Chapter 1

A Survey of Computer Graphics

The purpose of this chapter is to acquaint students with the extent of computer-graphics applications and to introduce briefly some of the terminology, concepts, and methods employed in computer graphics. In some courses, it may be appropriate to assign initial projects that survey a particular graphics application area or a group of related areas.

Chapter 2

Overview of Graphics Systems

Exercise questions for this chapter investigate the characteristics and applications of computer-graphics hardware components, the general computer-graphics software concepts, and features of the OpenGL libraries.

Exercises

2-1. Operating characteristics for display technologies are discussed in Sections 2-1 through 2-3 and in the references cited at the end of the chapter.

2-2. Applications appropriate for the various display technologies are discussed in Sections 2-1 through 2-3 and in the listed references.

2-3. A system manual can be consulted to obtain screen dimensions and the number of pixel positions that can be displayed in the x and y directions. If x and y screen dimensions are not listed, they can be measured. The ratio of pixel positions to screen dimensions gives the resolutions for the horizontal and vertical directions. Aspect ratio is the ratio of the y and x resolutions.

Alternatively, resolution can be determined by displaying equal-pixel, straight-line segments in the x and y directions and measuring the length of each line.

For example, if 100-pixel lines in the x and y directions measure 4 cm and 5 cm, respectively, then

$$\text{x resolution} = \frac{100}{4} \text{ pixels per cm}$$

$$\text{y resolution} = \frac{100}{5} \text{ pixels per cm}$$

and

$$\text{aspect ratio} = \frac{\text{y resolution}}{\text{x resolution}} = \frac{4}{5}$$

To adjust dimensions of objects to maintain relative proportions, we can make the adjustments either to the vertical or to the horizontal dimensions. That is, vertical dimensions can be scaled by a factor of 4/5, or horizontal dimensions can be scaled by a factor of 5/4.

2-4. Frame-buffer size for each of the systems is

$$640 \times 480 \times 12 \text{ bits} \div 8 \text{ bits per byte} = 450 \text{ KB}$$

$$1280 \times 1024 \times 12 \text{ bits} \div 8 \text{ bits per byte} = 1920 \text{ KB}$$

$$2560 \times 2048 \times 12 \text{ bits} \div 8 \text{ bits per byte} = 7680 \text{ KB}$$

For 24 bits of storage per pixel, each of the above values is doubled.

2-5. Storage needed for the frame buffer is

$$800 \times 1000 \times 6 \text{ bits} \div 8 \text{ bits per byte} \approx 486 \text{ KB}$$

2-6. The times to load the two frame buffers are

$$640 \times 480 \times 12 \text{ bits} \div 10^5 \text{ bits per sec} \approx 36.9 \text{ sec}$$

and

$$1280 \times 1024 \times 24 \text{ bits} \div 10^5 \text{ bits per sec} \approx 314.6 \text{ sec} \approx 5.24 \text{ min}$$

2-7. Total bits in the printer frame buffer is

$$8.5 \times 11 \times 300^2 \approx 8.4 \times 10^6 \text{ bits}$$

Therefore, loading time is

$$\frac{8.4 \times 10^6 \text{ bits}}{32 \times 10^6 \text{ bps}} \approx 0.263 \text{ sec}$$

2-8. For the 640 by 480 system, the access rate is

$$640 \times 480 \times 60 = 1.8432 \times 10^7 \text{ pixels per sec}$$

Thus, access time is approximately 54 nanoseconds per pixel.

Similarly, for the 1280 by 1024 system, access rate is

$$1280 \times 1024 \times 60 = 7.86432 \times 10^7 \text{ pixels per sec}$$

Here, the access time is approximately 12.7 nanoseconds per pixel.

2-9. The diameter of screen pixels is

$$\frac{12}{1280} = \frac{9.6}{1024} = 9.375 \times 10^{-3} \text{ inches}$$

2-10. The scan rate for each pixel row is

$$60 \text{ frames/sec} \times 1024 \text{ lines/frame} = 61,440 \text{ lines/sec}$$

And the scan time is approximately 16.3 microseconds per scan line. (Scan time per frame is 1/60 sec, or approximately 16.7 milliseconds.)

2-11. Refresh time per frame is $1/r$ seconds, and the total retrace time during refresh of each frame is

$$t_{retrace} = t_{vert} + m \cdot t_{horiz}$$

The fraction of the time spent in retrace is $r \cdot t_{retrace}$.

2-12. The fraction of the total refresh time per frame spent in retrace is

$$60 \times (500 \times 10^{-6} + 1024 \times 5 \times 10^{-6}) \approx 0.337$$

2-13. Total number of available colors is $2^{24} = 16,777,216$. Using a different color for each screen pixel, we could display $512^2 = 262,144$ colors at any one time.

2-14. Three-dimensional monitors and stereographic systems are discussed in Sections 2-1 and in the references.

2-15. Input and output devices used with virtual-reality systems are discussed in Sections 2-4 and 2-5.

2-16. Design applications for VR include interactively constructing systems and interactively studying system operating characteristics. Some VR design applications are given in Sections 1-2, 1-3, 2-1, 2-4, and in the references at the end of Chapters 1 and 2.

2-17. Some applications for large-screen displays are presented in Section 2-3.

2-18. Graphics software packages are discussed in Section 2-8. Basically, packages for graphics programming contain functions for specifying individual geometric components (primitives, such as lines and circles), attributes of individual primitives, and parameters for various graphics operations such as viewing and geometric transformations. A package for a specific application, such as architectural design, allows a user to create scenes in terms of architectural features, such as doors, windows, hallways, stairs, and rooms. Then a building can be rotated or viewed from a given direction, such as from the front, from above, or from the left side.

2-19. The OpenGL core library contains hardware-independent functions, such as those for specifying primitives, attributes, geometric transformations, and three-dimensional viewing parameters. The GLU library contains functions for some other, more specialized operations, such as quadric-surface generation, B-spline surface generation, surface texture mapping, two-dimensional viewing, and some three-dimensional viewing operations. The GLUT library primarily provides hardware-dependent functions, such as those for display-window management and for interacting with input devices, but it also contains functions for generating various plane-surface, quadric-surface, and cubic-surface solids, such as a cube, sphere, cone, or teapot.

2-20. We set the color of a display window to light gray by selecting a value between 0.5 and 1.0 and assigning that value to all three RGB parameters in the `glClearColor` command. For example, we can set the display window to light gray with

```
glClearColor (0.9, 0.9, 0.9, 0.0);
```

Similarly, we obtain dark gray by selecting a value between 0.0 and 0.5 for the RGB color components.

2-21. The following statements set the lower-right corner of a 100 pixel by 75 pixel display window at screen coordinates (200, 200):

```
glutInitWindowPosition (100, 125);
glutInitWindowSize (100, 75);
```

2-22. A callback function is a procedure that is to be invoked whenever a particular action occurs. Such a procedure is registered by listing the procedure name in an appropriate OpenGL function.

