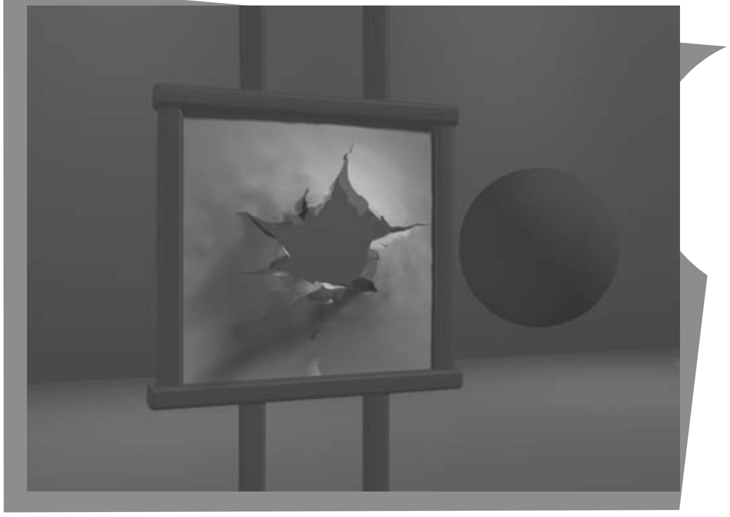
Table of Contents

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

From Chapter 4 of *Computer Graphics with OpenGL®*, Fourth Edition, Donald Hearn, M. Pauline Baker, Warren R. Carithers.

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.

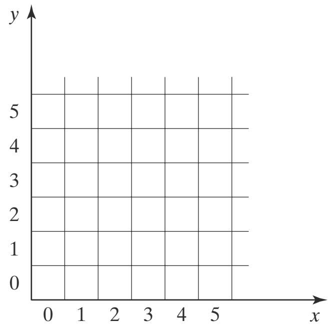


FIGURE 1
Pixel positions referenced with respect to the lower-left corner of a screen area.

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.

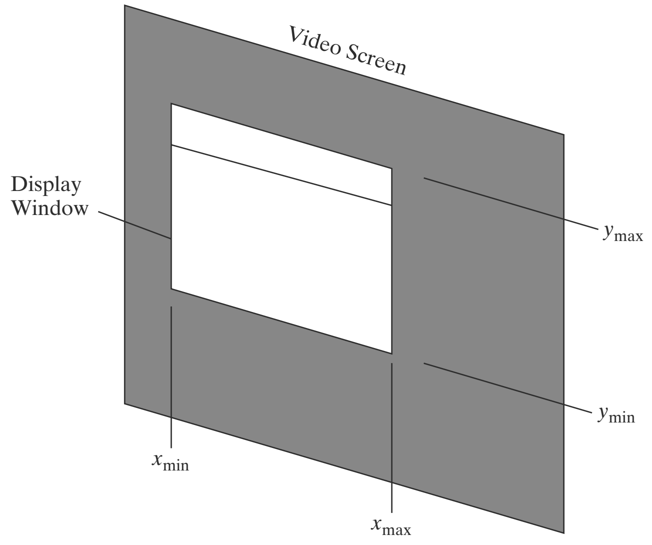


FIGURE 2
World-coordinate limits for a display window, as specified in the `glOrtho2D` function.

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.

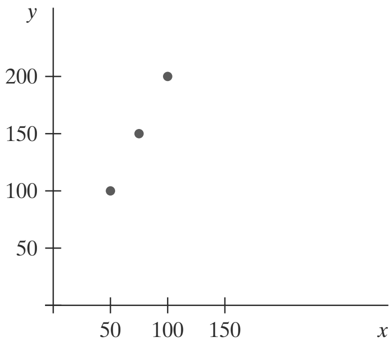


FIGURE 3
Display of three point positions generated with `glBegin (GL_POINTS)`.

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.

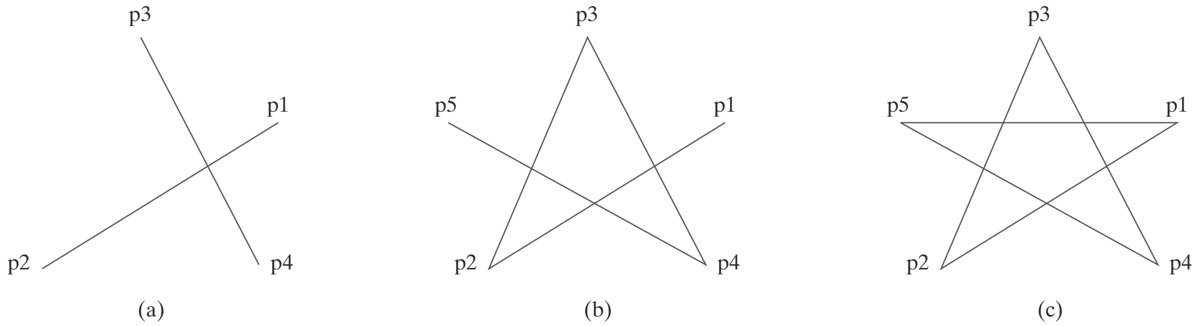


FIGURE 4
Line segments that can be displayed in OpenGL using a list of five endpoint coordinates. (a) An unconnected set of lines generated with the primitive line constant `GL_LINES`. (b) A polyline generated with `GL_LINE_STRIP`. (c) A closed polyline generated with `GL_LINE_LOOP`.

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

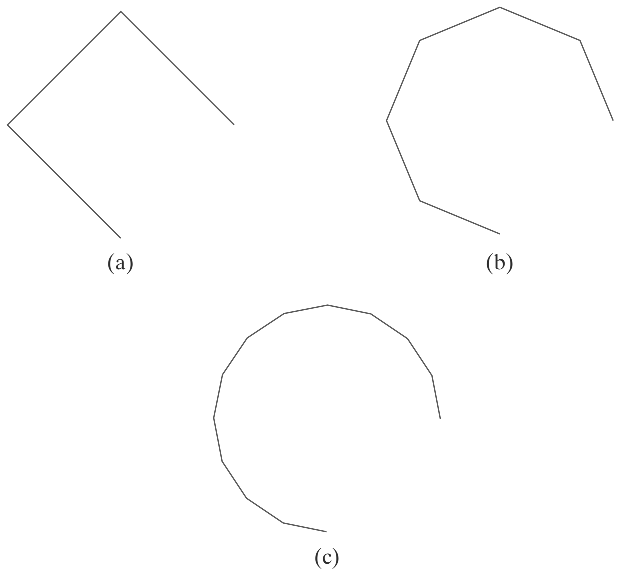


FIGURE 5
A circular arc approximated with (a) three straight-line segments, (b) six line segments, and (c) twelve line segments.

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

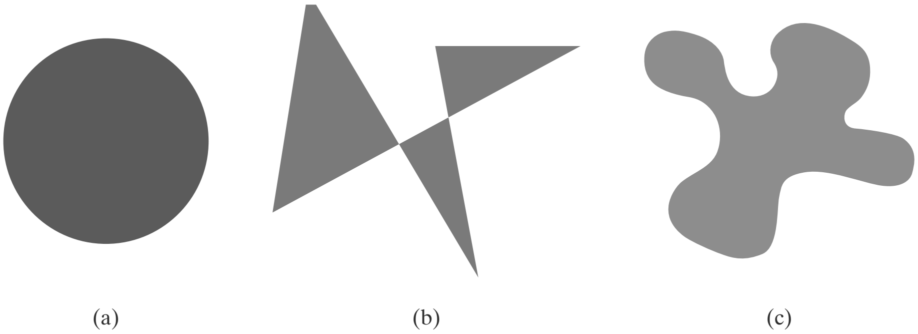


FIGURE 6
Solid-color fill areas specified with various boundaries. (a) A circular fill region. (b) A fill area bounded by a closed polyline. (c) A filled area specified with an irregular curved boundary.

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.**

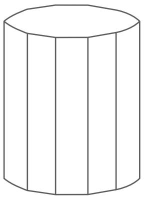


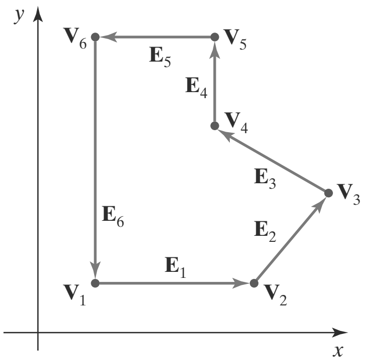
FIGURE 7
Wire-frame representation for a cylinder, showing only the front (visible) faces of the polygon mesh used to approximate the surfaces.

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.

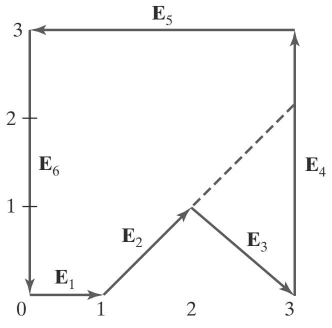


FIGURE 10
Splitting a concave polygon using the vector method.

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

$$\begin{array}{ll} \mathbf{E}_1 = (1, 0, 0) & \mathbf{E}_2 = (1, 1, 0) \\ \mathbf{E}_3 = (1, -1, 0) & \mathbf{E}_4 = (0, 2, 0) \\ \mathbf{E}_5 = (-3, 0, 0) & \mathbf{E}_6 = (0, -2, 0) \end{array}$$

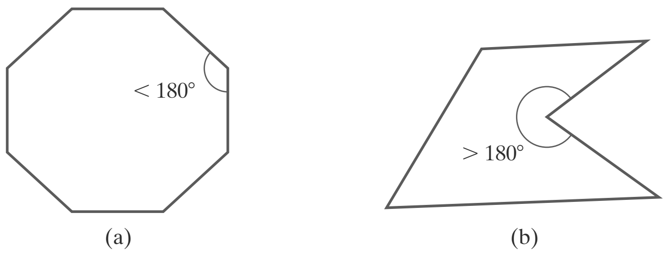


FIGURE 8
A convex polygon (a), and a concave polygon (b).

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 $E_{jx}E_{ky} - E_{kx}E_{jy}$:

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

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.

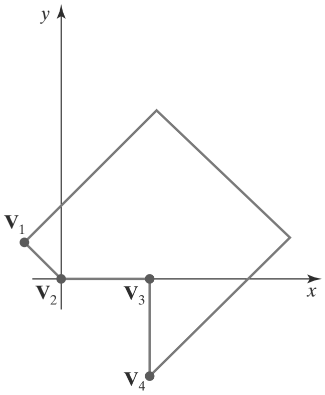


FIGURE 11
Splitting a concave polygon using the rotational method. After moving $\mathbf{V}_2$ to the coordinate origin and rotating $\mathbf{V}_3$ onto the x axis, we find that $\mathbf{V}_4$ is below the x axis. So we split the polygon along the line of $\overline{\mathbf{V}_2\mathbf{V}_3}$, which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

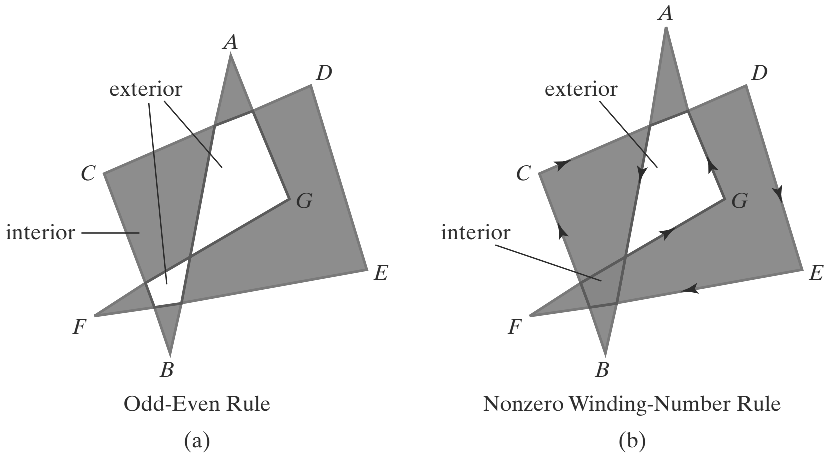


FIGURE 12
Identifying interior and exterior regions of a closed polyline that contains self-intersecting segments.

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.

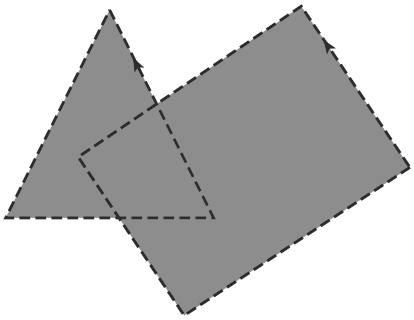


FIGURE 13
A fill area defined as a region that has a positive value for the winding number. This fill area is the union of two regions, each with a counterclockwise border direction.

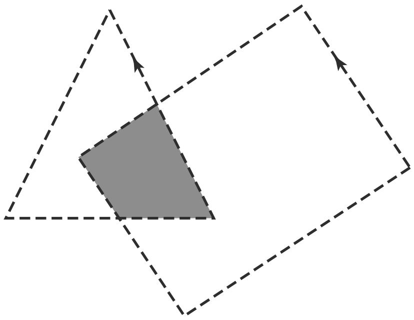


FIGURE 14
A fill area defined as a region with a winding number greater than 1. This fill area is the intersection of two regions, each with a counterclockwise border direction.

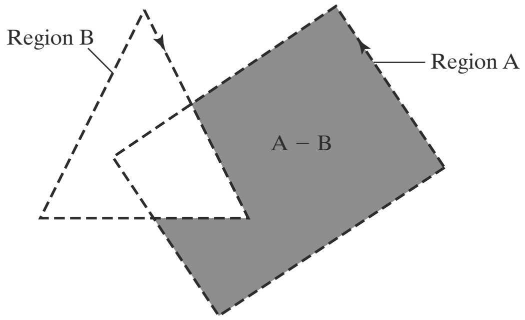
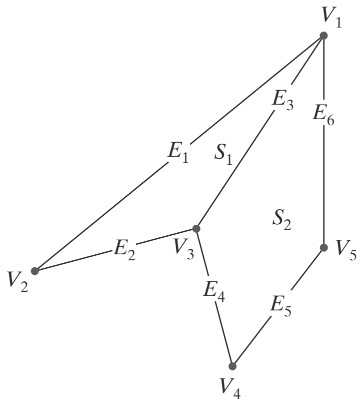


FIGURE 15
A fill area defined as a region with a positive value for the winding number. This fill area is the difference, A — B, of two regions, where region A has a positive border direction (counterclockwise) and region B has a negative border direction (clockwise).

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

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

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

if $\quad Ax + By + Cz + D < 0, \qquad$ the point (x, y, z) is behind the plane

if $\quad Ax + By + Cz + D > 0, \qquad$ 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.

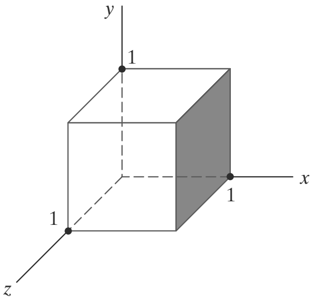


FIGURE 18
The shaded polygon surface of the unit cube has the plane equation $x - 1 = 0$.

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.

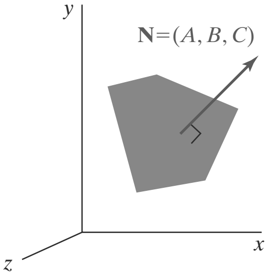


FIGURE 19
The normal vector **N** for a plane described with the equation $Ax + By + Cz + D = 0$ is perpendicular to the plane and has Cartesian components (A, B, C).

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 **N** and the position **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.

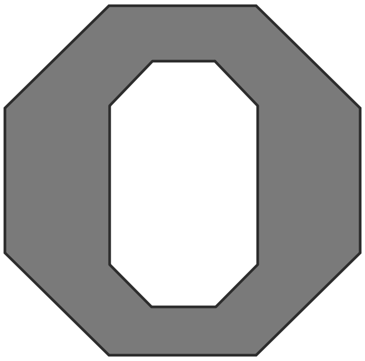


FIGURE 20
A polygon with a complex interior that cannot be specified with a single vertex list.

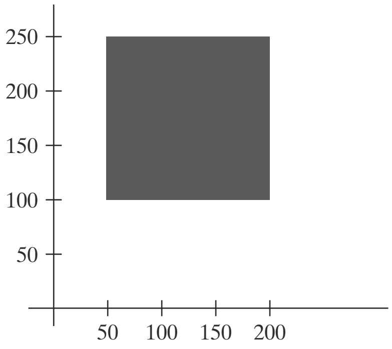


FIGURE 21
The display of a square fill area using the `glRect` function.

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

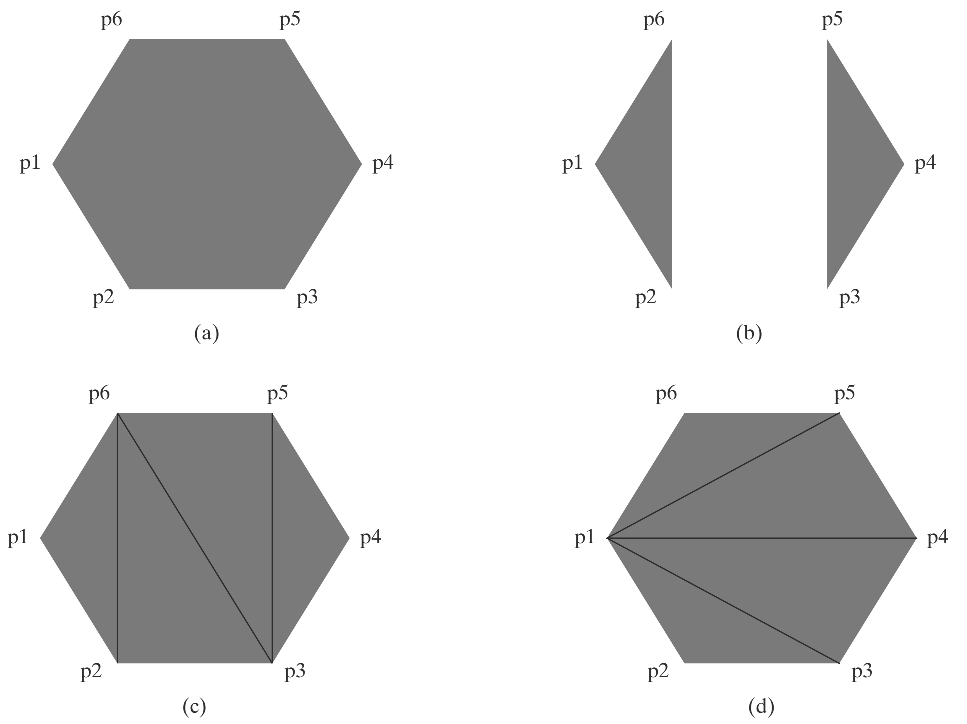


FIGURE 22
Displaying polygon fill areas using a list of six vertex positions. (a) A single convex polygon fill area generated with the primitive constant `GL_POLYGON`. (b) Two unconnected triangles generated with `GL_TRIANGLES`. (c) Four connected triangles generated with `GL_TRIANGLE_STRIP`. (d) Four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

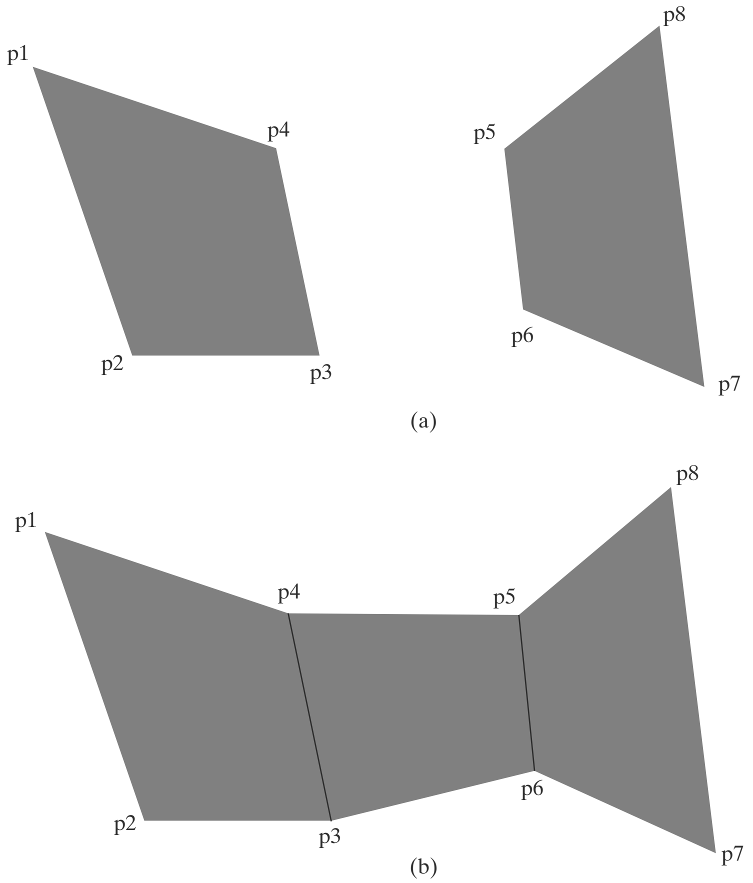


FIGURE 23
Displaying quadrilateral fill areas using a list of eight vertex positions. (a) Two unconnected quadrilaterals generated with `GL_QUADS`. (b) Three connected quadrilaterals generated with `GL_QUAD_STRIP`.

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 $n = 1, n = 2, \ldots, n = \frac{N}{2} - 1$. 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.

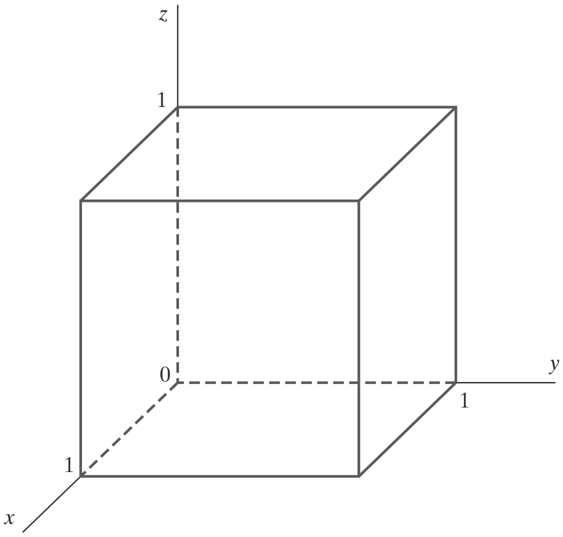


FIGURE 24
A cube with an edge length of 1.

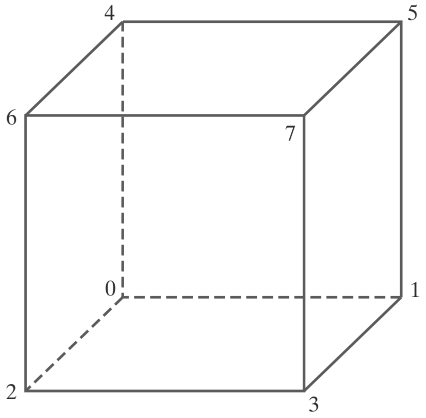


FIGURE 25
Subscript values for array `pt` corresponding to the vertex coordinates for the cube shown in Figure 24.

```
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*.

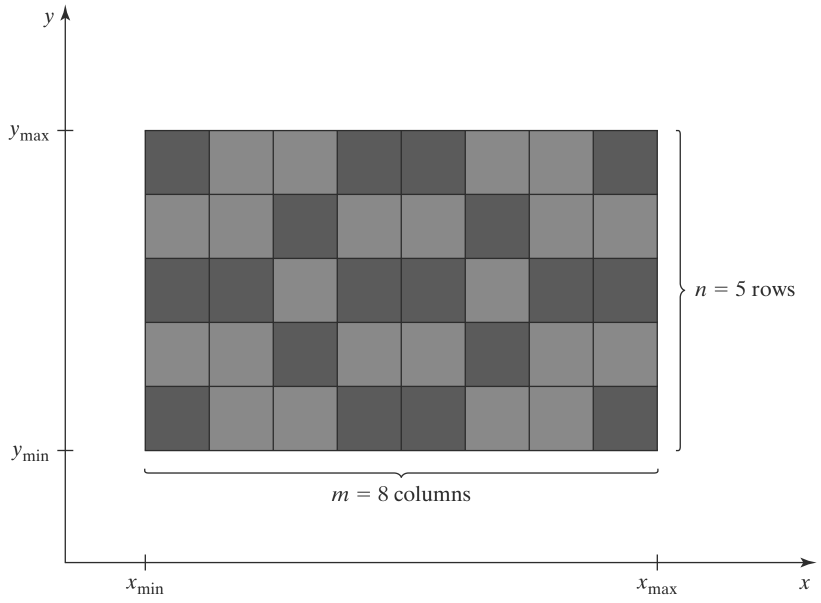


FIGURE 26
Mapping an *n* by *m* color array onto a region of the screen coordinates.

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.

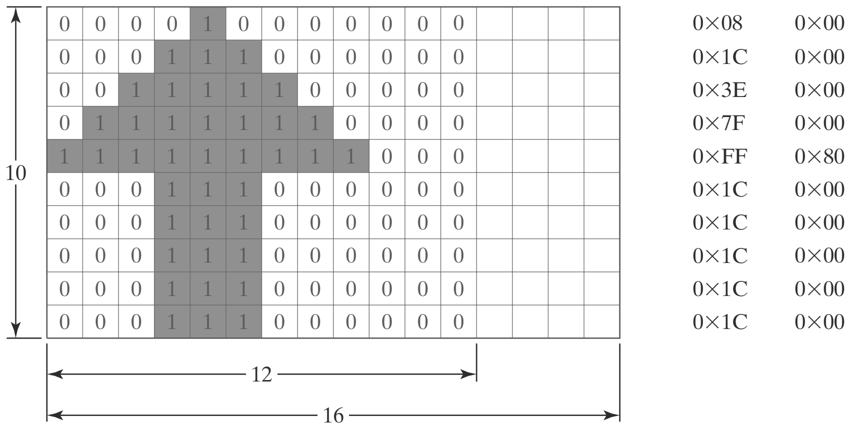
Each row of a rectangular bit array is stored in multiples of 8 bits, where the binary data is arranged as a set of 8-bit unsigned characters. But we can describe a shape using any convenient grid size. For example, Figure 27 shows a bit pattern defined on a 10-row by 9-column grid, where the binary 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);
```



0	0	0	0	1	0	0	0	0	0	0	0					0×08	0×00
0	0	0	1	1	1	0	0	0	0	0	0					0×1C	0×00
0	0	1	1	1	1	1	0	0	0	0	0					0×3E	0×00
0	1	1	1	1	1	1	1	0	0	0	0					0×7F	0×00
1	1	1	1	1	1	1	1	1	0	0	0					0×FF	0×80
0	0	0	1	1	1	0	0	0	0	0	0					0×1C	0×00
0	0	0	1	1	1	0	0	0	0	0	0					0×1C	0×00
0	0	0	1	1	1	0	0	0	0	0	0					0×1C	0×00
0	0	0	1	1	1	0	0	0	0	0	0					0×1C	0×00
0	0	0	1	1	1	0	0	0	0	0	0					0×1C	0×00

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.

