0, 1.0, 0.0);
   glVertex2i (75, 150);
glEnd ( );
```

Of course, if a single color is set for the triangle as a whole, the polygon is filled with that one color; and if we change the argument in the `glShadeModel` function to `GL_FLAT` in this example, the polygon is filled with the last color specified (green). The value `GL_SMOOTH` is the default shading, but we can include that specification to remind us that the polygon is to be filled as an interpolation of the vertex colors.

OpenGL Wire-Frame Methods

We can also choose to show only polygon edges. This produces a wire-frame or hollow display of the polygon; or we could display a polygon by plotting a set of points only at the vertex positions. These options are selected with the function

```
glPolygonMode (face, displayMode);
```

We use parameter `face` to designate which face of the polygon that we want to show as edges only or vertices only. This parameter is then assigned either `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`. Then, if we want only the polygon edges displayed for our selection, we assign the constant `GL_LINE` to parameter `displayMode`. To plot only the polygon vertex points, we assign the constant `GL_POINT` to parameter `displayMode`. A third option is `GL_FILL`; but this is the default display mode, so we usually invoke only `glPolygonMode` when we want to set attributes for the polygon edges or vertices.

Another option is to display a polygon with both an interior fill and a different color or pattern for its edges (or for its vertices). This is accomplished by specifying the polygon twice: once with parameter `displayMode` set to `GL_FILL` and then again with `displayMode` set to `GL_LINE` (or `GL_POINT`). For example, the following code section fills a polygon interior with a green color, and then the edges are assigned a red color:

```
glColor3f (0.0, 1.0, 0.0);
/* Invoke polygon-generating routine. */

glColor3f (1.0, 0.0, 0.0);
glPolygonMode (GL_FRONT, GL_LINE);
/* Invoke polygon-generating routine again. */
```

For a three-dimensional polygon (one that does not have all vertices in the xy plane), this method for displaying the edges of a filled polygon may produce gaps along the edges. This effect, sometimes referred to as **stitching,** is caused by

differences between calculations in the scan-line fill algorithm and calculations in the edge line-drawing algorithm. As the interior of a three-dimensional polygon is filled, the depth value (distance from the xy plane) is calculated for each (x, y) position. However, this depth value at an edge of the polygon is often not exactly the same as the depth value calculated by the line-drawing algorithm for the same (x, y) position. Therefore, when visibility tests are made, the interior fill color could be used instead of an edge color to display some points along the boundary of a polygon.

One way to eliminate the gaps along displayed edges of a three-dimensional polygon is to shift the depth values calculated by the fill routine so that they do not overlap with the edge depth values for that polygon. We do this with the following two OpenGL functions:

```
glEnable (GL_POLYGON_OFFSET_FILL);
glPolygonOffset (factor1, factor2);
```

The first function activates the offset routine for scan-line filling, and the second function is used to set a couple of floating-point values `factor1` and `factor2` that are used to calculate the amount of depth offset. The calculation for this depth offset is

$$\text{depthOffset} = \text{factor1} \cdot \text{maxSlope} + \text{factor2} \cdot \text{const} \tag{2}$$

where maxSlope is the maximum slope of the polygon and const is an implementation constant. For a polygon in the xy plane, the slope is 0. Otherwise, the maximum slope is calculated as the change in depth of the polygon divided by either the change in x or the change in y. A typical value for the two factors is either 0.75 or 1.0, although some experimentation with the factor values is often necessary to produce good results. As an example of assigning values to offset factors, we can modify the previous code segment as follows:

```
glColor3f (0.0, 1.0, 0.0);
glEnable (GL_POLYGON_OFFSET_FILL);
glPolygonOffset (1.0, 1.0);
/* Invoke polygon-generating routine. */
glDisable (GL_POLYGON_OFFSET_FILL);

glColor3f (1.0, 0.0, 0.0);
glPolygonMode (GL_FRONT, GL_LINE);
/* Invoke polygon-generating routine again. */
```

Now the interior fill of the polygon is pushed a little farther away in depth, so that it does not interfere with the depth values of its edges. It is also possible to implement this method by applying the offset to the line-drawing algorithm, by changing the argument of the `glEnable` function to `GL_POLYGON_OFFSET_LINE`. In this case, we want to use negative factors to bring the edge depth values closer. Also, if we just wanted to display different color points at the vertex positions, instead of highlighted edges, the argument in the `glEnable` function would be `GL_POLYGON_OFFSET_POINT`.

Another method for eliminating the stitching effect along polygon edges is to use the OpenGL stencil buffer to limit the polygon interior filling so that it does not overlap the edges. However, this approach is more complicated and generally slower, so the polygon depth-offset method is preferred.

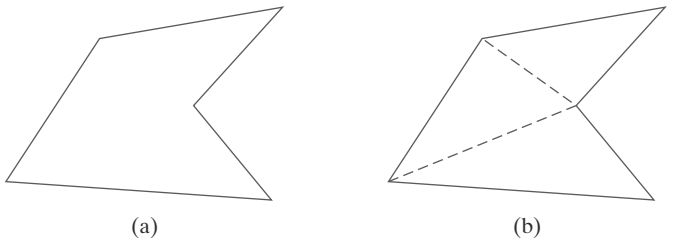


FIGURE 9
Dividing a concave polygon (a) into a set of triangles (b) produces triangle edges (dashed) that are interior to the original polygon.

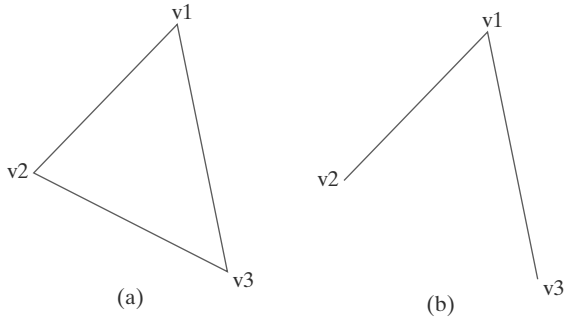


FIGURE 10
The triangle in (a) can be displayed as in (b) by setting the edge flag for vertex v2 to the value `GL_FALSE`, assuming that the vertices are specified in a counterclockwise order.

To display a concave polygon using OpenGL routines, we must first split it into a set of convex polygons. We typically divide a concave polygon into a set of triangles. Then we could display the concave polygon as a fill region by filling the triangles. Similarly, if we want to show only the polygon vertices, we plot the triangle vertices. To display the original concave polygon in a wire-frame form, however, we cannot just set the display mode to `GL_LINE` because that would show all the triangle edges that are interior to the original concave polygon (Figure 9).

Fortunately, OpenGL provides a mechanism that allows us to eliminate selected edges from a wire-frame display. Each polygon vertex is stored with a one-bit flag that indicates whether or not that vertex is connected to the next vertex by a boundary edge. So all we need do is set that bit flag to "off" and the edge following that vertex will not be displayed. We set this flag for an edge with the following function:

```
glEdgeFlag (flag);
```

To indicate that a vertex does not precede a boundary edge, we assign the OpenGL constant `GL_FALSE` to parameter `flag`. This applies to all subsequently specified vertices until the next call to `glEdgeFlag` is made. The OpenGL constant `GL_TRUE` turns the edge flag on again, which is the default. Function `glEdgeFlag` can be placed between `glBegin/glEnd` pairs. As an illustration of the use of an edge flag, the following code displays only two edges of the defined triangle (Figure 10):

```
glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);

glBegin (GL_POLYGON);
   glVertex3fv (v1);
   glEdgeFlag (GL_FALSE);
   glVertex3fv (v2);
   glEdgeFlag (GL_TRUE);
   glVertex3fv (v3);
glEnd ( );
```

Polygon edge flags can also be specified in an array that could be combined or associated with a vertex array (see Section 3). The statements for creating an array of edge flags are

```
glEnableClientState (GL_EDGE_FLAG_ARRAY);

glEdgeFlagPointer (offset, edgeFlagArray);
```

Parameter `offset` indicates the number of bytes between the values for the edge flags in the array `edgeFlagArray`. The default value for parameter `offset` is 0.

OpenGL Front-Face Function

Although, by default, the ordering of polygon vertices controls the identification of front and back faces, we can label selected surfaces in a scene independently as front or back with the function

```
glFrontFace (vertexOrder);
```

If we set parameter `vertexOrder` to the OpenGL constant `GL_CW`, then a subsequently defined polygon with a clockwise ordering for its vertices is considered to be front-facing. This OpenGL feature can be used to swap faces of a polygon for which we have specified vertices in a clockwise order. The constant `GL_CCW` labels a counterclockwise ordering of polygon vertices as front-facing, which is the default ordering.

11 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, dotted, 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 centimeters (or 0.013837 inch, which is approximately $\frac{1}{72}$ 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

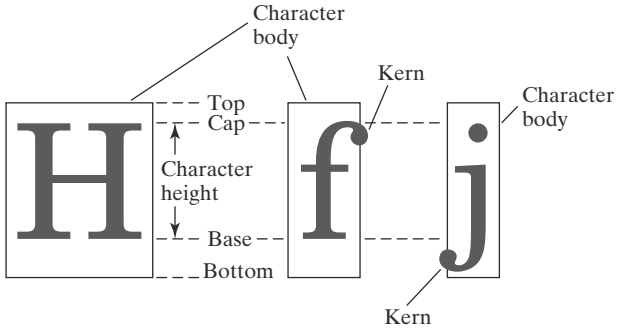


FIGURE 11
Examples of character bodies.

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 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

Height 1
Height 2
Height 3

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

width 0.5
width 1.0
width 2.0

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

Spacing 0.0
Spacing 0.5
Spacing 1.0

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

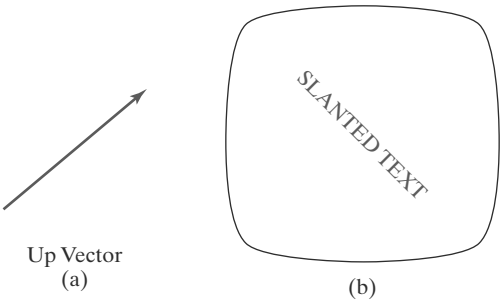


FIGURE 15
Direction of the up vector (a) controls the orientation of displayed text (b).

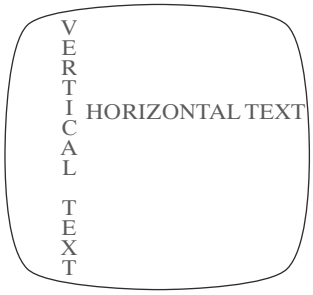


FIGURE 16
Text-path attributes can be set to produce horizontal or vertical arrangements of character strings.

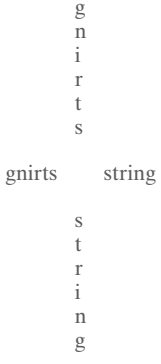


FIGURE 17
A text string displayed with the four text-path options: left, right, up, and down.

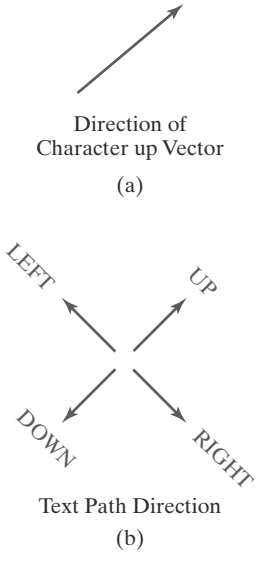


FIGURE 18
An up-vector specification (a) and associated directions for the text path (b).

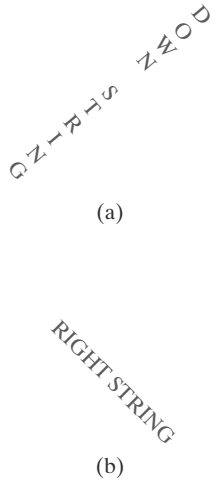


FIGURE 19
The 45° up vector in Figure 18 produces the display (a) for a *down* path and the display (b) for a *right* path.

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*.

12 OpenGL Character-Attribute Functions

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.

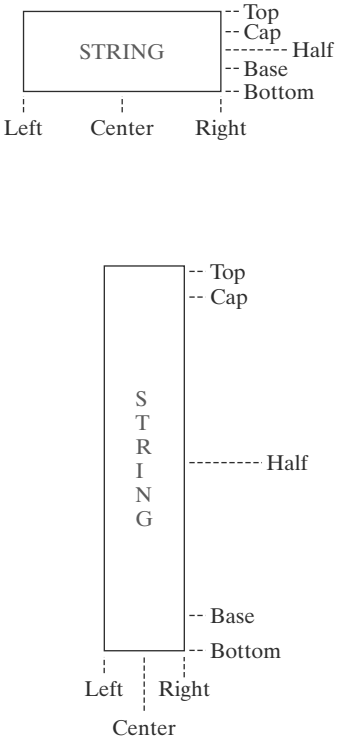


FIGURE 20
Character alignments for horizontal and vertical strings.

13 OpenGL Antialiasing Functions

Line segments and other graphics primitives generated by raster algorithms have a jagged, or stair-step, appearance because the sampling process digitizes coordinate points on an object to discrete integer pixel positions. This distortion of information due to low-frequency sampling (undersampling) is called **aliasing**. We can improve the appearance of displayed raster lines by applying **antialiasing** methods that compensate for the undersampling process.

OpenGL provides antialiasing support for three types of primitives. We activate the antialiasing routines with the function

```
glEnable (primitiveType);
```

where parameter `primitiveType` is assigned one of the symbolic constant values `GL_POINT_SMOOTH`, `GL_LINE_SMOOTH`, or `GL_POLYGON_SMOOTH`. Assuming that we are specifying color values using the RGBA mode, we also need to activate the OpenGL color-blending operations as follows:

```
glEnable (GL_BLEND);
```

Next, we apply the color-blending method described in Section 3 using the function

```
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

The smoothing operations are more effective if we use large alpha values in the color specifications for the objects.

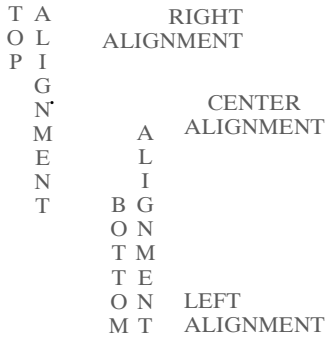


FIGURE 21
Character-string alignments.

Antialiasing can also be applied when we use color tables. However, in this color mode, we must create a `color ramp`, which is a table of color graduations from the background color to the object color. This color ramp is then used to antialias object boundaries.

14 OpenGL Query Functions

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 `glGetIntegerv` 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 line widths 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.

15 OpenGL Attribute Groups

Attributes and other OpenGL state parameters are arranged in **attribute groups.** Each group contains a set of related state parameters. For instance, the **point-attribute group** contains the size and point-smooth (antialiasing) parameters, and the **line-attribute group** contains the width, stipple status, stipple pattern, stipple repeat counter, and line-smooth status. Similarly, the **polygon-attribute group** contains eleven polygon parameters, such as fill pattern, front-face flag, and polygon-smooth status. Because color is an attribute for all primitives, it has its own attribute group; and some parameters are included in more than one group.

About twenty different attribute groups are available in OpenGL, and all parameters in one or more groups can be saved or reset with a single function. We save all parameters within a specified group using the following command:

```
glPushAttrib (attrGroup);
```

Parameter `attrGroup` is assigned an OpenGL symbolic constant that identifies an attribute group, such as `GL_POINT_BIT`, `GL_LINE_BIT`, or `GL_POLYGON_BIT`. To save color parameters, we use the symbolic constant `GL_CURRENT_BIT`, and we can save all state parameters in all attribute groups with the constant `GL_ALL_ATTRIB_BITS`. The `glPushAttrib` function places all parameters within the specified group onto an **attribute stack.**

We can also save parameters within two or more groups by combining their symbolic constants with a logical `OR` operation. The following statement places all parameters for points, lines, and polygons on the attribute stack:

```
glPushAttrib (GL_POINT_BIT | GL_LINE_BIT | GL_POLYGON_BIT);
```

Once we have saved a group of state parameters, we can reinstate all values on the attribute stack with this function:

```
glPopAttrib ( );
```

No arguments are used in the `glPopAttrib` function because it resets the current state of OpenGL using all values on the stack.

These commands for saving and resetting state parameters use a *server attribute stack*. There is also a *client attribute stack* available in OpenGL for saving and resetting client state parameters. The functions for accessing this stack are `glPushClientAttrib` and `glPopClientAttrib`. Only two client attribute groups are available: one for pixel-storage modes and the other for vertex arrays. Pixel-storage parameters include information such as byte alignment and the type of arrays used to store subimages of a display. Vertex-array parameters give information about the current vertex-array state, such as the enable/disable state of various arrays.

16 Summary

Attributes control the display characteristics of graphics primitives. In many graphics systems, attribute values are stored as state variables and primitives are generated using the current attribute values. When we change the value of a state variable, it affects only those primitives defined after the change.

A common attribute for all primitives is color, which is most often specified in terms of RGB (or RGBA) components. The red, green, and blue color values are

stored in the frame buffer, and they are used to control the intensity of the three electron guns in an RGB monitor. Color selections can also be made using color-lookup tables. In this case, a color in the frame buffer is indicated as a table index, and the table location at that index stores a particular set of RGB color values. Color tables are useful in data-visualization and image-processing applications, and they can also be used to provide a wide range of colors without requiring a large frame buffer. Often, computer-graphics packages provide options for using either color tables or storing color values directly in the frame buffer.

The basic point attributes are color and size. Line attributes are color, width, and style. Specifications for line width are given in terms of multiples of a standard, one-pixel-wide line. The line-style attributes include solid, dashed, and dotted lines, as well as various brush or pen styles. These attributes can be applied to both straight lines and curves.

Fill-area attributes include a solid-color fill, a fill pattern, and a hollow display that shows only the area boundaries. Various pattern fills can be specified in color arrays, which are then mapped to the interior of the region. Scan-line methods are commonly used to fill polygons, circles, and ellipses.

Areas can also be filled using color blending. This type of fill has applications in antialiasing and in painting packages. Soft-fill procedures provide a new fill color for a region that has the same variations as the previous fill color.

Characters can be displayed in different styles (fonts), colors, sizes, spacing, and orientations. To set the orientation of a character string, we can specify a direction for the character up vector and a direction for the text path. In addition, we can set the alignment of a text string in relation to the start coordinate position. Individual characters, called marker symbols, can be used for applications such as plotting data graphs. Marker symbols can be displayed in various sizes and colors using standard characters or special symbols.

Because scan conversion is a digitizing process on raster systems, displayed primitives have a jagged appearance. This is due to the undersampling of information, which rounds coordinate values to pixel positions. We can improve the appearance of raster primitives by applying antialiasing procedures that adjust pixel intensities.

In OpenGL, attribute values for the primitives are maintained as state variables. An attribute setting remains in effect for all subsequently defined primitives until that attribute value is changed. Changing an attribute value does not affect previously displayed primitives. We can specify colors in OpenGL using either the RGB (or RGBA) color mode or the color-index mode, which uses color-table indices to select colors. Also, we can blend color values using the alpha color component, and we can specify values in color arrays that are to be used in conjunction with vertex arrays. In addition to color, OpenGL provides functions for selecting point size, line width, line style, and convex-polygon fill style, as well as providing functions for the display of polygon fill areas as either a set of edges or a set of vertex points. We can also eliminate selected polygon edges from a display, and we can reverse the specification of front and back faces. We can generate text strings in OpenGL using bitmaps or routines that are available in GLUT. Attributes that can be set for the display of GLUT characters include color, font, size, spacing, line width, and line type. The OpenGL library also provides functions to antialias the display of output primitives. We can use query functions to obtain the current value for state variables, and we can also obtain all values within an OpenGL attribute group using a single function.

Table 2 summarizes the OpenGL attribute functions discussed in this chapter. In addition, the table lists some attribute-related functions.