Chapter 3

Graphics Output Primitives

In this chapter, the exercise questions explore raster algorithms for generating basic picture components, such as straight-line segments, curves, and fill areas, as well as the OpenGL primitive functions.

Exercises

3-1. Define a routine that accepts positive integer input values for n and coordinate positions $(x(k), y(k))$, with $k = 1, 2, \ldots, n$.

If $n = 1$, the routine plots the pixel at position $(x(1), y(1))$. This can be implemented using the `setPixel` command, which calls the OpenGL point-plotting functions.

If $n \geq 2$, the routine makes $n1$ calls to `procedure dda` to plot line segments between $(x(k), y(k))$ and $(x(k+1), y(k+1))$, for $k = 1, 2, \ldots, n-1$.

3-2. Given the endpoint coordinates for a line segment, determine which is the left endpoint and denote it as $(x0, y0)$. Label the right endpoint as $(xend, yend)$, where $xend \geq x0$.

Then set $\Delta x = xend - x0$ and $\Delta y = yend - y0$. A positive value for Δy indicates a positive slope ($m > 0$), and a negative value for Δy indicates a negative slope ($m < 0$). In all cases, Δx is nonnegative (either positive or 0).

For vertical lines ($\Delta x = 0$), horizontal lines ($\Delta y = 0$), and diagonal lines ($|\Delta y| = |\Delta x|$), pixel positions can be plotted without invoking the line-drawing algorithm.

Otherwise, set

$$sign(m) = \begin{cases} +1 & \text{if } \Delta y > 0 \\ -1 & \text{if } \Delta y < 0 \end{cases}$$

and plot the pixels along the line segment according to the following algorithm.

Case (1) $|\Delta y| < |\Delta x|$ $(|m| < 1)$:

Calculate $p_0 = 2\, sign(m)\, \Delta y - \Delta x$

Take positive unit steps in x from $x0$ to $xend$. Incremental steps for y are equal to $sign(m)$. Thus, if $m > 0$, y values are also increasing, and if $m < 0$, y values are decreasing.

At each step:

If $p_k < 0$, then $y_{k+1} = y_k$ and $p_{k+1} = p_k + 2\,sign(m)\Delta y$,
else $y_{k+1} = y_k + sign(m)$ and $p_{k+1} = p_k + 2\,sign(m)\,\Delta y - 2\,\Delta x$.

Case (2) $|\Delta y| > |\Delta x|$ $(|m| > 1)$:

Calculate $p_0 = 2\,\Delta x - sign(m)\,\Delta y$.

Take unit steps in y from $y0$ to $yend$, where each incremental step is equal to $sign(m)$. Thus, y values are increasing if $m > 0$, and y values are decreasing if $m < 0$.

At each step:

If $p_k < 0$, then $x_{k+1} = x_k$ and $p_{k+1} = p_k + 2\,\Delta x$,
else $x_{k+1} = x_k + 1$ and $p_{k+1} = p_k + 2\,\Delta x - 2\,sign(m)\,\Delta y$.

3-3. Define a routine that accepts positive integer input values for n and coordinate positions $(x(k),\, y(k))$, with $k = 1, 2, \ldots, n$.

If $n = 1$, the routine plots the pixel at position $(x(1),\, y(1))$. This can be implemented using the `setPixel` command, which calls the OpenGL point-plotting functions.

If $n \geq 2$, the routine makes $n1$ calls to the Bresenham algorithm in the previous algorithm to plot line segments between $(x(k),\, y(k))$ and $(x(k+1),\, y(k+1))$, for $k = 1, 2, \ldots, n-1$.

3-4. In the midpoint line-drawing algorithm for a line with slope between 0 and 1, we can define a line function as

$$f(x, y) = x\Delta y + b\Delta x - y\Delta x$$

where parameter b is the y intercept for the line, and Δx and Δy are the positive coordinate extents for the line in the x and y directions.

At any position $(x,\, y)$, we then have the following conditions:

$$f_{line}(x, y) \begin{cases} = 0 & \text{if } (x, y) \text{ is on the line path} \\ < 0 & \text{if } (x, y) \text{ is above the line path} \\ > 0 & \text{if } (x, y) \text{ is below the line path} \end{cases}$$

And the decision parameter for the line is

$$p_k = f_{line}(x_k + 1,\; y_k + 0.5)$$

or

$$p_k = x_k\Delta y - y_k\Delta x + const$$

which has the initial value

$$p_0 = \Delta y - \frac{1}{2}\Delta x$$

Also, if $p_k < 0$, then $p_{k+1} = p_k + \Delta y$. Otherwise, $p_{k+1} = \Delta y - \Delta x$.

But $2p_k$ has the same sign as p_k, so we can define the decision parameter to be 2 times the line function, which produces the same decision conditions as in Bresenham's algorithm.

3-5. Following the method in the previous exercise, we obtain the same decision parameters and algorithm as in Exercise 3-2.

3-6. A parallel version of Bresenham's line algorithm can be implemented by dividing the line into a number of segments and applying the algorithm to each segment (Section 3-6).

3-7. The method of the previous exercise can be applied to the procedures developed for Exercise 3-2.

3-8. The number of pixels that can be displayed for this system is 800 by 1000. Pixel positions are numbered horizontally from 0 to 799 and vertically from 0 to 999. Then,

$$addr(x,\, y) = 800\, y + x$$

where the step from one pixel position to the next is one byte.

3-9. The frame buffer address calculation here is the same as in the previous exercise, but with the step size modified to be in increments of 6 bits.

3-10. Frame buffer addresses for a line with positive slope are calculated using Equations 3-24 and 3-25. For a line with negative slope, pixel locations are calculated as

$$addr(x,\, y-1) = addr(x,\, y) - (x_{max} + 1)$$

or

$$addr(x+1,\, y-1) = addr(x,\, y) - x_{max}$$

3-11. Geometric magnitudes can be maintained for circles by lowering pixel positions one unit across the top of the circle and by shifting pixel positions to the left one unit on the right side of the circle. The appropriate adjustments are shown in Figure 3-39.

3-12. One method for parallelization of a circle-generation algorithm is to subdivide one octant and assign each subarc to a separate processor, as discussed in Section 3-12.

3-13. Decision parameters are obtained by interchanging x and y coordinates. Thus, for region 1:

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

At any step, if $p1_k < 0$, the next decision parameter is calculated as $p1_{k+1} = p1_k + 2\, r_x^2\, y_{k+1} + r_x^2$. Otherwise, the term $-2\, r_y^2\, x_{k+1}$ is added to this calculation.

Similarly for region 2. The decision parameter here is calculated as $p2_{k+1} = p2_k - 2\,r_y^2\,x_{k+1} + r_y^2$ when $p2_k < 0$. Otherwise, the term $2\,r_x^2\,y_{k+1}$ is added to the calculation for $p2_{k+1}$.

