CHAPTER 4

Attributes of Graphics Primitives

One frame from a computer-generated cartoon illustrating a variety of object colors and other attributes. *(Courtesy of SOFTIMAGE, Inc.)*

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

One way to incorporate attribute options into a graphics package is to extend the parameter list associated with each graphics-primitive function to include the appropriate attribute values. A line-drawing function, for example, could contain additional parameters to set the color, width, and other properties of a line. Another approach is to maintain a system list of current attribute values. Separate functions are then included in the graphics package for setting the current values in the attribute list. To generate a primitive, the system checks the relevant attributes and invokes the display routine for that primitive using the current attribute settings. Some graphics packages use a combination of methods for setting attribute values, and other libraries, including OpenGL, assign attributes using separate functions that update a system attribute list.

A graphics system that maintains a list for the current values of attributes and other parameters is referred to as a **state system** or **state machine.** Attributes of output primitives and some other parameters, such as the current frame-buffer

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

position, are referred to as **state variables** or **state parameters.** When we assign a value to one or more state parameters, we put the system into a particular state. And that state remains in effect until we change the value of a state parameter.

4-1 OpenGL STATE VARIABLES

Attribute values and other parameter settings are specified with separate functions that define the current OpenGL state. The state parameters in OpenGL include color and other primitive attributes, the current matrix mode, the elements of the model-view matrix, the current position for the frame buffer, and the parameters for the lighting effects in a scene. All OpenGL state parameters have default values, which remain in effect until new values are specified. At any time, we can query the system to determine the current value of a state parameter. In the following sections of this chapter, we discuss only the attribute settings for output primitives. Other state parameters are examined in later chapters.

All graphics primitives in OpenGL are displayed with the attributes in the current state list. Changing one or more of the attribute settings affects only those primitives that are specified after the OpenGL state is changed. Primitives that were defined before the state change retain their attributes. Thus we can display a green line, change the current color to red, and define another line segment. Both the green line and the red line will then be displayed. Also, some OpenGL state values can be specified within `glBegin/glEnd` pairs, along with the coordinate values, so that parameter settings can vary from one coordinate position to another.

4-2 COLOR AND GRAY SCALE

A basic attribute for all primitives is color. Various color options can be made available to a user, depending on the capabilities and design objectives of a particular system. Color options can be specified numerically or selected from menus or displayed slider scales. For a video monitor, these color codes are then converted to intensity-level settings for the electron beams. With color plotters, the codes might control ink-jet deposits or pen selections.

RGB Color Components

In a color raster system, the number of color choices available depends on the amount of storage provided per pixel in the frame buffer. Also, color information can be stored in the frame buffer in two ways: We can store RGB color codes directly in the frame buffer, or we can put the color codes into a separate table and use the pixel locations to store index values referencing the color-table entries. With the direct storage scheme, whenever a particular color code is specified in an application program, that color information is placed in the frame buffer at the location of each component pixel in the output primitives to be displayed in that color. A minimum number of colors can be provided in this scheme with 3 bits of storage per pixel, as shown in Table 4-1. Each of the three bit positions is used to control the intensity level (either on or off, in this case) of the corresponding electron gun in an RGB monitor. The leftmost bit controls the red gun, the middle bit controls the green gun, and the rightmost bit controls the blue gun. Adding more bits per pixel to the frame buffer increases the number of color choices we have. With 6 bits per pixel, 2 bits can be used for each gun. This allows four

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

TABLE 4-1

THE EIGHT RGB COLOR CODES FOR A THREE-BIT PER PIXEL FRAME BUFFER

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

different intensity settings for each of the three color guns, and a total of 64 color options are available for each screen pixel. As more color options are provided, the storage required for the frame buffer also increases. With a resolution of 1024 by 1024, a full-color (24-bit per pixel) RGB system needs 3 megabytes of storage for the frame buffer.

Color tables are an alternate means for providing extended color capabilities to a user without requiring large frame buffers. At one time, this was an important consideration. But today, hardware costs have decreased dramatically and extended color capabilities are fairly common, even in low-end personal computer systems. So most of our examples will simply assume that RGB color codes are stored directly in the frame buffer.

Color Tables

Figure 4-1 illustrates a possible scheme for storing color values in a **color lookup table** (or **color map**). Sometimes a color table is referred to as a **video lookup table.**

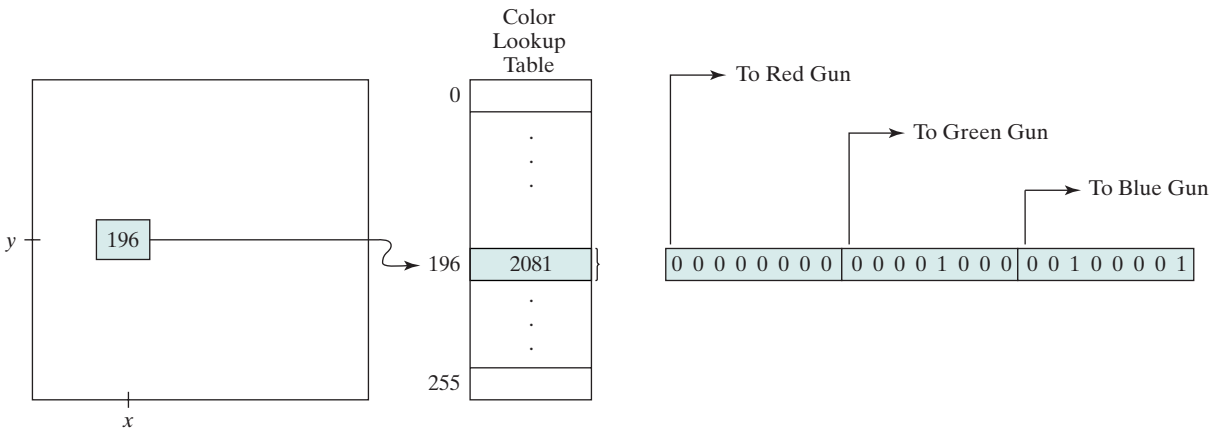


FIGURE 4–1 A color lookup table with 24 bits per entry that is accessed from a frame buffer with 8 bits per pixel. A value of 196 stored at pixel position (x, y) references the location in this table containing the hexadecimal value 0x0821 (a decimal value of 2081). Each 8-bit segment of this entry controls the intensity level of one of the three electron guns in an RGB monitor.

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

Values stored in the frame buffer are now used as indices into the color table. In this example, each pixel can reference any one of the 256 table positions, and each entry in the table uses 24 bits to specify an RGB color. For the hexadecimal color code 0x0821, a combination green-blue color is displayed for pixel location (x, y). Systems employing this particular lookup table allow a user to select any 256 colors for simultaneous display from a palette of nearly 17 million colors. Compared to a full-color system, this scheme reduces the number of simultaneous colors that can be displayed, but it also reduces the frame-buffer storage requirement to 1 megabyte. Multiple color tables are sometimes available for handling specialized rendering applications, such as antialiasing, and they are used with systems that contain more than one color output device.

A color table can be useful in a number of applications, and it can provide a "reasonable" number of simultaneous colors without requiring large frame buffers. For most applications, 256 or 512 different colors are sufficient for a single picture. Also, table entries can be changed at any time, allowing a user to be able to experiment easily with different color combinations in a design, scene, or graph without changing the attribute settings for the graphics data structure. When a color value is changed in the color table, all pixels with that color index immediately change to the new color. Without a color table, we can only change the color of a pixel by storing the new color at that frame-buffer location. Similarly, data-visualization applications can store values for some physical quantity, such as energy, in the frame buffer and use a lookup table to experiment with various color combinations without changing the pixel values. And in visualization and image-processing applications, color tables are a convenient means for setting color thresholds so that all pixel values above or below a specified threshold can be set to the same color. For these reasons, some systems provide both capabilities for storing color information. A user can then elect either to use color tables or to store color codes directly in the frame buffer.