TABLE 2

Summary of OpenGL Attribute Functions

Function	Description
`glutInitDisplayMode`	Selects the color mode, which can be either `GLUT_RGB` or `GLUT_INDEX`.
`glColor*`	Specifies an RGB or RGBA color.
`glIndex*`	Specifies a color using a color-table index.
`glutSetColor (index, r, g, b);`	Loads a color into a color-table position.
`glEnable (GL_BLEND);`	Activates color blending.
`glBlendFunc (sFact, dFact);`	Specifies factors for color blending.
`glEnableClientState (GL_COLOR_ARRAY);`	Activates color-array features of OpenGL.
`glColorPointer (size, type, stride, array);`	Specifies an RGB color array.
`glIndexPointer (type, stride, array);`	Specifies a color array using color-index mode.
`glPointSize (size)`	Specifies a point size.
`glLineWidth (width);`	Specifies a line width.
`glEnable (GL_LINE_STIPPLE);`	Activates line style.
`glEnable (GL_POLYGON_STIPPLE);`	Activates fill style.
`glLineStipple (repeat, pattern);`	Specifies a line-style pattern.
`glPolygonStipple (pattern);`	Specifies a fill-style pattern.
`glPolygonMode`	Displays front or back face as either a set of edges or a set of vertices.
`glEdgeFlag`	Sets fill-polygon edge flag to `GL_TRUE` or `GL_FALSE` to determine display status for an edge.
`glFrontFace`	Specifies front-face vertex order as either `GL_CCW` or `GL_CW`.
`glEnable`	Activates antialiasing with `GL_POINT_SMOOTH`, `GL_LINE_SMOOTH`, or `GL_POLYGON_SMOOTH`. (Also need to activate color blending.)
`glGet**`	Queries OpenGL to retrieve an attribute value of a specific data type, identified by the symbolic name of the attribute, placing the result in an array parameter.
`glPushAttrib`	Saves all state parameters within a specified attribute group.
`glPopAttrib ( );`	Reinstates all state parameter values that were last saved.

REFERENCES

Soft-fill techniques are given in Fishkin and Barsky (1984). Antialiasing techniques are discussed in Pitteway and Watinson (1980), Crow (1981), Turkowski (1982), Fujimoto and Iwata (1983), Korein and Badler (1983), Kirk and Arvo (1991), and Wu (1991). Grayscale applications are explored in Crow (1978). Other discussions of attributes and state parameters are available in Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994), and Paeth (1995).

Programming examples using OpenGL attribute functions are given in Woo, et al. (1999). A complete listing of OpenGL attribute functions is available in Shreiner (2000), and GLUT character attributes are discussed in Kilgard (1996).

EXERCISES

1 Use the `glutSetColor` function to set up a color table for an input set of color values.

2 Using vertex and color arrays, set up the description for a scene containing at least six two-dimensional objects.

3 Write a program to display the two-dimensional scene description in the previous exercise.

4 Using vertex and color arrays, set up the description for a scene containing at least four three-dimensional objects.

5 Write a program to display a two-dimensional, grayscale "target" scene, where the target is made up of a small, filled central circle and two concentric rings around the circle spaced as far apart as their thickness, which should be equal to the radius of the inner circle. The circle and rings are to be described as point patterns on a white background. The rings/circle should "fade in" from their outer edges so that the inner portion of the shape is darker than the outer portion. This can be achieved by varying the sizes and inter-point spacing of the points that make up the rings/circle. For example, the edges of a ring can be modeled with small, widely spaced, light-gray points, while the inner portion can be modeled with larger, more closely spaced, dark-gray points.

6 Modify the program in the previous exercise to display the circle and rings in various shades of red instead of gray.

7 Modify the code segments in Section 7 for displaying data line plots, so that the line-width parameter is passed to procedure `linePlot`.

8 Modify the code segments in Section 7 for displaying data line plots, so that the line-style parameter is passed to procedure `linePlot`.

9 Complete the program in Section 7 for displaying line plots using input values from a data file.

10 Complete the program in Section 7 for displaying line plots using input values from a data file. In addition, the program should provide labeling for the axes and the coordinates for the display area on the screen. The data sets are to be scaled to fit the coordinate range of the display window, and each plotted line is to be displayed in a different line style, width, and color.

11 Write a routine to display a bar graph in any specified screen area. Input is to include the data set, labeling for the coordinate axes, and the coordinates for the screen area. The data set is to be scaled to fit the designated screen area, and the bars are to be displayed in designated colors or patterns.

12 Write a procedure to display two data sets defined over the same x-coordinate range, with the data values scaled to fit a specified region of the display screen. The bars for one of the data sets are to be displaced horizontally to produce an overlapping bar pattern for easy comparison of the two sets of data. Use a different color or a different fill pattern for the two sets of bars.

13 Devise an algorithm for implementing a color lookup table.

14 Suppose you have a system with an 10 inch by 14 inch video screen that can display 120 pixels per inch. If a color lookup table with 256 positions is used with this system, what is the smallest possible size (in bytes) for the frame buffer?

15 Consider an RGB raster system that has a 1024-by-786 frame buffer with 16 bits per pixel and a color lookup table with 24 bits per pixel. (a) How many distinct gray levels can be displayed with this system? (b) How many distinct colors (including gray levels) can be displayed? (c) How many colors can be displayed at any one time? (d) What is the total memory size? (e) Explain two methods for reducing memory size while maintaining the same color capabilities.

16 Write a program to output a grayscale scatter plot of two data sets defined over the same x- and y-coordinate ranges. Inputs to the program are the two sets of data. The data sets are to be scaled to fit within a defined coordinate range for a display window. Each data set is to be plotted using points in a distinct shade of gray.

17 Modify the program in the previous exercise to plot the two data sets in different colors instead of shades of gray. Also, add a legend somewhere on the plot bordered by a solid black line. The legend should display the name of each data set (given as input) in the color associated with that data set.

IN MORE DEPTH

1 Develop an application and experiment with different methods of shading the simple shapes. Using the OpenGL functions for hollow, solid color, and pattern fills of polygons, assign a fill type to each shape in the scene and apply these fills. At least one of the objects should have a hollow fill, one should be filled with a solid color, and one should be filled with a bit pattern that you specify yourself. Don't worry if the fill patterns do not necessarily make sense for the objects in the scene. The goal here is to experiment with the different fill attributes available in OpenGL. In addition, experiment with different line drawing attributes to draw the boundaries of the shapes in your snapshot. Employ the use of solid boundary lines as well as dotted ones, each of varying thickness. Add the ability to turn anti-aliasing on and off, and examine the visual differences between the two cases.

2 Set up a small color table that serves as a color palette for your scene and draw the scene as it exists after the previous exercise using this color table instead of the standard OpenGL color functions as before. Once you produce your color table, compare its memory requirements and rendering capabilities with the standard color assignment method on your system. How many different colors can be displayed simultaneously by using the table? How much memory is saved when representing the frame buffer by using the color table instead of directly assigning colors to pixels? How small can you make the color table without noticing a significant difference in the rendering of the scene? Discuss the advantages and disadvantages to using the color table versus using direct color assignment.

This page intentionally left blank

Implementation Algorithms for Graphics Primitives and Attributes

In this chapter, we discuss the device-level algorithms for implementing OpenGL primitives. Exploring the implementation algorithms for a graphics library will give us valuable insight into the capabilities of these packages. It will also provide us with an understanding of how the functions work, perhaps how they could be improved, and how we might implement graphics routines ourselves for some special application. Research in computer graphics is continually discovering new and improved implementation techniques to provide us with methods for special applications, such as Internet graphics, and for developing faster and more realistic graphics displays in general.

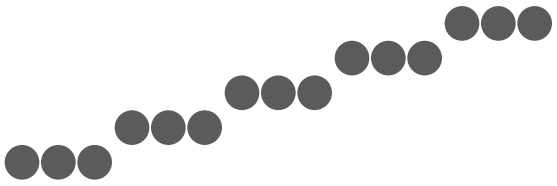


FIGURE 1
Stair-step effect (jaggies) produced when a line is generated as a series of pixel positions.

1 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 (see Section 15 for details).

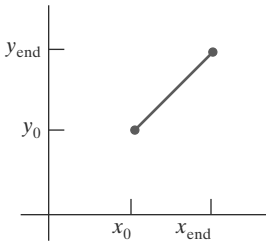


FIGURE 2
Line path between endpoint positions (x_0, y_0) and (x_{end}, y_{end}).

Line Equations

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

$$y = m \cdot x + b \tag{1}$$

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:

$$m = \frac{y_{end} - y_0}{x_{end} - x_0} \tag{2}$$

$$b = y_0 - m \cdot x_0 \tag{3}$$

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

For any given x interval δx along a line, we can compute the corresponding y interval, δy, from Equation 2 as

$$\delta y = m \cdot \delta x \tag{4}$$

Similarly, we can obtain the x interval δx corresponding to a specified δy as

$$\delta x = \frac{\delta y}{m} \tag{5}$$

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 $|m| < 1$, δx can be set proportional to a small horizontal deflection voltage, and the corresponding vertical deflection is then set proportional to δy as calculated from Equation 4. For lines

whose slopes have magnitudes $|m| > 1$, δy can be set proportional to a small vertical deflection voltage with the corresponding horizontal deflection voltage set proportional to δx, 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.

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.

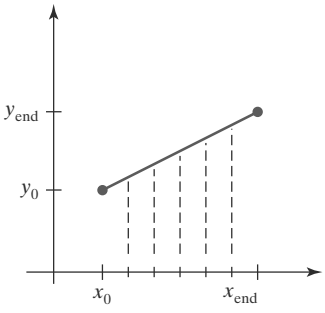


FIGURE 3
Straight-line segment with five sampling positions along the x axis between x_0 and x_{end}.

DDA Algorithm

The *digital differential analyzer* (*DDA*) is a scan-conversion line algorithm based on calculating either δy or δx, 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

$$y_{k+1} = y_k + m \tag{6}$$

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

$$x_{k+1} = x_k + \frac{1}{m} \tag{7}$$

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

$$y_{k+1} = y_k - m \tag{8}$$

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

$$x_{k+1} = x_k - \frac{1}{m} \tag{9}$$

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 $\frac{1}{m}$.

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

inline int round (const float a)  { return int (a + 0.5); }

void lineDDA (int x0, int y0, int xEnd, int yEnd)
{
   int dx = xEnd - x0,  dy = yEnd - y0,  steps,  k;
   float xIncrement, yIncrement, x = x0, y = y0;

   if (fabs (dx) > fabs (dy))
      steps = fabs (dx);
   else
      steps = fabs (dy);
   xIncrement = float (dx) / float (steps);
   yIncrement = float (dy) / float (steps);

   setPixel (round (x), round (y));
   for (k = 0; k < steps; k++) {
      x += xIncrement;
      y += yIncrement;
      setPixel (round (x), round (y));
   }
}
```

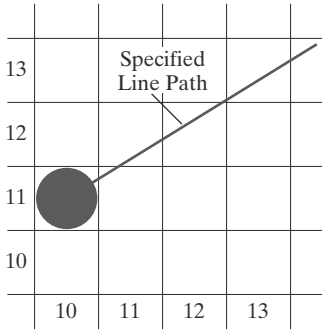


FIGURE 4
A section of a display screen where a straight-line segment is to be plotted, starting from the pixel at column 10 on scan line 11.

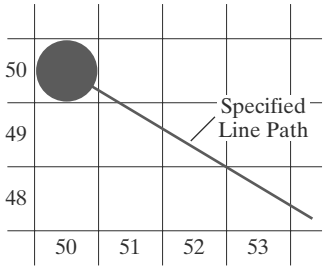


FIGURE 5
A section of a display screen where a negative slope line segment is to be plotted, starting from the pixel at column 50 on scan line 50.

The DDA algorithm is a faster method for calculating pixel positions than one that directly implements Equation 1. It eliminates the multiplication in Equation 1 by using raster characteristics, so that appropriate increments are applied in the x or y directions to step from one pixel position to another along the line path. The accumulation of round-off error in successive additions of the floating-point increment, however, can cause the calculated pixel positions to drift away from the true line path for long line segments. Furthermore, the rounding operations and floating-point arithmetic in this procedure are still time-consuming. We can improve the performance of the DDA algorithm by separating the increments m and $\frac{1}{m}$ into integer and fractional parts so that all calculations are reduced to integer operations. A method for calculating $\frac{1}{m}$ increments in integer steps is discussed in Section 10. In the next section, we consider a more general scan-line approach that can be applied to both lines and curves.

Bresenham's Line Algorithm

In this section, we introduce an accurate and efficient raster line-generating 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

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_0, y_0) 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 kth 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 $(x_k + 1, y_k + 1)$.

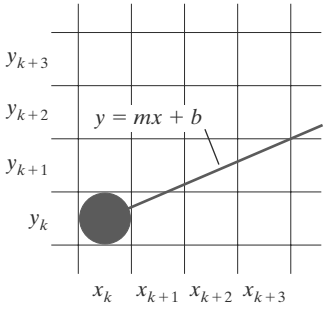


FIGURE 6
A section of the screen showing a pixel in column x_k on scan line y_k that is to be plotted along the path of a line segment with slope $0 < m < 1$.

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

$$y = m(x_k + 1) + b \tag{10}$$

Then

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

and

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

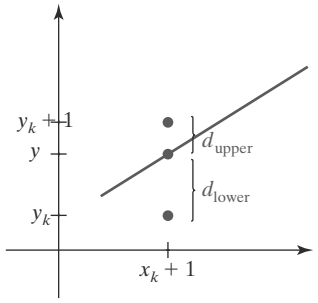


FIGURE 7
Vertical distances between pixel positions and the line y coordinate at sampling position $x_k + 1$.

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:

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

A decision parameter p_k for the kth 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 Δy and Δx are the vertical and horizontal separations of the endpoint positions, and defining the decision parameter as

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

The sign of p_k is the same as the sign of $d_{\text{lower}} - d_{\text{upper}}$, 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, $d_{\text{lower}} < d_{\text{upper}}$), then decision parameter p_k is negative. In that case, we plot the lower pixel; otherwise, we plot the upper pixel.

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

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

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:

$$p_0 = 2\Delta y - \Delta x \tag{16}$$

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 Δx times.

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

1. Input the two line endpoints and store the left endpoint in (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 Δx, Δy, $2\Delta y$, 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

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

The initial decision parameter has the value

$$p_0 = 2\Delta y - \Delta x$$
$$= 6$$

and the increments for calculating successive decision parameters are

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

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})
0	6	(21, 11)
1	2	(22, 12)
2	−2	(23, 12)
3	14	(24, 13)
4	10	(25, 14)
5	6	(26, 15)
6	2	(27, 16)
7	−2	(28, 16)
8	14	(29, 17)
9	10	(30, 18)

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

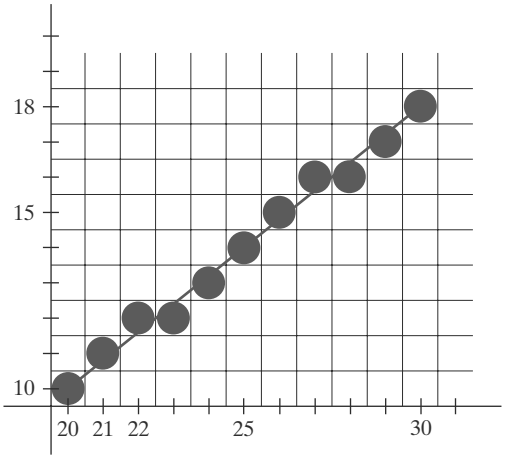


FIGURE 8
Pixel positions along the line path between endpoints (20, 10) and (30, 18), plotted with Bresenham's line algorithm.

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 lineBres (int x0, int y0, int xEnd, int yEnd)
{
   int dx = fabs (xEnd - x0),  dy = fabs(yEnd - 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 > xEnd) {
      x = xEnd;
      y = yEnd;
      xEnd = x0;
   }
```

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

    while (x < xEnd) {
       x++;
       if (p < 0)
          p += twoDy;
       else {
          y++;
          p += twoDyMinusDx;
       }
       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 ($d_{\text{lower}} = d_{\text{upper}}$). 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

$$\Delta x_p = \frac{\Delta x + n_p - 1}{n_p} \tag{17}$$

where Δx is the width of the line, and the value for partition width Δx_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 kth partition as

$$x_k = x_0 + k\Delta x_p \tag{18}$$

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 Δy_p in the y direction over each partition is calculated from the line slope m and partition width Δx_p:

$$\Delta y_p = m\Delta x_p \tag{19}$$

At the kth partition, the starting y coordinate is then

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

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

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

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

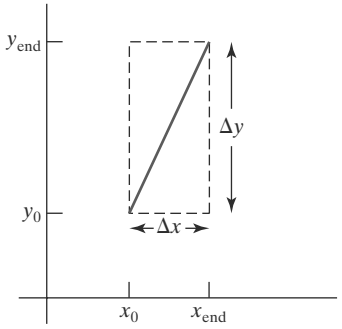


FIGURE 9
Bounding box for a line with endpoint separations Δx and Δy.

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

$$d = A\,x + B\,y + C \tag{22}$$

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

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

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.

3 Setting Frame-Buffer Values

A final stage in the implementation procedures for line segments and other objects is to set the frame-buffer color values. Because scan-conversion algorithms generate pixel positions at successive unit intervals, incremental operations can also be used to access the frame buffer efficiently at each step of the scan-conversion process.

As a specific example, suppose the frame buffer array is addressed in row-major order and that pixel positions are labeled from (0, 0) at the lower-left corner to (x_{max}, y_{max}) at the top-right corner (Figure 10) of the screen. For a bilevel system (one bit per pixel), the frame-buffer bit address for pixel position (x, y) is calculated as

$$\text{addr}(x, y) = \text{addr}(0, 0) + y(x_{max} + 1) + x \tag{23}$$

Moving across a scan line, we can calculate the frame-buffer address for the pixel at $(x + 1, y)$ as the following offset from the address for position (x, y):

$$\text{addr}(x + 1, y) = \text{addr}(x, y) + 1 \tag{24}$$

Stepping diagonally up to the next scan line from (x, y), we get to the frame-buffer address of $(x + 1, y + 1)$ with the calculation

$$\text{addr}(x + 1, y + 1) = \text{addr}(x, y) + x_{max} + 2 \tag{25}$$

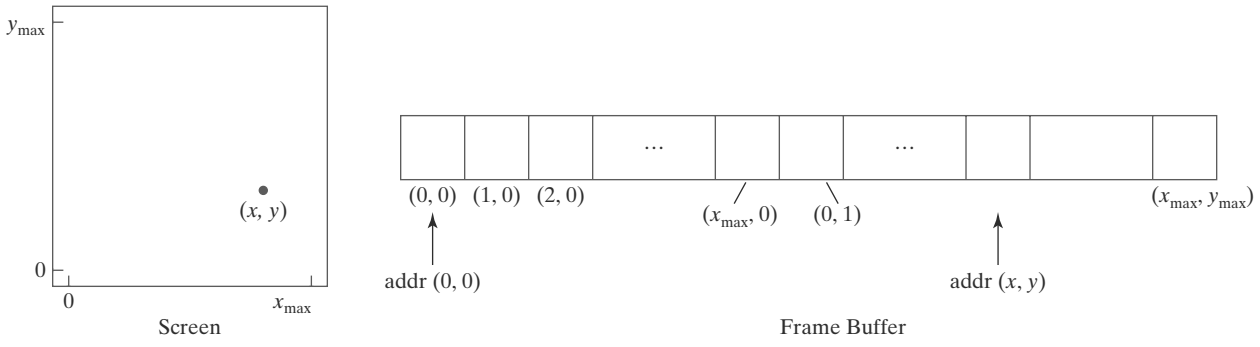


FIGURE 10
Pixel screen positions stored linearly in row-major order within the frame buffer.

where the constant $x_{max} + 2$ is precomputed once for all line segments. Similar incremental calculations can be obtained from Equation 23 for unit steps in the negative x and y screen directions. Each of the address calculations involves only a single integer addition.

