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 can be represented with a polygon approximation of a circular area and the side can be approximated with a set of rectangular strips. Vertex positions for the polygon mesh are calculated using given values for the radius and height of a cylinder.

First, compute a set of equally spaced points around the perimeter of the cylinder top. These points provide the vertices for a convex polygon representation of the top of the cylinder. Then, using the value for the cylinder height, create a set of vertical lines from the vertex positions at the top of the cylinder down to the bottom end of the cylinder. This produces a set of positions along the perimeter at the cylinder bottom, which are the remaining vertex positions needed for the polygon at the bottom and the rectangles along the sides of the cylinder. All information for vertex coordinates, the polygon edges, and the surface facets can be stored in polygon tables.

A complete algorithm for generating the vertex positions in the polygon mesh requires a reference position for the cylinder. For this exercise, the cylinder axis can be assumed to be along the z axis of a three-dimensional Cartesian reference frame, with the bottom of the cylinder in the xy plane. The center position for the bottom of the cylinder is then (0, 0, 0), and the center position for the top of the cylinder is (0, 0, h), where h is the cylinder height. Basic parameters defining the cylinder are its radius and its height. The number of subdivisions around the top perimeter of the cylinder could be a fixed value, or this could be specified as an additional parameter for the cylinder. Another option is to specify a different orientation for the cylinder, such as placing the cylinder base in the yz plane with its axis along the x axis.

To facilitate processing and to ensure that the vertex positions for each surface facet are coplanar, additional calculations can be performed to convert the polygon representation into a triangular mesh. This is accomplished for the top and bottom of the cylinder by adding radial lines from the circle center to the vertex positions on the circle perimeter. For each rectangular strip along the side of the cylinder, add a diagonal line. Also, the size of the polygons along the side of the cylinder can be reduced by first dividing each rectangle into a number of smaller rectangles, then splitting the reduced rectangles into triangles. In general, the advantage of converting a curved surface representation to a polygon-mesh approximation is that the surface is now described with linear equations, which reduces processing time for the viewing and rendering routines.

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 slope-intercept representations $y_k = m_k\,x + b_k$ (Eq. 3-1), providing neither line is vertical. If the resulting x value is outside the coordinate range for either line, the line segments do not intersect. As an example, intersection coordinates for lines labeled $k = 1$ and $k = 2$ are computed with the equations

$$x = -(b_2 - b_1)/(m_2 - m_1), \qquad y = m_1\,x + b_1$$

where the slope of each line is calculated from line endpoint coordinates.

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 locate the intersection coordinates for any two lines, regardless of their slopes.

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.