Gray Scale

Since color capabilities are now common in computer-graphics systems, we use RGB color functions to set shades of gray, or **gray scale,** in an application program. When an RGB color setting specifies an equal amount of red, green, and blue, the result is some shade of gray. Values close to 0 for the color components produce dark gray, and higher values near 1.0 produce light gray. Applications for gray-scale display methods include enhancing black-and-white photographs and generating visualization effects.

Other Color Parameters

In addition to an RGB specification, other three-component color representations are useful in computer-graphics applications. For example, color output on printers is described with cyan, magenta, and yellow color components, and color interfaces sometimes use parameters such as lightness and darkness to choose a color. Also, color, and light in general, is a complex subject, and many terms and concepts have been devised in the fields of optics, radiometry, and psychology to describe the various aspects of light sources and lighting effects. Physically, we can describe a color as electromagnetic radiation with a particular frequency range and energy distribution, but then there are also the characteristics of our perception of the color. Thus, we use the physical term *intensity* to quantify the

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

amount of light energy radiating in a particular direction over a period of time, and we use the psychological term *luminance* to characterize the perceived brightness of the light. We discuss these terms and other color concepts in greater detail when we consider methods for modeling lighting effects (Chapter 10) and the various models for describing color (Chapter 12).

4-3 OpenGL COLOR FUNCTIONS

In the example program at the end of Chapter 2, we introduced a few OpenGL color routines. We used one function to set the color for the display window, and we used another function to specify a color for the straight-line segment. Also, we set the **color display mode** to RGB with the statement

```
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
```

The first parameter in the argument list states that we are using a single buffer for the frame buffer, and the second parameter puts us into the RGB (or RGBA) mode, which is the default color mode. We can use either `GLUT_RGB` or `GLUT_RGBA` to select this color mode. If we wanted to specify colors by an index into a color table, we would replace the OpenGL constant `GLUT_RGB` with `GLUT_INDEX`.

The OpenGL RGB and RGBA Color Modes

Most color settings for OpenGL primitives are made in the **RGB mode,** which is basically the same as the **RGBA mode.** The only difference between RGB and RGBA is whether or not we are employing the alpha value for color blending. When we specify a particular set of color values for primitives, we define the *color state* of OpenGL. The current color is applied to all subsequently defined primitives until we change the color settings. A new color specification affects only the objects we define after the color change.

In RGB mode, we specify values for the red, green, and blue components of a color. As we noted in Section 2-9, the fourth color parameter, the **alpha coefficient,** is optional, and a four-dimensional color specification is called the RGBA color. This fourth color parameter can be used to control color blending for overlapping primitives. An important application of color blending is in the simulation of transparency effects. For these calculations, the value of alpha corresponds to a transparency (or, opacity) setting. In the RGB (or RGBA) mode, we select the current color components with the function:

```
glColor* (colorComponents);
```

Suffix codes are similar to those for the `glVertex` function. We use a code of either 3 or 4 to specify the RGB or RGBA mode along with the numerical data-type code and an optional vector suffix. The suffix codes for the numerical data types are `b` (byte), `i` (integer), `s` (short), `f` (float), and `d` (double), as well as unsigned numerical values. Floating-point values for the color components are in the range from 0.0 to 1.0, and the default color components for `glColor`, including the alpha value, are (1.0, 1.0, 1.0, 1.0), which sets the RGB color to white and the alpha value to 1.0. As an example, the following statement uses floating-point values

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

in RGB mode to set the current color for primitives to cyan (a combination of the highest intensities for green and blue).

```
glColor3f (0.0, 1.0, 1.0);
```

Using an array specification for the three color components, we could set the color in the above example as

```
glColor3fv (colorArray);
```

An OpenGL color selection can be assigned to individual point positions within `glBegin/glEnd` pairs.

Integer specifications for the color components depend on the capabilities of the system. For a full-color system, which allocates 8 bits per pixel (256 levels for each color component), integer color values range from 0 to 255. The corresponding floating-point values for the color components would then be 0.0, 1.0/255.0, 2.0/255.0, . . . , 255.0/255.0 = 1.0. With a full-color system, we can specify the cyan color in the previous example using integer values for the color components as

```
glColor3i (0, 255, 255);
```

Frame-buffer positions actually store integer values, so specifying the color values as integers avoids the conversions necessary when floating-point values are given. A specified color value in any format is scaled to an integer within the range of the number of bits available on a particular system.

OpenGL Color-Index Mode

Color specifications in OpenGL can also be given in the **color-index mode,** which references values in a color table. Using this mode, we set the current color by specifying an index into a color table:

```
glIndex* (colorIndex);
```

Parameter `colorIndex` is assigned a nonnegative integer value. This index value is then stored in the frame-buffer positions for subsequently specified primitives. We can specify the color index in any of the following data types: unsigned byte, integer, or floating point. Data type for parameter `colorIndex` is indicated with a suffix code of `ub`, `s`, `i`, `d`, or `f`, and the number of index positions in a color table is always a power of 2, such as 256 or 1024. The number of bits available at each table position depends on the hardware features of the system. As an example of specifying a color in index mode, the following statement sets the current color index to the value 196.

```
glIndexi (196);
```

All primitives defined after this statement will be assigned the color stored at that position in the color table, until the current color is changed.

There are no functions provided in the core OpenGL library for loading values into a color-lookup table, because table-processing routines are part of a window system. Also, some window systems support multiple color tables and full color, while other systems may have only one color table and limited color choices.

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

However, we do have a GLUT routine that interacts with a window system to set color specifications into a table at a given index position:

```
glutSetColor (index, red, green, blue);
```

Color parameters `red`, `green`, and `blue` are assigned floating-point values in the range from 0.0 to 1.0. This color is then loaded into the table at the position specified by the value of parameter `index`.

Routines for processing three other color tables are provided as extensions to the OpenGL core library. These routines are part of the **Imaging Subset** of OpenGL. Color values stored in these tables can be used to modify pixel values as they are processed through various buffers. Some examples of using these tables are setting camera focusing effects, filtering out certain colors from an image, enhancing certain intensities or making brightness adjustments, converting a gray-scale photograph to color, and antialiasing a display. And we can use these tables to change color models; that is, we can change RGB colors to another specification using three other "primary" colors (such as cyan, magenta, and yellow).

A particular color table in the Imaging Subset of OpenGL is activated with the `glEnable` function using one of the table names: `GL_COLOR_TABLE`, `GL_POST_CONVOLUTION_COLOR_TABLE`, or `GL_POST_COLOR_MATRIX_COLOR_TABLE`. We can then use routines in the Imaging Subset to select a particular color table, set color-table values, copy table values, or specify which component of a pixel's color we want to change and how we want to change it.

OpenGL Color Blending

In many applications, it is convenient to be able to combine the colors of overlapping objects or to blend an object with the background. Some examples are simulating a paintbrush effect, forming a composite image of two or more pictures, modeling transparency effects, and antialiasing the objects in a scene. Most graphics packages provide methods for producing various color-mixing effects, and these procedures are called **color-blending functions** or **image-compositing functions.** In OpenGL, the colors of two objects can be blended by first loading one object into the frame buffer, then combining the color of the second object with the frame-buffer color. The current frame-buffer color is referred to as the OpenGL *destination color* and the color of the second object is the OpenGL *source color*. Blending methods can be performed only in RGB or RGBA mode. To apply color blending in an application, we first need to activate this OpenGL feature using the following function.

```
glEnable (GL_BLEND);
```

And we turn off the color-blending routines in OpenGL with

```
glDisable (GL_BLEND);
```

If color blending is not activated, an object's color simply replaces the frame-buffer contents at the object's location.

