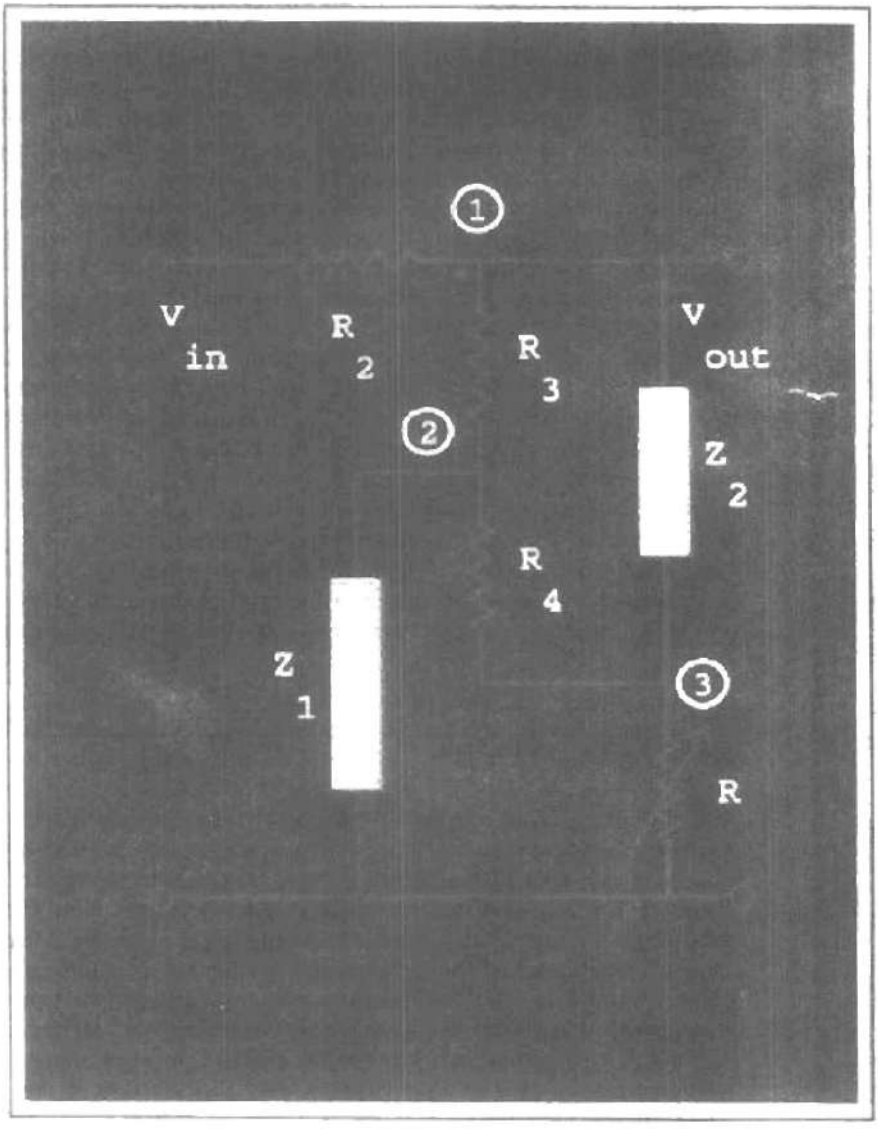
CHAPTER

3 Output Primitives



A picture can be described in several ways. Assuming we have a raster display, a picture is completely specified by the set of intensities for the pixel positions in the display. At the other extreme, we can describe a picture as a set of complex objects, such as trees and terrain or furniture and walls, positioned at specified coordinate locations within the scene. Shapes and colors of the objects can be described internally with pixel arrays or with sets of basic geometric structures, such as straight line segments and polygon color areas. The scene is then displayed either by loading the pixel arrays into the frame buffer or by scan converting the basic geometric-structure specifications into pixel patterns. Typically, graphics programming packages provide functions to describe a scene in terms of these basic geometric structures, referred to as **output primitives**, and to group sets of output primitives into more complex structures. Each output primitive is specified with input coordinate data and other information about the way that object is to be displayed. Points and straight line segments are the simplest geometric components of pictures. Additional output primitives that can be used to construct a picture include circles and other conic sections, quadric surfaces, spline curves and surfaces, polygon color areas, and character strings. We begin our discussion of picture-generation procedures by examining device-level algorithms for displaying two-dimensional output primitives, with particular emphasis on scan-conversion methods for raster graphics systems. In this chapter, we also consider how output functions can be provided in graphics packages, and we take a look at the output functions available in the PHIGS language.

3-1 POINTS AND LINES

Point plotting is accomplished by converting a single coordinate position furnished by an application program into appropriate operations for the output device in use. With a CRT monitor, for example, the electron beam is turned on to illuminate the screen phosphor at the selected location. How the electron beam is positioned depends on the display technology. A random-scan (vector) system stores point-plotting instructions in the display list, and coordinate values in these instructions are converted to deflection voltages that position the electron beam at the screen locations to be plotted during each refresh cycle. For a black-and-white raster system, on the other hand, a point is plotted by setting the bit value corresponding to a specified screen position within the frame buffer to 1. Then, as the electron beam sweeps across each horizontal scan line, it emits a

burst of electrons (plots a point) whenever a value of 1 is encountered in the frame buffer. With an RGB system, the frame buffer is loaded with the color codes for the intensities that are to be displayed at the screen pixel positions.

Line drawing is accomplished by calculating intermediate positions along the line path between two specified endpoint positions. An output device is then directed to fill in these positions between the endpoints. For analog devices, such as a vector pen plotter or a random-scan display, a straight line can be drawn smoothly from one endpoint to the other. Linearly varying horizontal and vertical deflection voltages are generated that are proportional to the required changes in the x and y directions to produce the smooth line.

Digital devices display a straight line segment by plotting discrete points between the two endpoints. Discrete coordinate positions along the line path are calculated from the equation of the line. For a raster video display, the line color (intensity) is then loaded into the frame buffer at the corresponding pixel coordinates. Reading from the frame buffer, the video controller then "plots" the screen pixels. Screen locations are referenced with integer values, so plotted positions may only approximate actual line positions between two specified endpoints. A computed line position of (10.48, 20.51), for example, would be converted to pixel position (10, 21). This rounding of coordinate values to integers causes lines to be displayed with a stairstep appearance ("the jaggies"), as represented in Fig 3-1. The characteristic stairstep shape of raster lines is particularly noticeable on systems with low resolution, and we can improve their appearance somewhat by displaying them on high-resolution systems. More effective techniques for smoothing raster lines are based on adjusting pixel intensities along the line paths.

For the raster-graphics device-level algorithms discussed in this chapter, object positions are specified directly in integer device coordinates. For the time being, we will assume that pixel positions are referenced according to scan-line number and column number (pixel position across a scan line). This addressing scheme is illustrated in Fig. 3-2. Scan lines are numbered consecutively from 0, starting at the bottom of the screen; and pixel columns are numbered from 0, left to right across each scan line. In Section 3-10, we consider alternative pixel addressing schemes.

To load a specified color into the frame buffer at a position corresponding to column x along scan line y, we will assume we have available a low-level procedure of the form

```
setPixel (x, y)
```

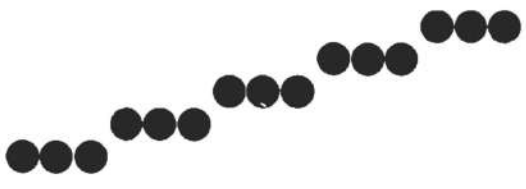


Figure 3-1
Stairstep effect (jaggies) produced when a line is generated as a series of pixel positions.

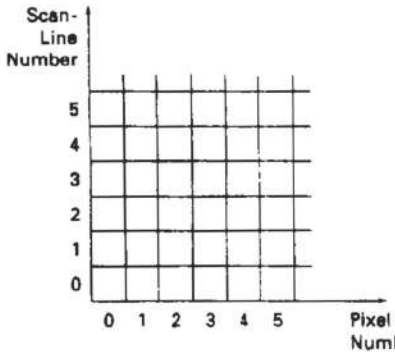


Figure 3-2
Pixel positions referenced by scan-line number and column number.

We sometimes will also want to be able to retrieve the current frame-buffer intensity setting for a specified location. We accomplish this with the low-level function

```
getPixel (x, y)
```

3-2 LINE-DRAWING ALGORITHMS

The Cartesian *slope-intercept equation* for a straight line is

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

with m representing 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_1, y_1) and (x_2, y_2), as shown in Fig. 3-3, we can determine values for the slope m and y intercept b with the following calculations:

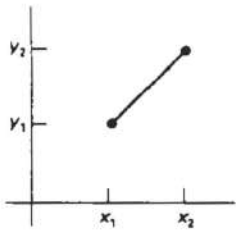


Figure 3-3
Line path between endpoint positions (x_1, y_1) and (x_2, y_2).

$$m = \frac{y_2 - y_1}{x_2 - x_1} \tag{3-2}$$

$$b = y_1 - m \cdot x_1 \tag{3-3}$$

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

$$\Delta y = m \, \Delta x \tag{3-4}$$

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 de-

vices. 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. 3-4, for a near horizontal line with discrete sample positions along the x axis.

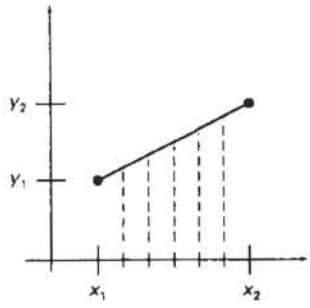


Figure 3-4
Straight line segment with five sampling positions along the x axis between x_1 and x_2.

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. We sample the line at unit intervals in one coordinate and determine corresponding integer values nearest the line path for the other coordinate.

Consider first a line with positive slope, as shown in Fig. 3-3. If the slope is less than or equal to 1, we sample at unit x intervals ($\Delta x = 1$) and compute each successive y value as

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

Subscript k takes integer values starting from 1, for the first point, and increases by 1 until the final endpoint is reached. Since m can be any real number between 0 and 1, the calculated y values must be rounded to the nearest integer.

For lines with a positive slope greater than 1, we reverse the roles of x and y. That is, we sample at unit y intervals ($\Delta y = 1$) and calculate each succeeding x value as

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

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. 3-3). 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

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

Equations 3-6 through 3-9 can also be used to calculate pixel positions along a line with negative slope. If the absolute value of the slope is less than 1 and the start endpoint is at the left, we set $\Delta x = 1$ and calculate y values with Eq. 3-6.

When the start endpoint is at the right (for the same slope), we set $\Delta x = -1$ and obtain y positions from Eq. 3-8. Similarly, when the absolute value of a negative slope is 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 the two endpoint pixel positions. 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 (x_a, y_a), 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 xa is less than xb, the values of the increments in the x and y directions are 1 and m, respectively. If the greater change is in the x direction, but xa is greater than xb, then the decrements -1 and $-m$ are used to generate each new point on the line. Otherwise, we use a unit increment (or decrement) in the y direction and an x increment (or decrement) of $1/m$.

```
#include "device.h"

#define ROUND(a) ((int)(a+0.5))

void lineDDA (int xa, int ya, int xb, int yb)
{
  int dx = xb - xa, dy = yb - ya, steps, k;
  float xIncrement, yIncrement, x = xa, y = ya;

  if (abs (dx) > abs (dy)) steps = abs (dx);
  else steps = abs  dy);
  xIncrement = dx / (float) steps;
  yIncrement = 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 the direct use of 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 direction to step to pixel positions along the line path. The accumulation of roundoff 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 procedure `lineDDA` are still time-consuming. We can improve the performance of the DDA algorithm by separating the increments m and $1/m$ into integer and fractional parts so that all calculations are reduced to integer operations. A method for calculating $1/m$ increments in integer steps is discussed in Section 3-11. In the following sections, we consider more general scan-line procedures that can be applied to both lines and curves.

Bresenham's Line Algorithm

An accurate and efficient raster line-generating algorithm, developed by Bresen-

ham, scan converts lines using only incremental integer calculations that can be adapted to display circles and other curves. Figures 3-5 and 3-6 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. 3-5, 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. 3-6 shows a negative slope line path starting from the left endpoint at pixel position (50, 50). In this one, do we select the next pixel position as (51, 50) or as (51, 49)? These questions are answered with Bresenham's line algorithm by testing the sign of an integer parameter, whose value is proportional to the difference between the separations of the two pixel positions from the actual line path.

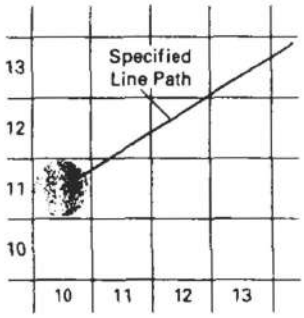


Figure 3-5
Section of a display screen where a straight line segment is to be plotted, starting from the pixel at column 10 on scan line 11.

To illustrate Bresenham's approach, we first consider the scan-conversion process for lines with positive slope less than 1. 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 3-7 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}. 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_1 and d_2 (Fig. 3-8). The y coordinate on the mathematical line at pixel column position x_k+1 is calculated as

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

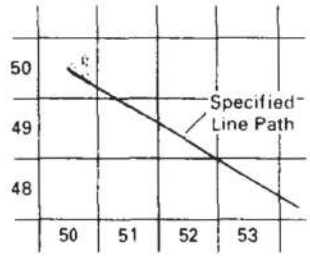


Figure 3-6
Section of a display screen where a negative slope line segment is to be plotted, starting from the pixel at column 50 on scan line 50.

Then

$$\begin{aligned} d_1 &= y - y_k \\ &= m(x_k + 1) + b - y_k \end{aligned}$$

and

$$\begin{aligned} d_2 &= (y_k + 1) - y \\ &= y_k + 1 - m(x_k + 1) - b \end{aligned}$$

The difference between these two separations is

$$d_1 - d_2 = 2m(x_k + 1) - 2y_k + 2b - 1 \tag{3-11}$$

A decision parameter p_k for the kth step in the line algorithm can be obtained by rearranging Eq. 3-11 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:

$$\begin{aligned} p_k &= \Delta x(d_1 - d_2) \\ &= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c \end{aligned} \tag{3-12}$$

The sign of p_k is the same as the sign of $d_1 - d_2$, 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

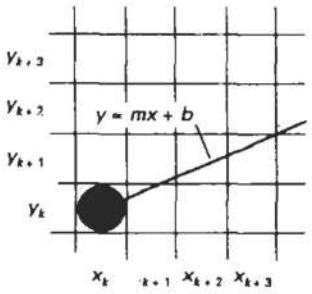


Figure 3-7
Section of the screen grid showing a pixel in column x_k on scan line y_k that is to be plotted along the path of a line segment with slope $0 < m < 1$.

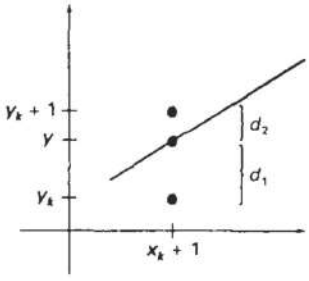


Figure 3-8
Distances between pixel positions and the line y coordinate at sampling position $x_k + 1$.

of 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, $d_1 < d_2$), 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. 3-12 as

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

Subtracting Eq. 3-12 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

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k) \tag{3-13}$$

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. 3-12 at the starting pixel position (x_0, y_0) and with m evaluated as $\Delta y / \Delta x$:

$$p_0 = 2\Delta y - \Delta x \tag{3-14}$$

We can summarize Bresenham line drawing for a line with a positive slope less than 1 in the following listed steps. 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 $|m| < 1$

1. Input the two line endpoints and store the left endpoint in (x_0, y_0).
2. Load (x_0, y_0) into the frame buffer; that is, plot the first point.
3. Calculate constants Δx, Δy, $2\Delta y$, and $2\Delta y - 2\Delta x$, and obtain the starting value for the decision parameter as

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

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

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

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

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

5. Repeat step 4 Δx 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

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

and the increments for calculating successive decision parameters are

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

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

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

A plot of the pixels generated along this line path is shown in Fig. 3-9.

An implementation of Bresenham line drawing for slopes in the range $0 < m < 1$ 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. The call to `setPixel` loads a preset color value into the frame buffer at the specified (x, y) pixel position.

```
#include "device.h"

void lineBres (int xa, int ya, int xb, int yb)
{
  int dx = abs (xa - xb), dy = abs (ya - yb);
  int p = 2 * dy - dx;
  int twoDy = 2 * dy, twoDyDx = 2 * (dy - dx);
  int x, y, xEnd;

  /* Determine which point to use as start, which as end */
  if (xa > xb) {
    x = xb;
    y = yb;
    xEnd = xa;
  }
  else {
```

```
    x = xa;
    y = ya;
    xEnd = xb;
  }
  setPixel (x, y);

  while (x < xEnd) {
    x++;
    if (p < 0)
      p += twoDy;
    else {
      y++;
      p += twoDyDx;
    }
    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, we interchange the roles of the x and y directions. That is, we step along the y direction in unit steps and calculate successive x values nearest the line path. Also, we could revise the program to plot pixels starting from either endpoint. If the initial position for a line with positive slope is the right endpoint, both x and y decrease as we step from right to left. To ensure that the same pixels are plotted regardless of the starting endpoint, we always choose the upper (or the lower) of the two candidate pixels whenever the two vertical separations from the line path are equal ($d_1 = d_2$). 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 with $|\Delta x| = |\Delta y|$ each can be loaded directly into the frame buffer without processing them through the line-plotting algorithm.

Parallel Line Algorithms

The line-generating algorithms we have discussed so far determine pixel positions sequentially. With a parallel computer, we can calculate pixel positions

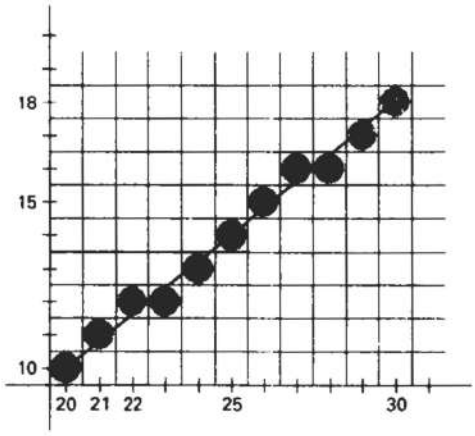


Figure 3-9
Pixel positions along the line path between endpoints (20, 10) and (30, 18), plotted with Bresenham's line algorithm.

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 $0 < m < 1$ 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

$$\Delta x_p = \frac{\Delta x + n_p - 1}{n_p} \tag{3-15}$$

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

$$x_k = x_0 + k\Delta x_p \tag{3-16}$$

As an example, suppose $\Delta x = 15$ and we have $n_p = 4$ processors. Then 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:

$$\Delta y_p = m\Delta x_p \tag{3-17}$$

At the kth partition, the starting y coordinate is then

$$y_k = y_0 + \text{round}(k\Delta y_p) \tag{3-18}$$

The initial decision parameter for Bresenham's algorithm at the start of the kth subinterval is obtained from Eq. 3-12:

$$p_k = (k\Delta x_p)(2\Delta y) - \text{round}(k\Delta y_p)(2\Delta x) + 2\Delta y - \Delta x \tag{3-19}$$

Each processor then calculates pixel positions over its assigned subinterval using the starting decision parameter value for that subinterval and the starting coordinates (x_k, y_k). We can also reduce the floating-point calculations to integer arithmetic in the computations for starting values y_k and p_k by substituting $m = \Delta y/\Delta x$ and rearranging terms. The extension of the parallel Bresenham algorithm to a line with slope greater than 1 is achieved by partitioning the line in the y di-

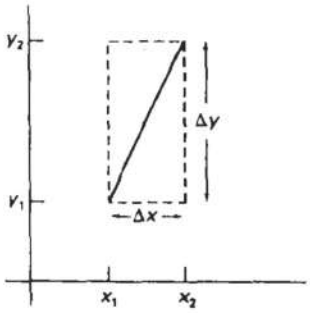


Figure 3-10
Bounding box for a line with coordinate extents Δx and Δy.

rection 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 (such as a Connection Machine CM-2 with over 65,000 processors), we can assign each processor to one pixel within some screen region. This approach can be adapted to line display by assigning one processor to each of the pixels within the limits of the line coordinate extents (*bounding rectangle*) 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. 3-10). Perpendicular distance d from the line in Fig. 3-10 to a pixel with coordinates (x, y) is obtained with the calculation

$$d = Ax + By + C \tag{3-20}$$

where

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

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

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

with

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

Once the constants A, B, and C have been evaluated for the line, each processor needs to 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 that processor. For a line with slope $|m| < 1$, each processor simply solves the line equation for y, given an x column value. For a line with slope magnitude greater than 1, 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 very efficiently using multiple processors.

3-3 LOADING THE FRAME BUFFER

When straight line segments and other objects are scan converted for display with a raster system, frame-buffer positions must be calculated. We have assumed that this is accomplished with the `setPixel` procedure, which stores intensity values for the pixels at corresponding addresses within the frame-buffer array. Scan-conversion algorithms generate pixel positions at successive unit in-

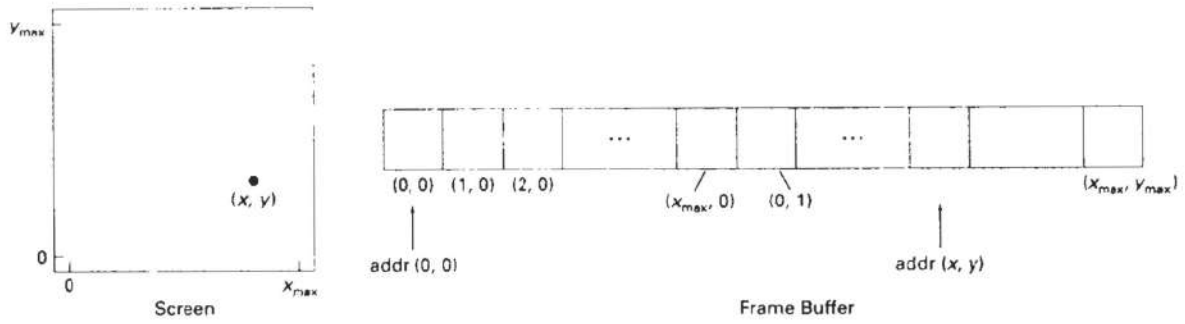


Figure 3-11
Pixel screen positions stored linearly in row-major order within the frame buffer.

tervals. This allows us to use incremental methods to calculate frame-buffer addresses.

As a specific example, suppose the frame-buffer array is addressed in row-major order and that pixel positions vary from (0, 0) at the lower left screen corner to (x_{max}, y_{max}) at the top right corner (Fig. 3-11). For a bilevel system (1 bit per pixel), the frame-buffer bit address for pixel position (x, y) is calculated as

$$\text{addr}(x, y) = \text{addr}(0, 0) + y(x_{max} + 1) + x \tag{3-21}$$

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):

$$\text{addr}(x + 1, y) = \text{addr}(x, y) + 1 \tag{3-22}$$

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

$$\text{addr}(x + 1, y + 1) = \text{addr}(x, y) + x_{max} + 2 \tag{3-23}$$

where the constant $x_{max} + 2$ is precomputed once for all line segments. Similar incremental calculations can be obtained from Eq. 3-21 for unit steps in the negative x and y screen directions. Each of these address calculations involves only a single integer addition.

Methods for implementing the `setPixel` procedure to store pixel intensity values 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 would include pixel width (number of bits), as well as the pixel screen location.

3-4 LINE FUNCTION

A procedure for specifying straight-line segments can be set up in a number of different forms. In PHIGS, GKS, and some other packages, the two-dimensional line function is

```
polyline (n, wcPoints)
```

where parameter n is assigned an integer value equal to the number of coordinate positions to be input, and `wcPoints` is the array of input world-coordinate values for line segment endpoints. This function is used to define a set of $n - 1$ connected straight line segments. Because series of connected line segments occur more often than isolated line segments in graphics applications, `polyline` provides a more general line function. To display a single straight-line segment, we set $n = 2$ and list the x and y values of the two endpoint coordinates in `wcPoints`.

As an example of the use of `polyline`, the following statements generate two connected line segments, with endpoints at (50, 100), (150, 250), and (250, 100):

```
wcPoints[1].x = 50;
wcPoints[1].y = 100;
wcPoints[2].x = 150;
wcPoints[2].y = 250;
wcPoints[3].x = 250;
wcPoints[3].y = 100;
polyline (3, wcPoints);
```

Coordinate references in the `polyline` function are stated as **absolute coordinate** values. This means that the values specified are the actual point positions in the coordinate system in use.

Some graphics systems employ line (and point) functions with **relative coordinate** specifications. In this case, coordinate values are stated as offsets from the last position referenced (called the **current position**). For example, if location (3, 2) 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, 1). An additional function is also available for setting the current position before the line routine is summoned. With these packages, a user lists only the single pair of offsets in the line command. This signals the system to display a line starting from the current position to a final position determined by the offsets. The current position is then updated to this final line position. A series of connected lines is produced with such packages by a sequence of line commands, one for each line section to be drawn. Some graphics packages provide options allowing the user to specify line endpoints using either relative or absolute coordinates.

Implementation of the `polyline` procedure is accomplished by first performing a series of coordinate transformations, then making a sequence of calls to a device-level line-drawing routine. In PHIGS, the input line endpoints are actually specified in modeling coordinates, which are then converted to world coordinates. Next, world coordinates are converted to normalized coordinates, then to device coordinates. We discuss the details for carrying out these two-dimensional coordinate transformations in Chapter 6. Once in device coordinates, we display the polyline by invoking a line routine, such as Bresenham's algorithm, $n - 1$ times to connect the n coordinate points. 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. To avoid setting the intensity of some endpoints twice, we could modify the line algorithm so that the last endpoint of each segment is not plotted. We discuss methods for avoiding overlap of displayed objects in more detail in Section 3-10.

3-5
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 most graphics packages. More generally, a single procedure can be provided to display either circular or elliptical curves.

Properties of Circles

A circle is defined as the set of points that are all at a given distance r from a center position (x_c, y_c) (Fig. 3-12). This distance relationship is expressed by the Pythagorean theorem in Cartesian coordinates as

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \tag{3-24}$$

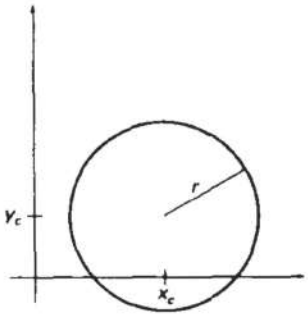


Figure 3-12
Circle with center coordinates (x_c, y_c) and radius r.

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

$$y = y_c \pm \sqrt{r^2 - (x_c - x)^2} \tag{3-25}$$

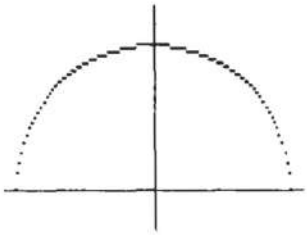


Figure 3-13
Positive half of a circle plotted with Eq. 3-25 and with $(x_c, y_c) = (0, 0)$.

