hearn-50265; ISBN: 0-13-015390-7 *book* *August 8, 2003* *14:53*

CHAPTER 3

Graphics Output Primitives

A scene from the wolfman video. The animated figure of this primitive lycanthrope is modeled with 61 bones and eight layers of fur. Each frame of the computer animation contains 100,000 surface polygons. *(Courtesy of the NVIDIA Corporation.)*

hearn-50265; ISBN: 0-13-015390-7 *book* *July 30, 2003* *15:46*

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. As we noted in Section 2-8, the set of library functions can be subdivided into several categories. One of the first things we need to do when creating a picture is to describe the component parts of the scene to be displayed. Picture components could be trees and terrain, furniture and walls, storefronts and street scenes, automobiles and billboards, atoms and molecules, or stars and galaxies. For each type of scene, we need to describe the structure of the individual objects and their coordinate locations within the scene. Those functions in a graphics package that we use to describe the various picture components are called the **graphics output primitives,** or simply **primitives.** The output primitives describing the geometry of objects are typically referred to as **geometric primitives.** Point positions and straight-line segments are the simplest geometric primitives. Additional geometric primitives that can be available in a graphics package include circles and other conic sections, quadric surfaces, spline curves and surfaces, and polygon color areas. And 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 we also discuss the device-level algorithms for implementing the 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.

3-1 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

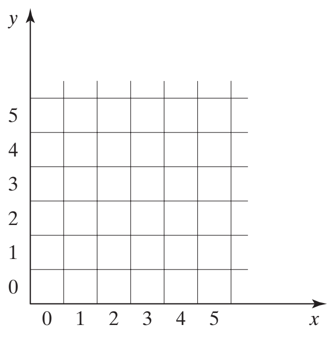


FIGURE 3-1 Pixel positions referenced with respect to the lower-left corner of a screen area.

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 (Fig. 3-1), or we could use noninteger Cartesian values for a picture description. The coordinate values we use to describe the geometry of a scene are then converted by the viewing routines to integer pixel positions within the frame buffer.

Scan-line algorithms for the graphics primitives use the defining coordinate descriptions to determine the locations of pixels that are to be displayed. For example, given the endpoint coordinates for a line segment, a display algorithm must calculate the positions for those pixels that lie along the line path between the endpoints. Since a pixel position occupies a finite area of the screen, the finite size of a pixel must be taken into account by the implementation algorithms. For the present, we assume that each integer screen position references the center of a pixel area. (In Section 3-13, we consider alternative pixel-addressing schemes.)

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 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.

3-2 SPECIFYING A TWO-DIMENSIONAL WORLD-COORDINATE REFERENCE FRAME IN OpenGL

In our first example program (Section 2-9), we introduced the `gluOrtho2D` command, which 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

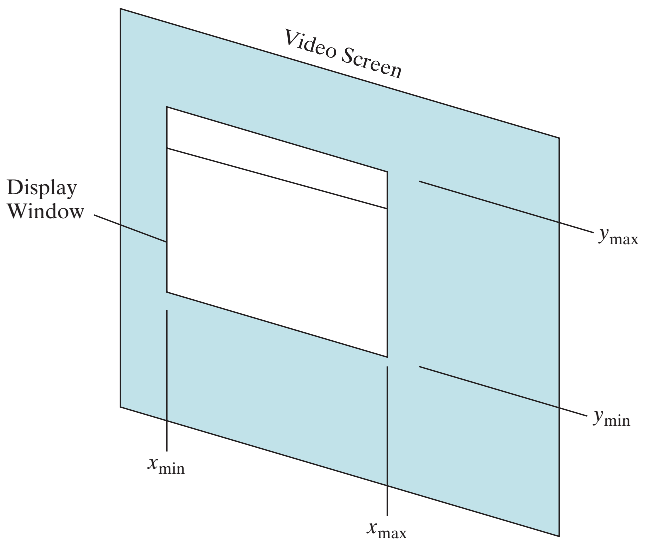


FIGURE 3-2
World-coordinate limits for a display window, as specified in the `glOrtho2D` function.

`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 Fig. 3-2.

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.

3-3 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 one screen pixel.

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

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. A `glVertex` function 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, and they are discussed in detail in Chapter 5. Since OpenGL treats two dimensions as a special case of three dimensions, any (x, y) coordinate specification is equivalent to $(x, y, 0)$ with $h = 1$.

We need to state also 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 a third suffix code: `v` (for "vector").

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

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

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

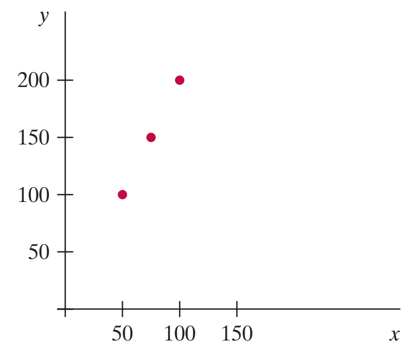


FIGURE 3-3 Display of three point positions generated with glBegin (GL_POINTS).

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 ( );
```

And 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 ( );
```

And we can use the OpenGL point-plotting functions within a C++ procedure to implement the `setPixel` command.

3-4 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. 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 Fig. 3-4(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.**

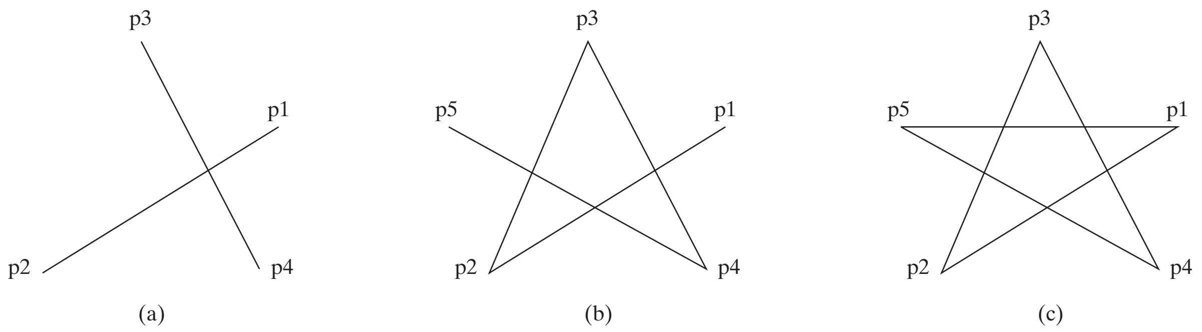


FIGURE 3-4 Line segments that can be displayed in OpenGL using a list of five endpoint coordinates. (a) An unconnected set of lines generated with the primitive line constant `GL_LINES`. (b) A polyline generated with `GL_LINE_STRIP`. (c) A closed polyline generated with `GL_LINE_LOOP`.

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. Using the same five coordinate positions as in the previous example, we obtain the display in Fig. 3-4(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.** An additional line is added to the line sequence from the previous example, so that the last coordinate endpoint in the sequence is connected to the first coordinate endpoint of the polyline. Figure 3-4(c) shows the display of our endpoint list when we select this line option.

```
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. In the next section, we take a look at the scan-conversion algorithms for implementing the OpenGL line functions.

3-5 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 (“the jaggies”), as represented in Fig. 3-5. 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

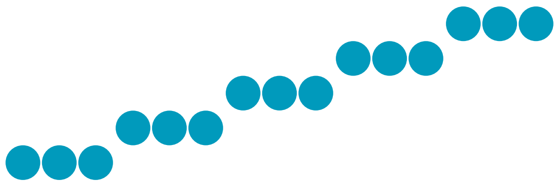


FIGURE 3-5 Stair-step effect (jaggies) produced when a line is generated as a series of pixel positions.

on high-resolution systems. More effective techniques for smoothing a raster line are based on adjusting pixel intensities along the line path (Section 4-17).

Line Equations

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

$$y = m \cdot x + b \tag{3-1}$$

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 Fig. 3-6, we can determine values for the slope m and y intercept b with the following calculations:

$$m = \frac{y_{\text{end}} - y_0}{x_{\text{end}} - x_0} \tag{3-2}$$

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

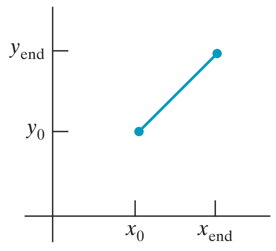


FIGURE 3-6 Line path between endpoint positions (x_0, y_0) and $(x_{\text{end}}, y_{\text{end}})$.

Algorithms for displaying straight lines are based on the line equation 3-1 and the calculations given in Eqs. 3-2 and 3-3.

For any given x interval δx along a line, we can compute the corresponding y interval δy from Eq. 3-2 as

$$\delta y = m \cdot \delta x \tag{3-4}$$

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

$$\delta x = \frac{\delta y}{m} \tag{3-5}$$

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 Eq. 3-4. 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 Eq. 3-5. 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 Fig. 3-7 with discrete sample positions along the x axis.

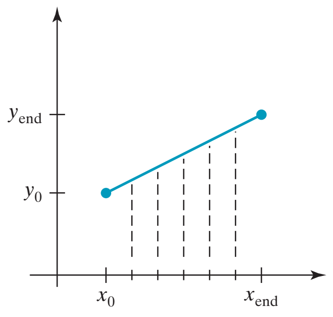


FIGURE 3-7 Straight-line segment with five sampling positions along the x axis between x_0 and x_{end}.

DDA Algorithm

The *digital differential analyzer* (DDA) is a scan-conversion line algorithm based on calculating either δy or δx, using Eq. 3-4 or Eq. 3-5. 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 Fig. 3-6. 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

$$y_{k+1} = y_k + m \tag{3-6}$$

Subscript k takes integer values starting from 0, for the first point, and increases by 1 until the final endpoint is reached. Since 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 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

$$x_{k+1} = x_k + \frac{1}{m} \tag{3-7}$$

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

Equations 3-6 and 3-7 are based on the assumption that lines are to be processed from the left endpoint to the right endpoint (Fig. 3-6). If this processing is reversed, so that the starting endpoint is at the right, then either we have $\delta x = -1$ and

$$y_{k+1} = y_k - m \tag{3-8}$$

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

$$x_{k+1} = x_k - \frac{1}{m} \tag{3-9}$$

Similar calculations are carried out using equations 3-6 through 3-9 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 Eq. 3-6. When the starting endpoint is at the right (for the same slope), we set $\delta x = -1$ and obtain y positions using Eq. 3-8. For a negative slope with absolute value greater than 1, we use $\delta y = -1$ and Eq. 3-9 or we use $\delta y = 1$ and Eq. 3-7.

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`. Starting with pixel position (`x0`, `y0`), we determine the offset needed at each step to generate the next pixel position along the line path. We loop through this process `steps` times. 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 $\frac{1}{m}$.

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