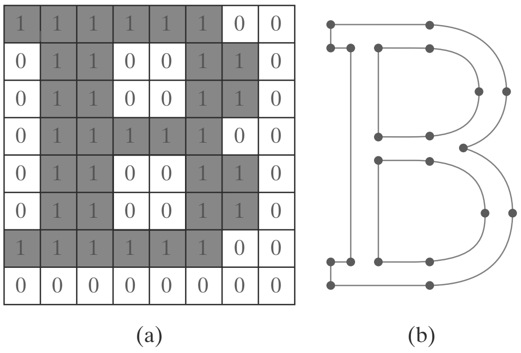
Fonts can be divided into two broad groups: *serif* and *sans serif.* Serif type has small lines or accents at the ends of the main character strokes, while sans-serif type does not have such accents. For example, this text is set in a serif font (Palatino). But this sentence is printed in a sans-serif 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.

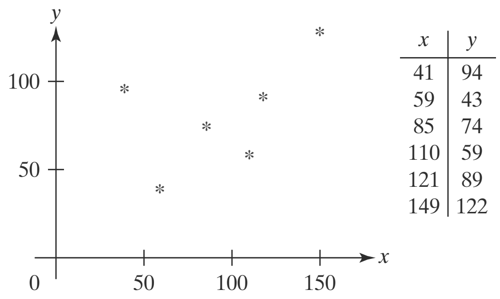
Bitmap fonts are the simplest to define and display: We just need to map the character grids to a frame-buffer position. In general, however, bitmap fonts require more storage space because each variation (size and format) must be saved in a *font cache.* It is possible to generate different sizes and 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,



1	1	1	1	1	1	0	0
0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0
0	1	1	1	1	1	0	0
0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0
1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0

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

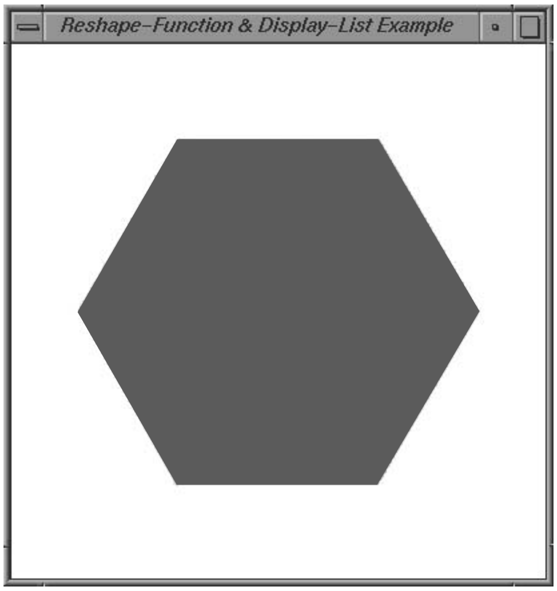


FIGURE 30
The display window generated by the example program illustrating the use of the reshape function.

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.

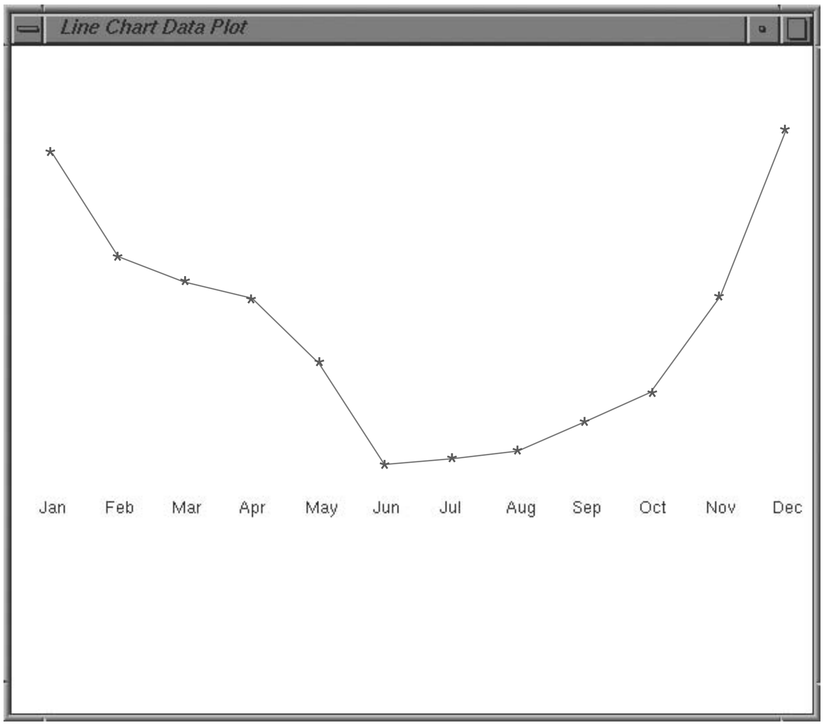


FIGURE 31
A polyline and polymarker plot of data points output by the `lineGraph` routine.

```
#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.

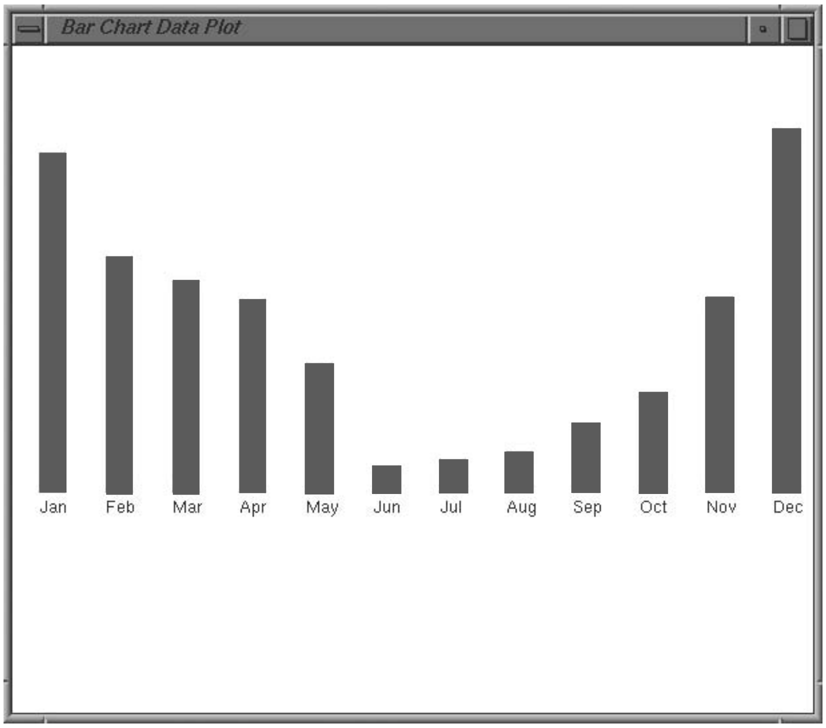


FIGURE 32
A bar chart generated by the `barChart` procedure.

```
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 ( );
}
```

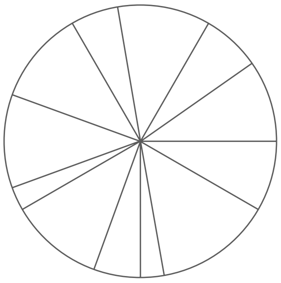


FIGURE 33
Output produced with the `pieChart` procedure.

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.

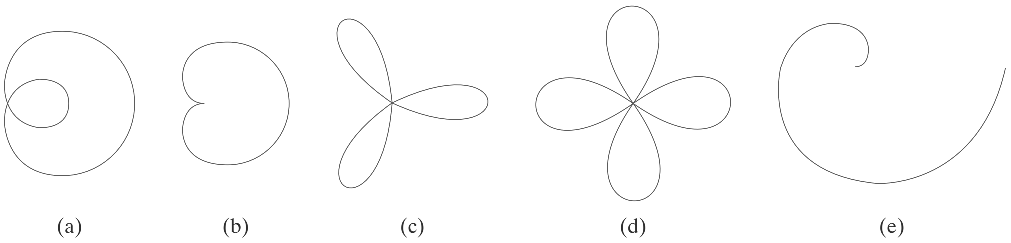


FIGURE 34
Curved figures displayed by the `drawCurve` procedure: (a) limaçon, (b) cardioid, (c) three-leaf curve, (d) four-leaf curve, and (e) spiral.

```
#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.

From Chapter 5 of *Computer Graphics with OpenGL®*, Fourth Edition, Donald Hearn, M. Pauline Baker, Warren R. Carithers.

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

Color Code	Stored Color Values in Frame Buffer			Displayed Color
	RED	GREEN	BLUE	
0	0	0	0	Black
1	0	0	1	Blue
2	0	1	0	Green
3	0	1	1	Cyan
4	1	0	0	Red
5	1	0	1	Magenta
6	1	1	0	Yellow
7	1	1	1	White

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

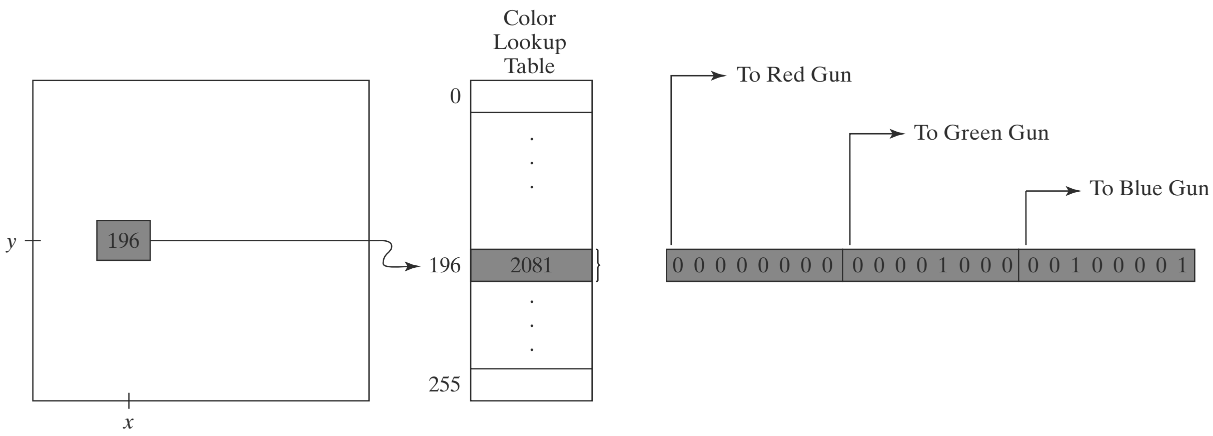


FIGURE 1
A color lookup table with 24 bits per entry that is accessed from a frame buffer with 8 bits per pixel. A value of 196 stored at pixel position (x, y) references the location in this table containing the hexadecimal value 0x0821 (a decimal value of 2081). Each 8-bit segment of this entry controls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

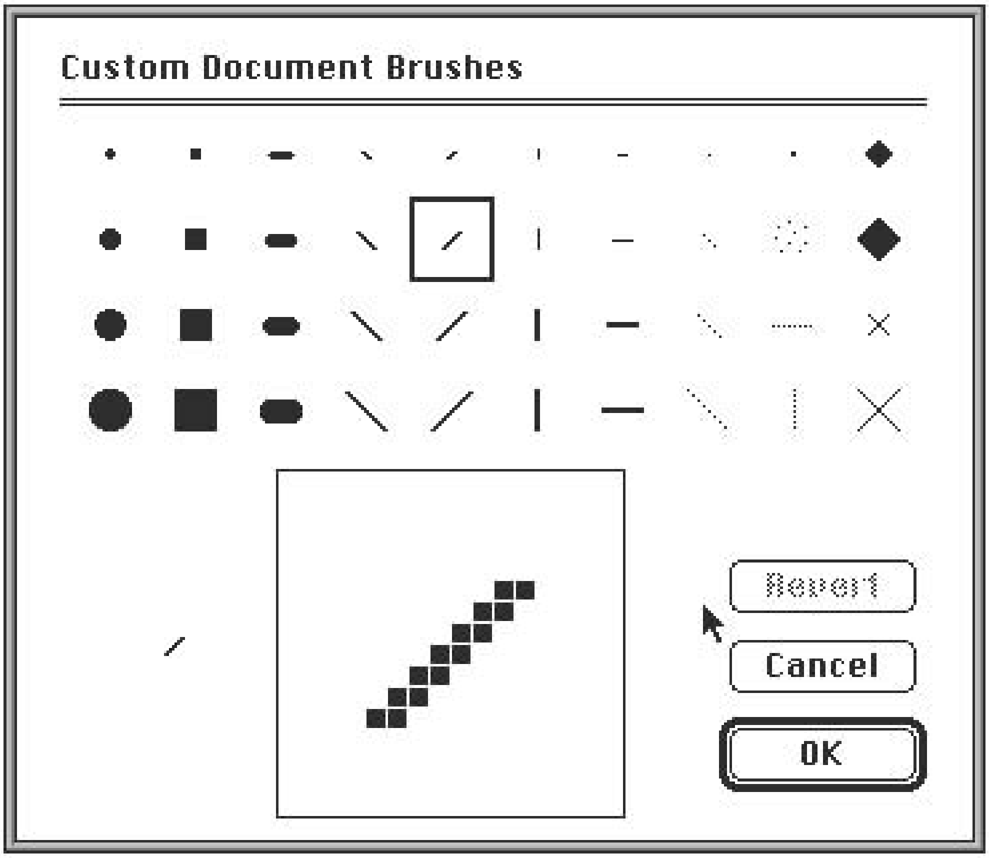


FIGURE 2
Pen and brush shapes for line display.

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);
```

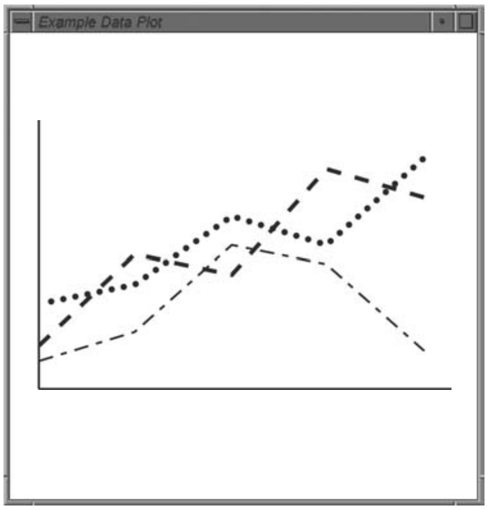


FIGURE 3
Plotting three data sets with three different OpenGL line styles and line widths: single-width dash-dot pattern, double-width dash pattern, and triple-width dot pattern.

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.

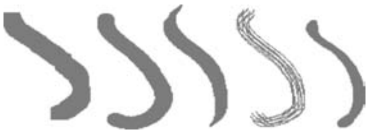


FIGURE 4
Curved lines drawn with a paint program using various shapes and patterns. From left to right, the brush shapes are square, round, diagonal line, dot pattern, and faded airbrush.

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.

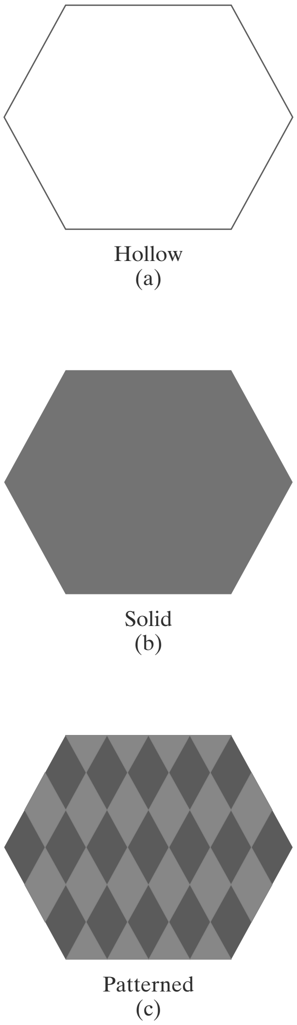


FIGURE 5
Basic polygon fill styles.

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.

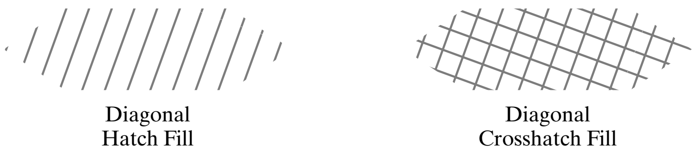


FIGURE 6
Areas filled with hatch patterns.

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

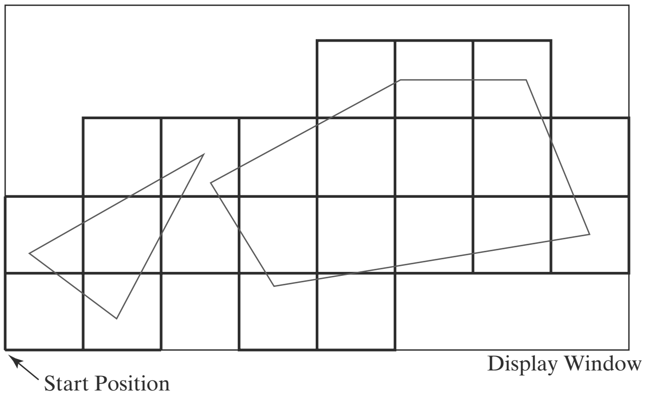


FIGURE 7
Tiling a rectangular fill pattern across a display window to fill two convex polygons.

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.

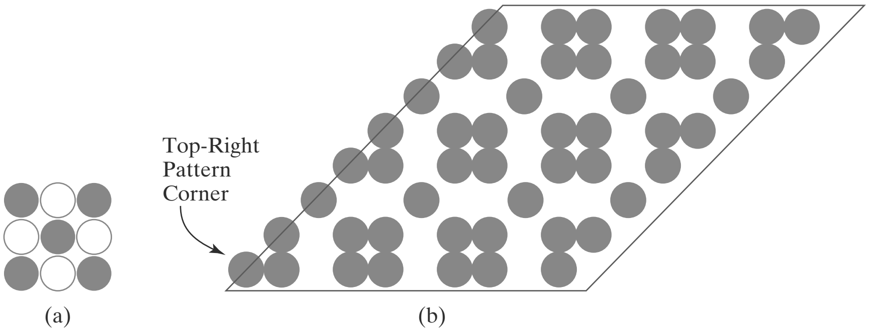


FIGURE 8
A **3** × **3** bit pattern (a) superimposed on a parallelogram to produce the fill area in (b), where the top-right corner of the pattern coincides with the lower-left corner of the parallelogram.

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.

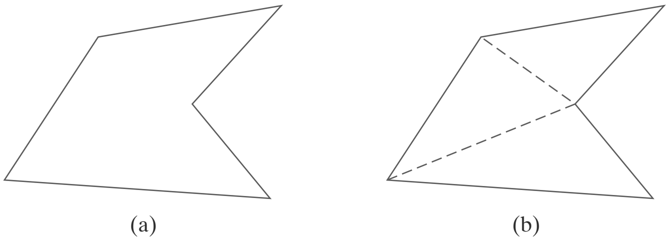


FIGURE 9
Dividing a concave polygon (a) into a set of triangles (b) produces triangle edges (dashed) that are interior to the original polygon.

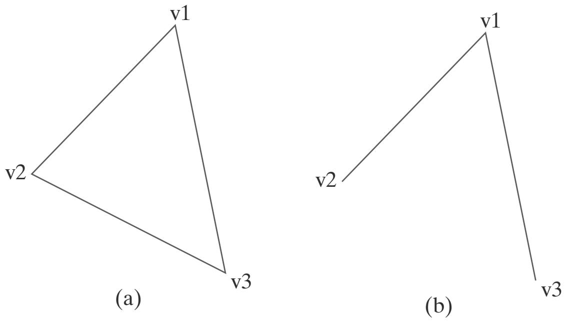


FIGURE 10
The triangle in (a) can be displayed as in (b) by setting the edge flag for vertex v2 to the value GL_FALSE, assuming that the vertices are specified in a counterclockwise order.

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

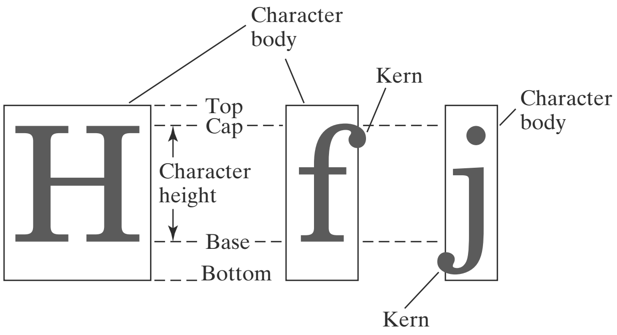


FIGURE 11
Examples of character bodies.

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

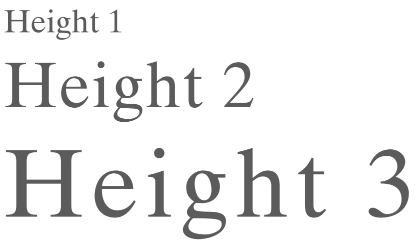


FIGURE 12
Text strings displayed with different character-height settings and a constant width-to-height ratio.

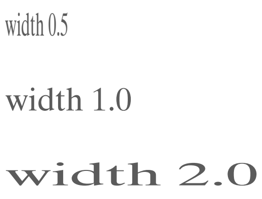


FIGURE 13
Text strings displayed with varying sizes for the character widths and a fixed height.

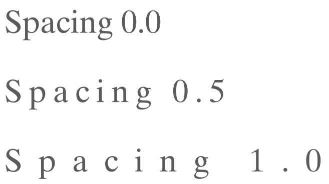


FIGURE 14
Text strings displayed with different character-spacing values.

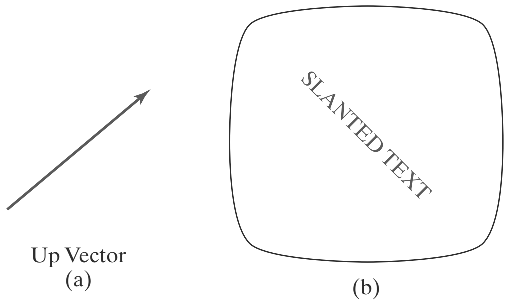


FIGURE 15
Direction of the up vector (a) controls the orientation of displayed text (b).

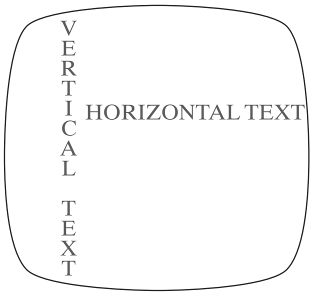


FIGURE 16
Text-path attributes can be set to produce horizontal or vertical arrangements of character strings.

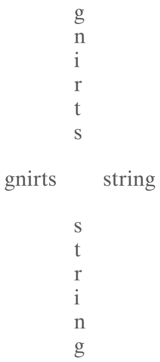


FIGURE 17
A text string displayed with the four text-path options: left, right, up, and down.

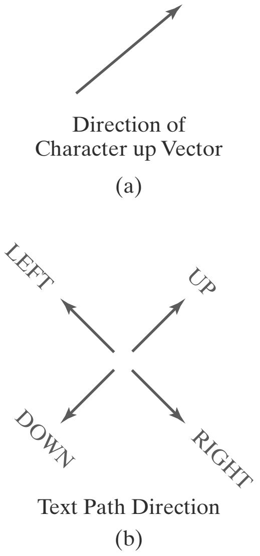


FIGURE 18
An up-vector specification (a) and associated directions for the text path (b).

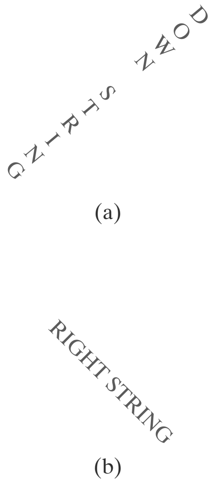


FIGURE 19
The 45° up vector in Figure 18 produces the display (a) for a *down* path and the display (b) for a *right* path.

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.

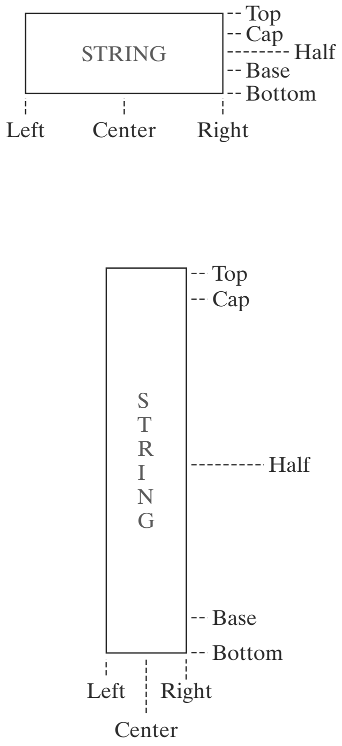


FIGURE 20
Character alignments for horizontal and vertical strings.

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.

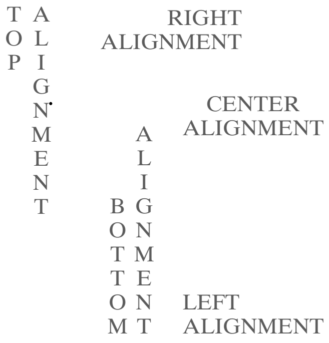


FIGURE 21
Character-string alignments.

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.

From Chapter 6 of *Computer Graphics with OpenGL®*, Fourth Edition, Donald Hearn, M. Pauline Baker, Warren R. Carithers.

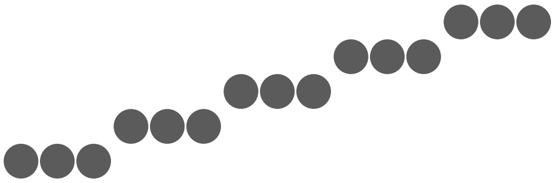


FIGURE 1
Stair-step effect (jaggies) produced when a line is generated as a series of pixel positions.

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).

Line Equations

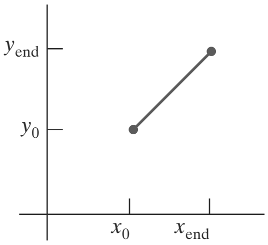


FIGURE 2
Line path between endpoint positions (x_0, y_0) and $(x_{\text{end}}, y_{\text{end}})$.

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 $(x_{\text{end}}, y_{\text{end}})$, as shown in Figure 2, we can determine values for the slope m and y intercept b with the following calculations:

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

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

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.

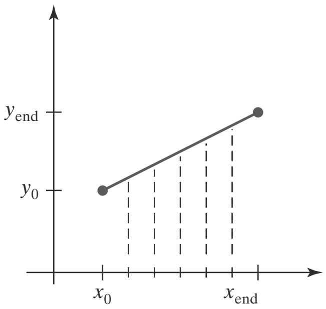


FIGURE 3
Straight-line segment with five sampling positions along the x axis between x_0 and x_{end}.

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));
   }
}
```

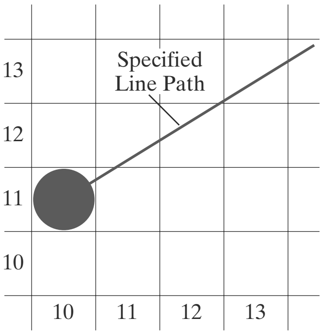


FIGURE 4

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.