Colors can be blended in a number of different ways, depending on the effects we want to achieve, and we generate different color effects by specifying two sets of *blending factors*. One set of blending factors is for the current object in the frame buffer (the "destination object"), and the other set of blending factors is for the

incoming ("source") object. The new, blended color that is then loaded into the frame buffer is calculated as

$$(S_r R_s + D_r R_d, S_g G_s + D_g G_d, S_b B_s + D_b B_d, S_a A_s + D_a A_d) \qquad (4\text{-}1)$$

where the RGBA source color components are (R_s, G_s, B_s, A_s), the destination color components are (R_d, G_d, B_d, A_d), the source blending factors are (S_r, S_g, S_b, S_a), and the destination blending factors are (D_r, D_g, D_b, D_a). Computed values for the combined color components are clamped to the range from 0.0 to 1.0. That is, any sum greater than 1.0 is set to the value 1.0, and any sum less than 0.0 is set to 0.0.

We select the blending-factor values with the OpenGL function

```
glBlendFunc (sFactor, dFactor);
```

Parameters `sFactor` and `dFactor`, the source and destination factors, are each assigned an OpenGL symbolic constant specifying a predefined set of four blending coefficients. For example, the constant `GL_ZERO` yields the blending factors (0.0, 0.0, 0.0, 0.0) and `GL_ONE` gives us the set (1.0, 1.0, 1.0, 1.0). We could set all four blending factors either to the destination alpha value or to the source alpha value using `GL_DST_ALPHA` or `GL_SRC_ALPHA`. Other OpenGL constants that are available for setting the blending factors include `GL_ONE_MINUS_DST_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`, `GL_DST_COLOR`, and `GL_SRC_COLOR`. These blending factors are often used for simulating transparency, and they are discussed in greater detail in Section 10-19. The default value for parameter `sFactor` is `GL_ONE`, and the default value for parameter `dFactor` is `GL_ZERO`. Hence, the default values for the blending factors result in the incoming color values replacing the current values in the frame buffer.

Additional functions have been included in an OpenGL extension called the Imaging Subset. These functions include a routine to set a blending color and another routine to specify a blending equation.

OpenGL Color Arrays

We can also specify color values for a scene in combination with the coordinate values in a vertex array (Section 3-17). This can be done either in RGB mode or in color-index mode. As with vertex arrays, we must first activate the color-array features of OpenGL:

```
glEnableClientState (GL_COLOR_ARRAY);
```

Then, for RGB color mode, we specify the location and format of the color components with

```
glColorPointer (nColorComponents, dataType, offset, colorArray);
```

Parameter `nColorComponents` is assigned a value of either 3 or 4, depending upon whether we are listing RGB or RGBA color components in the array `colorArray`. An OpenGL symbolic constant such as `GL_INT` or `GL_FLOAT` is assigned to parameter `dataType` to indicate the data type for the color values. For a separate color array, we can assign the value 0 to parameter `offset`. But if

we combine color data with vertex data in the same array, the `offset` value is the number of bytes between each set of color components in the array.

As an example of using color arrays, we can modify the vertex-array example in Section 3-17 to include a color array. The following code fragment sets the color of all vertices on the front face of the cube to blue, and all vertices of the back face are assigned the color red.

```
typedef GLint vertex3 [3], color3 [3];

vertex3 pt [8] = { {0, 0, 0}, {0, 1, 0}, {1, 0, 0},
    {1, 1, 0}, {0, 0, 1}, {0, 1, 1}, {1, 0, 1}, {1, 1, 1} };
color3 hue [8] = { {1, 0, 0}, {1, 0, 0}, {0, 0, 1},
    {0, 0, 1}, {1, 0, 0}, {1, 0, 0}, {0, 0, 1}, {0, 0, 1} };

glEnableClientState (GL_VERTEX_ARRAY);
glEnableClientState (GL_COLOR_ARRAY);

glVertexPointer (3, GL_INT, 0, pt);
glColorPointer (3, GL_INT, 0, hue);
```

We can even stuff both the colors and the vertex coordinates into one **interlaced array.** Each of the pointers would then reference the single interlaced array, with an appropriate `offset` value. For example,

```
static GLint hueAndPt [ ] =
    {1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0,
     0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0,
     1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1,
     0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1};

glVertexPointer (3, GL_INT, 6*sizeof(GLint), hueAndPt [3]);
glColorPointer (3, GL_INT, 6*sizeof(GLint), hueAndPt [0]);
```

The first three elements of this array specify an RGB color value, the next three elements specify a set of (x, y, z) vertex coordinates, and this pattern continues to the last color-vertex specification. We set the `offset` parameter to the number of bytes between successive color, or vertex, values, which is `6*sizeof(GLint)` for both. Color values start at the first element of the interlaced array, which is `hueAndPt [0]`, and vertex values start at the fourth element, which is `hueAndPt [3]`.

Since a scene generally contains several objects, each with multiple planar surfaces, OpenGL provides a function in which we can specify all the vertex and color arrays at once, as well as other types of information. If we change the color and vertex values in the above example to floating point, we use this function in the form

```
glInterleavedArrays (GL_C3F_V3F, 0, hueAndPt);
```

The first parameter is an OpenGL constant that indicates three-element floating-point specifications for both color (C) and vertex coordinates (V). And the elements of array `hueAndPt` are to be interlaced with the color for each vertex listed before the coordinates. This function also automatically enables both vertex and color arrays.

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

In color-index mode, we define an array of color indices with

```
glIndexPointer (type, stride, colorIndex);
```

Color indices are listed in the array `colorIndex` and the `type` and `stride` parameters are the same as in `glColorPointer`. No `size` parameter is needed since color-table indices are specified with a single value.

Other OpenGL Color Functions

In our first programming example in Section 2-9, we introduced the following function that selects RGB color components for a display window.

```
glClearColor (red, green, blue, alpha);
```

Each color component in the designation (red, green, and blue), as well as the alpha parameter, is assigned a floating-point value in the range from 0.0 to 1.0. The default value for all four parameters is 0.0, which produces the color black. If each color component is set to 1.0, the clear color is white. Shades of gray are obtained with identical values for the color components between 0.0 and 1.0. The fourth parameter, `alpha`, provides an option for blending the previous color with the current color. This can occur only if we activate the blending feature of OpenGL; color blending cannot be performed with values specified in a color table.

As we noted in Section 3-19, there are several *color buffers* in OpenGL that can be used as the current refresh buffer for displaying a scene, and the `glClearColor` function specifies the color for all the color buffers. We then apply the clear color to the color buffers with the command:

```
glClear (GL_COLOR_BUFFER_BIT);
```

We can also use the `glClear` function to set initial values for other buffers that are available in OpenGL. These are the *accumulation buffer,* which stores blended-color information, the *depth buffer,* which stores depth values (distances from the viewing position) for objects in a scene, and the *stencil buffer,* which stores information to define the limits of a picture.

In color-index mode, we use the following function (instead of `glClearColor`) to set the display-window color.

```
glClearIndex (index);
```

The window background color is then assigned the color that is stored at position `index` in the color table. And the window is displayed in this color when we issue the `glClear (GL_COLOR_BUFFER_BIT)` function.

Many other color functions are available in the OpenGL library for dealing with a variety of tasks, such as changing color models, setting lighting effects for a scene, specifying camera effects, and rendering the surfaces of an object. We examine other color functions as we explore each of the component processes in a computer-graphics system. For now, we limit our discussion to those functions relating to color specifications for graphics primitives.