```
#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));
   }
}
```

The DDA algorithm is a faster method for calculating pixel positions than one that directly implements Eq. 3-1. It eliminates the multiplication in Eq. 3-1 by making use of 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 4-10. And in the next section, we consider a more general scan-line approach that can be applied to both lines and curves.

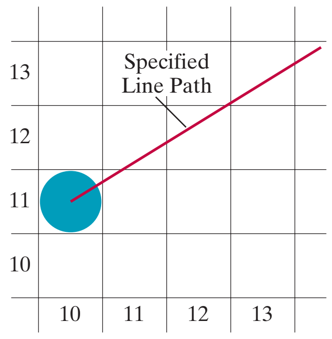


FIGURE 3-8 A section of a display screen where a straight-line segment is to be plotted, starting from the pixel at column 10 on scan line 11.

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 3-8 and 3-9 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 Fig. 3-8, 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, Fig 3-9 shows a negative-slope line path

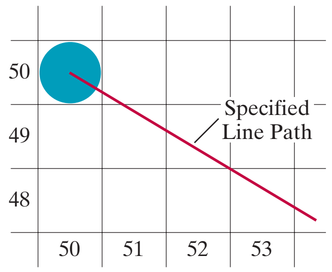


FIGURE 3-9 A section of a display screen where a negative slope line segment is to be plotted, starting from the pixel at column 50 on scan line 50.

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

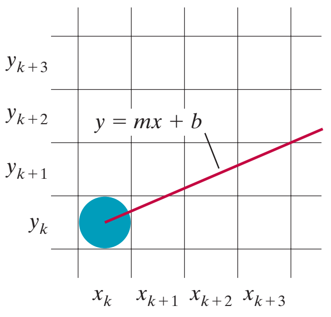


FIGURE 3-10 A section of the screen showing a pixel in column x_k on scan line y_k that is to be plotted along the path of a line segment with slope $0 < m < 1$.

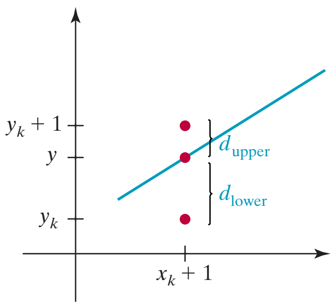


FIGURE 3-11 Vertical distances between pixel positions and the line y coordinate at sampling position $x_k + 1$.

starting from the left endpoint at pixel position (50, 50). In this one, do we select the next pixel position as (51, 50) or as (51, 49)? These questions are answered with Bresenham's line algorithm by testing the sign of an integer parameter whose value is proportional to the difference between the vertical separations of the two pixel positions from the actual line path.

To illustrate Bresenham's approach, we first consider the scan-conversion process for lines with positive slope less than 1.0. Pixel positions along a line path are then determined by sampling at unit x intervals. Starting from the left endpoint (x_0, y_0) of a given line, we step to each successive column (x position) and plot the pixel whose scan-line y value is closest to the line path. Figure 3-10 demonstrates the kth step in this process. Assuming we have determined that the pixel at (x_k, y_k) is to be displayed, we next need to decide which pixel to plot in column $x_{k+1} = x_k + 1$. Our choices are the pixels at positions $(x_k + 1, y_k)$ and $(x_k + 1, y_k + 1)$.

At sampling position $x_k + 1$, we label vertical pixel separations from the mathematical line path as d_{lower} and d_{upper} (Fig. 3-11). The y coordinate on the mathematical line at pixel column position $x_k + 1$ is calculated as

$$y = m(x_k + 1) + b \tag{3-10}$$

Then

$$\begin{aligned} d_{\text{lower}} &= y - y_k \\ &= m(x_k + 1) + b - y_k \end{aligned} \tag{3-11}$$

and

$$\begin{aligned} d_{\text{upper}} &= (y_k + 1) - y \\ &= y_k + 1 - m(x_k + 1) - b \end{aligned} \tag{3-12}$$

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:

$$d_{\text{lower}} - d_{\text{upper}} = 2m(x_k + 1) - 2y_k + 2b - 1 \tag{3-13}$$

A decision parameter p_k for the kth step in the line algorithm can be obtained by rearranging Eq. 3-13 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

$$\begin{aligned} p_k &= \Delta x(d_{\text{lower}} - d_{\text{upper}}) \\ &= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c \end{aligned} \tag{3-14}$$

The sign of p_k is the same as the sign of $d_{\text{lower}} - d_{\text{upper}}$, since $\Delta x > 0$ 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 directions. Therefore, we can obtain the values of successive decision parameters using incremental integer calculations. At step $k + 1$, the decision parameter is evaluated from Eq. 3-14 as

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

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

Subtracting Eq. 3-14 from the preceding equation, we have

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

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

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k) \tag{3-15}$$

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 Eq. 3-14 at the starting pixel position (x_0, y_0) and with m evaluated as $\Delta y/\Delta x$:

$$p_0 = 2\Delta y - \Delta x \tag{3-16}$$

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.

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. Perform step 4 $\Delta x - 1$ times.

EXAMPLE 3-1 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

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

hearn-50265; ISBN: 0-13-015390-7 *book* *July 30, 2003* *15:46*

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:

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)

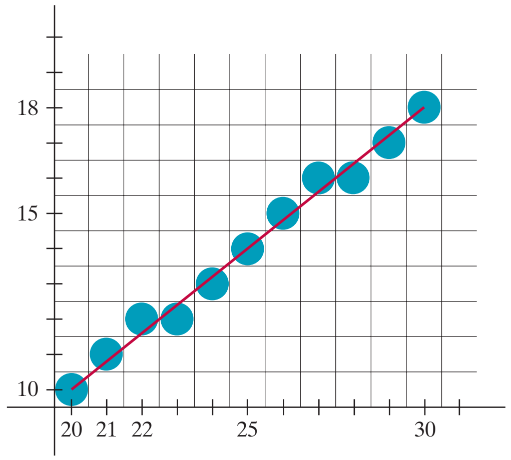


FIGURE 3-12 Pixel positions along the line path between endpoints (20, 10) and (30, 18), plotted with Bresenham's line algorithm.

A plot of the pixels generated along this line path is shown in Fig. 3-12. ■

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;
   }
```

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

```
      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 chose 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 overlap of displayed objects in more detail in Section 3-13.

3-6 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

$$\Delta x_p = \frac{\Delta x + n_p - 1}{n_p} \tag{3-17}$$

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

$$x_k = x_0 + k\Delta x_p \tag{3-18}$$

As an 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:

$$\Delta y_p = m\Delta x_p \tag{3-19}$$

At the kth partition, the starting y coordinate is then

$$y_k = y_0 + \text{round}(k\Delta y_p) \tag{3-20}$$

The initial decision parameter for Bresenham's algorithm at the start of the kth subinterval is obtained from Eq. 3-14:

$$p_k = (k\Delta x_p)(2\Delta y) - \text{round}(k\Delta y_p)(2\Delta x) + 2\Delta y - \Delta x \tag{3-21}$$

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

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

approach can be adapted to 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 $\Delta x \cdot \Delta y$ (Fig. 3-13). Perpendicular distance d from the line in Fig. 3-13 to a pixel with coordinates (x, y) is obtained with the calculation

$$d = Ax + By + C \tag{3-22}$$

where

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

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

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

with

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

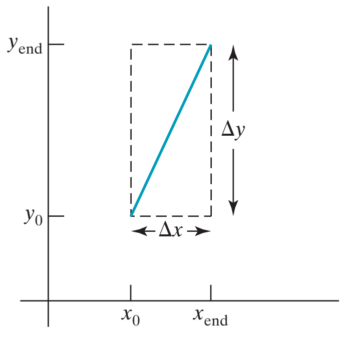


FIGURE 3-13 Bounding box for a line with endpoint separations Δx and Δy.

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.

3-7 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. Since 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 screen corner to $(x_{\max}, y_{\max})$ at the top-right corner (Fig. 3-14). For a bilevel system (one bit per pixel), the frame-buffer bit address for pixel position (x, y) is calculated as

$$\text{addr}(x, y) = \text{addr}(0, 0) + y(x_{\max} + 1) + x \tag{3-23}$$

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):

$$\text{addr}(x + 1, y) = \text{addr}(x, y) + 1 \tag{3-24}$$

Stepping diagonally up to the next scan line from (x, y), we get to the frame-buffer

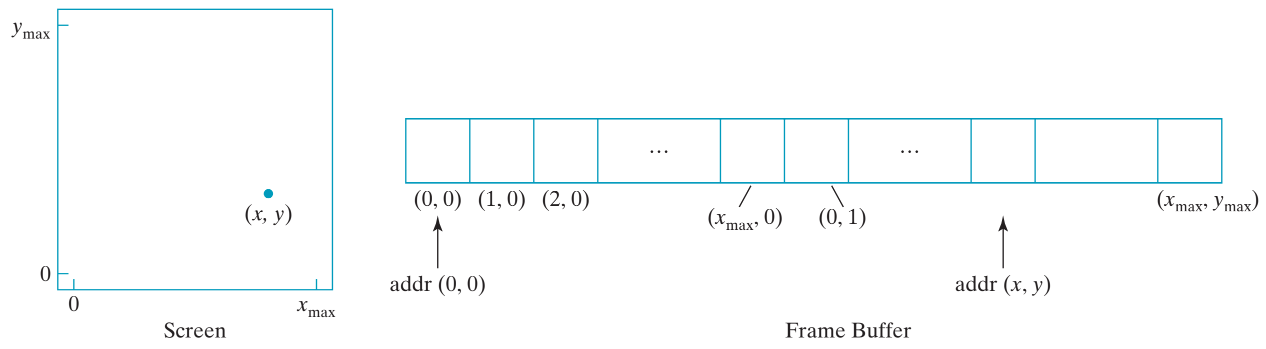


FIGURE 3-14 Pixel screen positions stored linearly in row-major order within the frame buffer.

address of $(x + 1, y + 1)$ with the calculation

$$\text{addr}(x + 1, y + 1) = \text{addr}(x, y) + x_{\text{max}} + 2 \tag{3-25}$$

where the constant $x_{\text{max}} + 2$ is precomputed once for all line segments. Similar incremental calculations can be obtained from Eq. 3-23 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.

3-8 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) has routines for three-dimensional quadrics, such as spheres and cylinders, as well as 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, so we defer further discussion of this group of functions until Chapter 8.

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, Fig. 3-15 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 the following sections. We first discuss efficient methods for circle and ellipse generation, then we take a look at procedures for displaying other conic sections, polynomials, and splines.

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

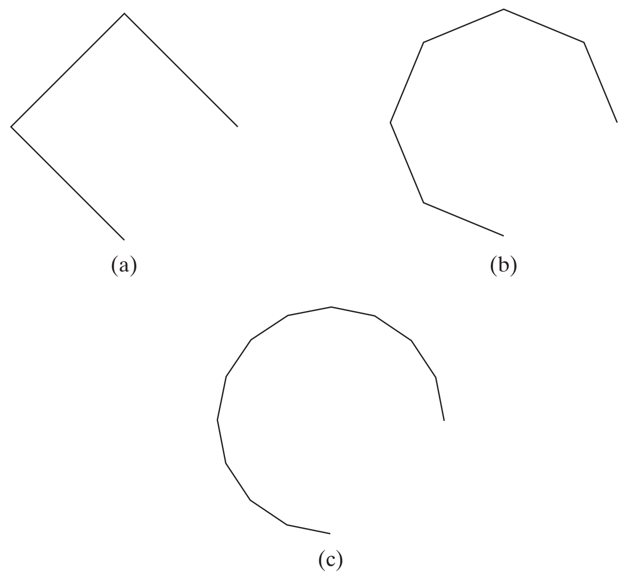


FIGURE 3-15 A circular arc approximated with (a) three straight-line segments, (b) six line segments, and (c) twelve line segments.

3-9 CIRCLE-GENERATING ALGORITHMS

Since 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. And sometimes a general function is available in a graphics library for displaying various kinds of curves, including circles and ellipses.

Properties of Circles

A circle (Fig. 3-16) 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

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \tag{3-26}$$

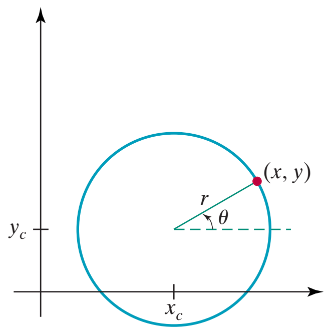


FIGURE 3-16 Circle with center coordinates (x_c, y_c) and radius r.

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

$$y = y_c \pm \sqrt{r^2 - (x_c - x)^2} \tag{3-27}$$

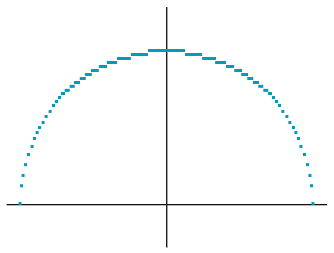


FIGURE 3-17 Upper half of a circle plotted with Eq. 3-27 and with $(x_c, y_c) = (0, 0)$.

But 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 Fig. 3-17. 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 Fig. 3-17 is to calculate points along the circular boundary using polar coordinates r and θ (Fig. 3-16). Expressing the circle equation in parametric polar form yields the pair of equations

$$x = x_c + r\cos\theta$$
$$y = y_c + r\sin\theta \tag{3-28}$$

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.

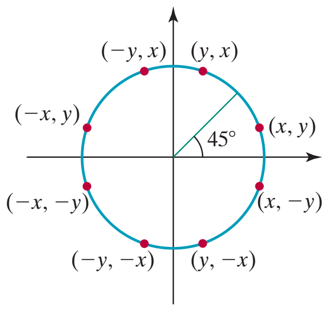


FIGURE 3-18 Symmetry of a circle. Calculation of a circle point (x, y) in one octant yields the circle points shown for the other seven octants.

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. And 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 Fig. 3-18, 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 Eq. 3-26 or Eq. 3-28 still requires a good deal of computation. The Cartesian equation 3-26 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 3-26, 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 more easily applied 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 one-half the pixel separation.

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

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

$$f_{\text{circ}}(x, y) = x^2 + y^2 - r^2 \tag{3-29}$$

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:

$$f_{\text{circ}}(x, y) \begin{cases} < 0, & \text{if } (x, y) \text{ is inside the circle boundary} \\ = 0, & \text{if } (x, y) \text{ is on the circle boundary} \\ > 0, & \text{if } (x, y) \text{ is outside the circle boundary} \end{cases} \tag{3-30}$$

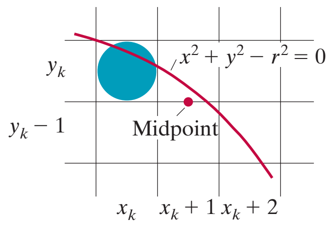


FIGURE 3-19 Midpoint between candidate pixels at sampling position $x_k + 1$ along a circular path.

The tests in 3-30 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 3-19 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 3-29 evaluated at the midpoint between these two pixels:

$$\begin{aligned} p_k &= f_{\text{circ}}\left(x_k + 1, y_k - \frac{1}{2}\right) \\ &= (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2 \end{aligned} \tag{3-31}$$

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

$$p_{k+1} = p_k + 2(x_k + 1) + \left(y_{k+1}^2 - y_k^2\right) - (y_{k+1} - y_k) + 1 \qquad (3\text{-}32)$$

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

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

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

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

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

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

or

$$p_0 = \frac{5}{4} - r \qquad (3\text{-}33)$$

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

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

since 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:
$$x = x + x_c, \qquad y = y + y_c$$
6. Repeat steps 3 through 5 until $x \geq y$.

EXAMPLE 3-2 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

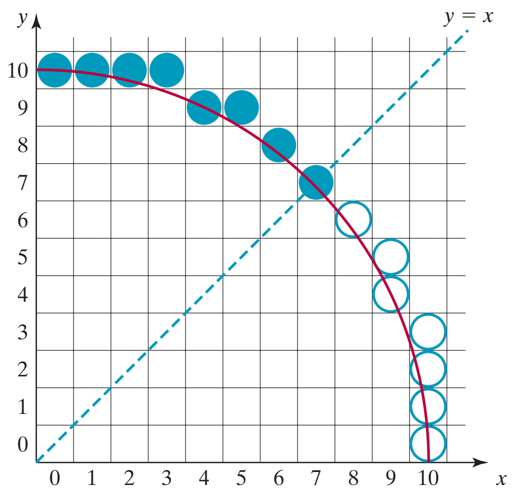


FIGURE 3-20 Pixel positions (solid circles) along a circle path centered on the origin and with radius $r = 10$, as calculated by the midpoint circle algorithm. Open ("hollow") circles show the symmetry positions in the first quadrant.

A plot of the generated pixel positions in the first quadrant is shown in Fig. 3-20. ■

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 coords for top point of circle.

    void circlePlotPoints (GLint, GLint, screenPt);
```