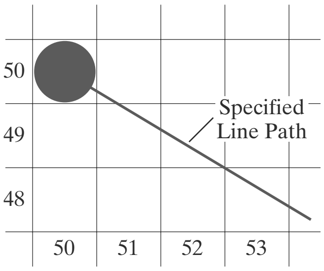


FIGURE 5

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.

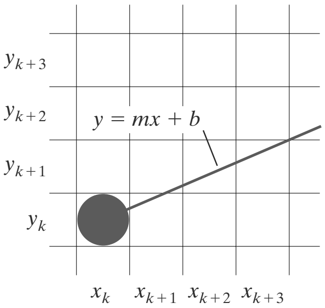


FIGURE 6
A section of the screen showing a pixel in column x_k on scan line y_k that is to be plotted along the path of a line segment with slope $0 < m < 1$.

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 $(x_k + 1, y_k + 1)$.

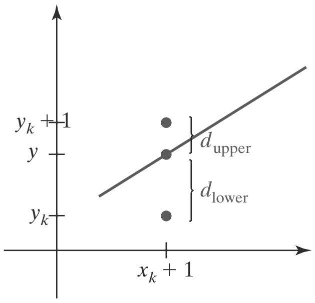


FIGURE 7
Vertical distances between pixel positions and the line y coordinate at sampling position $x_k + 1$.

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

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

Then

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

and

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

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

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

and the increments for calculating successive decision parameters are

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

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)

k	p_k	(x_{k+1}, y_{k+1})
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.

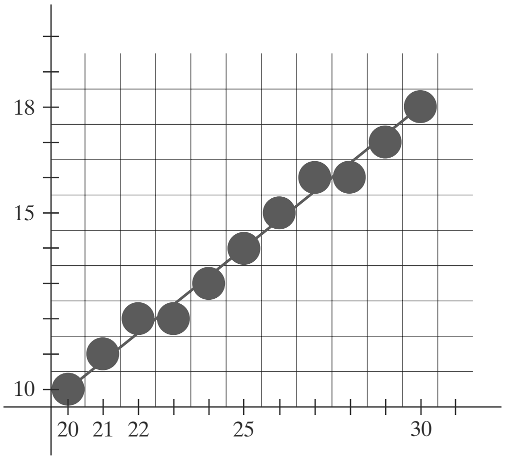


FIGURE 8
Pixel positions along the line path between endpoints (20, 10) and (30, 18), plotted with Bresenham's line algorithm.

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

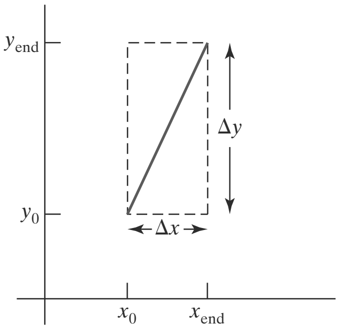


FIGURE 9
Bounding box for a line with endpoint separations Δx and Δy.

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

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

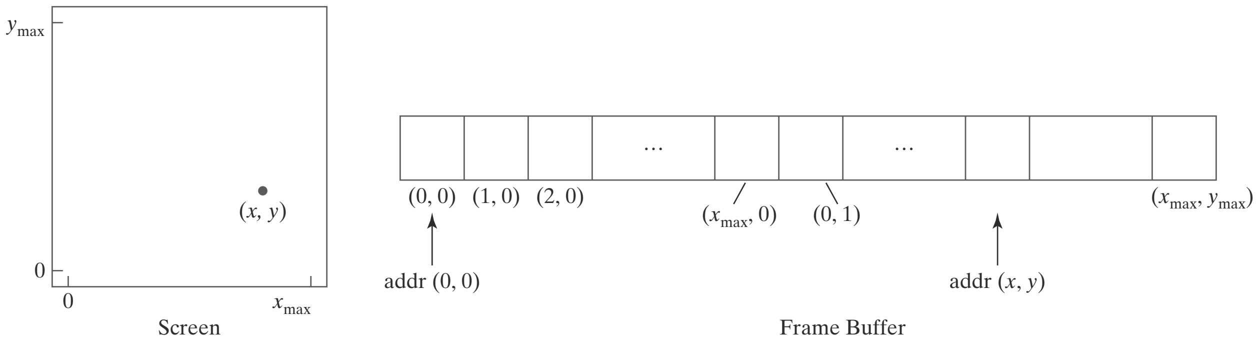


FIGURE 10
Pixel screen positions stored linearly in row-major order within the frame buffer.

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

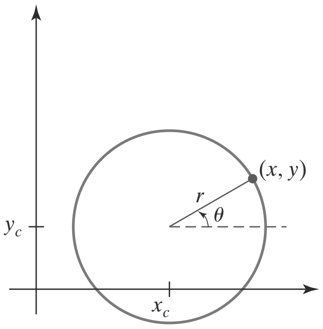


FIGURE 11
Circle with center coordinates (x_c, y_c) and radius r.

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

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

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

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

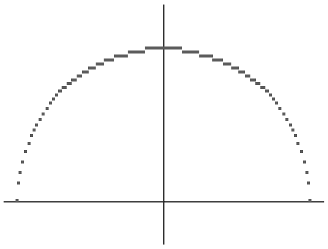


FIGURE 12
Upper half of a circle plotted with Equation 27 and with $(x_c, y_c) = (0, 0)$.

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

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

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

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

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.

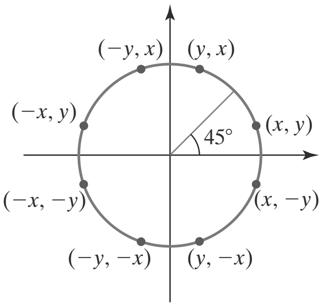


FIGURE 13
Symmetry of a circle. Calculation of a circle point (x, y) in one octant yields the circle points shown for the other seven octants.

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:

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

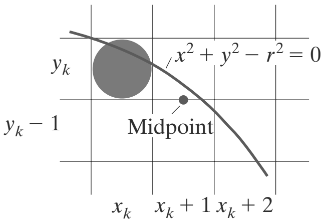


FIGURE 14
Midpoint between candidate pixels at sampling position $x_k + 1$ along a circular path.

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

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

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

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

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

or

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

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.

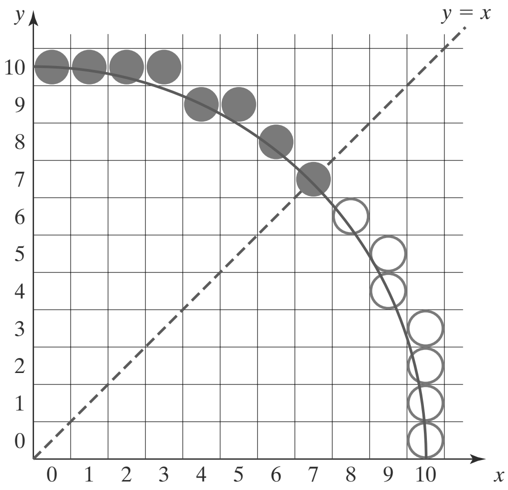


FIGURE 15
Pixel positions (solid circles) along a circle path centered on the origin and with radius $r = 10$, as calculated by the midpoint circle algorithm. Open ("hollow") circles show the symmetry positions in the first quadrant.

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.

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

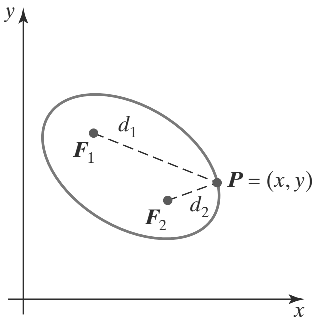


FIGURE 16
Ellipse generated about foci $\mathbf{F}_1$ and $\mathbf{F}_2$.

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

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

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

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.

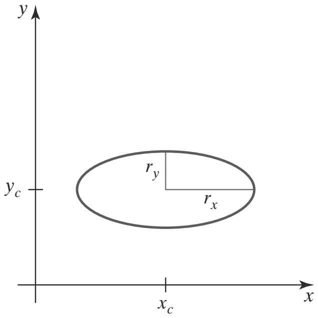


FIGURE 17
Ellipse centered at (x_c, y_c) with semimajor axis r_x and semiminor axis r_y.

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

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

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

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

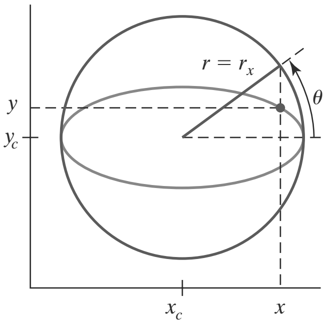


FIGURE 18
The bounding circle and eccentric angle θ for an ellipse with $r_x > r_y$.

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).

Midpoint Ellipse Algorithm

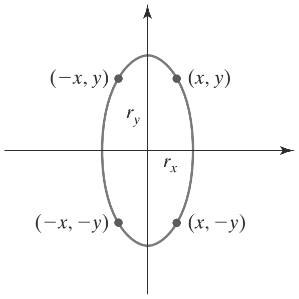


FIGURE 19
Symmetry of an ellipse. Calculation of a point (x, y) in one quadrant yields the ellipse points shown for the other three quadrants.

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

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

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

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

or

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

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

Decision parameters are incremented by the following amounts:

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

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

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

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

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

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

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

or

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

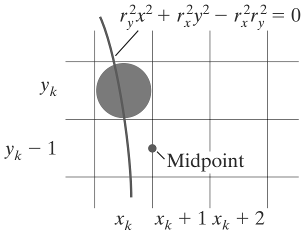


FIGURE 22
Midpoint between candidate pixels at sampling position $y_k - 1$ along an elliptical path.

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

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

If $p2_k > 0$, the 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.

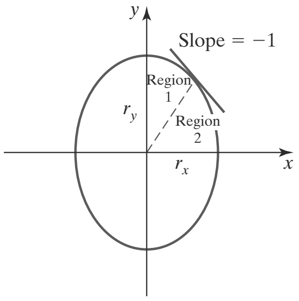


FIGURE 20
Ellipse processing regions. Over region 1, the magnitude of the ellipse slope is less than 1.0; over region 2, the magnitude of the slope is greater than 1.0.

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

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

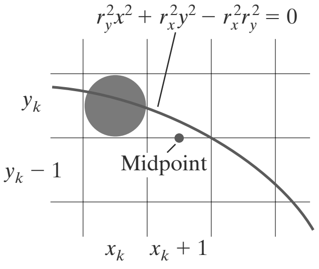


FIGURE 21
Midpoint between candidate pixels at sampling position $x_k + 1$ along an elliptical path.

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 $y_k - 1$.

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

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

or

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

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

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

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

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

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.

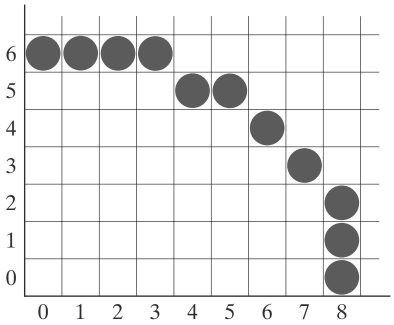


FIGURE 23
Pixel positions along an elliptical path centered on the origin with $r_x = 8$ and $r_y = 6$, using the midpoint algorithm to calculate locations within the first quadrant.

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 framebuffer 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.

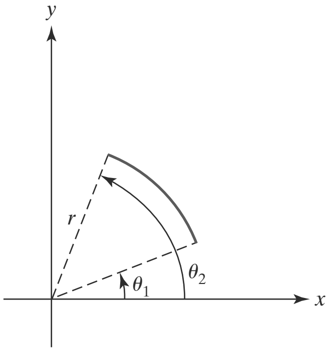


FIGURE 24
A circular arc, centered on the origin, defined with beginning angle θ_1, ending angle θ_2, and radius r.

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.

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

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

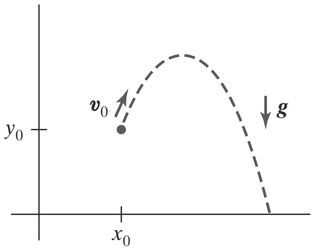


FIGURE 25
Parabolic path of an object tossed into a downward gravitational field at the initial position (x_0, y_0).

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**.

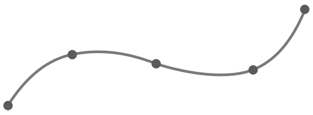


FIGURE 27
A spline curve formed with individual cubic polynomial sections between specified coordinate positions.

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:

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

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

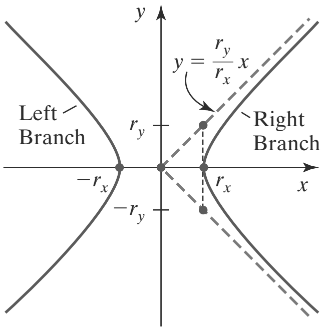


FIGURE 26
Left and right branches of a hyperbola in standard position with the symmetry axis along the x axis.

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

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.

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.

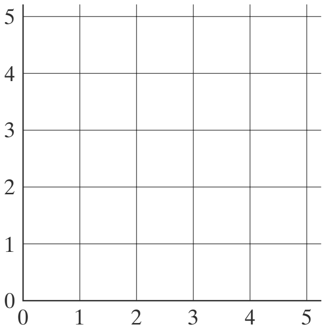


FIGURE 28
Lower-left section of a screen area with coordinate positions referenced by grid intersection lines.

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.

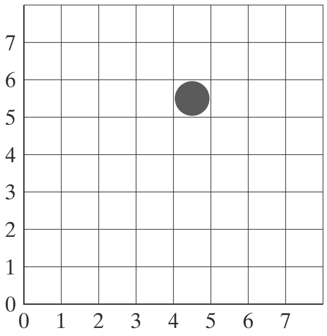


FIGURE 29
Illuminated pixel at raster position (4, 5).

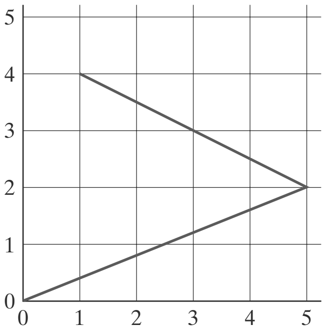


FIGURE 30
Line path for two connected line segments between screen grid-coordinate positions.

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

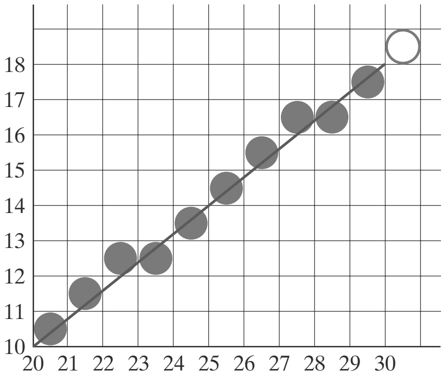
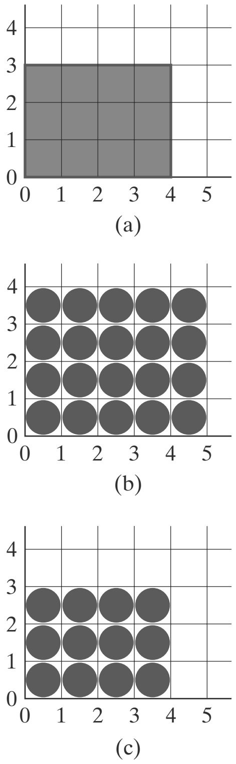


FIGURE 31
Line path and corresponding pixel display for grid endpoint coordinates (20, 10) and (30, 18).

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



(a)

(b)

(c)

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.

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 $y = 2$.

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

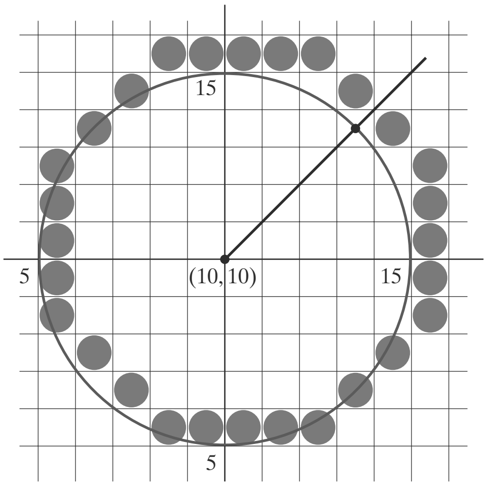


FIGURE 33
A midpoint-algorithm plot of the circle equation $(x - 10)^2 + (y - 10)^2 = 5^2$ using pixel-center coordinates.

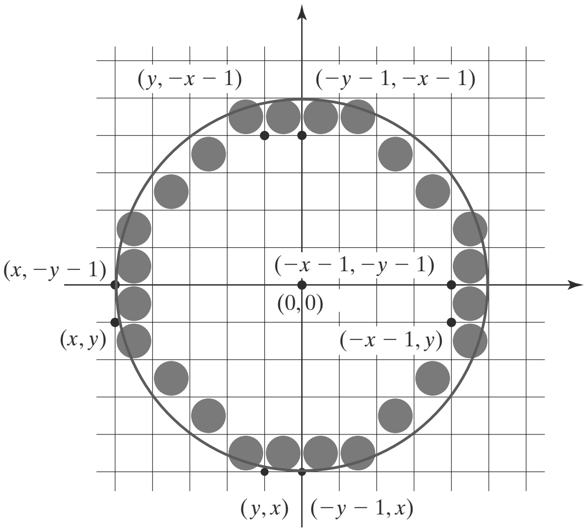


FIGURE 34
Modification of the circle plot in Figure 33 to maintain the specified circle diameter of 10.

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.

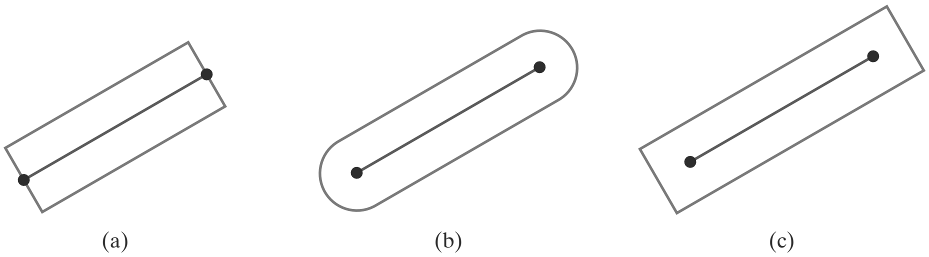


FIGURE 37
Thick lines drawn with (a) butt caps, (b) round caps, and (c) projecting square caps.

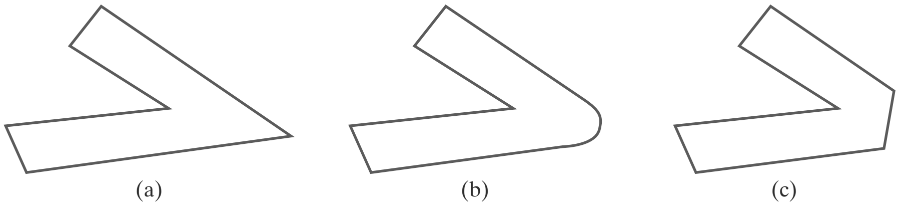


FIGURE 38
Thick line segments connected with a miter join (a), a round join (b), and a bevel join (c).

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

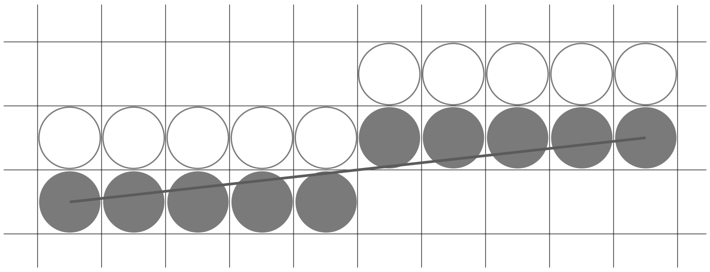


FIGURE 35
A double-wide raster line with slope $|m| < 1.0$ generated with vertical pixel spans.

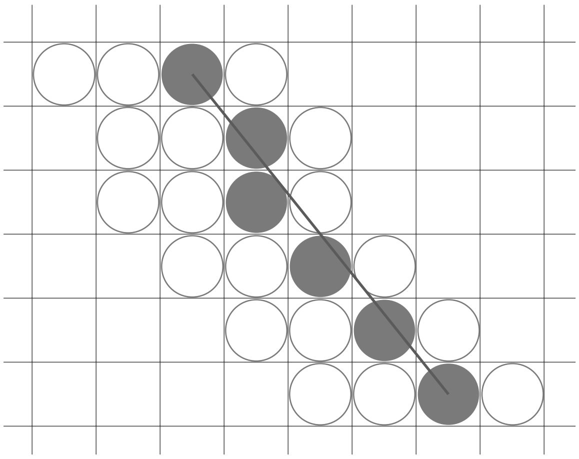


FIGURE 36
A raster line with slope $|m| > 1.0$ and a line width of 4 plotted using horizontal pixel spans.

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.

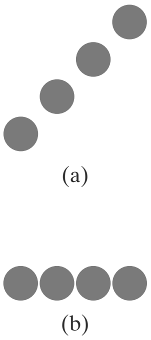


FIGURE 39
Unequal-length dashes displayed with the same number of pixels.

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.

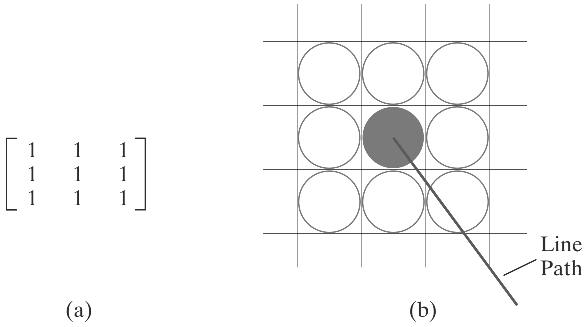


FIGURE 40
A pixel mask (a) for a rectangular pen, and the associated array of pixels (b) displayed by centering the mask over a specified pixel position.

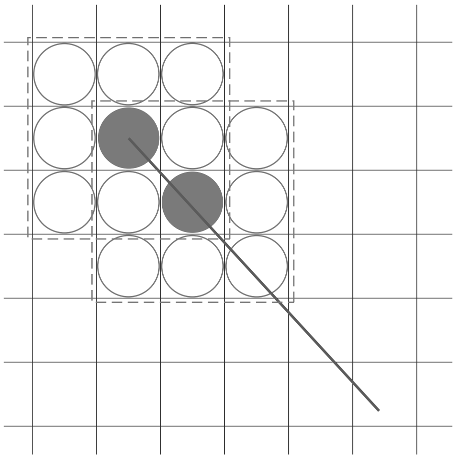


FIGURE 41
Generating a line with the pen shape of Figure 40.

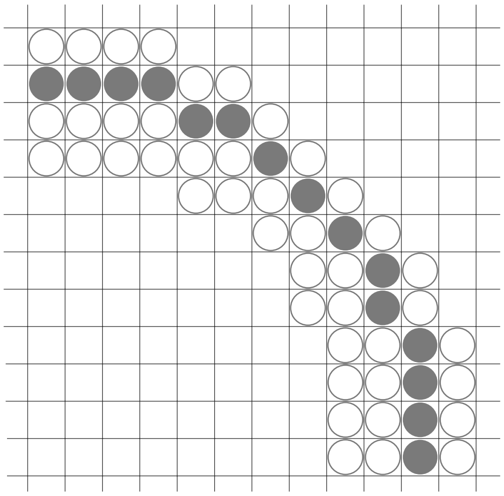


FIGURE 42
A circular arc of width 4 plotted with either vertical or horizontal pixel spans, depending on the slope.

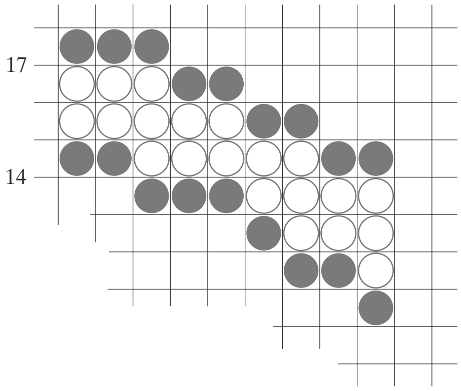


FIGURE 43
A circular arc of width 4 and radius 16 displayed by filling the region between two concentric arcs.

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.