3-14. As in Exercises 3-6 and 3-12, a parallel ellipse-generation algorithm can be set up by subdividing the first quadrant into a number of subintervals.

3-15. The general equation for a sine curve is

$$y = A\,sin(\omega\,x\,+\,\theta)$$

where parameter A specifies the amplitude, ω is the angular frequency $(2\,\pi\,f)$, and θ is the phase angle. Parameters A and ω are adjusted to position the curve within the bounds of the display region.

Since the sine function is periodic with period $2\,\pi$, only points within one cycle need be calculated. Moreover, by symmetry, points within one quadrant of a cycle can be used to obtain points in the other three quadrants of the cycle. For example, for each x value specified in the interval from $-\theta/\omega$ to $(\pi/2-\theta)/\omega$, we can plot four points: $(x,\,y)$, $(\pi - x,\,y)$, $(3\pi/2 - x,\,-y)$, and $(2\pi - x,\,-y)$.

Calculated points along the curve path can be joined with straight line segments, or more accurate methods can be used to plot adjacent points along the curve. Unit steps in the x direction will produce adjacent pixel positions when the slope has a magnitude less than or equal to 1. For larger magnitudes of the slope, unit steps in the y direction should be used to avoid gaps between calculated pixel positions.

3-16. We can first determine the number of complete cycles over the range of the curve, and the methods of the previous exercise can be applied to obtain points within each of these complete cycles. Next we can plot points within any complete quadrants at the beginning or end of the curve. Any fractions of a quadrant at the beginning or end of the curve can then be plotted directly from the curve equation.

3-17. Assuming that the phase angle is less than 360°, we can plot the curve for an integral number of cycles plus the fraction of a cycle at the beginning of the curve. The last cycle at the end of the curve will have a maximum amplitude of $A/10$. Or the last cycle could be plotted as a fraction of a cycle, ending at the value of x that yields $e^{-kx} = 0.1$. To obtain points along the path of the damped sine curve, we first apply the methods described in the previous exercise to obtain points along the path of a sine curve. Then, each point is scaled by the product of the amplitude A and the exponential function.

3-18. Midpoint methods can be applied as in the circle or ellipse algorithms by defining the following curve function,

$$f(x,\,y) = \frac{1}{12}x^3\,-\,y$$

The decision parameter is then the preceding curve function evaluated at midpoint positions. To determine pixel positions along the curve path, we use methods similar to those for generating an ellipse. Curve positions are determined through two regions in the first quadrant of the xy plane, starting at position (0, 0). Between $x = 0$ and $x = 2$ (region 1), the slope of the curve is less than 1. In region 2, where $x > 2$, the slope of the curve is greater than 1.

Since this function is symmetric about the origin, only points in the first quadrant of the xy plane need be calculated. Thus for any position (x, y) on the curve in the first quadrant, there is the corresponding curve position $(-x, -y)$ in the third quadrant.

3-19. As in the previous exercise, we define the curve function and decision parameter as

$$f(x, y) = 100 - x^2 - y$$

which is evaluated at the halfway position between the two candidate pixels at each step. This function is symmetric about the y axis. Thus, we need only process one region in the first quadrant of the xy plane, because the slope of the curve has magnitude greater than 1 for all values of x greater than 1/2 or less than -1/2.

3-20. The curve function can be defined as

$$f(x, y) = y^2 - x$$

and the methods similar to those in the previous exercise can be applied. In this case, the parabola is symmetric about the x axis.

3-21. This function is simply a generalization of the parabola in the previous two exercises. The decision parameter (curve function) can be defined as

$$f(x, y) = a\,x^2 + b - y$$

with the symmetry axis and curve slope determined by parameters a and b.

3-22. The vertex table lists the labels and coordinate positions for the eight vertices of the cube, from (0, 0, 0) to (1, 1, 1). The edge table lists the labels for the twelve edges, with references back to the vertex table for the two endpoint coordinate positions of each edge. The surface-facet table lists the labels for the six faces, with references back to the edge table for the four edges of each surface facet.

3-23. When the geometric data tables for a unit cube contain just a vertex table and a surface-facet table, we store the labels and coordinates for the eight vertices in the vertex table, and we store references to the four vertices for each of the six labeled cube faces in the surface-facet table.

With one table, we store a label for each of the six cube faces along with the coordinate positions of the four vertices for each face.

A single table requires more storage (72 coordinate values), since redundant coordinate information is stored. Using two tables does require pointers back to the vertex table,

but this takes less storage that using a single table since only 24 coordinate values are stored.

The major disadvantages of storing information in one table are (1) there is no explicit information regarding shared vertices and edges and (2) each shared edge is drawn twice when the object is displayed.

Representing polygons with two tables requires the least amount of storage. But this representation has the same drawbacks listed above for a single-table representation.

Three tables provide the most information for identifying shared edges and for consistency checks.

3-24. As shown in Fig. 3-41, each end of the cylinder is represented with a polygon approximation of a circular area and the side is approximated with a set of rectangular strips. The advantage of a polygon approximation is that all objects are described with linear equations, thus reducing processing time for surface rendering.

3-25. Labels are assigned to each vertex, each edge, and each polygon surface facet. The labels and coordinate positions for the vertices are then entered into the vertex table; the edge labels, along with the two vertex references for each edge, are entered into the edge table; and the facet labels, along with the edge references for each facet, are entered into the surface-facet table. Consistency checks can be made to ensure that each edge has two coordinate references, each facet has the correct number of edge references (three edges for a triangle, four edges for a quadrilateral, and so forth), and the polygon facets form a closed surface around the object interior. Also, the tables should be checked to ensure that no information is duplicated, all vertices are edge endpoints, and all edges appear in two surface facets.

3-26. The various data-table checks are discussed in the previous exercise and in Section 3-15.

3-27. Expressions for the plane parameters are given in Eqs. 3-62. For each convex polygon facet of the object surface, the coordinates for three vertex positions can be used in these equations, taking the vertices in counterclockwise order for a right-handed coordinate system.

3-28. For an input coordinate position (x, y, z), evaluate the expression $A\,x + B\,y + C\,z$. If this expression is positive, the input position is in front of the polygon surface. If the expression is negative, the input position is behind the polygon surface. This test assumes that the plane parameters were evaluated from three coordinate positions in the plane of the polygon, taken in a counterclockwise order in a right-handed Cartesian reference when viewing the front of the plane.

3-29. In a left-handed Cartesian reference system, vertices are selected in clockwise order to obtain the correct description for a plane surface. For three counterclockwise coordinate positions, $\mathbf{P}_1$, $\mathbf{P}_2$, $\mathbf{P}_3$, in a plane surface, we accomplish this by selecting the three points in the reverse order. Thus, we interchange the coordinate values for $\mathbf{P}_1$ and $\mathbf{P}_3$ in Eqs. 3-62, which changes the sign of the plane parameters A, B, C, and D.