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. 3-13. 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. 3-13 is to calculate points along the circular boundary using polar coordinates r and θ (Fig. 3-12). Expressing the circle equation in parametric polar form yields the pair of equations

$$\begin{aligned} x &= x_c + r\cos\theta \\ y &= y_c + r\sin\theta \end{aligned} \tag{3-26}$$

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. The step size chosen for θ depends on the application and the display device. Larger angular separations along the circumference can be connected with straight line segments to approximate the circular path. For a more continuous boundary on a raster display, we can set the step size at $1/r$. This plots pixel positions that are approximately one unit apart.

Computation can be reduced by considering the symmetry of circles. The shape of the circle is similar in each 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

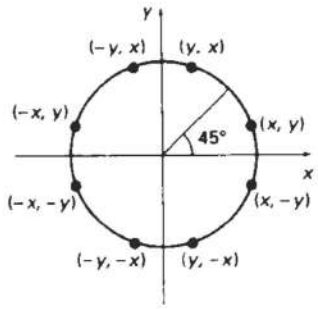


Figure 3-14
Symmetry of a circle. Calculation of a circle point (x, y) in one octant yields the circle points shown for the other seven octants.

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.3-14, 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 $x = y$.

Determining pixel positions along a circle circumference using either Eq. 3-24 or Eq. 3-26 still requires a good deal of computation time. The Cartesian equation 3-24 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 3-24, 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.

A method for direct distance comparison 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. Also, the error involved in locating pixel positions along any conic section using the midpoint test is limited to one-half the pixel separation.

Midpoint Circle Algorithm

As in the raster line algorithm, we sample at unit intervals and determine the closest pixel position to the specified circle path at each step. For a given radius r and screen center position (x_c, y_c), we can first set up our algorithm to calculate pixel positions around a circle path centered at the coordinate origin (0, 0). Then each calculated position (x, y) is moved to its proper screen position by adding x_c to x and y_c to y. Along the circle section from $x = 0$ to $x = y$ in the first quadrant, the slope of the curve varies from 0 to -1. 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 y positions is closer to the circle path at each step. Positions ifi the other seven octants are then obtained by symmetry.

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

$$f_{\text{circle}}(x, y) = x^2 + y^2 - r^2 \tag{3-27}$$

Any point (x, y) on the boundary of the circle with radius r satisfies the equation $f_{\text{circle}}(x, y) = 0$. 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:

$$f_{\text{circle}}(x, y) \begin{cases} < 0, & \text{if } (x, y) \text{ is inside the circle boundary} \\ = 0, & \text{if } (x, y) \text{ is on the circle boundary} \\ > 0, & \text{if } (x, y) \text{ is outside the circle boundary} \end{cases} \tag{3-28}$$

The circle-function tests in 3-28 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 3-15 shows the midpoint between the two candidate pixels at sampling position $x_k + 1$. Assuming 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 3-27 evaluated at the midpoint between these two pixels:

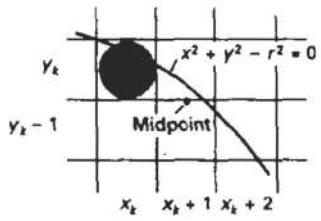


Figure 3-15
Midpoint between candidate pixels at sampling position x_k+1 along a circular path.

$$\begin{aligned} p_k &= f_{\text{circle}}\left(x_k + 1, y_k - \frac{1}{2}\right) \\ &= (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2 \end{aligned} \tag{3-29}$$

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 scanline $y_k - 1$.

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

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

or

$$p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1 \tag{3-30}$$

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

$$\begin{aligned} 2x_{k+1} &= 2x_k + 2 \\ 2y_{k+1} &= 2y_k - 2 \end{aligned}$$

At the start position $(0, r)$, these two terms have the values 0 and $2r$, respectively. Each successive value is obtained by adding 2 to the previous value of $2x$ and subtracting 2 from the previous value of $2y$.

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

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

or

$$p_0 = \frac{5}{4} - r \tag{3-31}$$

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), and obtain 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 on (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 $x \geq y$.

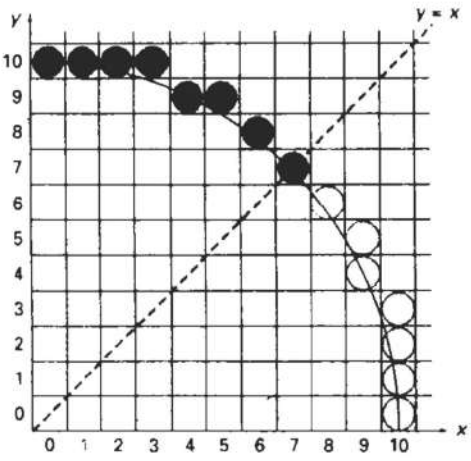


Figure 3-16
Selected pixel positions (solid circles) along a circle path with radius $r = 10$ centered on the origin, using the midpoint circle algorithm. Open circles show the symmetry positions in the first quadrant.

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 decision parameter values and positions along the circle path are calculated using the midpoint method as

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

A plot of the generated pixel positions in the first quadrant is shown in Fig. 3-16.

The following procedure displays a raster circle on a bilevel monitor using the midpoint algorithm. Input to the procedure are the coordinates for the circle center and the radius. Intensities for pixel positions along the circle circumference are loaded into the frame-buffer array with calls to the `setPixel` routine.

```
#include "device.h"

void circleMidpoint (int xCenter, int yCenter, int radius)
{
  int x = 0;
  int y = radius;
  int p = 1 - radius;
  void circlePlotPoints (int, int, int, int);

  /* Plot first set of points */
  circlePlotPoints (xCenter, yCenter, x, y);

  while (x < y) {
    x++;
    if (p < 0)
      p += 2 * x + 1;
    else {
      y--;
      p += 2 * (x - y) + 1;
    }
    circlePlotPoints (xCenter, yCenter, x, y);
  }
}

void circlePlotPoints (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);
  setPixel (xCenter + y, yCenter + x);
  setPixel (xCenter - y, yCenter + x);
  setPixel (xCenter + y, yCenter - x);
  setPixel (xCenter - y, yCenter - x);
}
```

3-6 ELLIPSE-GENERATING ALGORITHMS

Loosely stated, an ellipse is an elongated circle. Therefore, elliptical curves can be generated by modifying circle-drawing procedures to take into account the different dimensions of an ellipse along the major and minor axes.

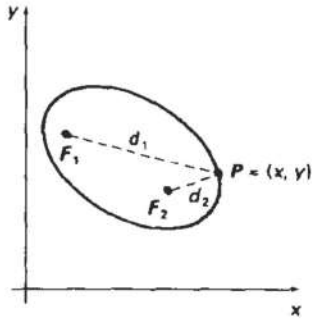


Figure 3-17
Ellipse generated about foci F_1 and F_2.

Properties of Ellipses

An ellipse is defined as the set of points such that the sum of the distances from two fixed positions (foci) is the same for all points (Fig. 3-17). If the distances to the two foci 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

$$d_1 + d_2 = \text{constant} \tag{3-32}$$

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

$$\sqrt{(x - x_1)^2 + (y - y_1)^2} + \sqrt{(x - x_2)^2 + (y - y_2)^2} = \text{constant} \tag{3-33}$$

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

$$Ax^2 + By^2 + Cxy + Dx + Ey + F = 0 \tag{3-34}$$

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, 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. 3-33. Then, the coefficients in Eq. 3-34 can be evaluated 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. 3-18, 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 of the ellipse shown in Fig. 3-18 can be written in terms of the ellipse center coordinates and parameters r_x and r_y as

$$\left(\frac{x - x_c}{r_x}\right)^2 + \left(\frac{y - y_c}{r_y}\right)^2 = 1 \tag{3-35}$$

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

$$\begin{aligned} x &= x_c + r_x \cos\theta \\ y &= y_c + r_y \sin\theta \end{aligned} \tag{3-36}$$

Symmetry considerations can be used to further 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 we obtain positions in the remaining three quadrants by symmetry (Fig 3-19).

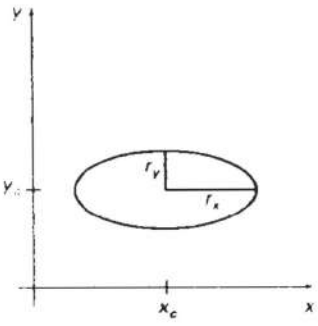


Figure 3-18
Ellipse centered at (x_c, y_c) with semimajor axis r_x and semiminor axis r_y.

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 points (x, y) for an ellipse in standard position centered on the origin, and then we shift the points so the ellipse is centered at (x_c, y_c). If we wish also to display the ellipse in nonstandard position, we could then rotate the ellipse about its center coordinates to reorient the major and minor axes. 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 3-20 shows the division of the first quadrant according to the slope of an ellipse with $r_x < r_y$. We process this quadrant by taking unit steps in the x direction where the slope of the curve has a magnitude less than 1, and taking unit steps in the y direction where the slope has a magnitude greater than 1.

Regions 1 and 2 (Fig. 3-20), can be processed in various ways. We can start at position $(0, r_y)$ and step clockwise along the elliptical path in the first quadrant,

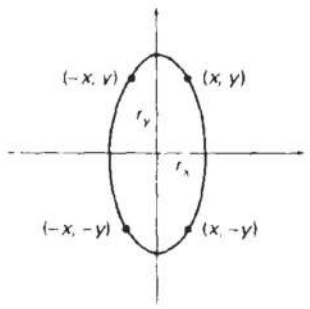


Figure 3-19
Symmetry of an ellipse. Calculation of a point (x, y) in one quadrant yields the ellipse points shown for the other three quadrants.

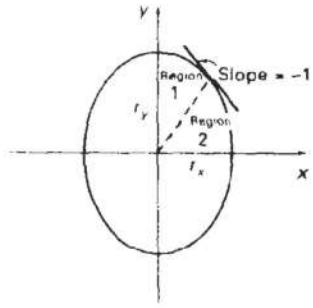


Figure 3-20
Ellipse processing regions. Over region 1, the magnitude of the ellipse slope is less than 1; over region 2, the magnitude of the slope is greater than 1.

shifting from unit steps in x to unit steps in y when the slope becomes less than -1. 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. 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. 3-35 with $(x_c, y_c) = (0, 0)$ as

$$f_{\text{ellipse}}(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2 \tag{3-37}$$

which has the following properties:

$$f_{\text{ellipse}}(x, y) \begin{cases} < 0, & \text{if } (x, y) \text{ is inside the ellipse boundary} \\ = 0, & \text{if } (x, y) \text{ is on the ellipse boundary} \\ > 0, & \text{if } (x, y) \text{ is outside the ellipse boundary} \end{cases} \tag{3-38}$$

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. 3-20). Then we switch to unit steps in the y direction over the remainder of the curve in the first quadrant. At each step, we need to test the value of the slope of the curve. The ellipse slope is calculated from Eq. 3-37 as

$$\frac{dy}{dx} = -\frac{2r_y^2 x}{2r_x^2 y} \tag{3-39}$$

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

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

Therefore, we move out of region 1 whenever

$$2r_y^2 x \geq 2r_x^2 y \tag{3-40}$$

Figure 3-21 shows the midpoint between the two candidate pixels at sampling position $x_k + 1$ in the first region. Assuming position (x_k, y_k) has been selected at the previous step, we determine the next position along the ellipse path by evaluating the decision parameter (that is, the ellipse function 3-37) at this midpoint:

$$\begin{aligned} p1_k &= f_{\text{ellipse}}\left(x_k + 1, y_k - \frac{1}{2}\right) \\ &= r_y^2 (x_k + 1)^2 + r_x^2 \left(y_k - \frac{1}{2}\right)^2 - r_x^2 r_y^2 \end{aligned} \tag{3-41}$$

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 $y_k - 1$.

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

$$p1_{k+1} = f_{\text{ellipse}}\left(x_{k+1} + 1, y_{k+1} - \frac{1}{2}\right)$$

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

or

$$p1_{k+1} = p1_k + 2r_y^2(x_k + 1) + r_y^2 + r_x^2\left[\left(y_{k+1} - \frac{1}{2}\right)^2 - \left(y_k - \frac{1}{2}\right)^2\right] \qquad (3\text{-}42)$$

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:

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

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

$$2r_y^2 x = 0 \qquad (3\text{-}43)$$

$$2r_x^2 y = 2r_x^2 r_y \qquad (3\text{-}44)$$

As x and y are incremented, updated values are obtained by adding $2r_y^2$ to 3-43 and subtracting $2r_x^2$ from 3-44. The updated values are compared at each step, and we move from region 1 to region 2 when condition 3-40 is satisfied.

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

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

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

or

$$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2 \qquad (3\text{-}45)$$

Over region 2, we sample at unit steps in the negative y direction, and the midpoint is now taken between horizontal pixels at each step (Fig. 3-22). For this region, the decision parameter is evaluated as

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

$$= r_y^2\left(x_k + \frac{1}{2}\right)^2 + r_x^2(y_k - 1)^2 - r_x^2 r_y^2 \qquad (3\text{-}46)$$

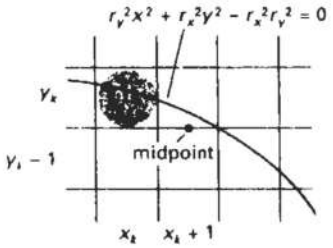


Figure 3-21
Midpoint between candidate pixels at sampling position $x_k + 1$ along an elliptical path.

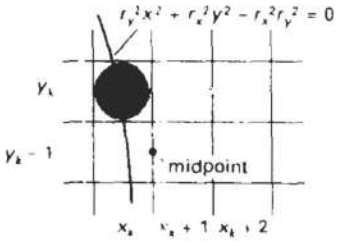


Figure 3-22
Midpoint between candidate pixels at sampling position $y_k - 1$ along an elliptical path.

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

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

$$\begin{aligned} p2_{k+1} &= f_{\text{ellipse}}\left(x_{k+1} + \frac{1}{2}, y_{k+1} - 1\right) \\ &= r_y^2\left(x_{k+1} + \frac{1}{2}\right)^2 + r_x^2[(y_k - 1) - 1]^2 - r_x^2 r_y^2 \end{aligned} \tag{3-47}$$

or

$$p2_{k+1} = p2_k - 2r_x^2(y_k - 1) + r_x^2 + r_y^2\left[\left(x_{k+1} + \frac{1}{2}\right)^2 - \left(x_k + \frac{1}{2}\right)^2\right] \tag{3-48}$$

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

$$\begin{aligned} p2_0 &= f_{\text{ellipse}}\left(x_0 + \frac{1}{2}, y_0 - 1\right) \\ &= r_y^2\left(x_0 + \frac{1}{2}\right)^2 + r_x^2(y_0 - 1)^2 - r_x^2 r_y^2 \end{aligned} \tag{3-49}$$

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.

The midpoint algorithm can be adapted to generate an ellipse in nonstandard position using the ellipse function Eq. 3-34 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 algorithm to determine curve positions, 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 only need 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. A summary of the midpoint ellipse algorithm is listed in the following steps:

Midpoint Ellipse Algorithm

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

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

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

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

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

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

Otherwise, the next point along the circle 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 using the last point (x_0, y_0) calculated in region 1 as

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

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 circle 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.

6. 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:

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

8. Repeat the steps for region 1 until $2r_y^2 x \geq 2r_x^2 y$.

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

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

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

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

Successive decision parameter values and positions along the ellipse path are calculated using the midpoint method as

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

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

For region 2, the initial point is $(x_0, y_0) = (7, 3)$ and the initial decision parameter is

$$p2_0 = f\left(7 + \frac{1}{2}, 2\right) = -151$$

The remaining positions along the ellipse path in the first quadrant are then calculated as

k	$p2_k$	(x_{k+1}, y_{k+1})	$2r_y^2x_{k+1}$	$2r_x^2y_{k+1}$
0	−151	(8, 2)	576	256
1	233	(8, 1)	576	128
2	745	(8, 0)	—	—

A plot of the selected positions around the ellipse boundary within the first quadrant is shown in Fig. 3-23.

In the following procedure, the midpoint algorithm is used to display an ellipse with input parameters `Rx`, `Ry`, `xCenter`, and `yCenter`. Positions along the

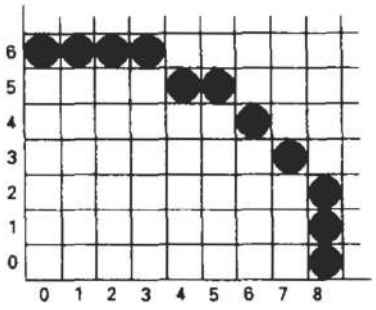


Figure 3-23
Positions along an elliptical path centered on the origin with $r_x = 8$ and $r_y = 6$ using the midpoint algorithm to calculate pixel addresses in the first quadrant.

curve in the first quadrant are generated and then shifted to their proper screen positions. Intensities for these positions and the symmetry positions in the other three quadrants are loaded into the frame buffer using the `setPixel` routine.

```
#include "device.h"

#define ROUND(a) ((int)(a+0.5))

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 first set of points */
  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);
}
```

3-7 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 straightforward method for displaying a specified curve function is to approximate it with straight line segments. Parametric representations are useful in this case for obtaining equally spaced line endpoint positions along the curve path. 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 graph a data set of discrete coordinate points. 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 along 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:

$$Ax^2 + By^2 + Cxy + Dx + Ey + F = 0 \tag{3-50}$$

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

$$B^2 - 4AC \begin{cases} < 0, & \text{generates an ellipse (or circle)} \\ = 0, & \text{generates a parabola} \\ > 0, & \text{generates a hyperbola} \end{cases} \tag{3-51}$$

For example, we get the circle equation 3-24 when $A = B = 1$, $C = 0$, $D = -2x_c$, $E = -2y_c$, and $F = x_c^2 + y_c^2 - r^2$. Equation 3-50 also describes the "degenerate" conics: points and straight lines.

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 ellipses; and an object projected into a uniform gravitational field travels along a parabolic trajectory. Figure 3-24 shows a parabolic path in standard position for a gravitational field acting in the negative y direction. The explicit equation for the parabolic trajectory of the object shown can be written as

$$y = y_0 + a(x - x_0)^2 + b(x - x_0) \tag{3-52}$$

with constants a and b determined by the initial velocity v_0 of the object and the acceleration g due to the uniform gravitational force. We can also describe such parabolic motions with parametric equations using a time parameter t, measured in seconds from the initial projection point:

$$\begin{aligned} x &= x_0 + v_{x0}t \\ y &= y_0 + v_{y0}t - \frac{1}{2}gt^2 \end{aligned} \tag{3-53}$$

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

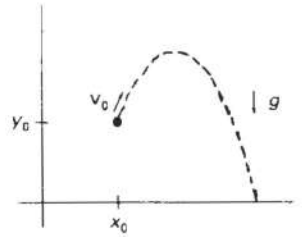


Figure 3-24
Parabolic path of an object tossed into a downward gravitational field at the initial position (x_0, y_0).

Hyperbolic motions (Fig. 3-25) 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. 3-25) 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. 3-25 as

$$\left(\frac{x}{r_x}\right)^2 - \left(\frac{y}{r_y}\right)^2 = 1 \tag{3-54}$$

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 3-35 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. We will return to the discussion of animation applications and methods in more detail in Chapter 16. And in Chapter 10, we discuss applications of computer graphics in scientific visualization.

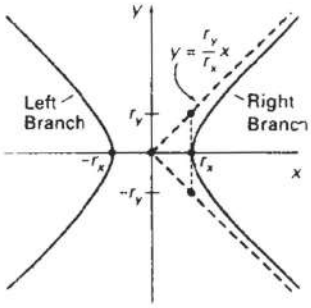


Figure 3-25
Left and right branches of a hyperbola in standard position with symmetry axis along the x axis.

Parabolas and hyperbolas possess a symmetry axis. For example, the parabola described by Eq. 3-53 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. 3-50 and then use the selected function to set up expressions for the decision parameters in the two regions.

Polynomials and Spline Curves

A polynomial function of nth degree in x is defined as

$$\begin{aligned} y &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} + a_n x^n \end{aligned} \tag{3-55}$$

where n is a nonnegative integer and the a_k are constants, with $a_n \neq 0$. We get a quadratic when $n = 2$; a cubic polynomial when $n = 3$; a quartic 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 done by specifying a few points to define the general curve contour, then fitting the selected points 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

$$x = a_{x0} + a_{x1}u + a_{x2}u^2 + a_{x3}u^3 \tag{3-56}$$

$$y = a_{y0} + a_{y1}u + a_{y2}u^2 + a_{y3}u^3 \tag{3-57}$$

Figure 3-26
A spline curve formed with individual cubic polynomial sections between specified coordinate points.

where parameter u varies over the interval 0 to 1. Values for the coefficients of u in the parametric 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. 3-26). 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 the various spline-generating methods are explored in Chapter 10.

3-8 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 90° to 45° 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 then calculated throughout each subarc, and positions in the other circle octants are then 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. Pixel positions in the other quadrants are determined by symmetry. A screen-partitioning scheme for circles and ellipses is to assign each scan line crossing 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.

3-9 CURVE FUNCTIONS

Routines for circles, splines, and other commonly used curves are included in many graphics packages. The PHIGS standard does not provide explicit functions for these curves, but it does include the following general curve function:

```
generalizedDrawingPrimitive (n, wcPoints, id, datalist)
```

where `wcPoints` is a list of `n` coordinate positions, `datalist` contains noncoordinate data values, and parameter `id` selects the desired function. At a particular installation, a circle might be referenced with `id` = 1, an ellipse with `id` = 2, and so on.

As an example of the definition of curves through this PHIGS function, a circle (`id` = 1, say) could be specified by assigning the two center coordinate values to `wcpoints` and assigning the radius value to `datalist`. The generalized drawing primitive would then reference the appropriate algorithm, such as the midpoint method, to generate the circle. With interactive input, a circle could be defined with two coordinate points: the center position and a point on the circumference. Similarly, interactive specification of an ellipse can be done with three points: the two foci and a point on the ellipse boundary, all stored in `wcpoints`. For an ellipse in standard position, *wcpoints* could be assigned only the center coordinates, with *datalist* assigned the values for r_x and r_y. Splines defined with control points would be generated by assigning the control point coordinates to `wcpoints`.

Functions to generate circles and ellipses often include the capability of drawing curve sections by specifying parameters for the line endpoints. Expanding the parameter list allows specification of the beginning and ending angular values for an arc, as illustrated in Fig. 3-27. Another method for designating a cir-

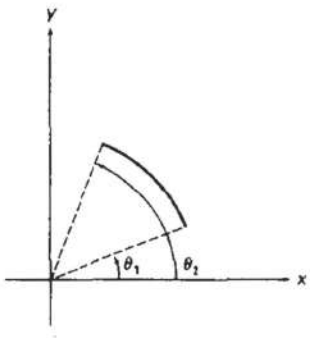


Figure 3-27
Circular arc specified by beginning and ending angles. Circle center is at the coordinate origin.

cular or elliptical arc is to input the beginning and ending coordinate positions of the arc.

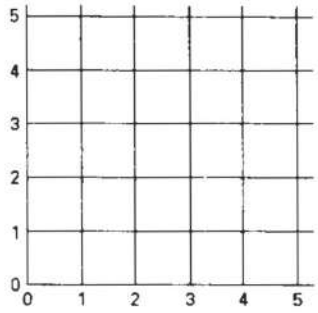


Figure 3-28
Lower-left section of the screen grid referencing integer coordinate positions.

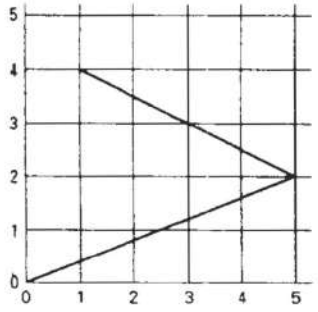


Figure 3-29
Line path for a series of connected line segments between screen grid coordinate positions.

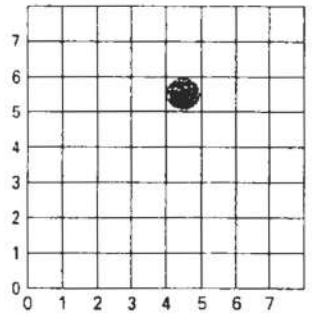


Figure 3-30
Illuminated pixel at raster position (4, 5).

3-10 PIXEL ADDRESSING AND OBJECT GEOMETRY

So far we have assumed that all input positions were given in terms of scan-line number and pixel-position number across the scan line. As we saw in Chapter 2, there are, in general, several coordinate references associated with the specification and generation of a picture. Object descriptions are given in a world-reference frame, chosen to suit a particular application, and input world coordinates are ultimately converted to screen display positions. World descriptions of objects are given in terms of precise coordinate positions, which are infinitesimally small mathematical points. Pixel coordinates, however, reference finite screen areas. If we want to preserve the specified geometry of world objects, we need to compensate for the mapping of mathematical input points to finite pixel areas. One way to do this is simply to adjust the dimensions of displayed objects to account for the amount of overlap of pixel areas with the object boundaries. 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.

Screen Grid Coordinates

An alternative to addressing display positions in terms of pixel centers is to reference screen coordinates with respect to the grid of horizontal and vertical pixel boundary lines spaced one unit apart (Fig. 3-28). A screen coordinate position is then the pair of integer values identifying a grid intersection position between two pixels. For example, the mathematical line path for a polyline with screen endpoints (0, 0), (5, 2), and (1, 4) is shown in Fig. 3-29.

With the coordinate origin at the lower left of the screen, each pixel area can be referenced by the integer grid coordinates of its lower left corner. Figure 3-30 illustrates this convention for an 8 by 8 section of a raster, with a single illuminated pixel at screen coordinate position (4, 5). In general, we 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 scheme has several advantages: It avoids half-integer pixel boundaries, it facilitates precise object representations, and it simplifies the processing involved in many scan-conversion algorithms and in 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 are now simply a measure of screen grid separation differences, rather than separation differences from pixel centers.

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

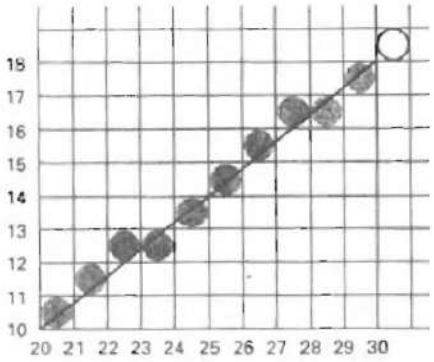


Figure 3-31
Line path and corresponding pixel display for input screen grid endpoint coordinates (20, 10) and (30, 18).

for an object, we need to account for the finite size of pixels when we transform the object definition to a screen display.