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

```
        /*  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 ( ));
    }
```

3-10 ELLIPSE-GENERATING ALGORITHMS

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

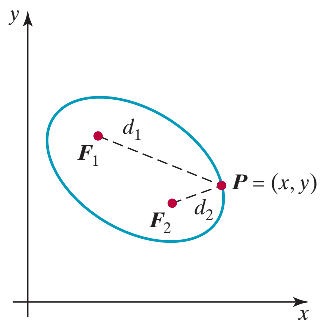


FIGURE 3-21 Ellipse generated about foci $\mathbf{F}_1$ and $\mathbf{F}_2$.

Properties of Ellipses

A precise definition of an ellipse can be given in terms of the distances from any point on the ellipse to two fixed positions, called the foci of the ellipse. The sum of these two distances is the same value for all points on the ellipse (Fig. 3-21). If the distances to the two focus positions from any point $\mathbf{P} = (x, y)$ on the ellipse are labeled d_1 and d_2, then the general equation of an ellipse can be stated as

$$d_1 + d_2 = \text{constant} \tag{3-34}$$

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

$$\sqrt{(x - x_1)^2 + (y - y_1)^2} + \sqrt{(x - x_2)^2 + (y - y_2)^2} = \text{constant} \tag{3-35}$$

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

$$A\,x^2 + B\,y^2 + C\,x\,y + D\,x + E\,y + F = 0 \tag{3-36}$$

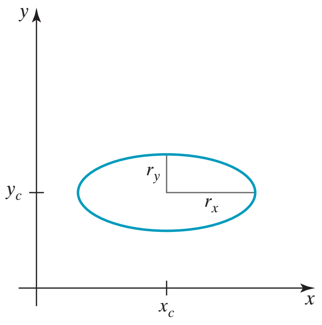


FIGURE 3-22 Ellipse centered at (x_c, y_c) with semimajor axis r_x and semiminor axis r_y.

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

An interactive method for specifying an ellipse in an arbitrary orientation is to input the two foci and a point on the ellipse boundary. With these three coordinate positions, we can evaluate the constant in Eq. 3-35. Then, the values for the coefficients in Eq. 3-36 can be computed and used to generate pixels along the elliptical path.

Ellipse equations are greatly simplified if the major and minor axes are oriented to align with the coordinate axes. In Fig. 3-22, 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 Fig. 3-22 can be written in terms of the ellipse center coordinates and parameters r_x and r_y as

$$\left(\frac{x - x_c}{r_x}\right)^2 + \left(\frac{y - y_c}{r_y}\right)^2 = 1 \tag{3-37}$$

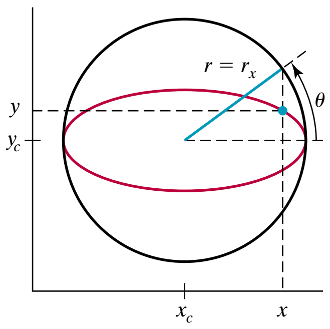


FIGURE 3-23 The bounding circle and eccentric angle θ for an ellipse with $r_x > r_y$.

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

$$\begin{aligned} x &= x_c + r_x \cos\theta \\ y &= y_c + r_y \sin\theta \end{aligned} \tag{3-38}$$

Angle θ, called the *eccentric angle* of the ellipse, is measured around the perimeter of a bounding circle. If $r_x > r_y$, the radius of the bounding circle is $r = r_x$ (Fig. 3-23). 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 (Fig. 3-24).

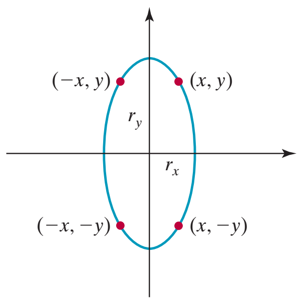


FIGURE 3-24 Symmetry of an ellipse. Calculation of a point (x, y) in one quadrant yields the ellipse points shown for the other three quadrants.

Midpoint Ellipse Algorithm

Our approach here is similar to that used in displaying a raster circle. Given parameters r_x, r_y, and (x_c, y_c), we determine curve positions (x, y) for an ellipse in standard position centered on the origin, then we shift all the points using a fixed offset so that the ellipse is centered at (x_c, y_c). If we wish also to display the ellipse in nonstandard position, we could rotate the ellipse about its center coordinates to reorient the major and minor axes in the desired directions. For the present, we consider only the display of ellipses in standard position. We discuss general methods for transforming object orientations and positions in Chapter 5.

The midpoint ellipse method is applied throughout the first quadrant in two parts. Figure 3-25 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.

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

Regions 1 and 2 (Fig. 3-25) can be processed in various ways. We can start at position $(0, r_y)$ and step clockwise along the elliptical path in the first quadrant, shifting from unit steps in x to unit steps in y when the slope becomes less than -1.0. Alternatively, we could start at $(r_x, 0)$ and select points in a counterclockwise order, shifting from unit steps in y to unit steps in x when the slope becomes greater than -1.0. With parallel processors, we could calculate pixel positions in the two regions simultaneously. As an example of a sequential implementation of the midpoint algorithm, we take the start position at $(0, r_y)$ and step along the ellipse path in clockwise order throughout the first quadrant.

We define an ellipse function from Eq. 3-37 with $(x_c, y_c) = (0, 0)$ as

$$f_{\text{ellipse}}(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2 \tag{3-39}$$

which has the following properties:

$$f_{\text{ellipse}}(x, y) \begin{cases} < 0, & \text{if } (x, y) \text{ is inside the ellipse boundary} \\ = 0, & \text{if } (x, y) \text{ is on the ellipse boundary} \\ > 0, & \text{if } (x, y) \text{ is outside the ellipse boundary} \end{cases} \tag{3-40}$$

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 (Fig. 3-25). 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 Eq. 3-39 as

$$\frac{dy}{dx} = -\frac{2r_y^2 x}{2r_x^2 y} \tag{3-41}$$

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

$$2r_y^2 x \geq 2r_x^2 y \tag{3-42}$$

Figure 3-26 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 3-39) at this midpoint:

$$\begin{aligned} p1_k &= f_{\text{ellipse}}\left(x_k + 1, y_k - \frac{1}{2}\right) \\ &= r_y^2(x_k + 1)^2 + r_x^2\left(y_k - \frac{1}{2}\right)^2 - r_x^2 r_y^2 \end{aligned} \tag{3-43}$$

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

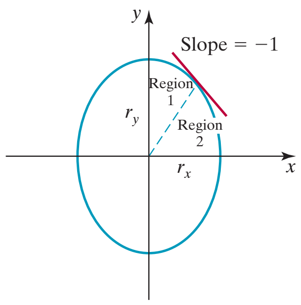


FIGURE 3-25 Ellipse processing regions. Over region 1, the magnitude of the ellipse slope is less than 1.0; over region 2, the magnitude of the slope is greater than 1.0.

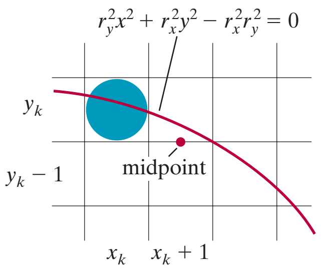


FIGURE 3-26 Midpoint between candidate pixels at sampling position $x_k + 1$ along an elliptical path.

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

$$p1_{k+1} = p1_k + 2r_y^2(x_k + 1) + r_y^2 + r_x^2\left[\left(y_{k+1} - \frac{1}{2}\right)^2 - \left(y_k - \frac{1}{2}\right)^2\right] \qquad (3\text{-}44)$$

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, since 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

$$2r_y^2 x = 0 \qquad (3\text{-}45)$$

$$2r_x^2 y = 2r_x^2 r_y \qquad (3\text{-}46)$$

As x and y are incremented, updated values are obtained by adding $2r_y^2$ to the current value of the increment term in Eq. 3-45 and subtracting $2r_x^2$ from the current value of the increment term in Eq. 3-46. The updated increment values are compared at each step, and we move from region 1 to region 2 when condition 3-42 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

$$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2 \qquad (3\text{-}47)$$

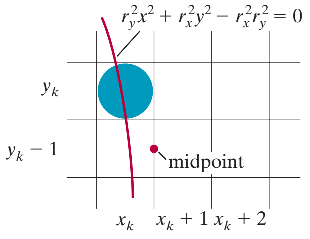


FIGURE 3-27 Midpoint between candidate pixels at sampling position $y_k - 1$ along an elliptical path.

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 (Fig. 3-27). For this region, the decision parameter is evaluated as

$$\begin{aligned} p2_k &= f_{\text{ellipse}}\left(x_k + \frac{1}{2},\, y_k - 1\right) \\ &= r_y^2\left(x_k + \frac{1}{2}\right)^2 + r_x^2(y_k - 1)^2 - r_x^2 r_y^2 \end{aligned} \qquad (3\text{-}48)$$

If $p2_k > 0$, the midposition is outside the ellipse boundary, and we select the pixel

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

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}.