3-30. Using the coordinates of the first three points in the vertex list, calculate the plane parameters A, B, C, and D. If the coordinates for the fourth vertex satisfy Eq. 3-59 (the plane equation), that vertex lies in the plane of the first three vertices. Otherwise, the vertex list specifies a nonplanar object.

3-31. Using the coordinates of the first three points in the vertex list, calculate the plane parameters A, B, C, and D. If the coordinates for any subsequent position in the vertex list do not satisfy the plane equation (Eq. 3-59), the vertex list specifies a nonplanar object.

3-32. A procedure for splitting a polygon into a set of triangles is given in Section 3-15. For a quadrilateral with vertices labeled as 1, 2, 3, 4, the first triangle vertex list is {1, 2, 3}. This leaves just three vertices in the original list, so the final triangle vertex list is {1, 3, 4}.

3-33. A procedure for splitting a polygon into a set of triangles is given in Section 3-15. For a list of vertices labeled as 1, 2, 3, ..., n, the vertex lists for the $n - 2$ triangles are {1, 2, 3}, {1, 3, 4}, {1, 4, 5}, ..., {1, $n - 1$, n}.

3-34. A repeated vertex is detected by checking the coordinates of that vertex against the coordinates for every other point in the vertex list. This can be accomplished in a loop that successively checks each point in the list against the other points.

Collinear points are detected by checking the slopes of adjacent edges. For a polygon in the xy plane, compare the ratios $\Delta y/\Delta x$ of each pair of successive edges. If the plane of the polygon is not the xy plane, the polygon plane could be transformed to the xy plane by transforming the normal vector to the z axis, or three-dimensional slopes could be compared.

3-35. One method for locating the intersection position for two line segments is to equate y values using the line representation Eq. 3-1, providing neither line is vertical. If the resulting x value is outside the range of either line, the line segments do not intersect.

A general, and more efficient, approach is to equate the parametric representations (Eq. 3-58) for the two lines. This method can be used to intersection coordinates for two lines of any slope.

A parametric representation for a line segment in the xy plane can be written as

$$x(u) = x_0 + (x_{end} - x_0)u, \qquad y(u) = y_0 + (y_{end} - y_0)u$$

or

$$x(u) = x_0 + \Delta x\, u, \qquad y(u) = y_0 + \Delta y\, u$$

where parameter u varies from 0.0 to 1.0. When $u = 0.0$, we have the coordinate position for the first endpoint (x_0, y_0). At the other end of the line, $u = 1.0$ and the endpoint coordinates are (x_{end}, y_{end}). Equating the two parametric equations for each line yields two equations in two unknowns: the parameters u_1 and u_2 for the two lines. If these

parameters are in the range from 0.0 to 1.0, the two lines intersect and the intersection coordinates can be computed from either set of parametric equations.

The solutions for the two line parameters are

$$u_1 = \frac{\Delta x_2\,\Delta y_0 \;-\; \Delta y_2\,\Delta x_0}{\Delta x_2\,\Delta y_1 \;-\; \Delta x_1\,\Delta y_2}$$

and

$$u_2 = \frac{\Delta x_1\,\Delta y_0 \;-\; \Delta y_1\,\Delta x_0}{\Delta x_2\,\Delta y_1 \;-\; \Delta x_1\,\Delta y_2}$$

where $\Delta x_0 = x_{02} - x_{01}$ and $\Delta y_0 = y_{02} - y_{01}$. If the denominator of the expression for u_1 (which is the same as the denominator for parameter u_2) is 0, the two lines have the same slope and the two line segments are parallel (and could be collinear).

The value for either u_1 or u_2 can be computed and used in the corresponding set of parametric equations to locate the intersection point. But before performing the division calculation, the numeric values of the numerator and denominator can be checked to determine whether parameter u_1 (or parameter u_2) is outside the interval from 0.0 to 1.0. There is no intersection if $u_1 < 0.0$ or $u_1 > 1.0$. The conditions for no intersection are:

$u_1 < 0.0$, if the numerator and denominator have opposite signs.

$u_1 > 1.0$, if the numerator and denominator have the same sign and the absolute value of the numerator is greater than the absolute value of the denominator.

3-36. An algorithm for identifying a concave polygon is discussed in Section 3-15 and illustrated in Fig. 3-43. Given a set of vertices, first determine the Cartesian components for the edge vectors of the polygon. Then calculate the vector cross product for each pair of adjoining edges, in a counterclockwise order around the polygon perimeter. Check the z components of the cross products: if some are positive and some are negative, the polygon is concave.

3-37. An algorithm for splitting a concave polygon using the vector method is given in Section 3-15 and Fig. 3-44. Form the edge vectors and calculate vector cross products for successive pairs of edges, traversing the edges in a counterclockwise order. If the z components of some cross products are positive, while others are negative, the polygon is concave.

When a negative cross product is encountered for two edges, $\mathbf{E}_k$ and $\mathbf{E}_{k+1}$, extend the line of the first edge ($\mathbf{E}_k$) from its second endpoint to the first intersection with another polygon edge. (See Exercise 3-35 for line intersection calculations.) Form a new polygon with one edge that is the extension of $\mathbf{E}_k$ to the intersection point, another edge that is $\mathbf{E}_{k+1}$, and the remaining edges of the original polygon that join the second endpoint of $\mathbf{E}_{k+1}$ to the calculated intersection point (Fig. 3-44). Then revise the original polygon vertex list so that: (1) $\mathbf{E}_k$ is extended from its first endpoint to the calculated intersection point, and (2) the intersected edge is shortened so that its starting endpoint is the calculated intersection position. Finally, repeat the algorithm for each set of newly formed polygons until all edge cross products are positive.

3-38. An algorithm for splitting a concave polygon using the rotational method is outlined in Section 3-15 and illustrated in Fig. 3-45. The basic steps are: (1) move the polygon so that the first vertex is at the coordinate origin, (2) rotate the polygon so that its first edge is on the x axis, (3) if the second vertex of the polygon is below the x axis, split the polygon along the x axis into two or more separate polygons, (4) repeat the preceding steps for all vertices in order, then repeat for all new polygons that have been formed.

For a polygon in the xy plane, label the vertices as $\mathbf{V}_k = (x_k, y_k)$, with $k = 1, 2, \ldots, n$. To shift the position of the polygon so that its first vertex is at the coordinate origin, subtract x_1 from all vertex x values and subtract y_1 from all vertex y values. The shifted coordinates for $\mathbf{V}_1$ are now (0, 0).

If the shifted position of $\mathbf{V}_2$ is on the x axis (its new y coordinate is 0), shift the polygon again so that $\mathbf{V}_2$ is now at the coordinate origin and check the position of $\mathbf{V}_3$. Assuming the polygon is not degenerate (collinear edges), $\mathbf{V}_3$ must be either above or below the x axis.