Figure 3-31 shows the line plotted in the Bresenham line-algorithm example of Section 3-2. 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 3-2, 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. If we think of screen coordinates as addressing pixel boundaries, as shown in Fig. 3-31, we plot a line using only those pixels that are "interior" to the line path; that is, only those pixels that are between the line endpoints. For our example, we would plot the leftmost pixel at (20, 10) and the rightmost pixel at (29, 17). This displays a line that

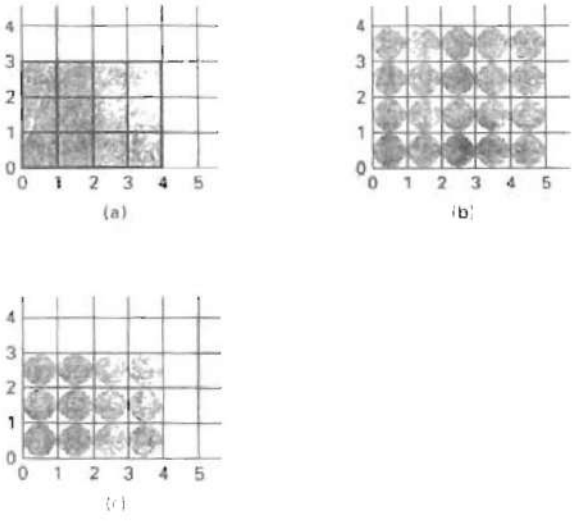


Figure 3-32
Conversion of rectangle (a) with vertices at screen coordinates (0, 0), (4, 0), (4, 3), and (0, 3) into display (b) that includes the right and top boundaries and into display (c) that maintains geometric magnitudes.

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 only with those pixels that are interior to the object boundaries. The rectangle defined with the screen coordinate vertices shown in Fig. 3-32(a), for example, is larger when we display it filled with pixels up to and including the border pixel lines joining the specified vertices. As defined, the area of the rectangle is 12 units, but as displayed in Fig. 3-32(b), it has an area of 20 units. In Fig. 3-32(c), the original rectangle measurements are maintained by displaying

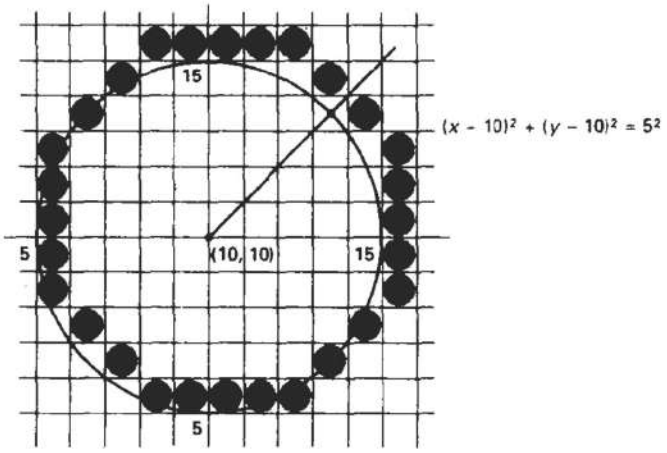


Figure 3-33
Circle path and midpoint circle algorithm plot of a circle with radius 5 in screen coordinates.

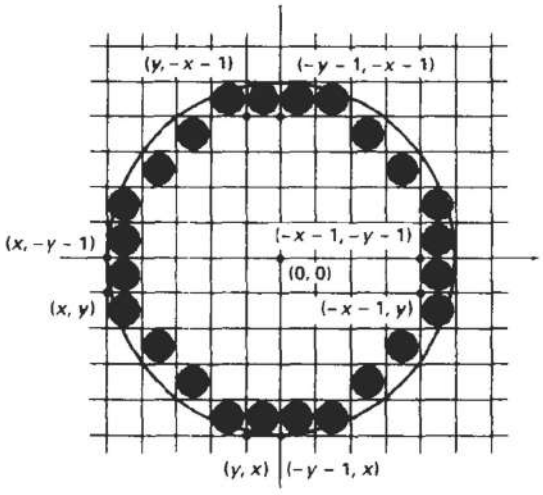


Figure 3-34
Modification of the circle plot in Fig. 3-33 to maintain the specified circle diameter of 10.

only the internal pixels. The right boundary of the input rectangle is at $x = 4$. To maintain this boundary in the display, we set the rightmost pixel grid coordinate at $x = 3$. The pixels in this vertical column then 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 width along object boundaries can be applied to other polygons and to curved figures so that the raster display maintains the input object specifications. A circle of radius 5 and center position (10, 10), for instance, would be displayed as in Fig. 3-33 by the midpoint circle algorithm using screen grid 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. 3-34. One way to do this is to generate points clockwise along the circular arc in the third quadrant, starting at screen coordinates (10, 5). For each generated point, the other seven circle symmetry points are generated by decreasing the x coordinate values by 1 along scan lines and decreasing the y coordinate values by 1 along pixel columns. Similar methods are applied in ellipse algorithms to maintain the specified proportions in the display of an ellipse.

3-11
FILLED-AREA PRIMITIVES

A standard output primitive in general graphics packages is a solid-color or patterned polygon area. Other kinds of area primitives are sometimes available, but polygons are easier to process since they have linear boundaries

There are two basic approaches to area filling on raster systems. One way to fill an area is to determine the overlap intervals for scan lines that cross the area. Another method for area filling is to start from a given interior position and paint outward from this point until we encounter the specified boundary conditions. The scan-line approach is typically used in general graphics packages to fill polygons, circles, ellipses, and other simple curves. Fill methods starting from an interior point are useful with more complex boundaries and in interactive painting systems. In the following sections, we consider methods for solid fill of specified areas. Other fill options are discussed in Chapter 4.

Scan-Line Polygon Fill Algorithm

Figure 3-35 illustrates the scan-line procedure for solid filling of polygon areas. For each scan line crossing a polygon, the area-fill algorithm locates the intersection points of the scan line with the polygon edges. These intersection points are then sorted from left to right, and the corresponding frame-buffer positions between each intersection pair are set to the specified fill color. In the example of Fig. 3-35, the four pixel intersection positions with the polygon boundaries define two stretches of interior pixels from $x = 10$ to $x = 14$ and from $x = 18$ to $x = 24$.

Some scan-line intersections at polygon vertices require special handling. A scan line passing through a vertex intersects two polygon edges at that position, adding two points to the list of intersections for the scan line. Figure 3-36 shows two scan lines at positions y and y' that intersect edge endpoints. Scan line y intersects five polygon edges. Scan line y', however, intersects an even number of edges although it also passes through a vertex. Intersection points along scan line

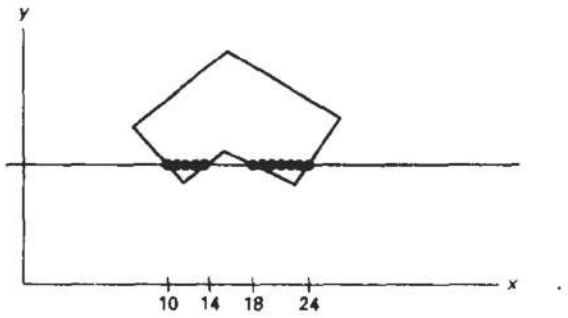


Figure 3-35
Interior pixels along a scan line passing through a polygon area.

y' correctly identify the interior pixel spans. But with scan line y, we need to do some additional processing to determine the correct interior points.

The topological difference between scan line y and scan line y' in Fig. 3-36 is identified by noting the position of the intersecting edges relative to the scan line. For scan line y, the two intersecting edges sharing a vertex are on opposite sides of the scan line. But for scan line y', the two intersecting edges are both above the scan line. Thus, the vertices that require additional processing are those that have connecting edges on opposite sides of the scan line. We can identify these vertices by tracing around the polygon boundary either in clockwise or counterclockwise order and observing the relative changes in vertex y coordinates as we move from one edge to the next. If the endpoint y values of two consecutive edges monotonically increase or decrease, we need to count the middle vertex as a single intersection point for any scan line passing through that vertex. Otherwise, the shared vertex represents a local extremum (minimum or maximum) on the polygon boundary, and the two edge intersections with the scan line passing through that vertex can be added to the intersection list.

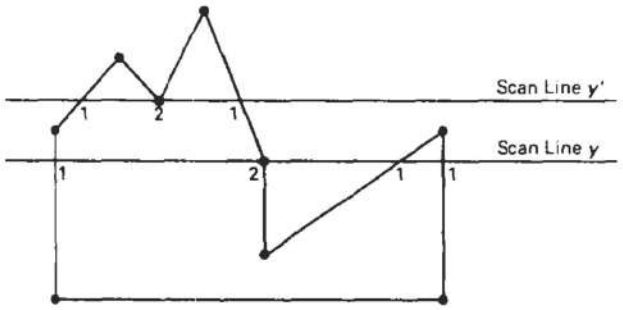


Figure 3-36
Intersection points along scan lines that intersect polygon vertices. Scan line y generates an odd number of intersections, but scan line y' generates an even number of intersections that can be paired to identify correctly the interior pixel spans.

One way to resolve the question as to whether we should count a vertex as one intersection or two is to shorten some polygon edges to split those vertices that should be counted as one intersection. We can process nonhorizontal edges around the polygon boundary in the order specified, either clockwise or counterclockwise. As we process each edge, we can check to determine whether that edge and the next nonhorizontal edge have either monotonically increasing or decreasing endpoint y values. If so, the lower edge can be shortened to ensure that only one intersection point is generated for the scan line going through the common vertex joining the two edges. Figure 3-37 illustrates shortening of an edge. When the endpoint y coordinates of the two edges are increasing, the y value of the upper endpoint for the current edge is decreased by 1, as in Fig. 3-37(a). When the endpoint y values are monotonically decreasing, as in Fig. 3-37(b), we decrease the y coordinate of the upper endpoint of the edge following the current edge.

Calculations performed in scan-conversion and other graphics algorithms typically take advantage of various **coherence** properties of a scene that is to be displayed. What we mean by coherence is simply that the properties of one part of a scene are related in some way to other parts of the scene so that the relationship can be used to reduce processing. Coherence methods often involve incremental calculations applied along a single scan line or between successive scan lines. In determining edge intersections, we can set up incremental coordinate calculations along any edge by exploiting the fact that the slope of the edge is constant from one scan line to the next. Figure 3-38 shows two successive scan lines crossing a left edge of a polygon. The slope of this polygon boundary line can be expressed in terms of the scan-line intersection coordinates:

$$m = \frac{y_{k+1} - y_k}{x_{k+1} - x_k} \tag{3-58}$$

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

$$y_{k+1} - y_k = 1 \tag{3-59}$$

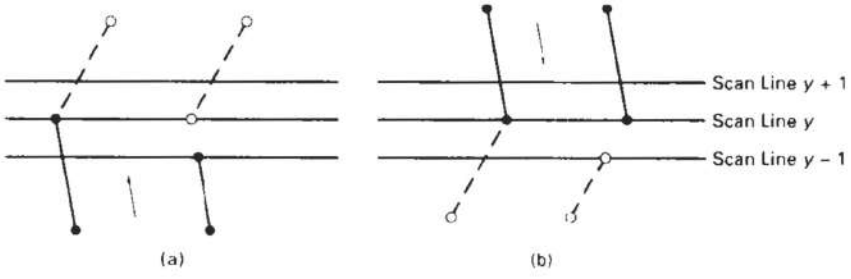


Figure 3-37
Adjusting endpoint y values for a polygon, as we process edges in order around the polygon perimeter. The edge currently being processed is indicated as a solid line. In (a), the y coordinate of the upper endpoint of the current edge is decreased by 1. In (b), the y coordinate of the upper endpoint of the next edge is decreased by 1.

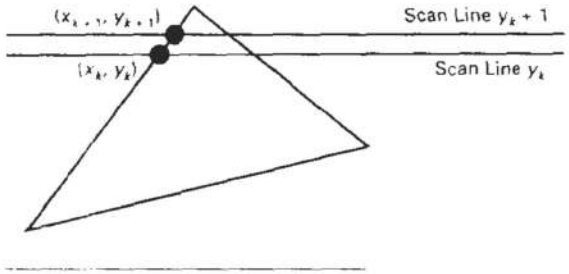


Figure 3-38
Two successive scan lines intersecting a polygon boundary.

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

$$x_{k+1} = x_k + \frac{1}{m} \tag{3-60}$$

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

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

$$x_k = x_0 + \frac{k}{m} \tag{3-61}$$

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

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

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

$$x_{k+1} = x_k + \frac{\Delta x}{\Delta y} \tag{3-62}$$

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

As an example of integer incrementing, suppose we have an edge with slope $m = 7/3$. At the initial scan line, we set the counter to 0 and the counter in-

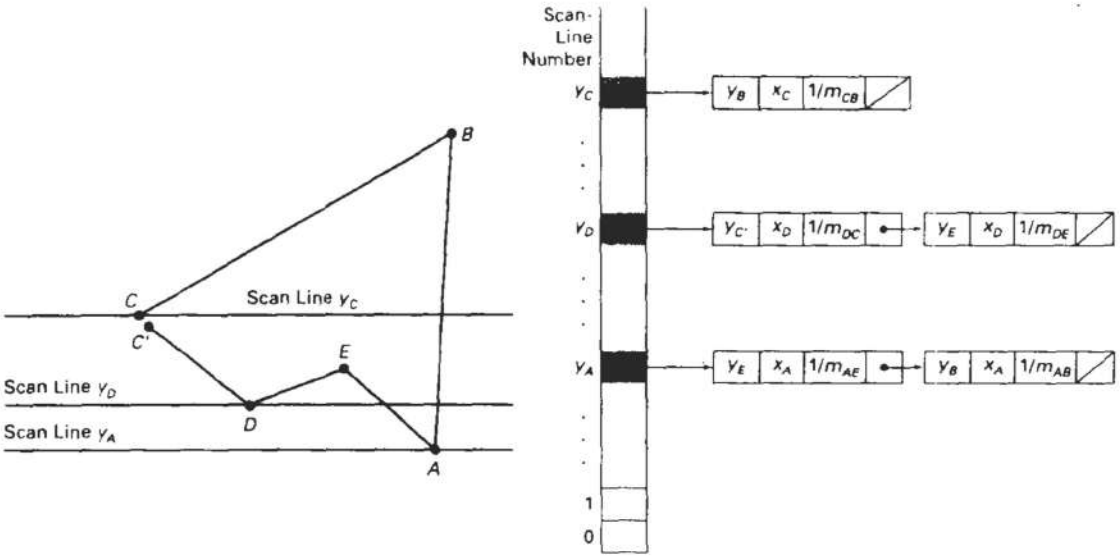


Figure 3-39
A polygon and its sorted edge table, with edge $\overline{DC}$ shortened by one unit in the y direction.

crement to 3. As we move up to the next three scan lines along this edge, the counter is successively assigned the values 3, 6, and 9. On the third scan line above the initial scan line, the counter now has a value greater than 7. So we increment the x-intersection coordinate by 1, and reset the counter to the value $9 - 7 = 2$. We continue determining the scan-line intersections in this way until we reach the upper endpoint of the edge. Similar calculations are carried out to obtain intersections for edges with negative slopes.

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

To efficiently perform a polygon fill, we can first store the polygon boundary in a *sorted edge table* that contains all the information necessary to process the scan lines efficiently. Proceeding around the edges in either a clockwise or a counterclockwise order, we can use a bucket sort to store the edges, sorted on the smallest y value of each edge, in the correct scan-line positions. Only nonhorizontal edges are entered into the sorted edge table. As the edges are processed, we can also shorten certain edges to resolve the vertex-intersection question. Each entry in the table for a particular scan line contains the maximum y value for that edge, the x-intercept value (at the lower vertex) for the edge, and the inverse slope of the edge. For each scan line, the edges are in sorted order from left to right. Figure 3-39 shows a polygon and the associated sorted edge table.

Next, we process the scan lines from the bottom of the polygon to its top, producing an *active edge list* for each scan line crossing the polygon boundaries. The active edge list for a scan line contains all edges crossed by that scan line, with iterative coherence calculations used to obtain the edge intersections.

Implementation of edge-intersection calculations can also be facilitated by storing Δx and Δy values in the sorted edge table. Also, to ensure that we correctly fill the interior of specified polygons, we can apply the considerations discussed in Section 3-10. For each scan line, we fill in the pixel spans for each pair of x-intercepts starting from the leftmost x-intercept value and ending at one position before the rightmost x intercept. And each polygon edge can be shortened by one unit in the y direction at the top endpoint. These measures also guarantee that pixels in adjacent polygons will not overlap each other.

The following procedure performs a solid-fill scan conversion for an input set of polygon vertices. For each scan line within the vertical extents of the polygon, an active edge list is set up and edge intersections are calculated. Across each scan line, the interior fill is then applied between successive pairs of edge intersections, processed from left to right.

```
#include "device.h"

typedef struct tEdge {
  int yUpper;
  float xIntersect, dxPerScan;
  struct tEdge * next;
} Edge;

/* Inserts edge into list in order of increasing xIntersect field. */
void insertEdge (Edge * list, Edge * edge)
{
  Edge * p, * q = list;

  p = q->next;
  while (p != NULL) {
    if (edge->xIntersect < p->xIntersect)
      p = NULL;
    else {
      q = p;
      p = p->next;
    }
  }
  edge->next = q->next;
  q->next = edge;
}

/* For an index, return y-coordinate of next nonhorizontal line */
int yNext (int k, int cnt, dcPt * pts)
{
  int j;

  if ((k+1) > (cnt-1))
    j = 0;
  else
    j = k + 1;
  while (pts[k].y == pts[j].y)
    if ((j+1) > (cnt-1))
      j = 0;
    else
```

```
      j++;
  return (pts[j].y);
}

/* Store lower-y coordinate and inverse slope for each edge.  Adjust
   and store upper-y coordinate for edges that are the lower member
   of a monotonically increasing or decreasing pair of edges */
void makeEdgeRec
  (dcPt lower, dcPt upper, int yComp, Edge * edge, Edge * edges[])
{
  edge->dxPerScan =
    (float) (upper.x - lower.x) / (upper.y - lower.y);
  edge->xIntersect = lower.x;
  if (upper.y < yComp)
    edge->yUpper = upper.y - 1;
  else
    edge->yUpper = upper.y;
  insertEdge (edges[lower.y], edge);
}

void buildEdgeList (int cnt, dcPt * pts, Edge * edges[])
{
  Edge * edge;
  dcPt v1, v2;
  int i, yPrev = pts[cnt - 2].y;

  v1.x = pts[cnt-1].x; v1.y = pts[cnt-1].y;
  for (i=0; i<cnt; i++) {
    v2 = pts[i];
    if (v1.y != v2.y) {                /* nonhorizontal line */
      edge = (Edge *) malloc (sizeof (Edge));
      if (v1.y < v2.y)                 /* up-going edge      */
        makeEdgeRec (v1, v2, yNext (i, cnt, pts), edge, edges);
      else                             /* down-going edge    */
        makeEdgeRec (v2, v1, yPrev, edge, edges);
    }
    yPrev = v1.y;
    v1 = v2;
  }
}

void buildActiveList (int scan, Edge * active, Edge * edges[])
{
  Edge * p, * q;

  p = edges[scan]->next;
  while (p) {
    q = p->next;
    insertEdge (active, p);
    p = q;
  }
}

void fillScan (int scan, Edge * active)
{
  Edge * p1, * p2;
  int i;

  p1 = active->next;
  while (p1) {
    p2 = p1->next;
```

```
    for (i=p1->xIntersect; i<p2->xIntersect; i++)
      setPixel ((int) i, scan);
    p1 = p2->next;
  }
}

void deleteAfter (Edge * q)
{
  Edge * p = q->next;

  q->next = p->next;
  free (p);
}

/* Delete completed edges. Update 'xIntersect' field for others */
void updateActiveList (int scan, Edge * active)
{
  Edge * q = active, * p = active->next;

  while (p)
    if (scan >= p->yUpper) {
      p = p->next;
      deleteAfter (q);
    }
    else {
      p->xIntersect = p->xIntersect + p->dxPerScan;
      q = p;
      p = p->next;
    }
}

void resortActiveList (Edge * active)
{
  Edge * q, * p = active->next;

  active->next = NULL;
  while (p) {
    q = p->next;
    insertEdge (active, p);
    p = q;
  }
}

void scanFill (int cnt, dcPt * pts)
{
  Edge * edges[WINDOW_HEIGHT], * active;
  int i, scan;

  for (i=0; i<WINDOW_HEIGHT; i++) {
    edges[i] = (Edge *) malloc (sizeof (Edge));
    edges[i]->next = NULL;
  }
  buildEdgeList (cnt, pts, edges);
  active = (Edge *) malloc (sizeof (Edge));
  active->next = NULL;

  for (scan=0; scan<WINDOW_HEIGHT; scan++) {
    buildActiveList (scan, active, edges);
    if (active->next) {
      fillScan (scan, active);
      updateActiveList (scan, active);
      resortActiveList (active);
    }
```

```
    }
    /* Free edge records that have been malloc'ed ... */
}
```

Inside-Outside Tests

Area-filling algorithms and other graphics processes often need to identify interior regions of objects. So far, we have discussed area filling only in terms of standard polygon shapes. In elementary geometry, a polygon is usually defined as having no self-intersections. Examples of standard polygons include triangles, rectangles, octagons, and decagons. The component edges of these objects are joined only at the vertices, and otherwise the edges have no common points in the plane. Identifying the interior regions of standard polygons is generally a straightforward process. But in most graphics applications, we can specify any sequence for the vertices of a fill area, including sequences that produce intersecting edges, as in Fig. 3-40. 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. Graphics packages normally use either the odd-even rule or the nonzero winding number rule to identify interior regions of an object.

We apply the **odd-even rule**, also called the **odd parity rule** or the **even-odd rule**, by conceptually drawing a line from any position **P** to a distant point outside the coordinate extents of the object and counting the number of edge crossings along the line. If the number of polygon edges crossed by this line is odd, then **P** is an *interior* point. Otherwise, **P** is an *exterior* point. To obtain an accurate edge count, we must be sure that the line path we choose does not intersect any polygon vertices. Figure 3-40(a) shows the interior and exterior regions obtained from the odd-even rule for a self-intersecting set of edges. The scan-line polygon fill algorithm discussed in the previous section is an example of area filling using the odd-even rule.

Another method for defining interior regions is the **nonzero winding number rule**, which counts the number of times the polygon edges wind around a particular point in the counterclockwise direction. This count is called the **winding number**, and the interior points of a two-dimensional object are defined to be

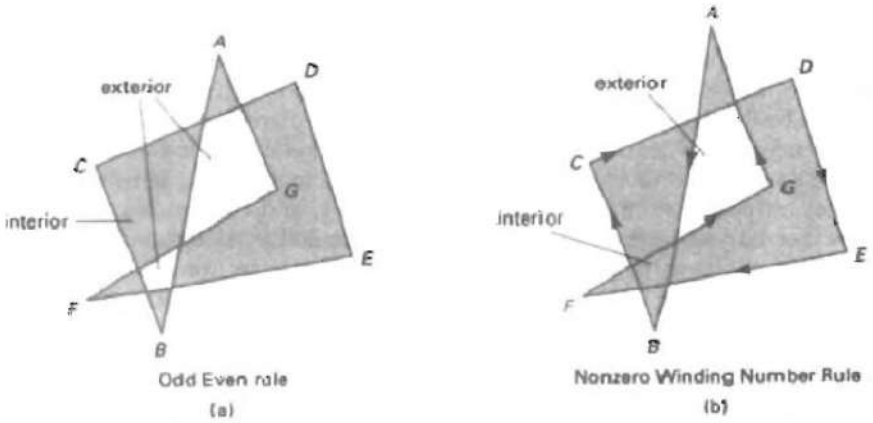


Figure 3-40
Identifying interior and exterior regions for a self-intersecting polygon.

those that have a nonzero value for the winding number. We apply the nonzero winding number rule to polygons 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 vertices. As we move along the line from position **P** to the distant point, we count the number of edges that cross the line in each direction. We add 1 to the winding number every time we intersect a polygon edge that crosses the line from right to left, and we subtract 1 every time we intersect an edge that crosses from left to right. The final value of the winding number, after all edge crossings have been counted, determines the relative position of **P**. If the winding number is nonzero, **P** is defined to be an interior point. Otherwise, **P** is taken to be an exterior point. Figure 3-40(b) shows the interior and exterior regions defined by the nonzero winding number rule for a self-intersecting set of edges. For standard polygons and other simple shapes, the nonzero winding number rule and the odd-even rule give the same results. But for more complicated shapes, the two methods may yield different interior and exterior regions, as in the example of Fig. 3-40.

One way to determine directional edge crossings is to take the vector cross product of a vector **u** along the line from **P** to a distant point with the edge vector **E** for each edge that crosses the line. If the z component of the cross product $\mathbf{u} \times \mathbf{E}$ for a particular edge is positive, that edge crosses from right to left and we add 1 to the winding number. Otherwise, the edge crosses from left to right and we subtract 1 from the winding number. An edge vector is calculated by subtracting the starting vertex position for that edge from the ending vertex position. For example, the edge vector for the first edge in the example of Fig. 3-40 is

$$\mathbf{E}_{AB} = \mathbf{V}_B - \mathbf{V}_A$$

where $\mathbf{V}_A$ and $\mathbf{V}_B$ represent the point vectors for vertices A and B. A somewhat simpler way to compute directional edge crossings is to use vector dot products instead of cross products. To do this, we set up a vector that is perpendicular to **u** and that points from right to left as we look along the line from P in the direction of **u**. If the components of **u** are (u_x, u_y), then this perpendicular to **u** has components $(-u_y, u_x)$ (Appendix A). Now, if the dot product of the perpendicular and an edge vector is positive, that edge crosses the line from right to left and we add 1 to the winding number. Otherwise, the edge crosses the line from left to right, and we subtract 1 from the winding number.

Some graphics packages use the nonzero winding number rule to implement area filling, since it is more versatile than the odd-even rule. In general, objects can be defined with multiple, unconnected sets of vertices or disjoint sets of closed curves, and the direction specified for each set can be used to define the interior regions of objects. 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 determining intersections with the curve path, instead of finding edge intersections. Similarly, with the nonzero winding number rule, we need to calculate tangent vectors to the curves at the crossover intersection points with the line from position **P**.

Scan-Line Fill of Curved Boundary Areas

In general, scan-line fill of regions with curved boundaries requires more work than polygon filling, since intersection calculations now involve nonlinear boundaries. For simple curves such as circles or ellipses, performing a scan-line fill is a straightforward process. We only need to calculate the two scan-line inter-

sections on opposite sides of the curve. This is the same as generating pixel positions along the curve boundary, and we can do that with the midpoint method. Then we simply fill in the horizontal pixel spans between the boundary points on opposite sides of the curve. Symmetries between quadrants (and between octants for circles) are used to reduce the boundary calculations.

Similar methods can be used to generate a fill area for a curve section. An elliptical arc, for example, can be filled as in Fig. 3-41. The interior region is bounded by the ellipse section and a straight-line segment that closes the curve by joining the beginning and ending positions of the arc. Symmetries and incremental calculations are exploited whenever possible to reduce computations.

Figure 3-41
Interior fill of an elliptical arc.

Boundary-Fill Algorithm

Another approach to area filling is to start at a point inside a region and paint the interior outward toward the boundary. If the boundary is specified in a single color, the fill algorithm proceeds outward pixel by pixel until the boundary color is encountered. This method, called the **boundary-fill algorithm**, is particularly useful in interactive painting packages, where interior points are easily selected. Using a graphics tablet or other interactive device, an artist or designer can sketch a figure outline, select a fill color or pattern from a color menu, and pick an interior point. The system then paints the figure interior. To display a solid color region (with no border), the designer can choose the fill color to be the same as the boundary color.