Methods for implementing these procedures depend on the capabilities of a particular system and the design requirements of the software package. With systems that can display a range of intensity values for each pixel, frame-buffer address calculations include pixel width (number of bits), as well as the pixel screen location.

4 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

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \tag{26}$$

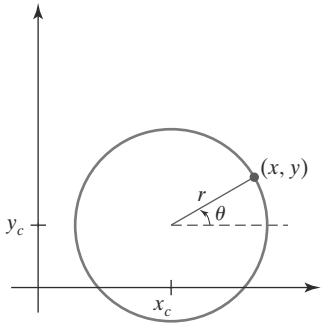


FIGURE 11
Circle with center coordinates (x_c, y_c) and radius r.

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

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

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.

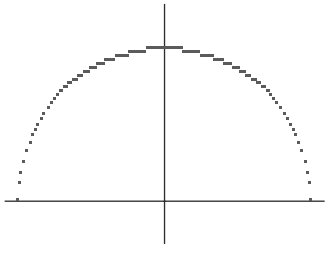


FIGURE 12
Upper half of a circle plotted with Equation 27 and with $(x_c, y_c) = (0, 0)$.

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

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

$$x = x_c + r\cos\theta$$
$$y = y_c + r\sin\theta \tag{28}$$

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 $\frac{1}{r}$. This plots pixel positions that are approximately one unit apart. Although polar coordinates provide equal point spacing, the trigonometric calculations are still time-consuming.

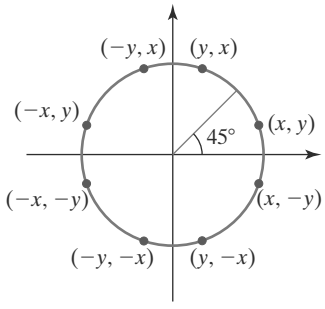


FIGURE 13
Symmetry of a circle. Calculation of a circle point (x, y) in one octant yields the circle points shown for the other seven octants.

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° 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.

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

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

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:

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

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.

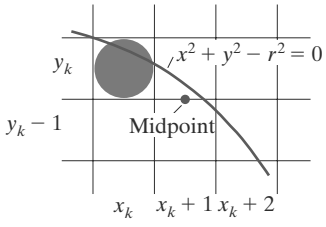


FIGURE 14
Midpoint between candidate pixels at sampling position $x_k + 1$ along a circular path.

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:

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

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 midposition 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

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

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

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

$$p_0 = f_{\text{circ}}\left(1, r - \frac{1}{2}\right)$$
$$= 1 + \left(r - \frac{1}{2}\right)^2 - r^2$$

or

$$p_0 = \frac{5}{4} - r \tag{33}$$

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

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

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:
$$x = x + x_c, \qquad y = y + y_c$$
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

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

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.

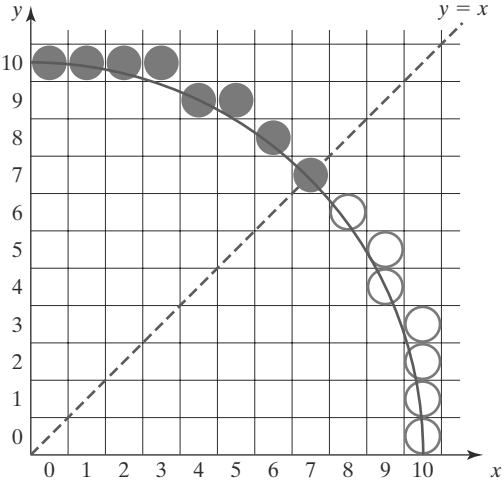


FIGURE 15
Pixel positions (solid circles) along a circle path centered on the origin and with radius $r = 10$, as calculated by the midpoint circle algorithm. Open ("hollow") circles show the symmetry positions in the first quadrant.

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 += 2 * circPt.getx ( ) + 1;
        else {
            circPt.decrementy ( );
            p += 2 * (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.gety ( ), yc + circPt.getx ( ));
    setPixel (xc - circPt.gety ( ), yc + circPt.getx ( ));
    setPixel (xc + circPt.gety ( ), yc - circPt.getx ( ));
    setPixel (xc - circPt.gety ( ), yc - circPt.getx ( ));
}
```

5 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.

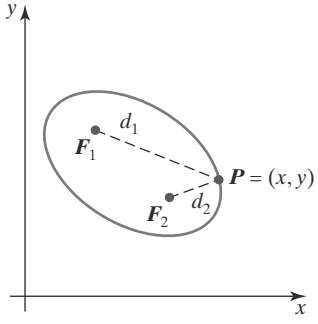


FIGURE 16
Ellipse generated about foci F_1 and F_2.

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 $\mathbf{P} = (x, y)$ on the ellipse are labeled d_1 and d_2, then the general equation of an ellipse can be stated as

$$d_1 + d_2 = \text{constant} \tag{34}$$

Expressing distances d_1 and d_2 in terms of the focal coordinates $\mathbf{F}_1 = (x_1, y_1)$ and $\mathbf{F}_2 = (x_2, y_2)$, we have

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

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

$$A x^2 + B y^2 + C x y + D x + E y + F = 0 \tag{36}$$

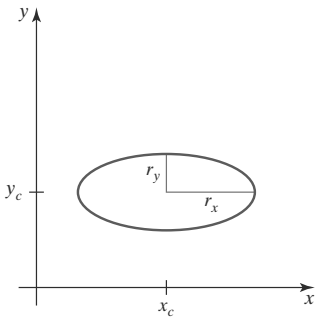


FIGURE 17
Ellipse centered at (x_c, y_c) with semimajor axis r_x and semiminor axis r_y.

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

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

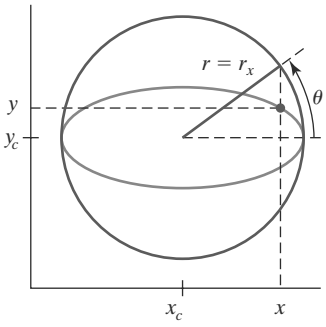


FIGURE 18
The bounding circle and eccentric angle θ for an ellipse with $r_x > r_y$.

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

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

Angle θ, 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).

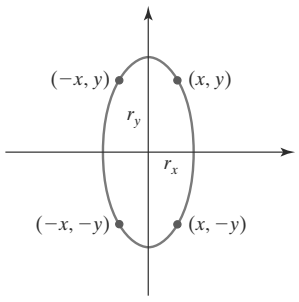


FIGURE 19
Symmetry of an ellipse. Calculation of a point (x, y) in one quadrant yields the ellipse points shown for the other three quadrants.

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

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

or

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

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

$$2r_y^2 x = 0 \tag{45}$$
$$2r_x^2 y = 2r_x^2 r_y \tag{46}$$

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

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

or

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

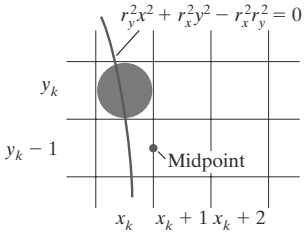


FIGURE 22
Midpoint between candidate pixels at sampling position $y_k - 1$ along an elliptical path.

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

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

If $p2_k > 0$, the midposition 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}.

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 $(x_c, y_c) = (0, 0)$ as

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

which has the following properties:

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

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

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

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

$$2r_y^2 x \geq 2r_x^2 y \tag{42}$$

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:

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

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 midposition is outside or on the ellipse boundary, and we select the pixel on scan line $y_k - 1$.

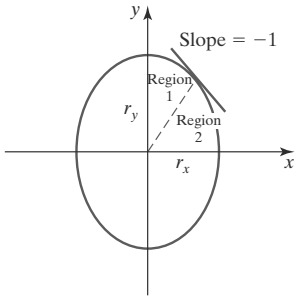


FIGURE 20
Ellipse processing regions. Over region 1, the magnitude of the ellipse slope is less than 1.0; over region 2, the magnitude of the slope is greater than 1.0.

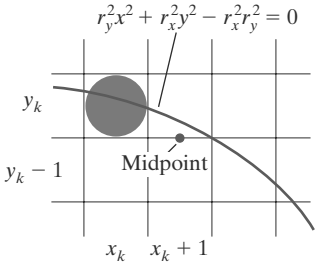


FIGURE 21
Midpoint between candidate pixels at sampling position $x_k + 1$ along an elliptical path.

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

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

or

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

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

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

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
$$p1_{k+1} = p1_k + 2r_y^2 x_{k+1} + r_y^2$$
Otherwise, the next point along the ellipse is $(x_k + 1, y_k - 1)$ and
$$p1_{k+1} = p1_k + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_y^2$$

with

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

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

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

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

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

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

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

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

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_c, y_c) and plot these coordinate values:

$$x = x + x_c, \qquad y = y + y_c$$

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

$$2r_y^2 x = 0 \qquad \text{(with increment } 2r_y^2 = 72\text{)}$$
$$2r_x^2 y = 2r_x^2 r_y \qquad \text{(with increment } -2r_x^2 = -128\text{)}$$

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

$$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4}r_x^2 = -332$$

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})	$2r_y^2 x_{k+1}$	$2r_x^2 y_{k+1}$
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 $2r_y^2 x > 2r_x^2 y$.

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})	$2r_y^2 x_{k+1}$	$2r_x^2 y_{k+1}$
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.

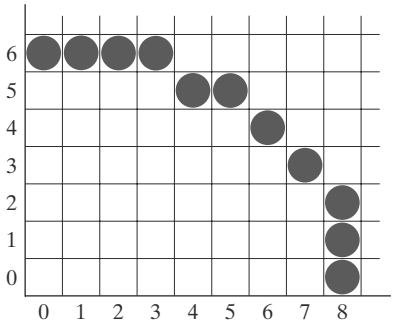


FIGURE 23
Pixel positions along an elliptical path centered on the origin with $r_x = 8$ and $r_y = 6$, using the midpoint algorithm to calculate locations within the first quadrant.

In the following code segment, example procedures are given for implementing the midpoint ellipse algorithm. Values for the ellipse parameters `Rx`, `Ry`, `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 twoRx2 = 2 * Rx2;
   int twoRy2 = 2 * Ry2;
   int p;
   int x = 0;
   int y = Ry;
   int px = 0;
   int py = twoRx2 * y;
   void ellipsePlotPoints (int, int, int, int);
```

```
   /* Plot the initial point in each quadrant. */
   ellipsePlotPoints (xCenter, yCenter, x, y);

   /* Region 1 */
   p = round (Ry2 - (Rx2 * Ry) + (0.25 * Rx2));
   while (px < py) {
      x++;
      px += twoRy2;
      if (p < 0)
         p += Ry2 + px;
      else {
         y--;
         py -= twoRx2;
         p += Ry2 + px - py;
      }
      ellipsePlotPoints (xCenter, yCenter, x, y);
   }

   /* Region 2 */
   p = round (Ry2 * (x+0.5) * (x+0.5) + Rx2 * (y-1) * (y-1) - Rx2 * Ry2);
   while (y > 0) {
      y--;
      py -= twoRx2;
      if (p > 0)
         p += Rx2 - py;
      else {
         x++;
         px += twoRy2;
         p += Rx2 - 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);
}
```

6 Other Curves

Various curve functions are useful in object modeling, animation path specifications, data and function graphing, and other graphics applications. Commonly encountered curves include conics, trigonometric and exponential functions, probability distributions, general polynomials, and spline functions. Displays of these curves can be generated with methods similar to those discussed for the circle and ellipse functions. We can obtain positions along curve paths directly from explicit representations $y = f(x)$ or from parametric forms. Alternatively, we could apply the incremental midpoint method to plot curves described with implicit functions $f(x, y) = 0$.

A simple method for displaying a curved line is to approximate it with straight-line segments. Parametric representations are often useful in this case for obtaining equally spaced positions along the curve path for the line endpoints. We can also generate equally spaced positions from an explicit representation by choosing the independent variable according to the slope of the curve. Where the slope of $y = f(x)$ has a magnitude less than 1, we choose x as the independent variable and calculate y values at equal x increments. To obtain equal spacing where the slope has a magnitude greater than 1, we use the inverse function, $x = f^{-1}(y)$, and calculate values of x at equal y steps.

Straight-line or curve approximations are used to generate a line graph for a set of discrete data values. We could join the discrete points with straight-line segments, or we could use linear regression (least squares) to approximate the data set with a single straight line. A nonlinear least-squares approach is used to display the data set with some approximating function, usually a polynomial.

As with circles and ellipses, many functions possess symmetries that can be exploited to reduce the computation of coordinate positions along curve paths. For example, the normal probability distribution function is symmetric about a center position (the mean), and all points within one cycle of a sine curve can be generated from the points in a 90° interval.

Conic Sections

In general, we can describe a **conic section** (or **conic**) with the second-degree equation

$$A x^2 + B y^2 + C x y + D x + E y + F = 0 \tag{52}$$

where the values for parameters A, B, C, D, E, and F determine the kind of curve that we are to display. Given this set of coefficients, we can determine the particular conic that will be generated by evaluating the discriminant $B^2 - 4AC$:

$$B^2 - 4AC \begin{cases} < 0, & \text{generates an ellipse (or circle)} \\ = 0, & \text{generates a parabola} \\ > 0, & \text{generates a hyperbola} \end{cases} \tag{53}$$

For example, we get the circle equation 26 when $A = B = 1$, $C = 0$, $D = -2x_c$, $E = -2y_c$, and $F = x_c^2 + y_c^2 - r^2$. Equation 52 also describes the "degenerate" conics: points and straight lines.

In some applications, circular and elliptical arcs are conveniently specified with the beginning and ending angular values for the arc, as illustrated in Figure 24. Such arcs are sometimes defined by their endpoint coordinate positions. For either case, we could generate the arc with a modified midpoint method, or we could display a set of approximating straight-line segments.

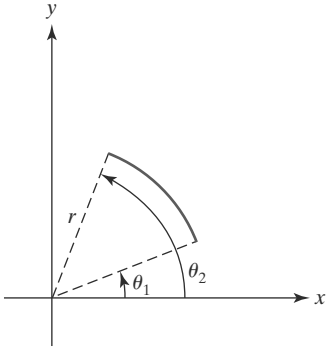


FIGURE 24
A circular arc, centered on the origin, defined with beginning angle θ_1, ending angle θ_2, and radius r.

Ellipses, hyperbolas, and parabolas are particularly useful in certain animation applications. These curves describe orbital and other motions for objects subjected to gravitational, electromagnetic, or nuclear forces. Planetary orbits in the solar system, for example, are approximated with ellipses; and an object projected into a uniform gravitational field travels along a parabolic trajectory. Figure 25 shows a parabolic path in standard position for a gravitational field acting in the negative y direction. The explicit equation for the parabolic trajectory of the object shown can be written as

$$y = y_0 + a(x - x_0)^2 + b(x - x_0) \tag{54}$$

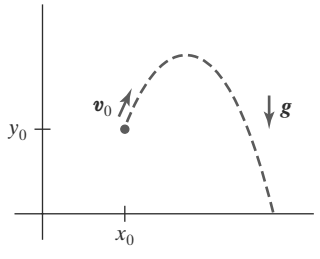


FIGURE 25
Parabolic path of an object tossed into a downward gravitational field at the initial position (x_0, y_0).

construct a cubic polynomial curve section between each pair of specified points. Each curve section is then described in parametric form as

$$\begin{aligned} x &= a_{x0} + a_{x1}u + a_{x2}u^2 + a_{x3}u^3 \\ y &= a_{y0} + a_{y1}u + a_{y2}u^2 + a_{y3}u^3 \end{aligned} \tag{58}$$

where parameter u varies over the interval from 0 to 1.0. Values for the coefficients of u in the preceding equations are determined from boundary conditions on the curve sections. One boundary condition is that two adjacent curve sections have the same coordinate position at the boundary, and a second condition is to match the two curve slopes at the boundary so that we obtain one continuous, smooth curve (Figure 27). Continuous curves that are formed with polynomial pieces are called **spline curves,** or simply **splines**.

FIGURE 27
A spline curve formed with individual cubic polynomial sections between specified coordinate positions.

7 Parallel Curve Algorithms

Methods for exploiting parallelism in curve generation are similar to those used in displaying straight-line segments. We can either adapt a sequential algorithm by allocating processors according to curve partitions, or we could devise other methods and assign processors to screen partitions.

A parallel midpoint method for displaying circles is to divide the circular arc from 45° to 90° into equal subarcs and assign a separate processor to each subarc. As in the parallel Bresenham line algorithm, we then need to set up computations to determine the beginning y value and decision parameter p_k value for each processor. Pixel positions are calculated throughout each subarc, and positions in the other circle octants can be obtained by symmetry. Similarly, a parallel ellipse midpoint method divides the elliptical arc over the first quadrant into equal subarcs and parcels these out to separate processors. Again, pixel positions in the other quadrants are determined by symmetry. A screen-partitioning scheme for circles and ellipses is to assign each scan line that crosses the curve to a separate processor. In this case, each processor uses the circle or ellipse equation to calculate curve intersection coordinates.

For the display of elliptical arcs or other curves, we can simply use the scan-line partitioning method. Each processor uses the curve equation to locate the intersection positions along its assigned scan line. With processors assigned to individual pixels, each processor would calculate the distance (or distance squared) from the curve to its assigned pixel. If the calculated distance is less than a predefined value, the pixel is plotted.

8 Pixel Addressing and Object Geometry

In discussing the raster algorithms for displaying graphics primitives, we assumed that frame-buffer coordinates referenced the center of a screen pixel position. We now consider the effects of different addressing schemes and an alternate pixel-addressing method used by some graphics packages, including OpenGL.

An object description that is input to a graphics program is given in terms of precise world-coordinate positions, which are infinitesimally small mathematical points. However, when the object is scan-converted into the frame buffer, the input description is transformed to pixel coordinates which reference finite screen

with constants a and b determined by the initial velocity $\mathbf{v}_0$ of the object and the acceleration g due to the uniform gravitational force. We can also describe such parabolic motions with parametric equations using a time parameter t, measured in seconds from the initial projection point:

$$\begin{aligned} x &= x_0 + v_{x0}\, t \\ y &= y_0 + v_{y0}\, t - \frac{1}{2} g t^2 \end{aligned} \tag{55}$$

Here, v_{x0} and v_{y0} are the initial velocity components, and the value of g near the surface of the earth is approximately 980 cm/sec^2. Object positions along the parabolic path are then calculated at selected time steps.

Hyperbolic curves (Figure 26) are useful in various scientific-visualization applications. Motions of objects along hyperbolic paths occur in connection with the collision of charged particles and in certain gravitational problems. For example, comets or meteorites moving around the sun may travel along hyperbolic paths and escape to outer space, never to return. The particular branch (left or right, in Figure 26) describing the motion of an object depends on the forces involved in the problem. We can write the standard equation for the hyperbola centered on the origin in Figure 26 as

$$\left(\frac{x}{r_x}\right)^2 - \left(\frac{y}{r_y}\right)^2 = 1 \tag{56}$$

with $x \le -r_x$ for the left branch and $x \ge r_x$ for the right branch. Because this equation differs from the standard ellipse equation 39 only in the sign between the x^2 and y^2 terms, we can generate points along a hyperbolic path with a slightly modified ellipse algorithm.

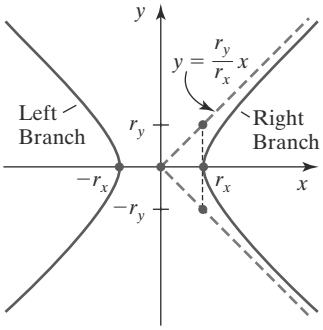


FIGURE 26
Left and right branches of a hyperbola in standard position with the symmetry axis along the x axis.