If the shifted position of $\mathbf{V}_2$ is not on the x axis, we next rotate the first edge of the polygon onto the x axis. The angle between this edge and the x axis is

$$\theta = \tan^{-1}\left(\frac{y_2}{x_2}\right)$$

where (x_2, y_2) are the coordinates for the shifted position of $\mathbf{V}_2$. If $y_2 > 0$, the polygon is rotated by applying the following transformation calculations to all vertices.

$$\begin{aligned} x_k' &= x_k \cos\theta + y_k \sin\theta \\ y_k' &= -x_k \sin\theta + y_k \cos\theta \end{aligned}$$

This brings the first edge of the polygon (from vertex $\mathbf{V}_1$ to vertex $\mathbf{V}_2$) onto the x axis. If $y_2 < 0$, angle θ is negative, so we use the conjugate angle $\theta = 2\pi - |\theta|$ in the preceding transformation calculations for the polygon vertices.

Next, if the rotated y_3 value is positive, repeat the preceding steps by shifting the polygon so that $\mathbf{V}_2$ is now at the origin and rotating the second edge (from $\mathbf{V}_2$ to $\mathbf{V}_3$) onto the x axis. But if vertex $\mathbf{V}_3$ is below the x axis, split the polygon along the line of the x axis. This is accomplished by checking the polygon edges for intersections with the line $y = 0$. Then form a new vertex list for each separate section of the polygon that is below the x axis, and also revise the original vertex list for the polygon.

After all vertices in the original polygon have been processed, repeat these procedures for each polygon that was cut from the original polygon. By counting the number of edge cross products that are negative in the original polygon, we can predetermine the number of times that we need to split the concave polygon to form a set of convex polygons.

3-39. Vector cross-product calculations for determining the winding number are discussed in Section 3-15. For any specified position in the xy plane, determine the components of a vector $\mathbf{u}$ along a reference line from that point to a position beyond the coordinate extents of the vertex list, initialize the winding-number parameter to 0, and calculate

the components of the edge vectors $\mathbf{E}_k$ formed by successive pairs of vertices in the list. The reference line must not intersect any coordinate position in the vertex list. Coordinate extents of the edges and the reference line can then be used to perform initial tests to identify those lines that are outside the coordinate extents of the reference line. Intersection calculations are then performed for the remaining lines (see Exercise 3-35). For each edge that crosses the reference line, calculate the vector cross product $\mathbf{u} \times \mathbf{E}_k$. If the z component of the cross product is positive, add 1 to the winding number. If the z component of the cross product is negative, subtract 1 from the winding number. A nonzero value for the winding number indicates an interior point. Otherwise, the specified position is an exterior point.

3-40. Vector dot-product calculations for determining the winding number are discussed in Section 3-15. For any specified position in the xy plane, determine the components of a vector $\mathbf{u} = (u_x, u_y)$ along a reference line from that point to a position beyond the coordinate extents of the vertex list, set the components for the perpendicular vector $(-u_y, u_x)$, initialize the winding-number parameter to 0, and calculate the components of the edge vectors formed by successive pairs of vertices in the list. The reference line must not intersect any coordinate position in the vertex list. Coordinate extents of the edges and the reference line can then be used to perform initial tests to identify those lines that are outside the coordinate extents of the reference line. Intersection calculations are then performed for the remaining lines (see Exercise 3-35). For each edge that crosses the reference line, calculate the vector dot product for that edge vector and the perpendicular reference-line vector. If this dot product is positive, add 1 to the winding number. If the dot product is negative, subtract 1 from the winding number. A nonzero value for the winding number indicates an interior point. Otherwise, the specified position is an exterior point.

3-41. The shaded triangular region with one vertex at position A in Fig. 3-46 has a positive winding number (+1). All other shaded regions in Fig. 3-46(b) have a negative winding number (either -1 or -2). No regions have a winding number greater than 1, but the four-sided polygonal area near vertices B and F has a winding number equal to -2.

3-42. A text-string function can be implemented using OpenGL routines. A world-coordinate reference frame can be specified using the methods described in Section 3-2. Text characters can be individually defined as bitmaps and stored in a font list, or a set of routines could be designed to generate character outlines using polyline or curve patterns. Alternatively, a GLUT character function (Section 3-21) could be used to generate characters within a selected font.

For characters specially defined with rectangular pixel grids, the lower left corner of the first character pattern can be placed at the input coordinate position. Successive characters in the string are then displayed using horizontal offsets equal to the character width, assuming all characters have the same width.

Similar methods are applied for specially designed outline-font routines. The input position can be used as the start position for the lower-left corner of the string. And, the

coordinate extents of the character patterns can be used to offset successive characters as they are displayed.

If the GLUT routines are used to display characters, positioning can be accomplished with the `glRasterPosition` function. Examples are given in Section 3-21 and in the first two programs in the Example Programs section at the end of Chapter 3.

3-43. A specified character or shape is to be displayed at a series of locations. As in the previous exercise, a marker symbol can be specially designed or a GLUT function can be used to display an available character. An example implementation is given in the first program in the Example Programs section at the end of Chapter 3.

3-44. A centered hexagon can be displayed by eliminating the reshape operations. To do this, delete the `glutReshapeFunc` command in procedure `main`, delete procedure `winReshapeFcn`, and put the commands `glMatrixMode (GL_PROJECTION)` and `gluOrtho2D (0.0, winWidth, 0.0, winHeight)` at the beginning of procedure `init`. Default reshape operations are then used, as in the Section 2-9 example program. Although this keeps the hexagon in the center of the display window, its size and aspect ratio change as the dimensions of the display window change.

Another method for maintaining a centered hexagon in the display window is to specify the polygon without using a display list and redefine the coordinate reference using the `glViewport (0, 0, newWidth, newHeight)` function (Section 6-4) in the `winReshapeFcn` routine. In this case, the size and aspect ratio of the hexagon do not change when the dimensions of the display window are changed.

3-45. In addition to the data values and the labeling information, the size of the display window can be specified as input, as well as the spacing between the bars. A procedure for generating the bars and labels is given in the Example Programs section at the end of Chapter 3, with bars generated using the OpenGL `glRect` function. Additional routines are needed for retrieving the input values (typically from a file), scaling the data values, drawing the x and y axes, and displaying the axes and graph labeling.

Data values are scaled so that the maximum data value is near the top of the display window and the minimum value is near the bottom of the display window, as in Fig. 3-68. For an input data value, the corresponding y position within the display window can be calculated as

$$y = yMin + \frac{yMax - yMin}{dataMax - dataMin}(dataValue - dataMin)$$

Similarly, the bars are spaced across the width of the display window so that a small margin is provided on both sides.