A boundary-fill procedure accepts as input the coordinates of an interior point (x, y), a fill color, and a boundary color. Starting from (x, y), the procedure tests neighboring positions to determine whether they are of the boundary color. If not, they are painted with the fill color, and their neighbors are tested. This process continues until all pixels up to the boundary color for the area have been tested. Both inner and outer boundaries can be set up to specify an area, and some examples of defining regions for boundary fill are shown in Fig. 3-42.

Figure 3-43 shows two methods for proceeding to neighboring pixels from the current test position. In Fig. 3-43(a), four neighboring points are tested. These are the pixel positions that are right, left, above, and below the current pixel. Areas filled by this method are called **4-connected**. The second method, shown in Fig. 3-43(b), is used to fill more complex figures. Here the set of neighboring positions to be tested includes the four diagonal pixels. Fill methods using this approach are called **8-connected**. An 8-connected boundary-fill algorithm would correctly fill the interior of the area defined in Fig. 3-44, but a 4-connected boundary-fill algorithm produces the partial fill shown.

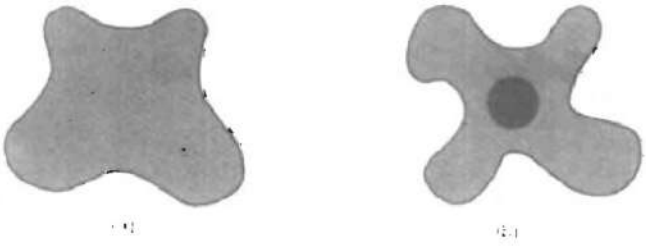


Figure 3-42
Example color boundaries for a boundary-fill procedure.

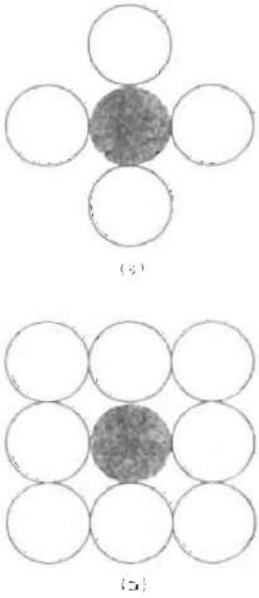


Figure 3-43
Fill methods applied to a 4-connected area (a) and to an 8-connected area (b). Open circles represent pixels to be tested from the current test position, shown as a solid color

The following procedure illustrates a recursive method for filling a 4-connected area with an intensity specified in parameter `fill` up to a boundary color specified with parameter `boundary`. We can extend this procedure to fill an 8-connected region by including four additional statements to test diagonal positions, such as $(x + 1, y + 1)$.

```
void boundaryFill4 (int x, int y, int fill, int boundary)
{
  int current;

  current = getPixel (x, y);
  if ((current != boundary) && (current != fill)) {
    setColor (fill);
    setPixel (x, y);
    boundaryFill4 (x+1, y, fill, boundary);
    boundaryFill4 (x-1, y, fill, boundary);
    boundaryFill4 (x, y+1, fill, boundary);
    boundaryFill4 (x, y-1, fill, boundary);
  }
}
```

Recursive boundary-fill algorithms may not fill regions correctly if some interior pixels are already displayed in the fill color. This occurs because the algorithm checks next pixels both for boundary color and for fill color. Encountering a pixel with the fill color can cause a recursive branch to terminate, leaving other interior pixels unfilled. To avoid this, we can first change the color of any interior pixels that are initially set to the fill color before applying the boundary-fill procedure.

Also, since this procedure requires considerable stacking of neighboring points, more efficient methods are generally employed. These methods fill horizontal pixel spans across scan lines, instead of proceeding to 4-connected or 8-connected neighboring points. Then we need only stack a beginning position for each horizontal pixel span, instead of stacking all unprocessed neighboring positions around the current position. Starting from the initial interior point with this method, we first fill in the contiguous span of pixels on this starting scan line. Then we locate and stack starting positions for spans on the adjacent scan lines, where spans are defined as the contiguous horizontal string of positions

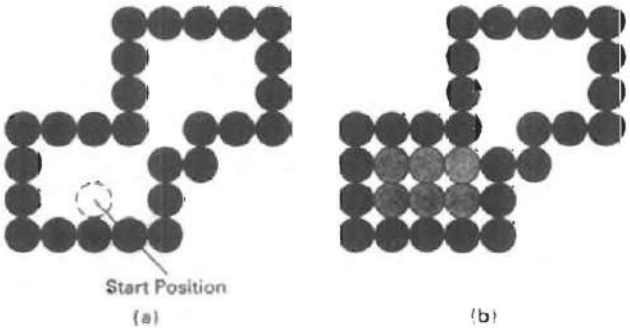


Figure 3-44
The area defined within the color boundary (a) is only partially filled in (b) using a 4-connected boundary-fill algorithm.

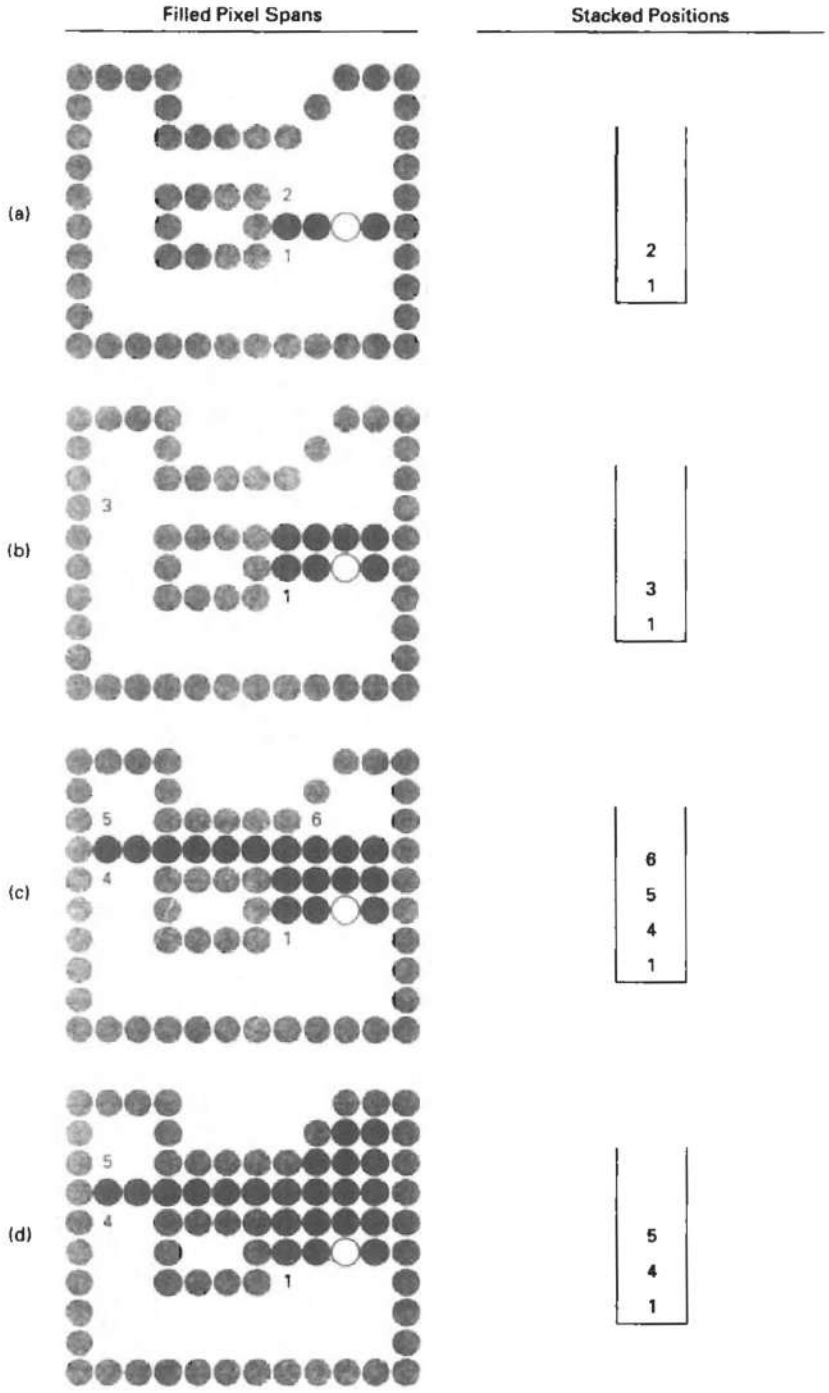


Figure 3-45
Boundary fill across pixel spans for a 4-connected area. (a) The filled initial pixel span, showing the position of the initial point (open circle) and the stacked positions for pixel spans on adjacent scan lines. (b) Filled pixel span on the first scan line above the initial scan line and the current contents of the stack. (c) Filled pixel spans on the first two scan lines above the initial scan line and the current contents of the stack. (d) Completed pixel spans for the upper-right portion of the defined region and the remaining stacked positions to be processed.

bounded by pixels displayed in the area border color. At each subsequent step, we unstack the next start position and repeat the process.

An example of how pixel spans could be filled using this approach is illustrated for the 4-connected fill region in Fig. 3-45. In this example, we first process scan lines successively from the start line to the top boundary. After all upper scan lines are processed, we fill in the pixel spans on the remaining scan lines in order down to the bottom boundary. The leftmost pixel position for each horizontal span is located and stacked, in left to right order across successive scan lines, as shown in Fig. 3-45. In (a) of this figure, the initial span has been filled, and starting positions 1 and 2 for spans on the next scan lines (below and above) are stacked. In Fig. 3-45(b), position 2 has been unstacked and processed to produce the filled span shown, and the starting pixel (position 3) for the single span on the next scan line has been stacked. After position 3 is processed, the filled spans and stacked positions are as shown in Fig. 3-45(c). And Fig. 3-45(d) shows the filled pixels after processing all spans in the upper right of the specified area. Position 5 is next processed, and spans are filled in the upper left of the region; then position 4 is picked up to continue the processing for the lower scan lines.

Figure 3-46
An area defined within multiple color boundaries.

Flood-Fill Algorithm

Sometimes we want to fill in (or recolor) an area that is not defined within a single color boundary. Figure 3-46 shows an area bordered by several different color regions. We can paint such areas by replacing a specified interior color instead of searching for a boundary color value. This approach is called a **flood-fill algorithm**. We start from a specified interior point (x, y) and reassign all pixel values that are currently set to a given interior color with the desired fill color. If the area we want to paint has more than one interior color, we can first reassign pixel values so that all interior points have the same color. Using either a 4-connected or 8-connected approach, we then step through pixel positions until all interior points have been repainted. The following procedure flood fills a 4-connected region recursively, starting from the input position.

```
void floodFill4 (int x, int y, int fillColor, int oldColor)
{
  if (getPixel (x, y) == oldColor) {
    setColor (fillColor);
    setPixel (x, y);
    floodFill4 (x+1, y, fillColor, oldColor);
    floodFill4 (x-1, y, fillColor, oldColor);
    floodFill4 (x, y+1, fillColor, oldColor);
    floodFill4 (x, y-1, fillColor, oldColor);
  }
}
```

We can modify procedure `floodFill4` to reduce the storage requirements of the stack by filling horizontal pixel spans, as discussed for the boundary-fill algorithm. In this approach, we stack only the beginning positions for those pixel spans having the value `oldColor` . The steps in this modified flood-fill algorithm are similar to those illustrated in Fig. 3-45 for a boundary fill. Starting at the first position of each span, the pixel values are replaced until a value other than `oldColor` is encountered.

3-12 FILL-AREA FUNCTIONS

We display a filled polygon in PHIGS and GKS with the function

```
fillArea (n, wcVertices)
```

The displayed polygon area is bounded by a series of n straight line segments connecting the set of vertex positions specified in `wcVertices`. These packages do not provide fill functions for objects with curved boundaries.

Implementation of the `fillArea` function depends on the selected type of interior fill. We can display the polygon boundary surrounding a hollow interior, or we can choose a solid color or pattern fill with no border for the display of the polygon. For solid fill, the `fillArea` function is implemented with the scan-line fill algorithm to display a single color area. The various attribute options for displaying polygon fill areas in PHIGS are discussed in the next chapter.

Another polygon primitive available in PHIGS is `fillAreaSet`. This function allows a series of polygons to be displayed by specifying the list of vertices for each polygon. Also, in other graphics packages, functions are often provided for displaying a variety of commonly used fill areas besides general polygons. Some examples are `fillRectangle`, `fillCircle`, `fillCircleArc`, `fillEllipse`, and `fillEllipseArc`.

3-13 CELL ARRAY

The **cell array** is a primitive that allows users to display an arbitrary shape defined as a two-dimensional grid pattern. A predefined matrix of color values is mapped by this function onto a specified rectangular coordinate region. The PHIGS version of this function is

```
cellArray (wcPoints, n, m, colorArray
```

where `colorArray` is the n by m matrix of integer color values and `wcPoints` lists the limits of the rectangular coordinate region: (x_{min}, y_{min}) and (x_{max}, y_{max}). Figure 3-47 shows the distribution of the elements of the color matrix over the coordinate rectangle.

Each coordinate cell in Fig. 3-47 has width $(x_{max} - x_{min})/n$ and height $(y_{max} - y_{min})/m$. Pixel color values are assigned according to the relative positions of the pixel center coordinates. If the center of a pixel lies within one of the n by m coordinate cells, that pixel is assigned the color of the corresponding element in the matrix `colorArray`.

3-14 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 **type-**

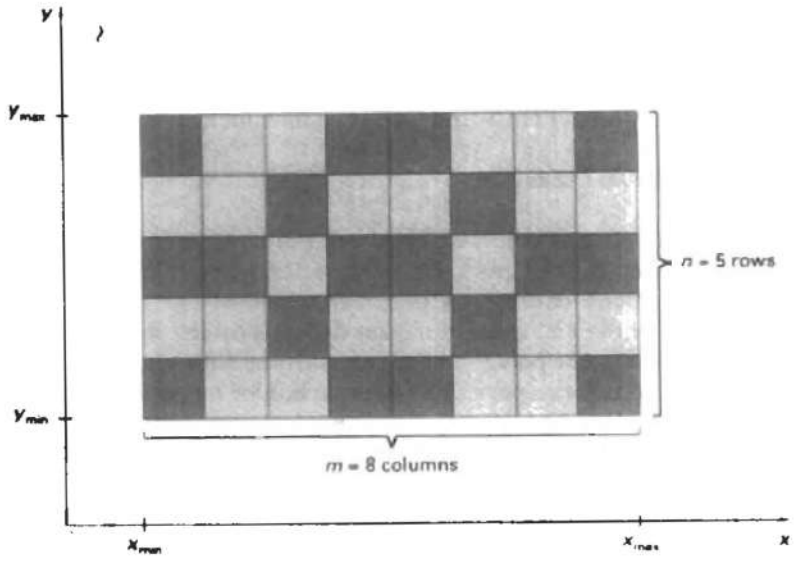


Figure 3-47
Mapping an n by m cell array into a rectangular coordinate region.

face. Today, there are hundreds 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. Now, the terms font and typeface are often used interchangeably, since 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 (Optima). 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.

Two different representations are used for storing computer fonts. A simple method for representing the character shapes in a particular typeface is to use rectangular grid patterns. The set of characters are then referred to as a **bitmap font** (or **bitmapped 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**. Figure 3-48 illustrates the two methods for character representation. When the pattern in Fig. 3-48(a) is copied to an area of the frame buffer, the 1 bits designate which pixel positions are to be displayed on the monitor. To display the character shape in Fig. 3-48(b), the interior of the character outline must be filled using the scan-line fill procedure (Section 3-11).

Bitmap fonts are the simplest to define and display: The character grid only needs to be mapped to a frame-buffer position. In general, however, bitmap fonts

require more space, because each variation (size and format) must be stored in a *font cache*. It is possible to generate different sizes and other variations, such as bold and italic, from one set, but this usually does not produce good results.

In contrast to bitmap fonts, outline fonts require less storage since 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, because they must be scan converted into the frame buffer.

A character string is displayed in PHIGS with the following function:

```
text (wcPoint, string)
```

Parameter `string` is assigned a character sequence, which is then displayed at coordinate position `wcPoint` = (x, y). For example, the statement

```
text (wcPoint, ''Population Distribution'')
```

along with the coordinate specification for `wcPoint`, could be used as a label on a distribution graph.

Just how the string is positioned relative to coordinates (x, y) is a user option. The default is that (x, y) sets the coordinate location for the lower left corner of the first character of the horizontal string to be displayed. Other string orientations, such as vertical, horizontal, or slanting, are set as attribute options and will be discussed in the next chapter.

Another convenient character function in PHIGS is one that places a designated character, called a **marker symbol**, at one or more selected positions. This function is defined with the same parameter list as in the line function:

```
polymarker (n, wcPoints)
```

A predefined character is then centered at each of the `n` coordinate positions in the list `wcPoints`. The default symbol displayed by polymarker depends on the

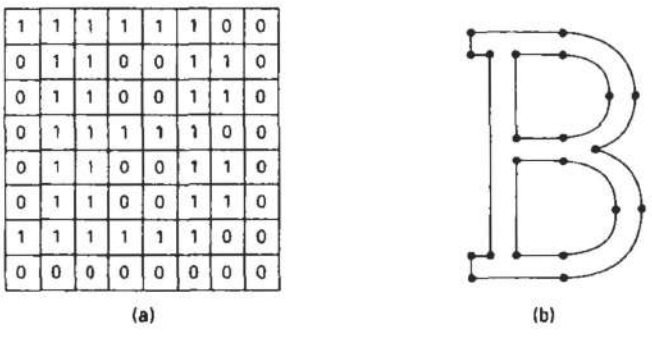
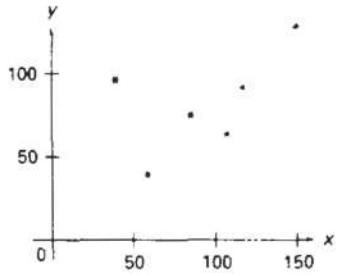


Figure 3-48
The letter B represented in (a) with an 8 by 8 bilevel bitmap pattern and in (b) with an outline shape defined with straight-line and curve segments.



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

Figure 3-49
Sequence of data values plotted with the `polymarker` function.

particular implementation, but we assume for now that an asterisk is to be used. Figure 3-49 illustrates plotting of a data set with the statement

```
polymarker (6, wcPoints)
```

SUMMARY

The output primitives discussed in this chapter provide the basic tools for constructing pictures with straight lines, curves, filled areas, cell-array patterns, and text. Examples of pictures generated with these primitives are given in Figs. 3-50 and 3-51.

Three methods that can be used to plot pixel positions along a straight-line path are the DDA algorithm, Bresenham's algorithm, and the midpoint method. For straight lines, Bresenham's algorithm and the midpoint method are identical and are the most efficient Frame-buffer access in these methods can also be performed efficiently by incrementally calculating memory addresses. Any of the line-generating algorithms can be adapted to a parallel implementation by partitioning line segments.

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 design applications. Parallel implementation of curve generation can be accomplished by partitioning the curve paths.

To account for the fact that displayed lines and curves have finite widths, we must 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.

Filled area primitives in many graphics packages refer to filled polygons. A common method for providing polygon fill on raster systems is the scan-line fill algorithm, which determines interior pixel spans across scan lines that intersect the polygon. The scan-line algorithm can also be used to fill the interior of objects with curved boundaries. Two other methods for filling the interior regions of objects are the boundary-fill algorithm and the flood-fill algorithm. These two fill procedures paint the interior, one pixel at a time, outward from a specified interior point.

The scan-line fill algorithm is an example of filling object interiors using the odd-even rule to locate the interior regions. Other methods for defining object interiors are also useful, particularly with unusual, 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.

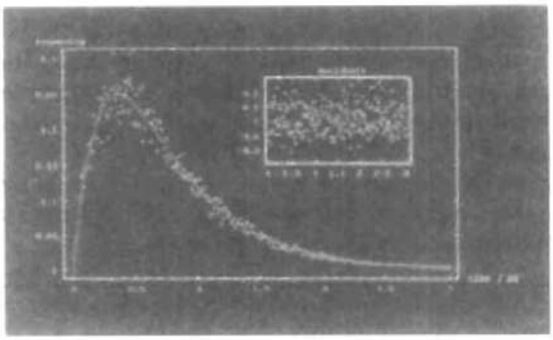


Figure 3-50
A data plot generated with straight line segments, a curve, circles (or markers), and text. *(Courtesy of Wolfram Research, Inc., The Maker of Mathematica.)*

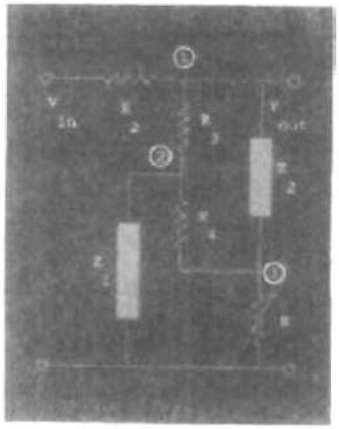


Figure 3-51
An electrical diagram drawn with straight line sections, circles, filled rectangles, and text. *(Courtesy of Wolfram Research, Inc., The Maker of Mathematica.)*

Additional primitives available in graphics packages include cell arrays, character strings, and marker symbols. Cell arrays are used to define and store color patterns. Character strings are used to provide picture and graph labeling. And marker symbols are useful for plotting the position of data points.

Table 3-1 lists implementations for some of the output primitives discussed in this chapter.

TABLE 3-1
OUTPUT PRIMITIVE IMPLEMENTATIONS

`typedef struct { float x, y; } wcPt2;`
Defines a location in 2-dimensional world-coordinates.

`pPolyline (int n, wcPt2 * pts)`
Draw a connected sequence of `n-1` line segments, specified in `pts`.

`pCircle (wcPt2 center, float r)`
Draw a circle of radius `r` at `center`.

`pFillarea (int n, wcPt2 * pts)`
Draw a filled polygon with `n` vertices, specified in `pts`.

`pCellArray (wcPt2 * pts, int n, int m, int colors)`
Map an `n` by `m` array of `colors` onto a rectangular area defined by `pts`.

`pText (wcPt2 position, char * txt)`
Draw the character string `txt` at `position`.

`pPolymarker (int n, wcPt2 * pts)`
Draw a collection of `n` marker symbols at `pts`.

Applications

Here, we present a few example programs illustrating applications of output primitives. Functions listed in Table 3-1 are defined in the header file `graphics.h`, along with the routines `openGraphics`, `closeGraphics`, `setColor`, and `setBackground`.

The first program produces a line graph for monthly data over a period of one year. Output of this procedure is drawn in Fig. 3-52. This data set is also used by the second program to produce the bar graph in Fig. 3-53 .

```
#include <stdio.h>
#include "graphics.h"

#define WINDOW_WIDTH 600
#define WINDOW_HEIGHT 500
/* Amount of space to leave on each side of the chart */
#define MARGIN_WIDTH 0.05 * WINDOW_WIDTH
#define N_DATA 12

typedef enum
{ Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec } Months;

char * monthNames[N_DATA] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                              "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };

int readData (char * inFile, float * data)
{
  int fileError = FALSE;
  FILE * fp;
  Months month;

  if ((fp = fopen (inFile, "r")) == NULL)
    fileError = TRUE;
  else {
    for (month = Jan; month <= Dec; month++)
      fscanf (fp, "%f", &data[month]);
    fclose (fp);
  }
  return (fileError);
}

void lineChart (float * data)
{
  wcPt2 dataPos[N_DATA], labelPos;
  Months m;
  float mWidth = (WINDOW_WIDTH - 2 * MARGIN_WIDTH) / N_DATA;
  int chartBottom = 0.1 * WINDOW_HEIGHT;
  int offset = 0.05 * WINDOW_HEIGHT; /* Space between data and labels */
  int labelLength = 24;     /* Assuming fixed-width 8-pixel characters */

  labelPos.y = chartBottom;
  for (m = Jan; m <= Dec; m++) {
    /* Calculate x and y positions for data markers */
    dataPos[m].x = MARGIN_WIDTH + m * mWidth + 0.5 * mWidth;
    dataPos[m].y = chartBottom + offset + data[m];
    /* Shift the label to the left by one-half its length */
    labelPos.x = dataPos[m].x - 0.5 * labelLength;
    pText (labelPos, monthNames[m]);
  }
  pPolyline (N_DATA, dataPos);
  pPolymarker (N_DATA, dataPos);
```

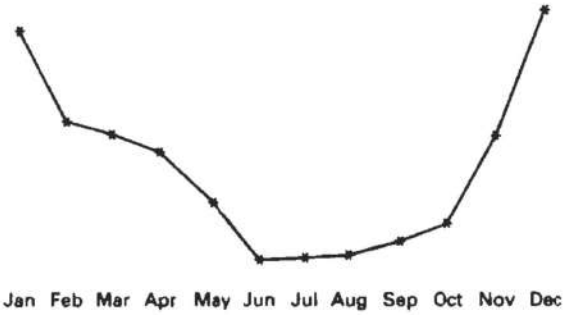


Figure 3-52
A line plot of data points output by the `lineChart` procedure.

```
}

void main (int argc, char ** argv)
{
  float data[N_DATA];
  int dataError = FALSE;
  long windowID;

  if (argc < 2) {
    fprintf (stderr, "Usage: %s dataFileName\n", argv[0]);
    exit ();
  }
  dataError = readData (argv[1], data);
  if (dataError) {
    fprintf (stderr, "%s error.  Can't read file %s\n", argv[1]);
    exit ();
  }
  windowID = openGraphics (*argv, WINDOW_WIDTH, WINDOW_HEIGHT);
  setBackground (WHITE);
  setColor (BLACK);
  lineChart (data);
  sleep (10);
  closeGraphics (windowID);
}
```

```
void barChart (float * data)
{
  wcPt2 dataPos[4], labelPos;
  Months m;
  float x, mWidth = (WINDOW_WIDTH - 2 * MARGIN_WIDTH) / N_DATA;
  int chartBottom = 0.1 * WINDOW_HEIGHT;
  int offset = 0.05 * WINDOW_HEIGHT; /* Space between data and labels */
  int labelLength = 24;     /* Assuming fixed-width 8-pixel characters */

  labelPos.y = chartBottom;
  for (m = Jan; m <= Dec; m++) {
    /* Find the center of this month's bar */
    x = MARGIN_WIDTH + m * mWidth + 0.5 * mWidth;

    /* Shift the label to the left by one-half its assumed length */
    labelPos.x = x - 0.5 * labelLength;
```