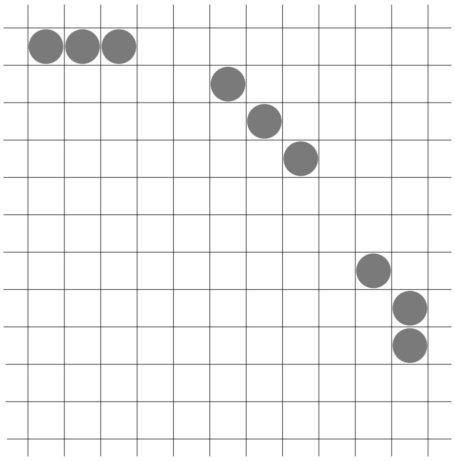


FIGURE 44
A dashed circular arc displayed with a dash span of 3 pixels and an inter-dash spacing of 2 pixels.

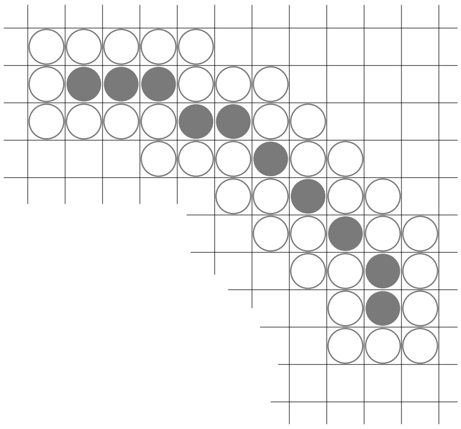


FIGURE 45
A circular arc displayed with a rectangular pen.

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

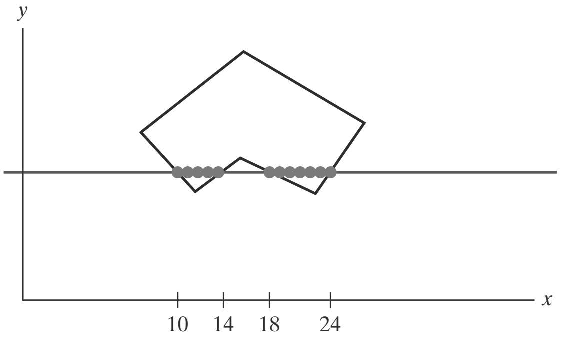


FIGURE 46
Interior pixels along a scan line passing through a polygon fill area.

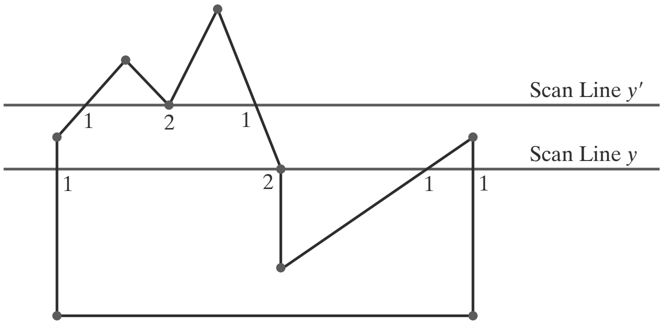


FIGURE 47
Intersection points along scan lines that intersect polygon vertices. Scan line y generates an odd number of intersections, but scan line y' generates an even number of intersections that can be paired to identify correctly the interior pixel spans.

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

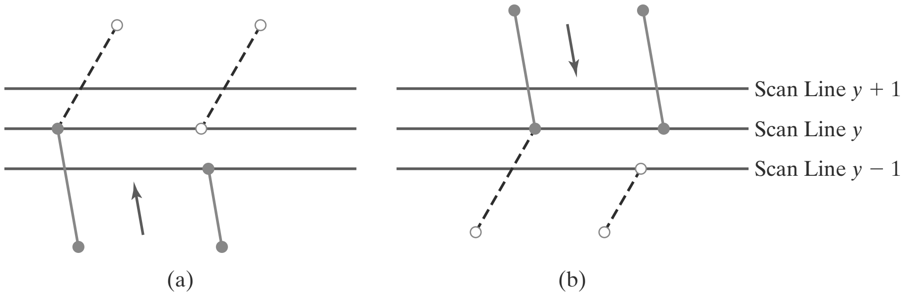


FIGURE 48 Adjusting endpoint y values for a polygon, as we process edges in order around the polygon perimeter. The edge currently being processed is indicated as a solid line. In (a), the y coordinate of the upper endpoint of the current edge is decreased by 1. In (b), the y coordinate of the upper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

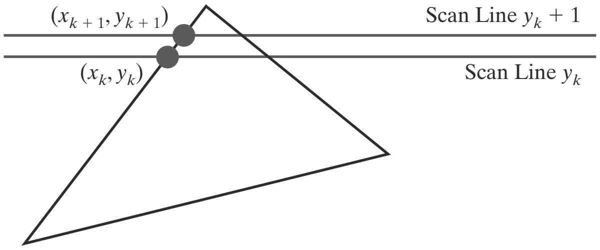


FIGURE 49 Two successive scan lines intersecting a polygon boundary.

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

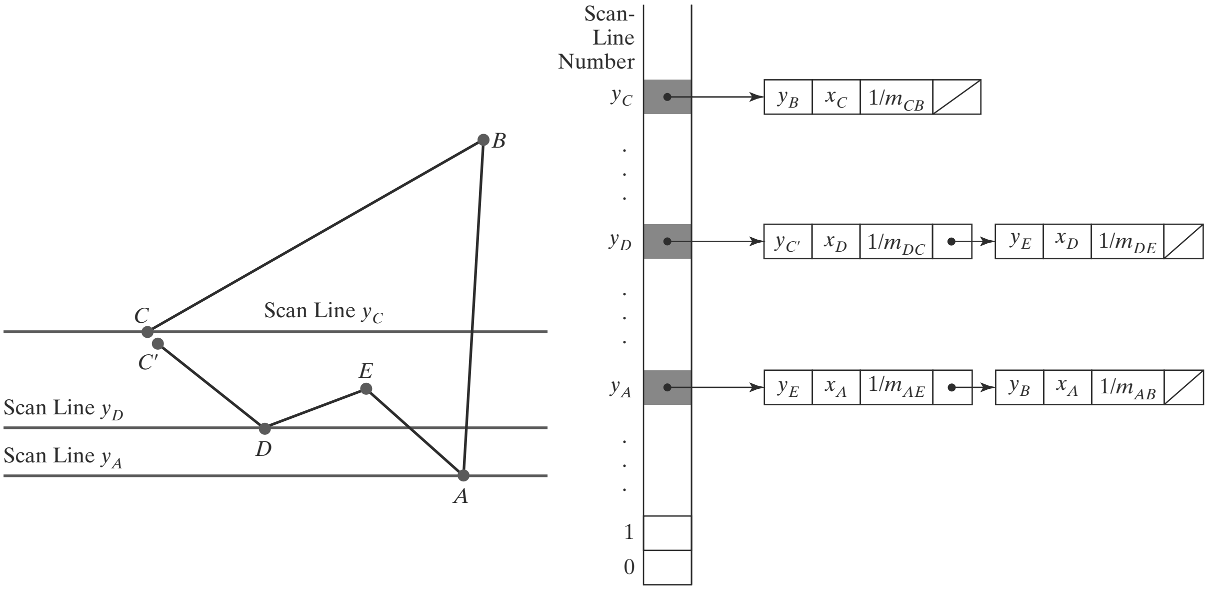

B
C
Scan Line y_C
C′
E
Scan Line y_D
D
Scan Line y_A
A
Scan-Line Number
y_C
y_B | x_C | $1/m_{CB}$
y_D
$y_{C'}$ | x_D | $1/m_{DC}$
y_E | x_D | $1/m_{DE}$
y_A
y_E | x_A | $1/m_{AE}$
y_B | x_A | $1/m_{AB}$
1
0

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.

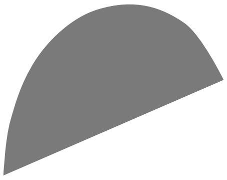


FIGURE 51
Interior fill of an elliptical arc.

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

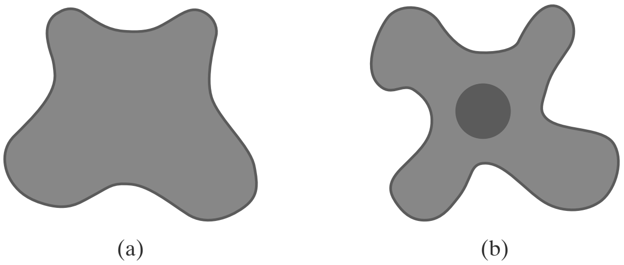


FIGURE 52
Example of color boundaries for a boundary-fill procedure.

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 ($x \pm 1$, $y \pm 1$).

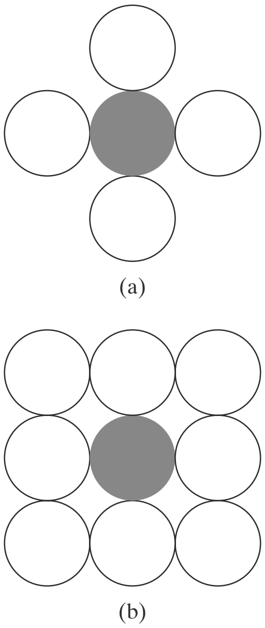


FIGURE 53
Fill methods applied to a 4-connected area (a) and to an 8-connected area (b). Hollow circles represent pixels to be tested from the current test position, shown as a solid color.

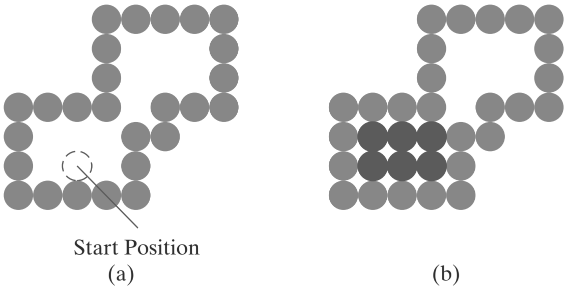


FIGURE 54
The area defined within the color boundary (a) is only partially filled in (b) using a 4-connected boundary-fill algorithm.

```
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.

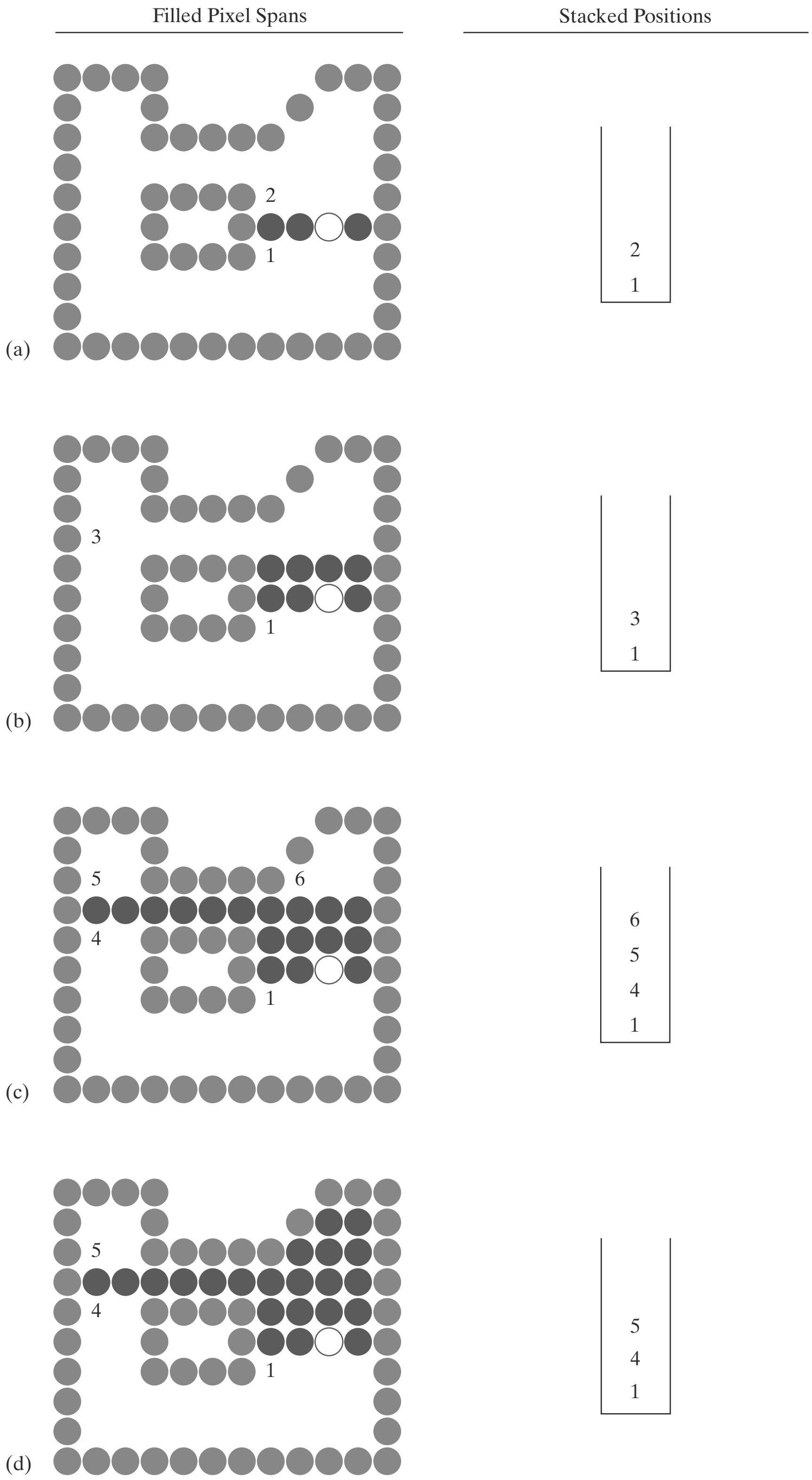


FIGURE 55
Boundary fill across pixel spans for a 4-connected area: (a) Initial scan line with a filled pixel span, showing the position of the initial point (hollow) and the stacked positions for pixel spans on adjacent scan lines. (b) Filled pixel span on the first scan line above the initial scan line 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

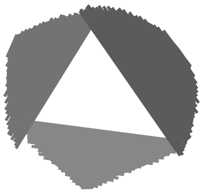


FIGURE 56
An area defined within multiple color boundaries.

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

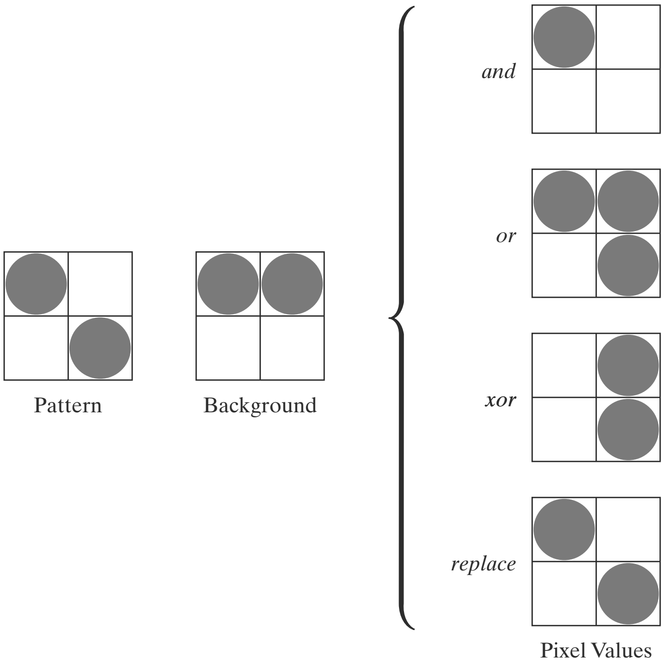


FIGURE 57
Combining a fill pattern with a background pattern using logical operations *and*, *or*, and *xor* (*exclusive or*), and using simple replacement.

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:

$$f_s = 2 f_{\text{max}} \tag{68}$$

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 $\Delta x_{\text{cycle}} = 1/f_{\text{max}}$. 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

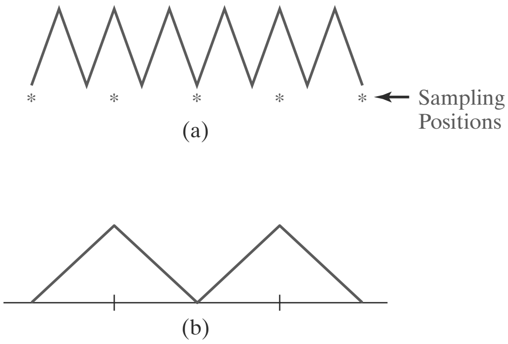


FIGURE 58
Sampling the periodic shape in (a) at the indicated positions produces the aliased lower-frequency representation in (b).

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

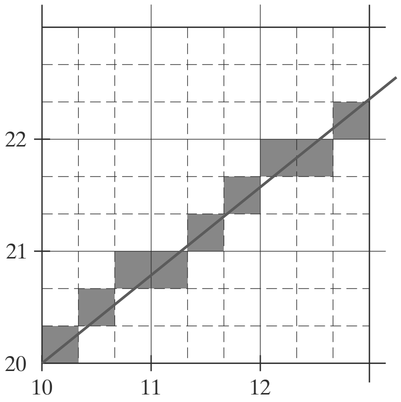


FIGURE 59
Supersampling subpixel positions along a straight-line segment whose left endpoint is at screen coordinates (10, 20).

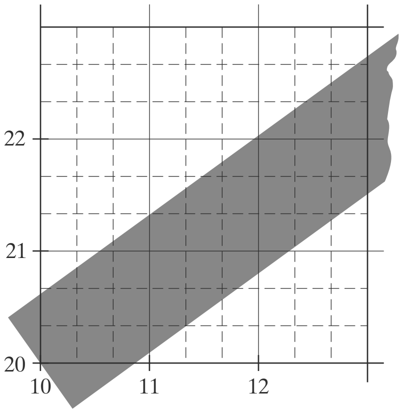


FIGURE 60
Supersampling subpixel positions in relation to the interior of a line of finite width.

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.

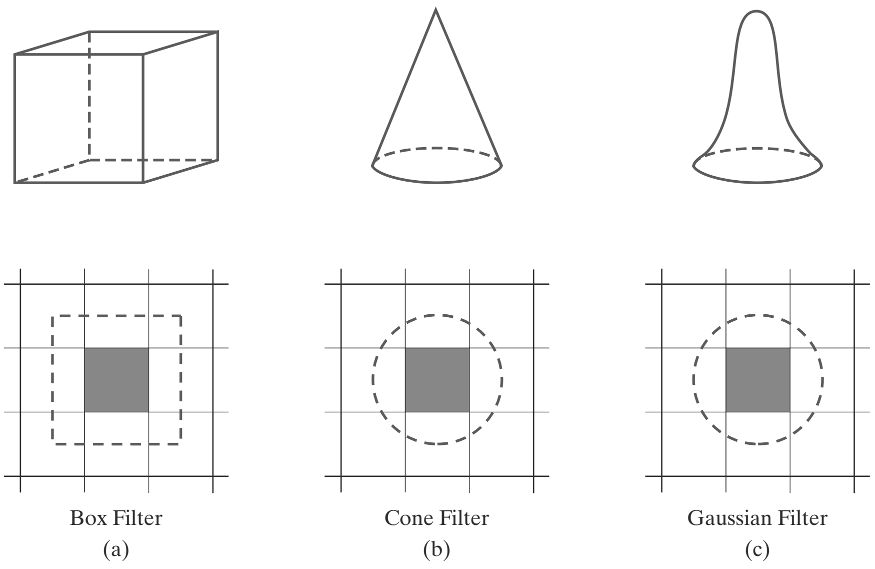


FIGURE 62
Common filter functions used to antialias line paths. The volume of each filter is normalized to 1.0, and the height gives the relative weight at any subpixel position.

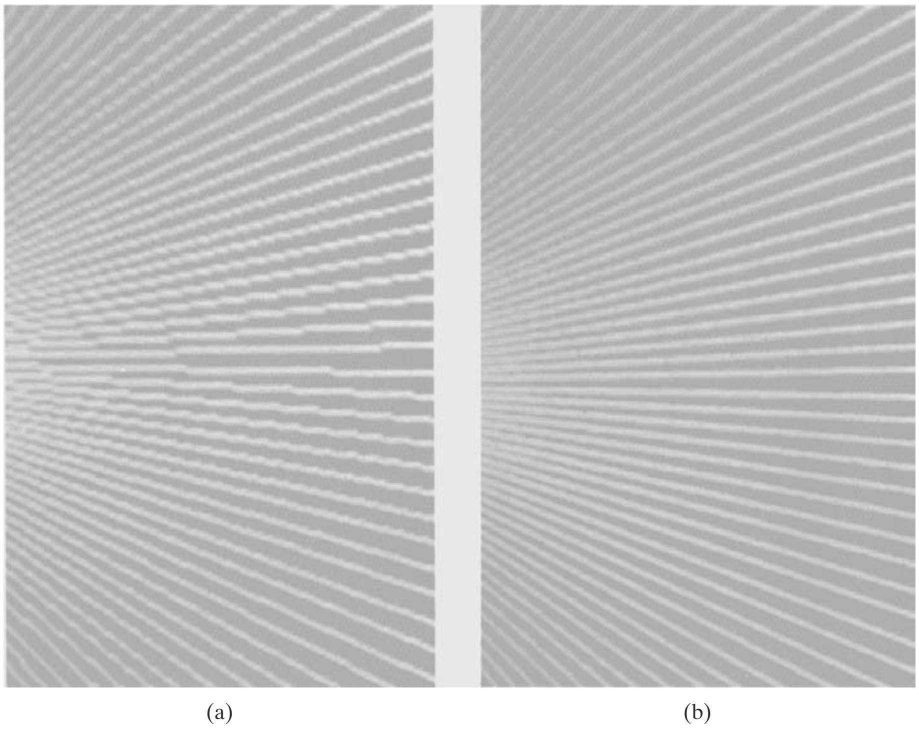


FIGURE 63
Jagged lines (a), plotted on the Merlin 9200 system, are smoothed (b) with an antialiasing technique called pixel phasing. This technique increases the number of addressable points on the system from 768 by 576 to 3072 by 2304. (*Courtesy of Peritek Corp.*)

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

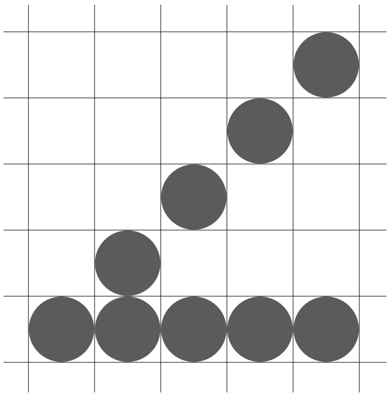


FIGURE 64
Unequal length lines displayed with the same number of pixels in each line.

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

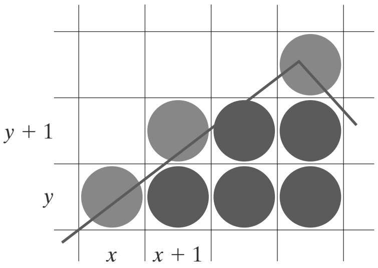


FIGURE 65
Adjusting pixel intensities along an area boundary.

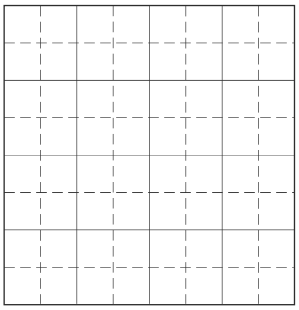


FIGURE 66
A 4 × 4 pixel section of a raster display subdivided into an 8 × 8 grid.

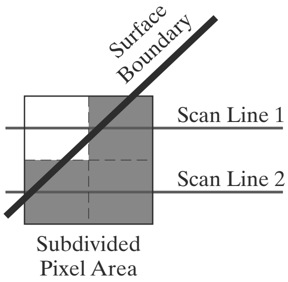


FIGURE 67
A subdivided pixel area with three subdivisions inside an object boundary line.

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

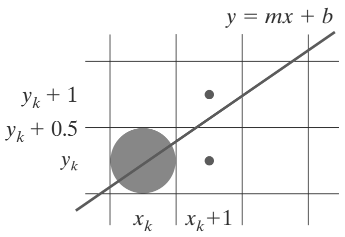


FIGURE 68
Boundary edge of a fill area passing through a pixel grid section.

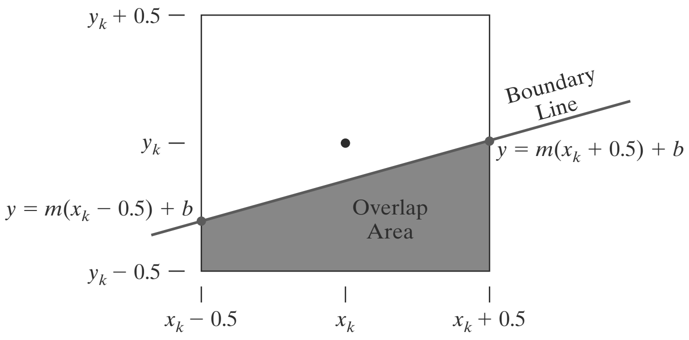


FIGURE 69
Overlap area of a pixel rectangle, centered at position (x_k, y_k), with the interior of a polygon fill area.

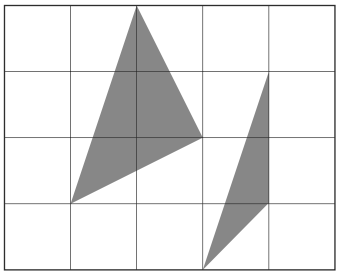


FIGURE 70
Polygons with more than one boundary line passing through individual pixel regions.

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 $-10 \leq x \leq 10$.

$$y = \frac{1}{12}x^3$$

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 $-10 \leq x \leq 10$.

23 Use the midpoint method and symmetry considerations to scan convert the parabola

$$y = 50 - x^2$$

over the interval $-5 \leq x \leq 5$.

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.

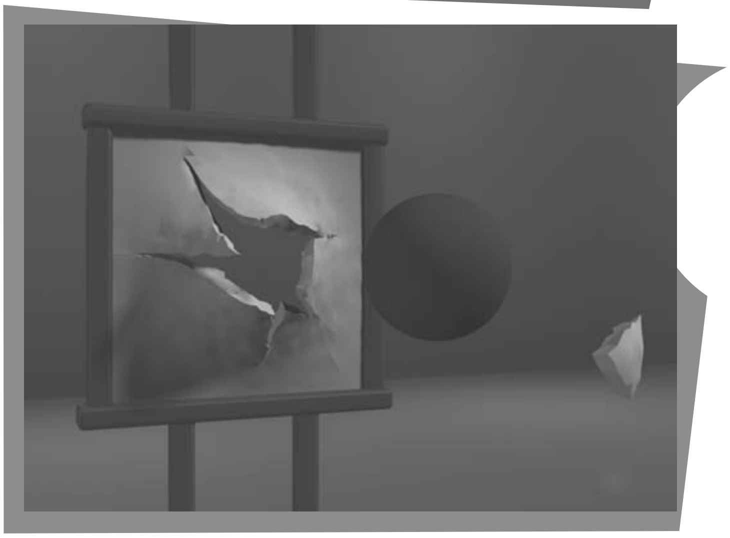
IN MORE DEPTH

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

Two-Dimensional Geometric Transformations



So far, we have seen how we can describe a scene in terms of graphics primitives, such as line segments and fill areas, and the attributes associated with these primitives. Also, we have explored the scan-line algorithms for displaying output primitives on a raster device. Now, we take a look at transformation operations that we can apply to objects to reposition or resize them. These operations are also used in the viewing routines that convert a world-coordinate scene description to a display for an output device. In addition, they are used in a variety of other applications, such as computer-aided design (CAD) and computer animation. An architect, for example, creates a layout by arranging the orientation and size of the component parts of a design, and a computer animator develops a video sequence by moving the "camera" position or the objects in a scene along specified paths. Operations that are applied to the geometric description of an object to change its position, orientation, or size are called **geometric transformations.**

Sometimes geometric transformations are also referred to as *modeling transformations*, but some graphics packages make a

From Chapter 7 of *Computer Graphics with OpenGL®*, Fourth Edition, Donald Hearn, M. Pauline Baker, Warren R. Carithers.

distinction between the two. In general, modeling transformations are used to construct a scene or to give the hierarchical description of a complex object that is composed of several parts, which in turn could be composed of simpler parts, and so forth. For example, an aircraft consists of wings, tail, fuselage, engine, and other components, each of which can be specified in terms of second-level components, and so on, down the hierarchy of component parts. Thus, the aircraft can be described in terms of these components and an associated "modeling" transformation for each one that describes how that component is to be fitted into the overall aircraft design. Geometric transformations, on the other hand, can be used to describe how objects might move around in a scene during an animation sequence or simply to view them from another angle. Therefore, some graphics packages provide two sets of transformation routines, while other packages have a single set of functions that can be used for both geometric transformations and modeling transformations.

1 Basic Two-Dimensional Geometric Transformations

The geometric-transformation functions that are available in all graphics packages are those for translation, rotation, and scaling. Other useful transformation routines that are sometimes included in a package are reflection and shearing operations. To introduce the general concepts associated with geometric transformations, we first consider operations in two dimensions. Once we understand the basic concepts, we can easily write routines to perform geometric transformations on objects in a two-dimensional scene.

Two-Dimensional Translation

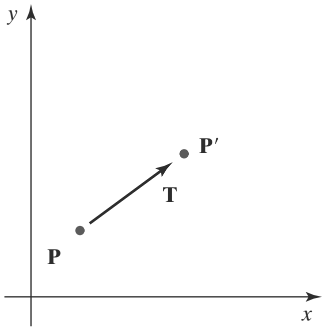


FIGURE 1
Translating a point from position **P** to position **P**′ using a translation vector **T**.

We perform a **translation** on a single coordinate point by adding offsets to its coordinates so as to generate a new coordinate position. In effect, we are moving the original point position along a straight-line path to its new location. Similarly, a translation is applied to an object that is defined with multiple coordinate positions, such as a quadrilateral, by relocating all the coordinate positions by the same displacement along parallel paths. Then the complete object is displayed at the new location.

To translate a two-dimensional position, we add **translation distances** t_x and t_y to the original coordinates (x, y) to obtain the new coordinate position (x', y') as shown in Figure 1.

$$x' = x + t_x, \qquad y' = y + t_y \tag{1}$$

The translation distance pair (t_x, t_y) is called a **translation vector** or **shift vector.**

We can express Equations 1 as a single matrix equation by using the following column vectors to represent coordinate positions and the translation vector:

$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix}, \qquad \mathbf{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \qquad \mathbf{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} \tag{2}$$

This allows us to write the two-dimensional translation equations in the matrix form

$$\mathbf{P}' = \mathbf{P} + \mathbf{T} \tag{3}$$

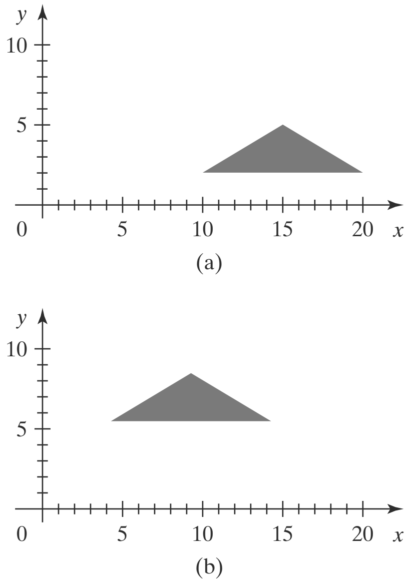


FIGURE 2
Moving a polygon from position (a) to position (b) with the translation vector (−5.50, 3.75).

Translation is a *rigid-body transformation* that moves objects without deformation. That is, every point on the object is translated by the same amount. A straight-line segment is translated by applying Equation 3 to each of the two line endpoints and redrawing the line between the new endpoint positions. A polygon is translated similarly. We add a translation vector to the coordinate position of each vertex and then regenerate the polygon using the new set of vertex coordinates. Figure 2 illustrates the application of a specified translation vector to move an object from one position to another.

The following routine illustrates the translation operations. An input translation vector is used to move the *n* vertices of a polygon from one world-coordinate position to another, and OpenGL routines are used to regenerate the translated polygon.