The x axis should be displayed near the bottom of the display window, and the y axis near the left edge. Axes labeling is placed under the x axis and to the left of the y axis. A label for the entire graph can be placed either at the top or the bottom of the display window.

3-46. This is a modification of the previous exercise. Additional input is needed to specify an area within the display window for the location of the bar graph, otherwise the processing is the same.

3-47. The line graph is displayed using methods similar to those outlined for Exercises 3-45 and 3-46. Instead of constructing rectangular bars, the scaled data points are to be displayed with a selected marker symbol and joined with straight-line segments. This exercise is an extension of the routines used to generate the sample data set in Fig. 3-67.

3-48. Routines for generating a simple, unlabeled pie chart are given in the Example Programs section at the end of Chapter 3, and a sample output is shown in Fig. 3-69. Additional routines are needed for retrieving the data values to be plotted (typically from a file) and for displaying the labeling.

The overall label for the pie chart can be placed either above or below the chart. Depending on the size of the display and the size of the character font, section labels could outside the pie chart or inside the individual sections. These labels can be positioned halfway between the section limits by determining the angular values corresponding to these halfway positions. If the labels are outside, coordinate positions just beyond the circumference at these angular values can then be used as starting positions for labels on the right half of the pie chart and as ending positions for labels on the left half of the pie chart.

As a further extension, an exploded pie chart could be displayed. This requires moving each slice of the chart radially out from the circle center. To accomplish this, move the endpoint positions of the two straight-line segments for each pie section to new positions along radial lines. Then a circle segment is generated, using the cusp of each pie section as the center position for the circle.

Chapter 4

Attributes of Output Primitives

Exercises in this chapter explore methods for displaying primitives with various attributes, which include color, line styles, fill styles, and character styles. In addition, the exercises investigate the use of the OpenGL functions for specifying attribute parameters.

Exercises

4-1. A file containing a list of *RGB* color values and the number of colors in the list can be provided as the input. The `glutSetColor` function is described in Section 4-3. This function stores the values for the *RGB* components of a color in a table at an assigned index location.

As an option, the colors could be stored in the color table in a specified order, such as sorting the colors according to the values of the red components. Another option is to create a two-dimensional scene and assign color-table values to the objects using the `glIndex` function.

4-2. Section 3-17 discusses the OpenGL vertex-array functions, and Section 4-3 introduces color arrays and gives an example of combining a color array with a vertex array for a single object. For this exercise, the example in Section 4-3 is extended to an array specification for a two-dimensional scene containing six objects. The objects can be simple polygons, such as rectangles and triangles, or just a set of six rectangles. Each object can be specified in a different color or color pattern, with the color values assigned to the object vertices. Alternatively, object type, color, and position could be fixed or provided in an input file.

4-3. Example programs for displaying two-dimensional objects are given at the end of Chapters 2 and 3. The scene description from the previous exercise can be substituted into one of these example programs.

4-4. Section 3-17 discusses the OpenGL vertex-array functions, and Section 4-3 introduces color arrays and gives an example of combining a color array with a vertex array for a single object. For this exercise, the example in Section 4-3 is extended to an array specification for a three-dimensional scene containing four objects. The objects could be

simple polyhedrons, such as boxes (rectangular parallelepipeds) and pyramids (regular tetrahedrons), or just a set of four cubes. Each object can be specified in a different color or color pattern, with the color values assigned to the object vertices. Alternatively, object type, color, and position could be fixed or provided in an input file.

4-5. A cloud photograph could be traced on a sheet of finely spaced grid paper, with the individual positions in the grid marked to indicate the brightness level of the regions. Or a cloud pattern could be sketched without tracing a photograph. Once the dark and light regions have been marked, scales for the point size and spacing can be set. The white parts of the cloud can be modeled with standard-size, closely spaced white pixels. Then the other regions are modeled with shades of gray, graduated pixel sizes, and graduated pixel spacing. For example, near-white regions could be modeled using a light-gray color, the standard-size pixel, and a four-pixel separation. And darker regions are modeled with darker gray colors, larger pixels, and a spacing that depends on the pixel size and shade of gray. Each scan line across the cloud can be encoded as a set of gray-scale regions, along with the length (number of pixels or points) in each region.

Because a point with size n is displayed as an n by n block of pixels, very large points appear as squares. To avoid the square appearance of points in the cloud pattern, either set the maximum point size at 3.0 or 4.0 or activate antialiasing.

The cloud pixels are to be plotted in a display window that has a sky background. Shades of blue can be generated by setting the blue component of an RGB color to 1.0 and varying the value for the green component, with a setting at or near 0.0 for the red component. Gray-scale colors are generated with equal values for R, G, and B between 0.0 and 1.0. The OpenGL color functions are discussed in Section 4-3, and point size is set with the `glPointSize` function (Section 4-7).

Modeling the cloud with points of varying sizes and point spacing simulates the halftone printing process used in newspapers and magazines. Figure 10-37 shows a black-and-white halftoning example, and Fig. 10-38 shows a color halftoning example.

4-6. This is a color modification of the previous exercise.

4-7. Various line styles can be generated by including parameters within the line-drawing routine to control the number of successive pixels to be displayed and the number of successive points to be skipped along the line path. For example skipping three points between every plotted point produces a dotted line, while skipping four pixels between every sequence of six plotted pixels produces a dashed line. An input parameter can be used to specify the line style, using a single pattern for either a dashed or dotted line. As an alternative, an additional parameter could be used to specify dot spacing, and other input parameters could be used to specify dash length and spacing. Another option is the display of lines with a dash-dot pattern.

4-8. The decision parameter for a midpoint line algorithm is given in the solutions for Exercises 3-4 and 3-5, and the methods for displaying line styles using this algorithm are the same as in Exercise 4-7.

4-9. Parallel line-generation procedures are outlined in the solutions for Exercises 3-6 and 3-7, and the methods for incorporating line-style parameters into these procedures are the same as in Exercise 4-7.

4-10. Line-width algorithms are discussed in Section 4-5 and in the solution for Exercise 4-12. Parallel line-generation procedures are outlined in the solutions for Exercises 3-6 and 3-7.

4-11. The rectangle can be formed with one edge as the specified line segment, one edge parallel to this line, and two edges that are perpendicular to the given line segment. Thus, two vertices for the rectangle are the two specified line endpoints, and the other two vertices are perpendicular offsets from these endpoints. For a horizontal or vertical line, no computations are needed to obtain the two offset vertex positions. Otherwise, the x and y offset values for these vertices are calculated using the line segment parameters and the input value for the line width parameter lw.

Since all interior angles of a rectangle are 90°, each edge has a slope that is the negative inverse of an adjoining edge. Therefore, the x and y offset distances can be calculated by comparing similar triangles. Given the coordinates for the line endpoints, calculate the line width (Δx), the line height (Δy), and the line length (L). The x offset distance is then computed as $lw(\Delta y/L)$, and the y offset distance is $lw(\Delta x/L)$.