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

4-4 POINT ATTRIBUTES

Basically, we can set two attributes for points: color and size. In a state system, the displayed color and size of a point is determined by the current values stored in the attribute list. Color components are set with RGB values or an index into a color table. For a raster system, point size is an integer multiple of the pixel size, so that a large point is displayed as a square block of pixels.

4-5 LINE ATTRIBUTES

A straight-line segment can be displayed with three basic attributes: color, width, and style. Line color is typically set with the same function for all graphics primitives, while line width and line style are selected with separate line functions. Additionally, lines may be generated with other effects, such as pen and brush strokes.

Line Width

Implementation of line-width options depends on the capabilities of the output device. A heavy line could be displayed on a video monitor as adjacent parallel lines, while a pen plotter might require pen changes to draw a thick line.

For raster implementations, a standard-width line is generated with single pixels at each sample position, as in the Bresenham algorithm. Thicker lines are displayed as positive integer multiples of the standard line by plotting additional pixels along adjacent parallel line paths. If a line has slope magnitude less than or equal to 1.0, we can modify a line-drawing routine to display thick lines by plotting a vertical span of pixels in each column (x position) along the line. The number of pixels to be displayed in each column is set equal to the integer value of the line width. In Fig. 4-2 we display a double-width line by generating a parallel line above the original line path. At each x sampling position, we calculate the corresponding y coordinate and plot pixels at screen coordinates (x, y) and $(x, y + 1)$. We could display lines with a width of 3 or greater by alternately plotting pixels above and below the single-width line path.

With a line slope greater than 1.0 in magnitude, we can display thick lines using horizontal spans, alternately picking up pixels to the right and left of the line path. This scheme is demonstrated in Fig. 4-3, where a line segment with a width of 4 is plotted using multiple pixels across each scan line. Similarly, a thick line with slope less than or equal to 1.0 can be displayed using vertical pixel spans. We can implement this procedure by comparing the magnitudes of the horizontal and vertical separations (Δx and Δy) of the line endpoints. If $|\Delta x| \geq |\Delta y|$,

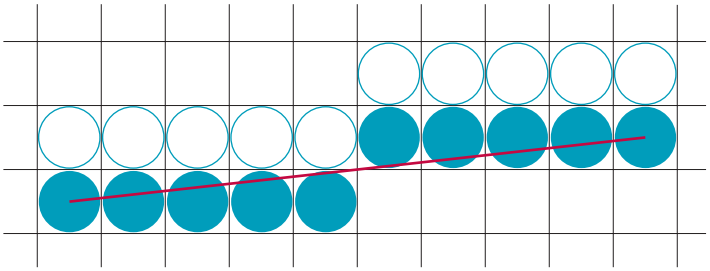


FIGURE 4–2 A double-wide raster line with slope $|m| < 1.0$ generated with vertical pixel spans.

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

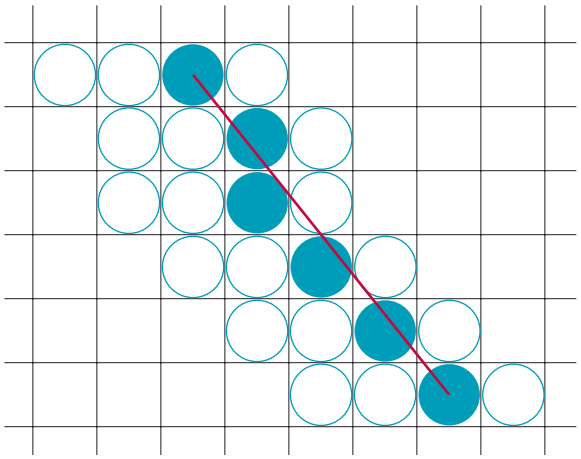


FIGURE 4–3 A raster line with slope $|m| > 1.0$ and a line width of 4 plotted using horizontal pixel spans.

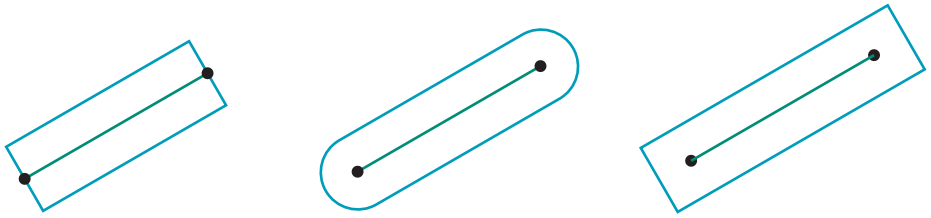


FIGURE 4–4 Thick lines drawn with (a) butt caps, (b) round caps, and (c) projecting square caps.

pixels are replicated along columns. Otherwise, multiple pixels are plotted across rows.

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

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

Other methods for producing thick lines include displaying the line as a filled rectangle or generating the line with a selected pen or brush pattern, as discussed in the next section. To obtain a rectangle representation for the line boundary, we calculate the position of the rectangle vertices along perpendiculars to the line path so that the rectangle vertex coordinates are displaced from the original line-endpoint positions by one-half the line width. The rectangular line then appears

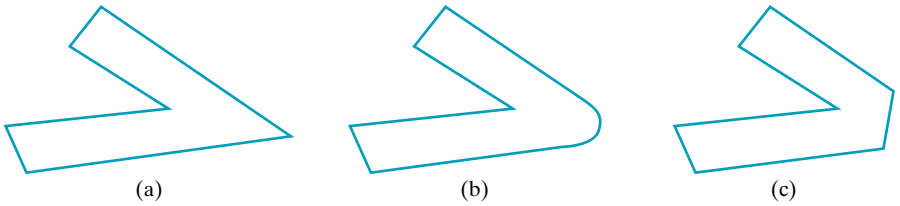


FIGURE 4–5 Thick line segments connected with a miter join (a), a round join (b), and a bevel join (c).

as in Fig. 4-4 (a). We could add round caps to the filled rectangle, or we could extend its length to display projecting square caps.

Generating thick polylines requires some additional considerations. In general, the methods we have considered for displaying a single line segment will not produce a smoothly connected series of line segments. Displaying thick polylines using horizontal and vertical pixel spans, for example, leaves pixel gaps at the boundaries between line segments with different slopes where there is a shift from horizontal pixel spans to vertical spans. We can generate thick polylines that are smoothly joined at the cost of additional processing at the segment endpoints. Figure 4-5 shows three possible methods for smoothly joining two line segments. A *miter join* is accomplished by extending the outer boundaries of each of the two line segments until they meet. A *round join* is produced by capping the connection between the two segments with a circular boundary whose diameter is equal to the line width. And a *bevel join* is generated by displaying the line segments with butt caps and filling in the triangular gap where the segments meet. If the angle between two connected line segments is very small, a miter join can generate a long spike that distorts the appearance of the polyline. A graphics package can avoid this effect by switching from a miter join to a bevel join when, for example, the angle between any two consecutive segments is small.

Line Style

Possible selections for the line-style attribute include solid lines, dashed lines, and dotted lines. We modify a line-drawing algorithm to generate such lines by setting the length and spacing of displayed solid sections along the line path. With many graphics packages, we can select the length of both the dashes and the inter-dash spacing.

Raster line algorithms display line-style attributes by plotting pixel spans. For dashed, dotted, and dot-dashed patterns, the line-drawing procedure outputs sections of contiguous pixels along the line path, skipping over a number of intervening pixels between the solid spans. Pixel counts for the span length and inter-span spacing can be specified in a **pixel mask,** which is a pattern of binary digits indicating which positions to plot along the line path. The linear mask 11111000, for instance, could be used to display a dashed line with a dash length of five pixels and an inter-dash spacing of three pixels. Pixel positions corresponding to the 1 bits are assigned the current color, and pixel positions corresponding to the 0 bits are displayed in the background color.