Parabolas and hyperbolas possess a symmetry axis. For example, the parabola described by Equation 55 is symmetric about the axis

$$x = x_0 + v_{x0} v_{y0} / g$$

The methods used in the midpoint ellipse algorithm can be applied directly to obtain points along one side of the symmetry axis of hyperbolic and parabolic paths in the two regions: (1) where the magnitude of the curve slope is less than 1, and (2) where the magnitude of the slope is greater than 1. To do this, we first select the appropriate form of Equation 52 and then use the selected function to set up expressions for the decision parameters in the two regions.

Polynomials and Spline Curves

A polynomial function of nth degree in x is defined as

$$\begin{aligned} y &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} + a_n x^n \end{aligned} \tag{57}$$

where n is a nonnegative integer and the a_k are constants, with $a_n \neq 0$. We obtain a quadratic curve when $n = 2$, a cubic polynomial when $n = 3$, a quartic curve when $n = 4$, and so forth. We have a straight line when $n = 1$. Polynomials are useful in a number of graphics applications, including the design of object shapes, the specification of animation paths, and the graphing of data trends in a discrete set of data points.

Designing object shapes or motion paths is typically accomplished by first specifying a few points to define the general curve contour, then the selected points are fitted with a polynomial. One way to accomplish the curve fitting is to

areas, and the displayed raster image may not correspond exactly with the relative dimensions of the input object. If it is important to preserve the specified geometry of world objects, we can compensate for the mapping of mathematical input points to finite pixel areas. One way to do this is simply to adjust the pixel dimensions of displayed objects so as to correspond to the dimensions given in the original mathematical description of the scene. For example, if a rectangle is specified as having a width of 40 cm, then we could adjust the screen display so that the rectangle has a width of 40 pixels, with the width of each pixel representing one centimeter. Another approach is to map world coordinates onto screen positions between pixels, so that we align object boundaries with pixel boundaries instead of pixel centers.

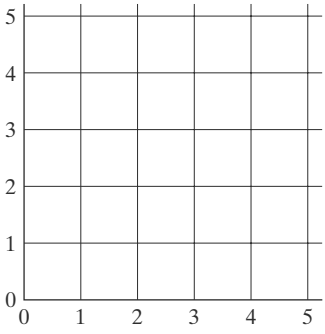


FIGURE 28
Lower-left section of a screen area with coordinate positions referenced by grid intersection lines.

Screen Grid Coordinates

Figure 28 shows a screen section with grid lines marking pixel boundaries, one unit apart. In this scheme, a screen position is given as the pair of integer values identifying a grid-intersection position between two pixels. The address for any pixel is now at its lower-left corner, as illustrated in Figure 29. A straight-line path is now envisioned as between grid intersections. For example, the mathematical line path for a polyline with endpoint coordinates (0, 0), (5, 2), and (1, 4) would then be as shown in Figure 30.

Using screen grid coordinates, we now identify the area occupied by a pixel with screen coordinates (x, y) as the unit square with diagonally opposite corners at (x, y) and $(x + 1, y + 1)$. This pixel-addressing method has several advantages: it avoids half-integer pixel boundaries, it facilitates precise object representations, and it simplifies the processing involved in many scan-conversion algorithms and other raster procedures.

The algorithms for line drawing and curve generation discussed in the preceding sections are still valid when applied to input positions expressed as screen grid coordinates. Decision parameters in these algorithms would now be a measure of screen grid separation differences, rather than separation differences from pixel centers.

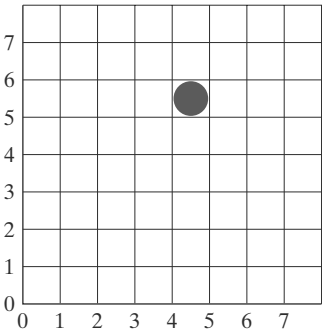


FIGURE 29
Illuminated pixel at raster position (4, 5).

Maintaining Geometric Properties of Displayed Objects

When we convert geometric descriptions of objects into pixel representations, we transform mathematical points and lines into finite screen areas. If we are to maintain the original geometric measurements specified by the input coordinates for an object, we need to account for the finite size of pixels when we transform the object definition to a screen display.

Figure 31 shows the line plotted in the Bresenham line-algorithm example of Section 1. Interpreting the line endpoints (20, 10) and (30, 18) as precise grid-crossing positions, we see that the line should not extend past screen-grid position (30, 18). If we were to plot the pixel with screen coordinates (30, 18), as in the example given in Section 1, we would display a line that spans 11 horizontal units and 9 vertical units. For the mathematical line, however, $\Delta x = 10$ and $\Delta y = 8$. If we are addressing pixels by their center positions, we can adjust the length of the displayed line by omitting one of the endpoint pixels. But if we think of screen coordinates as addressing pixel boundaries, as shown in Figure 31, we plot a line using only those pixels that are "interior" to the line path; that is, only those pixels that are between the line endpoints. For our example, we would plot the leftmost pixel at (20, 10) and the rightmost pixel at (29, 17). This displays a

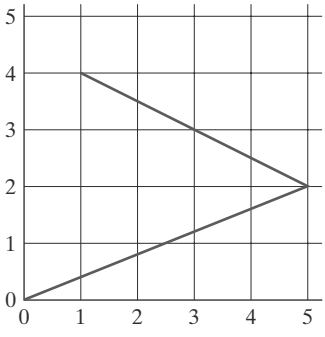


FIGURE 30
Line path for two connected line segments between screen grid-coordinate positions.

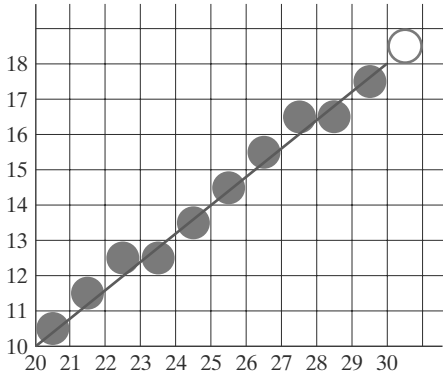


FIGURE 31
Line path and corresponding pixel display for grid endpoint coordinates (20, 10) and (30, 18).

line that has the same geometric magnitudes as the mathematical line from (20, 10) to (30, 18).

For an enclosed area, input geometric properties are maintained by displaying the area using only those pixels that are interior to the object boundaries. The rectangle defined with the screen coordinate vertices shown in Figure 32(a), for example, is larger when we display it filled with pixels up to and including the border pixel lines joining the specified vertices [Figure 32(b)]. As defined, the area of the rectangle is 12 units, but as displayed in Figure 32(b), it has an area of 20 units. In Figure 32(c), the original rectangle measurements are maintained by displaying only the internal pixels. The right boundary of the input rectangle is at $x = 4$. To maintain the rectangle width in the display, we set the rightmost pixel grid coordinate for the rectangle at $x = 3$ because the pixels in this vertical column span the interval from $x = 3$ to $x = 4$. Similarly, the mathematical top boundary of the rectangle is at $y = 3$, so we set the top pixel row for the displayed rectangle at $y = 2$.

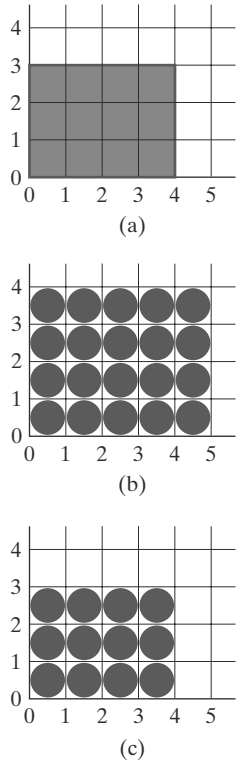


FIGURE 32
Conversion of rectangle (a) with vertices at screen coordinates (0, 0), (4, 0), (4, 3), and (0, 3) into display (b), which includes the right and top boundaries, and into display (c), which maintains geometric magnitudes.

These compensations for finite pixel size can be applied to other objects, including those with curved boundaries, so that the raster display maintains the input object specifications. A circle with radius 5 and center position (10, 10), for instance, would be displayed as in Figure 33 by the midpoint circle algorithm

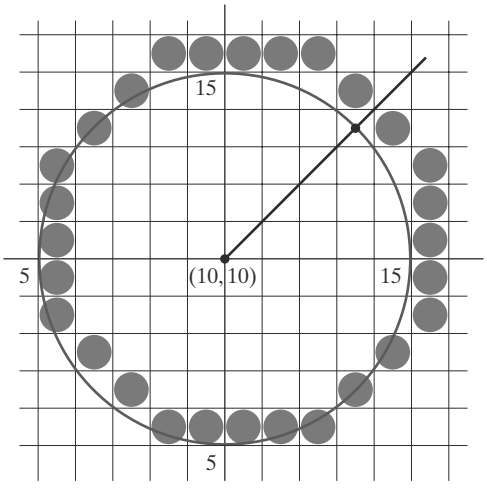


FIGURE 33
A midpoint-algorithm plot of the circle equation $(x - 10)^2 + (y - 10)^2 = 5^2$ using pixel-center coordinates.

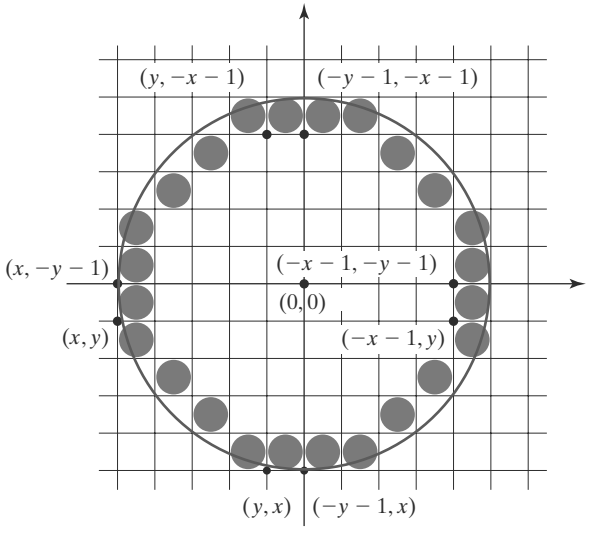


FIGURE 34
Modification of the circle plot in Figure 33 to maintain the specified circle diameter of 10.

using pixel centers as screen-coordinate positions. However, the plotted circle has a diameter of 11. To plot the circle with the defined diameter of 10, we can modify the circle algorithm to shorten each pixel scan line and each pixel column, as in Figure 34. One way to do this is to generate points clockwise along the circular arc in the third quadrant, starting at screen coordinates (10, 5). For each generated point, the other seven circle symmetry points are generated by decreasing the x coordinate values by 1 along scan lines and decreasing the y coordinate values by 1 along pixel columns. Similar methods are applied in ellipse algorithms to maintain the specified proportions in the display of an ellipse.

9 Attribute Implementations for Straight-Line Segments and Curves

Recall that line segment primitives can be displayed with three basic attributes: color, width, and style. Of these, line width and style are selected with separate line functions.

Line Width

Implementation of line-width options depends on the capabilities of the output device. 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. If a line has slope magnitude less than or equal to 1.0, we can modify a line-drawing routine to display thick lines by plotting a vertical span of pixels in each column (x position) along the line. The number of pixels to be displayed in each column is set equal to the integer value of the line width. In Figure 35, we display a double-width line by generating a parallel line above the original line path. At each x sampling position, we calculate the corresponding y coordinate and plot pixels at screen coordinates (x, y) and $(x, y + 1)$. We could display lines with a width of 3 or greater by alternately plotting pixels above and below the single-width line path.

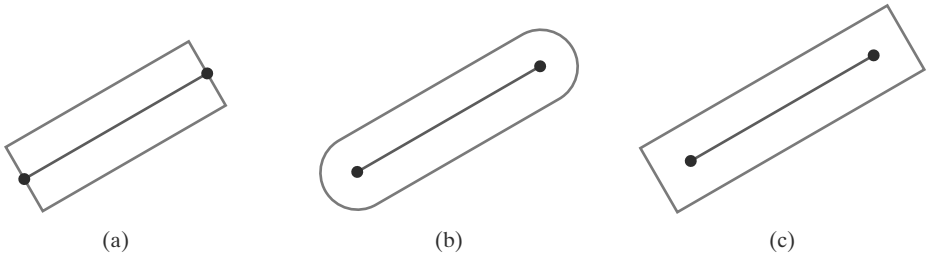


FIGURE 37
Thick lines drawn with (a) butt caps, (b) round caps, and (c) projecting square caps.

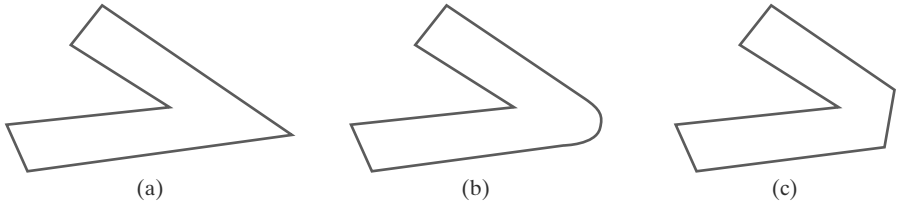


FIGURE 38
Thick line segments connected with a miter join (a), a round join (b), and a bevel join (c).

and add butt caps that are positioned half of the line width beyond the specified endpoints.

Other methods for producing thick lines include displaying the line as a filled rectangle or generating the line with a selected pen or brush pattern, as discussed in the next section. To obtain a rectangle representation for the line boundary, we calculate the position of the rectangle vertices along perpendiculars to the line path so that the rectangle vertex coordinates are displaced from the original line-endpoint positions by half the line width. The rectangular line then appears as in Figure 37(a). We could add round caps to the filled rectangle, or we could extend its length to display projecting square caps.

Generating thick polylines requires some additional considerations. In general, the methods that we have considered for displaying a single line segment will not produce a smoothly connected series of line segments. Displaying thick polylines using horizontal and vertical pixel spans, for example, leaves pixel gaps at the boundaries between line segments with different slopes where there is a shift from horizontal pixel spans to vertical spans. We can generate thick polylines that are smoothly joined at the cost of additional processing at the segment endpoints. Figure 38 shows three possible methods for smoothly joining two line segments. A *miter join* is accomplished by extending the outer boundaries of each of the two line segments until they meet. A *round join* is produced by capping the connection between the two segments with a circular boundary whose diameter is equal to the line width. A *bevel join* is generated by displaying the line segments with butt caps and filling in the triangular gap where the segments meet. If the angle between two connected line segments is very small, a miter join can generate a long spike that distorts the appearance of the polyline. A graphics package can avoid this effect by switching from a miter join to a bevel join when, for example, the angle between any two consecutive segments is small.

Line Style

Raster line algorithms display line-style attributes by plotting pixel spans. For dashed, dotted, and dot-dashed patterns, the line-drawing procedure outputs sections of contiguous pixels along the line path, skipping over a number of intervening pixels between the solid spans. Pixel counts for the span length and

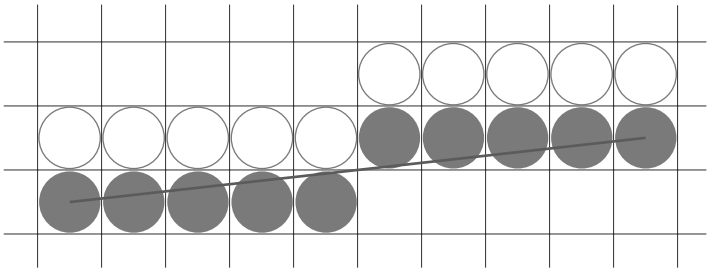


FIGURE 35
A double-wide raster line with slope $|m| < 1.0$ generated with vertical pixel spans.

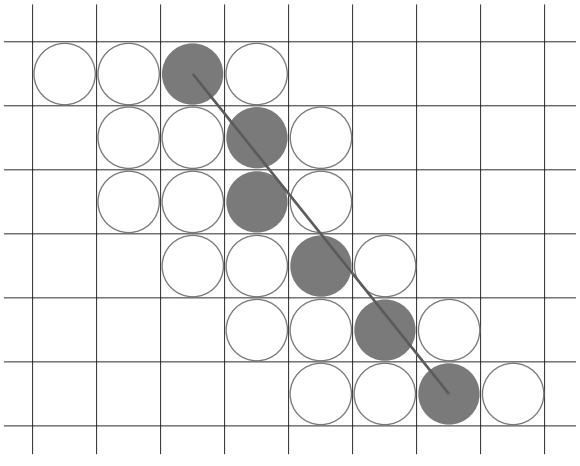


FIGURE 36
A raster line with slope $|m| > 1.0$ and a line width of 4 plotted using horizontal pixel spans.

With a line slope greater than 1.0 in magnitude, we can display thick lines using horizontal spans, alternately picking up pixels to the right and left of the line path. This scheme is demonstrated in Figure 36, where a line segment with a width of 4 is plotted using multiple pixels across each scan line. Similarly, a thick line with slope less than or equal to 1.0 can be displayed using vertical pixel spans. We can implement this procedure by comparing the magnitudes of the horizontal and vertical separations (Δx and Δy) of the line endpoints. If $|\Delta x| \geq |\Delta y|$, pixels are replicated along columns. Otherwise, multiple pixels are plotted across rows.

Although thick lines are generated quickly by plotting horizontal or vertical pixel spans, the displayed width of a line (measured perpendicular to the line path) depends on its slope. A 45° line will be displayed thinner by a factor of $1/\sqrt{2}$ compared to a horizontal or vertical line plotted with the same-length pixel spans.

Another problem with implementing width options using horizontal or vertical pixel spans is that the method produces lines whose ends are horizontal or vertical regardless of the slope of the line. This effect is more noticeable with very thick lines. We can adjust the shape of the line ends to give them a better appearance by adding **line caps** (Figure 37). One kind of line cap is the *butt cap*, which has square ends that are perpendicular to the line path. If the specified line has slope m, the square ends of the thick line have slope $-1/m$. Each of the component parallel lines is then displayed between the two perpendicular lines at each end of the specified line path. Another line cap is the *round cap* obtained by adding a filled semicircle to each butt cap. The circular arcs are centered at the middle of the thick line and have a diameter equal to the line thickness. A third type of line cap is the *projecting square cap*. Here, we simply extend the line

inter-span spacing can be specified in a **pixel mask,** which is a pattern of binary digits indicating which positions to plot along the line path. The linear mask 11111000, for instance, could be used to display a dashed line with a dash length of five pixels and an inter-dash spacing of three pixels. Pixel positions corresponding to the 1 bits are assigned the current color, and pixel positions corresponding to the 0 bits are displayed in the background color.

Plotting dashes with a fixed number of pixels results in unequal length dashes for different line orientations, as illustrated in Figure 39. Both dashes shown are plotted with four pixels, but the diagonal dash is longer by a factor of $\sqrt{2}$. For precision drawings, dash lengths should remain approximately constant for any line orientation. To accomplish this, we could adjust the pixel counts for the solid spans and inter-span spacing according to the line slope. In Figure 39, we can display approximately equal length dashes by reducing the diagonal dash to three pixels. Another method for maintaining dash length is to treat dashes as individual line segments. Endpoint coordinates for each dash are located and passed to the line routine, which then calculates pixel positions along the dash path.

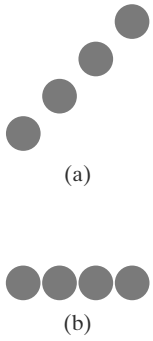


FIGURE 39
Unequal-length dashes displayed with the same number of pixels.

Pen and Brush Options

Pen and brush shapes can be stored in a pixel mask that identifies the array of pixel positions that are to be set along the line path. For example, a rectangular pen could be implemented with the mask shown in Figure 40 by moving the center (or one corner) of the mask along the line path, as in Figure 41. To avoid setting pixels more than once in the frame buffer, we can simply accumulate the horizontal spans generated at each position of the mask and keep track of the beginning and ending x positions for the spans across each scan line.