For a line with positive slope, the second pair of rectangle vertices can be calculated using either a positive x offset and a negative y offset, or a negative x offset and a positive y offset. For a line with negative slope, the second pair of rectangle vertices can be calculated using either a positive x offset and a positive y offset, or a negative x offset and a negative y offset.

Instead of using the specified line segment as a rectangle edge, we can form the rectangle around the line segment as the centerline for the rectangle. In that case, the perpendicular offset distances to the four rectangle vertices from the line endpoints are $\pm lw/2$.

4-12. Algorithms for generating wide lines are discussed in Section 4-5. First, Bresenham's line algorithm can be used to determine pixel positions along the line path from the first endpoint to the second endpoint. If the magnitude of the line slope is less than 1.0, a vertical column of pixels can then be plotted from each calculated pixel position. Otherwise, a row of pixels can be plotted from each calculated pixel position.

If the input line width is assumed to specify a perpendicular distance to the line path, the number of pixels to plot can be calculated from the input line width lw as either

$$n_{vert} = \frac{lw}{|cos\theta|}$$

or

$$n_{horiz} = \frac{lw}{|sin\theta|}$$

where $\tan\theta$ is the slope of the line.

This procedure is straightforward, but it displays a parallellogram instead of a rectangle. To display a wide line as a rectangle, the pixel-plotting procedure can be modified at

the starting and ending positions of the line. This is accomplished by calculating the coordinates for the corners of the rectangular area of a line with a given perpendicular width lw (see Exercise 4-11).

An alternative procedure is to use an area filling method to fill the interior of the rectangle, once the four vertices have been determined.

4-13. This exercise can be implemented using a modified Bresenham algorithm (Exercise 4-7). Alternatively, the example OpenGL program in Section 4-8 can be modified to display the graphs in different line styles only. In either case, data sets and the corresponding labeling can be provided in data files. Also, the line styles for each data set could be an assigned parameter. Methods for scaling the data sets to fit within a specified region are discussed in the solution to Exercise 3-45.

4-14. Color values for this exercise could be specified along with the other input data or left as an option.

4-15. Methods for displaying the three line caps are given in Section 4-5. Input parameters to this algorithm should include the line endpoints, the line width, and the type of line cap to be used.

4-16. The three methods for joining two or more straight-line segments are discussed in Section 4-5. The algorithm in the preceding exercise is to be modified to display a polyline with a selected connection shape between each line pair.

4-17. Arguments for the procedure will now include a line-width parameter along with parameters for the data values. The program should then output a single line plot using the specified line width.

4-18. Arguments for the procedure will now include a line-style parameter along with parameters for the data values. The program should then output a single line plot using the specified line style. A set of line styles can also be specified as part of this problem definition.

4-19. This program can be completed using modified procedures from previous programming examples in Chapters 2 through 4 and an input file of data values. The program can then output a single polyline graph of the data, or other options could be specified as in previous exercises.

4-20. This program can be completed using modified procedures from previous programming examples in Chapters 2 through 4 and using several input data sets. Methods for this exercise are similar to those discussed in the solution to Exercise 4-13 and other exercises.

4-21. Methods for simulating brush shapes are discussed in Section 4-6. Two or more brush patterns can be specified for this problem, using an input parameter to select a particular pattern. The brush strokes can include solid curved lines with different end patterns, as well as various pixel masks. For example, 3 by 3 or 4 by 4 masks can be devised with some pixels set to the foreground color and some set to the background color.

4-22. Assuming that horizontal and vertical lines are to be displayed with intensity I and diagonal lines are to be displayed with intensity $\sqrt{2}\,I$, the intensity I' for any line with slope m could be calculated as

$$I' = I\left[|m|(\sqrt{2}-1)+1\right] \qquad \text{for } |m| <= 1$$

$$I' = I\left[(1/|m|)(\sqrt{2}-1)+1\right] \qquad \text{for } |m| > 1$$

This computed intensity value can be used for each RGB color component of the line segment.

4-23. The methods for displaying an ellipse with various line styles are the same as those for straight-line segments (Exercise 4-7).

4-24. Displaying an ellipse with a specified width w can be accomplished by generating a set of $w/2$ adjacent concentric ellipses on each side of the original ellipse path.

4-25. Methods for generating a bar chart within a given space are discussed in the solution to Exercise 3-46. In addition, input parameters are to be used to designate a color or pattern for the bars (Section 4-9).

4-26. This is a modification of the preceding exercise, with pairs of overlapping bars centered on the x coordinate corresponding to each value in the two data sets.

4-27. A color table can be set up as a set of three arrays, one for each of the RGB components in a color. An integer value assigned to a pixel is then used as the index into each of the three color-component arrays, and the pixel is set to the corresponding RGB color.

4-28. With 100 pixels per inch and an 8 inch by 10 inch screen, the number of pixels is 800×1000. Multiplying the total number of pixels by 6 bits per pixel, we obtain a minimum frame buffer size of approximately 586 KB.

4-29. (a) The color table can store 2^8 different values for each of the RGB components, and $R = G = B$ for any shade of gray. Therefore, the number of distinct gray levels is 2^8.

(b) Since each color is represented with 24 bits, the number of distinct colors is $2^{24} =$ 16,777,216.

(c) If each pixel is assigned a different color, the screen will display the maximum number of simultaneous colors, which is $512 \times 512 = 2^{18} = 262{,}144$ different colors.

(d) The frame buffer size is 20×2^{18} bits, and the color table size is $24 \times 20 \times 2^{18}$ bits.

(e) Memory size can be reduced by eliminating the color table and assigning 24 bits to each pixel position in the frame buffer. Also, the maximum number of colors that can be displayed at one time is 2^{18}. Therefore, size of the color table could be reduced by selecting a set of 2^{18} different colors for storage in the color table.

4-30. The lower left corner (or any other point) of the rectangular pattern can be positioned at the specified starting position. Then the pattern is tiled across the polygon interior, as in Fig. 4-31, and each pixel position across each polygon scan-line is assigned the color value corresponding to the pattern color mapped to that position.

4-31. The midpoint algorithm can be used to generate pixel positions along the ellipse path. These positions are the intersection points of the scan lines that cross the interior of the ellipse. Each pixel along the interior of each scan line is then assigned the specified fill color.

4-32. As in the preceding exercise, the ellipse boundary positions can be generated using the midpoint algorithm, which determines the endpoints for the ellipse scan lines. The parameters r_x, r_y, and the ellipse center coordinates can be used to determine the coordinate extents for the ellipse, and the specified fill pattern can then be mapped into the area bounded by these coordinate extents. Each pixel position along an interior scan line is then assigned the pattern color that maps to that position. The start position for the lower-left corner of the pattern can be the lower-left corner of the bounding rectangle for the ellipse, the ellipse center position, or some other starting point. Also, any other pattern position, such as its center, can be used as the reference position.