Plotting dashes with a fixed number of pixels results in unequal length dashes for different line orientations, as illustrated in Fig. 4-6. Both dashes shown are plotted with four pixels but the diagonal dash is longer by a factor of $\sqrt{2}$. For precision drawings, dash lengths should remain approximately constant for any line orientation. To accomplish this, we could adjust the pixel counts for the solid spans and inter-span spacing according to the line slope. In Fig. 4-6, we can display approximately equal length dashes by reducing the diagonal dash to three pixels.

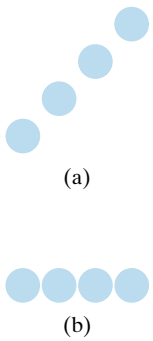


FIGURE 4–6 Unequal length dashes displayed with the same number of pixels.

Another method for maintaining dash length is to treat dashes as individual line segments. Endpoint coordinates for each dash are located and passed to the line routine, which then calculates pixel positions along the dash path.

Pen and Brush Options

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

Lines generated with pen (or brush) shapes can be displayed in various widths by changing the size of the mask. For example, the rectangular pen line in Fig. 4-9 could be narrowed with a 2 by 2 rectangular mask or widened with a 4 by 4 mask. Also, lines can be displayed with selected patterns by superimposing the pattern values onto the pen or brush mask.

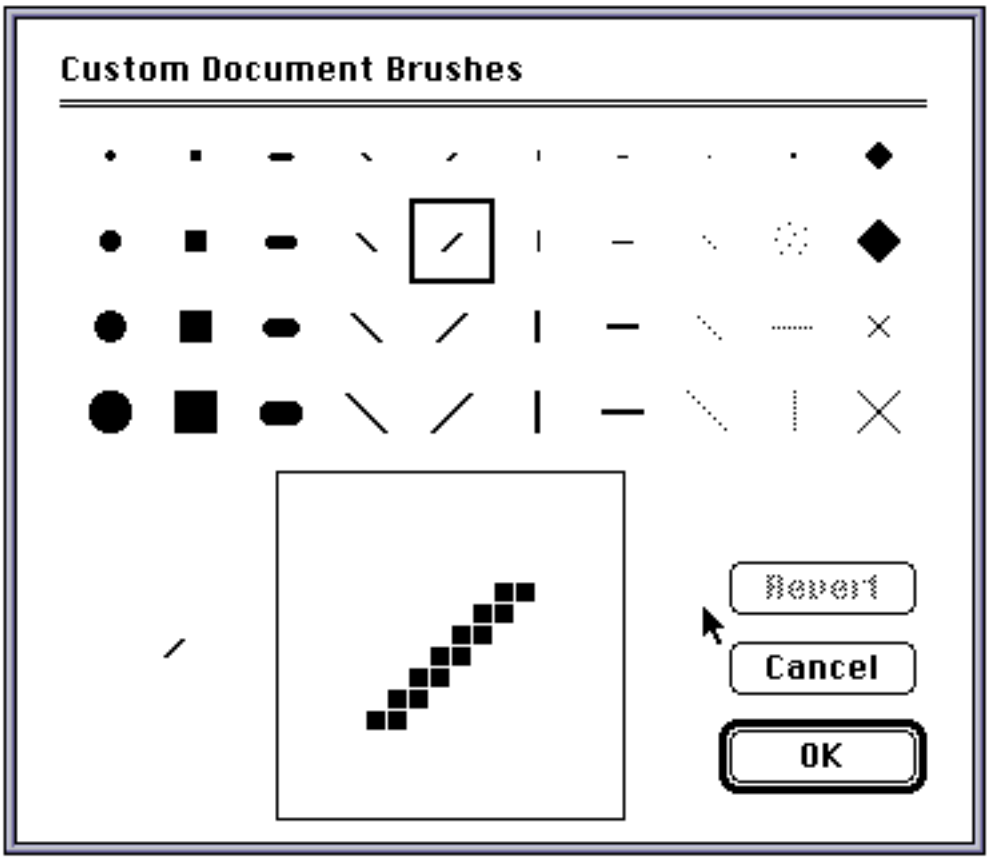


FIGURE 4–7 Pen and brush shapes for line display.

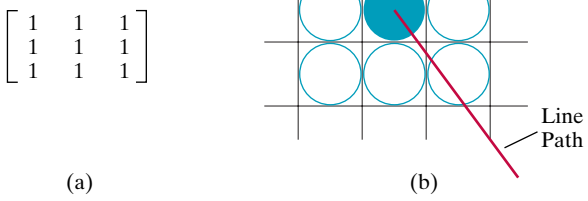


FIGURE 4–8 A pixel mask (a) for a rectangular pen, and the associated array of pixels (b) displayed by centering the mask over a specified pixel position.

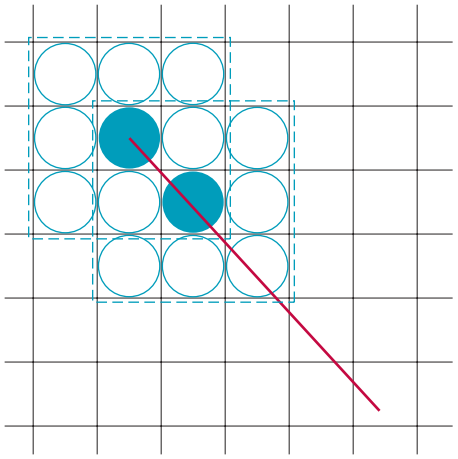


FIGURE 4–9 Generating a line with the pen shape of Fig. 4-8.

4-6 CURVE ATTRIBUTES

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

Raster curves of various widths can be displayed using the method of horizontal or vertical pixel spans. Where the magnitude of the curve slope is less than or equal to 1.0, we plot vertical spans; where the slope magnitude is greater than 1.0, we plot horizontal spans. Figure 4-10 demonstrates this method for displaying a circular arc of width 4 in the first quadrant. Using circle symmetry, we generate the circle path with vertical spans in the octant from $x = 0$ to $x = y$, and then reflect pixel positions about the line $y = x$ to obtain the remainder of the curve shown. Circle sections in the other quadrants are obtained by reflecting pixel positions in the first quadrant about the coordinate axes. The thickness of curves displayed with this method is again a function of curve slope. Circles, ellipses, and other curves will appear thinnest where the slope has a magnitude of 1.

Another method for displaying thick curves is to fill in the area between two parallel curve paths, whose separation distance is equal to the desired width. We could do this using the specified curve path as one boundary and setting up the second boundary either inside or outside the original curve path. This approach, however, shifts the original curve path either inward or outward, depending on which direction we choose for the second boundary. We can maintain the original curve position by setting the two boundary curves at a distance of one-half the width on either side of the specified curve path. An example of this approach is shown in Figure 4-11 for a circle segment with radius 16 and a specified width of 4. The boundary arcs are then set at a separation distance of 2 on either side of the radius of 16. To maintain the proper dimensions of the circular arc, as discussed in Section 3-13, we can set the radii for the concentric boundary arcs at $r = 14$ and $r = 17$. Although this method is accurate for generating thick circles,

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

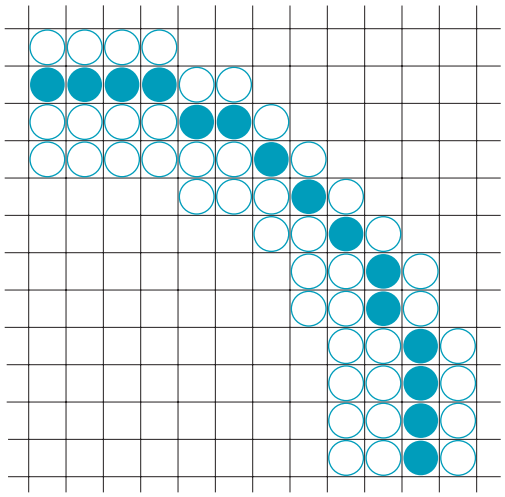


FIGURE 4–10 A circular arc of width 4 plotted with either vertical or horizontal pixel spans, depending on the slope.