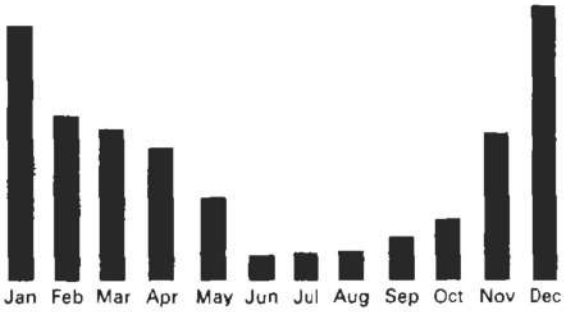


Figure 3-53
A bar-chart plot output by the `barChart` procedure.

```
    pText (labelPos, monthNames[m]);

    /* Get the coordinates for this month's bar */
    dataPos[0].x = dataPos[3].x = x - 0.5 * labelLength;
    dataPos[1].x = dataPos[2].x = x + 0.5 * labelLength;
    dataPos[0].y = dataPos[1].y = chartBottom + offset;
    dataPos[2].y = dataPos[3].y = chartBottom + offset + data[m];
    pFillArea (4, dataPos);
  }
}
```

Pie charts are used to show the percentage contribution of individual parts to the whole. The next procedure constructs a pie chart, with the number and relative size of the slices determined by input. A sample output from this procedure appears in Fig. 3-54.

```
#define TWO_PI 6.28

void pieChart (float * data)
{
  wcPt2 pts[2], center;
  float radius = WINDOW_HEIGHT / 4.0;
  float newSlice, total = 0.0, lastSlice = 0.0;
  Months month;

  center.x = WINDOW_WIDTH / 2;
  center.y = WINDOW_HEIGHT / 2;
  pCircle (center, radius);
  for (month = Jan; month <= Dec; month++)
    total += data[month];
  pts[0].x = center.x; pts[0].y = center.y;
  for (month = Jan; month <= Dec; month++) {
    newSlice = TWO_PI * data[month] / total + lastSlice;
    pts[1].x = center.x + radius * cosf (newSlice);
    pts[1].y = center.y + radius * sinf (newSlice);
    pPolyline (2, pts);
    lastSlice = newSlice;
  }
}
```

Some variations on the circle equations are output by this next procedure. The shapes shown in Fig. 3-55 are generated by varying the radius r of a circle. Depending on how we vary r, we can produce a spiral, cardioid, limaçon, or other similar figure.

```
#include <stdio.h>
#include <math.h>
#include "graphics.h"

#define TWO_PI 6.28

/* Limacon equation is r = a * ccs(theta) + b.  Cardioid is the same,
   with a == b, so r = a * (1 + cos(theta)).
*/
typedef enum { spiral, cardioid, threeLeaf, fourLeaf, limacon } Fig;

void drawCurlyFig (Fig figure, wcPt2 pos, int * p)
{
  float r, theta = 0.0, dtheta = 1.0 / (float) p[0];
  int nPoints = (int) ceilf (TWO_PI * p[0]) + 1;
  wcPt2 * pt;

  if ((pt = (wcPt2 *) malloc (nPoints * sizeof (wcPt2))) == NULL) {
    fprintf (stderr, "Couldn't allocate points\n");
    return;
  }

  /* Set first point for figure */
  pt[0].y = pos.y;
  switch (figure) {
  case spiral:    pt[0].x = pos.x;                break;
  case limacon:   pt[0].x = pos.x + p[0] + p[1]; break;
  case cardioid:  pt[0].x = pos.x + p[0] * 2;    break;
  case threeLeaf: pt[0].x = pos.x + p[0];        break;
  case fourLeaf:  pt[0].x = pos.x + p[0];        break;
  }
  nPoints = 1;
  while (theta < TWO_PI) {
    switch (figure) {
    case spiral:    r = p[0] * theta;                break;
    case limacon:   r = p[0] * cosf (theta) + p[1]; break;
    case cardioid:  r = p[0] * (1 + cosf (theta));  break;
    case threeLeaf: r = p[0] * cosf (3 * theta);    break;
    case fourLeaf:  r = p[0] * cosf (2 * theta);    break;
    }
    pt[nPoints].x = pos.x + r * cosf (theta);
    pt[nPoints].y = pos.y + r * sinf (theta);
    nPoints++;
    theta += dtheta;
  }

  pPolyline (nPoints, pt);
  free (pt);
}

void main (int argc, char ** argv)
{
```

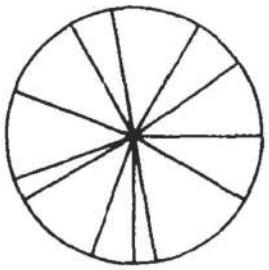


Figure 3-54
Output generated from the `pieChart` procedure.

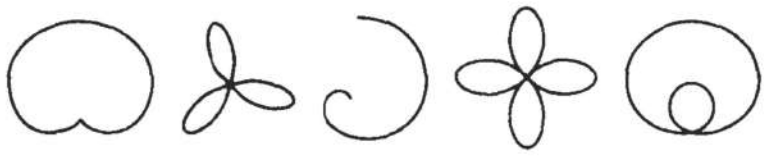


Figure 3-55
Curved figures produced with the `drawShape` procedure.

```
    long windowID = openGraphics (*argv, 400, 100);
    Fig f;
    /* Center positions for each figure */
    wcPt2 center[] = { 50, 50, 100, 50, 175, 50, 250, 50, 300, 50 };

    /* Parameters to define each figure.  First four need one parameter.
       Fifth figure (limacon) needs two. */
    int p[5][2] = { 5, -1, 20, -1, 30, -1, 30, -1, 40, 10 };

    setBackground (WHITE);
    setColor (BLACK);
    for (f=spiral; f<=limacon; f++)
      drawCurlyFig (f, center[f], p[f]);
    sleep (10);
    closeGraphics (windowID);
}
```

REFERENCES

Information on Bresenham's algorithms can be found in Bresenham (1965, 1977). For midpoint methods, see Kappel (1985). Parallel methods for generating lines and circles are discussed in Pang (1990) and in Wright (1990).

Additional programming examples and information on PHIGS primitives can be found in Howard, et al. (1991), Hopgood and Duce (1991), Gaskins (1992), and Blake (1993). For information on GKS output primitive functions, see Hopgood et al. (1983) and Enderle, Kansy, and Pfaff (1984).

EXERCISES

3-1. Implement the `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. Implement the `polyline` function using this algorithm as a routine that displays the set of straight lines connecting the n input points. For $n = 1$, the routine displays a single point.

3-3. Devise a consistent scheme for implementing the `polyline` function, for any set of input line endpoints, using a modified Bresenham line algorithm so that geometric magnitudes are maintained (Section 3-10).

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 of 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. Implement the `setPixel` routine in Bresenham's line algorithm using iterative techniques for calculating frame-buffer addresses (Section 3-3).

3-11. Revise the midpoint circle algorithm to display so that geometric magnitudes are maintained (Section 3-10).

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.

3-16. 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 $A/10$.

3-17. Using the midpoint method, and taking symmetry into account, develop an efficient algorithm for scan conversion of the following curve over the interval $-10 \le x \le 10$:

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

3-18. Use the midpoint method and symmetry considerations to scan convert the parabola

$$y = 100 - x^2$$

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

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

$$x = y^2$$

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

3-20. Set up a midpoint algorithm, taking symmetry considerations into account to scan convert any paraboia of the form

$$y = ax^2 - b$$

with input values for parameters a, b, and the range of x.

3-21. Write a program to scan convert the interior of a specified ellipse into a solid color.

3-22. 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 of edge crossings

3-23. 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 of edge crossings.

3-24. Write a procedure for filling the interior of any specified set of "polygon" vertices using the nonzero winding number rule to identify interior regions.

3-25. Modify the boundary-fill algorithm for a 4-connected region to avoid excessive stacking by incorporating scan-line methods.

3-26. Write a boundary-fill procedure to fill an 8-connected region.

3-27. Explain how an ellipse displayed with the midpoint method could be properly filled with a boundary-fill algorithm.

3-28. Develop and implement a flood-fill algorithm to fill the interior of any specified area.

3-29. Write a routine to implement the `text` function.

3-30. Write a routine to implement the `polymarker` function.

3-31. Write a program to display a bar graph using the `polyline` function. 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 screen area.

3-32. Write a program to display a bar graph in any selected screen area. Use the `polyline` function to draw the bars.

3-33. 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.)

3-34. 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.

CHAPTER

4 Attributes of Output Primitives

In general, any parameter that affects the way a primitive is to be displayed is referred to as an **attribute parameter**. Some attribute parameters, such as color and size, determine the fundamental characteristics of a primitive. Others specify how the primitive is to be displayed under special conditions. Examples of attributes in this class include depth information for three-dimensional viewing and visibility or detectability options for interactive object-selection programs. These special-condition attributes will be considered in later chapters. Here, we consider only those attributes that control the basic display properties of primitives, without regard for special situations. For example, lines can be dotted or dashed, fat or thin, and blue or orange. Areas might be filled with one color or with a multicolor pattern. Text can appear reading from left to right, slanted diagonally across the screen, or in vertical columns. Individual characters can be displayed in different fonts, colors, and sizes. And we can apply intensity variations at the edges of objects to smooth out the raster stairstep effect.

One way to incorporate attribute options into a graphics package is to extend the parameter list associated with each output primitive function to include the appropriate attributes. A line-drawing function, for example, could contain parameters to set color, width, and other properties, in addition to endpoint coordinates. Another approach is to maintain a system list of current attribute values. Separate functions are then included in the graphics package for setting the current values in the attribute list. To generate an output primitive, the system checks the relevant attributes and invokes the display routine for that primitive using the current attribute settings. Some packages provide users with a combination of attribute functions and attribute parameters in the output primitive commands. With the GKS and PHIGS standards, attribute settings are accomplished with separate functions that update a system attribute list.

4-1 LINE ATTRIBUTES

Basic attributes of a straight line segment are its type, its width, and its color. In some graphics packages, lines can also be displayed using selected pen or brush options. In the following sections, we consider how line-drawing routines can be modified to accommodate various attribute specifications.

Line Type

Possible selections for the line-type attribute include solid lines, dashed lines, and dotted lines. We modify a line-drawing algorithm to generate such lines by setting the length and spacing of displayed solid sections along the line path. A dashed line could be displayed by generating an interdash spacing that is equal to the length of the solid sections. Both the length of the dashes and the interdash spacing are often specified as user options. A dotted line can be displayed by

generating very short dashes with the spacing equal to or greater than the dash size. Similar methods are used to produce other line-type variations.

To set line type attributes in a PHIGS application program, a user invokes the function

```
setLinetype (lt)
```

where parameter `lt` is assigned a positive integer value of 1, 2, 3, or 4 to generate lines that are, respectively, solid, dashed, dotted, or dash-dotted. Other values for the line-type parameter `lt` could be used to display variations in the dot-dash patterns. Once the line-type parameter has been set in a PHIGS application program, all subsequent line-drawing commands produce lines with this line type. The following program segment illustrates use of the `linetype` command to display the data plots in Fig. 4-1.

```
#include <stdio.h>
#include "graphics.h"

#define MARGIN_WIDTH 0.05 * WINDOW_WIDTH

int readData (char * inFile, float * data)
{
  int fileError = FALSE;
  FILE * fp;
  int month;

  if ((fp = fopen (inFile, "r")) == NULL)
    fileError = TRUE;
  else {
    for (month=0; month<12; month++)
      fscanf (fp, "%f", &data[month]);
    fclose (fp);
  }
  return (fileError);
}

void chartData (float * data, pLineType lineType)
{
  wcPt2 pts[12];
  float monthWidth = (WINDOW_WIDTH - 2 * MARGIN_WIDTH) / 12;
  int i;

  for (i=0; i<12; i++) {
    pts[i].x = MARGIN_WIDTH + i * monthWidth + 0.5 * monthWidth;
    pts[i].y = data[i];
  }
  pSetLineType (lineType);
  pPolyline (12, pts);
}

int main (int argc, char ** argv)
{
  long windowID = openGraphics (*argv, WINDOW_WIDTH, WINDOW_HEIGHT);
  float data[12];

  setBackground (WHITE);
  setColor (BLUE);
  readData ("../data/data1960", data);
  chartData (data, SOLID);
  readData ("../data/data1970", data);
  chartData (data, DASHED);
  readData ("../data/data1980", data);
  chartData (data, DOTTED);
  sleep (10);
  closeGraphics (windowID);
}
```

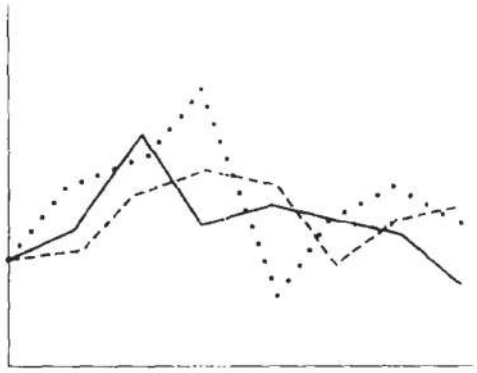


Figure 4-1
Plotting three data sets with three different line types, as output by the `chartData` procedure.

Raster line algorithms display line-type attributes by plotting pixel spans. For the various dashed, dotted, and dot-dashed patterns, the line-drawing procedure outputs sections of contiguous pixels along the line path, skipping over a number of intervening pixels between the solid spans. Pixel counts for the span length and interspan spacing can be specified in a pixel **mask**, which is a string containing the digits 1 and 0 to indicate which positions to plot along the line path. The mask 1111000, for instance, could be used to display a dashed line with a dash length of four pixels and an interdash spacing of three pixels. On a bilevel system, the mask gives the bit values that should be loaded into the frame buffer along the line path to display the selected line type.

Plotting dashes with a fixed number of pixels results in unequal-length dashes for different line orientations, as illustrated in Fig. 4-2. Both dashes shown are plotted with four pixels, but the diagonal dash is longer by a factor of $\sqrt{2}$. For precision drawings, dash lengths should remain approximately constant for any line orientation. To accomplish this, we can adjust the pixel counts for the solid spans and interspan spacing according to the line slope. In Fig. 4-2, we can display approximately equal-length dashes by reducing the diagonal dash to three pixels. Another method for maintaining dash length is to treat dashes as individual line segments. Endpoint coordinates for each dash are located and passed to the line routine, which then calculates pixel positions along the dash path.

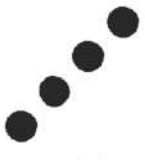


Figure 4-2
Unequal-length dashes displayed with the same number of pixels.

Line Width

Implementation of line-width options depends on the capabilities of the output device. A heavy line on a video monitor could be displayed as adjacent parallel lines, while a pen plotter might require pen changes. As with other PHIGS attributes, a line-width command is used to set the current line-width value in the attribute list. This value is then used by line-drawing algorithms to control the thickness of lines generated with subsequent output primitive commands.

We set the line-width attribute with the command:

```
setLinewidthScaleFactor (lw)
```

Line-width parameter `lw` is assigned a positive number to indicate the relative width of the line to be displayed. A value of 1 specifies a standard-width line. On a pen plotter, for instance, a user could set `lw` to a value of 0.5 to plot a line whose width is half that of the standard line. Values greater than 1 produce lines thicker than the standard.

For raster implementation, a standard-width line is generated with single pixels at each sample position, as in the Bresenham algorithm. Other-width lines are displayed as positive integer multiples of the standard line by plotting additional pixels along adjacent parallel line paths. For lines with slope magnitude less than 1, we can modify a line-drawing routine to display thick lines by plotting a vertical span of pixels at each x position along the line. The number of pixels in each span is set equal to the integer magnitude of parameter `lw`. In Fig. 4-3, we plot a double-width line by generating a parallel line above the original line path. At each x sampling position, we calculate the corresponding y coordinate and plot pixels with screen coordinates (x, y) and $(x, y+1)$. We display lines with `lw` ≥ 3 by alternately plotting pixels above and below the single-width line path.

For lines with slope magnitude greater than 1, we can plot thick lines with horizontal spans, alternately picking up pixels to the right and left of the line path. This scheme is demonstrated in Fig. 4-4, where a line width of 4 is plotted with horizontal pixel spans.

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

Another problem with implementing width options using horizontal or vertical pixel spans is that the method produces lines whose ends are horizontal or vertical regardless of the slope of the line. This effect is more noticeable with very thick lines. We can adjust the shape of the line ends to give them a better appearance by adding **line caps** (Fig. 4-5). One kind of line cap is the *butt cap* obtained by adjusting the end positions of the component parallel lines so that the thick line is displayed with square ends that are perpendicular to the line path. If the specified line has slope m, the square end of the thick line has slope $-1/m$. Another line cap is the *round cap* obtained by adding a filled semicircle to each butt cap. The circular arcs are centered on the line endpoints and have a diameter equal to the line thickness. A third type of line cap is the *projecting square cap*. Here, we simply extend the line and add butt caps that are positioned one-half of the line width beyond the specified endpoints.

Other methods for producing thick lines include displaying the line as a filled rectangle or generating the line with a selected pen or brush pattern, as discussed in the next section. To obtain a rectangle representation for the line

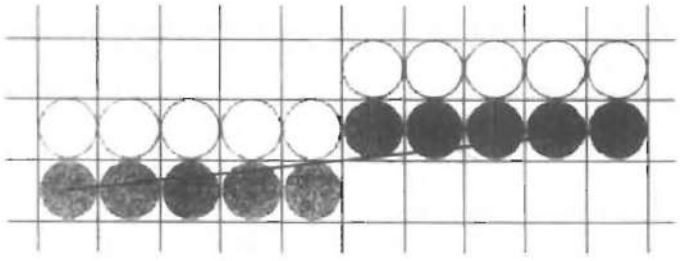


Figure 4-3
Double-wide raster line with slope $|m| < 1$ generated with vertical pixel spans.

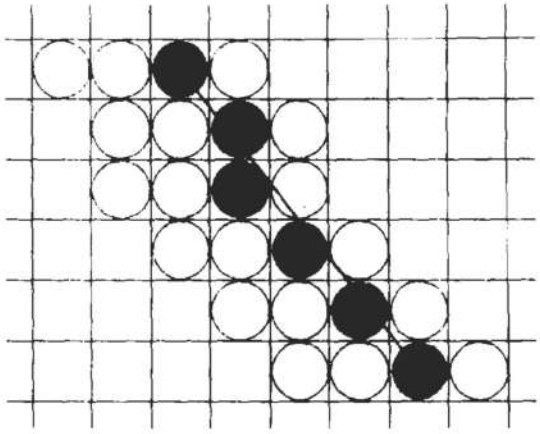


Figure 4-4
Raster line with slope $|m| > 1$ and line-width parameter $lw = 4$ plotted with horizontal pixel spans.

boundary, we calculate the position of the rectangle vertices along perpendiculars to the line path so that vertex coordinates are displaced from the line endpoints by one-half the line width. The rectangular line then appears as in Fig. 4-5(a). We could then add round caps to the filled rectangle or extend its length to display projecting square caps.

Generating thick polylines requires some additional considerations. In general, the methods we have considered for displaying a single line segment will not produce a smoothly connected series of line segments. Displaying thick lines using horizontal and vertical pixel spans, for example, leaves pixel gaps at the boundaries between lines of different slopes where there is a shift from horizontal spans to vertical spans. We can generate thick polylines that are smoothly joined at the cost of additional processing at the segment endpoints. Figure 4-6 shows three possible methods for smoothly joining two line segments. A *miter join* is accomplished by extending the outer boundaries of each of the two lines until they meet. A *round join* is produced by capping the connection between the two segments with a circular boundary whose diameter is equal to the line

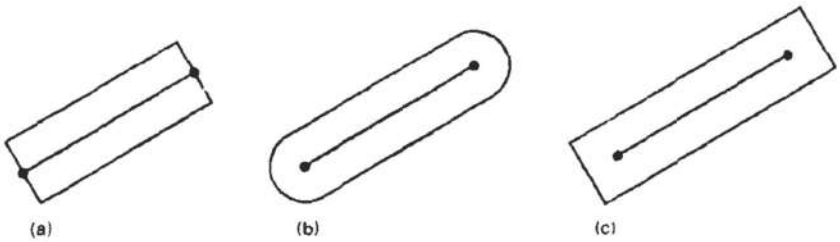


Figure 4-5
Thick lines drawn with (a) butt caps, (b) round caps, and (c) projecting square caps.

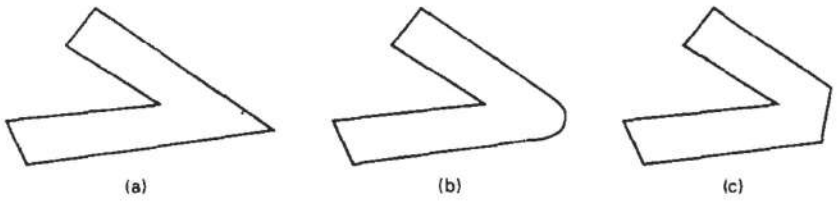


Figure 4-6
Thick line segments connected with (a) miter join, (b) round join, and (c) bevel join.

width. And a *bevel join* is generated by displaying the line segments with butt caps and filling in the triangular gap where the segments meet. If the angle between two connected line segments is very small, a miter join can generate a long spike that distorts the appearance of the polyline. A graphics package can avoid this effect by switching from a miter join to a bevel join, say, when any two consecutive segments meet at a small enough angle.

Pen and Brush Options

With some packages, lines can be displayed with pen or brush selections. Options in this category include shape, size, and pattern. Some possible pen or brush shapes are given in Fig. 4-7. These shapes can be stored in a pixel mask that identifies the array of pixel positions that are to be set along the line path. For example, a rectangular pen can be implemented with the mask shown in Fig. 4-8 by moving the center (or one corner) of the mask along the line path, as in Fig. 4-9. To avoid setting pixels more than once in the frame buffer, we can simply accumulate the horizontal spans generated at each position of the mask and keep track of the beginning and ending x positions for the spans across each scan line.

Lines generated with pen (or brush) shapes can be displayed in various widths by changing the size of the mask. For example, the rectangular pen line in Fig. 4-9 could be narrowed with a 2×2 rectangular mask or widened with a 4×4 mask. Also, lines can be displayed with selected patterns by superimposing the pattern values onto the pen or brush mask. Some examples of line patterns are shown in Fig. 4-10. An additional pattern option that can be provided in a paint package is the display of simulated brush strokes. Figure 4-11 illustrates some patterns that can be displayed by modeling different types of brush strokes.

Line Color

When a system provides color (or intensity) options, a parameter giving the current color index is included in the list of system-attribute values. A polyline routine displays a line in the current color by setting this color value in the frame buffer at pixel locations along the line path using the `setPixel` procedure. The number of color choices depends on the number of bits available per pixel in the frame buffer.

We set the line color value in PHIGS with the function

```
setPolylineColourIndex (lc)
```

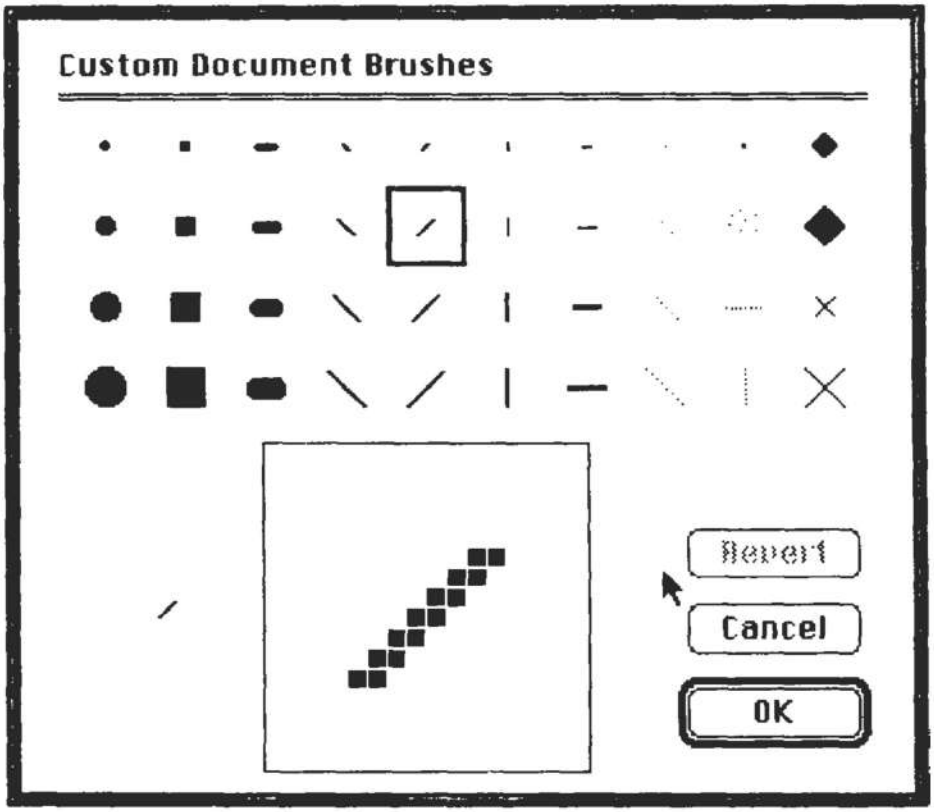


Figure 4-7
Pen and brush shapes for line display.

Nonnegative integer values, corresponding to allowed color choices, are assigned to the line color parameter `lc`. A line drawn in the background color is invisible, and a user can erase a previously displayed line by respecifying it in the background color (assuming the line does not overlap more than one background color area).

An example of the use of the various line attribute commands in an applications program is given by the following sequence of statements:

```
setLinetype (2);
setLinewidthScaleFactor (2);
setPolylineColourIndex (5);
polyline (n1, wcpoints1);

setPolylineColourIndex (6);
polyline (n2, wcpoints2);
```

This program segment would display two figures, drawn with double-wide dashed lines. The first is displayed in a color corresponding to code 5, and the second in color 6.

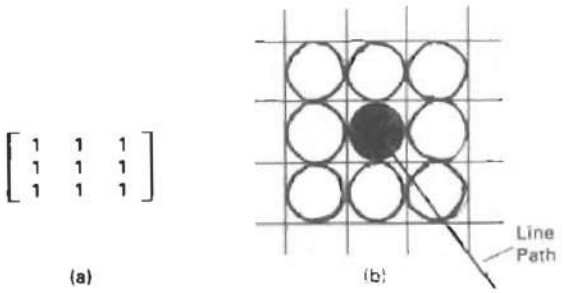


Figure 4-8
(a) A pixel mask for a rectangular pen, and (b) the associated array of pixels displayed by centering the mask over a specified pixel position.

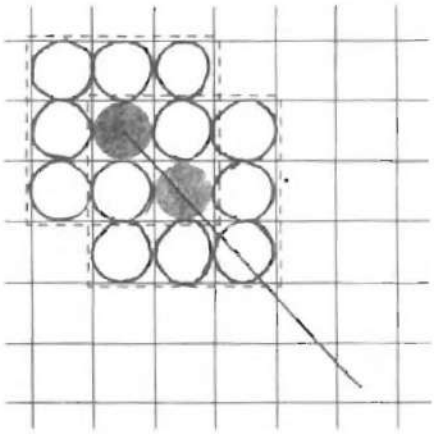


Figure 4-9
Generating a line with the pen shape of Fig. 4-8.