Lines generated with pen (or brush) shapes can be displayed in various widths by changing the size of the mask. For example, the rectangular pen line in Figure 41 could be narrowed with a 2×2 rectangular mask or widened with a 4×4 mask. Also, lines can be displayed with selected patterns by superimposing the pattern values onto the pen or brush mask.

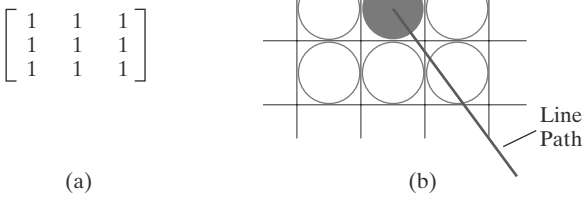


FIGURE 40
A pixel mask (a) for a rectangular pen, and the associated array of pixels (b) displayed by centering the mask over a specified pixel position.

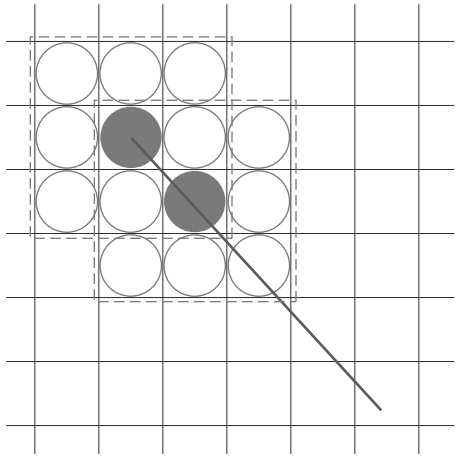


FIGURE 41
Generating a line with the pen shape of Figure 40.

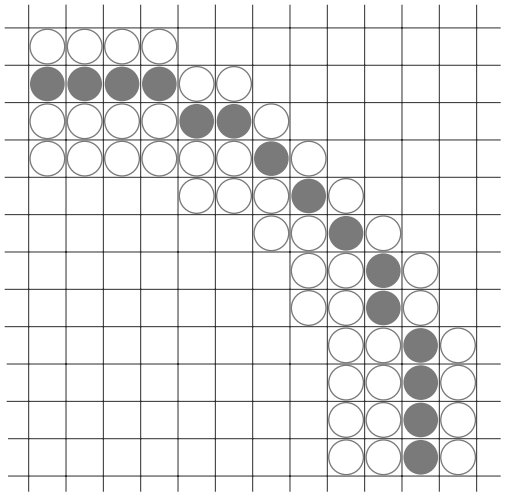


FIGURE 42
A circular arc of width 4 plotted with either vertical or horizontal pixel spans, depending on the slope.

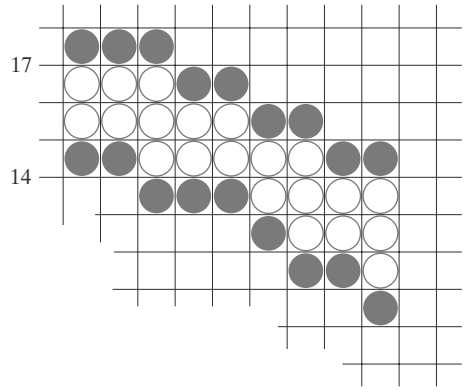


FIGURE 43
A circular arc of width 4 and radius 16 displayed by filling the region between two concentric arcs.

Curve Attributes

Methods for adapting curve-drawing algorithms to accommodate attribute selections are similar to those for line drawing. Raster curves of various widths can be displayed using the method of horizontal or vertical pixel spans. Where the magnitude of the curve slope is less than or equal to 1.0, we plot vertical spans; where the slope magnitude is greater than 1.0, we plot horizontal spans. Figure 42 demonstrates this method for displaying a circular arc with a width of 4 in the first quadrant. Using circle symmetry, we generate the circle path with vertical spans in the octant from $x = 0$ to $x = y$, and then reflect pixel positions about the line $y = x$ to obtain the remainder of the curve shown. Circle sections in the other quadrants are obtained by reflecting pixel positions in the first quadrant about the coordinate axes. The thickness of curves displayed with this method is again a function of curve slope. Circles, ellipses, and other curves will appear thinnest where the slope has a magnitude of 1.

Another method for displaying thick curves is to fill in the area between two parallel curve paths, whose separation distance is equal to the desired width. We could do this using the specified curve path as one boundary and setting up the second boundary either inside or outside the original curve path. This approach, however, shifts the original curve path either inward or outward, depending on which direction we choose for the second boundary. We can maintain the original curve position by setting the two boundary curves at a distance of half the width on either side of the specified curve path. An example of this approach is shown in Figure 43 for a circle segment with a radius of 16 and a specified width of 4. The boundary arcs are then set at a separation distance of 2 on either side of the radius of 16. To maintain the proper dimensions of the circular arc, as discussed in Section 8, we can set the radii for the concentric boundary arcs at $r = 14$ and $r = 17$. Although this method is accurate for generating thick circles, it provides, in general, only an approximation to the true area of other thick curves. For example, the inner and outer boundaries of a fat ellipse generated with this method do not have the same foci.

The pixel masks discussed for implementing line-style options could also be used in raster curve algorithms to generate dashed or dotted patterns. For example, the mask 11100 produces the dashed circular arc shown in Figure 44.

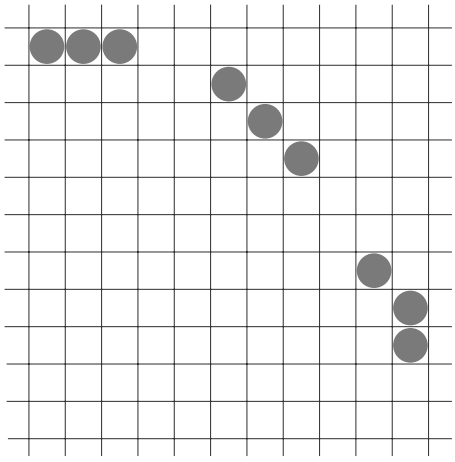


FIGURE 44
A dashed circular arc displayed with a dash span of 3 pixels and an inter-dash spacing of 2 pixels.

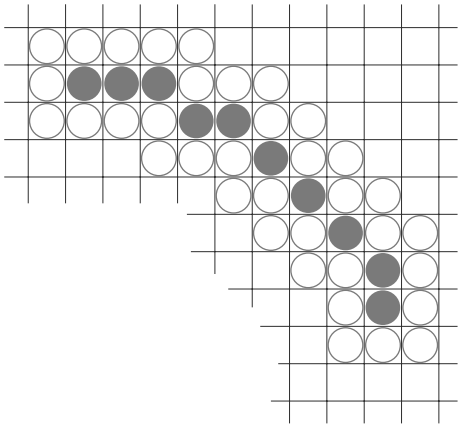


FIGURE 45
A circular arc displayed with a rectangular pen.

We can generate the dashes in the various octants using circle symmetry, but we must shift the pixel positions to maintain the correct sequence of dashes and spaces as we move from one octant to the next. Also, as in straight-line algorithms, pixel masks display dashes and inter-dash spaces that vary in length according to the slope of the curve. If we want to display constant length dashes, we need to adjust the number of pixels plotted in each dash as we move around the circle circumference. Instead of applying a pixel mask with constant spans, we plot pixels along equal angular arcs to produce equal-length dashes.

Pen (or brush) displays of curves are generated using the same techniques discussed for straight-line segments. We replicate a pen shape along the line path, as illustrated in Figure 45 for a circular arc in the first quadrant. Here, the center of the rectangular pen is moved to successive curve positions to produce the curve shape shown. Curves displayed with a rectangular pen in this manner will be thicker where the magnitude of the curve slope is 1. A uniform curve thickness can be displayed by rotating the rectangular pen to align it with the slope direction as we move around the curve or by using a circular pen shape. Curves drawn with pen and brush shapes can be displayed in different sizes and with superimposed patterns or simulated brush strokes.

10 General Scan-Line Polygon-Fill Algorithm

A scan-line fill of a region is performed by first determining the intersection positions of the boundaries of the fill region with the screen scan lines. Then the fill colors are applied to each section of a scan line that lies within the interior of the fill region. The scan-line fill algorithm identifies the same interior regions as the odd-even rule. The simplest area to fill is a polygon because each scan-line intersection point with a polygon boundary is obtained by solving a pair of simultaneous linear equations, where the equation for the scan line is simply $y =$ constant.

Figure 46 illustrates the basic scan-line procedure for a solid-color fill of a polygon. For each scan line that crosses the polygon, the edge intersections are

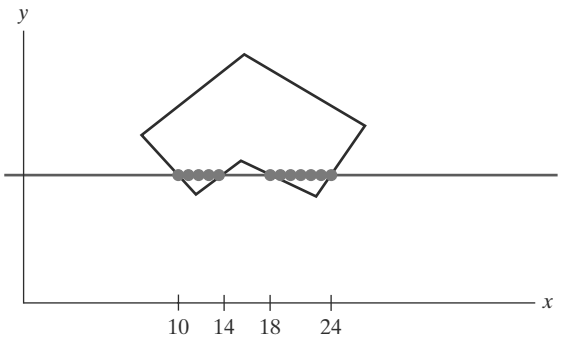


FIGURE 46
Interior pixels along a scan line passing through a polygon fill area.

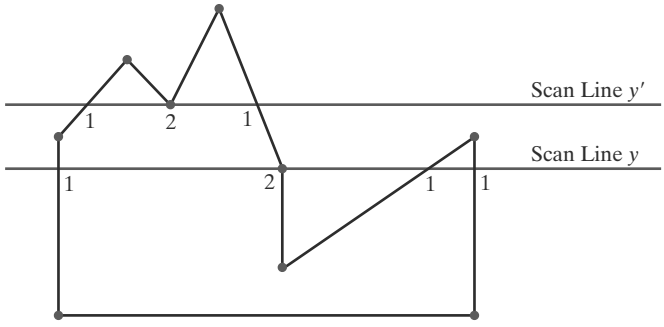


FIGURE 47
Intersection points along scan lines that intersect polygon vertices. Scan line y generates an odd number of intersections, but scan line y' generates an even number of intersections that can be paired to identify correctly the interior pixel spans.

sorted from left to right, and then the pixel positions between, and including, each intersection pair are set to the specified fill color. In the example of Figure 46, the four pixel intersection positions with the polygon boundaries define two stretches of interior pixels. Thus, the fill color is applied to the five pixels from $x = 10$ to $x = 14$ and to the seven pixels from $x = 18$ to $x = 24$. If a pattern fill is to be applied to the polygon, then the color for each pixel along a scan line is determined from its overlap position with the fill pattern.

However, the scan-line fill algorithm for a polygon is not quite as simple as Figure 46 might suggest. Whenever a scan line passes through a vertex, it intersects two polygon edges at that point. In some cases, this can result in an odd number of boundary intersections for a scan line. Figure 47 shows two scan lines that cross a polygon fill area and intersect a vertex. Scan line y' intersects an even number of edges, and the two pairs of intersection points along this scan line correctly identify the interior pixel spans. But scan line y intersects five polygon edges. To identify the interior pixels for scan line y, we must count the vertex intersection as only one point. Thus, as we process scan lines, we need to distinguish between these cases.

We can detect the topological difference between scan line y and scan line y' in Figure 47 by noting the position of the intersecting edges relative to the scan line. For scan line y, the two edges sharing an intersection vertex are on opposite sides of the scan line. But for scan line y', the two intersecting edges are both above the scan line. Thus, a vertex that has adjoining edges on opposite sides of an intersecting scan line should be counted as just one boundary intersection point. We can identify these vertices by tracing around the polygon boundary in either clockwise or counterclockwise order and observing the relative changes in vertex y coordinates as we move from one edge to the next. If the three endpoint y values of two consecutive edges monotonically increase or decrease, we need to count the shared (middle) vertex as a single intersection point for the scan line passing through that vertex. Otherwise, the shared vertex represents a local extremum (minimum or maximum) on the polygon boundary, and the two edge intersections with the scan line passing through that vertex can be added to the intersection list.

One method for implementing the adjustment to the vertex-intersection count is to shorten some polygon edges to split those vertices that should be counted as one intersection. We can process nonhorizontal edges around the polygon boundary in the order specified, either clockwise or counterclockwise. As we

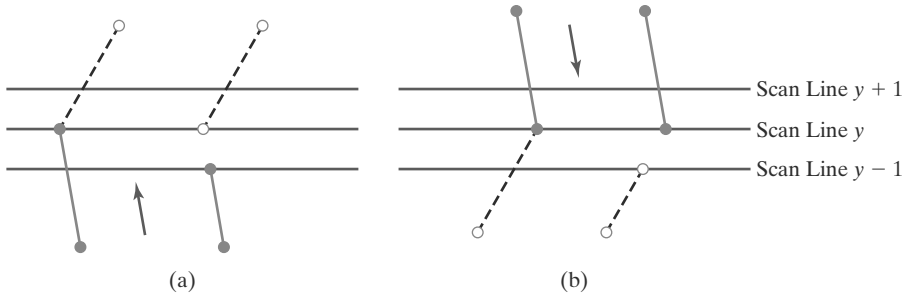


FIGURE 48
Adjusting endpoint y values for a polygon, as we process edges in order around the polygon perimeter. The edge currently being processed is indicated as a solid line. In (a), the y coordinate of the upper endpoint of the current edge is decreased by 1. In (b), the y coordinate of the upper endpoint of the next edge is decreased by 1.

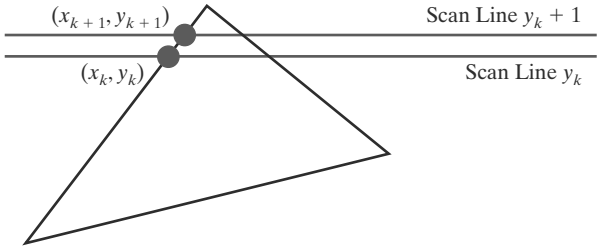


FIGURE 49
Two successive scan lines intersecting a polygon boundary.

process each edge, we can check to determine whether that edge and the next nonhorizontal edge have either monotonically increasing or decreasing endpoint y values. If so, the lower edge can be shortened to ensure that only one intersection point is generated for the scan line going through the common vertex joining the two edges. Figure 48 illustrates the shortening of an edge. When the endpoint y coordinates of the two edges are increasing, the y value of the upper endpoint for the current edge is decreased by 1, as in Figure 48(a). When the endpoint y values are monotonically decreasing, as in Figure 48(b), we decrease the y coordinate of the upper endpoint of the edge following the current edge.

Typically, certain properties of one part of a scene are related in some way to the properties in other parts of the scene, and these **coherence properties** can be used in computer-graphics algorithms to reduce processing. Coherence methods often involve incremental calculations applied along a single scan line or between successive scan lines. For example, in determining fill-area edge intersections, we can set up incremental coordinate calculations along any edge by exploiting the fact that the slope of the edge is constant from one scan line to the next. Figure 49 shows two successive scan lines crossing the left edge of a triangle. The slope of this edge can be expressed in terms of the scan-line intersection coordinates:

$$m = \frac{y_{k+1} - y_k}{x_{k+1} - x_k} \tag{59}$$

Because the change in y coordinates between the two scan lines is simply

$$y_{k+1} - y_k = 1 \tag{60}$$

the x-intersection value x_{k+1} on the upper scan line can be determined from the x-intersection value x_k on the preceding scan line as

$$x_{k+1} = x_k + \frac{1}{m} \tag{61}$$

Each successive x intercept can thus be calculated by adding the inverse of the slope and rounding to the nearest integer.

An obvious parallel implementation of the fill algorithm is to assign each scan line that crosses the polygon to a separate processor. Edge intersection calculations are then performed independently. Along an edge with slope m, the intersection x_k value for scan line k above the initial scan line can be calculated as

$$x_k = x_0 + \frac{k}{m} \tag{62}$$

In a sequential fill algorithm, the increment of x values by the amount $\frac{1}{m}$ along an edge can be accomplished with integer operations by recalling that the slope m is the ratio of two integers:

$$m = \frac{\Delta y}{\Delta x}$$

where Δx and Δy are the differences between the edge endpoint x and y coordinate values. Thus, incremental calculations of x intercepts along an edge for successive scan lines can be expressed as

$$x_{k+1} = x_k + \frac{\Delta x}{\Delta y} \tag{63}$$

Using this equation, we can perform integer evaluation of the x intercepts by initializing a counter to 0, then incrementing the counter by the value of Δx each time we move up to a new scan line. Whenever the counter value becomes equal to or greater than Δy, we increment the current x intersection value by 1 and decrease the counter by the value Δy. This procedure is equivalent to maintaining integer and fractional parts for x intercepts and incrementing the fractional part until we reach the next integer value.

As an example of this integer-incrementing scheme, suppose that we have an edge with slope $m = \frac{7}{3}$. At the initial scan line, we set the counter to 0 and the counter increment to 3. As we move up to the next three scan lines along this edge, the counter is successively assigned the values 3, 6, and 9. On the third scan line above the initial scan line, the counter now has a value greater than 7. So we increment the x intersection coordinate by 1 and reset the counter to the value $9 - 7 = 2$. We continue determining the scan-line intersections in this way until we reach the upper endpoint of the edge. Similar calculations are carried out to obtain intersections for edges with negative slopes.

We can round to the nearest pixel x intersection value, instead of truncating to obtain integer positions, by modifying the edge-intersection algorithm so that the increment is compared to $\Delta y/2$. This can be done with integer arithmetic by incrementing the counter with the value $2\Delta x$ at each step and comparing the increment to Δy. When the increment is greater than or equal to Δy, we increase the x value by 1 and decrement the counter by the value of $2\Delta y$. In our previous example with $m = \frac{7}{3}$, the counter values for the first few scan lines above the initial scan line on this edge would now be 6, 12 (reduced to -2), 4, 10 (reduced to -4), 2, 8 (reduced to -6), 0, 6, and 12 (reduced to -2). Now x would be incremented on scan lines 2, 4, 6, 9, and so forth, above the initial scan line for this edge. The extra calculations required for each edge are $2\Delta x = \Delta x + \Delta x$ and $2\Delta y = \Delta y + \Delta y$, which are carried out as preprocessing steps.

To perform a polygon fill efficiently, we can first store the polygon boundary in a *sorted edge table* that contains all the information necessary to process the scan lines efficiently. Proceeding around the edges in either a clockwise or a counterclockwise order, we can use a bucket sort to store the edges, sorted on the smallest y value of each edge, in the correct scan-line positions. Only nonhorizontal edges are entered into the sorted edge table. As the edges are processed, we can also

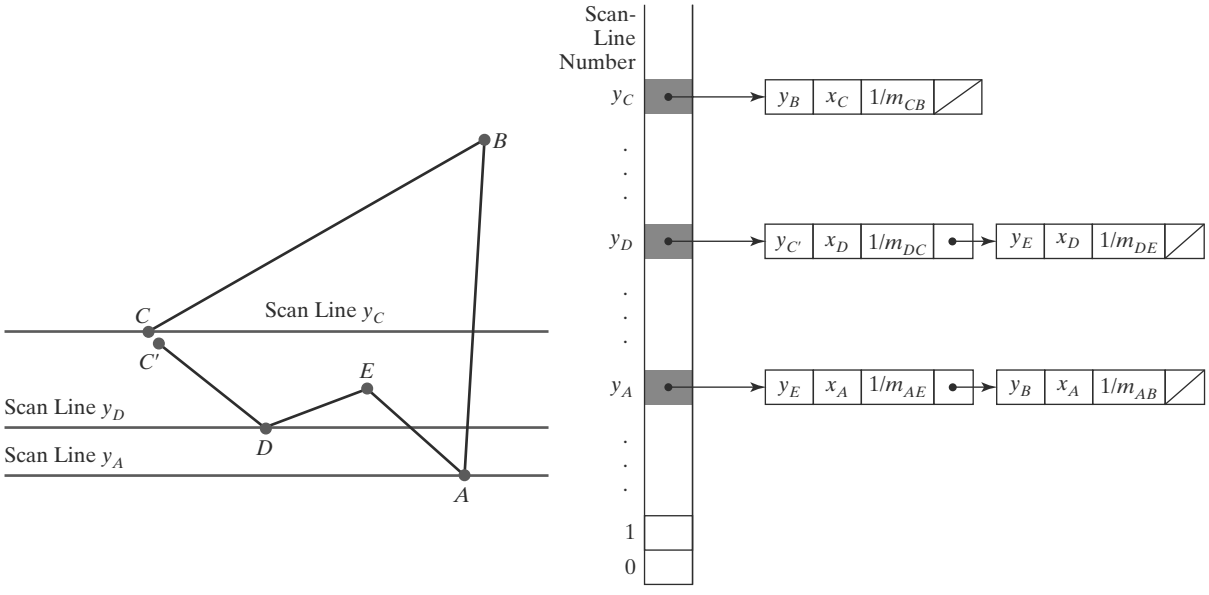


FIGURE 50
A polygon and its sorted edge table, with edge $\overline{DC}$ shortened by one unit in the y direction.

shorten certain edges to resolve the vertex-intersection question. Each entry in the table for a particular scan line contains the maximum y value for that edge, the x-intercept value (at the lower vertex) for the edge, and the inverse slope of the edge. For each scan line, the edges are in sorted order from left to right. Figure 50 shows a polygon and the associated sorted edge table.