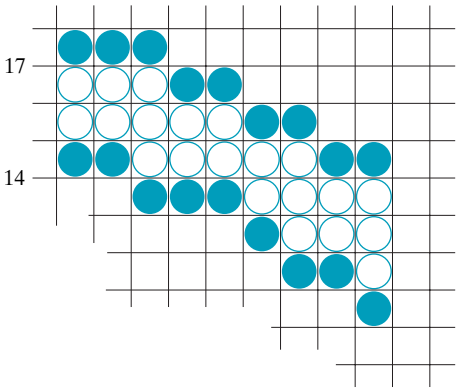


FIGURE 4–11 A circular arc of width 4 and radius 16 displayed by filling the region between two concentric arcs.

it provides, in general, only an approximation to the true area of other thick curves. For example, the inner and outer boundaries of a fat ellipse generated with this method do not have the same foci.

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

Pen (or brush) displays of curves are generated using the same techniques discussed for straight-line segments. We replicate a pen shape along the line path, as illustrated in Figure 4-13 for a circular arc in the first quadrant. Here, the center of the rectangular pen is moved to successive curve positions to produce the curve shape shown. Curves displayed with a rectangular pen in this manner will be thicker where the magnitude of the curve slope is 1. A uniform curve thickness can be displayed by rotating the rectangular pen to align it with the slope direction as we move around the curve or by using a circular pen shape. Curves drawn with pen and brush shapes can be displayed in different sizes and with superimposed patterns or simulated brush strokes.

Painting and drawing programs allow pictures to be constructed interactively by using a pointing device, such as a stylus and a graphics tablet, to sketch various curve shapes. Some examples of such curve patterns are shown in Fig. 4-14. An additional pattern option that can be provided in a paint package is the display of simulated brush strokes. Figure 4-15 illustrates some patterns that can be produced by modeling different types of brush strokes.

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

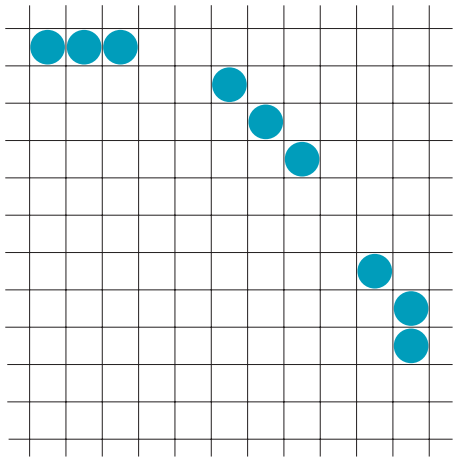


FIGURE 4–12 A dashed circular arc displayed with a dash span of 3 pixels and an inter-dash spacing of 2 pixels.

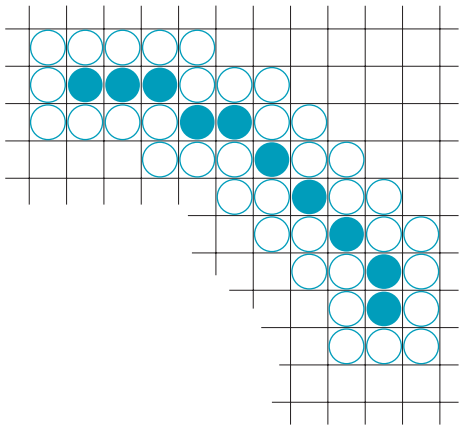


FIGURE 4–13 A circular arc displayed with a rectangular pen.

FIGURE 4–14 Curved lines drawn with a paint program using various shapes and patterns. From left to right, the brush shapes are square, round, diagonal line, dot pattern, and faded airbrush.

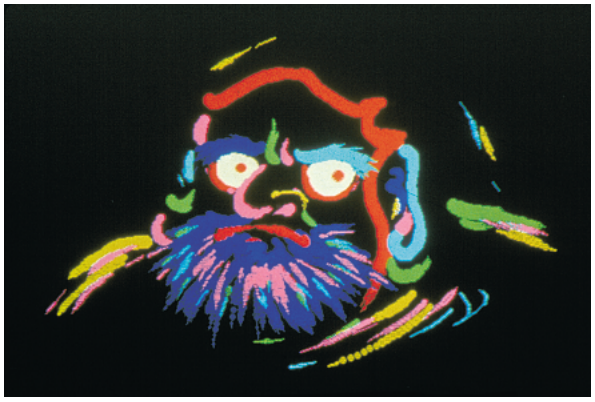


FIGURE 4–15 A daruma doll, a symbol of good fortune in Japan, drawn by computer artist Koichi Kozaki using a paintbrush system. Daruma dolls actually come without eyes. One eye is painted in when a wish is made, and the other is painted in when the wish comes true. (*Courtesy of Wacom Technology, Corp.*)

4-7 OpenGL POINT-ATTRIBUTE FUNCTIONS

The displayed color of a designated point position is controlled by the current color values in the state list. And a color is specified with either the `glColor` function or the `glIndex` function.

We set the size for an OpenGL point with

```
glPointSize (size);
```

and the point is then displayed as a square block of pixels. Parameter `size` is assigned a positive floating-point value, which is rounded to an integer (unless

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

the point is to be antialiased). The number of horizontal and vertical pixels in the display of the point is determined by parameter `size`. Thus a point size of 1.0 displays a single pixel, and a point size of 2.0 displays a 2 by 2 pixel array. If we activate the antialiasing features of OpenGL, the size of a displayed block of pixels will be modified to smooth the edges. The default value for point size is 1.0.

Attribute functions may be listed inside or outside of a `glBegin/glEnd` pair. For example, the following code segment plots three points in varying colors and sizes. The first is a standard-size red point, the second is a double-size green point, and the third is a triple-size blue point.

```
glColor3f (1.0, 0.0, 0.0);
glBegin (GL_POINTS);
   glVertex2i (50, 100);
   glPointSize (2.0);
   glColor3f (0.0, 1.0, 0.0);
   glVertex2i (75, 150);
   glPointSize (3.0);
   glColor3f (0.0, 0.0, 1.0);
   glVertex2i (100, 200);
glEnd ( );
```

4-8 OpenGL LINE-ATTRIBUTE FUNCTIONS

We can control the appearance of a straight-line segment in OpenGL with three attribute settings: line color, line width, and line style. We have already seen how to make a color selection, and OpenGL provides a function for setting the width of a line and another function for specifying a line style, such as a dashed or dotted line.

OpenGL Line-Width Function

Line width is set in OpenGL with the function

```
glLineWidth (width);
```

We assign a floating-point value to parameter `width`, and this value is rounded to the nearest nonnegative integer. If the input value rounds to 0.0, the line is displayed with a standard width of 1.0, which is the default width. However, when antialiasing is applied to the line, its edges are smoothed to reduce the raster stair-step appearance and fractional widths are possible. Some implementations of the line-width function might support only a limited number of widths, and some might not support widths other than 1.0.

The OpenGL line-width function is implemented using the methods described in Section 4-5. That is, the magnitude of the horizontal and vertical separations of the line endpoints, Δx and Δy, are compared to determine whether to generate a thick line using vertical pixel spans or horizontal pixel spans.