Figure 4-10
Curved lines drawn with a paint program using various shapes and patterns. From left to right, the brush shapes are square, round, diagonal line, dot pattern, and faded airbrush.

Figure 4-11
A daruma doll, a symbol of good fortune in Japan, drawn by computer artist Koichi Kozaki using a paintbrush system. Daruma dolls actually come without eyes. One eye is painted in when a wish is made, and the other is painted in when the wish comes true. (*Courtesy of Wacom Technology, Inc.*)

4-2 CURVE ATTRIBUTES

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

The pixel masks discussed for implementing line-type options are also used in raster curve algorithms to generate dashed and dotted patterns. For example, the mask 11100 produces the dashed circle shown in Fig. 4-12. We can generate the dashes in the various octants using circle symmetry, but we must shift the pixel positions to maintain the correct sequence of dashes and spaces as we move from one octant to the next. Also, as in line algorithms, pixel masks display dashes and interdash spaces that vary in length according to the slope of the curve. If we want to display constant-length dashes, we need to adjust the number of pixels plotted in each dash as we move around the circle circumference. Instead of applying a pixel mask with constant spans, we plot pixels along equal angular arcs to produce equal length dashes.

Raster curves of various widths can be displayed using the method of horizontal or vertical pixel spans. Where the magnitude of the curve slope is less than 1, we plot vertical spans; where the slope magnitude is greater than 1, we plot horizontal spans. Figure 4-13 demonstrates this method for displaying a circular arc of width 4 in the first quadrant. Using circle symmetry, we generate the circle path with vertical spans in the octant from $x = 0$ to $x = y$, and then reflect pixel positions about the line $y = x$ to obtain the remainder of the curve shown. Circle sections in the other quadrants are obtained by reflecting pixel positions in the

first quadrant about the coordinate axes. The thickness of curves displayed with this method is again a function of curve slope. Circles, ellipses, and other curves will appear thinnest where the slope has a magnitude of 1.

Another method for displaying thick curves is to fill in the area between two parallel curve paths, whose separation distance is equal to the desired width. We could do this using the specified curve path as one boundary and setting up the second boundary either inside or outside the original curve path. This approach, however, shifts the original curve path either inward or outward, depending on which direction we choose for the second boundary. We can maintain the original curve position by setting the two boundary curves at a distance of one-half the width on either side of the specified curve path. An example of this approach is shown in Fig. 4-14 for a circle segment with radius 16 and a specified width of 4. The boundary arcs are then set at a separation distance of 2 on either side of the radius of 16. To maintain the proper dimensions of the circular arc, as discussed in Section 3-10, we can set the radii for the concentric boundary arcs at $r = 14$ and $r = 17$. Although this method is accurate for generating thick circles, in general, it provides only an approximation to the true area of other thick

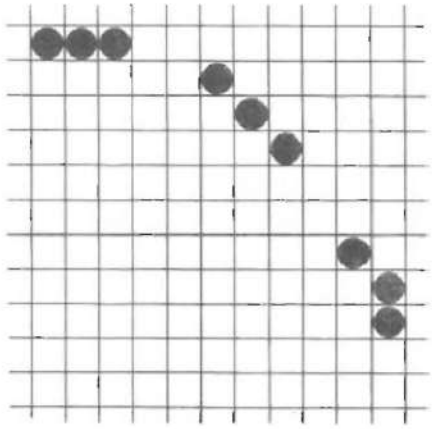


Figure 4-12
A dashed circular arc displayed with a dash span of 3 pixels and an interdash spacing of 2 pixels.

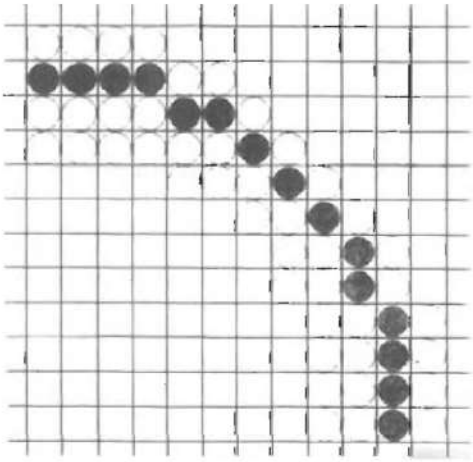


Figure 4-13
Circular arc of width 4 plotted with pixel spans.

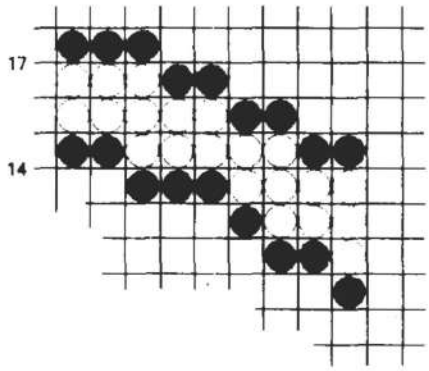


Figure 4-14
A circular arc of width 4 and radius 16 displayed by filling the region between two concentric arcs.

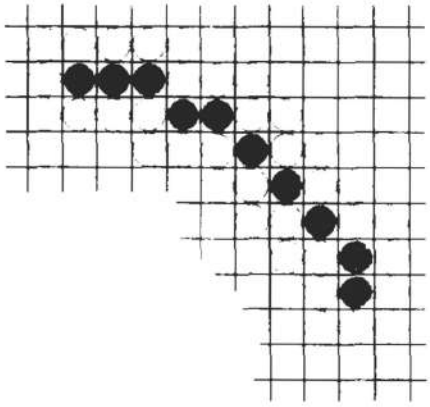


Figure 4-15
Circular arc displayed with a rectangular pen.

curves. For example, the inner and outer boundaries of a fat ellipse generated with this method do not have the same foci.

Pen (or brush) displays of curves are generated using the same techniques discussed for straight line segments. We replicate a pen shape along the line path, as illustrated in Fig. 4-15 for a circular arc in the first quadrant. Here, the center of the rectangular pen is moved to successive curve positions to produce the curve shape shown. Curves displayed with a rectangular pen in this manner will be thicker where the magnitude of the curve slope is 1. A uniform curve thickness can be displayed by rotating the rectangular pen to align it with the slope direction as we move around the curve or by using a circular pen shape. Curves drawn with pen and brush shapes can be displayed in different sizes and with superimposed patterns or simulated brush strokes.

4-3
COLOR AND GRAYSCALE LEVELS

Various color and intensity-level options can be made available to a user, depending on the capabilities and design objectives of a particular system. General-purpose raster-scan systems, for example, usually provide a wide range of colors, while random-scan monitors typically offer only a few color choices, if any. Color

options are numerically coded with values ranging from 0 through the positive integers. For CRT monitors, these color codes are then converted to intensity-level settings for the electron beams. With color plotters, the codes could control ink-jet deposits or pen selections.

In a color raster system, the number of color choices available depends on the amount of storage provided per pixel in the frame buffer. Also, color information can be stored in the frame buffer in two ways: We can store color codes directly in the frame buffer, or we can put the color codes in a separate table and use pixel values as an index into this table. With the direct storage scheme, whenever a particular color code is specified in an application program, the corresponding binary value is placed in the frame buffer for each component pixel in the output primitives to be displayed in that color. A minimum number of colors can be provided in this scheme with 3 bits of storage per pixel, as shown in Table 4-1. Each of the three bit positions is used to control the intensity level (either on or off) of the corresponding electron gun in an RGB monitor. The leftmost bit controls the red gun, the middle bit controls the green gun, and the rightmost bit controls the blue gun. Adding more bits per pixel to the frame buffer increases the number of color choices. With 6 bits per pixel, 2 bits can be used for each gun. This allows four different intensity settings for each of the three color guns, and a total of 64 color values are available for each screen pixel. With a resolution of 1024 by 1024, a full-color (24-bit per pixel) RGB system needs 3 megabytes of storage for the frame buffer. Color tables are an alternate means for providing extended color capabilities to a user without requiring large frame buffers. Lower-cost personal computer systems, in particular, often use color tables to reduce frame-buffer storage requirements.

Color Tables

Figure 4-16 illustrates a possible scheme for storing color values in a **color lookup table** (or **video lookup table**), where frame-buffer values are now used as indices into the color table. In this example, each pixel can reference any one of the 256 table positions, and each entry in the table uses 24 bits to specify an RGB color. For the color code 2081, a combination green-blue color is displayed for pixel location (x, y). Systems employing this particular lookup table would allow

TABLE 4-1
THE EIGHT COLOR CODES FOR A THREE-BIT PER PIXEL FRAME BUFFER

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

a user to select any 256 colors for simultaneous display from a palette of nearly 17 million colors. Compared to a full-color system, this scheme reduces the number of simultaneous colors that can be displayed, but it also reduces the frame-buffer storage requirements to 1 megabyte. Some graphics systems provide 9 bits per pixel in the frame buffer, permitting a user to select 512 colors that could be used in each display.

A user can set color-table entries in a PHIGS applications program with the function

```
setColourRepresentation (ws, ci, colorptr)
```

Parameter `ws` identifies the workstation output device; parameter `ci` specifies the color index, which is the color-table position number (0 to 255 for the example in Fig. 4-16); and parameter `colorptr` points to a trio of RGB color values *(r, g, b)* each specified in the range from 0 to 1. An example of possible table entries for color monitors is given in Fig. 4-17.

There are several advantages in storing color codes in a lookup table. Use of a color table can provide a "reasonable" number of simultaneous colors without requiring large frame buffers. For most applications, 256 or 512 different colors are sufficient for a single picture. Also, table entries can be changed at any time, allowing a user to be able to experiment easily with different color combinations in a design, scene, or graph without changing the attribute settings for the graphics data structure. Similarly, visualization applications can store values for some physical quantity, such as energy, in the frame buffer and use a lookup table to try out various color encodings without changing the pixel values. And in visualization and image-processing applications, color tables are a convenient means for setting color thresholds so that all pixel values above or below a specified threshold can be set to the same color. For these reasons, some systems provide both capabilities for color-code storage, so that a user can elect either to use color tables or to store color codes directly in the frame buffer.

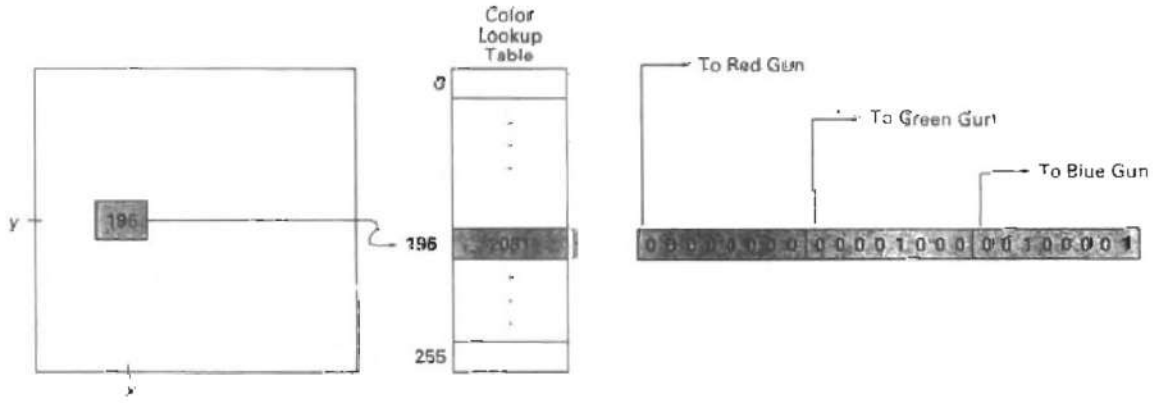


Figure 4-16
A color lookup table with 24 bits per entry accessed from a frame buffer with 8 bits per pixel. A value of 196 stored at pixel position (x, y) references the location in this table containing the value 2081. Each 8-bit segment of this entry controls the intensity level of one of the three electron guns in an RGB monitor.

WS = 1

Ci	Color
0	(0, 0, 0)
1	(0, 0, 0.2)
.	.
.	.
.	.
192	(0, 0.03, 0.13)
.	.
.	.
.	.

WS = 2

Ci	Color
0	(1, 1, 1)
1	(0.9, 1, 1)
2	(0.8, 1, 1)
.	
.	
.	

Figure 4-17
Workstation color tables.

Grayscale

With monitors that have no color capability, color functions can be used in an application program to set the shades of gray, or **grayscale,** for displayed primitives. Numeric values over the range from 0 to 1 can be used to specify grayscale levels, which are then converted to appropriate binary codes for storage in the raster. This allows the intensity settings to be easily adapted to systems with differing grayscale capabilities.

Table 4-2 lists the specifications for intensity codes for a four-level grayscale system. In this example, any intensity input value near 0.33 would be stored as the binary value 01 in the frame buffer, and pixels with this value would be displayed as dark gray. If additional bits per pixel are available in the frame buffer, the value of 0.33 would be mapped to the nearest level. With 3 bits per pixel, we can accommodate 8 gray levels; while 8 bits per pixel would give us 256 shades of gray. An alternative scheme for storing the intensity information is to convert each intensity code directly to the voltage value that produces this grayscale level on the output device in use.

When multiple output devices are available at an installation, the same color-table interface may be used for all monitors. In this case, a color table for a monochrome monitor can be set up using a range of RGB values as in Fig. 4-17, with the display intensity corresponding to a given color index `ci` calculated as

$$\text{intensity} = 0.5[\min(r, g, b) + \max(r, g, b)]$$

TABLE 4-2
INTENSITY CODES FOR A FOUR-LEVEL GRAYSCALE SYSTEM

Intensity Codes	*Stored Intensity Values In The Frame Buffer (Binary Code)*		*Displayed Grayscale*
0.0	0	(00)	Black
0.33	1	(01)	Dark gray
0.67	2	(10)	Light gray
1.0	3	(11)	White

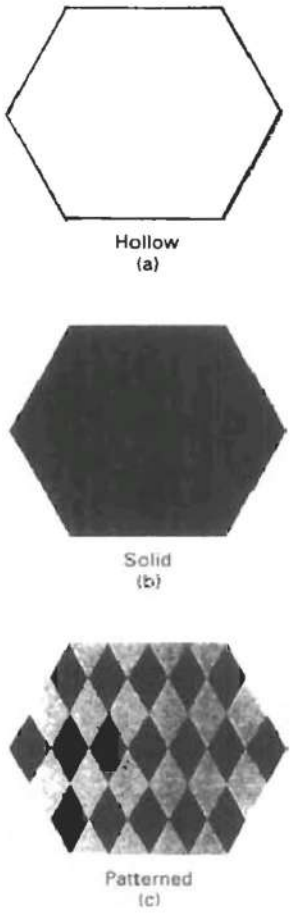


Figure 4-18
Polygon fill styles.

4-4 AREA-FILL ATTRIBUTES

Options for filling a defined region include a choice between a solid color or a patterned fill and choices for the particular colors and patterns. These fill options can be applied to polygon regions or to areas defined with curved boundaries, depending on the capabilities of the available package. In addition, areas can be painted using various brush styles, colors, and transparency parameters.

Fill Styles

Areas are displayed with three basic fill styles: hollow with a color border, filled with a solid color, or filled with a specified pattern or design. A basic fill style is selected in a PHIGS program with the function

```
setInteriorStyle (fs)
```

Values for the fill-style parameter `fs` include *hollow*, *solid*, and *pattern* (Fig. 4-18). Another value for fill style is *hatch*, which is used to fill an area with selected hatching patterns—parallel lines or crossed lines—as in Fig. 4-19. As with line attributes, a selected fill-style value is recorded in the list of system attributes and applied to fill the interiors of subsequently specified areas. Fill selections for parameter `fs` are normally applied to polygon areas, but they can also be implemented to fill regions with curved boundaries.

Hollow areas are displayed using only the boundary outline, with the interior color the same as the background color. A solid fill is displayed in a single color up to and including the borders of the region. The color for a solid interior or for a hollow area outline is chosen with

```
setInteriorColourIndex (fc)
```

where fill-color parameter `fc` is set to the desired color code. A polygon hollow fill is generated with a line-drawing routine as a closed polyline. Solid fill of a region can be accomplished with the scan-line procedures discussed in Section 3-11.

Other fill options include specifications for the edge type, edge width, and edge color of a region. These attributes are set independently of the fill style or fill color, and they provide for the same options as the line-attribute parameters (line type, line width, and line color). That is, we can display area edges dotted or dashed, fat or thin, and in any available color regardless of how we have filled the interior.

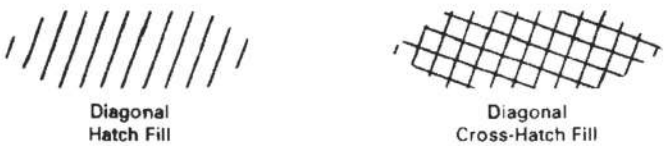


Figure 4-19
Polygon fill using hatch patterns.

Pattern Fill

We select fill patterns with

```
setInteriorStyleIndex (pi)
```

where pattern index parameter `pi` specifies a table position. For example, the following set of statements would fill the area defined in the `fillArea` command with the second pattern type stored in the pattern table:

```
setInteriorStyle (pattern);
setInteriorStyleIndex (2);
fillArea (n, points);
```

Separate tables are set up for hatch patterns. If we had selected *hatch* fill for the interior style in this program segment, then the value assigned to parameter `pi` is an index to the stored patterns in the hatch table.

For fill style *pattern*, table entries can be created on individual output devices with

```
setPatternRepresentation (ws, pi, nx, ny, cp)
```

Parameter `pi` sets the pattern index number for workstation code ws, and `cp` is a two-dimensional array of color codes with `nx` columns and `ny` rows. The following program segment illustrates how this function could be used to set the first entry in the pattern table for workstation 1.

```
cp[1,1] := 4;          cp[2,2] := 4;

cp[1,2] := 0;          cp[2,1] := 0;

setPatternRepresentation (1, 1, 2, 2, cp);
```

Table 4-3 shows the first two entries for this color table. Color array `cp` in this example specifies a pattern that produces alternate red and black diagonal pixel lines on an eight-color system.

When a color array `cp` is to be applied to fill a region, we need to specify the size of the area that is to be covered by each element of the array. We do this by setting the rectangular coordinate extents of the pattern:

```
setPatternSize (dx, dy)
```

where parameters `dx` and `dy` give the coordinate width and height of the array mapping. An example of the coordinate size associated with a pattern array is given in Fig. 4-20. If the values for `dx` and `dy` in this figure are given in screen coordinates, then each element of the color array would be applied to a 2 by 2 screen grid containing four pixels.

A reference position for starting a *pattern* fill is assigned with the statement

```
setPatternReferencePoint (position)
```

Parameter `position` is a pointer to coordinates (xp, yp) that fix the lower left corner of the rectangular pattern. From this starting position, the pattern is then replicated in the x and y directions until the defined area is covered by nonover-

TABLE 4-3
A WORKSTATION PATTERN TABLE WITH TWO ENTRIES, USING THE COLOR CODES OF TABLE 4-1

Index (pi)	*Pattern (cp)*
1	$\begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix}$
2	$\begin{bmatrix} 2 & 1 & 2 \\ 1 & 2 & 1 \\ 2 & 1 & 2 \end{bmatrix}$

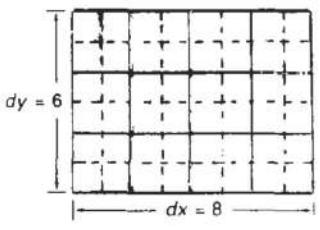


Figure 4-20
A pattern array with 4 columns and 3 rows mapped to an 8 by 6 coordinate rectangle.

lapping copies of the pattern array. The process of filling an area with a rectangular pattern is called **tiling** and rectangular fill patterns are sometimes referred to as **tiling patterns**. Figure 4-21 demonstrates tiling of a triangular fill area starting from a pattern reference point.

To illustrate the use of the pattern commands, the following program example displays a black-and-white pattern in the interior of a parallelogram fill area (Fig. 4-22). The pattern size in this program is set to map each array element to a single pixel.

```
#define WS 1

void patternFill ()
{
  wcPt2 pts[4];
  int bwPattern[3][3] = { 1, 0, 0, 0, 1, 1, 1, 0, 0 };

  pSetPatternRepresentation (WS, 8, 3, 3, bwPattern);

  pts[0].x = 10; pts[0].y = 10;
  pts[1].x = 20; pts[1].y = 10;
  pts[2].x = 28; pts[2].y = 18;
  pts[3].x = 18; pts[3].y = 18;

  pSetFillAreaInteriorStyle (PATTERN);
  pSetFillAreaPatternIndex (8);
  pSetPatternReferencePoint (14, 11);

  pFillArea (4, pts);
}
```

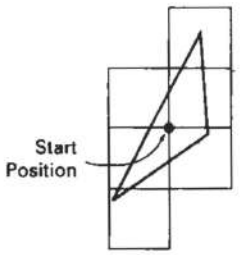


Figure 4-21
Tiling an area from a designated start position. Nonoverlapping adjacent patterns are laid out to cover all scan lines passing through the defined area.

Pattern fill can be implemented by modifying the scan-line procedures discussed in Chapter 3 so that a selected pattern is superimposed onto the scan lines. Beginning from a specified start position for a pattern fill, the rectangular patterns would be mapped vertically to scan lines between the top and bottom of the fill area and horizontally to interior pixel positions across these scan lines. Horizontally, the pattern array is repeated at intervals specified by the value of size parameter dx. Similarly, vertical repeats of the pattern are separated by intervals set with parameter dy. This scan-line pattern procedure applies both to polygons and to areas bounded by curves.

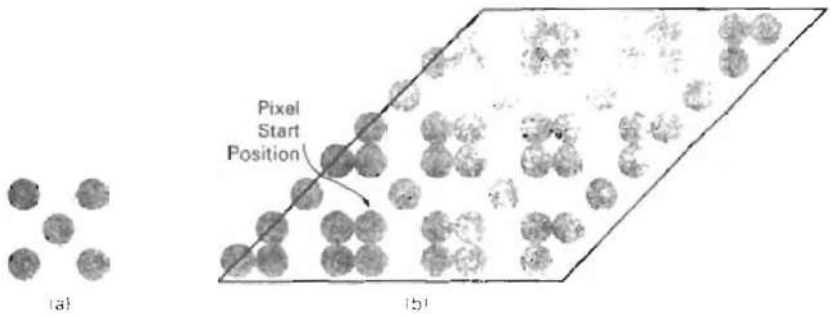


Figure 4-22
A pattern array (a) superimposed on a parallelogram fill area to produce the display (b).

Hatch fill is applied to regions by displaying sets of parallel lines. The fill procedures are implemented to draw either single hatching or cross hatching. Spacing and slope for the hatch lines can be set as parameters in the hatch table. On raster systems, a hatch fill can be specified as a pattern array that sets color values for groups of diagonal pixels.

In many systems, the pattern reference point (xp, yp) is assigned by the system. For instance, the reference point could be set automatically at a polygon vertex. In general, for any fill region, the reference point can be chosen as the lower left corner of the *bounding rectangle* (or *bounding box*) determined by the coordinate extents of the region (Fig. 4-23). To simplify selection of the reference coordinates, some packages always use the screen coordinate origin as the pattern start position, and window systems often set the reference point at the coordinate origin of the window. Always setting (xp, yp) at the coordinate origin also simplifies the tiling operations when each color-array element of a pattern is to be mapped to a single pixel. For example, if the row positions in the pattern array are referenced in reverse (that is, from bottom to top starting at 1), a pattern value is then assigned to pixel position (x, y) in screen or window coordinates as

```
setPixel ( x, y, cp(y mod ny + 1, x mod nx + 1) )
```

where ny and nx specify the number of rows and number of columns in the pattern array. Setting the pattern start position at the coordinate origin, however, effectively attaches the pattern fill to the screen or window background, rather than to the fill regions. Adjacent or overlapping areas filled with the same pattern would show no apparent boundary between the areas. Also, repositioning and refilling an object with the same pattern can result in a shift in the assigned pixel values over the object interior. A moving object would appear to be transparent against a stationary pattern background, instead of moving with a fixed interior pattern.

It is also possible to combine a fill pattern with background colors (including grayscale) in various ways. With a bitmap pattern containing only the digits 1 and 0, the 0 values could be used as transparency indicators to let the background show through. Alternatively, the 1 and 0 digits can be used to fill an interior with two-color patterns. In general, color-fill patterns can be combined in several other ways with background colors. The pattern and background colors can be combined using Boolean operations, or the pattern colors can simply replace the background colors. Figure 4-24 demonstrates how the Boolean and replace operations for a 2 by 2 fill pattern would set pixel values on a binary (black-and-white) system against a particular background pattern.

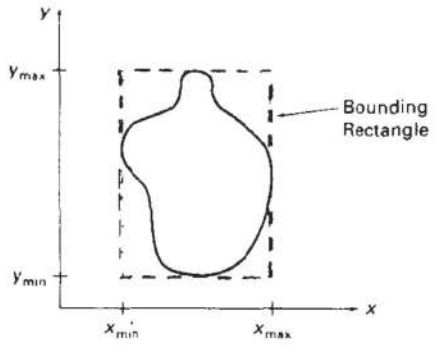


Figure 4-23
Bounding rectangle for a region with coordinate extents x_{min}, x_{max}, y_{min}, and y_{max} in the x and y directions.

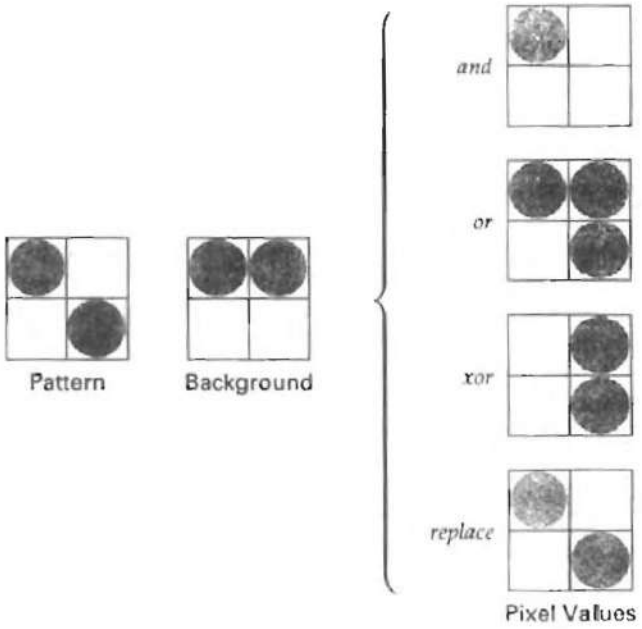


Figure 4-24
Combining a fill pattern with a background pattern using Boolean operations, *and*, *or*, and *xor* (*exclusive or*), and using simple replacement.

Soft Fill

Modified boundary-fill and flood-fill procedures that are applied to repaint areas so that the fill color is combined with the background colors are referred to as **soft-fill** or **tint-fill** algorithms. One use for these fill methods is to soften the fill colors at object borders that have been blurred to antialias the edges. Another is to allow repainting of a color area that was originally filled with a semitransparent brush, where the current color is then a mixture of the brush color and the background colors "behind" the area. In either case, we want the new fill color to have the same variations over the area as the current fill color.