Next, we process the scan lines from the bottom of the polygon to its top, producing an *active edge list* for each scan line crossing the polygon boundaries. The active edge list for a scan line contains all edges crossed by that scan line, with iterative coherence calculations used to obtain the edge intersections.

Implementation of edge-intersection calculations can be facilitated by storing Δx and Δy values in the sorted edge list. Also, to ensure that we correctly fill the interior of specified polygons, we can apply the considerations discussed in Section 8. For each scan line, we fill in the pixel spans for each pair of x intercepts starting from the leftmost x intercept value and ending at one position before the rightmost x intercept. Each polygon edge can be shortened by one unit in the y direction at the top endpoint. These measures also guarantee that pixels in adjacent polygons will not overlap.

11 Scan-Line Fill of Convex Polygons

When we apply a scan-line fill procedure to a convex polygon, there can be no more than a single interior span for each screen scan line. So we need to process the polygon edges only until we have found two boundary intersections for each scan line crossing the polygon interior.

The general polygon scan-line algorithm discussed in the preceding section can be simplified considerably for convex-polygon fill. We again use coordinate extents to determine which edges cross a scan line. Intersection calculations with these edges then determine the interior pixel span for that scan line, where any

vertex crossing is counted as a single boundary intersection point. When a scan line intersects a single vertex (at an apex, for example), we plot only that point. Some graphics packages further restrict fill areas to be triangles. This makes filling even easier because each triangle has just three edges to process.

12 Scan-Line Fill for Regions with Curved Boundaries

Because an area with curved boundaries is described with nonlinear equations, a scan-line fill generally takes more time than a polygon scan-line fill. We can use the same general approach detailed in Section 10, but the boundary intersection calculations are performed with curve equations. In addition, the slope of the boundary is continuously changing, so we cannot use the straightforward incremental calculations that are possible with straight-line edges.

For simple curves such as circles or ellipses, we can apply fill methods similar to those for convex polygons. Each scan line crossing a circle or ellipse interior has just two boundary intersections; and we can determine these two intersection points along the boundary of a circle or an ellipse using the incremental calculations in the midpoint method. Then we simply fill in the horizontal pixel spans from one intersection point to the other. Symmetries between quadrants (and between octants for circles) are used to reduce the boundary calculations.

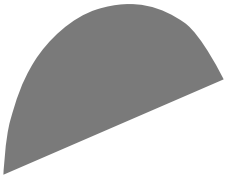


FIGURE 51
Interior fill of an elliptical arc.

Similar methods can be used to generate a fill area for a curve section. For example, an area bounded by an elliptical arc and a straight line section (Figure 51) can be filled using a combination of curve and line procedures. Symmetries and incremental calculations are exploited whenever possible to reduce computations.

Filling other curve areas can involve considerably more processing. We could use similar incremental methods in combination with numerical techniques to determine the scan-line intersections, but usually such curve boundaries are approximated with straight-line segments.

13 Fill Methods for Areas with Irregular Boundaries

Another approach for filling a specified area is to start at an inside position and "paint" the interior, point by point, out to the boundary. This is a particularly useful technique for filling areas with irregular borders, such as a design created with a paint program. Generally, these methods require an input starting position inside the area to be filled and some color information about either the boundary or the interior.

We can fill irregular regions with a single color or with a color pattern. For a pattern fill, we overlay a color mask. As each pixel within the region is processed, its color is determined by the corresponding values in the overlaid pattern.

Boundary-Fill Algorithm

If the boundary of some region is specified in a single color, we can fill the interior of this region, pixel by pixel, until the boundary color is encountered. This

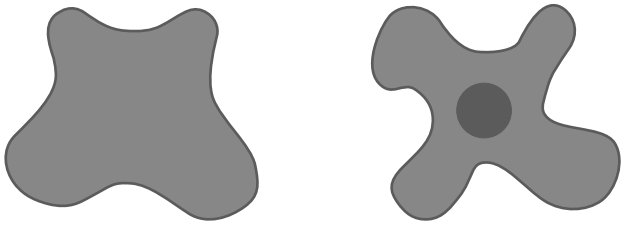


FIGURE 52
Example of color boundaries for a boundary-fill procedure.

method, called the **boundary-fill algorithm,** is employed in interactive painting packages, where interior points are easily selected. Using a graphics tablet or other interactive device, an artist or designer can sketch a figure outline, select a fill color from a color menu, specify the area boundary color, and pick an interior point. The figure interior is then painted in the fill color. Both inner and outer boundaries can be set up to define an area for boundary fill, and Figure 52 illustrates examples for specifying color regions.

Basically, a boundary-fill algorithm starts from an interior point (x, y) and tests the color of neighboring positions. If a tested position is not displayed in the boundary color, its color is changed to the fill color and its neighbors are tested. This procedure continues until all pixels are processed up to the designated boundary color for the area.

Figure 53 shows two methods for processing neighboring pixels from a current test position. In Figure 53(a), four neighboring points are tested. These are the pixel positions that are right, left, above, and below the current pixel. Areas filled by this method are called **4-connected.** The second method, shown in Figure 53(b), is used to fill more complex figures. Here the set of neighboring positions to be tested includes the four diagonal pixels, as well as those in the cardinal directions. Fill methods using this approach are called **8-connected.** An 8-connected boundary-fill algorithm would correctly fill the interior of the area defined in Figure 54, but a 4-connected boundary-fill algorithm would fill only part of that region.

The following procedure illustrates a recursive method for painting a 4-connected area with a solid color, specified in parameter `fillColor`, up to a boundary color specified with parameter `borderColor`. We can extend this procedure to fill an 8-connected region by including four additional statements to test the diagonal positions $(x \pm 1, y \pm 1)$.

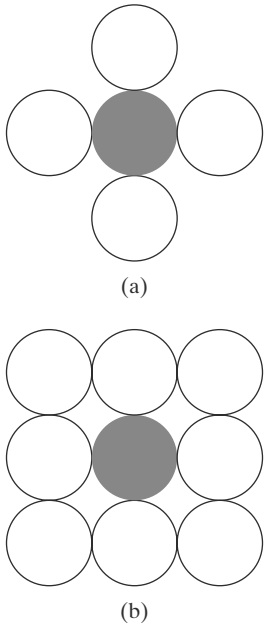


FIGURE 53
Fill methods applied to a 4-connected area (a) and to an 8-connected area (b). Hollow circles represent pixels to be tested from the current test position, shown as a solid color.

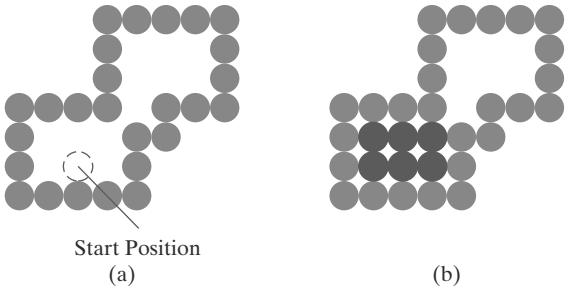


FIGURE 54
The area defined within the color boundary (a) is only partially filled in (b) using a 4-connected boundary-fill algorithm.

```
void boundaryFill4 (int x, int y, int fillColor, int borderColor)
{
   int interiorColor;

   /* Set current color to fillColor, then perform the following operations. */
   getPixel (x, y, interiorColor);
   if ((interiorColor != borderColor) && (interiorColor != fillColor)) {
      setPixel (x, y);    // Set color of pixel to fillColor.
      boundaryFill4 (x + 1, y , fillColor, borderColor);
      boundaryFill4 (x - 1, y , fillColor, borderColor);
      boundaryFill4 (x , y + 1, fillColor, borderColor);
      boundaryFill4 (x , y - 1, fillColor, borderColor)
   }
}
```

Recursive boundary-fill algorithms may not fill regions correctly if some interior pixels are already displayed in the fill color. This occurs because the algorithm checks the next pixels both for boundary color and for fill color. Encountering a pixel with the fill color can cause a recursive branch to terminate, leaving other interior pixels unfilled. To avoid this, we can first change the color of any interior pixels that are initially set to the fill color before applying the boundary-fill procedure.

Also, because this procedure requires considerable stacking of neighboring points, more efficient methods are generally employed. These methods fill horizontal pixel spans across scan lines, instead of proceeding to 4-connected or 8-connected neighboring points. Then we need only stack a beginning position for each horizontal pixel span, instead of stacking all unprocessed neighboring positions around the current position. Starting from the initial interior point with this method, we first fill in the contiguous span of pixels on this starting scan line. Then we locate and stack starting positions for spans on the adjacent scan lines, where spans are defined as the contiguous horizontal string of positions bounded by pixels displayed in the border color. At each subsequent step, we retrieve the next start position from the top of the stack and repeat the process.

An example of how pixel spans could be filled using this approach is illustrated for the 4-connected fill region in Figure 55. In this example, we first process scan lines successively from the start line to the top boundary. After all upper scan lines are processed, we fill in the pixel spans on the remaining scan lines in order down to the bottom boundary. The leftmost pixel position for each horizontal span is located and stacked, in left-to-right order across successive scan lines, as shown in Figure 55. In (a) of this figure, the initial span has been filled, and starting positions 1 and 2 for spans on the next scan lines (below and above) are stacked. In Figure 55(b), position 2 has been unstacked and processed to produce the filled span shown, and the starting pixel (position 3) for the single span on the next scan line has been stacked. After position 3 is processed, the filled spans and stacked positions are as shown in Figure 55(c). Figure 55(d) shows the filled pixels after processing all spans in the upper-right portion of the specified area. Position 5 is next processed, and spans are filled in the upper-left portion of the region; then position 4 is picked up to continue the processing for the lower scan lines.

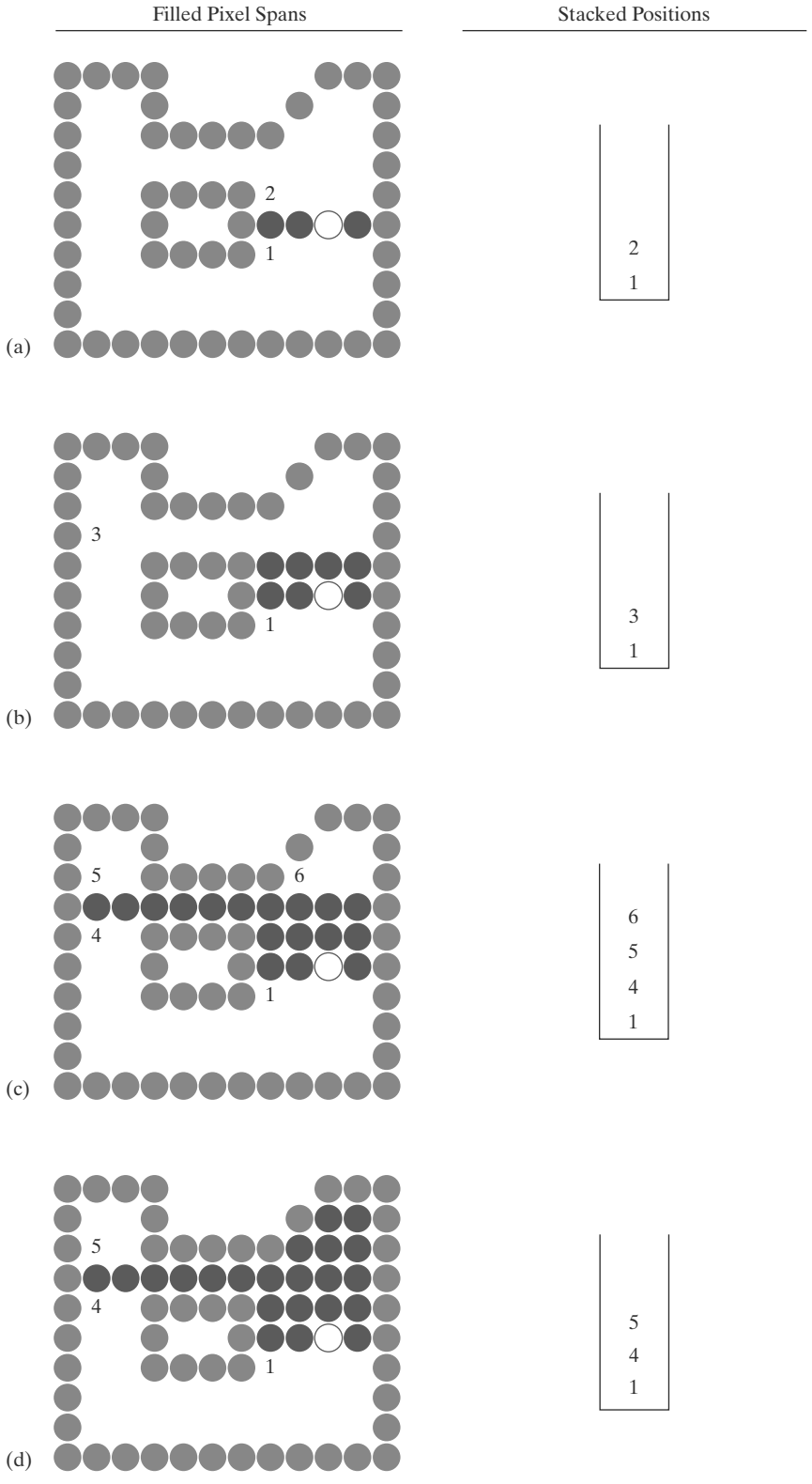


FIGURE 55
Boundary fill across pixel spans for a 4-connected area: (a) Initial scan line with a filled pixel span, showing the position of the initial point (hollow) and the stacked positions for pixel spans on adjacent scan lines. (b) Filled pixel span on the first scan line above the initial scan line and the current contents of the stack. (c) Filled pixel spans on the first two scan lines above the initial scan line and the current contents of the stack. (d) Completed pixel spans for the upper-right portion of the defined region and the remaining stacked positions to be processed.

Flood-Fill Algorithm

FIGURE 56
An area defined within multiple color boundaries.

Sometimes we want to fill in (or recolor) an area that is not defined within a single color boundary. Figure 56 shows an area bordered by several different color regions. We can paint such areas by replacing a specified interior color instead of searching for a particular boundary color. This fill procedure is called a **flood-fill algorithm.** We start from a specified interior point (x, y) and reassign all pixel values that are currently set to a given interior color with the desired fill color. If the area that we want to paint has more than one interior color, we can first reassign pixel values so that all interior points have the same color. Using either a 4-connected or 8-connected approach, we then step through pixel positions until all interior points have been repainted. The following procedure flood fills a 4-connected region recursively, starting from the input position.

```
void floodFill4 (int x, int y, int fillColor, int interiorColor)
{
   int color;

   /* Set current color to fillColor, then perform the following operations. */
   getPixel (x, y, color);
   if (color = interiorColor) {
      setPixel (x, y);    // Set color of pixel to fillColor.
      floodFill4 (x + 1, y, fillColor, interiorColor);
      floodFill4 (x - 1, y, fillColor, interiorColor);
      floodFill4 (x, y + 1, fillColor, interiorColor);
      floodFill4 (x, y - 1, fillColor, interiorColor)
   }
}
```

We can modify the above procedure to reduce the storage requirements of the stack by filling horizontal pixel spans, as discussed for the boundary-fill algorithm. In this approach, we stack only the beginning positions for those pixel spans having the value `interiorColor`. The steps in this modified flood-fill algorithm are similar to those illustrated in Figure 55 for a boundary fill. Starting at the first position of each span, the pixel values are replaced until a value other than `interiorColor` is encountered.

14 Implementation Methods for Fill Styles

There are two basic procedures for filling an area on raster systems, once the definition of the fill region has been mapped to pixel coordinates. One procedure first determines the overlap intervals for scan lines that cross the area. Then, pixel positions along these overlap intervals are set to the fill color. Another method for area filling is to start from a given interior position and "paint" outward, pixel-by-pixel, from this point until we encounter specified boundary conditions. The scan-line approach is usually applied to simple shapes such as circles or regions with polyline boundaries, and general graphics packages use this fill method. Fill algorithms that use a starting interior point are useful for filling areas with more complex boundaries and in interactive painting systems.

Fill Styles

We can implement a pattern fill by determining where the pattern overlaps those scan lines that cross a fill area. Beginning from a specified start position for a pattern fill, we map the rectangular patterns vertically across scan lines and horizontally across pixel positions on the scan lines. Each replication of the pattern array is performed at intervals determined by the width and height of the mask. Where the pattern overlaps the fill area, pixel colors are set according to the values stored in the mask.

Hatch fill could be applied to regions by drawing sets of line segments to display either single hatching or cross-hatching. Spacing and slope for the hatch lines could be set as parameters in a hatch table. Alternatively, hatch fill can be specified as a pattern array that produces sets of diagonal lines.

A reference point (xp, yp) for the starting position of a fill pattern can be set at any convenient position, inside or outside the fill region. For instance, the reference point could be set at a polygon vertex; or the reference point could be chosen as the lower-left corner of the bounding rectangle (or bounding box) determined by the coordinate extents of the region. To simplify selection of the reference coordinates, some packages always use the coordinate origin of the display window as the pattern start position. Always setting (xp, yp) at the coordinate origin also simplifies the tiling operations when each element of a pattern is to be mapped to a single pixel. For example, if the row positions in the pattern array are referenced from bottom to top, starting with the value 1, a color value is then assigned to pixel position (x, y) in screen coordinates from pattern position $(y \bmod ny + 1, x \bmod nx + 1)$. Here, ny and nx specify the number of rows and number of columns in the pattern array. Setting the pattern start position at the coordinate origin, however, effectively attaches the pattern fill to the screen background rather than to the fill regions. Adjacent or overlapping areas filled with the same pattern would show no apparent boundary between the areas. Also, repositioning and refilling an object with the same pattern can result in a shift in the assigned pixel values over the object interior. A moving object would appear to be transparent against a stationary pattern background instead of moving with a fixed interior pattern.

Color-Blended Fill Regions

Color-blended regions can be implemented using either transparency factors to control the blending of background and object colors, or using simple logical or replace operations as shown in Figure 57, which demonstrates how these operations would combine a 2×2 fill pattern with a background pattern for a binary (black-and-white) system.