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

$$\begin{aligned} p2_{k+1} &= f_{\text{ellipse}}\left(x_{k+1} + \frac{1}{2}, y_{k+1} - 1\right) \\ &= r_y^2\left(x_{k+1} + \frac{1}{2}\right)^2 + r_x^2[(y_k - 1) - 1]^2 - r_x^2 r_y^2 \end{aligned} \tag{3-49}$$

or

$$p2_{k+1} = p2_k - 2r_x^2(y_k - 1) + r_x^2 + r_y^2\left[\left(x_{k+1} + \frac{1}{2}\right)^2 - \left(x_k + \frac{1}{2}\right)^2\right] \tag{3-50}$$

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

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

$$\begin{aligned} p2_0 &= f_{\text{ellipse}}\left(x_0 + \frac{1}{2}, y_0 - 1\right) \\ &= r_y^2\left(x_0 + \frac{1}{2}\right)^2 + r_x^2(y_0 - 1)^2 - r_x^2 r_y^2 \end{aligned} \tag{3-51}$$

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 Eq. 3-36 and calculating pixel positions over the entire elliptical path. Alternatively, we could reorient the ellipse axes to standard position, using transformation methods discussed in Chapter 5, 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

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

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

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 the coordinate values:

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

EXAMPLE 3-3 Midpoint Ellipse Drawing

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

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

For region 1, the initial point for the ellipse centered on the origin is

$(x_0, y_0) = (0, 6)$, and the initial decision parameter value is

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

Successive midpoint decision-parameter values and the pixel positions along the ellipse are listed in the following table.

k	$p1_k$	(x_{k+1}, y_{k+1})	$2r_y^2 x_{k+1}$	$2r_x^2 y_{k+1}$
0	−332	(1, 6)	72	768
1	−224	(2, 6)	144	768
2	−44	(3, 6)	216	768
3	208	(4, 5)	288	640
4	−108	(5, 5)	360	640
5	288	(6, 4)	432	512
6	244	(7, 3)	504	384

We now move out of region 1, since $2r_y^2 x > 2r_x^2 y$.
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)	—	—

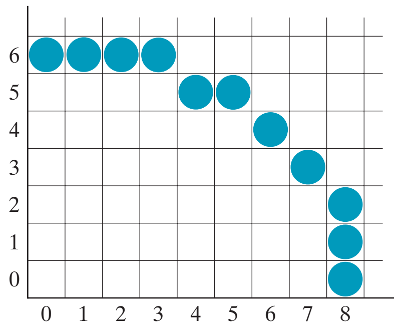


FIGURE 3-28 Pixel positions along an elliptical path centered on the origin with $r_x = 8$ and $r_y = 6$, using the midpoint algorithm to calculate locations within the first quadrant.

A plot of the calculated positions for the ellipse within the first quadrant is shown in Fig. 3-28. ■

In the following code segment, example procedures are given for implementing the midpoint ellipse algorithm. Values for the ellipse parameters `Rx`, `Ry`, `xCenter`, and `yCenter` are input to procedure `ellipseMidpoint`. Positions along the curve in the first quadrant are then calculated and passed to procedure `ellipsePlotPoints`. Symmetry is used to obtain ellipse positions in the other three quadrants, and the `setPixel` routine sets the ellipse color in the frame-buffer locations corresponding to these positions.

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

/*  The following procedure accepts values for an ellipse
 *  center position and its semimajor and semiminor axes, then
 *  calculates ellipse positions using the midpoint algorithm.
 */
void ellipseMidpoint (int xCenter, int yCenter, int Rx, int Ry)
{
   int Rx2 = Rx * Rx;
   int Ry2 = Ry * Ry;
   int twoRx2 = 2 * Rx2;
   int twoRy2 = 2 * Ry2;
   int p;
   int x = 0;
   int y = Ry;
   int px = 0;
   int py = twoRx2 * y;
   void ellipsePlotPoints (int, int, int, int);

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

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

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

```
void ellipsePlotPoints (int xCenter, int yCenter, int x, int y);
{
   setPixel (xCenter + x, yCenter + y);
   setPixel (xCenter - x, yCenter + y);
   setPixel (xCenter + x, yCenter - y);
   setPixel (xCenter - x, yCenter - y);
}
```

3-11 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

$$A\,x^2 + B\,y^2 + C\,x\,y + D\,x + E\,y + F = 0 \qquad (3\text{-}52)$$

where values for parameters A, B, C, D, E, and F determine the kind of curve we are to display. Given this set of coefficients, we can determine the particular

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

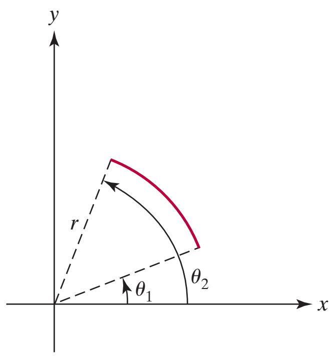


FIGURE 3-29 A circular arc, centered on the origin, defined with beginning angle θ_1, ending angle θ_2, and radius r.

conic that will be generated by evaluating the discriminant $B^2 - 4AC$:

$$B^2 - 4AC \begin{cases} < 0, & \text{generates an ellipse (or circle)} \\ = 0, & \text{generates a parabola} \\ > 0, & \text{generates a hyperbola} \end{cases} \tag{3-53}$$

For example, we get the circle equation 3-26 when $A = B = 1$, $C = 0$, $D = -2x_c$, $E = -2y_c$, and $F = x_c^2 + y_c^2 - r^2$. Equation 3-52 also describes the "degenerate" conics: points and straight lines.

In some applications, circular and elliptical arcs are conveniently specified with the beginning and ending angular values for the arc, as illustrated in Fig. 3-29. And 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 3-30 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

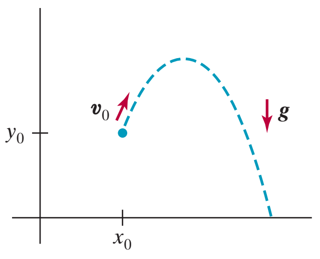


FIGURE 3-30 Parabolic path of an object tossed into a downward gravitational field at the initial position (x_0, y_0).

$$y = y_0 + a(x - x_0)^2 + b(x - x_0) \tag{3-54}$$

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

$$\begin{aligned} x &= x_0 + v_{x0}\, t \\ y &= y_0 + v_{y0}\, t - \frac{1}{2} g t^2 \end{aligned} \tag{3-55}$$

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 (Fig. 3-31) 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 Fig. 3-31) 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 Fig. 3-31 as

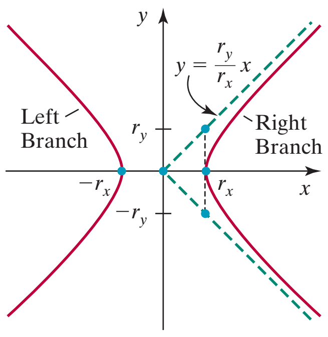


FIGURE 3-31 Left and right branches of a hyperbola in standard position with the symmetry axis along the x axis.

$$\left(\frac{x}{r_x}\right)^2 - \left(\frac{y}{r_y}\right)^2 = 1 \tag{3-56}$$

with $x \le -r_x$ for the left branch and $x \ge r_x$ for the right branch. Since this equation differs from the standard ellipse equation 3-39 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 Eq. 3-55 is symmetric about the axis

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

The methods used in the midpoint ellipse algorithm can be directly applied 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 Eq. 3-52 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 *n*th degree in *x* is defined as

$$\begin{aligned} y &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} + a_n x^n \end{aligned} \tag{3-57}$$

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

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

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 (Fig. 3-32). Continuous curves that are formed with polynomial pieces are called **spline curves,** or simply **splines.** There are other ways to set up spline curves, and various spline-generating methods are explored in Chapter 8.

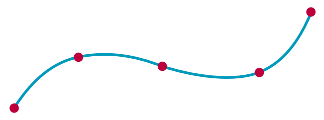


FIGURE 3-32 A spline curve formed with individual cubic polynomial sections between specified coordinate positions.

3-12 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.

3-13 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. But when the object is scan converted into the frame buffer, the input description is transformed to pixel coordinates which reference finite screen 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.

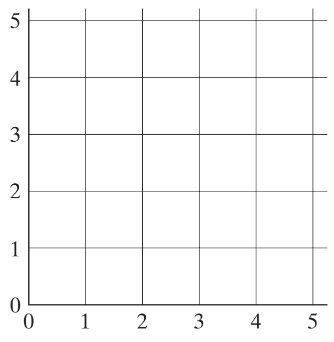


FIGURE 3-33 Lower-left section of a screen area with coordinate positions referenced by grid intersection lines.

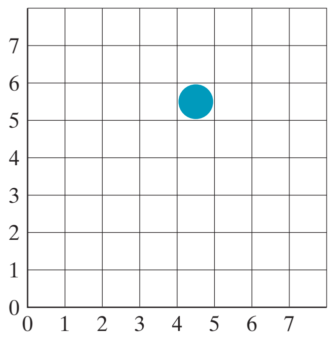


FIGURE 3-34 Illuminated pixel at raster position (4, 5).

Screen Grid Coordinates

Figure 3-33 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 Fig. 3-34. And 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 Fig. 3-35.

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:

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

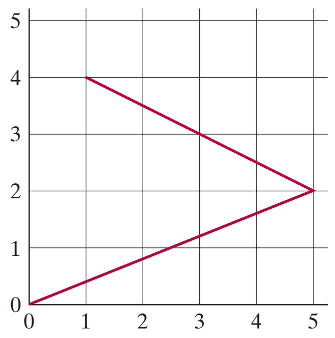


FIGURE 3-35 Line path for two connected line segments between screen grid-coordinate positions.

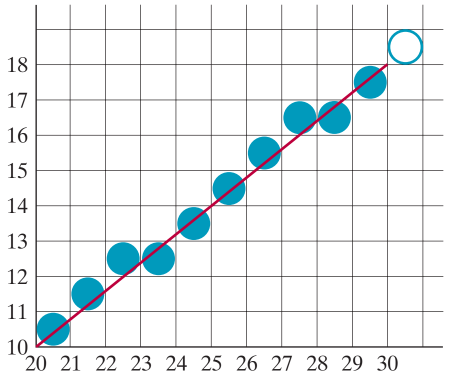


FIGURE 3-36 Line path and corresponding pixel display for grid endpoint coordinates (20, 10) and (30, 18).

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.

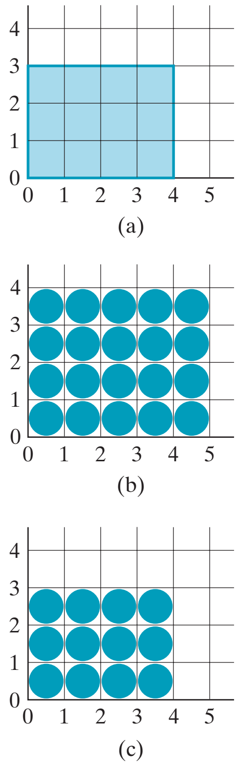


FIGURE 3-37 Conversion of rectangle (a) with vertices at screen coordinates (0, 0), (4, 0), (4, 3), and (0, 3) into display (b) that includes the right and top boundaries and into display (c) that maintains geometric magnitudes.

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 3-36 shows the line plotted in the Bresenham line-algorithm example of Section 3-5. 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 3-5, 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 Fig. 3-36, 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 line that has the same geometric magnitudes as the mathematical line from (20, 10) to (30, 18).

For an enclosed area, input geometric properties are maintained by displaying the area using only those pixels that are interior to the object boundaries. The rectangle defined with the screen coordinate vertices shown in Fig. 3-37(a), for example, is larger when we display it filled with pixels up to and including

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

the border pixel lines joining the specified vertices (Fig. 3-37(b)). As defined, the area of the rectangle is 12 units, but as displayed in Fig. 3-37(b), it has an area of 20 units. In Fig. 3-37(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 $x = 3$, since 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 Fig 3-38 by the midpoint circle algorithm using pixel centers as screen-coordinate positions. But 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 Fig. 3-39.