As an example of this type of fill, the **linear soft-fill** algorithm repaints an area that was originally painted by merging a foreground color **F** with a single background color **B**, where $\mathbf{F} \neq \mathbf{B}$. Assuming we know the values for **F** and **B**, we can determine how these colors were originally combined by checking the current color contents of the frame buffer. The current RGB color **P** of each pixel within the area to be refilled is some linear combination of **F** and **B**:

$$\mathbf{P} = t\mathbf{F} + (1 - t)\mathbf{B} \tag{4-1}$$

where the "transparency" factor t has a value between 0 and 1 for each pixel. For values of t less than 0.5, the background color contributes more to the interior color of the region than does the fill color. Vector Equation 4-1 holds for each

RGB component of the colors, with

$$\mathbf{P} = (P_R, P_G, P_B), \qquad \mathbf{F} = (F_R, F_G, F_B), \qquad \mathbf{B} = (B_R, B_G, B_B) \tag{4-2}$$

We can thus calculate the value of parameter t using one of the RGB color components as

$$t = \frac{P_k - B_k}{F_k - B_k} \tag{4-3}$$

where $k = R, G$, or B; and $F_k \neq B_k$. Theoretically, parameter t has the same value for each RGB component, but roundoff to integer codes can result in different values of t for different components. We can minimize this roundoff error by selecting the component with the largest difference between **F** and **B**. This value of t is then used to mix the new fill color **NF** with the background color, using either a modified flood-fill or boundary-fill procedure.

Similar soft-fill procedures can be applied to an area whose foreground color is to be merged with multiple background color areas, such as a checkerboard pattern. When two background colors B_1 and B_2 are mixed with foreground color **F**, the resulting pixel color **P** is

$$\mathbf{P} = t_0\mathbf{F} + t_1\mathbf{B_1} + (1 - t_0 - t_1)\mathbf{B_2} \tag{4-4}$$

where the sum of the coefficients t_0, t_1, and $(1 - t_0 - t_1)$ on the color terms must equal 1. We can set up two simultaneous equations using two of the three RGB color components to solve for the two proportionality parameters, t_0 and t_1. These parameters are then used to mix the new fill color with the two background colors to obtain the new pixel color. With three background colors and one foreground color, or with two background and two foreground colors, we need all three RGB equations to obtain the relative amounts of the four colors. For some foreground and background color combinations, however, the system of two or three RGB equations cannot be solved. This occurs when the color values are all very similar or when they are all proportional to each other.

4-5 CHARACTER ATTRIBUTES

The appearance of displayed characters is controlled by attributes such as font, size, color, and orientation. Attributes can be set both for entire character strings (text) and for individual characters defined as marker symbols.

Text Attributes

There are a great many text options that can be made available to graphics programmers. First of all, there is the choice of font (or typeface), which is a set of characters with a particular design style such as New York, Courier, Helvetica, London, Times Roman, and various special symbol groups. The characters in a selected font can also be displayed with assorted underlining styles (solid, dotted, double), in **boldface**, in *italics*, and in outline or shadow styles. A particular

font and associated style is selected in a PHIGS program by setting an integer code for the text font parameter `tf` in the function

```
setTextFont (tf)
```

Font options can be made available as predefined sets of grid patterns or as character sets designed with polylines and spline curves.

Color settings for displayed text are stored in the system attribute list and used by the procedures that load character definitions into the frame buffer. When a character string is to be displayed, the current color is used to set pixel values in the frame buffer corresponding to the character shapes and positions. Control of text color (or intensity) is managed from an application program with

```
setTextColourIndex (tc)
```

where text color parameter `tc` specifies an allowable color code.

We can adjust text size by scaling the overall dimensions (height and width) of characters or by scaling only the character width. Character size is specified by printers and compositors in *points*, where 1 point is 0.013837 inch (or approximately 1/72 inch). For example, the text you are now reading is a 10-point font. Point measurements specify the size of the *body* of a character (Fig. 4-25), but different fonts with the same point specifications can have different character sizes, depending on the design of the typeface. The distance between the *bottomline* and the *topline* of the character body is the same for all characters in a particular size and typeface, but the body width may vary. *Proportionally spaced fonts* assign a smaller body width to narrow characters such as i, j, l, and f compared to broad characters such as W or M. *Character height* is defined as the distance between the *baseline* and the *capline* of characters. Kerned characters, such as f and j in Fig. 4-25, typically extend beyond the character-body limits, and letters with descenders (g, j, p, q, y) extend below the baseline. Each character is positioned within the character body by a font designer to allow suitable spacing along and between print lines when text is displayed with character bodies touching.

Text size can be adjusted without changing the width-to-height ratio of characters with

```
setCharacterHeight (ch)
```

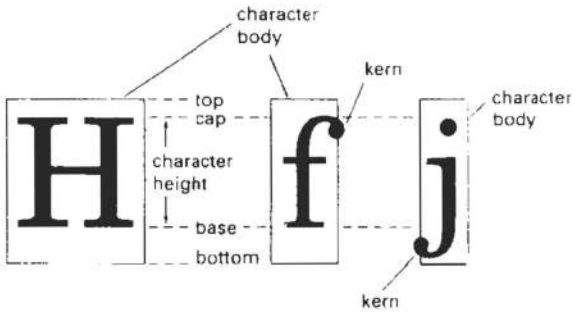


Figure 4-25
Character body.

Figure 4-26
The effect of different character-height settings on displayed text.

Parameter ch is assigned a real value greater than 0 to set the coordinate height of capital letters: the distance between baseline and capline in user coordinates. This setting also affects character-body size, so that the width and spacing of characters is adjusted to maintain the same text proportions. For instance, doubling the height also doubles the character width and the spacing between characters. Figure 4-26 shows a character string displayed with three different character heights.

The width only of text can be set with the function

```
setCharacterExpansionFactor (cw)
```

where the character-width parameter cw is set to a positive real value that scales the body width of characters. Text height is unaffected by this attribute setting. Examples of text displayed with different character expansions is given in Fig. 4-27.

Spacing between characters is controlled separately with

```
setCharacterSpacing (cs)
```

where the character-spacing parameter cs can be assigned any real value. The value assigned to cs determines the spacing between character bodies along print lines. Negative values for cs overlap character bodies; positive values insert space to spread out the displayed characters. Assigning the value 0 to cs causes text to be displayed with no space between character bodies. The amount of spacing to be applied is determined by multiplying the value of cs by the character height (distance between baseline and capline). In Fig. 4-28, a character string is displayed with three different settings for the character-spacing parameter.

The orientation for a displayed character string is set according to the direction of the **character up vector:**

```
setCharacterUpVector (upvect)
```

Parameter upvect in this function is assigned two values that specify the x and y vector components. Text is then displayed so that the orientation of characters from baseline to capline is in the direction of the up vector. For example, with upvect = (1, 1), the direction of the up vector is 45° and text would be displayed as shown in Fig. 4-29. A procedure for orienting text rotates characters so that the sides of character bodies, from baseline to capline, are aligned with the up vector. The rotated character shapes are then scan converted into the frame buffer.

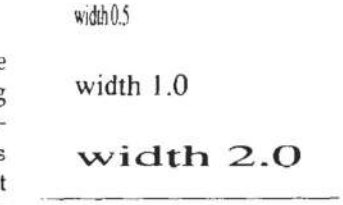


Figure 4-27
The effect of different character-width settings on displayed text.

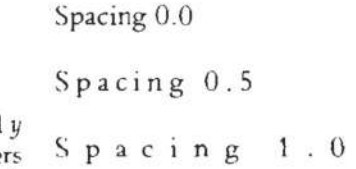


Figure 4-28
The effect of different character spacings on displayed text.

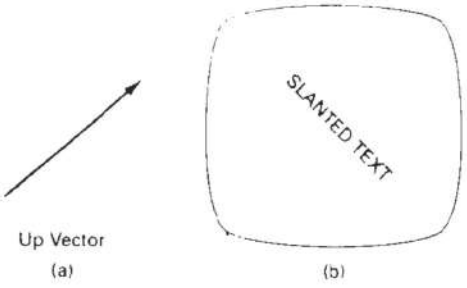


Figure 4-29
Direction of the up vector (a) controls the orientation of displayed text (b).

It is useful in many applications to be able to arrange character strings vertically or horizontally (Fig. 4-30). An attribute parameter for this option is set with the statement

```
setTextPath (tp)
```

where the text-path parameter `tp` can be assigned the value: *right*, *left*, *up*, or *down*. Examples of text displayed with these four options are shown in Fig. 4-31. A procedure for implementing this option must transform the character patterns into the specified orientation before transferring them to the frame buffer.

Character strings can also be oriented using a combination of up-vector and text-path specifications to produce slanted text. Figure 4-32 shows the directions of character strings generated by the various text-path settings for a 45° up vector. Examples of text generated for text-path values *down* and *right* with this up vector are illustrated in Fig. 4-33.

Another handy attribute for character strings is alignment. This attribute specifies how text is to be positioned with respect to the start coordinates. Alignment attributes are set with

```
setTextAlignment (h, v)
```

where parameters `h` and `v` control horizontal and vertical alignment, respectively. Horizontal alignment is set by assigning `h` a value of *left*, *centre*, or *right*. Vertical alignment is set by assigning `v` a value of *top*, *cap*, *half*, *base*, or *bottom*. The interpretation of these alignment values depends on the current setting for the text path. Figure 4-34 shows the position of the alignment settings when text is to be displayed horizontally to the right or vertically down. Similar interpretations apply to text path values of *left* and *up*. The "most natural" alignment for a particular text path is chosen by assigning the value *normal* to the `h` and `v` parameters. Figure 4-35 illustrates common alignment positions for horizontal and vertical text labels.

A precision specification for text display is given with

```
setTextPrecision (tpr)
```

where text precision parameter `tpr` is assigned one of the values: *string*, *char*, or *stroke*. The highest-quality text is displayed when the precision parameter is set to the value *stroke*. For this precision setting, greater detail would be used in defining the character shapes, and the processing of attribute selections and other

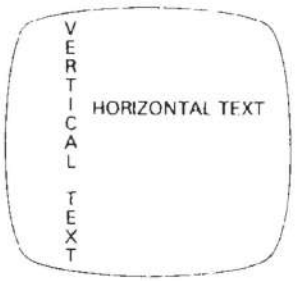


Figure 4-30
Text path attributes can be set to produce horizontal or vertical arrangements of character strings.

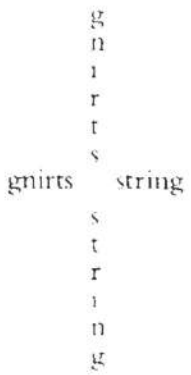


Figure 4-31
Text displayed with the four text-path options.

string-manipulation procedures would be carried out to the highest possible accuracy. The lowest-quality precision setting, *string*, is used for faster display of character strings. At this precision, many attribute selections such as text path are ignored, and string-manipulation procedures are simplified to reduce processing time.

Marker Attributes

A marker symbol is a single character that can be displayed in different colors and in different sizes. Marker attributes are implemented by procedures that load the chosen character into the raster at the defined positions with the specified color and size.

We select a particular character to be the marker symbol with

```
setMarkerType (mt)
```

where marker type parameter `mt` is set to an integer code. Typical codes for marker type are the integers 1 through 5, specifying, respectively, a dot (·), a vertical cross (+), an asterisk (*), a circle (o), and a diagonal cross (×). Displayed marker types are centered on the marker coordinates.

We set the marker size with

```
setMarkerSizeScaleFactor (ms)
```

with parameter marker size `ms` assigned a positive number. This scaling parameter is applied to the nominal size for the particular marker symbol chosen. Values greater than 1 produce character enlargement; values less than 1 reduce the marker size.

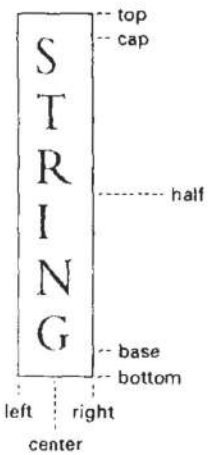


Figure 4-34
Alignment attribute values for horizontal and vertical strings.

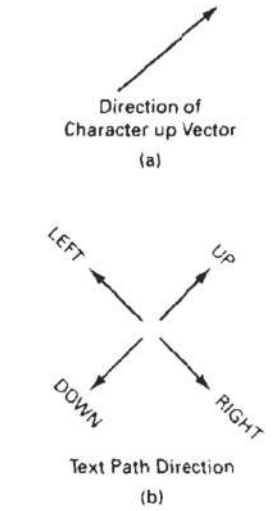


Figure 4-32
An up-vector specification (a) controls the direction of the text path (b).

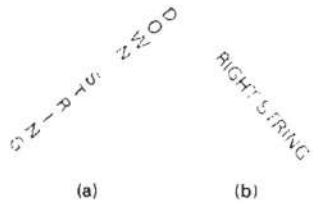


Figure 4-33
The 45° up vector in Fig. 4-32 produces the display (a) for a *down* path and the display (b) for a *right* path.

Figure 4-35
Character-string alignments.

Marker color is specified with

```
setPolymarkerColourIndex (mc)
```

A selected color code for parameter mc is stored in the current attribute list and used to display subsequently specified marker primitives.

4-6 BUNDLED ATTRIBUTES

With the procedures we have considered so far, each function references a single attribute that specifies exactly how a primitive is to be displayed with that attribute setting. These specifications are called **individual** (or **unbundled**) attributes, and they are meant to be used with an output device that is capable of displaying primitives in the way specified. If an application program, employing individual attributes, is interfaced to several output devices, some of the devices may not have the capability to display the intended attributes. A program using individual color attributes, for example, may have to be modified to produce acceptable output on a monochromatic monitor.

Individual attribute commands provide a simple and direct method for specifying attributes when a single output device is used. When several kinds of output devices are available at a graphics installation, it is convenient for a user to be able to say how attributes are to be interpreted on each of the different devices. This is accomplished by setting up tables for each output device that lists sets of attribute values that are to be used on that device to display each primitive type. A particular set of attribute values for a primitive on each output device is then chosen by specifying the appropriate table index. Attributes specified in this manner are called **bundled** attributes. The table for each primitive that defines groups of attribute values to be used when displaying that primitive on a particular output device is called a **bundle table**.

Attributes that may be bundled into the workstation table entries are those that do not involve coordinate specifications, such as color and line type. The choice between a bundled or an unbundled specification is made by setting a switch called the **aspect source flag** for each of these attributes:

```
setIndividualASF (attributeptr, flagptr)
```

where parameter `attributeptr` points to a list of attributes, and parameter `flagptr` points to the corresponding list of aspect source flags. Each aspect source flag can be assigned a value of *individual* or *bundled*. Attributes that may be bundled are listed in the following sections.

Bundled Line Attributes

Entries in the bundle table for line attributes on a specified workstation are set with the function

```
setPolylineRepresentation (ws, li, lt, lw, lc)
```

Parameter ws is the workstation identifier, and line index parameter li defines the bundle table position. Parameters lt, lw, and lc are then bundled and assigned values to set the line type, line width, and line color specifications, respectively, for the designated table index. For example, the following statements define groups of line attributes that are to be referenced as index number 3 on two different workstations:

```
setPolylineRepresentation (1, 3, 2, 0.5, 1);

setPolylineRepresentation (4, 3, 1, 1, 7);
```

A polyline that is assigned a table index value of 3 would then be displayed using dashed lines at half thickness in a blue color on workstation 1; while on workstation 4, this same index generates solid, standard-sized white lines.

Once the bundle tables have been set up, a group of bundled line attributes is chosen for each workstation by specifying the table index value:

```
setPolylineIndex (li)
```

Subsequent polyline commands would then generate lines on each workstation according to the set of bundled attribute values defined at the table position specified by the value of the line index parameter li.

Bundled Area-Fill Attributes

Table entries for bundled area-fill attributes are set with

```
setInteriorRepresentation (ws, fi, fs, pi, fc)
```

which defines the attribute list corresponding to fill index fi on workstation ws. Parameters fs, pi, and fc are assigned values for the fill style, pattern index, and fill color, respectively, on the designated workstation. Similar bundle tables can also be set up for edge attributes of polygon fill areas.

A particular attribute bundle is then selected from the table with the function

```
setInteriorIndex (fi)
```

Subsequently defined fill areas are then displayed on each active workstation according to the table entry specified by the fill index parameter fi. Other fill-area attributes, such as pattern reference point and pattern size, are independent of the workstation designation and are set with the functions previously described.

Bundled Text Attributes

The function

```
setTextRepresentation (ws, ti, tf, tp, te, ts, tc)
```

bundles values for text font, precision, expansion factor, size, and color in a table position for workstation ws that is specified by the value assigned to text index

parameter `ti`. Other text attributes, including character up vector, text path, character height, and text alignment are set individually.

A particular text index value is then chosen with the function

```
setTextIndex (ti)
```

Each text function that is then invoked is displayed on each workstation with the set of attributes referenced by this table position.

Bundled Marker Attributes

Table entries for bundled marker attributes are set up with

```
setPolymarkerRepresentation (ws, mi, mt, ms, mc)
```

This defines the marker type, marker scale factor, and marker color for index `mi` on workstation `ws`. Bundle table selections are then made with the function

```
setPolymarkerIndex (mi)
```

4-7 INQUIRY FUNCTIONS

Current settings for attributes and other parameters, such as workstation types and status, in the system lists can be retrieved with inquiry functions. These functions allow current values to be copied into specified parameters, which can then be saved for later reuse or used to check the current state of the system if an error occurs.

We check current attribute values by stating the name of the attribute in the inquiry function. For example, the functions

```
inquirePolylineIndex (lastli)
```

and

```
inquireInteriorColourIndex (lastfc)
```

copy the current values for line index and fill color into parameters `lastli` and `lastfc`. The following program segment illustrates reusing the current line type value after a set of lines are drawn with a new line type.

```
inquireLinetype (oldlt);

setLinetype (newlt);
     .
     .
     .

setLinetype (oldlt);
```

4-8 ANTIALIASING

Displayed primitives generated by the raster algorithms discussed in Chapter 3 have a jagged, or stairstep, appearance because the sampling process digitizes coordinate points on an object to discrete integer pixel positions. This distortion of information due to low-frequency sampling (undersampling) is called **aliasing**. We can improve the appearance of displayed raster lines by applying **antialiasing** methods that compensate for the undersampling process.

An example of the effects of undersampling is shown in Fig. 4-36. To avoid losing information from such periodic objects, we need to set the sampling frequency to at least twice that of the highest frequency occurring in the object, referred to as the **Nyquist sampling frequency** (or Nyquist sampling rate) f_s:

$$f_s = 2f_{\max} \tag{4-5}$$

Another way to state this is that the sampling interval should be no larger than one-half the cycle interval (called the **Nyquist sampling interval**). For x-interval sampling, the Nyquist sampling interval Δx_s is

$$\Delta x_s = \frac{\Delta x_{\text{cycle}}}{2} \tag{4-6}$$

where $\Delta x_{\text{cycle}} = 1/f_{\max}$. In Fig. 4-36, our sampling interval is one and one-half times the cycle interval, so the sampling interval is at least three times too big. If we want to recover all the object information for this example, we need to cut the sampling interval down to one-third the size shown in the figure.

One way to increase sampling rate with raster systems is simply to display objects at higher resolution. But even at the highest resolution possible with current technology, the jaggies will be apparent to some extent. There is a limit to how big we can make the frame buffer and still maintain the refresh rate at 30 to 60 frames per second. And to represent objects accurately with continuous parameters, we need arbitrarily small sampling intervals. Therefore, unless hardware technology is developed to handle arbitrarily large frame buffers, increased screen resolution is not a complete solution to the aliasing problem.

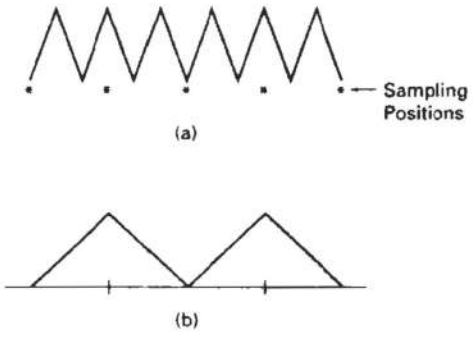


Figure 4-36
Sampling the periodic shape in (a) at the marked positions produces the aliased lower-frequency representation in (b).

With raster systems that are capable of displaying more than two intensity levels (color or gray scale), we can apply antialiasing methods to modify pixel intensities. By appropriately varying the intensities of pixels along the boundaries of primitives, we can smooth the edges to lessen the jagged appearance.

A straightforward antialiasing method is to increase sampling rate by treating the screen as if it were covered with a finer grid than is actually available. We can then use multiple sample points across this finer grid to determine an appropriate intensity level for each screen pixel. This technique of sampling object characteristics at a high resolution and displaying the results at a lower resolution is called **supersampling** (or **postfiltering**, since the general method involves computing intensities at subpixel grid positions, then combining the results to obtain the pixel intensities). Displayed pixel positions are spots of light covering a finite area of the screen, and not infinitesimal mathematical points. Yet in the line and fill-area algorithms we have discussed, the intensity of each pixel is determined by the location of a single point on the object boundary. By supersampling, we obtain intensity information from multiple points that contribute to the overall intensity of a pixel.

An alternative to supersampling is to determine pixel intensity by calculating the areas of overlap of each pixel with the objects to be displayed. Antialiasing by computing overlap areas is referred to as **area sampling** (or **prefiltering**, since the intensity of the pixel as a whole is determined without calculating subpixel intensities). Pixel overlap areas are obtained by determining where object boundaries intersect individual pixel boundaries.

Raster objects can also be antialiased by shifting the display location of pixel areas. This technique, called **pixel phasing**, is applied by "micropositioning" the electron beam in relation to object geometry.

Supersampling Straight Line Segments

Supersampling straight lines can be performed in several ways. For the gray-scale display of a straight-line segment, we can divide each pixel into a number of subpixels and count the number of subpixels that are along the line path. The intensity level for each pixel is then set to a value that is proportional to this subpixel count. An example of this method is given in Fig. 4-37. Each square pixel area is divided into nine equal-sized square subpixels, and the shaded regions show the subpixels that would be selected by Bresenham's algorithm. This scheme provides for three intensity settings above zero, since the maximum number of subpixels that can be selected within any pixel is three. For this example, the pixel at position (10, 20) is set to the maximum intensity (level 3); pixels at (11, 21) and (12, 21) are each set to the next highest intensity (level 2); and pixels at (11, 20) and (12, 22) are each set to the lowest intensity above zero (level 1). Thus the line intensity is spread out over a greater number of pixels, and the stairstep effect is smoothed by displaying a somewhat blurred line path in the vicinity of the stair steps (between horizontal runs). If we want to use more intensity levels to antialiase the line with this method, we increase the number of sampling positions across each pixel. Sixteen subpixels gives us four intensity levels above zero; twenty-five subpixels gives us five levels; and so on.

In the supersampling example of Fig. 4-37, we considered pixel areas of finite size, but we treated the line as a mathematical entity with zero width. Actually, displayed lines have a width approximately equal to that of a pixel. If we take the finite width of the line into account, we can perform supersampling by setting each pixel intensity proportional to the number of subpixels inside the

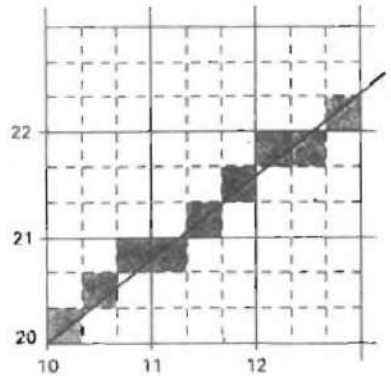


Figure 4-37
Supersampling subpixel positions along a straight line segment whose left endpoint is at screen coordinates (10, 20).

polygon representing the line area. A subpixel can be considered to be inside the line if its lower left corner is inside the polygon boundaries. An advantage of this supersampling procedure is that the number of possible intensity levels for each pixel is equal to the total number of subpixels within the pixel area. For the example in Fig. 4-37, we can represent this line with finite width by positioning the polygon boundaries parallel to the line path as in Fig. 4-38. And each pixel can now be set to one of nine possible brightness levels above zero.

Another advantage of supersampling with a finite-width line is that the total line intensity is distributed over more pixels. In Fig. 4-38, we now have the pixel at grid position (10, 21) turned on (at intensity level 2), and we also pick up contributions from pixels immediately below and immediately to the left of position (10, 21). Also, if we have a color display, we can extend the method to take background colors into account. A particular line might cross several different color areas, and we can average subpixel intensities to obtain pixel color settings. For instance, if five subpixels within a particular pixel area are determined to be inside the boundaries for a red line and the remaining four pixels fall within a blue background area, we can calculate the color for this pixel as

$$\text{pixel}_{\text{color}} = (5 \cdot \text{red} + 4 \cdot \text{blue})/9$$

The trade-off for these gains from supersampling a finite-width line is that identifying interior subpixels requires more calculations than simply determining which subpixels are along the line path. These calculations are also complicated by the positioning of the line boundaries in relation to the line path. This posi-

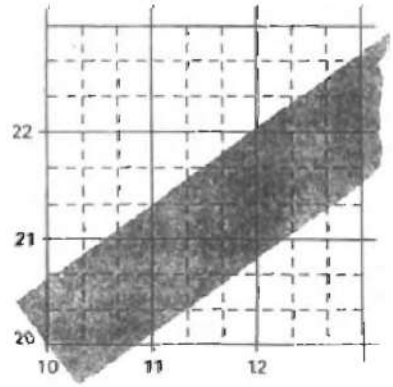


Figure 4-38
Supersampling subpixel positions in relation to the interior of a line of finite width.

tioning depends on the slope of the line. For a 45° line, the line path is centered on the polygon area; but for either a horizontal or a vertical line, we want the line path to be one of the polygon boundaries. For instance, a horizontal line passing through grid coordinates (10, 20) would be represented as the polygon bounded by horizontal grid lines $y = 20$ and $y = 21$. Similarly, the polygon representing a vertical line through (10, 20) would have vertical boundaries along grid lines $x = 10$ and $x = 11$. For lines with slope $|m| < 1$, the mathematical line path is positioned proportionately closer to the lower polygon boundary; and for lines with slope $|m| > 1$, this line path is placed closer to the upper polygon boundary.

Pixel-Weighting Masks

Supersampling algorithms are often implemented by giving more weight to subpixels near the center of a pixel area, since we would expect these subpixels to be more important in determining the overall intensity of a pixel. For the 3 by 3 pixel subdivisions we have considered so far, a weighting scheme as in Fig. 4-39 could be used. The center subpixel here is weighted four times that of the corner subpixels and twice that of the remaining subpixels. Intensities calculated for each grid of nine subpixels would then be averaged so that the center subpixel is weighted by a factor of 1/4; the top, bottom, and side subpixels are each weighted by a factor of 1/8; and the corner subpixels are each weighted by a factor of 1/16. An array of values specifying the relative importance of subpixels is sometimes referred to as a "mask" of subpixel weights. Similar masks can be set up for larger subpixel grids. Also, these masks are often extended to include contributions from subpixels belonging to neighboring pixels, so that intensities can be averaged over adjacent pixels.