The *linear soft-fill algorithm* repaints an area that was originally painted by merging a foreground color **F** with a single background color **B**, where $\mathbf{F} \neq \mathbf{B}$. Assuming we know the values for **F** and **B**, we can check the current contents of the frame buffer to determine how these colors were combined. The current color **P** of each pixel within the area to be refilled is some linear combination of **F** and **B**:

$$\mathbf{P} = t\mathbf{F} + (1 - t)\mathbf{B} \tag{64}$$

where the transparency factor t has a value between 0 and 1 for each pixel. For values of t less than 0.5, the background color contributes more to the interior color of the region than does the fill color. If our color values are represented using separate red, green, and blue (RGB) components, Equation 64 holds for

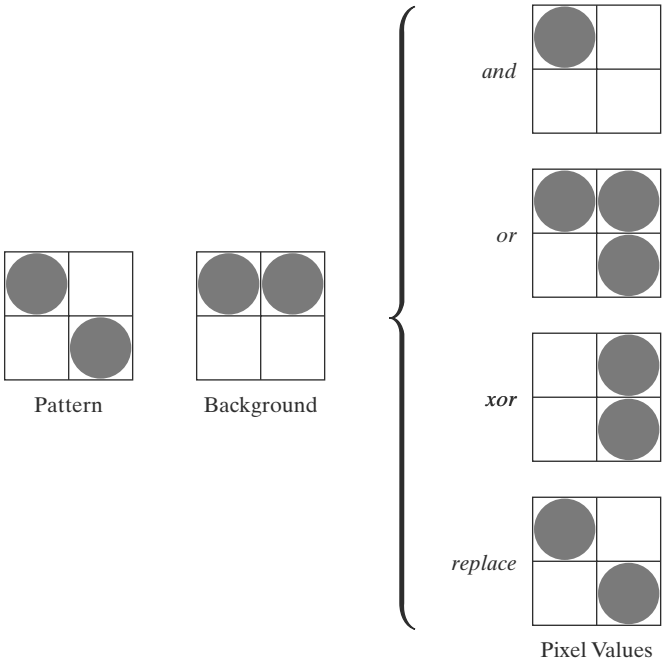


FIGURE 57
Combining a fill pattern with a background pattern using logical operations *and, or,* and *xor* (*exclusive or*), and using simple replacement.

each component of the colors, with

$$\mathbf{P} = (P_R, P_G, P_B), \qquad \mathbf{F} = (F_R, F_G, F_B), \qquad \mathbf{B} = (B_R, B_G, B_B) \tag{65}$$

We can thus calculate the value of parameter t using one of the RGB color components as follows:

$$t = \frac{P_k - B_k}{F_k - B_k} \tag{66}$$

where $k = R$, G, or B; and $F_k \neq B_k$. Theoretically, parameter t has the same value for each RGB component, but the round-off calculations to obtain integer codes can result in different values of t for different components. We can minimize this round-off error by selecting the component with the largest difference between **F** and **B**. This value of t is then used to mix the new fill color **NF** with the background color. We can accomplish this mixing using either a modified flood-fill or boundary-fill procedure, as described in Section 13.

Similar color-blending procedures can be applied to an area whose foreground color is to be merged with multiple background color areas, such as a checkerboard pattern. When two background colors B_1 and B_2 are mixed with foreground color **F**, the resulting pixel color **P** is

$$\mathbf{P} = t_0\mathbf{F} + t_1\mathbf{B_1} + (1 - t_0 - t_1)\mathbf{B_2} \tag{67}$$

where the sum of the color-term coefficients t_0, t_1, and $(1 - t_0 - t_1)$ must equal 1. We can set up two simultaneous equations using two of the three RGB color components to solve for the two proportionality parameters, t_0 and t_1. These parameters are then used to mix the new fill color with the two background colors to obtain the new pixel color. With three background colors and one foreground color, or with two background and two foreground colors, we need all three RGB equations to obtain the relative amounts of the four colors. For some foreground and background color combinations, however, the system of two or three RGB

equations cannot be solved. This occurs when the color values are all very similar or when they are all proportional to each other.

15 Implementation Methods for Antialiasing

Line segments and other graphics primitives generated by the raster algorithms discussed earlier in this chapter have a jagged, or stair-step, appearance because the sampling process digitizes coordinate points on an object to discrete integer pixel positions. This distortion of information due to low-frequency sampling (undersampling) is called **aliasing.** We can improve the appearance of displayed raster lines by applying **antialiasing** methods that compensate for the undersampling process.

An example of the effects of undersampling is shown in Figure 58. To avoid losing information from such periodic objects, we need to set the sampling frequency to at least twice that of the highest frequency occurring in the object, referred to as the **Nyquist sampling frequency** (or Nyquist sampling rate) f_s:

$$f_s = 2 f_{\max} \tag{68}$$

Another way to state this is that the sampling interval should be no larger than one-half the cycle interval (called the **Nyquist sampling interval**). For x-interval sampling, the Nyquist sampling interval Δx_s is

$$\Delta x_s = \frac{\Delta x_{\text{cycle}}}{2} \tag{69}$$

where $\Delta x_{\text{cycle}} = 1/f_{\max}$. In Figure 58, our sampling interval is one and one-half times the cycle interval, so the sampling interval is at least three times too large. If we want to recover all the object information for this example, we need to cut the sampling interval down to one-third the size shown in the figure.

One way to increase sampling rate with raster systems is simply to display objects at higher resolution. However, even at the highest resolution possible with current technology, the jaggies will be apparent to some extent. There is a limit to how big we can make the frame buffer and still maintain the refresh rate at 60 frames or more per second. To represent objects accurately with continuous parameters, we need arbitrarily small sampling intervals. Therefore, unless hardware technology is developed to handle arbitrarily large frame buffers, increased screen resolution is not a complete solution to the aliasing problem.

With raster systems that are capable of displaying more than two intensity levels per color, we can apply antialiasing methods to modify pixel intensities. By

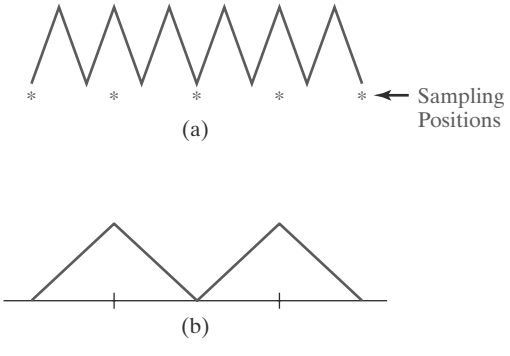


FIGURE 58
Sampling the periodic shape in (a) at the indicated positions produces the aliased lower-frequency representation in (b).

appropriately varying the intensities of pixels along the boundaries of primitives, we can smooth the edges to lessen their jagged appearance.

A straightforward antialiasing method is to increase sampling rate by treating the screen as if it were covered with a finer grid than is actually available. We can then use multiple sample points across this finer grid to determine an appropriate intensity level for each screen pixel. This technique of sampling object characteristics at a high resolution and displaying the results at a lower resolution is called **supersampling** (or **postfiltering,** because the general method involves computing intensities at subpixel grid positions and then combining the results to obtain the pixel intensities). Displayed pixel positions are spots of light covering a finite area of the screen, and not infinitesimal mathematical points. Yet in the line and fill-area algorithms we have discussed, the intensity of each pixel is determined by the location of a single point on the object boundary. By supersampling, we obtain intensity information from multiple points that contribute to the overall intensity of a pixel.

An alternative to supersampling is to determine pixel intensity by calculating the areas of overlap of each pixel with the objects to be displayed. Antialiasing by computing overlap areas is referred to as **area sampling** (or **prefiltering,** because the intensity of the pixel as a whole is determined without calculating subpixel intensities). Pixel overlap areas are obtained by determining where object boundaries intersect individual pixel boundaries.

Raster objects can also be antialiased by shifting the display location of pixel areas. This technique, called **pixel phasing,** is applied by "micropositioning" the electron beam in relation to object geometry. For example, pixel positions along a straight-line segment can be moved closer to the defined line path to smooth out the raster stair-step effect.

Supersampling Straight-Line Segments

We can perform supersampling in several ways. For a straight-line segment, we can divide each pixel into a number of subpixels and count the number of subpixels that overlap the line path. The intensity level for each pixel is then set to a value that is proportional to this subpixel count. An example of this method is given in Figure 59. Each square pixel area is divided into nine equal-sized square subpixels, and the shaded regions show the subpixels that would be selected by Bresenham's algorithm. This scheme provides for three intensity settings above zero, because the maximum number of subpixels that can be selected within any pixel is three. For this example, the pixel at position (10, 20) is set to the maximum intensity (level 3); pixels at (11, 21) and (12, 21) are each set to the next highest intensity (level 2); and pixels at (11, 20) and (12, 22) are each set to the lowest intensity above zero (level 1). Thus, the line intensity is spread out over a greater number of pixels to smooth the original jagged effect. This procedure displays a somewhat blurred line in the vicinity of the stair steps (between horizontal runs). If we want to use more intensity levels to antialiase the line with this method, we increase the number of sampling positions across each pixel. Sixteen subpixels gives us four intensity levels above zero; twenty-five subpixels gives us five levels; and so on.

In the supersampling example of Figure 59, we considered pixel areas of finite size, but we treated the line as a mathematical entity with zero width. Actually, displayed lines have a width approximately equal to that of a pixel. If we take the finite width of the line into account, we can perform supersampling by setting pixel intensity proportional to the number of subpixels inside the polygon representing the line area. A subpixel can be considered to be inside the

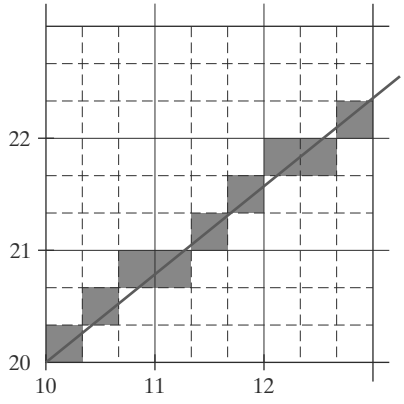


FIGURE 59
Supersampling subpixel positions along a straight-line segment whose left endpoint is at screen coordinates (10, 20).

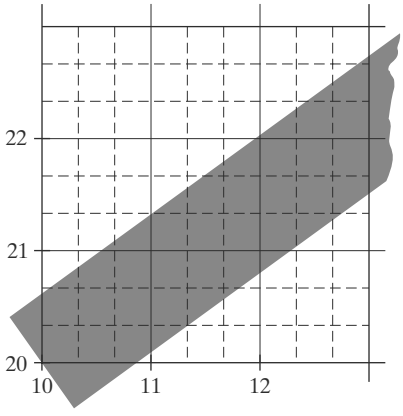


FIGURE 60
Supersampling subpixel positions in relation to the interior of a line of finite width.

line if its lower-left corner is inside the polygon boundaries. An advantage of this supersampling procedure is that the number of possible intensity levels for each pixel is equal to the total number of subpixels within the pixel area. For the example in Figure 59, we can represent this line with finite width by positioning the polygon boundaries parallel to the line path as in Figure 60. In addition, each pixel can now be set to one of nine possible brightness levels above zero.

Another advantage of supersampling with a finite-width line is that the total line intensity is distributed over more pixels. In Figure 60, we now have the pixel at grid position (10, 21) turned on (at intensity level 1), and we also pick up contributions from pixels immediately below and immediately to the left of position (10, 21). Also, if we have a color display, we can extend the method to take background colors into account. A particular line might cross several different color areas, and we can average subpixel intensities to obtain pixel color settings. For instance, if five subpixels within a particular pixel area are determined to be inside the boundaries for a red line and the remaining four subpixels fall within a blue background area, we can calculate the color for this pixel as

$$\text{pixel}_{\text{color}} = \frac{(5 \cdot \text{red} + 4 \cdot \text{blue})}{9}$$

The trade-off for these gains from supersampling a finite-width line is that identifying interior subpixels requires more calculations than simply determining which subpixels are along the line path. Also, we need to take into account the positioning of the line boundaries in relation to the line path. This positioning depends on the slope of the line. For a 45° line, the line path is centered on the polygon area; but for either a horizontal or a vertical line, we want the line path to be one of the polygon boundaries. For example, a horizontal line passing through grid coordinates (10, 20) could be represented as the polygon bounded by horizontal grid lines $y = 20$ and $y = 21$. Similarly, the polygon representing a vertical line through (10, 20) can have vertical boundaries along grid lines $x = 10$ and $x = 11$. For lines with slope $|m| < 1$, the mathematical line path will be positioned proportionately closer to either the lower or the upper polygon boundary depending upon where the line intersects the polygon; in Figure 59, for instance, the line intersects the pixel at (10, 20) closer to the lower boundary, but intersects the pixel at (11, 20) closer to the upper boundary. Similarly, for lines with slope $|m| > 1$, the line path is placed closer to the left or right polygon boundary depending on where it intersects the polygon.

1	2	1
2	4	2
1	2	1

FIGURE 61
Relative weights for a grid of 3×3 subpixels.

Subpixel Weighting Masks

Supersampling algorithms are often implemented by giving more weight to subpixels near the center of a pixel area because we would expect these subpixels to be more important in determining the overall intensity of a pixel. For the 3×3 pixel subdivisions we have considered so far, a weighting scheme as in Figure 61 could be used. The center subpixel here is weighted four times that of the corner subpixels and twice that of the remaining subpixels. Intensities calculated for each of the nine subpixels would then be averaged so that the center subpixel is weighted by a factor of $\frac{1}{4}$; the top, bottom, and side subpixels are each weighted by a factor of $\frac{1}{8}$; and the corner subpixels are each weighted by a factor of $\frac{1}{16}$. An array of values specifying the relative importance of subpixels is usually referred to as a *weighting mask*. Similar masks can be set up for larger subpixel grids. Also, these masks are often extended to include contributions from subpixels belonging to neighboring pixels, so that intensities can be averaged with adjacent pixels to provide a smoother intensity variation between pixels.

Area Sampling Straight-Line Segments

We perform area sampling for a straight line by setting pixel intensity proportional to the area of overlap of the pixel with the finite-width line. The line can be treated as a rectangle, and the section of the line area between two adjacent vertical (or two adjacent horizontal) screen grid lines is then a polygon. Overlap areas for pixels are calculated by determining how much of the polygon overlaps each pixel in that column (or row). In Figure 60, the pixel with screen grid coordinates (10, 20) is about 90 percent covered by the line area, so its intensity would be set to 90 percent of the maximum intensity. Similarly, the pixel at (10, 21) would be set to an intensity of about 15 percent of maximum. A method for estimating pixel overlap areas is illustrated by the supersampling example in Figure 60. The total number of subpixels within the line boundaries is approximately equal to the overlap area, and this estimation is improved by using finer subpixel grids.

Filtering Techniques

A more accurate method for antialiasing lines is to use **filtering** techniques. The method is similar to applying a weighted pixel mask, but now we imagine a continuous *weighting surface* (or *filter function*) covering the pixel. Figure 62 shows examples of rectangular, conical, and Gaussian filter functions. Methods for applying the filter function are similar to those for applying a weighting mask, but now we integrate over the pixel surface to obtain the weighted average intensity. To reduce computation, table lookups are commonly used to evaluate the integrals.

Pixel Phasing

On raster systems that can address subpixel positions within the screen grid, pixel phasing can be used to antialias objects. A line display is smoothed with this technique by moving (micropositioning) pixel positions closer to the line path. Systems incorporating *pixel phasing* are designed so that the electron beam can be shifted by a fraction of a pixel diameter. The electron beam is typically shifted by $\frac{1}{4}$, $\frac{1}{2}$, or $\frac{3}{4}$ of a pixel diameter to plot points closer to the true path of a line or object edge. Some systems also allow the size of individual pixels to be adjusted as an additional means for distributing intensities. Figure 63 illustrates the antialiasing effects of pixel phasing on a variety of line paths.

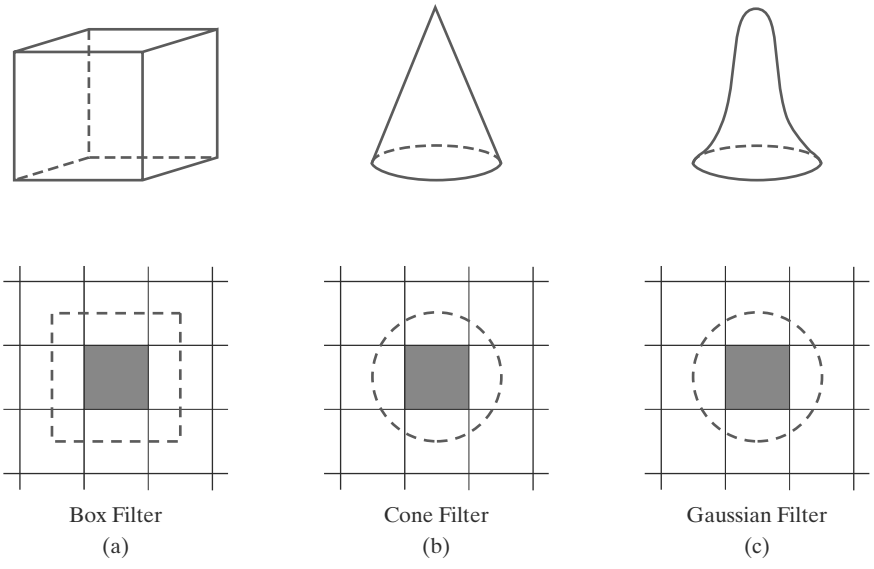


FIGURE 62
Common filter functions used to antialias line paths. The volume of each filter is normalized to 1.0, and the height gives the relative weight at any subpixel position.

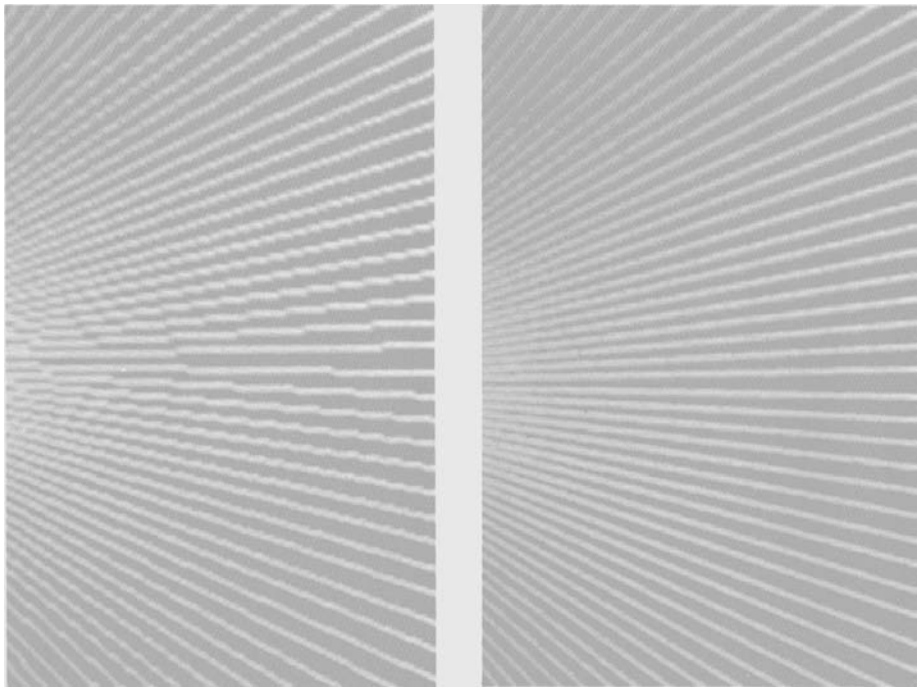


FIGURE 63
Jagged lines (a), plotted on the Merlin 9200 system, are smoothed (b) with an antialiasing technique called pixel phasing. This technique increases the number of addressable points on the system from 768 by 576 to 3072 by 2304. (*Courtesy of Peritek Corp.*)

Compensating for Line-Intensity Differences

Antialiasing a line to soften the stair-step effect also compensates for another raster effect, illustrated in Figure 64. Both lines are plotted with the same number of pixels, yet the diagonal line is longer than the horizontal line by a factor of $\sqrt{2}$. For example, if the horizontal line had a length of 10 centimeters, the diagonal line would have a length of more than 14 centimeters. The visual effect of this is that the diagonal line appears less bright than the horizontal line, because the diagonal line is displayed with a lower intensity per unit length. A line-drawing algorithm could be adapted to compensate for this effect by adjusting the intensity of each

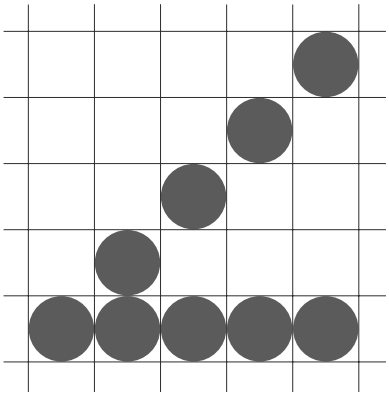


FIGURE 64
Unequal length lines displayed with the same number of pixels in each line.

line according to its slope. Horizontal and vertical lines would be displayed with the lowest intensity, while 45° lines would be given the highest intensity. But if antialiasing techniques are applied to a display, intensities are compensated automatically. When the finite width of a line is taken into account, pixel intensities are adjusted so that the line displays a total intensity proportional to its length.