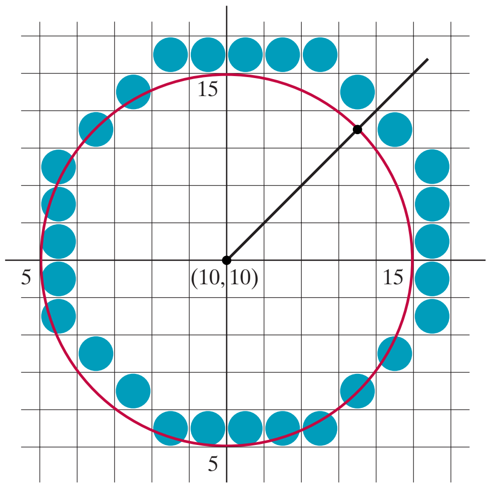


FIGURE 3-38
A midpoint-algorithm plot of the circle equation $(x - 10)^2 + (y - 10)^2 = 5^2$ using pixel-center coordinates.

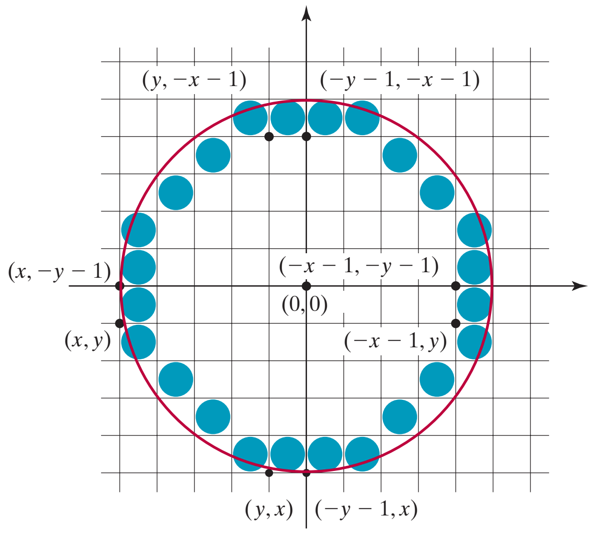


FIGURE 3-39
Modification of the circle plot in Fig. 3-38 to maintain the specified circle diameter of 10.

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

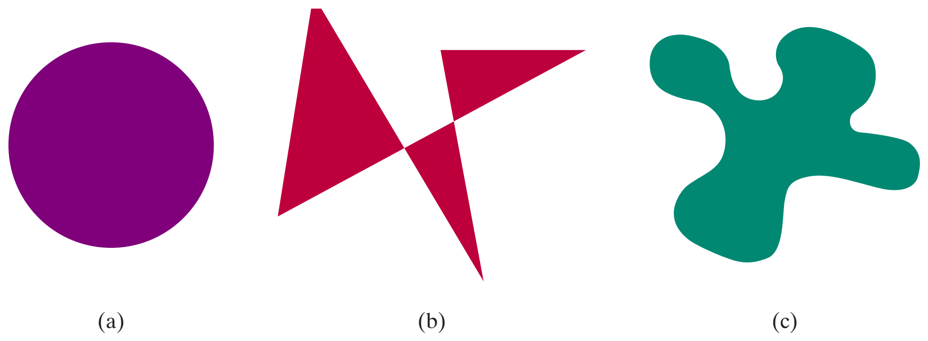


FIGURE 3-40 Solid-color fill areas specified with various boundaries. (a) A circular fill region. (b) A fill area bounded by a closed polyline. (c) A filled area specified with an irregular curved boundary.

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.

3-14 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 some color option. Figure 3-40 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. Other fill options are discussed in Chapter 4.

Although any fill-area shape is possible, graphics libraries generally do not support specifications for arbitrary fill shapes. Most library routines require that 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. And 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 3-41 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.**

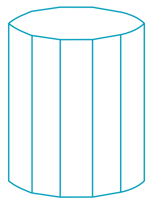


FIGURE 3-41 Wire-frame representation for a cylinder, showing only the front (visible) faces of the polygon mesh used to approximate the surfaces.

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

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

border. Other fill-area methods for objects with curved boundaries are given in Chapter 4.

3-15 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 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 subsection 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 3-42 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.

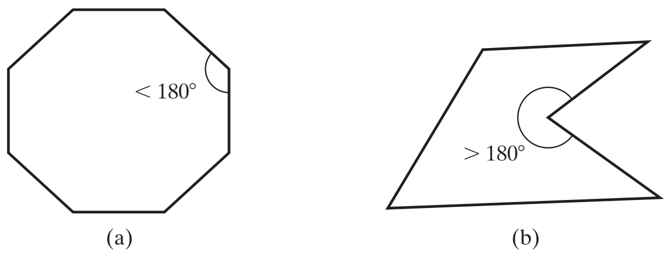


FIGURE 3-42 A convex polygon (a), and a concave polygon (b).

hearn-50265; ISBN: 0-13-015390-7 *book* *July 30, 2003* *15:46*

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

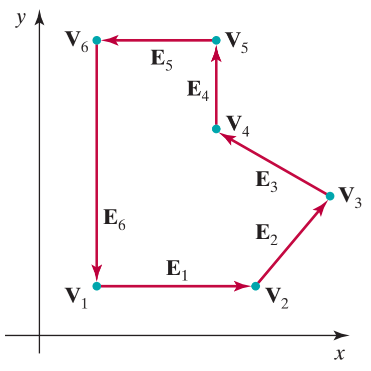
A concave polygon has at least one interior angle greater than 180°. Also, the extension of some edges of a concave polygon will intersect other edges, and some pair of interior points will produce a line segment that intersects the polygon boundary. Therefore, we can use any one of these characteristics 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 3-43 illustrates the edge-vector, cross-product method for identifying concave polygons.

Another way to identify a concave polygon is to take a 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



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

FIGURE 3-43 Identifying a concave polygon by calculating cross products of successive pairs of edge vectors.

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

the xy plane, but we can always move it into that plane using the transformation methods discussed in Chapter 5.

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 a counterclockwise order. If any cross product has a negative z component (as in Fig. 3-43), 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.

EXAMPLE 3-4 Vector Method for Splitting Concave Polygons

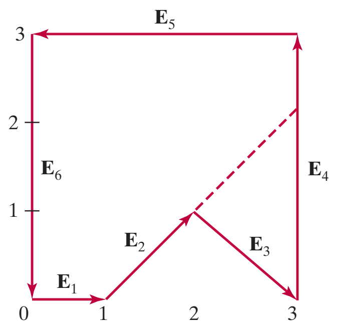


FIGURE 3-44 Splitting a concave polygon using the vector method.

Figure 3-44 shows a concave polygon with six edges. Edge vectors for this polygon can be expressed as

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

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

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

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

We can also split a concave polygon using a **rotational method.** Proceeding counterclockwise around the polygon edges, we shift the position of the polygon so that each vertex $\mathbf{V}_k$ in turn is at the coordinate origin. Then, we rotate the polygon about the origin in a clockwise direction so that the next vertex $\mathbf{V}_{k+1}$ is on the x axis. If the following vertex, $\mathbf{V}_{k+2}$, is below the x axis, the polygon is concave. We then split the polygon along the x axis to form two new polygons, and we repeat the concave test for each of the two new polygons. The steps above are repeated until we have tested all vertices in the polygon list. Methods for rotating and shifting the position of an object are discussed in detail in Chapter 5. Figure 3-45 illustrates the rotational method for splitting a concave polygon.

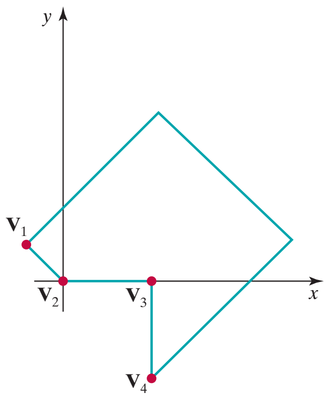


FIGURE 3-45 Splitting a concave polygon using the rotational method. After moving $\mathbf{V}_2$ to the coordinate origin and rotating $\mathbf{V}_3$ onto the x axis, we find that $\mathbf{V}_4$ is below the x axis. So we split the polygon along the line of $\overline{\mathbf{V}_2\mathbf{V}_3}$, which 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 by divided into a set of triangles using this approach, as long as 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 Fig. 3-46. 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 **P** to a distant point 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 3-46(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,

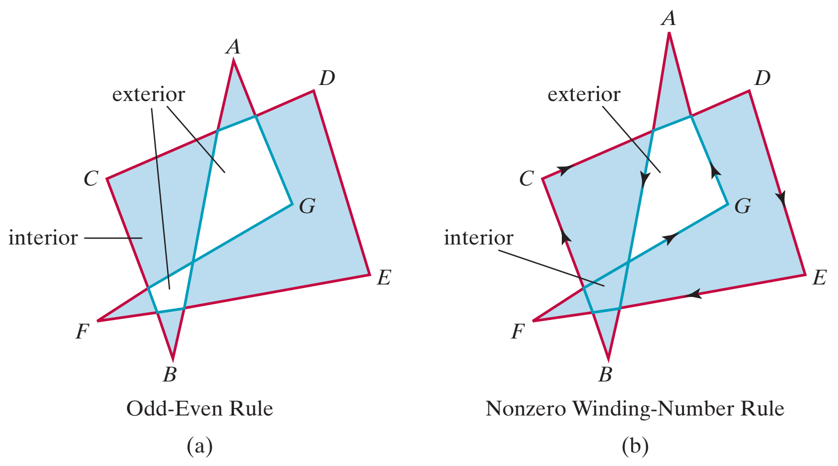


FIGURE 3-46 Identifying interior and exterior regions of a closed polyline that contains self-intersecting segments.

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

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 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 3-46(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 Fig. 3-46.

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 $(-u_y, u_x)$ (Appendix A). 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**.

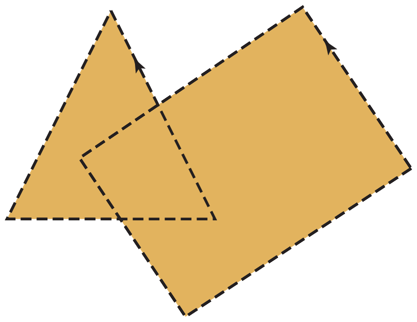


FIGURE 3-47 A fill area defined as a region that has a positive value for the winding number. This fill area is the union of two regions, each with a counterclockwise border direction.

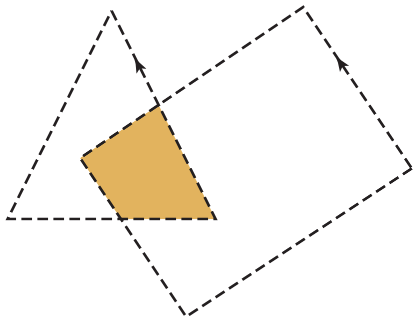


FIGURE 3-48 A fill area defined as a region with a winding number greater than 1. This fill area is the intersection of two regions, each with a counterclockwise border direction.

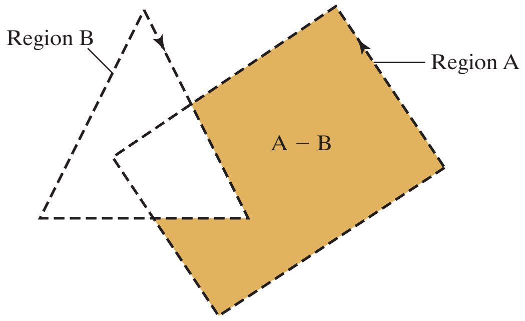


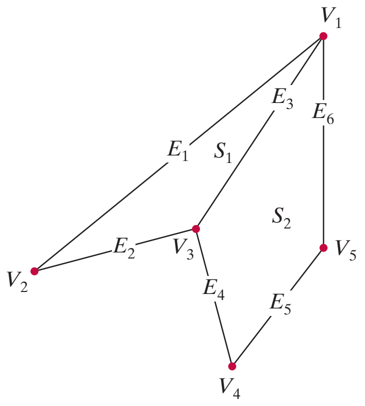
FIGURE 3-49 A fill area defined as a region with a positive value for the winding number. This fill area is the difference, A − B, of two regions, where region A has a positive border direction (counterclockwise) and region B has a negative border direction (clockwise).

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 (Fig. 3-47). Similarly, the intersection of two regions with counterclockwise boundaries would contain those points whose winding number is greater than 1, as illustrated in Fig. 3-48. 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 (Fig. 3-49) is the set of all points whose winding number is positive.