```
class wcPt2D {
   public:
      GLfloat x, y;
};

void translatePolygon (wcPt2D * verts, GLint nVerts, GLfloat tx, GLfloat ty)
{
   GLint k;

   for (k = 0; k < nVerts; k++) {
      verts [k].x = verts [k].x + tx;
      verts [k].y = verts [k].y + ty;
   }
   glBegin (GL_POLYGON);
      for (k = 0; k < nVerts; k++)
         glVertex2f (verts [k].x, verts [k].y);
   glEnd ( );
}
```

If we want to delete the original polygon, we could display it in the background color before translating it. Other methods for deleting picture components are available in some graphics packages. Also, if we want to save the original polygon position, we can store the translated positions in a different array.

Similar methods are used to translate other objects. To change the position of a circle or ellipse, we translate the center coordinates and redraw the figure in the new location. For a spline curve, we translate the points that define the curve path and then reconstruct the curve sections between the new coordinate positions.

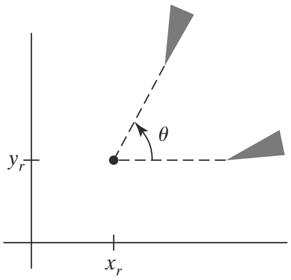


FIGURE 3
Rotation of an object through angle θ about the pivot point (x_r, y_r).

Two-Dimensional Rotation

We generate a **rotation** transformation of an object by specifying a **rotation axis** and a **rotation angle**. All points of the object are then transformed to new positions by rotating the points through the specified angle about the rotation axis.

A two-dimensional rotation of an object is obtained by repositioning the object along a circular path in the xy plane. In this case, we are rotating the object about a rotation axis that is perpendicular to the xy plane (parallel to the coordinate z axis). Parameters for the two-dimensional rotation are the rotation angle θ and a position (x_r, y_r), called the **rotation point** (or **pivot point**), about which the object is to be rotated (Figure 3). The pivot point is the intersection position of the rotation axis with the xy plane. A positive value for the angle θ defines a counterclockwise rotation about the pivot point, as in Figure 3, and a negative value rotates objects in the clockwise direction.

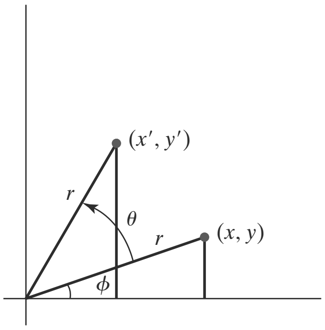


FIGURE 4
Rotation of a point from position (x, y) to position (x', y') through an angle θ relative to the coordinate origin. The original angular displacement of the point from the x axis is ϕ.

To simplify the explanation of the basic method, we first determine the transformation equations for rotation of a point position **P** when the pivot point is at the coordinate origin. The angular and coordinate relationships of the original and transformed point positions are shown in Figure 4. In this figure, r is the constant distance of the point from the origin, angle ϕ is the original angular position of the point from the horizontal, and θ is the rotation angle. Using standard trigonometric identities, we can express the transformed coordinates in terms of angles θ and ϕ as

$$\begin{aligned} x' &= r\cos(\phi+\theta) = r\cos\phi\cos\theta - r\sin\phi\sin\theta \\ y' &= r\sin(\phi+\theta) = r\cos\phi\sin\theta + r\sin\phi\cos\theta \end{aligned} \tag{4}$$

The original coordinates of the point in polar coordinates are

$$x = r\cos\phi, \qquad y = r\sin\phi \tag{5}$$

Substituting expressions 5 into 4, we obtain the transformation equations for rotating a point at position (x, y) through an angle θ about the origin:

$$\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \end{aligned} \tag{6}$$

With the column-vector representations 2 for coordinate positions, we can write the rotation equations in the matrix form

$$\mathbf{P}' = \mathbf{R}\cdot\mathbf{P} \tag{7}$$

where the rotation matrix is

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{8}$$

A column-vector representation for a coordinate position **P**, as in Equations 2, is standard mathematical notation. However, early graphics systems sometimes used a row-vector representation for point positions. This changes the order in which the matrix multiplication for a rotation would be performed. But now, graphics packages such as OpenGL, Java, PHIGS, and GKS all follow the standard column-vector convention.

Rotation of a point about an arbitrary pivot position is illustrated in Figure 5. Using the trigonometric relationships indicated by the two right triangles in this figure, we can generalize Equations 6 to obtain the transformation equations for rotation of a point about any specified rotation position (x_r, y_r):

$$\begin{aligned} x' &= x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta \\ y' &= y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta \end{aligned} \tag{9}$$

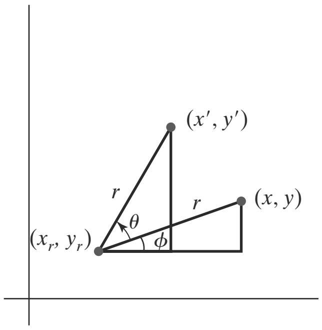


FIGURE 5
Rotating a point from position (x, y) to position (x', y') through an angle θ about rotation point (x_r, y_r).

These general rotation equations differ from Equations 6 by the inclusion of additive terms, as well as the multiplicative factors on the coordinate values. The matrix expression 7 could be modified to include pivot coordinates by including the matrix addition of a column vector whose elements contain the additive (translational) terms in Equations 9. There are better ways, however, to formulate such matrix equations, and in Section 2, we discuss a more consistent scheme for representing the transformation equations.

As with translations, rotations are rigid-body transformations that move objects without deformation. Every point on an object is rotated through the same angle. A straight-line segment is rotated by applying Equations 9 to each of the two line endpoints and redrawing the line between the new endpoint positions. A polygon is rotated by displacing each vertex using the specified rotation angle and then regenerating the polygon using the new vertices. We rotate a curve by repositioning the defining points for the curve and then redrawing it. A circle or an ellipse, for instance, can be rotated about a noncentral pivot point by moving the center position through the arc that subtends the specified rotation angle. In addition, we could rotate an ellipse about its center coordinates simply by rotating the major and minor axes.

In the following code example, a polygon is rotated about a specified world-coordinate pivot point. Parameters input to the rotation procedure are the original vertices of the polygon, the pivot-point coordinates, and the rotation angle `theta` specified in radians. Following the transformation of the vertex positions, the polygon is regenerated using OpenGL routines.

```
class wcPt2D {
   public:
      GLfloat x, y;
};

void rotatePolygon (wcPt2D * verts, GLint nVerts, wcPt2D pivPt,
                      GLdouble theta)
{
   wcPt2D * vertsRot;
   GLint k;

   for (k = 0; k < nVerts; k++) {
      vertsRot [k].x = pivPt.x + (verts [k].x - pivPt.x) * cos (theta)
                            - (verts [k].y - pivPt.y) * sin (theta);
      vertsRot [k].y = pivPt.y + (verts [k].x - pivPt.x) * sin (theta)
                            + (verts [k].y - pivPt.y) * cos (theta);
   }
   glBegin {GL_POLYGON};
      for (k = 0; k < nVerts; k++)
         glVertex2f (vertsRot [k].x, vertsRot [k].y);
   glEnd ( );
}
```

Two-Dimensional Scaling

To alter the size of an object, we apply a **scaling** transformation. A simple two-dimensional scaling operation is performed by multiplying object positions (x, y) by **scaling factors** s_x and s_y to produce the transformed coordinates (x', y'):

$$x' = x \cdot s_x, \qquad y' = y \cdot s_y \tag{10}$$

Scaling factor s_x scales an object in the x direction, while s_y scales in the y direction. The basic two-dimensional scaling equations 10 can also be written in the following matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \tag{11}$$

or

$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P} \tag{12}$$

where **S** is the 2 × 2 scaling matrix in Equation 11.

Any positive values can be assigned to the scaling factors s_x and s_y. Values less than 1 reduce the size of objects; values greater than 1 produce enlargements. Specifying a value of 1 for both s_x and s_y leaves the size of objects unchanged. When s_x and s_y are assigned the same value, a **uniform scaling** is produced, which maintains relative object proportions. Unequal values for s_x and s_y result in a **differential scaling** that is often used in design applications, where pictures are constructed from a few basic shapes that can be adjusted by scaling and positioning transformations (Figure 6). In some systems, negative values can also be specified for the scaling parameters. This not only resizes an object, it reflects it about one or more of the coordinate axes.

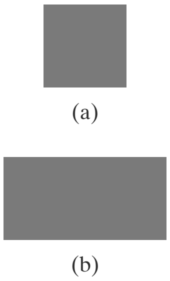


FIGURE 6
Turning a square (a) into a rectangle (b) with scaling factors $s_x = 2$ and $s_y = 1$.

Objects transformed with Equation 11 are both scaled and repositioned. Scaling factors with absolute values less than 1 move objects closer to the coordinate origin, while absolute values greater than 1 move coordinate positions farther from the origin. Figure 7 illustrates scaling of a line by assigning the value 0.5 to both s_x and s_y in Equation 11. Both the line length and the distance from the origin are reduced by a factor of $\frac{1}{2}$.

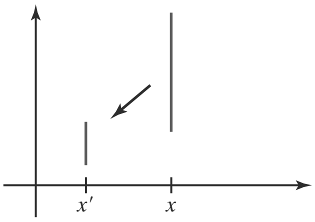


FIGURE 7
A line scaled with Equation 12 using $s_x = s_y = 0.5$ is reduced in size and moved closer to the coordinate origin.

We can control the location of a scaled object by choosing a position, called the **fixed point,** that is to remain unchanged after the scaling transformation. Coordinates for the fixed point, (x_f, y_f), are often chosen at some object position, such as its centroid (see Appendix A), but any other spatial position can be selected. Objects are now resized by scaling the distances between object points and the fixed point (Figure 8). For a coordinate position (x, y), the scaled coordinates (x', y') are then calculated from the following relationships:

$$x' - x_f = (x - x_f)s_x, \qquad y' - y_f = (y - y_f)s_y \tag{13}$$

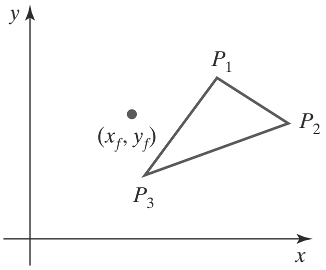


FIGURE 8
Scaling relative to a chosen fixed point (x_f, y_f). The distance from each polygon vertex to the fixed point is scaled by Equations 13.

We can rewrite Equations 13 to separate the multiplicative and additive terms as

$$\begin{aligned} x' &= x \cdot s_x + x_f(1 - s_x) \\ y' &= y \cdot s_y + y_f(1 - s_y) \end{aligned} \tag{14}$$

where the additive terms $x_f(1 - s_x)$ and $y_f(1 - s_y)$ are constants for all points in the object.

Including coordinates for a fixed point in the scaling equations is similar to including coordinates for a pivot point in the rotation equations. We can set up a column vector whose elements are the constant terms in Equations 14, then add this column vector to the product **S** · **P** in Equation 12. In the next section, we discuss a matrix formulation for the transformation equations that involves only matrix multiplication.

Polygons are scaled by applying transformations 14 to each vertex, then regenerating the polygon using the transformed vertices. For other objects, we apply the scaling transformation equations to the parameters defining the objects. To change the size of a circle, we can scale its radius and calculate the new coordinate positions around the circumference. And to change the size of an ellipse, we apply scaling parameters to its two axes and then plot the new ellipse positions about its center coordinates.

The following procedure illustrates an application of the scaling calculations for a polygon. Coordinates for the polygon vertices and for the fixed point are input parameters, along with the scaling factors. After the coordinate transformations, OpenGL routines are used to generate the scaled polygon.

```
class wcPt2D {
   public:
      GLfloat x, y;
};

void scalePolygon (wcPt2D * verts, GLint nVerts, wcPt2D fixedPt,
                           GLfloat sx, GLfloat sy)
{
   wcPt2D vertsNew;
   GLint k;