Antialiasing Area Boundaries

The antialiasing concepts that we have discussed for lines can also be applied to the boundaries of areas to remove their jagged appearance. We can incorporate these procedures into a scan-line algorithm to smooth out the boundaries as the area is generated.

If system capabilities permit the repositioning of pixels, we could smooth area boundaries by shifting pixel positions closer to the boundary. Other methods adjust pixel intensity at a boundary position according to the percent of the pixel area that is interior to the object. In Figure 65, the pixel at position (x, y) has about half its area inside the polygon boundary. Therefore, the intensity at that position would be adjusted to one-half its assigned value. At the next position $(x + 1, y + 1)$ along the boundary, the intensity is adjusted to about one-third the assigned value for that point. Similar adjustments, based on the percent of pixel area coverage, are applied to the other intensity values around the boundary.

Supersampling methods can be applied by determining the number of subpixels that are in the interior of an object. A partitioning scheme with four subareas per pixel is shown in Figure 66. The original 4×4 grid of pixels is turned into an 8×8 grid, and we now process eight scan lines across this grid instead of four. Figure 67 shows one of the pixel areas in this grid that overlaps an object

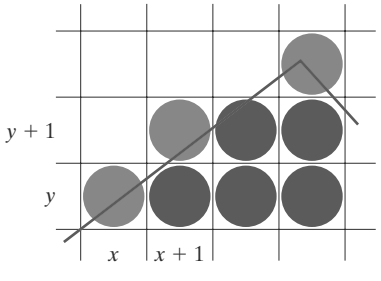


FIGURE 65
Adjusting pixel intensities along an area boundary.

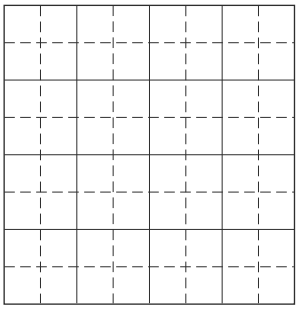


FIGURE 66
A 4×4 pixel section of a raster display subdivided into an 8×8 grid.

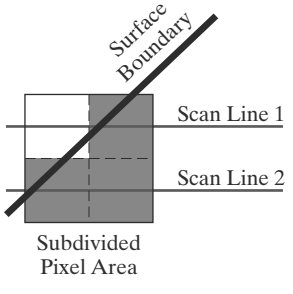


FIGURE 67
A subdivided pixel area with three subdivisions inside an object boundary line.

boundary. Along the two scan lines, we determine that three of the subpixel areas are inside the boundary. So we set the pixel intensity at 75 percent of its maximum value.

Another method for determining the percentage of pixel area within a fill region, developed by Pitteway and Watkinson, is based on the midpoint line algorithm. This algorithm selects the next pixel along a line by testing the location of the midposition between two pixels. As in the Bresenham algorithm, we set up a decision parameter p whose sign tells us which of the next two candidate pixels is closer to the line. By slightly modifying the form of p, we obtain a quantity that also gives the percentage of the current pixel area that is covered by an object.

We first consider the method for a line with slope m in the range from 0 to 1. In Figure 68, a straight-line path is shown on a pixel grid. Assuming that the pixel at position (x_k, y_k) has been plotted, the next pixel nearest the line at $x = x_k + 1$ is either the pixel at y_k or the one at $y_k + 1$. We can determine which pixel is nearer with the calculation

$$y - y_{\text{mid}} = [m(x_k + 1) + b] - (y_k + 0.5) \tag{70}$$

This gives the vertical distance from the actual y coordinate on the line to the halfway point between pixels at position y_k and $y_k + 1$. If this difference calculation is negative, the pixel at y_k is closer to the line. If the difference is positive, the pixel at $y_k + 1$ is closer. We can adjust this calculation so that it produces a positive number in the range from 0 to 1 by adding the quantity $1 - m$:

$$p = [m(x_k + 1) + b] - (y_k + 0.5) + (1 - m) \tag{71}$$

Now the pixel at y_k is nearer if $p < 1 - m$, and the pixel at $y_k + 1$ is nearer if $p > 1 - m$.

Parameter p also measures the amount of the current pixel that is overlapped by the area. For the pixel at (x_k, y_k) in Figure 69, the interior part of the pixel is trapezoidal and has an area that can be calculated as

$$\text{area} = m \cdot x_k + b - y_k + 0.5 \tag{72}$$

This expression for the overlap area of the pixel at (x_k, y_k) is the same as that for parameter p in Equation 71. Therefore, by evaluating p to determine the next pixel position along the polygon boundary, we also determine the percentage of area coverage for the current pixel.

We can generalize this algorithm to accommodate lines with negative slopes and lines with slopes greater than 1. This calculation for parameter p could then

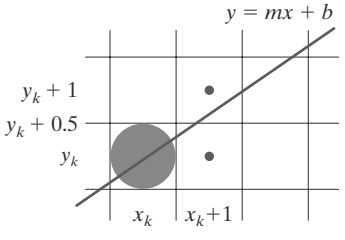


FIGURE 68
Boundary edge of a fill area passing through a pixel grid section.

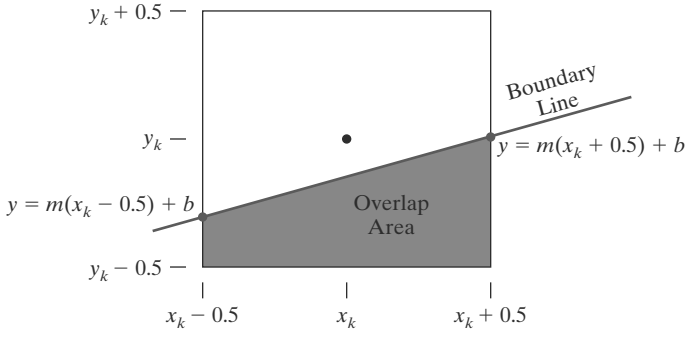


FIGURE 69
Overlap area of a pixel rectangle, centered at position (x_k, y_k), with the interior of a polygon fill area.

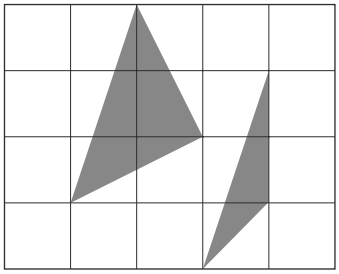


FIGURE 70
Polygons with more than one boundary line passing through individual pixel regions.

be incorporated into a midpoint line algorithm to locate pixel positions along a polygon edge and, concurrently, adjust pixel intensities along the boundary lines. Also, we can adjust the calculations to reference pixel coordinates at their lower-left coordinates and maintain area proportions, as discussed in Section 8.

At polygon vertices and for very skinny polygons, as shown in Figure 70, we have more than one boundary edge passing through a pixel area. For these cases, we need to modify the Pitteway-Watkinson algorithm by processing all edges passing through a pixel and determining the correct interior area.

Filtering techniques discussed for line antialiasing can also be applied to area edges. In addition, the various antialiasing methods can be applied to polygon areas or to regions with curved boundaries. Equations describing the boundaries are used to estimate the amount of pixel overlap with the area to be displayed, and coherence techniques are used along and between scan lines to simplify the calculations.

16 Summary

Three methods that can be used to locate pixel positions along a straight-line path are the DDA algorithm, Bresenham's algorithm, and the midpoint method. Bresenham's line algorithm and the midpoint line method are equivalent, and they are the most efficient. Color values for the pixel positions along the line path are efficiently stored in the frame buffer by incrementally calculating the memory addresses. Any of the line-generating algorithms can be adapted to a parallel implementation by partitioning the line segments and distributing the partitions among the available processors.

Circles and ellipses can be efficiently and accurately scan-converted using midpoint methods and taking curve symmetry into account. Other conic sections (parabolas and hyperbolas) can be plotted with similar methods. Spline curves, which are piecewise continuous polynomials, are widely used in animation and in CAD. Parallel implementations for generating curve displays can be accomplished with methods similar to those for parallel line processing.

To account for the fact that displayed lines and curves have finite widths, we can adjust the pixel dimensions of objects to coincide to the specified geometric dimensions. This can be done with an addressing scheme that references pixel positions at their lower-left corner, or by adjusting line lengths.

Scan-line methods are commonly used to fill polygons, circles, and ellipses. Across each scan line, the interior fill is applied to pixel positions between each pair of boundary intersections, left to right. For polygons, scan-line intersections with vertices can result in an odd number of intersections. This can be resolved by shortening some polygon edges. Scan-line fill algorithms can be simplified if fill areas are restricted to convex polygons. A further simplification is achieved if all fill areas in a scene are triangles. The interior pixels along each scan line are assigned appropriate color values, depending on the fill-attribute specifications. Painting programs generally display fill regions using a boundary-fill method or a flood-fill method. Each of these two fill methods requires an initial interior point. The interior is then painted pixel by pixel from the initial point out to the region boundaries.

Soft-fill procedures provide a new fill color for a region that has the same variations as the previous fill color. One example of this approach is the linear soft-fill algorithm that assumes that the previous fill was a linear combination of foreground and background colors. This same linear relationship is then

determined from the frame buffer settings and used to repaint the area in a new color.

We can improve the appearance of raster primitives by applying antialiasing procedures that adjust pixel intensities. One method for doing this is to supersample. That is, we consider each pixel to be composed of subpixels and we calculate the intensity of the subpixels and average the values of all subpixels. We can also weight the subpixel contributions according to position, giving higher weights to the central subpixels. Alternatively, we can perform area sampling and determine the percentage of area coverage for a screen pixel, then set the pixel intensity proportional to this percentage. Another method for antialiasing is to build special hardware configurations that can shift pixel positions.

REFERENCES

Basic information on Bresenham's algorithms can be found in Bresenham (1965 and 1977). For midpoint methods, see Kappel (1985). Parallel methods for generating lines and circles are discussed in Pang (1990) and in Wright (1990). Many other methods for generating and processing graphics primitives are discussed in Foley, et al. (1990), Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994), and Paeth (1995).

Soft-fill techniques are given in Fishkin and Barsky (1984). Antialiasing techniques are discussed in Pitteway and Watinson (1980), Crow (1981), Turkowski (1982), Fujimoto and Iwata (1983), Korein and Badler (1983), Kirk and Arvo (1991), and Wu (1991). Gray-scale applications are explored in Crow (1978). Other discussions of attributes and state parameters are available in Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994), and Paeth (1995).

EXERCISES

1 Implement a polyline function using the DDA algorithm, given any number (n) of input points. A single point is to be plotted when $n = 1$.

2 Extend Bresenham's line algorithm to generate lines with any slope, taking symmetry between quadrants into account.

3 Implement a polyline function, using the algorithm from the previous exercise, to display the set of straight lines connecting a list of n input points. For $n = 1$, the routine displays a single point.

4 Use the midpoint method to derive decision parameters for generating points along a straight-line path with slope in the range $0 < m < 1$. Show that the midpoint decision parameters are the same as those in the Bresenham line algorithm.

5 Use the midpoint method to derive decision parameters that can be used to generate straight-line segments with any slope.

6 Set up a parallel version of Bresenham's line algorithm for slopes in the range $0 < m < 1$.

7 Set up a parallel version of Bresenham's algorithm for straight lines with any slope.

8 Suppose you have a system with an 8 inch by 10 inch video monitor that can display 100 pixels per inch. If memory is organized in one byte words, the starting frame buffer address is 0, and each pixel is assigned one byte of storage, what is the frame buffer address of the pixel with screen coordinates (x, y)?

9 Suppose you have a system with a 12 inch by 14 inch video monitor that can display 120 pixels per inch. If memory is organized in one byte words, the starting frame buffer address is 0, and each pixel is assigned one byte of storage, what is the frame buffer address of the pixel with screen coordinates (x, y)?

10 Suppose you have a system with a 12 inch by 14 inch video monitor that can display 120 pixels per inch. If memory is organized in one byte words, the starting frame buffer address is 0, and each pixel is assigned 4 bits of storage, what is the frame buffer address of the pixel with screen coordinates (x, y)?

11 Incorporate the iterative techniques for calculating frame-buffer addresses (Section 3) into the Bresenham line algorithm.

12 Revise the midpoint circle algorithm to display circles with input geometric magnitudes preserved (Section 8).

13 Set up a procedure for a parallel implementation of the midpoint circle algorithm.

14 Derive decision parameters for the midpoint ellipse algorithm assuming the start position is $(r_x, 0)$ and points are to be generated along the curve path in counterclockwise order.

15 Set up a procedure for a parallel implementation of the midpoint ellipse algorithm.

16 Devise an efficient algorithm that takes advantage of symmetry properties to display a sine function over one cycle.

17 Modify the algorithm in the preceding exercise to display a sine curve over any specified angular interval.

18 Devise an efficient algorithm, taking function symmetry into account, to display a plot of damped harmonic motion:

$$y = Ae^{-kx}\sin(\omega x + \theta)$$

where ω is the angular frequency and θ is the phase of the sine function. Plot y as a function of x for several cycles of the sine function or until the maximum amplitude is reduced to $\frac{A}{10}$.

19 Use the algorithm developed in the previous exercise to write a program that displays one cycle of a sine curve. The curve should begin at the left edge of the display window and complete at the right edge, and the amplitude should be scaled so that the maximum and minimum values of the curve are equal to the maximum and minimum y values of the display window.

20 Using the midpoint method, and taking symmetry into account, develop an efficient algorithm for scan conversion of the following curve over the interval $-10 \le x \le 10$.

$$y = \frac{1}{12}x^3$$

21 Use the algorithm developed in the previous exercise to write a program that displays a portion of a sine curve determined by an input angular interval. The curve should begin at the left edge of the display window and complete at the right edge, and the amplitude should be scaled so that the maximum and minimum values of the curve are equal to the maximum and minimum y values of the display window.

22 Use the midpoint method and symmetry considerations to scan convert the parabola

$$x = y^2 - 5$$

over the interval $-10 \le x \le 10$.

23 Use the midpoint method and symmetry considerations to scan convert the parabola

$$y = 50 - x^2$$

over the interval $-5 \le x \le 5$.

24 Set up a midpoint algorithm, taking symmetry considerations into account to scan convert any parabola of the form

$$y = ax^2 + b$$

with input values for parameters a, b, and the range for x.

25 Define an efficient polygon-mesh representation for a cylinder and justify your choice of representation.

26 Implement a general line-style function by modifying Bresenham's line-drawing algorithm to display solid, dashed, or dotted lines.

27 Implement a line-style function using a midpoint line algorithm to display solid, dashed, or dotted lines.

28 Devise a parallel method for implementing a line-style function.

29 Devise a parallel method for implementing a line-width function.

30 A line specified by two endpoints and a width can be converted to a rectangular polygon with four vertices and then displayed using a scan-line method. Develop an efficient algorithm for computing the four vertices needed to define such a rectangle, with the line endpoints and line width as input parameters.

31 Implement a line-width function in a line-drawing program so that any one of three line widths can be displayed.

32 Write a program to output a line graph of three data sets defined over the same x-coordinate range. Input to the program is to include the three sets of data values and the labels for the graph. The data sets are to be scaled to fit within a defined coordinate range for a display window. Each data set is to be plotted with a different line style.

33 Modify the program in the previous exercise to plot the three data sets in different colors, as well as different line styles.

34 Set up an algorithm for displaying thick lines with butt caps, round caps, or projecting square caps. These options can be provided in an option menu.

35 Devise an algorithm for displaying thick polylines with a miter join, a round join, or a bevel join. These options can be provided in an option menu.

36 Implement pen and brush menu options for a line-drawing procedure, including at least two options: round and square shapes.

37 Modify a line-drawing algorithm so that the intensity of the output line is set according to its slope. That is, by adjusting pixel intensities according to the value of the slope, all lines are displayed with the same intensity per unit length.

38 Define and implement a function for controlling the line style (solid, dashed, dotted) of displayed ellipses.

39 Define and implement a function for setting the width of displayed ellipses.

40 Modify the scan-line algorithm to apply any specified rectangular fill pattern to a polygon interior, starting from a designated pattern position.

41 Write a program to scan convert the interior of a specified ellipse into a solid color.

42 Write a procedure to fill the interior of a given ellipse with a specified pattern.

43 Write a procedure for filling the interior of any specified set of fill-area vertices, including one with crossing edges, using the nonzero winding number rule to identify interior regions.

44 Modify the boundary-fill algorithm for a 4-connected region to avoid excessive stacking by incorporating scan-line methods.

45 Write a boundary-fill procedure to fill an 8-connected region.

46 Explain how an ellipse displayed with the midpoint method could be properly filled with a boundary-fill algorithm.

47 Develop and implement a flood-fill algorithm to fill the interior of any specified area.

48 Define and implement a procedure for changing the size of an existing rectangular fill pattern.

49 Write a procedure to implement a soft-fill algorithm. Carefully define what the soft-fill algorithm is to accomplish and how colors are to be combined.

50 Devise an algorithm for adjusting the height and width of characters defined as rectangular grid patterns.

51 Implement routines for setting the character up vector and the text path for controlling the display of character strings.

52 Write a program to align text as specified by input values for the alignment parameters.

53 Develop procedures for implementing marker attributes (size and color).

54 Implement an antialiasing procedure by extending Bresenham's line algorithm to adjust pixel intensities in the vicinity of a line path.

55 Implement an antialiasing procedure for the midpoint line algorithm.

56 Develop an algorithm for antialiasing elliptical boundaries.

57 Modify the scan-line algorithm for area fill to incorporate antialiasing. Use coherence techniques to reduce calculations on successive scan lines.

58 Write a program to implement the Pitteway-Watkinson antialiasing algorithm as a scan-line procedure to fill a polygon interior, using the OpenGL point-plotting function.

IN MORE DEPTH

1 Write routines to implement Bresenham's line-drawing algorithm and the DDA line-drawing algorithm and use them to draw the outlines of the shapes in the current snapshot of your application. Record the runtimes of the algorithms and compare the performance of the two. Next examine the polygons that represent the objects in your scene and either choose a few that would be better represented using ellipses or other curves or add a few objects with this property. Implement a midpoint algorithm to draw the ellipses or curves that represent these objects and use it to draw the outlines of those objects. Discuss ways in which you could improve the performance of the algorithms you developed if you had direct access to parallel hardware.

2 Implement the general scan-line polygon-fill algorithm to fill in the polygons that make up the objects in your scene with solid colors. Next, implement a scan-line curve-filling algorithm to fill the curved objects you added in the previous exercise. Finally, implement a boundary fill algorithm to fill all of the objects in your scene. Compare the run times of the two approaches to filling in the shapes in your scene.

This page intentionally left blank