Area Sampling Straight Line Segments

We perform area sampling for a straight line by setting each pixel intensity proportional to the area of overlap of the pixel with the finite-width line. The line can be treated as a rectangle, and the section of the line area between two adjacent vertical (or two adjacent horizontal) screen grid lines is then a trapezoid. Overlap areas for pixels are calculated by determining how much of the trapezoid overlaps each pixel in that vertical column (or horizontal row). In Fig. 4-38, the pixel with screen grid coordinates (10, 20) is about 90 percent covered by the line area, so its intensity would be set to 90 percent of the maximum intensity. Similarly, the pixel at (10, 21) would be set to an intensity of about 15 percent of maximum. A method for estimating pixel overlap areas is illustrated by the supersampling example in Fig. 4-38. The total number of subpixels within the line boundaries is approximately equal to the overlap area, and this estimation is improved by using finer subpixel grids. With color displays, the areas of pixel overlap with different color regions is calculated and the final pixel color is taken as the average color of the various overlap areas.

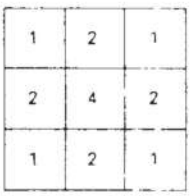


Figure 4-39
Relative weights for a grid of 3 by 3 subpixels.

Filtering Techniques

A more accurate method for antialiasing lines is to use **filtering** techniques. The method is similar to applying a weighted pixel mask, but now we imagine a continuous *weighting surface* (or *filter function*) covering the pixel. Figure 4-40 shows examples of rectangular, conical, and Gaussian filter functions. Methods for applying the filter function are similar to applying a weighting mask, but now we

integrate over the pixel surface to obtain the weighted average intensity. To reduce computation, table lookups are commonly used to evaluate the integrals.

Pixel Phasing

On raster systems that can address subpixel positions within the screen grid, pixel phasing can be used to antialias objects. Stairsteps along a line path or object boundary are smoothed out by moving (micropositioning) the electron beam to more nearly approximate positions specified by the object geometry. Systems incorporating this technique are designed so that individual pixel positions can be shifted by a fraction of a pixel diameter. The electron beam is typically shifted by 1/4, 1/2, or 3/4 of a pixel diameter to plot points closer to the true path of a line or object edge. Some systems also allow the size of individual pixels to be adjusted as an additional means for distributing intensities. Figure 4-41 illustrates the antialiasing effects of pixel phasing on a variety of line paths.

Compensating for Line Intensity Differences

Antialiasing a line to soften the stairstep effect also compensates for another raster effect, illustrated in Fig. 4-42. Both lines are plotted with the same number of pixels, yet the diagonal line is longer than the horizontal line by a factor of $\sqrt{2}$. The visual effect of this is that the diagonal line appears less bright than the horizontal line, because the diagonal line is displayed with a lower intensity per unit length. A line-drawing algorithm could be adapted to compensate for this effect by adjusting the intensity of each line according to its slope. Horizontal and vertical lines would be displayed with the lowest intensity, while 45° lines would be given the highest intensity. But if antialiasing techniques are applied to a display,

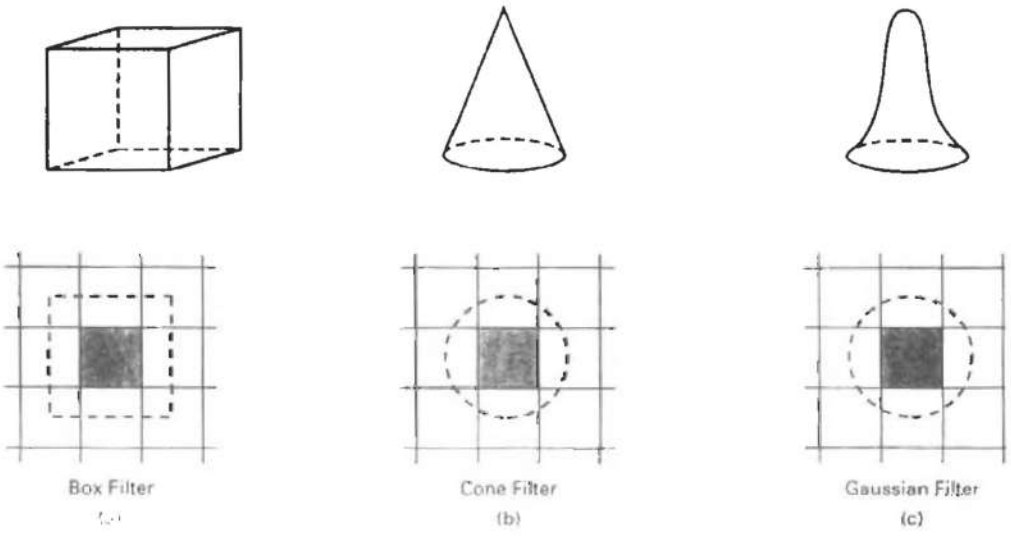


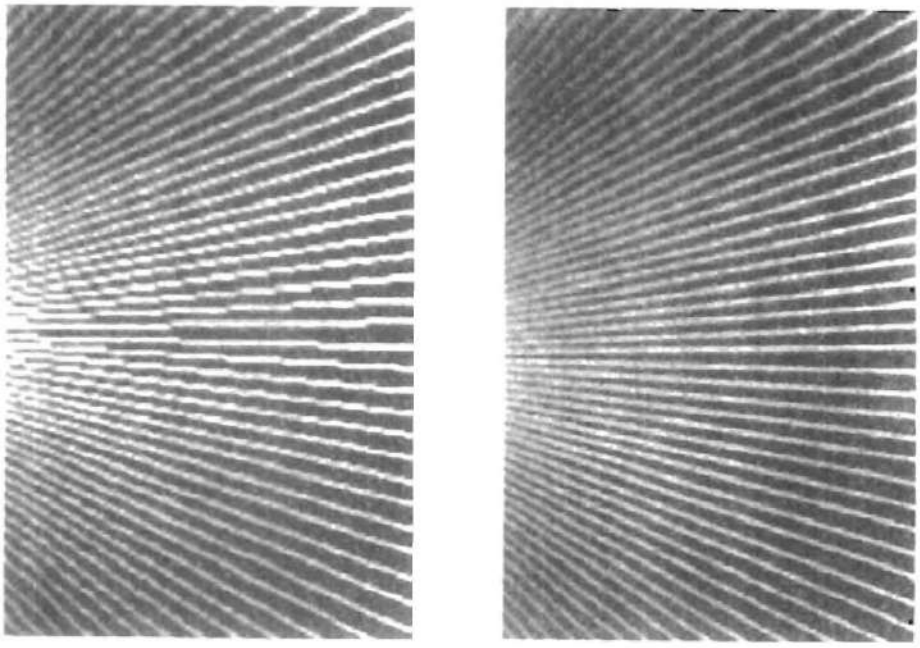
Figure 4-40
Common filter functions used to antialias line paths. The volume of each filter is normalized to 1, and the height gives the relative weight at any subpixel position.

intensities are automatically compensated. When the finite width of lines is taken into account, pixel intensities are adjusted so that lines display a total intensity proportional to their length.

Antialiasing Area Boundaries

The antialiasing concepts we have discussed for lines can also be applied to the boundaries of areas to remove their jagged appearance. We can incorporate these procedures into a scan-line algorithm to smooth the area outline as the area is generated.

If system capabilities permit the repositioning of pixels, area boundaries can be smoothed by adjusting boundary pixel positions so that they are along the line defining an area boundary. Other methods adjust each pixel intensity at a boundary position according to the percent of pixel area that is inside the boundary. In Fig. 4-43, the pixel at position (x, y) has about half its area inside the polygon boundary. Therefore, the intensity at that position would be adjusted to one-half its assigned value. At the next position $(x + 1, y + 1)$ along the boundary, the intensity is adjusted to about one-third the assigned value for that point. Similar adjustments, based on the percent of pixel area coverage, are applied to the other intensity values around the boundary.



(a) (b)

Figure 4-41
Jagged lines (a), plotted on the Merlin 9200 system, are smoothed (b) with an antialiasing technique called pixel phasing. This technique increases the number of addressable points on the system from 768 × 576 to 3072 × 2304. (*Courtesy of Megatek Corp.*)

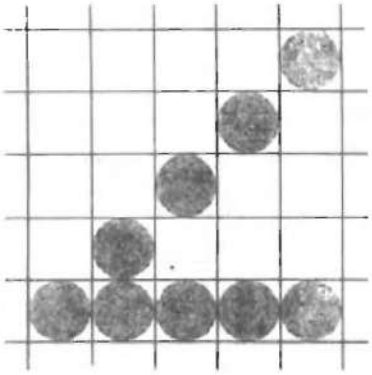


Figure 4-42
Unequal-length lines displayed with the same number of pixels in each line.

Supersampling methods can be applied by subdividing the total area and determining the number of subpixels inside the area boundary. A pixel partitioning into four subareas is shown in Fig. 4-44. The original 4 by 4 grid of pixels is turned into an 8 by 8 grid, and we now process eight scan lines across this grid instead of four. Figure 4-45 shows one of the pixel areas in this grid that overlaps an object boundary. Along the two scan lines we determine that three of the subpixel areas are inside the boundary. So we set the pixel intensity at 75 percent of its maximum value.

Another method for determining the percent of pixel area within a boundary, developed by Pitteway and Watkinson, is based on the midpoint line algorithm. This algorithm selects the next pixel along a line by determining which of two pixels is closer to the line by testing the location of the midposition between the two pixels. As in the Bresenham algorithm, we set up a decision parameter p whose sign tells us which of the next two candidate pixels is closer to the line. By slightly modifying the form of p, we obtain a quantity that also gives the percent of the current pixel area that is covered by an object.

We first consider the method for a line with slope m in the range from 0 to 1. In Fig. 4-46, a straight line path is shown on a pixel grid. Assuming that the pixel at position (x_k, y_k) has been plotted, the next pixel nearest the line at $x = x_k + 1$ is either the pixel at y_k or the one at $y_k + 1$. We can determine which pixel is nearer with the calculation

$$y - y_{\text{mid}} = [m(x_k + 1) + b] - (y_k + 0.5) \tag{4-7}$$

This gives the vertical distance from the actual y coordinate on the line to the halfway point between pixels at position y_k and $y_k + 1$. If this difference calculation is negative, the pixel at y_k is closer to the line. If the difference is positive, the

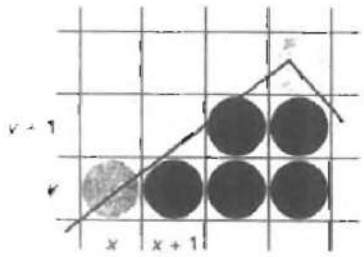


Figure 4-43
Adjusting pixel intensities along an area boundary.

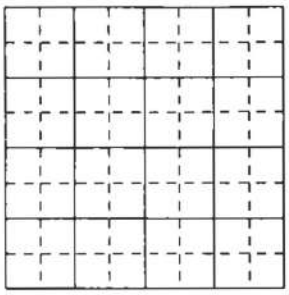


Figure 4-44
A 4 by 4 pixel section of a raster display subdivided into an 8 by 8 grid.

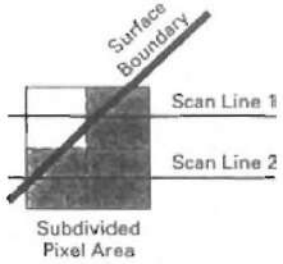


Figure 4-45
A subdivided pixel area with three subdivisions inside an object boundary line.

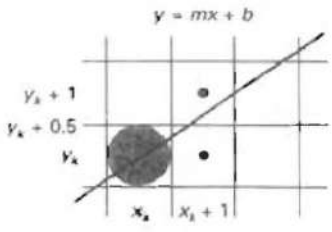


Figure 4-46
Boundary edge of an area passing through a pixel grid section.

pixel at $y_k + 1$ is closer. We can adjust this calculation so that it produces a positive number in the range from 0 to 1 by adding the quantity $1 - m$:

$$p = [m(x_k + 1) + b] - (y_k + 0.5) + (1 - m) \tag{4-8}$$

Now the pixel at y_k is nearer if $p < 1 - m$, and the pixel at $y_k + 1$ is nearer if $p > 1 - m$.

Parameter p also measures the amount of the current pixel that is overlapped by the area. For the pixel at (x_k, y_k) in Fig. 4-47, the interior part of the pixel has an area that can be calculated as

$$\text{area} = mx_k + b - y_k + 0.5 \tag{4-9}$$

This expression for the overlap area of the pixel at (x_k, y_k) is the same as that for parameter p in Eq. 4-8. Therefore, by evaluating p to determine the next pixel position along the polygon boundary, we also determine the percent of area coverage for the current pixel.

We can generalize this algorithm to accommodate lines with negative slopes and lines with slopes greater than 1. This calculation for parameter p could then be incorporated into a midpoint line algorithm to locate pixel positions and an object edge and to concurrently adjust pixel intensities along the boundary lines. Also, we can adjust the calculations to reference pixel coordinates at their lower left coordinates and maintain area proportions as discussed in Section 3-10.

At polygon vertices and for very skinny polygons, as shown in Fig. 4-48, we have more than one boundary edge passing through a pixel area. For these cases, we need to modify the Pitteway–Watkinson algorithm by processing all edges passing through a pixel and determining the correct interior area.

Filtering techniques discussed for line antialiasing can also be applied to area edges. Also, the various antialiasing methods can be applied to polygon areas or to regions with curved boundaries. Boundary equations are used to estimate area overlap of pixel regions with the area to be displayed. And coherence techniques are used along and between scan lines to simplify the calculations.

SUMMARY

In this chapter, we have explored the various attributes that control the appearance of displayed primitives. Procedures for displaying primitives use attribute settings to adjust the output of algorithms for line-generation, area-filling, and text-string displays.

The basic line attributes are line type, line color, and line width. Specifications for line type include solid, dashed, and dotted lines. Line-color specifications can be given in terms of RGB components, which control the intensity of the three electron guns in an RGB monitor. Specifications for line width are given in terms of multiples of a standard, one-pixel-wide line. These attributes can be applied to both straight lines and curves.

To reduce the size of the frame buffer, some raster systems use a separate color lookup table. This limits the number of colors that can be displayed to the size of the lookup table. Full-color systems are those that provide 24 bits per pixel and no separate color lookup table.

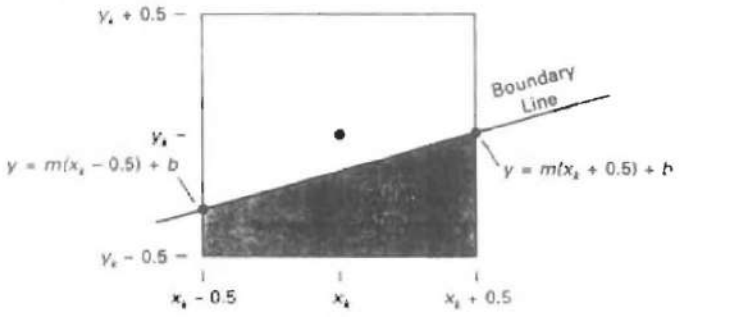


Figure 4-47
Overlap area of a pixel rectangle, centered at position (x_k, y_k), with the interior of a polygon area.

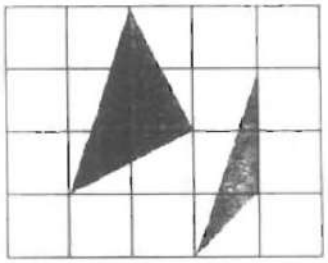


Figure 4-48
Polygons with more than one boundary line passing through individual pixel regions.

Fill-area attributes include the fill style and the fill color or the fill pattern. When the fill style is to be solid, the fill color specifies the color for the solid fill of the polygon interior. A hollow-fill style produces an interior in the background color and a border in the fill color. The third type of fill is patterned. In this case, a selected array pattern is used to fill the polygon interior.

An additional fill option provided in some packages is soft fill. This fill has applications in antialiasing and in painting packages. Soft-fill procedures provide a new fill color for a region that has the same variations as the previous fill color. One example of this approach is the linear soft-fill algorithm that assumes that the previous fill was a linear combination of foreground and background colors. This same linear relationship is then determined from the frame-buffer settings and used to repaint the area in a new color.

Characters, defined as pixel grid patterns or as outline fonts, can be displayed in different colors, sizes, and orientations. To set the orientation of a character string, we select a direction for the character up vector and a direction for the text path. In addition, we can set the alignment of a text string in relation to the start coordinate position. Marker symbols can be displayed using selected characters of various sizes and colors.

Graphics packages can be devised to handle both unbundled and bundled attribute specifications. Unbundled attributes are those that are defined for only one type of output device. Bundled attribute specifications allow different sets of attributes to be used on different devices, but accessed with the same index number in a bundle table. Bundle tables may be installation-defined, user-defined, or both. Functions to set the bundle table values specify workstation type and the attribute list for a given attribute index.

To determine current settings for attributes and other parameters, we can invoke inquiry functions. In addition to retrieving color and other attribute information, we can obtain workstation codes and status values with inquiry functions.

Because scan conversion is a digitizing process on raster systems, displayed primitives have a jagged appearance. This is due to the undersampling of information which rounds coordinate values to pixel positions. We can improve the appearance of raster primitives by applying antialiasing procedures that adjust pixel intensities. One method for doing this is to supersample. That is, we consider each pixel to be composed of subpixels and we calculate the intensity of the

subpixels and average the values of all subpixels. Alternatively, we can perform area sampling and determine the percentage of area coverage for a screen pixel, then set the pixel intensity proportional to this percentage. We can also weight the subpixel contributions according to position, giving higher weights to the central subpixels. Another method for antialiasing is to build special hardware configurations that can shift pixel positions.

Table 4-4 lists the attributes discussed in this chapter for the output primitive classifications: line, fill area, text, and marker. The attribute functions that can be used in graphics packages are listed for each category.

TABLE 4-4
SUMMARY OF ATTRIBUTES

Output Primitive Type	*Associated Attributes*	*Attribute-Setting Functions*	*Bundled-Attribute Functions*
Line	Type	`setLinetype`	`setPolylineIndex`
	Width	`setLineWidthScaleFactor`	`setPolylineRepresentation`
	Color	`setPolylineColourIndex`	
Fill Area	Fill Style	`setInteriorStyle`	`setInteriorIndex`
	Fill Color	`setInteriorColorIndex`	`setInteriorRepresentation`
	Pattern	`setInteriorStyleIndex`	
		`setPatternRepresentation`	
		`setPatternSize`	
		`setPatternReferencePoint`	
Text	Font	`setTextFont`	`setTextIndex`
	Color	`setTextColourIndex`	`setTextRepresentation`
	Size	`setCharacterHeight`	
		`setCharacterExpansionFactor`	
	Orientation	`setCharacterUpVector`	
		`setTextPath`	
		`setTextAlignment`	
Marker	Type	`setMarkerType`	`setPolymarkerIndex`
	Size	`setMarkerSizeScaleFactor`	`setPolymarkerRepresentation`
	Color	`setPolymarkerColourIndex`	

REFERENCES

Color and grayscale considerations are discussed in Crow (1978) and in Heckbert (1982).

Soft-fill techniques are given in Fishkin and Barsky (1984).

Antialiasing techniques are discussed in Pitteway and Watkinson (1980), Crow (1981), Turkowski (1982), Korein and Badler (1983), and Kirk and Avro, Schilling, and Wu (1991).

Attribute functions in PHIGS are discussed in Howard et al. (1991), Hopgood and Duce (1991), Gaskins (1992), and Blake (1993). For information on GKS workstations and attributes, see Hopgood et al. (1983) and Enderle, Kansy, and Pfaff (1984).

EXERCISES

4-1. Implement the line-type function by modifying Bresenham's line-drawing algorithm to display either solid, dashed, or dotted lines.

4-2. Implement the line-type function with a midpoint line algorithm to display either solid, dashed, or dotted lines.

4-3. Devise a parallel method for implementing the line-type function.

4-4. Devise a parallel method for implementing the line-width function.

4-5. A line specified by two endpoints and a width can be converted to a rectangular polygon with four vertices and then displayed using a scan-line method. Develop an efficient algorithm for computing the four vertices needed to define such a rectangle using the line endpoints and line width.

4-6. Implement the line-width function in a line-drawing program so that any one of three line widths can be displayed.

4-7. Write a program to output a line graph of three data sets defined over the same x coordinate range. Input to the program is to include the three sets of data values, labeling for the axes, and the coordinates for the display area on the screen. The data sets are to be scaled to fit the specified area, each plotted line is to be displayed in a different line type (solid, dashed, dotted), and the axes are to be labeled. (Instead of changing the line type, the three data sets can be plotted in different colors.)

4-8. Set up an algorithm for displaying thick lines with either butt caps, round caps, or projecting square caps. These options can be provided in an option menu.

4-9. Devise an algorithm for displaying thick polylines with either a miter join, a round join, or a bevel join. These options can be provided in an option menu.

4-10. Implement pen and brush menu options for a line-drawing procedure, including at least two options: round and square shapes.

4-11. Modify a line-drawing algorithm so that the intensity of the output line is set according to its slope. That is, by adjusting pixel intensities according to the value of the slope, all lines are displayed with the same intensity per unit length.

4-12. Define and implement a function for controlling the line type (solid, dashed, dotted) of displayed ellipses.

4-13. Define and implement a function for setting the width of displayed ellipses.

4-14. Write a routine to display a bar graph in any specified screen area. Input is to include the data set, labeling for the coordinate axes, and the coordinates for the screen area. The data set is to be scaled to fit the designated screen area, and the bars are to be displayed in designated colors or patterns.

4-15. Write a procedure to display two data sets defined over the same x-coordinate range, with the data values scaled to fit a specified region of the display screen. The bars for one of the data sets are to be displaced horizontally to produce an overlapping bar pattern for easy comparison of the two sets of data. Use a different color or a different fill pattern for the two sets of bars.

4-16. Devise an algorithm for implementing a color lookup table and the `setColourRepresentation` operation.

4-17. Suppose you have a system with an 8-inch by 10-inch video screen that can display 100 pixels per inch. If a color lookup table with 64 positions is used with this system, what is the smallest possible size (in bytes) for the frame buffer?

4-18. Consider an RGB raster system that has a 512-by-512 frame buffer with a 20 bits per pixel and a color lookup table with 24 bits per pixel. (a) How many distinct gray levels can be displayed with this system? (b) How many distinct colors (including gray levels) can be displayed? (c) How many colors can be displayed at any one time? (d) What is the total memory size? (e) Explain two methods for reducing memory size while maintaining the same color capabilities.

4-19. Modify the scan-line algorithm to apply any specified rectangular fill pattern to a polygon interior, starting from a designated pattern position.

4-20. Write a procedure to fill the interior of a given ellipse with a specified pattern.

4-21. Write a procedure to implement the `setPatternRepresentation` function.

4-22. Define and implement a procedure for changing the size of an existing rectangular fill pattern.

4-23. Write a procedure to implement a soft-fill algorithm. Carefully define what the soft-fill algorithm is to accomplish and how colors are to be combined.

4-24. Devise an algorithm for adjusting the height and width of characters defined as rectangular grid patterns

4-25. Implement routines for setting the character up vector and the text path for controlling the display of character strings.

4-26. Write a program to align text as specified by input values for the alignment parameters.

4-27. Develop procedures for implementing the marker attribute functions.

4-28. Compare attribute-implementation procedures needed by systems that employ bundled attributes to those needed by systems using unbundled attributes.

4-29. Develop procedures for storing and accessing attributes in unbundled system attribute tables. The procedures are to be designed to store designated attribute values in the system tables, to pass attributes to the appropriate output routines, and to pass attributes to memory locations specified in inquiry commands.

4-30. Set up the same procedures described in the previous exercise for bundled system attribute tables.

4-31. Implement an antialiasing procedure by extending Bresenham's line algorithm to adjust pixel intensities in the vicinity of a line path.

4-32. Implement an antialiasing procedure for the midpoint line algorithm.

4-33. Develop an algorithm for antialiasing elliptical boundaries.

4-34. Modify the scan-line algorithm for area fill to incorporate antialiasing. Use coherence techniques to reduce calculations on successive scan lines

4-35. Write a program to implement the Pitteway–Watkinson antialiasing algorithm as a scan-line procedure to fill a polygon interior. Use the routine `setPixel (x, y, intensity)` to load the intensity value into the frame buffer at location (x, y).

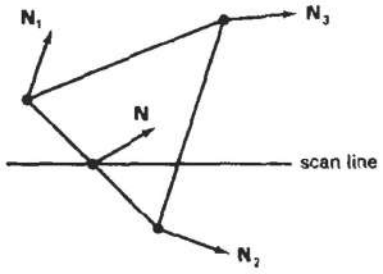


Figure 14-48
Interpolation of surface normals along a polygon edge

$$\mathbf{N} = \frac{y - y_2}{y_1 - y_2}\mathbf{N}_1 + \frac{y_1 - y}{y_1 - y_2}\mathbf{N}_2 \tag{14-42}$$

Incremental methods are used to evaluate normals between scan lines and along each individual scan line. At each pixel position along a scan line, the illumination model is applied to determine the surface intensity at that point.

Intensity calculations using an approximated normal vector at each point along the scan line produce more accurate results than the direct interpolation of intensities, as in Gouraud shading. The trade-off, however, is that Phong shading requires considerably more calculations.

Fast Phong Shading

Surface rendering with Phong shading can be speeded up by using approximations in the illumination-model calculations of normal vectors. **Fast Phong shading** approximates the intensity calculations using a Taylor-series expansion and triangular surface patches.

Since Phong shading interpolates normal vectors from vertex normals, we can express the surface normal **N** at any point (x, y) over a triangle as

$$\mathbf{N} = \mathbf{A}x + \mathbf{B}y + \mathbf{C} \tag{14-43}$$

where vectors **A**, **B**, and **C** are determined from the three vertex equations:

$$\mathbf{N}_k = \mathbf{A}x_k + \mathbf{B}y_k + \mathbf{C}, \qquad k = 1, 2, 3 \tag{14-44}$$

with (x_k, y_k) denoting a vertex position.

Omitting the reflectivity and attenuation parameters, we can write the calculation for light-source diffuse reflection from a surface point (x, y) as

$$\begin{aligned} I_{\text{diff}}(x, y) &= \frac{\mathbf{L} \cdot \mathbf{N}}{|\mathbf{L}|\,|\mathbf{N}|} \\ &= \frac{\mathbf{L} \cdot (\mathbf{A}x + \mathbf{B}y + \mathbf{C})}{|\mathbf{L}|\,|\mathbf{A}x + \mathbf{B}y + \mathbf{C}|} \\ &= \frac{(\mathbf{L} \cdot \mathbf{A})x + (\mathbf{L} \cdot \mathbf{B})y + \mathbf{L} \cdot \mathbf{C}}{|\mathbf{L}|\,|\mathbf{A}x + \mathbf{B}y + \mathbf{C}|} \end{aligned} \tag{14-45}$$