   for (k = 0; k < nVerts; k++) {
     vertsNew [k].x = verts [k].x * sx + fixedPt.x * (1 - sx);
     vertsNew [k].y = verts [k].y * sy + fixedPt.y * (1 - sy);
   }
   glBegin {GL_POLYGON};
      for (k = 0; k < nVerts; k++)
         glVertex2f (vertsNew [k].x, vertsNew [k].y);
   glEnd ( );
}
```

2 Matrix Representations and Homogeneous Coordinates

Many graphics applications involve sequences of geometric transformations. An animation might require an object to be translated and rotated at each increment of the motion. In design and picture construction applications, we perform translations, rotations, and scalings to fit the picture components into their proper positions. The viewing transformations involve sequences of translations and rotations to take us from the original scene specification to the display on an output device. Here, we consider how the matrix representations discussed in the previous sections can be reformulated so that such transformation sequences can be processed efficiently.

We have seen in Section 1 that each of the three basic two-dimensional transformations (translation, rotation, and scaling) can be expressed in the general matrix form

$$\mathbf{P}' = \mathbf{M}_1 \cdot \mathbf{P} + \mathbf{M}_2 \tag{15}$$

with coordinate positions **P** and **P**′ represented as column vectors. Matrix $\mathbf{M}_1$ is a 2 × 2 array containing multiplicative factors, and $\mathbf{M}_2$ is a two-element column matrix containing translational terms. For translation, $\mathbf{M}_1$ is the identity matrix. For rotation or scaling, $\mathbf{M}_2$ contains the translational terms associated with the pivot point or scaling fixed point. To produce a sequence of transformations with these equations, such as scaling followed by rotation and then translation, we could calculate the transformed coordinates one step at a time. First, coordinate positions are scaled, then these scaled coordinates are rotated, and finally, the rotated coordinates are translated. A more efficient approach, however, is to combine the transformations so that the final coordinate positions are obtained directly from the initial coordinates, without calculating intermediate coordinate values. We can do this by reformulating Equation 15 to eliminate the matrix addition operation.

Homogeneous Coordinates

Multiplicative and translational terms for a two-dimensional geometric transformation can be combined into a single matrix if we expand the representations to 3 × 3 matrices. Then we can use the third column of a transformation matrix for the translation terms, and all transformation equations can be expressed as matrix multiplications. But to do so, we also need to expand the matrix representation for a two-dimensional coordinate position to a three-element column matrix. A standard technique for accomplishing this is to expand each two-dimensional coordinate-position representation (x, y) to a three-element representation (x_h, y_h, h), called **homogeneous coordinates,** where the **homogeneous parameter** h is a nonzero value such that

$$x = \frac{x_h}{h}, \qquad y = \frac{y_h}{h} \tag{16}$$

Therefore, a general two-dimensional homogeneous coordinate representation could also be written as $(h \cdot x, h \cdot y, h)$. For geometric transformations, we can choose the homogeneous parameter h to be any nonzero value. Thus, each coordinate point (x, y) has an infinite number of equivalent homogeneous representations. A convenient choice is simply to set $h = 1$. Each two-dimensional position is then represented with homogeneous coordinates $(x, y, 1)$. Other values for parameter h are needed, for example, in matrix formulations of three-dimensional viewing transformations.

The term *homogeneous coordinates* is used in mathematics to refer to the effect of this representation on Cartesian equations. When a Cartesian point (x, y) is converted to a homogeneous representation (x_h, y_h, h), equations containing x and y, such as $f(x, y) = 0$, become homogeneous equations in the three parameters x_h, y_h, and h. This just means that if each of the three parameters is replaced by any value v times that parameter, the value v can be factored out of the equations.

Expressing positions in homogeneous coordinates allows us to represent all geometric transformation equations as matrix multiplications, which is the standard method used in graphics systems. Two-dimensional coordinate positions are represented with three-element column vectors, and two-dimensional transformation operations are expressed as 3 × 3 matrices.

Two-Dimensional Translation Matrix

Using a homogeneous-coordinate approach, we can represent the equations for a two-dimensional translation of a coordinate position using the following matrix multiplication:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{17}$$

This translation operation can be written in the abbreviated form

$$\mathbf{P}' = \mathbf{T}(t_x, t_y) \cdot \mathbf{P} \tag{18}$$

with $\mathbf{T}(t_x, t_y)$ as the 3×3 translation matrix in Equation 17. In situations where there is no ambiguity about the translation parameters, we can simply represent the translation matrix as **T**.

Two-Dimensional Rotation Matrix

Similarly, two-dimensional rotation transformation equations about the coordinate origin can be expressed in the matrix form

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{19}$$

or as

$$\mathbf{P}' = \mathbf{R}(\theta) \cdot \mathbf{P} \tag{20}$$

The rotation transformation operator $\mathbf{R}(\theta)$ is the 3×3 matrix in Equation 19 with rotation parameter θ. We can also write this rotation matrix simply as **R**.

In some graphics libraries, a two-dimensional rotation function generates only rotations about the coordinate origin, as in Equation 19. A rotation about any other pivot point must then be performed as a sequence of transformation operations. An alternative approach in a graphics package is to provide additional parameters in the rotation routine for the pivot-point coordinates. A rotation routine that includes a pivot-point parameter then sets up a general rotation matrix without the need to invoke a succession of transformation functions.

Two-Dimensional Scaling Matrix

Finally, a scaling transformation relative to the coordinate origin can now be expressed as the matrix multiplication

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{21}$$

or

$$\mathbf{P}' = \mathbf{S}(s_x, s_y) \cdot \mathbf{P} \tag{22}$$

The scaling operator $\mathbf{S}(s_x, s_y)$ is the 3×3 matrix in Equation 21 with parameters s_x and s_y. And, in most cases, we can represent the scaling matrix simply as **S**.

Some libraries provide a scaling function that can generate only scaling with respect to the coordinate origin, as in Equation 21. In this case, a scaling transformation relative to another reference position is handled as a succession of transformation operations. However, other systems do include a general scaling routine that can construct the homogeneous matrix for scaling with respect to a designated fixed point.

3 Inverse Transformations

For translation, we obtain the inverse matrix by negating the translation distances. Thus, if we have two-dimensional translation distances t_x and t_y, the inverse translation matrix is

$$\mathbf{T}^{-1} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{23}$$

This produces a translation in the opposite direction, and the product of a translation matrix and its inverse produces the identity matrix.

An inverse rotation is accomplished by replacing the rotation angle by its negative. For example, a two-dimensional rotation through an angle θ about the coordinate origin has the inverse transformation matrix

$$\mathbf{R}^{-1} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{24}$$

Negative values for rotation angles generate rotations in a clockwise direction, so the identity matrix is produced when any rotation matrix is multiplied by its inverse. Because only the sine function is affected by the change in sign of the rotation angle, the inverse matrix can also be obtained by interchanging rows and columns. That is, we can calculate the inverse of any rotation matrix **R** by evaluating its transpose ($\mathbf{R}^{-1} = \mathbf{R}^T$).

We form the inverse matrix for any scaling transformation by replacing the scaling parameters with their reciprocals. For two-dimensional scaling with parameters s_x and s_y applied relative to the coordinate origin, the inverse transformation matrix is

$$\mathbf{S}^{-1} = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{25}$$

The inverse matrix generates an opposite scaling transformation, so the multiplication of any scaling matrix with its inverse produces the identity matrix.

4 Two-Dimensional Composite Transformations

Using matrix representations, we can set up a sequence of transformations as a **composite transformation matrix** by calculating the product of the individual transformations. Forming products of transformation matrices is often referred to as a **concatenation,** or **composition,** of matrices. Because a coordinate position is represented with a homogeneous column matrix, we must premultiply the column matrix by the matrices representing any transformation sequence. Also, because many positions in a scene are typically transformed by the same sequence, it is more efficient to first multiply the transformation matrices to form a single composite matrix. Thus, if we want to apply two transformations to point position **P**, the transformed location would be calculated as

$$\begin{aligned} \mathbf{P}' &= \mathbf{M}_2 \cdot \mathbf{M}_1 \cdot \mathbf{P} \\ &= \mathbf{M} \cdot \mathbf{P} \end{aligned} \tag{26}$$

The coordinate position is transformed using the composite matrix **M**, rather than applying the individual transformations $\mathbf{M}_1$ and then $\mathbf{M}_2$.

Composite Two-Dimensional Translations

If two successive translation vectors (t_{1x}, t_{1y}) and (t_{2x}, t_{2y}) are applied to a two-dimensional coordinate position **P**, the final transformed location $\mathbf{P}'$ is calculated as

$$\begin{aligned} \mathbf{P}' &= \mathbf{T}(t_{2x}, t_{2y}) \cdot \{\mathbf{T}(t_{1x}, t_{1y}) \cdot \mathbf{P}\} \\ &= \{\mathbf{T}(t_{2x}, t_{2y}) \cdot \mathbf{T}(t_{1x}, t_{1y})\} \cdot \mathbf{P} \end{aligned} \tag{27}$$

where **P** and $\mathbf{P}'$ are represented as three-element, homogeneous-coordinate column vectors. We can verify this result by calculating the matrix product for the two associative groupings. Also, the composite transformation matrix for this sequence of translations is

$$\begin{bmatrix} 1 & 0 & t_{2x} \\ 0 & 1 & t_{2y} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_{1x} \\ 0 & 1 & t_{1y} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{1x} + t_{2x} \\ 0 & 1 & t_{1y} + t_{2y} \\ 0 & 0 & 1 \end{bmatrix} \tag{28}$$

or

$$\mathbf{T}(t_{2x}, t_{2y}) \cdot \mathbf{T}(t_{1x}, t_{1y}) = \mathbf{T}(t_{1x} + t_{2x}, t_{1y} + t_{2y}) \tag{29}$$

which demonstrates that two successive translations are additive.

Composite Two-Dimensional Rotations

Two successive rotations applied to a point **P** produce the transformed position

$$\begin{aligned} \mathbf{P}' &= \mathbf{R}(\theta_2) \cdot \{\mathbf{R}(\theta_1) \cdot \mathbf{P}\} \\ &= \{\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1)\} \cdot \mathbf{P} \end{aligned} \tag{30}$$

By multiplying the two rotation matrices, we can verify that two successive rotations are additive:

$$\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1) = \mathbf{R}(\theta_1 + \theta_2) \tag{31}$$

so that the final rotated coordinates of a point can be calculated with the composite rotation matrix as

$$\mathbf{P}' = \mathbf{R}(\theta_1 + \theta_2) \cdot \mathbf{P} \tag{32}$$

Composite Two-Dimensional Scalings

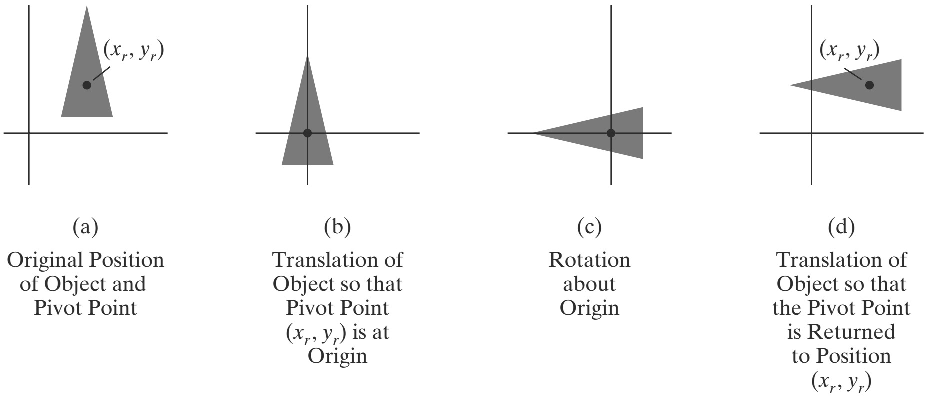
Concatenating transformation matrices for two successive scaling operations in two dimensions produces the following composite scaling matrix:

$$\begin{bmatrix} s_{2x} & 0 & 0 \\ 0 & s_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{1x} & 0 & 0 \\ 0 & s_{1y} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{1x} \cdot s_{2x} & 0 & 0 \\ 0 & s_{1y} \cdot s_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{33}$$

or

$$\mathbf{S}(s_{2x}, s_{2y}) \cdot \mathbf{S}(s_{1x}, s_{1y}) = \mathbf{S}(s_{1x} \cdot s_{2x}, \quad s_{1y} \cdot s_{2y}) \tag{34}$$

The resulting matrix in this case indicates that successive scaling operations are multiplicative. That is, if we were to triple the size of an object twice in succession, the final size would be nine times that of the original.



(a)
Original Position of Object and Pivot Point

(b)
Translation of Object so that Pivot Point (x_r, y_r) is at Origin

(c)
Rotation about Origin

(d)
Translation of Object so that the Pivot Point is Returned to Position (x_r, y_r)

FIGURE 9
A transformation sequence for rotating an object about a specified pivot point using the rotation matrix $\mathbf{R}(\theta)$ of transformation 19.

General Two-Dimensional Pivot-Point Rotation

When a graphics package provides only a rotate function with respect to the coordinate origin, we can generate a two-dimensional rotation about any other pivot point (x_r, y_r) by performing the following sequence of translate-rotate-translate operations:

1. Translate the object so that the pivot-point position is moved to the coordinate origin.
2. Rotate the object about the coordinate origin.
3. Translate the object so that the pivot point is returned to its original position.

This transformation sequence is illustrated in Figure 9. The composite transformation matrix for this sequence is obtained with the concatenation

$$\begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r\sin\theta \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r\sin\theta \\ 0 & 0 & 1 \end{bmatrix} \tag{35}$$

which can be expressed in the form

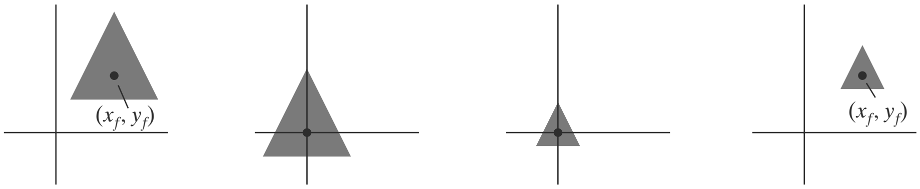
$$\mathbf{T}(x_r, y_r) \cdot \mathbf{R}(\theta) \cdot \mathbf{T}(-x_r, -y_r) = \mathbf{R}(x_r, y_r, \theta) \tag{36}$$

where $\mathbf{T}(-x_r, -y_r) = \mathbf{T}^{-1}(x_r, y_r)$. In general, a rotate function in a graphics library could be structured to accept parameters for pivot-point coordinates, as well as the rotation angle, and to generate automatically the rotation matrix of Equation 35.

General Two-Dimensional Fixed-Point Scaling

Figure 10 illustrates a transformation sequence to produce a two-dimensional scaling with respect to a selected fixed position (x_f, y_f), when we have a function that can scale relative to the coordinate origin only. This sequence is

1. Translate the object so that the fixed point coincides with the coordinate origin.



(a)
Original Position
of Object and
Fixed Point

(b)
Translate Object
so that Fixed Point
(x_f, y_f) is at Origin

(c)
Scale Object
with Respect
to Origin

(d)
Translate Object
so that the Fixed
Point is Returned
to Position (x_f, y_f)

FIGURE 10
A transformation sequence for scaling an object with respect to a specified fixed position using the scaling matrix $\mathbf{S}(s_x, s_y)$ of transformation 21.

2. Scale the object with respect to the coordinate origin.
3. Use the inverse of the translation in step (1) to return the object to its original position.

Concatenating the matrices for these three operations produces the required scaling matrix:

$$\begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & x_f(1-s_x) \\ 0 & s_y & y_f(1-s_y) \\ 0 & 0 & 1 \end{bmatrix} \tag{37}$$

or

$$\mathbf{T}(x_f, y_f) \cdot \mathbf{S}(s_x, s_y) \cdot \mathbf{T}(-x_f, -y_f) = \mathbf{S}(x_f, y_f, s_x, s_y) \tag{38}$$

This transformation is generated automatically in systems that provide a scale function that accepts coordinates for the fixed point.

General Two-Dimensional Scaling Directions

Parameters s_x and s_y scale objects along the x and y directions. We can scale an object in other directions by rotating the object to align the desired scaling directions with the coordinate axes before applying the scaling transformation.

Suppose we want to apply scaling factors with values specified by parameters s_1 and s_2 in the directions shown in Figure 11. To accomplish the scaling without changing the orientation of the object, we first perform a rotation so that the directions for s_1 and s_2 coincide with the x and y axes, respectively. Then the scaling transformation $\mathbf{S}(s_1, s_2)$ is applied, followed by an opposite rotation to return points to their original orientations. The composite matrix resulting from the product of these three transformations is

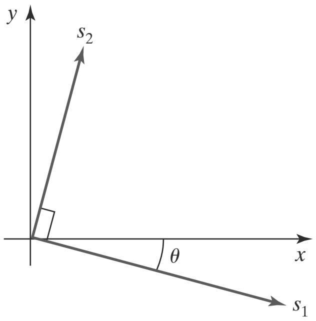


FIGURE 11
Scaling parameters s_1 and s_2 along orthogonal directions defined by the angular displacement θ.

$$\mathbf{R}^{-1}(\theta) \cdot \mathbf{S}(s_1, s_2) \cdot \mathbf{R}(\theta) = \begin{bmatrix} s_1 \cos^2\theta + s_2 \sin^2\theta & (s_2 - s_1)\cos\theta \sin\theta & 0 \\ (s_2 - s_1)\cos\theta\sin\theta & s_1 \sin^2\theta + s_2\cos^2\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{39}$$

As an example of this scaling transformation, we turn a unit square into a parallelogram (Figure 12) by stretching it along the diagonal from (0, 0) to (1, 1). We first rotate the diagonal onto the y axis using $\theta = 45^\circ$, then we double its length with the scaling values $s_1 = 1$ and $s_2 = 2$, and then we rotate again to return the diagonal to its original orientation.

In Equation 39, we assumed that scaling was to be performed relative to the origin. We could take this scaling operation one step further and concatenate the matrix with translation operators, so that the composite matrix would include parameters for the specification of a scaling fixed position.

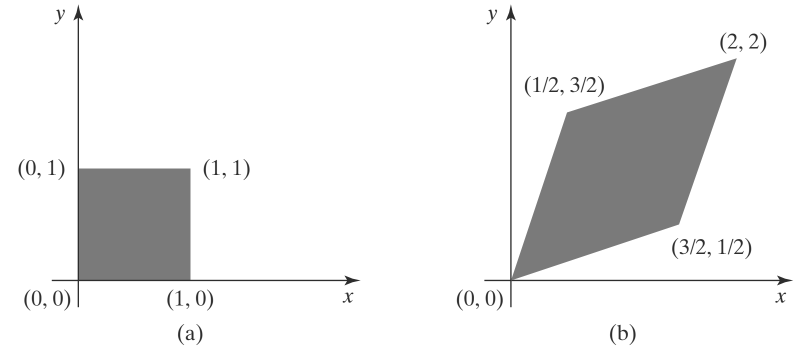


FIGURE 12
A square (a) is converted to a parallelogram (b) using the composite transformation matrix 39, with $s_1 = 1$, $s_2 = 2$, and $\theta = 45°$.

Matrix Concatenation Properties

Multiplication of matrices is associative. For any three matrices, $\mathbf{M}_1$, $\mathbf{M}_2$, and $\mathbf{M}_3$, the matrix product $\mathbf{M}_3 \cdot \mathbf{M}_2 \cdot \mathbf{M}_1$ can be performed by first multiplying $\mathbf{M}_3$ and $\mathbf{M}_2$ or by first multiplying $\mathbf{M}_2$ and $\mathbf{M}_1$:

$$\mathbf{M}_3 \cdot \mathbf{M}_2 \cdot \mathbf{M}_1 = (\mathbf{M}_3 \cdot \mathbf{M}_2) \cdot \mathbf{M}_1 = \mathbf{M}_3 \cdot (\mathbf{M}_2 \cdot \mathbf{M}_1) \tag{40}$$

Therefore, depending upon the order in which the transformations are specified, we can construct a composite matrix either by multiplying from left to right (premultiplying) or by multiplying from right to left (postmultiplying). Some graphics packages require that transformations be specified in the order in which they are to be applied. In that case, we would first invoke transformation $\mathbf{M}_1$, then $\mathbf{M}_2$, then $\mathbf{M}_3$. As each successive transformation routine is called, its matrix is concatenated on the left of the previous matrix product. Other graphics systems, however, postmultiply matrices, so that this transformation sequence would have to be invoked in the reverse order: the last transformation invoked (which is $\mathbf{M}_1$ for this example) is the first to be applied, and the first transformation that is called ($\mathbf{M}_3$ in this case) is the last to be applied.

Transformation products, on the other hand, may not be commutative. The matrix product $\mathbf{M}_2 \cdot \mathbf{M}_1$ is not equal to $\mathbf{M}_1 \cdot \mathbf{M}_2$, in general. This means that if we want to translate and rotate an object, we must be careful about the order in which the composite matrix is evaluated (Figure 13). For some special cases—such as a sequence of transformations that are all of the same kind—the multiplication of transformation matrices is commutative. As an example, two successive rotations could be performed in either order and the final position would be the same. This commutative property holds also for two successive translations or two successive scalings. Another commutative pair of operations is rotation and uniform scaling ($s_x = s_y$).

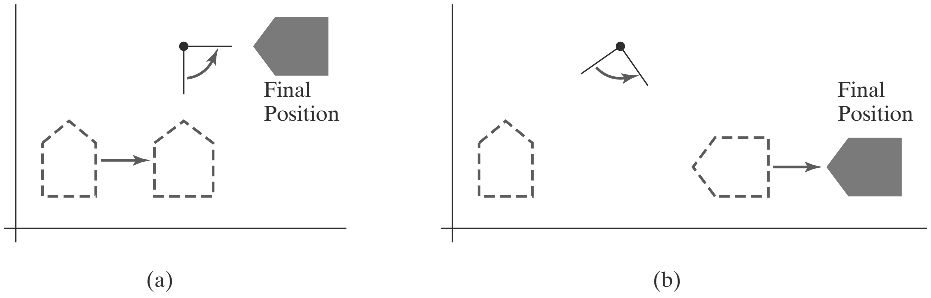


FIGURE 13
Reversing the order in which a sequence of transformations is performed may affect the transformed position of an object. In (a), an object is first translated in the x direction, then rotated counterclockwise through an angle of 45°. In (b), the object is first rotated 45° counterclockwise, then translated in the x direction.

General Two-Dimensional Composite Transformations and Computational Efficiency

A two-dimensional transformation, representing any combination of translations, rotations, and scalings, can be expressed as

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} rs_{xx} & rs_{xy} & trs_x \\ rs_{yx} & rs_{yy} & trs_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{41}$$

The four elements rs_{jk} are the multiplicative rotation-scaling terms in the transformation, which involve only rotation angles and scaling factors. Elements trs_x and trs_y are the translational terms, containing combinations of translation distances, pivot-point and fixed-point coordinates, rotation angles, and scaling parameters. For example, if an object is to be scaled and rotated about its centroid coordinates (x_c, y_c) and then translated, the values for the elements of the composite transformation matrix are

$$\begin{aligned} &\mathbf{T}(t_x, t_y) \cdot \mathbf{R}(x_c, y_c, \theta) \cdot \mathbf{S}(x_c, y_c, s_x, s_y) \\ &= \begin{bmatrix} s_x \cos\theta & -s_y \sin\theta & x_c(1 - s_x \cos\theta) + y_c s_y \sin\theta + t_x \\ s_x \sin\theta & s_y \cos\theta & y_c(1 - s_y \cos\theta) - x_c s_x \sin\theta + t_y \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{42}$$

Although Equation 41 requires nine multiplications and six additions, the explicit calculations for the transformed coordinates are

$$x' = x \cdot rs_{xx} + y \cdot rs_{xy} + trs_x, \qquad y' = x \cdot rs_{yx} + y \cdot rs_{yy} + trs_y \tag{43}$$

Thus, we need actually perform only four multiplications and four additions to transform coordinate positions. This is the maximum number of computations required for any transformation sequence, once the individual matrices have been concatenated and the elements of the composite matrix evaluated. Without concatenation, the individual transformations would be applied one at a time, and the number of calculations could be increased significantly. An efficient implementation for the transformation operations, therefore, is to formulate transformation matrices, concatenate any transformation sequence, and calculate transformed coordinates using Equations 43. On parallel systems, direct matrix multiplications with the composite transformation matrix of Equation 41 can be equally efficient.

Because rotation calculations require trigonometric evaluations and several multiplications for each transformed point, computational efficiency can become an important consideration in rotation transformations. In animations and other applications that involve many repeated transformations and small rotation angles, we can use approximations and iterative calculations to reduce computations in the composite transformation equations. When the rotation angle is small, the trigonometric functions can be replaced with approximation values based on the first few terms of their power series expansions. For small-enough angles (less than 10°), $\cos\theta$ is approximately 1.0 and $\sin\theta$ has a value very close to the value of θ in radians. If we are rotating in small angular steps about the origin, for instance, we can set $\cos\theta$ to 1.0 and reduce transformation calculations at each step to two multiplications and two additions for each set of coordinates to be rotated. These rotation calculations are

$$x' = x - y \sin\theta, \qquad y' = x \sin\theta + y \tag{44}$$

where $\sin\theta$ is evaluated once for all steps, assuming the rotation angle does not change. The error introduced by this approximation at each step decreases as the rotation angle decreases; but even with small rotation angles, the accumulated error over many steps can become quite large. We can control the accumulated error by estimating the error in x' and y' at each step and resetting object positions when the error accumulation becomes too great. Some animation applications automatically reset object positions at fixed intervals, such as every 360° or every 180°.

Composite transformations often involve inverse matrices. For example, transformation sequences for general scaling directions and for some reflections and shears (Section 5) require inverse rotations. As we have noted, the inverse matrix representations for the basic geometric transformations can be generated with simple procedures. An inverse translation matrix is obtained by changing the signs of the translation distances, and an inverse rotation matrix is obtained by performing a matrix transpose (or changing the sign of the sine terms). These operations are much simpler than direct inverse matrix calculations.

Two-Dimensional Rigid-Body Transformation

If a transformation matrix includes only translation and rotation parameters, it is a **rigid-body transformation matrix.** The general form for a two-dimensional rigid-body transformation matrix is

$$\begin{bmatrix} r_{xx} & r_{xy} & tr_x \\ r_{yx} & r_{yy} & tr_y \\ 0 & 0 & 1 \end{bmatrix} \tag{45}$$

where the four elements r_{jk} are the multiplicative rotation terms, and the elements tr_x and tr_y are the translational terms. A rigid-body change in coordinate position is also sometimes referred to as a **rigid-motion** transformation. All angles and distances between coordinate positions are unchanged by the transformation. In addition, matrix 45 has the property that its upper-left 2×2 submatrix is an *orthogonal matrix*. This means that if we consider each row (or each column) of the submatrix as a vector, then the two row vectors (r_{xx}, r_{xy}) and (r_{yx}, r_{yy}) (or the two column vectors) form an orthogonal set of unit vectors. Such a set of vectors is also referred to as an *orthonormal* vector set. Each vector has unit length as follows:

$$r_{xx}^2 + r_{xy}^2 = r_{yx}^2 + r_{yy}^2 = 1 \tag{46}$$

and the vectors are perpendicular (their dot product is 0):

$$r_{xx}r_{yx} + r_{xy}r_{yy} = 0 \tag{47}$$

Therefore, if these unit vectors are transformed by the rotation submatrix, then the vector (r_{xx}, r_{xy}) is converted to a unit vector along the x axis and the vector (r_{yx}, r_{yy}) is transformed into a unit vector along the y axis of the coordinate system:

$$\begin{bmatrix} r_{xx} & r_{xy} & 0 \\ r_{yx} & r_{yy} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{xx} \\ r_{xy} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \tag{48}$$

$$\begin{bmatrix} r_{xx} & r_{xy} & 0 \\ r_{yx} & r_{yy} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{yx} \\ r_{yy} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \tag{49}$$

For example, the following rigid-body transformation first rotates an object through an angle θ about a pivot point (x_r, y_r) and then translates the object:

$$\mathbf{T}(t_x, t_y) \cdot \mathbf{R}(x_r, y_r, \theta) = \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r\sin\theta + t_x \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r\sin\theta + t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{50}$$

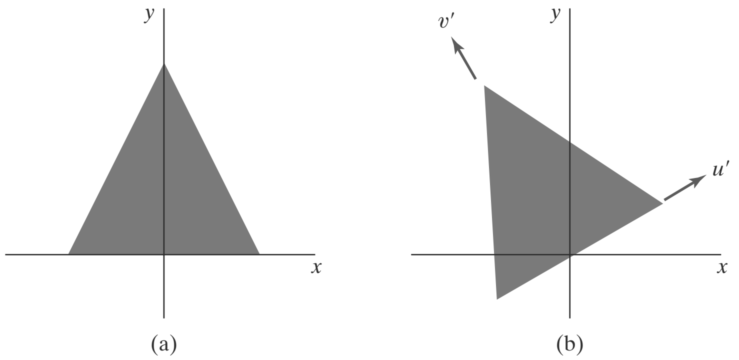


FIGURE 14
The rotation matrix for revolving an object from position (a) to position (b) can be constructed with the values of the unit orientation vectors **u**′ and **v**′ relative to the original orientation.

Here, orthogonal unit vectors in the upper-left 2 × 2 submatrix are $(\cos\theta, -\sin\theta)$ and $(\sin\theta, \cos\theta)$, and

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta \\ -\sin\theta \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \tag{51}$$

Similarly, unit vector $(\sin\theta, \cos\theta)$ is converted by the preceding transformation matrix to the unit vector (0, 1) in the y direction.

Constructing Two-Dimensional Rotation Matrices

The orthogonal property of rotation matrices is useful for constructing the matrix when we know the final orientation of an object, rather than the amount of angular rotation necessary to put the object into that position. This orientation information could be determined by the alignment of certain objects in a scene or by reference positions within the coordinate system. For example, we might want to rotate an object to align its axis of symmetry with the viewing (camera) direction, or we might want to rotate one object so that it is above another object. Figure 14 shows an object that is to be aligned with the unit direction vectors **u**′ and **v**′. Assuming that the original object orientation, as shown in Figure 14(a), is aligned with the coordinate axes, we construct the desired transformation by assigning the elements of **u**′ to the first row of the rotation matrix and the elements of **v**′ to the second row. In a modeling application, for instance, we can use this method to obtain the transformation matrix within an object's local coordinate system when we know what its orientation is to be within the overall world-coordinate scene. A similar transformation is the conversion of object descriptions from one coordinate system to another, and we take up these methods in more detail in Section 8.

Two-Dimensional Composite-Matrix Programming Example

An implementation example for a sequence of geometric transformations is given in the following program. Initially, the composite matrix, `compMatrix`, is constructed as the identity matrix. For this example, a left-to-right concatenation order is used to construct the composite transformation matrix, and we invoke the transformation routines in the order that they are to be executed. As each of the basic transformation routines (scale, rotate, and translate) is invoked, a matrix is set up for that transformation and left-concatenated with the composite matrix. When all transformations have been specified, the composite transformation is applied to transform a triangle. The triangle is first scaled with respect to its centroid position, then the triangle is rotated about its centroid, and, lastly, it is translated. Figure 15 shows the original and final positions of the triangle that is transformed by this sequence. Routines in OpenGL are used to dispaly the initial and final position of the triangle.

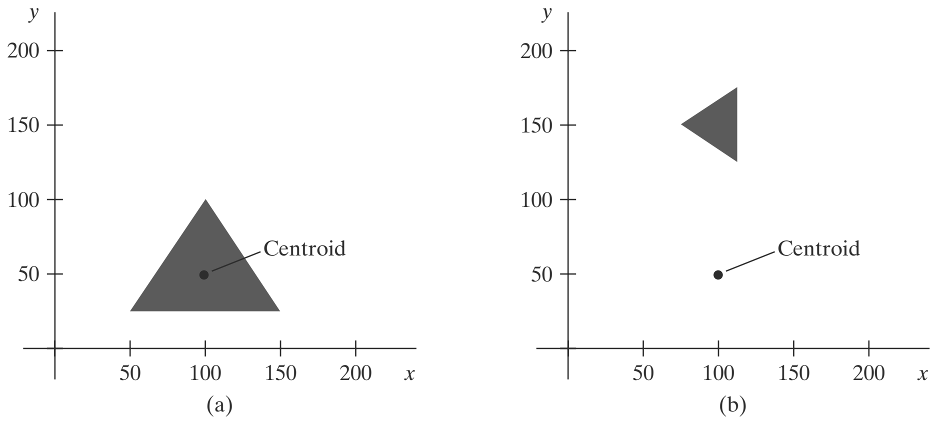


FIGURE 15
A triangle (a) is transformed into position (b) using the composite-matrix calculations in procedure `transformVerts2D`.

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

/*  Set initial display-window size. */
GLsizei winWidth = 600, winHeight = 600;

/*  Set range for world coordinates.  */
GLfloat xwcMin = 0.0, xwcMax = 225.0;
GLfloat ywcMin = 0.0, ywcMax = 225.0;

class wcPt2D {
   public:
      GLfloat x, y;
};

typedef GLfloat Matrix3x3 [3][3];

Matrix3x3 matComposite;

const GLdouble pi = 3.14159;

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

/* Construct the 3 x 3 identity matrix. */
void matrix3x3SetIdentity (Matrix3x3 matIdent3x3)
{
   GLint row, col;

   for (row = 0; row < 3; row++)
      for (col = 0; col < 3; col++)
         matIdent3x3 [row][col] = (row == col);
}
```

```
/* Premultiply matrix m1 times matrix m2, store result in m2. */
void matrix3x3PreMultiply (Matrix3x3 m1, Matrix3x3 m2)
{
   GLint row, col;
   Matrix3x3 matTemp;

   for (row = 0; row < 3; row++)
      for (col = 0; col < 3 ; col++)
         matTemp [row][col] = m1 [row][0] * m2 [0][col] + m1 [row][1] *
                            m2 [1][col] + m1 [row][2] * m2 [2][col];

   for (row = 0; row < 3; row++)
      for (col = 0; col < 3; col++)
         m2 [row][col] = matTemp [row][col];
}

void translate2D (GLfloat tx, GLfloat ty)
{
   Matrix3x3 matTransl;

   /*  Initialize translation matrix to identity.  */
   matrix3x3SetIdentity (matTransl);

   matTransl [0][2] = tx;
   matTransl [1][2] = ty;

   /*  Concatenate matTransl with the composite matrix.  */
   matrix3x3PreMultiply (matTransl, matComposite);
}

void rotate2D (wcPt2D pivotPt, GLfloat theta)
{
   Matrix3x3 matRot;

   /*  Initialize rotation matrix to identity.  */
   matrix3x3SetIdentity (matRot);

   matRot [0][0] = cos (theta);
   matRot [0][1] = -sin (theta);
   matRot [0][2] = pivotPt.x * (1 - cos (theta)) +
                        pivotPt.y * sin (theta);
   matRot [1][0] = sin (theta);
   matRot [1][1] = cos (theta);
   matRot [1][2] = pivotPt.y * (1 - cos (theta)) -
                        pivotPt.x * sin (theta);

   /*  Concatenate matRot with the composite matrix.  */
   matrix3x3PreMultiply (matRot, matComposite);
}

void scale2D (GLfloat sx, GLfloat sy, wcPt2D fixedPt)
{
   Matrix3x3 matScale;
   /*  Set geometric transformation parameters.  */
   wcPt2D pivPt, fixedPt;
   pivPt = centroidPt;
   fixedPt = centroidPt;

   GLfloat tx = 0.0, ty = 100.0;
   GLfloat sx = 0.5, sy = 0.5;
   GLdouble theta = pi/2.0;

   glClear (GL_COLOR_BUFFER_BIT);   //  Clear display window.

   glColor3f (0.0, 0.0, 1.0);       //  Set initial fill color to blue.
   triangle (verts);                //  Display blue triangle.

   /*  Initialize composite matrix to identity.  */
   matrix3x3SetIdentity (matComposite);

   /*  Construct composite matrix for transformation sequence.  */
   scale2D (sx, sy, fixedPt);   //  First transformation: Scale.
   rotate2D (pivPt, theta);     //  Second transformation: Rotate
   translate2D (tx, ty);        //  Final transformation: Translate.

   /*  Apply composite matrix to triangle vertices.  */
   transformVerts2D (nVerts, verts);

   glColor3f (1.0, 0.0, 0.0);  // Set color for transformed triangle.
   triangle (verts);           // Display red transformed triangle.

   glFlush ( );
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ( );
   gluOrtho2D (xwcMin, xwcMax, ywcMin, ywcMax);

   glClear (GL_COLOR_BUFFER_BIT);
}

void main (int argc, char ** argv)
{
   glutInit (&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowPosition (50, 50);
   glutInitWindowSize (winWidth, winHeight);
   glutCreateWindow ("Geometric Transformation Sequence");

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

   glutMainLoop ( );
}
```

```
   /*  Initialize scaling matrix to identity.  */
   matrix3x3SetIdentity (matScale);

   matScale [0][0] = sx;
   matScale [0][2] = (1 - sx) * fixedPt.x;
   matScale [1][1] = sy;
   matScale [1][2] = (1 - sy) * fixedPt.y;

   /*  Concatenate matScale with the composite matrix.  */
   matrix3x3PreMultiply (matScale, matComposite);
}

/* Using the composite matrix, calculate transformed coordinates. */
void transformVerts2D (GLint nVerts, wcPt2D * verts)
{
   GLint k;
   GLfloat temp;

   for (k = 0; k < nVerts; k++) {
      temp = matComposite [0][0] * verts [k].x + matComposite [0][1] *
             verts [k].y + matComposite [0][2];
      verts [k].y = matComposite [1][0] * verts [k].x + matComposite [1][1] *
                  verts [k].y + matComposite [1][2];
         verts [k].x = temp;
   }
}

void triangle (wcPt2D *verts)
{
    GLint k;

    glBegin (GL_TRIANGLES);
       for (k = 0; k < 3; k++)
          glVertex2f (verts [k].x, verts [k].y);
    glEnd ( );
}

void displayFcn (void)
{
   /*  Define initial position for triangle.  */
   GLint nVerts = 3;
   wcPt2D verts [3] = { {50.0, 25.0}, {150.0, 25.0}, {100.0, 100.0} };

   /*  Calculate position of triangle centroid.  */
   wcPt2D centroidPt;

   GLint k, xSum = 0, ySum = 0;
   for (k = 0; k < nVerts;  k++) {
      xSum += verts [k].x;
      ySum += verts [k].y;
   }
   centroidPt.x = GLfloat (xSum) / GLfloat (nVerts);
   centroidPt.y = GLfloat (ySum) / GLfloat (nVerts);
```

5 Other Two-Dimensional Transformations

Basic transformations such as translation, rotation, and scaling are standard components of graphics libraries. Some packages provide a few additional transformations that are useful in certain applications. Two such transformations are reflection and shear.