OpenGL Line-Style Function

By default, a straight-line segment is displayed as a solid line. But we can also display dashed lines, dotted lines, or a line with a combination of dashes and dots. And we can vary the length of the dashes and the spacing between dashes

or dots. We set a current display style for lines with the OpenGL function:

```
glLineStipple (repeatFactor, pattern);
```

Parameter `pattern` is used to reference a 16-bit integer that describes how the line should be displayed. A 1 bit in the pattern denotes an "on" pixel position, and a 0 bit indicates an "off" pixel position. The pattern is applied to the pixels along the line path starting with the low-order bits in the pattern. The default pattern is 0xFFFF (each bit position has a value of 1), which produces a solid line. Integer parameter `repeatFactor` specifies how many times each bit in the pattern is to be repeated before the next bit in the pattern is applied. The default repeat value is 1.

With a polyline, a specified line-style pattern is not restarted at the beginning of each segment. It is applied continuously across all the segments, starting at the first endpoint of the polyline and ending at the final endpoint for the last segment in the series.

As an example of specifying a line style, suppose parameter `pattern` is assigned the hexadecimal representation 0x00FF and the repeat factor is 1. This would display a dashed line with eight pixels in each dash and eight pixel positions that are "off" (an eight-pixel space) between two dashes. Also, since low-order bits are applied first, a line begins with an eight-pixel dash starting at the first endpoint. This dash is followed by an eight-pixel space, then another eight-pixel dash, and so forth, until the second endpoint position is reached.

Before a line can be displayed in the current line-style pattern, we must activate the line-style feature of OpenGL. We accomplish this with the following function.

```
glEnable (GL_LINE_STIPPLE);
```

If we forget to include this enable function, solid lines are displayed; that is, the default pattern 0xFFFF is used to display line segments. At any time, we can turn off the line-pattern feature with

```
glDisable (GL_LINE_STIPPLE);
```

This replaces the current line-style pattern with the default pattern (solid lines).

In the following program outline, we illustrate use of the OpenGL line-attribute functions by plotting three line graphs in different styles and widths. Figure 4-16 shows the data plots that could be generated by this program.

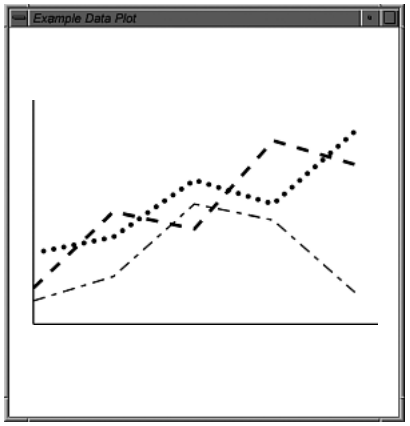


FIGURE 4–16 Plotting three data sets with three different OpenGL line styles and line widths: single-width dash-dot pattern, double-width dash pattern, and triple-width dot pattern.

```
/* Define a two-dimensional world-coordinate data type. */
typedef struct { float x, y; } wcPt2D;

wcPt2D dataPts [5];

void linePlot (wcPt2D dataPts [5])
{
   int k;

   glBegin (GL_LINE_STRIP);
      for (k = 0; k < 5; k++)
         glVertex2f (dataPts [k].x, dataPts [k].y);

   glFlush ( );

   glEnd ( );
}

/* Invoke a procedure here to draw coordinate axes.  */

glEnable (GL_LINE_STIPPLE);

/* Input first set of (x, y) data values. */
glLineStipple (1, 0x1C47);    // Plot a dash-dot, standard-width polyline.
linePlot (dataPts);

/* Input second set of (x, y) data values. */
glLineStipple (1, 0x00FF);    // Plot a dashed, double-width polyline.
glLineWidth (2.0);
linePlot (dataPts);

/* Input third set of (x, y) data values. */
glLineStipple (1, 0x0101);    // Plot a dotted, triple-width polyline.
glLineWidth (3.0);
linePlot (dataPts);

glDisable (GL_LINE_STIPPLE);
```

Other OpenGL Line Effects

In addition to specifying width, style, and a solid color, we can display lines with color gradations. For example, we can vary the color along the path of a solid line by assigning a different color to each line endpoint as we define the line. In the following code segment we illustrate this by assigning a blue color to one endpoint of a line and a red color to the other endpoint. The solid line is then displayed as a linear interpolation of the colors at the two endpoints.

```
glShadeModel (GL_SMOOTH);

glBegin (GL_LINES);
   glColor3f (0.0, 0.0, 1.0);
   glVertex2i (50, 50);
   glColor3f (1.0, 0.0, 0.0);
   glVertex2i (250, 250);
glEnd ( );
```

Function `glShadeModel` can also be given the argument `GL_FLAT`. In that case, the line segment would have been displayed in a single color: the color of the second endpoint, (250, 250). That is, we would have generated a red line. Actually, `GL_SMOOTH` is the default, so we would generate a smoothly interpolated color line segment even if we did not include this function in our code.

We can produce other effects by displaying adjacent lines that have different colors and patterns. And we can also make use of the color-blending features of OpenGL by superimposing lines or other objects with varying alpha values. A brush stroke, and other painting effects, can be simulated with a pixelmap and color blending. The pixelmap can then be moved interactively to generate line segments. Individual pixels in the pixmap can be assigned different alpha values to display lines as brush or pen strokes.

4-9 FILL-AREA ATTRIBUTES

Most graphics packages limit fill areas to polygons, because they are described with linear equations. A further restriction requires fill areas to be convex polygons, so that scan lines do not intersect more than two boundary edges. However, in general, we can fill any specified regions, including circles, ellipses, and other objects with curved boundaries. And applications systems, such as paint programs, provide fill options for arbitrarily shaped regions.

There are two basic procedures for filling an area on raster systems, once the definition of the fill region has been mapped to pixel coordinates. One procedure first determines the overlap intervals for scan lines that cross the area. Then, pixel positions along these overlap intervals are set to the fill color. Another method for area filling is to start from a given interior position and "paint" outward, pixel-by-pixel, from this point until we encounter specified boundary conditions. The scan-line approach is usually applied to simple shapes such as circles or regions with polyline boundaries, and general graphics packages use this fill method. Fill algorithms that use a starting interior point are useful for filling areas with more complex boundaries and in interactive painting systems.

Fill Styles

A basic fill-area attribute provided by a general graphics library is the display style of the interior. We can display a region with a single color, a specified fill pattern, or in a "hollow" style by showing only the boundary of the region. These three fill styles are illustrated in Fig. 4-17. We can also fill selected regions of a scene using various brush styles, color-blending combinations, or textures. Other options include specifications for the display of the boundaries of a fill area. For polygons, we could show the edges in different colors, widths, and styles. And we can select different display attributes for the front and back faces of a region.

Fill patterns can be defined in rectangular color arrays that list different colors for different positions in the array. Or, a fill pattern could be specified as a bit array that indicates which relative positions are to be displayed in a single selected color. An array specifying a fill pattern is a *mask* that is to be applied to the display area. Some graphics systems provide an option for selecting an arbitrary initial position for overlaying the mask. From this starting position, the mask is replicated in the horizontal and vertical directions until the display area is filled with nonoverlapping copies of the pattern. Where the pattern overlaps

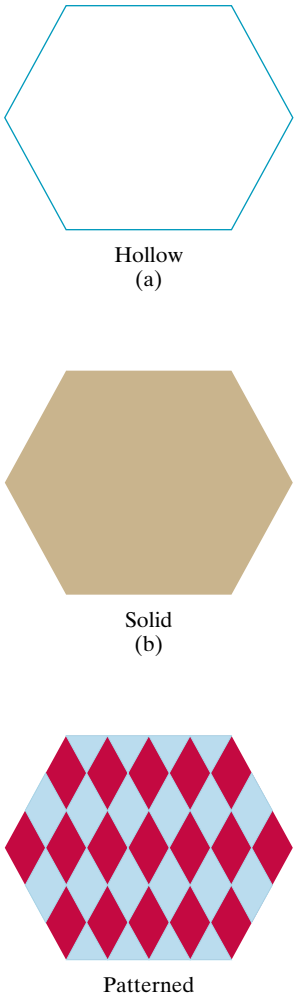


FIGURE 4–17 Basic polygon fill styles.