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

hearn-50265; ISBN: 0-13-015390-7 *book* *July 30, 2003* *15:46*



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 3-50 Geometric data-table representation for two adjacent polygon surface facets, formed with six edges and five vertices.

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 Fig. 3-50 for two 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 Fig. 3-50, 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 Fig. 3-50 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

hearn-50265; ISBN: 0-13-015390-7 *book* *July 30, 2003* *15:46*

could be identified more rapidly (Fig. 3-51). This is particularly useful for rendering procedures that must vary surface shading smoothly across the edges from one polygon to the next. Similarly, the vertex table could be expanded to reference corresponding edges, for faster information retrieval.

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

Since 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.

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 3-51 Edge table for the surfaces of Fig. 3-50 expanded to include pointers into the surface-facet table.

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

$$A\,x + B\,y + C\,z + D = 0 \tag{3-59}$$

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:

$$(A/D)x_k + (B/D)y_k + (C/D)z_k = -1, \qquad k = 1, 2, 3 \tag{3-60}$$

The solution to this set of equations can be obtained in determinant form, using

Cramer's rule, as

$$A = \begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{vmatrix} \qquad B = \begin{vmatrix} x_1 & 1 & z_1 \\ x_2 & 1 & z_2 \\ x_3 & 1 & z_3 \end{vmatrix}$$
$$C = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \qquad D = -\begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix} \tag{3-61}$$

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

$$\begin{aligned} A &= y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2) \\ B &= z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2) \\ C &= x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2) \\ D &= -x_1(y_2z_3 - y_3z_2) - x_2(y_3z_1 - y_1z_3) - x_3(y_1z_2 - y_2z_1) \end{aligned} \tag{3-62}$$

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

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

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

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

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

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

These inequality tests are valid in a right-handed Cartesian system, provided the plane parameters A, B, C, and D were calculated using coordinate positions selected in a strictly counterclockwise order when viewing the surface along a front-to-back direction. For example, in Fig. 3-52, any point outside (in front of) the plane of the shaded polygon satisfies the inequality $x - 1 > 0$, while any point inside (in back of) the plane has an x-coordinate value less than 1.

Orientation of a polygon surface in space can be described with the **normal vector** for the plane containing that polygon, as shown in Fig. 3-53. 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 Eqs. 3-62. The normal vector points in a direction from inside the plane to the outside; that is, from the back face of the polygon to the front face.

As an example of calculating the components of the normal vector for a polygon, which also gives us the plane parameters, we choose three of the vertices of the shaded face of the unit cube in Fig. 3-52. 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 Eqs. 3-62 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:

$$\mathbf{N} = (\mathbf{V}_2 - \mathbf{V}_1) \times (\mathbf{V}_3 - \mathbf{V}_1) \tag{3-63}$$

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 the plane equation 3-59 and solving for D. The plane equation can be expressed in vector form using the normal $\mathbf{N}$ and the position $\mathbf{P}$ of any point in the plane as

$$\mathbf{N} \cdot \mathbf{P} = -D \tag{3-64}$$

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.

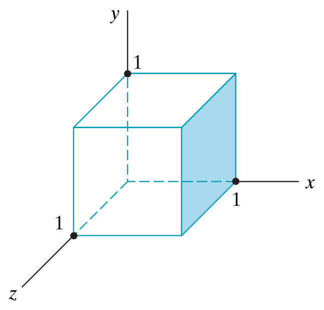


FIGURE 3-52 The shaded polygon surface of the unit cube has plane equation $x - 1 = 0$.

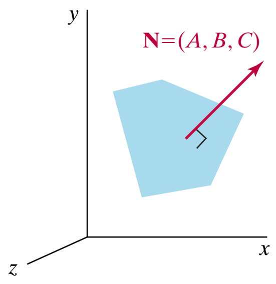


FIGURE 3-53 The normal vector $\mathbf{N}$ for a plane described with the equation $Ax + By + Cz + D = 0$ is perpendicular to the plane and has Cartesian components (A, B, C).

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

3-16 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 (which are described in the next chapter), 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 Fig. 3-54. 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 for that polygon.

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*

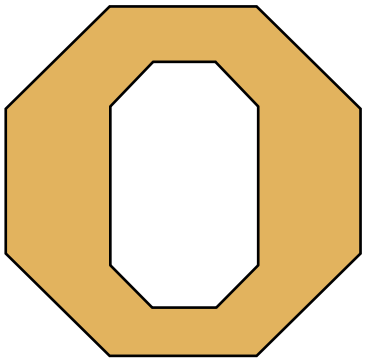


FIGURE 3-54 A polygon with a complex interior, which cannot be specified with a single vertex list.

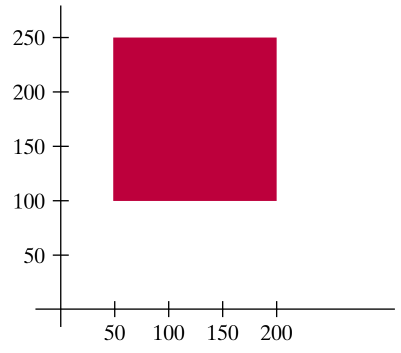


FIGURE 3-55 Display of a square fill area using the `glRect` function.

coordinate axes. As an example, the following statement defines the square shown in Fig. 3-55.

```
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 the first vertex. 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. In Chapter 4, we discuss another way that we can reverse the specification of front and back polygon faces.

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 Fig. 3-56(a). For this example, we assume that we have a list of six points, labeled p1 through p6, specifying 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.

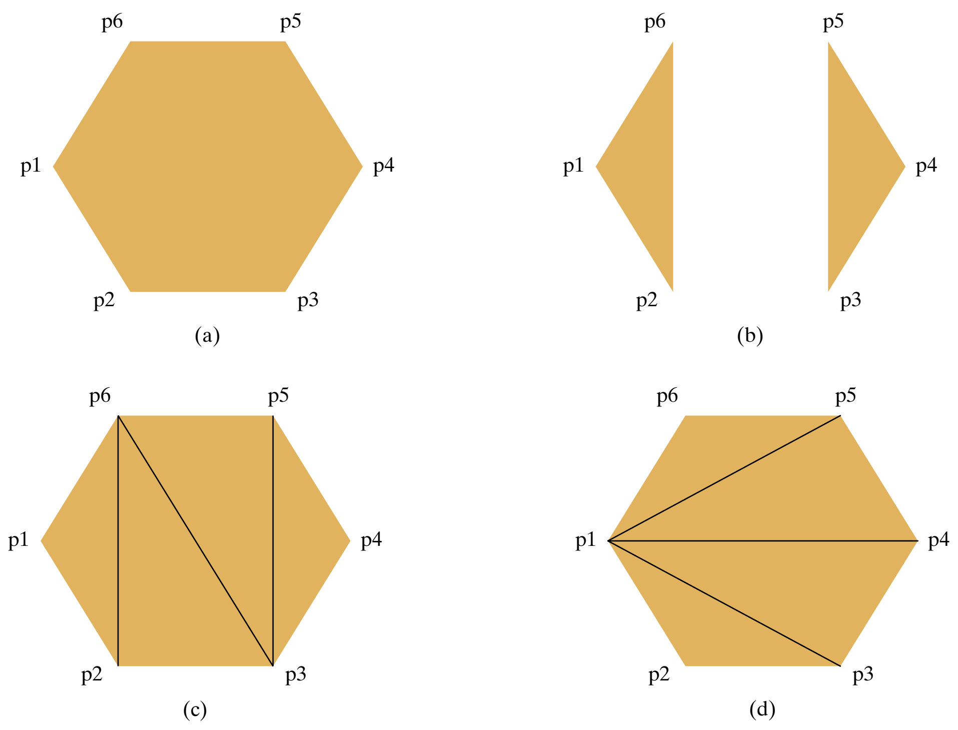


FIGURE 3-56 Displaying polygon fill areas using a list of six vertex positions. (a) A single convex polygon fill area generated with the primitive constant `GL_POLYGON`. (b) Two unconnected triangles generated with `GL_TRIANGLES`. (c) Four connected triangles generated with `GL_TRIANGLE_STRIP`. (d) Four connected triangles generated with `GL_TRIANGLE_FAN`.

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 Fig. 3-56(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 three, 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 Fig. 3-56(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 Fig. 3-56(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 be specified in the proper order to correctly define front and back faces for each triangle. 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

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

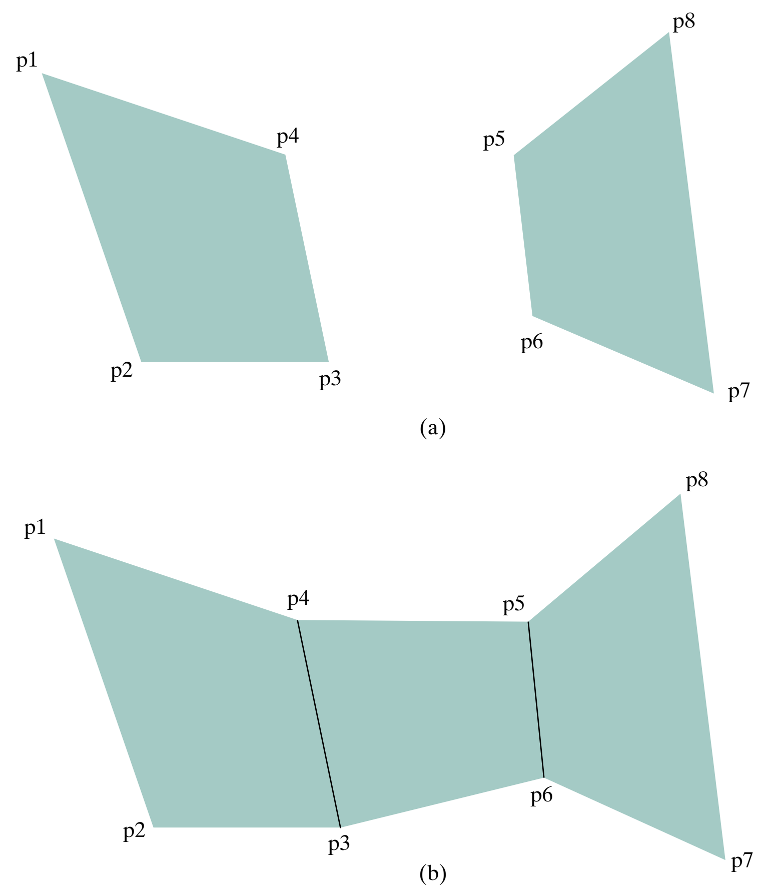


FIGURE 3-57 Displaying quadrilateral fill areas using a list of eight vertex positions. (a) Two unconnected quadrilaterals generated with GL_QUADS. (b) Three connected quadrilaterals generated with GL_QUAD_STRIP.

vertices, specified as two-dimensional coordinate arrays, we can generate the display shown in Fig. 3-57(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 four, 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 Fig. 3-57(b).

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

```
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 processing of the polygon facets. In OpenGL, primitives 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 one million or more 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 OpenGL Utility (GLU) 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.

3-17 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 Fig. 3-58, with coordinates given in integers to simplify the following discussion. A straightforward method for defining the vertex coordinates is to

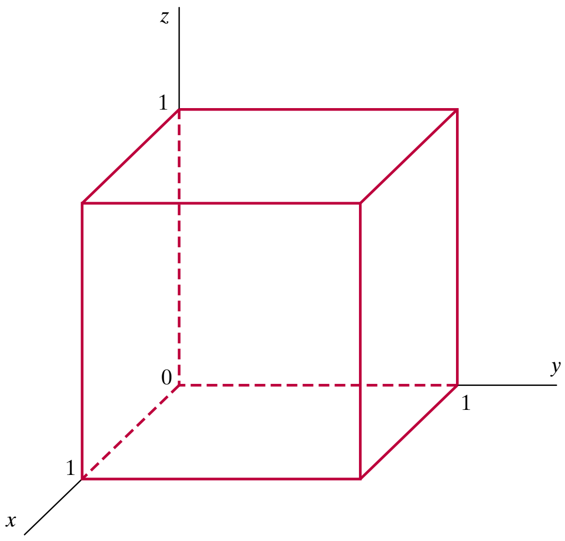


FIGURE 3-58 A cube with an edge length of 1.