Reflection

A transformation that produces a mirror image of an object is called a **reflection.** For a two-dimensional reflection, this image is generated relative to an **axis of reflection** by rotating the object 180° about the reflection axis. We can choose an axis of reflection in the xy plane or perpendicular to the xy plane. When the reflection axis is a line in the xy plane, the rotation path about this axis is in a plane perpendicular to the xy plane. For reflection axes that are perpendicular to the xy plane, the rotation path is in the xy plane. Some examples of common reflections follow.

Reflection about the line $y = 0$ (the x axis) is accomplished with the transformation matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{52}$$

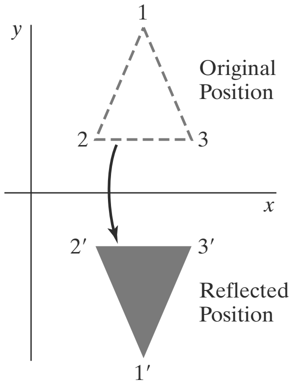


FIGURE 16
Reflection of an object about the x axis.

This transformation retains x values, but "flips" the y values of coordinate positions. The resulting orientation of an object after it has been reflected about the x axis is shown in Figure 16. To envision the rotation transformation path for this reflection, we can think of the flat object moving out of the xy plane and rotating 180° through three-dimensional space about the x axis and back into the xy plane on the other side of the x axis.

A reflection about the line $x = 0$ (the y axis) flips x coordinates while keeping y coordinates the same. The matrix for this transformation is

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{53}$$

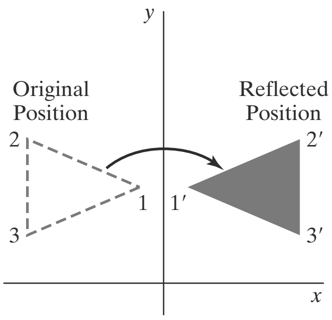


FIGURE 17
Reflection of an object about the y axis.

Figure 17 illustrates the change in position of an object that has been reflected about the line $x = 0$. The equivalent rotation in this case is 180° through three-dimensional space about the y axis.

We flip both the x and y coordinates of a point by reflecting relative to an axis that is perpendicular to the xy plane and that passes through the coordinate origin. This reflection is sometimes referred to as a reflection relative to the coordinate origin, and it is equivalent to reflecting with respect to both coordinate axes. The matrix representation for this reflection is

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{54}$$

An example of reflection about the origin is shown in Figure 18. The reflection matrix 54 is the same as the rotation matrix $\mathbf{R}(\theta)$ with $\theta = 180°$. We are simply rotating the object in the xy plane half a revolution about the origin.

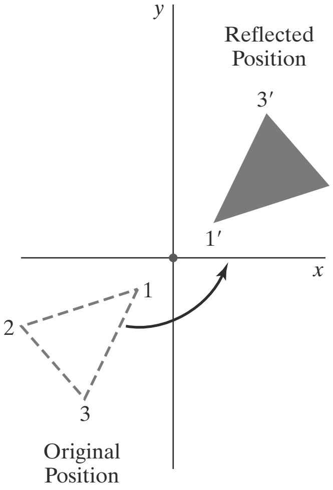


FIGURE 18
Reflection of an object relative to the coordinate origin. This transformation can be accomplished with a rotation in the xy plane about the coordinate origin.

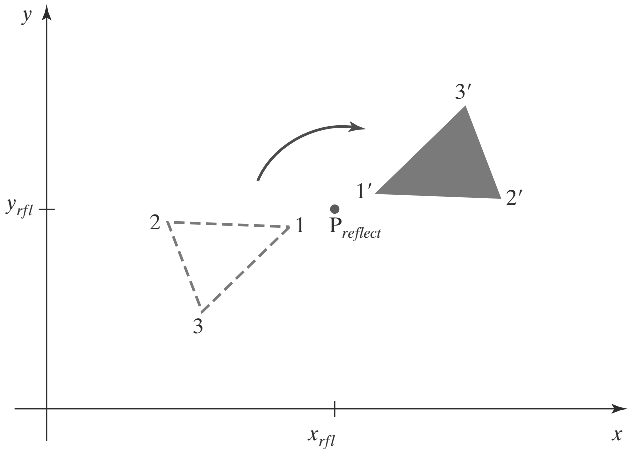


FIGURE 19
Reflection of an object relative to an axis perpendicular to the xy plane and passing through point $\mathbf{P}_{\text{reflect}}$.

Reflection 54 can be generalized to any reflection point in the xy plane (Figure 19). This reflection is the same as a 180° rotation in the xy plane about the reflection point.

If we choose the reflection axis as the diagonal line $y = x$ (Figure 20), the reflection matrix is

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{55}$$

We can derive this matrix by concatenating a sequence of rotation and coordinate-axis reflection matrices. One possible sequence is shown in Figure 21. Here, we first perform a clockwise rotation with respect to the origin through a 45° angle, which rotates the line $y = x$ onto the x axis. Next, we perform a reflection with respect to the x axis. The final step is to rotate the line $y = x$ back to its original position with a counterclockwise rotation through 45°. Another equivalent sequence of transformations is to first reflect the object about the x axis, then rotate it counterclockwise 90°.

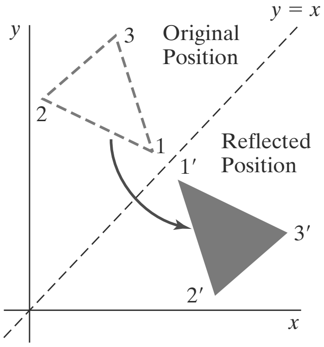


FIGURE 20
Reflection of an object with respect to the line $y = x$.

To obtain a transformation matrix for reflection about the diagonal $y = -x$, we could concatenate matrices for the transformation sequence: (1) clockwise rotation by 45°, (2) reflection about the y axis, and (3) counterclockwise rotation by 45°. The resulting transformation matrix is

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{56}$$

Figure 22 shows the original and final positions for an object transformed with this reflection matrix.

Reflections about any line $y = mx + b$ in the xy plane can be accomplished with a combination of translate-rotate-reflect transformations. In general, we first translate the line so that it passes through the origin. Then we can rotate the line onto one of the coordinate axes and reflect about that axis. Finally, we restore the line to its original position with the inverse rotation and translation transformations.

We can implement reflections with respect to the coordinate axes or coordinate origin as scaling transformations with negative scaling factors. Also, elements of the reflection matrix can be set to values other than ±1. A reflection parameter with a magnitude greater than 1 shifts the mirror image of a point farther from the reflection axis, and a parameter with magnitude less than 1 brings the mirror image of a point closer to the reflection axis. Thus, a reflected object can also be enlarged, reduced, or distorted.

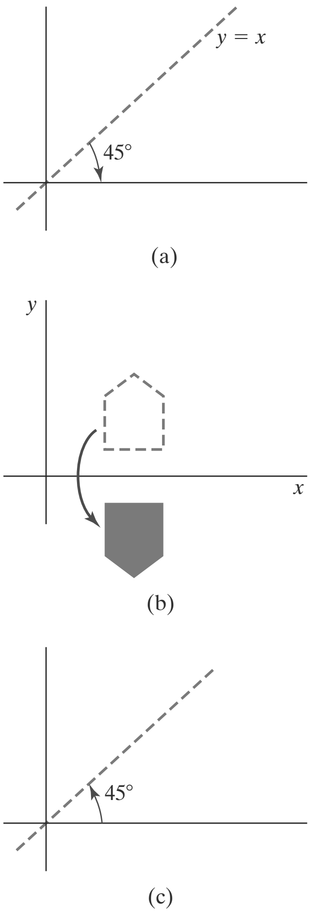


FIGURE 21
Sequence of transformations to produce a reflection about the line $y = x$: A clockwise rotation of 45° (a), a reflection about the x axis (b), and a counterclockwise rotation by 45° (c).

Shear

A transformation that distorts the shape of an object such that the transformed shape appears as if the object were composed of internal layers that had been caused to slide over each other is called a **shear.** Two common shearing transformations are those that shift coordinate x values and those that shift y values.

An x-direction shear relative to the x axis is produced with the transformation matrix

$$\begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{57}$$

which transforms coordinate positions as

$$x' = x + sh_x \cdot y, \qquad y' = y \tag{58}$$

Any real number can be assigned to the shear parameter sh_x. A coordinate position (x, y) is then shifted horizontally by an amount proportional to its perpendicular distance (y value) from the x axis. Setting parameter sh_x to the value 2, for example, changes the square in Figure 23 into a parallelogram. Negative values for sh_x shift coordinate positions to the left.

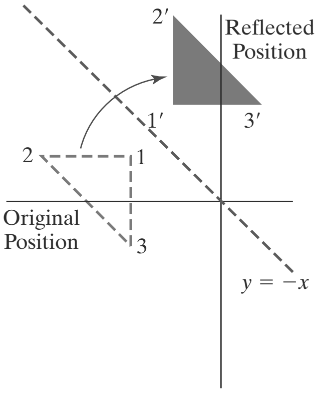


FIGURE 22
Reflection with respect to the line $y = -x$.

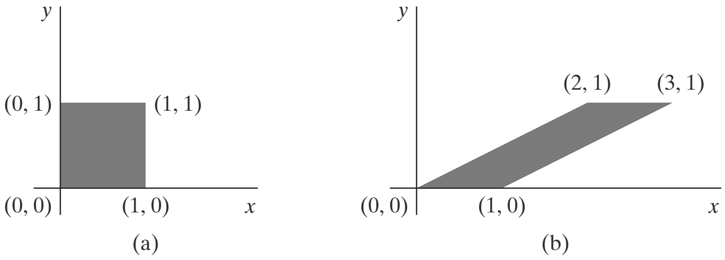


FIGURE 23
A unit square (a) is converted to a parallelogram (b) using the x-direction shear matrix 57 with $sh_x = 2$.

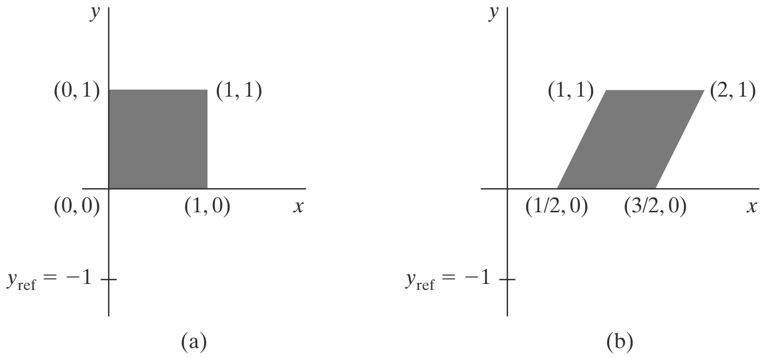


FIGURE 24
A unit square (a) is transformed to a shifted parallelogram (b) with $sh_x = 0.5$ and $y_{\text{ref}} = -1$ in the shear matrix 59.

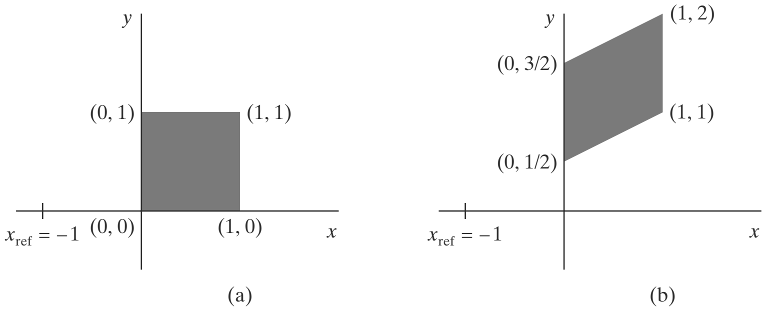


FIGURE 25
A unit square (a) is turned into a shifted parallelogram (b) with parameter values $sh_y = 0.5$ and $x_{\text{ref}} = -1$ in the y-direction shearing transformation 61.

We can generate x-direction shears relative to other reference lines with

$$\begin{bmatrix} 1 & sh_x & -sh_x \cdot y_{\text{ref}} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{59}$$

Now, coordinate positions are transformed as

$$x' = x + sh_x(y - y_{\text{ref}}), \qquad y' = y \tag{60}$$

An example of this shearing transformation is given in Figure 24 for a shear parameter value of $\frac{1}{2}$ relative to the line $y_{\text{ref}} = -1$.

A y-direction shear relative to the line $x = x_{\text{ref}}$ is generated with the transformation matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & -sh_y \cdot x_{\text{ref}} \\ 0 & 0 & 1 \end{bmatrix} \tag{61}$$

which generates the transformed coordinate values

$$x' = x, \qquad y' = y + sh_y(x - x_{\text{ref}}) \tag{62}$$

This transformation shifts a coordinate position vertically by an amount proportional to its distance from the reference line $x = x_{\text{ref}}$. Figure 25 illustrates the conversion of a square into a parallelogram with $sh_y = 0.5$ and $x_{\text{ref}} = -1$.

Shearing operations can be expressed as sequences of basic transformations. The x-direction shear matrix 57, for example, can be represented as a composite transformation involving a series of rotation and scaling matrices. This composite transformation scales the unit square of Figure 23 along its diagonal, while maintaining the original lengths and orientations of edges parallel to the x axis. Shifts in the positions of objects relative to shearing reference lines are equivalent to translations.

6 Raster Methods for Geometric Transformations

The characteristics of raster systems suggest an alternate method for performing certain two-dimensional transformations. Raster systems store picture information as color patterns in the frame buffer. Therefore, some simple object transformations can be carried out rapidly by manipulating an array of pixel values. Few arithmetic operations are needed, so the pixel transformations are particularly efficient.

Functions that manipulate rectangular pixel arrays are called *raster operations* and moving a block of pixel values from one position to another is termed a *block transfer,* a *bitblt,* or a *pixblt*. Routines for performing some raster operations are usually available in a graphics package.

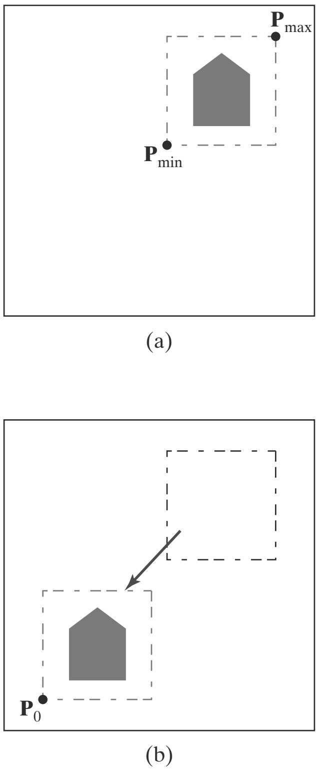


FIGURE 26
Translating an object from screen position (a) to the destination position shown in (b) by moving a rectangular block of pixel values. Coordinate positions $\mathbf{P}_{\min}$ and $\mathbf{P}_{\max}$ specify the limits of the rectangular block to be moved, and $\mathbf{P}_0$ is the destination reference position.

Figure 26 illustrates a two-dimensional translation implemented as a block transfer of a refresh-buffer area. All bit settings in the rectangular area shown are copied as a block into another part of the frame buffer. We can erase the pattern at the original location by assigning the background color to all pixels within that block (assuming that the pattern to be erased does not overlap other objects in the scene).

Rotations in 90-degree increments are accomplished easily by rearranging the elements of a pixel array. We can rotate a two-dimensional object or pattern 90° counterclockwise by reversing the pixel values in each row of the array, then interchanging rows and columns. A 180° rotation is obtained by reversing the order of the elements in each row of the array, then reversing the order of the rows. Figure 27 demonstrates the array manipulations that can be used to rotate a pixel block by 90° and by 180°.

For array rotations that are not multiples of 90°, we need to do some extra processing. The general procedure is illustrated in Figure 28. Each destination pixel area is mapped onto the rotated array and the amount of overlap with the rotated pixel areas is calculated. A color for a destination pixel can then be computed by averaging the colors of the overlapped source pixels, weighted by their percentage of area overlap. Alternatively, we could use an approximation method, as in antialiasing, to determine the color of the destination pixels.

We can use similar methods to scale a block of pixels. Pixel areas in the original block are scaled, using specified values for s_x and s_y, and then mapped onto a set of destination pixels. The color of each destination pixel is then assigned according to its area of overlap with the scaled pixel areas (Figure 29).

An object can be reflected using raster transformations that reverse row or column values in a pixel block, combined with translations. Shears are produced with shifts in the positions of array values along rows or columns.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \qquad \begin{bmatrix} 3 & 6 & 9 & 12 \\ 2 & 5 & 8 & 11 \\ 1 & 4 & 7 & 10 \end{bmatrix} \qquad \begin{bmatrix} 12 & 11 & 10 \\ 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

(a) (b) (c)

FIGURE 27
Rotating an array of pixel values. The original array is shown in (a), the positions of the array elements after a **90°** counterclockwise rotation are shown in (b), and the positions of the array elements after a **180°** rotation are shown in (c).

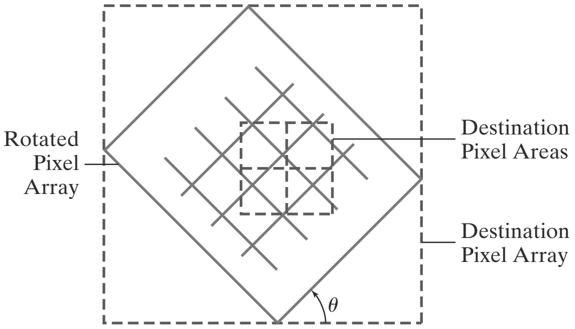


FIGURE 28
A raster rotation for a rectangular block of pixels can be accomplished by mapping the destination pixel areas onto the rotated block.

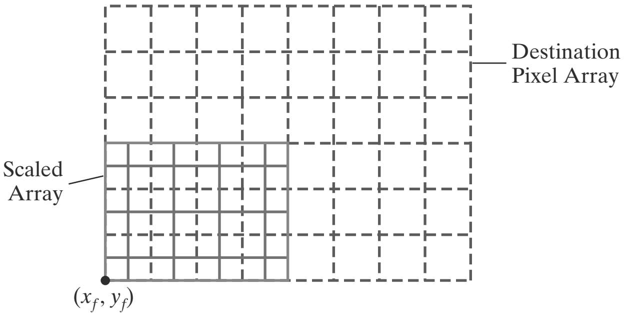


FIGURE 29
Mapping destination pixel areas onto a scaled array of pixel values. Scaling factors $s_x = s_y = 0.5$ are applied relative to fixed point (x_f, y_f).

7 OpenGL Raster Transformations

You should already be familiar with most of the OpenGL functions for performing raster operations. A translation of a rectangular array of pixel-color values from one buffer area to another can be accomplished in OpenGL as the following copy operation:

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

The first four parameters in this function give the location and dimensions of the pixel block; and the OpenGL symbolic constant GL_COLOR specifies that it is color values are to be copied. This array of pixels is to be copied to a rectangular area of a refresh buffer whose lower-left corner is at the location specified by the current raster position. Pixel-color values are copied as either RGBA values or color-table indices, depending on the current setting for the color mode. Both the region to be copied (the source) and the destination area should lie completely within the bounds of the screen coordinates. This translation can be carried out on any of the OpenGL buffers used for refreshing, or even between different buffers. A source buffer for the glCopyPixels function is chosen with the glReadBuffer routine, and a destination buffer is selected with the glDrawBuffer routine.

We can rotate a block of pixel-color values in 90-degree increments by first saving the block in an array, then rearranging the elements of the array and placing it back in the refresh buffer. A block of RGB color values in a buffer can be saved in an array with the function

```
glReadPixels (xmin, ymin, width, height, GL_RGB,
                    GL_UNSIGNED_BYTE, colorArray);
```

If color-table indices are stored at the pixel positions, we replace the constant GL_RGB with GL_COLOR_INDEX. To rotate the color values, we rearrange the rows and columns of the color array, as described in the previous section. Then we put the rotated array back in the buffer with

```
glDrawPixels (width, height, GL_RGB, GL_UNSIGNED_BYTE,
                    colorArray);
```

The lower-left corner of this array is placed at the current raster position. We select the source buffer containing the original block of pixel values with glReadBuffer, and we designate a destination buffer with glDrawBuffer.

A two-dimensional scaling transformation can be performed as a raster operation in OpenGL by specifying scaling factors and then invoking either `glCopyPixels` or `glDrawPixels`. For the raster operations, we set the scaling factors with

```
glPixelZoom (sx, sy);
```

where parameters `sx` and `sy` can be assigned any nonzero floating-point values. Positive values greater than 1.0 increase the size of an element in the source array, and positive values less than 1.0 decrease element size. A negative value for `sx` or `sy`, or both, produces a reflection and scales the array elements. Thus, if `sx` = `sy` = −3.0, the source array is reflected with respect to the current raster position and each color element of the array is mapped to a 3 × 3 block of destination pixels. If the center of a destination pixel lies within the rectangular area of a scaled color element of an array, it is assigned the color of that array element. Destination pixels whose centers are on the left or top boundary of the scaled array element are also assigned the color of that element. The default value for both `sx` and `sy` is 1.0.

We can also combine raster transformations with logical operations to produce various effects. With the *exclusive or* operator, for example, two successive copies of a pixel array to the same buffer area restores the values that were originally present in that area. This technique can be used in an animation application to translate an object across a scene without altering the background pixels.

8 Transformations between Two-Dimensional Coordinate Systems

Computer-graphics applications involve coordinate transformations from one reference frame to another during various stages of scene processing. The viewing routines transform object descriptions from world coordinates to device coordinates. For modeling and design applications, individual objects are typically defined in their own local Cartesian references. These local-coordinate descriptions must then be transformed into positions and orientations within the overall scene coordinate system. A facility-management program for office layouts, for instance, has individual coordinate descriptions for chairs and tables and other furniture that can be placed into a floor plan, with multiple copies of the chairs and other items in different positions.

Also, scenes are sometimes described in non-Cartesian reference frames that take advantage of object symmetries. Coordinate descriptions in these systems must be converted to Cartesian world coordinates for processing. Some examples of non-Cartesian systems are polar coordinates, spherical coordinates, elliptical coordinates, and parabolic coordinates. Here, we consider only the transformations involved in converting from one two-dimensional Cartesian frame to another.

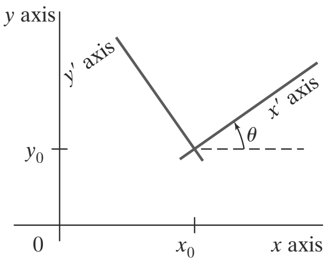


FIGURE 30
A Cartesian $x'y'$ system positioned at (x_0, y_0) with orientation θ in an xy Cartesian system.

