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.