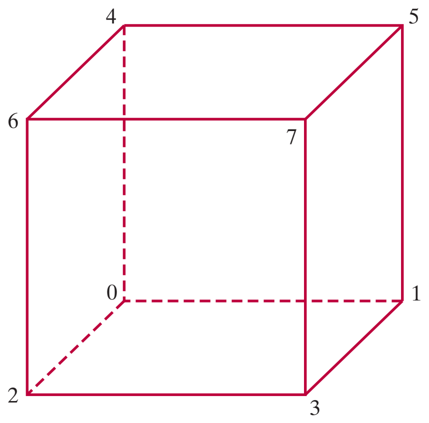


FIGURE 3-59 Subscript values for array `pt` corresponding to the vertex coordinates for the cube shown in Fig. 3-58.

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} };
```

Or 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 3-59 shows the subscript values for array `pt` corresponding to the cube vertex positions.

```
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 one hundred 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 Fig. 3-58) 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. As we noted in Chapter 2, the server (our workstation, for example) generates commands and displays the picture. Of course, a single machine can be both client and server.

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

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. Since we are only giving 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`. Since a quadrilateral requires just four vertices and we specified 24, the `glDrawElements` function continues to display another cube face after each successive set of four 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. Since 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. And we can interlace the various arrays for greater efficiency. We take a look at the methods for implementing these attribute arrays in the next chapter.

3-18 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. As we noted in Chapter 2, a pixel array of color values is typically referred to as a *pixmap*.

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 3-60 gives an example of mapping a pixel-color array onto a screen area.

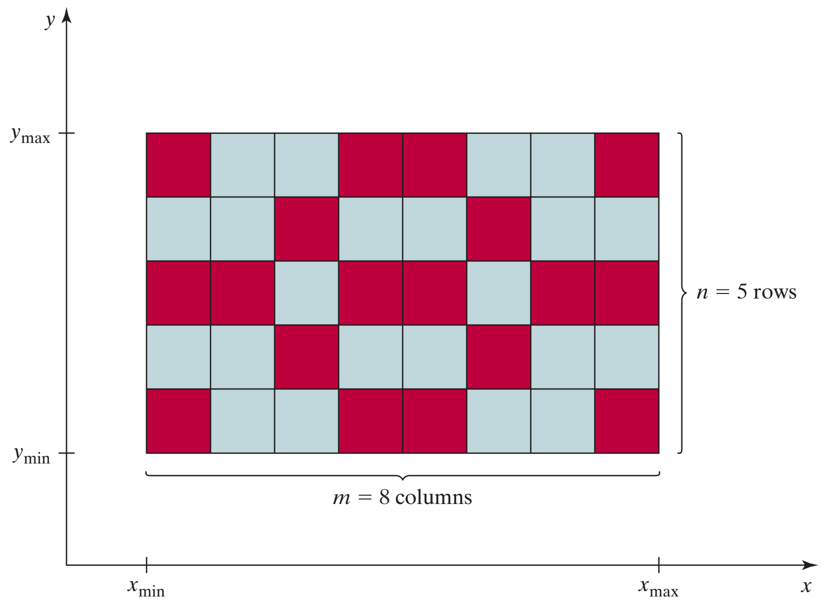


FIGURE 3-60 Mapping an n by m color array onto a region of the screen coordinates.

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 or not a pixel is to be assigned (or combined with) a preset color.

3-19 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 is a bitmap and the other is a pixmap. 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 set 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. As an example, Fig. 3-61 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 Fig. 3-61 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);
```

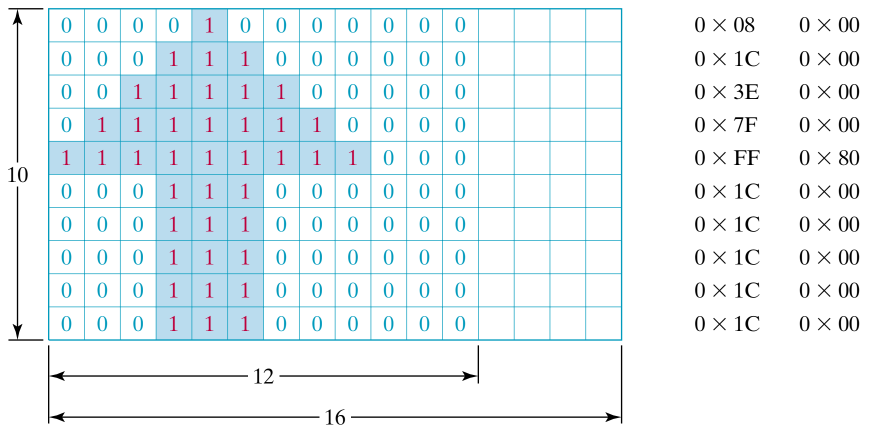

0 0 0 0 1 0 0 0 0 0 0 0 — 0 × 08 — 0 × 00
0 0 0 1 1 1 0 0 0 0 0 0 — 0 × 1C — 0 × 00
0 0 1 1 1 1 1 0 0 0 0 0 — 0 × 3E — 0 × 00
0 1 1 1 1 1 1 1 0 0 0 0 — 0 × 7F — 0 × 00
1 1 1 1 1 1 1 1 1 0 0 0 — 0 × FF — 0 × 80
0 0 0 1 1 1 0 0 0 0 0 0 — 0 × 1C — 0 × 00
0 0 0 1 1 1 0 0 0 0 0 0 — 0 × 1C — 0 × 00
0 0 0 1 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
10
12
16

FIGURE 3-61 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 make no use of 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, and we examine color selections in greater detail in the next chapter. 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-by-128 array of RGB color values.

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

Since 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. We examine both of these buffers in more detail in later chapters.

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

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

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. And 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.

3-20 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

hearn-50265; ISBN: 0-13-015390-7 *book* *July 30, 2003* *15:46*

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.

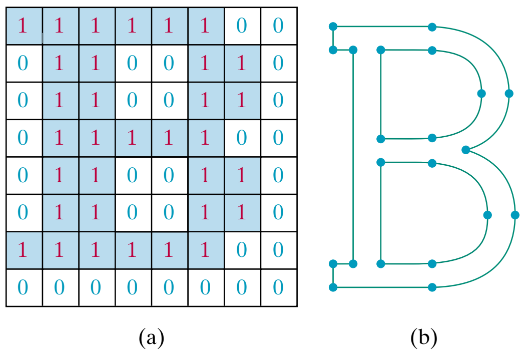
Letters, numbers, and other characters can be displayed in a variety of sizes and styles. The overall design style for a set (or family) of characters is called a **typeface.** Today, there are thousands of typefaces available for computer applications. Examples of a few common typefaces are Courier,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

Typefaces (or 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 accents. For example, the text in this book 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 quickly recognized, this typeface 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.

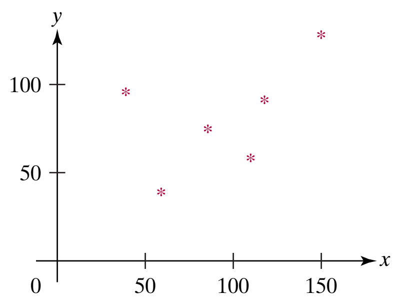
Two different representations are used for storing computer fonts. A simple method for representing the character shapes in a particular typeface is to set up a pattern of binary values on a rectangular grid. The set of characters is then referred to as a **bitmap font** (or **bitmapped font**). A bitmapped 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 3-62 illustrates the two methods for character representation. When the pattern in Fig. 3-62(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 Fig. 3-62(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



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 3-62 The letter "B" represented with an 8-by-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 3-63 A polymarker graph of a set of data values.

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

There are 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 is one that 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 3-63 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.

3-21 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 Fig. 3-61, 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 OpenGL Utility Toolkit (GLUT). 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 type faces, and parameter `character` is assigned either the ASCII code or the specific character we wish to display. Thus, to display the upper-case 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 lower-case 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 (Chapter 5) before executing the `glutStrokeCharacter` routine. After each character is displayed, a coordinate offset is automatically applied 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

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

with line segments. Thus, they can be viewed from various directions, and we can shrink or expand them without distortion, or transform them in other ways. But they are slower to render, compared to bitmapped fonts.

3-22 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. And we can also use this feature of a graphics package for hierarchical modeling (Chapter 14), 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++.

3-23 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

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

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:

```
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.

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

```
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.

3-24 OpenGL DISPLAY-WINDOW RESHAPE FUNCTION

In our introductory OpenGL program (Section 2-9), we discussed the functions for setting up an initial display window. But 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 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

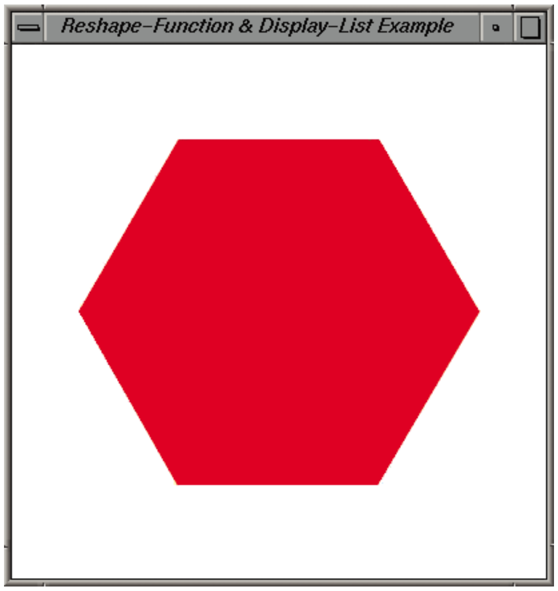


FIGURE 3-64 Display window generated by the example program illustrating the use of the reshape function.

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

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 3-23. 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 Fig. 3-64.