Figure 30 shows a Cartesian $x'y'$ system specified with coordinate origin (x_0, y_0) and orientation angle θ in a Cartesian xy reference frame. To transform object descriptions from xy coordinates to $x'y'$ coordinates, we set up a transformation that superimposes the $x'y'$ axes onto the xy axes. This is done in two steps:

1. Translate so that the origin (x_0, y_0) of the $x'y'$ system is moved to the origin (0, 0) of the xy system.
2. Rotate the x' axis onto the x axis.

Translation of the coordinate origin is accomplished with the matrix transformation

$$\mathbf{T}(-x_0, -y_0) = \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{63}$$

The orientation of the two systems after the translation operation would then appear as in Figure 31. To get the axes of the two systems into coincidence, we then perform the clockwise rotation

$$\mathbf{R}(-\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{64}$$

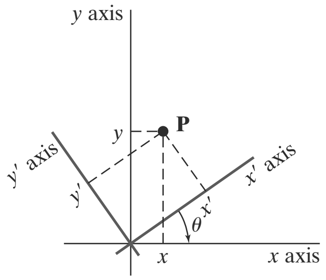


FIGURE 31
Position of the reference frames shown in Figure 30 after translating the origin of the $x'y'$ system to the coordinate origin of the xy system.

Concatenating these two transformation matrices gives us the complete composite matrix for transforming object descriptions from the xy system to the $x'y'$ system:

$$\mathbf{M}_{xy,x'y'} = \mathbf{R}(-\theta) \cdot \mathbf{T}(-x_0, -y_0) \tag{65}$$

An alternate method for describing the orientation of the $x'y'$ coordinate system is to specify a vector $\mathbf{V}$ that indicates the direction for the positive y' axis, as shown in Figure 32. We can specify vector $\mathbf{V}$ as a point in the xy reference frame relative to the origin of the xy system, which we can convert to the unit vector

$$\mathbf{v} = \frac{\mathbf{V}}{|\mathbf{V}|} = (v_x, v_y) \tag{66}$$

We obtain the unit vector $\mathbf{u}$ along the x' axis by applying a 90° clockwise rotation to vector $\mathbf{v}$:

$$\begin{aligned} \mathbf{u} &= (v_y, -v_x) \\ &= (u_x, u_y) \end{aligned} \tag{67}$$

In Section 4, we noted that the elements of any rotation matrix could be expressed as elements of a set of orthonormal vectors. Therefore, the matrix to rotate the $x'y'$ system into coincidence with the xy system can be written as

$$R = \begin{bmatrix} u_x & u_y & 0 \\ v_x & v_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{68}$$

For example, suppose that we choose the orientation for the y' axis as $\mathbf{V} = (-1, 0)$. Then the x' axis is in the positive y direction, and the rotation transformation matrix is

$$\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

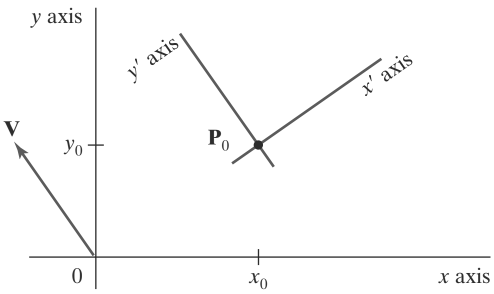


FIGURE 32
Cartesian system $x'y'$ with origin at $\mathbf{P}_0 = (x_0, y_0)$ and y' axis parallel to vector $\mathbf{V}$.

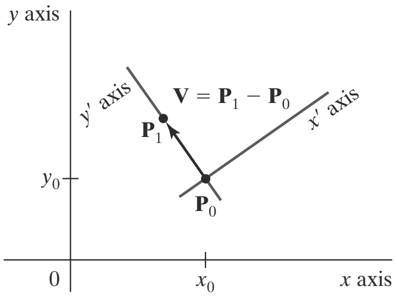


FIGURE 33
A Cartesian $x'y'$ system defined with two coordinate positions, P_0 and P_1, within an xy reference frame.

Equivalently, we can obtain this rotation matrix from Equation 64 by setting the orientation angle as $\theta = 90°$.

In an interactive application, it may be more convenient to choose the direction of **V** relative to position $\mathbf{P}_0$ than to specify it relative to the xy-coordinate origin. Unit vectors **u** and **v** would then be oriented as shown in Figure 33. The components of **v** are now calculated as

$$\mathbf{v} = \frac{\mathbf{P}_1 - \mathbf{P}_0}{|\mathbf{P}_1 - \mathbf{P}_0|} \tag{69}$$

and **u** is obtained as the perpendicular to **v** that forms a right-handed Cartesian system.

9 OpenGL Functions for Two-Dimensional Geometric Transformations

In the core library of OpenGL, a separate function is available for each of the basic geometric transformations. Because OpenGL is designed as a three-dimensional graphics application programming interface (API), all transformations are specified in three dimensions. Internally, all coordinate positions are represented as four-element column vectors, and all transformations are represented using 4 × 4 matrices. Fortunately, performing two-dimensional transformations within OpenGL is generally just a matter of using a value for the transformation in the third (z) dimension that causes no change in that dimension.

To perform a translation, we invoke the translation routine and set the components for the three-dimensional translation vector. In the rotation function, we specify the angle and the orientation for a rotation axis that intersects the coordinate origin. In addition, a scaling function is used to set the three coordinate scaling factors relative to the coordinate origin. In each case, the transformation routine sets up a 4 × 4 matrix that is applied to the coordinates of objects that are referenced after the transformation call.

Basic OpenGL Geometric Transformations

A 4 × 4 translation matrix is constructed with the following routine:

```
glTranslate* (tx, ty, tz);
```

Translation parameters `tx`, `ty`, and `tz` can be assigned any real-number values, and the single suffix code to be affixed to this function is either `f` (float) or `d` (double). For two-dimensional applications, we set `tz` = 0.0; and a two-dimensional position is represented as a four-element column matrix with the z component equal to 0.0. The translation matrix generated by this function is used to transform positions of objects defined after this function is invoked. For example, we translate subsequently defined coordinate positions 25 units in the x direction and -10 units in the y direction with the statement

```
glTranslatef (25.0, -10.0, 0.0);
```

Similarly, a 4 × 4 rotation matrix is generated with

```
glRotate* (theta, vx, vy, vz);
```

where the vector `v` = (`vx`, `vy`, `vz`) can have any floating-point values for its components. This vector defines the orientation for a rotation axis that passes through the coordinate origin. If `v` is not specified as a unit vector, then it is normalized automatically before the elements of the rotation matrix are computed. The suffix code can be either `f` or `d`, and parameter `theta` is to be assigned a rotation angle in degrees, which the routine converts to radians for the trigonometric calculations. The rotation specified here will be applied to positions defined after this function call. Rotation in two-dimensional systems is rotation about the z axis, specified as a unit vector with x and y components of zero, and a z component of 1.0. For example, the statement

```
glRotatef (90.0, 0.0, 0.0, 1.0);
```

sets up the matrix for a 90° rotation about the z axis. We should note here that internally, this function generates a rotation matrix using *quaternions*. This method is more efficient when rotation is about an arbitrarily-specific axis.

We obtain a 4 × 4 scaling matrix with respect to the coordinate origin with the following routine:

```
glScale* (sx, sy, sz);
```

The suffix code is again either `f` or `d`, and the scaling parameters can be assigned any real-number values. Scaling in a two-dimensional system involves changes in the x and y dimensions, so a typical two-dimensional scaling operation has a z scaling factor of 1.0 (which causes no change in the z coordinate of positions). Because the scaling parameters can be any real-number value, this function will also generate reflections when negative values are assigned to the scaling parameters. For example, the following statement produces a matrix that scales by a factor of 2 in the x direction, scales by a factor of 3 in the y direction, and reflects with respect to the x axis:

```
glScalef (2.0, -3.0, 1.0);
```

A zero value for any scaling parameter can cause a processing error because an inverse matrix cannot be calculated. The scale-reflect matrix is applied to subsequently defined objects.

It is important to note that internally OpenGL uses composite matrices to hold transformations. As a result, transformations are cumulative—that is, if we apply a translation and then apply a rotation, objects whose positions are specified after that will have both transformations applied to them. If that is not the behavior we desired, we must be able to remove the effects of previous transformations. This requires additional functions for manipulating the composite matrices.

OpenGL Matrix Operations

The `glMatrixMode` routine is used to set the *projection mode,* which designates the matrix that is to be used for the projection transformation. This transformation. This transformation determines how a scene is to be projected onto the screen. We use the same routine to set up a matrix for the geometric transformations. In this case, however, the matrix is referred to as the *modelview matrix*, and it is used to store and combine the geometric transformations. It is also used to combine the geometric transformations with the transformation to a viewing-coordinate system. We specify the *modelview mode* with the statement

```
glMatrixMode (GL_MODELVIEW);
```

which designates the 4 × 4 modelview matrix as the **current matrix.** The OpenGL transformation routines discussed in the previous section are all applied to whatever composite matrix is the current matrix, so it is important to use `glMatrixMode` to change to the modelview matrix before applying geometric transformations. Following this call, OpenGL transformation routines are used to modify the modelview matrix, which is then applied to transform coordinate positions in a scene. Two other modes that we can set with the `glMatrixMode` function are the *texture mode* and the *color mode*. The texture matrix is used for mapping texture patterns to surfaces, and the color matrix is used to convert from one color model to another. We discuss viewing, projection, texture, and color transformations in later chapters. For now, we limit our discussion to the details of the geometric transformations. The default argument for the `glMatrixMode` function is `GL_MODELVIEW`.

Once we are in the modelview mode (or any other mode), a call to a transformation routine generates a matrix that is multiplied by the current matrix for that mode. In addition, we can assign values to the elements of the current matrix, and there are two functions in the OpenGL library for this purpose. With the following function, we assign the identity matrix to the current matrix:

```
glLoadIdentity ( );
```

Alternatively, we can assign other values to the elements of the current matrix using

```
glLoadMatrix* (elements16);
```

A single-subscripted, 16-element array of floating-point values is specified with parameter `elements16`, and a suffix code of either `f` or `d` is used to designate the data type. The elements in this array must be specified in *column-major* order. That is, we first list the four elements in the first column, and then we list the four elements in the second column, the third column, and finally the fourth column. To illustrate this ordering, we initialize the modelview matrix with the following code:

```
glMatrixMode (GL_MODELVIEW);

GLfloat elems [16];
GLint k;

for (k = 0; k < 16; k++)
   elems [k] = float (k);
glLoadMatrixf (elems);
```

which produces the matrix

$$\mathbf{M} = \begin{bmatrix} 0.0 & 4.0 & 8.0 & 12.0 \\ 1.0 & 5.0 & 9.0 & 13.0 \\ 2.0 & 6.0 & 10.0 & 14.0 \\ 3.0 & 7.0 & 11.0 & 15.0 \end{bmatrix}$$

We can also concatenate a specified matrix with the current matrix as follows:

```
glMultMatrix* (otherElements16);
```

Again, the suffix code is either `f` or `d`, and parameter `otherElements16` is a 16-element, single-subscripted array that lists the elements of some other matrix in column-major order. The current matrix is *postmultiplied* by the matrix specified in `glMultMatrix`, and this product replaces the current matrix. Thus, assuming that the current matrix is the modelview matrix, which we designate as **M**, then the updated modelview matrix is computed as

$$\mathbf{M} = \mathbf{M} \cdot \mathbf{M}'$$

where $\mathbf{M}'$ represents the matrix whose elements are specified by parameter `otherElements16` in the preceding `glMultMatrix` statement.

The `glMultMatrix` function can also be used to set up any transformation sequence with individually defined matrices. For example,

```
glMatrixMode (GL_MODELVIEW);

glLoadIdentity ( );       // Set current matrix to the identity.
glMultMatrixf (elemsM2);  // Postmultiply identity with matrix M2.
glMultMatrixf (elemsM1);  // Postmultiply M2 with matrix M1.
```

produces the following current modelview matrix:

$$\mathbf{M} = \mathbf{M}_2 \cdot \mathbf{M}_1$$

The first transformation to be applied in this sequence is the last one specified in the code. Thus, if we set up a transformation sequence in an OpenGL program, we can think of the individual transformations as being loaded onto a stack, so the last operation specified is the first one applied. This is not what actually happens, but the stack analogy may help you remember that in an OpenGL program, a transformation sequence is applied in the opposite order from which it is specified.

It is also important to keep in mind that OpenGL stores matrices in column-major order. In addition, a reference to a matrix element such as m_{jk} in OpenGL is a reference to the element in column j and row k. This is the reverse of the standard mathematical convention, where the row number is referenced first. However, we can usually avoid errors in row-column references by always specifying matrices in OpenGL as 16-element, single-subscript arrays and listing the elements in a column-major order.

OpenGL actually maintains a stack of composite matrices for each of the four modes that we can select with the glMatrixMode routine.

10 OpenGL Geometric-Transformation Programming Examples

In the following code segment, we apply each of the basic geometric transformations, one at a time, to a rectangle. Initially, the modelview matrix is the identity matrix and we display a blue rectangle. Next, we reset the current color to red, specify two-dimensional translation parameters, and display the red translated rectangle (Figure 34). Because we do not want to combine transformations, we next reset the current matrix to the identity. Then a rotation matrix is constructed and concatenated with the current matrix (the identity matrix). When the original rectangle is again referenced, it is rotated about the *z* axis and displayed in red (Figure 35). We repeat this process once more to generate the scaled and reflected rectangle shown in Figure 36.

```
glMatrixMode (GL_MODELVIEW);

glColor3f (0.0, 0.0, 1.0);
glRecti (50, 100, 200, 150);       // Display blue rectangle.

glColor3f (1.0, 0.0, 0.0);
glTranslatef (-200.0, -50.0, 0.0); // Set translation parameters.
glRecti (50, 100, 200, 150);       // Display red, translated rectangle.

glLoadIdentity ( );                // Reset current matrix to identity.
glRotatef (90.0, 0.0, 0.0, 1.0);   // Set 90-deg. rotation about z axis.
glRecti (50, 100, 200, 150);       // Display red, rotated rectangle.

glLoadIdentity ( );                // Reset current matrix to identity.
glScalef (-0.5, 1.0, 1.0);         // Set scale-reflection parameters.
glRecti (50, 100, 200, 150);       // Display red, transformed rectangle.
```

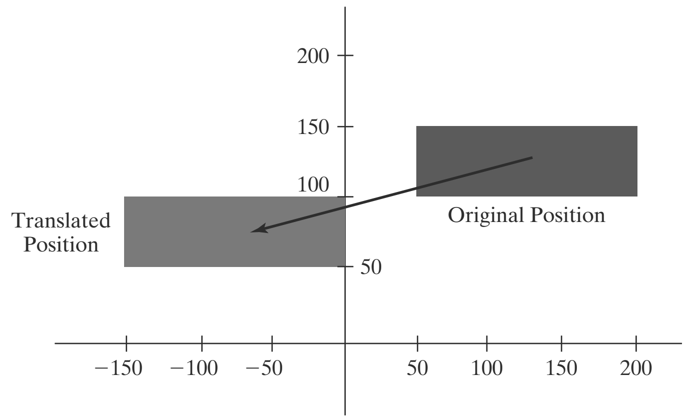


FIGURE 34
Translating a rectangle using the OpenGL function glTranslatef (−200.0, −50.0, 0.0).

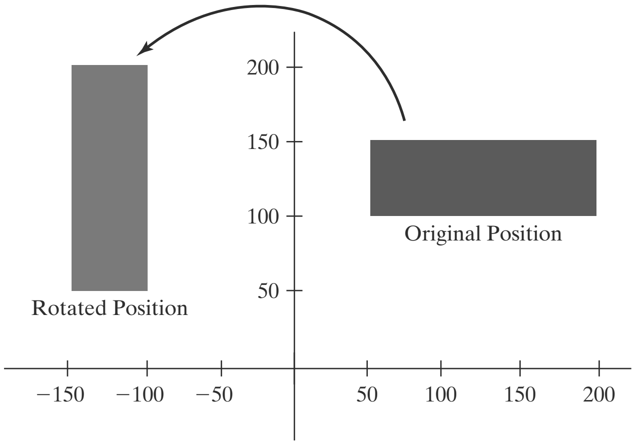


FIGURE 35
Rotating a rectangle about the *z* axis using the OpenGL function `glRotatef (90.0, 0.0, 0.0, 1.0)`.

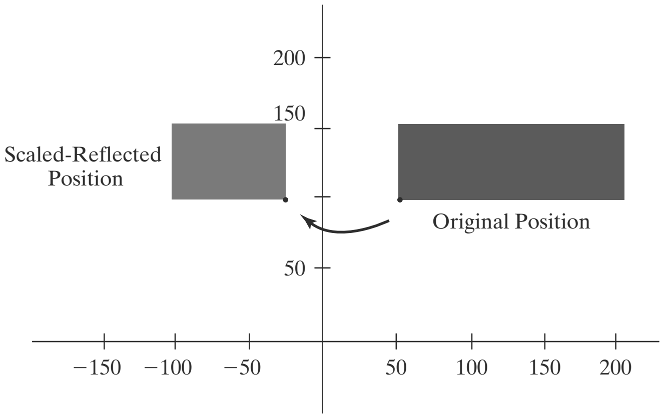


FIGURE 36
Scaling and reflecting a rectangle using the OpenGL function `glScalef (−0.5, 1.0, 1.0)`.

11 Summary

The basic geometric transformations are translation, rotation, and scaling. Translation moves an object in a straight-line path from one position to another. Rotation moves an object from one position to another along a circular path around a specified rotation axis. For two-dimensional applications, the rotation path is in the xy plane about an axis that is parallel to the z axis. Scaling transformations change the dimensions of an object relative to a fixed position.

We can express two-dimensional transformations as 3 × 3 matrix operators, so that sequences of transformations can be concatenated into a single composite matrix. Performing geometric transformations with matrices is an efficient formulation because it allows us to reduce computations by applying a composite matrix to an object description to obtain its transformed position. To do this, we express coordinate positions as column matrices. We choose a column-matrix representation for coordinate points because this is the standard mathematical convention, and most graphics packages now follow this convention. A three-element column matrix (vector) is referred to as a homogeneous-coordinate representation. For geometric transformations, the homogeneous coefficient is assigned the value 1.

As with two-dimensional systems, transformations between three-dimensional Cartesian coordinate systems are accomplished with a sequence of translate-rotate transformations that brings the two systems into coincidence.

TABLE 1

Summary of OpenGL Geometric Transformation Functions

Function	Description
`glTranslate*`	Specifies translation parameters.
`glRotate*`	Specifies parameters for rotation about any axis through the origin.
`glScale*`	Specifies scaling parameters with respect to coordinate origin.
`glMatrixMode`	Specifies current matrix for geometric-viewing transformations, projection transformations, texture transformations, or color transformations.
`glLoadIdentity`	Sets current matrix to identity.
`glLoadMatrix*` (elems);	Sets elements of current matrix.
`glMultMatrix*` (elems);	Postmultiplies the current matrix by the specified matrix.
`glPixelZoom`	Specifies two-dimensional scaling parameters for raster operations.

However, in a three-dimensional system, we must specify two of the three axis directions, not just one (as in a two-dimensional system).

The OpenGL basic library contains three functions for applying individual translate, rotate, and scale transformations to coordinate positions. Each function generates a matrix that is premultiplied by the modelview matrix. Thus, a sequence of geometric-transformation functions must be specified in reverse order: the last transformation invoked is the first to be applied to coordinate positions. Transformation matrices are applied to subsequently defined objects. In addition to accumulating transformation sequences in the modelview matrix, we can set this matrix to the identity or some other matrix. We can also form products with the modelview matrix and any specified matrices. Several operations are available in OpenGL for performing raster transformations. A block of pixels can be translated, rotated, scaled, or reflected with these OpenGL raster operations.

Table 1 summarizes the OpenGL geometric-transformation functions and matrix routines discussed in this chapter.

REFERENCES

For additional techniques involving matrices and geometric transformations, see Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994), and Paeth (1995). Discussions of homogeneous coordinates in computer graphics can be found in Blinn and Newell (1978) and in Blinn (1993, 1996, and 1998).

Additional programming examples using OpenGL geometric-transformation functions are given in Woo, et al. (1999). Programming examples for the OpenGL geometric-transformation functions are also available at Nate Robins's tutorial website: http://www.xmission.com/~nate/opengl.html. Finally, a complete listing of OpenGL geometric-transformation functions is provided in Shreiner (2000).

EXERCISES

1 Write an animation program that implements the example two-dimensional rotation procedure of Section 1. An input polygon is to be rotated repeatedly in small steps around a pivot point in the xy plane. Small angles are to be used for each successive step in the rotation, and approximations to the sine and cosine functions are to be used to speed up the calculations. To avoid excessive accumulation of round-off errors, reset the original coordinate values for the object at the start of each new revolution.

2 Show that the composition of two rotations is additive by concatenating the matrix representations for $\mathbf{R}(\theta_1)$ and $\mathbf{R}(\theta_2)$ to obtain

$$\mathbf{R}(\theta_1) \cdot \mathbf{R}(\theta_2) = \mathbf{R}(\theta_1 + \theta_2)$$

3 Modify the two-dimensional transformation matrix (39), for scaling in an arbitrary direction, to include coordinates for any specified scaling fixed point (x_f, y_f).

4 Prove that the multiplication of transformation matrices for each of the following sequences is commutative:

(a) Two successive rotations.

(b) Two successive translations.

(c) Two successive scalings.

5 Prove that a uniform scaling and a rotation form a commutative pair of operations but that, in general, scaling and rotation are not commutative operations.

6 Multiple the individual scale, rotate, and translate matrices in Equation 42 to verify the elements in the composite transformation matrix.

7 Modify the example program in Section 4 so that transformation parameters can be specified as user input.

8 Modify the program from the previous exercise so that the transformation sequence can be applied to any polygon, with vertices specified as user input.

9 Modify the example program in Section 4 so that the order of the geometric transformation sequence can be specified as user input.

10 Show that transformation matrix (55), for a reflection about the line $y = x$, is equivalent to a reflection relative to the x axis followed by a counterclockwise rotation of 90°.

11 Show that transformation matrix (56), for a reflection about the line $y = -x$, is equivalent to a reflection relative to the y axis followed by a counterclockwise rotation of 90°.

12 Show that two successive reflections about either the x axis or the y axis is equivalent to a single rotation in the xy plane about the coordinate origin.

13 Determine the form of the two-dimensional transformation matrix for a reflection about any line: $y = mx + b$.

14 Show that two successive reflections about any line in the xy plane that intersects the coordinate origin is equivalent to a rotation in the xy plane about the origin.

15 Determine a sequence of basic transformations that is equivalent to the x-direction shearing matrix (57).

16 Determine a sequence of basic transformations that is equivalent to the y-direction shearing matrix (61).

17 Set up a shearing procedure to display two-dimensional italic characters, given a vector font definition. That is, all character shapes in this font are defined with straight-line segments, and italic characters are formed with shearing transformations. Determine an appropriate value for the shear parameter by comparing italics and plain text in some available font. Define a simple vector font for input to your routine.

18 Derive the following equations for transforming a coordinate point $\mathbf{P} = (x, y)$ in one two-dimensional Cartesian system to the coordinate values (x', y') in another Cartesian system that is rotated counterclockwise by an angle θ relative to the first system. The transformation equations can be obtained by projecting point $\mathbf{P}$ onto each of the four axes and analyzing the resulting right triangles.

$$x' = x\cos\theta + y\sin\theta \qquad y' = -x\sin\theta + y\cos\theta$$

19 Write a procedure to compute the elements of the matrix for transforming object descriptions from one two-dimensional Cartesian coordinate system to another. The second coordinate system is to be defined with an origin point $\mathbf{P}_0$ and a vector $\mathbf{V}$ that gives the direction for the positive y' axis of this system.

20 Set up procedures for implementing a block transfer of a rectangular area of a frame buffer, using one function to read the area into an array and another function to copy the array into the designated transfer area.

21 Determine the results of performing two successive block transfers into the same area of a frame buffer using the various Boolean operations.

22 What are the results of performing two successive block transfers into the same area of a frame buffer using the binary arithmetic operations?

23 Implement a routine to perform block transfers in a frame buffer using any specified Boolean operation or a replacement (copy) operation.

24 Write a routine to implement rotations in increments of 90° in frame-buffer block transfers.

25 Write a routine to implement rotations by any specified angle in a frame-buffer block transfer.

26 Write a routine to implement scaling as a raster transformation of a pixel block.

27 Write a program to display an animation of a black square on a white background tracing a circular, clockwise path around the display window with the path's center at the display window's center (like the tip of the minute hand on a clock). The orientation of the square should not change. Use only basic OpenGL geometric transformations to do this.

28 Repeat the previous exercise using OpenGL matrix operations.

29 Modify the program in Exercise 27 to have the square rotate clockwise about its own center as it moves along its path. The square should complete one revolution about its center for each quarter of its path around the window that it completes. Use only basic OpenGL geometric transformations to do this.

30 Repeat the previous exercise using OpenGL matrix operations.

31 Modify the program in Exercise 29 to have the square additionally "pulse" as it moves along its path. That is, for every revolution about its own center that it makes, it should go through one pulse cycle that begins with the square at full size, reduces smoothly in size down to 50normal size by the end of the cycle. Do this using only basic OpenGL geometric transformations.

32 Repeat the previous exercise using only OpenGL matrix operations.

IN MORE DEPTH

1 In this exercise, you'll set up the routines necessary to make a crude animation of the objects in your application using two-dimensional geometric transformations. Decide on some simple motion behaviors for the objects in your application that can be achieved with the types of transformations discussed in this chapter (translations, rotations, scaling, shearing, and reflections). These behaviors may be motion patterns that certain objects will always be exhibiting, or they may be trajectories that are triggered or guided by user input (you can generate fixed example trajectories since we have not yet included user input). Set up the transformation matrices needed to produce these behaviors via composition of matrices. The matrices should be defined in homogeneous coordinates. If two or more objects act as a single "unit" in certain behaviors that are easier to model in terms of relative positions, you can use the techniques in Section 8 to convert the local transformations of the objects relative to each other (in their own coordinate frame) into transformations in the world coordinate frame.

2 Use the matrices you designed in the previous exercise to produce an animation of the behaviors of the objects in your scene. You should employ the OpenGL matrix operations and have the matrices produce small changes in position for each of the objects in the scene. The scene should then be redrawn several times per second to produce the animation, with the transformations being applied each time. Set the animation up so that it "loops"; that is, the behaviors should either be cyclical, or once the trajectories you designed for the objects have completed, the positions of all of the objects in the scene should be reset to their starting positions and the animation begun again.