4-33. The nonzero winding-number rule is discussed in Section 3-14. From the y-coordinate extents of the object, determine the set of scan lines that cross the object interior. The inside and outside regions of the object along each scan line can then be determined by calculating the winding number at edge intersection positions along that scan line, as positions along the scan line are processed from left to right.

Sort the object edges from left to right, omitting all horizontal edges from the sorted edge list. For each scan line, initialize the winding number to 0 and form a sublist of edges that cross that scan line. (If the endpoint y coordinates for an edge are on opposite sides of a scan line, then the scan line intersects that edge.) Next, locate the leftmost edge that intersects that scan line. If the y coordinate for the first endpoint of the leftmost intersected edge is below the scan line, then the edge crosses the scan line from right to left and 1 is added to the winding number. Otherwise, 1 is subtracted from the winding number. In either case, the region to the right of the single leftmost intersected edge is an interior region. If the leftmost scan-line intersection is at the intersection of two or more crossing edges, determine the contribution to the winding number from each of the crossing edges. (Also, it could be assumed that no more than two edges intersect at any coordinate position.)

Continue to process the edges from left to right, updating the winding number at each intersected edge. Whenever the new value for the winding number at an intersected edge (or edges) is 0, the region to the right of that edge is exterior. Otherwise, the region is interior. If a scan line intersects an edge endpoint for two edges that cross the scan line in the same direction, treat that endpoint as one edge intersection. Otherwise, ignore that edge endpoint intersection. Pixel positions within interior regions are set to the fill color, which can be either a specified single color for the object interior or a fill-pattern color (Exercise 4-30).

4-34. Section 4-13 and Fig. 4-29 illustrate a scan-line modification for the boundary-fill algorithm. Basically, this procedure completes the processing of pixel positions across a horizontal pixel span before proceeding to the next scan line. Thus, only the start position for each pixel span need be stacked. For a 4-connected region, only the horizontal and vertical neighbors of a pixel are tested for the boundary color.

4-35. To avoid excessive stacking, scan-line methods should be used to process the pixel positions. The methods of the preceding exercise are extended to 8-connected regions by testing diagonal pixels, as well as the neighboring horizontal and vertical pixel positions.

4-36. An ellipse can be filled correctly with a 4-connected method. But the 8-connected method would allow the fill color to leak outside the ellipse boundary along diagonal pixels from some positions.

4-37. This algorithm is essentially the same as the boundary-fill methods discussed in the solutions to Exercises 4-34 and 4-35. The only difference is that pixels are tested against a specified interior color instead of a boundary color.

4-38. The size of a fill pattern can be changed by specifying the dimensions of the rectangular pixel area that is to be covered by each element of the pattern. For example, instead of mapping each element of a pattern array to a single pixel, each element could be mapped to a 4 by 4 group of pixels.

4-39. Color blending methods are discussed in Section 4-9. A foreground color $\mathbf{F}$ can be combined with a background color $\mathbf{B}$, using the calculation in Eq. 4-2, where $\mathbf{F} \neq \mathbf{B}$. The transparency factor (blending factor) t can be an input parameter. This procedure could also blend multiple colors (Eq. 4-5).

4-40. The size of character bit patterns can changed by specifying the dimensions of the rectangular pixel area that is to be covered by each element in the character grid. For example, if the total size of a character is to be increased by a factor of 3, then each bit in the pattern is mapped to a 3 by 3 group of pixels. Similar mappings can be applied to change either the width or the height separately.

4-41. An up vector is defined by specifying its direction, which is the angle θ between the up vector and the positive x axis. We can also define an up vector by giving its width Δx and height Δy. Then $\tan\theta = \Delta y/\Delta x$. Once angle θ is known, character patterns can be rotated using the methods described in the solution to Exercise 3-38. The colors for the characters can then be loaded into the frame buffer at the rotated positions, according to the sequence set by the text path parameter.

4-42. One or more input parameters can used to specify the text alignment. One parameter can give a reference position and another parameter can specify the alignment with respect to that position (left, right, top, bottom, or center). Also, characters can be aligned above or below the reference position, or to the left or right of the reference position, depending on the text direction.

4-43. Marker attributes are color, character type, and size. Color is typically set by the current color state. A character type can be chosen with a character function within a graphics library, or a set of defined marker symbols, such as a set of bit maps, can be defined for a particular application. The size of a bit-map marker symbol can be adjusted using methods described in the solution to Exercise 4-40. If a marker symbol is defined as a pattern of line segments or fill areas, the scaling methods described in Chapter 5 can be used to change the size of the symbol.

4-44. Antialiasing methods are discussed in Section 4-17. An approximate method for antialiasing a line using the Bresenham algorithm is to assign an intensity value to a pixel according to its distance from the line path. If the slope of the line is less than or equal to one in magnitude, the vertical (y-coordinate) distance can be used as an approximate measure of pixel intensity:

$$I = I_{max}(1.0 - |dy|)$$

where dy is the fractional distance from the pixel to the line path. Intensity I is maximum when $dy = 0$, and I decreases to 0 for pixels at a distance of 1.0 or more from the line path. More accurate calculations could be assigned for this problem, with an increase in computation time.

For a steep line (slope magnitude greater than 1), horizontal displacement can be used as a measure of pixel distance from the line path. Horizontal distance dx then replaces dy in the above calculation.

4-45. The methods outlined in the previous exercise can be applied to any line algorithm. Basically, the midpoint line algorithm is the same as the Bresenham algorithm. The decision parameters for the midpoint line algorithm can be used in their original forms (Exercises 3-4 and 3-5), or they can be converted to the Bresenham decision parameters.

4-46. Antialiasing methods are discussed in Section 4-17. The midpoint ellipse algorithm can be modified using methods discussed in Exercise 4-44.

4-47. Section 4-17 surveys techniques for area antialiasing. A fill area is antialiased by smoothing ("softening") the nonhorizontal and nonvertical edges of the area. Therefore, for a scan line that is not along a horizontal edge, antialiasing is applied at the end points of the scan-line, and a line algorithm could be modified to determine the intensity variations, as pixel positions are calculated along the edges of the fill area.

4-48. The Pitteway-Watkinson algorithm (Section 4-17) is a modified midpoint line algorithm that is applied to fill-area edges that are not vertical or horizontal. In this algorithm, the value of the decision parameter is the fraction of area coverage for the current pixel. Therefore, as the decision parameter is calculated at each step, the algorithm specifies the next pixel along an edge and the floating-point value for the antialiased intensity of the current pixel. This value is then mapped to the nearest available intensity level. For instance, if 5 intensity levels (labeled 0 through 4) are to be used in the algorithm, then a decision-parameter value of 0.6 would be mapped to intensity level 2.