```
#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 ( );
}
```

3-25 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. Examples of displays generated with output primitives are illustrated in Figs. 3-65 and 3-66.

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 computer-aided design. 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.

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, since each scan line crossing a triangle intersects exactly two polygon edges (assuming 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.

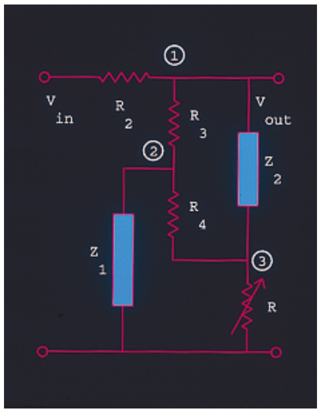


FIGURE 3-66 An electrical diagram drawn with straight-line sections, circles, filled rectangles, and text. (*Courtesy of Wolfram Research, Inc., The Maker of Mathematica.*)

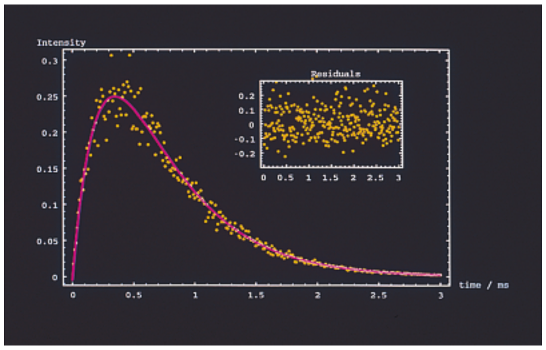


FIGURE 3-65 A data plot generated with straight-line segments, curves, character marker symbols, and text. (*Courtesy of Wolfram Research, Inc., The Maker of Mathematica.*)

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

TABLE 3-1

SUMMARY OF OpenGL OUTPUT PRIMITIVE FUNCTIONS AND RELATED ROUTINES

Function	*Description*
`gluOrtho2D`	Specify a 2D world-coordinate reference.
`glVertex*`	Select a coordinate position. This function must be placed within a `glBegin/glEnd` pair.
`glBegin (GL_POINTS);`	Plot 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);`	Display 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);`	Display a polyline, specified using the same structure as `GL_LINES`.
`glBegin (GL_LINE_LOOP);`	Display a closed polyline, specified using the same structure as `GL_LINES`.
`glRect*`	Display a fill rectangle in the *xy* plane.
`glBegin (GL_POLYGON);`	Display a fill polygon, whose vertices are given in `glVertex` functions and terminated with a `glEnd` statement.
`glBegin (GL_TRIANGLES);`	Display a set of fill triangles using the same structure as `GL_POLYGON`.
`glBegin (GL_TRIANGLE_STRIP);`	Display a fill-triangle mesh, specified using the same structure as `GL_POLYGON`.
`glBegin (GL_TRIANGLE_FAN);`	Display 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);`	Display a set of fill quadrilaterals, specified with same structure as `GL_POLYGON`.
`glBegin (GL_QUAD_STRIP);`	Display a fill-quadrilateral mesh, specified with same structure as `GL_POLYGON`.
`glEnableClientState (GL_VERTEX_ARRAY);`	Activate vertex-array features of OpenGL.
`glVertexPointer (size, type, stride, array);`	Specify an array of coordinate values.
`glDrawElements (prim, num, type, array);`	Display a specified primitive type from array data.

Function	Description
`glNewList (listID, listMode)`	Define a set of commands as a display list, terminate with a `glEndList` statement.
`glGenLists`	Generate one or more display-list identifiers.
`glIsList`	Query function to determine whether a display-list identifier is in use.
`glCallList`	Execute a single display list.
`glListBase`	Specify an offset value for an array of display-list identifiers.
`glCallLists`	Execute multiple display lists.
`glDeleteLists`	Eliminate a specified sequence of display lists.
`glRasterPos*`	Specify 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);`	Specify a binary pattern that is to be mapped to pixel positions relative to the current position.
`glDrawPixels (w, h, type, format, pattern);`	Specify a color pattern that is to be mapped to pixel positions relative to the current position.
`glDrawBuffer`	Select one or more buffers for storing a pixmap.
`glReadPixels`	Save a block of pixels in a selected array.
`glCopyPixels`	Copy a block of pixels from one buffer position to another.
`glLogicOp`	Select a logical operation for combining two pixel arrays, after enabling with the constant `GL_COLOR_LOGIC_OP`.
`glutBitmapCharacter (font, char);`	Specify a font and a bitmap character for display.
`glutStrokeCharacter (font, char);`	Specify a font and an outline character for display.
`glutReshapeFunc`	Specify actions to be taken when display-window dimensions are changed.

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

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 efficiently generate the display using OpenGL vertex arrays to specify geometric and other data.

In Table 3-1, 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 3-1. A display window is set up for the output from each program using the GLUT routines discussed in Chapter 2.

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. Since the bit maps 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 3-67 shows the output of the line-graph program.

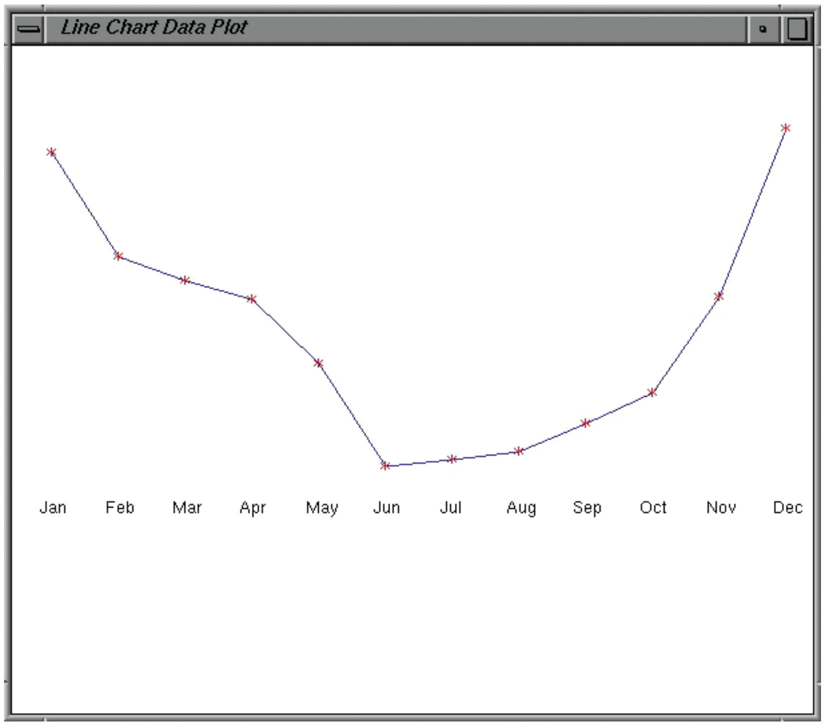


FIGURE 3-67 A polyline and polymarker plot of data points output by the `lineGraph` routine.

```
#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 Fig. 3-68. This program illustrates an application of rectangular fill areas, as well as bit-mapped character labels.

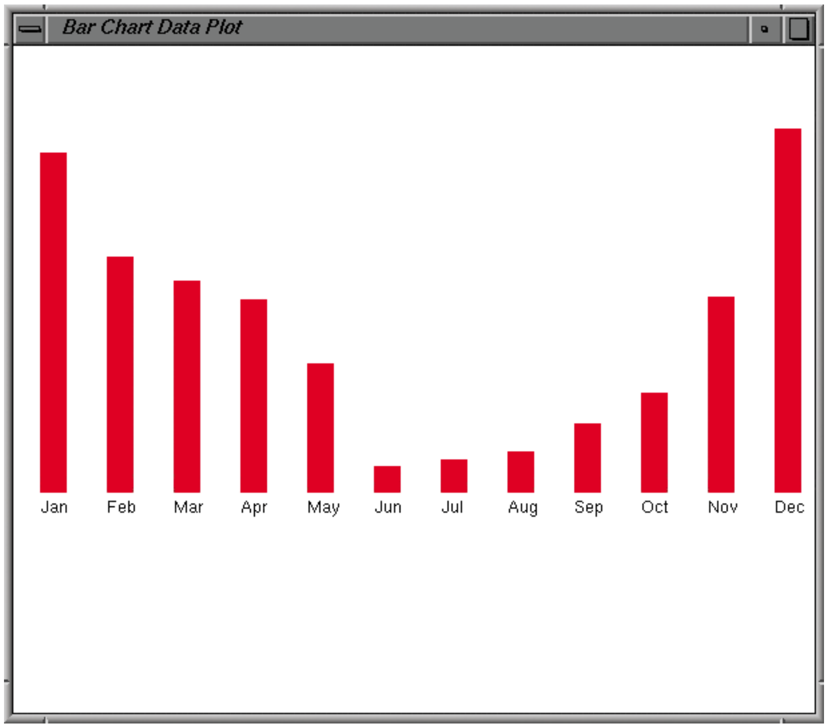


FIGURE 3-68 A bar chart generated by the `barChart` procedure.

```
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 ( );
}
```

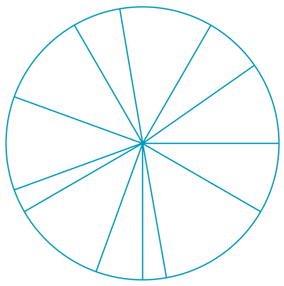


FIGURE 3-69 Output produced with the `pieChart` procedure.

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 Fig. 3-69.

```
#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;
    }
}
```

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

```
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 (3-28) 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 Fig. 3-70 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.

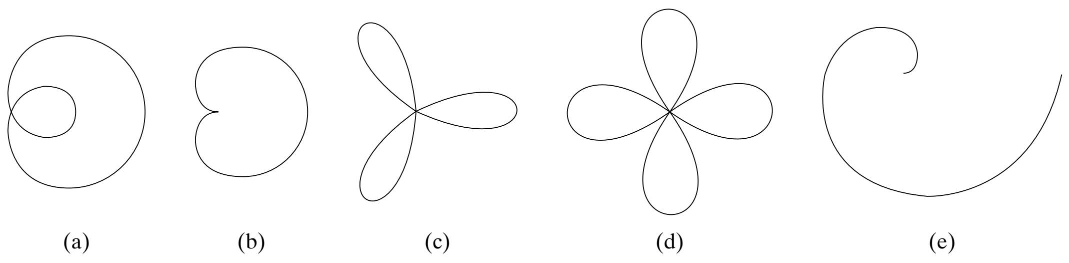


FIGURE 3-70 Curved figures displayed by the `drawCurve` procedure: (a) limaçon, (b) cardiod, (c) three-leaf curve, (d) four-leaf curve, and (e) spiral.

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

```
#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 cardiod 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;
```

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 15:46

```
    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, Neider, Davis, and Shreiner (1999). A listing of all OpenGL primitive functions is available in Shreiner (2000). For a complete reference to GLUT, see Kilgard (1996).

EXERCISES

3-1 Implement a polyline function using the DDA algorithm, given any number (n) of input points. A single point is to be plotted when $n = 1$.

3-2 Extend Bresenham's line algorithm to generate lines with any slope, taking symmetry between quadrants into account.

3-3 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.

3-4 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.

3-5 Use the midpoint method to derive decision parameters that can be used to generate straight-line segments with any slope.

3-6 Set up a parallel version of Bresenham's line algorithm for slopes in the range $0 < m < 1$.

3-7 Set up a parallel version of Bresenham's algorithm for straight lines with any slope.

3-8 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)?

3-9 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 6 bits of storage, what is the frame buffer address (or addresses) of the pixel with screen coordinates (x, y)?

3-10 Incorporate the iterative techniques for calculating frame-buffer addresses (Section 3-7) into the Bresenham line algorithm.

3-11 Revise the midpoint circle algorithm to display circles with input geometric magnitudes preserved (Section 3-13).

3-12 Set up a procedure for a parallel implementation of the midpoint circle algorithm.

3-13 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.

3-14 Set up a procedure for a parallel implementation of the midpoint ellipse algorithm.

3-15 Devise an efficient algorithm that takes advantage of symmetry properties to display a sine function over one cycle.

3-16 Modify the algorithm in the preceding exercise to display a sine curve over any specified angular interval.

3-17 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 $\frac{A}{10}$.

3-18 Using the midpoint method, and taking symmetry into account, develop an efficient algorithm for scan conversion of the following curve over the interval $-10 \le x \le 10$.

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

3-19 Use the midpoint method and symmetry considerations to scan convert the parabola

$$y = 100 - x^2$$

over the interval $-10 \le x \le 10$.

3-20 Use the midpoint method and symmetry considerations to scan convert the parabola

$$x = y^2$$

for the interval $-10 \le y \le 10$.

3-21 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.

hearn-50265; ISBN: 0-13-015390-7 *book* *July 30, 2003* *15:46*

3-22 Set up geometric data tables as in Fig. 3-50 for a unit cube.

3-23 Set up geometric data tables for a unit cube 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 Exercise 3-22. Estimate the storage requirements for each.

3-24 Define an efficient polygon-mesh representation for a cylinder and justify your choice of representation.

3-25 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.

3-26 Devise routines for checking the three geometric data tables in Fig. 3-50 to ensure consistency and completeness.

3-27 Write a program for calculating parameters A, B, C, and D for an input mesh of polygon-surface facets.

3-28 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.

3-29 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?

3-30 Develop a procedure for identifying a nonplanar vertex list for a quadrilateral.

3-31 Extend the algorithm of the previous exercise to identify a nonplanar vertex list that contains more than four coordinate positions.

3-32 Write a procedure to split a set of four polygon vertex positions into a set of triangles.

3-33 Devise an algorithm for splitting a set of n polygon vertex positions, with $n > 4$, into a set of triangles.

3-34 Set up an algorithm for identifying a degenerate polygon vertex list that may contain repeated vertices or collinear vertices.

3-35 Devise an algorithm for identifying a polygon vertex list that contains intersecting edges.

3-36 Write a routine to identify concave polygons by calculating cross products of pairs of edge vectors.

3-37 Write a routine to split a concave polygon, using the vector method.

3-38 Write a routine to split a concave polygon, using the rotational method.

3-39 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.

3-40 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.

3-41 What regions of the self-intersecting polyline shown in Fig. 3-46 have a positive winding number? What are the regions that have a negative winding number? What regions have a winding number greater than 1?

3-42 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.

3-43 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.

3-44 Modify the example program in Section 3-24 so that the displayed hexagon is always at the center of the display window, regardless of how the display window may be resized.

3-45 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.

3-46 Write a program to display a bar chart in any selected area of a display window.

3-47 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.)

3-48 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.