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

Diagonal
Hatch Fill

Diagonal
Crosshatch Fill

FIGURE 4-18 Areas filled with hatch patterns.

specified fill areas, the array pattern indicates which pixels should be displayed in a particular color. This process of filling an area with a rectangular pattern is called **tiling,** and a rectangular fill pattern is sometimes referred to as a **tiling pattern.** Sometimes, predefined fill patterns are available in a system, such as the *hatch* fill patterns shown in Fig. 4-18.

We can implement a pattern fill by determining where the pattern overlaps those scan lines that cross a fill area. Beginning from a specified start position for a pattern fill, we map the rectangular patterns vertically across scan lines and horizontally across pixel positions on the scan lines. Each replication of the pattern array is performed at intervals determined by the width and height of the mask. Where the pattern overlaps the fill area, pixel colors are set according to the values stored in the mask.

Hatch fill could be applied to regions by drawing sets of line segments to display either single hatching or crosshatching. Spacing and slope for the hatch lines could be set as parameters in a hatch table. Alternatively, hatch fill can be specified as a pattern array that produces sets of diagonal lines.

A reference point (xp, yp) for the starting position of a fill pattern can be set at any convenient position, inside or outside the fill region. For instance, the reference point could be set at a polygon vertex. Or the reference point could be chosen as the lower left corner of the bounding rectangle (or bounding box) determined by the coordinate extents of the region. To simplify selection of the reference coordinates, some packages always use the coordinate origin of the display window as the pattern start position. Always setting (xp, yp) at the coordinate origin also simplifies the tiling operations when each element of a pattern is to be mapped to a single pixel. For example, if the row positions in the pattern array are referenced from bottom to top, starting with the value 1, a color value is then assigned to pixel position (x, y) in screen coordinates from pattern position $(y \bmod ny + 1, x \bmod nx + 1)$. Here, ny and nx specify the number of rows and number of columns in the pattern array. Setting the pattern start position at the coordinate origin, however, effectively attaches the pattern fill to the screen background, rather than to the fill regions. Adjacent or overlapping areas filled with the same pattern would show no apparent boundary between the areas. Also, repositioning and refilling an object with the same pattern can result in a shift in the assigned pixel values over the object interior. A moving object would appear to be transparent against a stationary pattern background, instead of moving with a fixed interior pattern.

Color-Blended Fill Regions

It is also possible to combine a fill pattern with background colors in various ways. A pattern could be combined with background colors using a *transparency factor* that determines how much of the background should be mixed with the object color. Or we could use simple logical or replace operations. Figure 4-19 demonstrates how logical and replace operations would combine a 2 by 2 fill pattern with a background pattern for a binary (black-and-white) system.

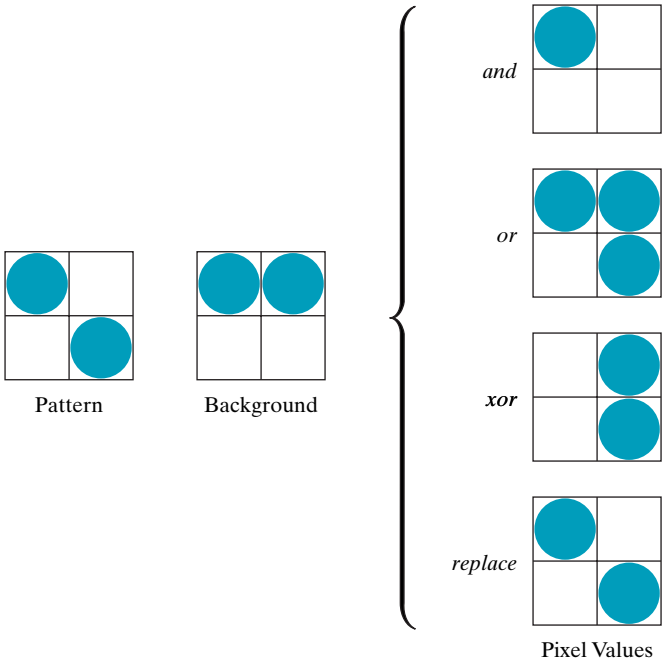


FIGURE 4–19 Combining a fill pattern with a background pattern using logical operations *and*, *or*, and *xor* (*exclusive or*), and using simple replacement.

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

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

$$\mathbf{P} = t\mathbf{F} + (1 - t)\mathbf{B} \tag{4-2}$$

where the transparency factor t has a value between 0 and 1 for each pixel. For values of t less than 0.5, the background color contributes more to the interior color of the region than does the fill color. Vector Eq. 4-2 holds for each RGB component of the colors, with

$$\mathbf{P} = (P_R, P_G, P_B), \qquad \mathbf{F} = (F_R, F_G, F_B), \qquad \mathbf{B} = (B_R, B_G, B_B) \tag{4-3}$$

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

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

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

where $k = R$, G, or B; and $F_k \neq B_k$. Theoretically, parameter t has the same value for each RGB component, but the round-off calculations to obtain integer codes can result in different values of t for different components. We can minimize this round-off error by selecting the component with the largest difference between **F** and **B**. This value of t is then used to mix the new fill color **NF** with the background color. We can accomplish this mixing using either a modified flood-fill or boundary-fill procedure, as described in Section 4-13.

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

$$\mathbf{P} = t_0\mathbf{F} + t_1\mathbf{B_1} + (1 - t_0 - t_1)\mathbf{B_2} \tag{4-5}$$

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

4-10 GENERAL SCAN-LINE POLYGON-FILL ALGORITHM

A scan-line fill of a region is performed by first determining the intersection positions of the boundaries of the fill region with the screen scan lines. Then the fill colors are applied to each section of a scan line that lies within the interior of the fill region. The scan-line fill algorithm identifies the same interior regions as the odd-even rule (Section 3-15). The simplest area to fill is a polygon, because each scan-line intersection point with a polygon boundary is obtained by solving a pair of simultaneous linear equations, where the equation for the scan line is simply $y =$ constant.

Figure 4-20 illustrates the basic scan-line procedure for a solid-color fill of a polygon. For each scan line that crosses the polygon, the edge intersections are sorted from left to right, and then the pixel positions between, and including, each intersection pair are set to the specified fill color. In the example of Fig. 4-20, the four pixel intersection positions with the polygon boundaries define two stretches of interior pixels. Thus, the fill color is applied to the five pixels from $x = 10$ to $x = 14$ and to the seven pixels from $x = 18$ to $x = 24$. If a pattern fill is to be applied to the polygon, then the color for each pixel along a scan line is determined from its overlap position with the fill pattern.

However, the scan-line fill algorithm for a polygon is not quite as simple as Fig. 4-20 might suggest. Whenever a scan line passes through a vertex, it intersects two polygon edges at that point. In some cases, this can result in an odd number of boundary intersections for a scan line. Figure 4-21 shows two scan lines that cross a polygon fill area and intersect a vertex. Scan line y' intersects an even number of edges, and the two pairs of intersection points along this scan line correctly identify the interior pixel spans. But scan line y intersects five polygon edges. To

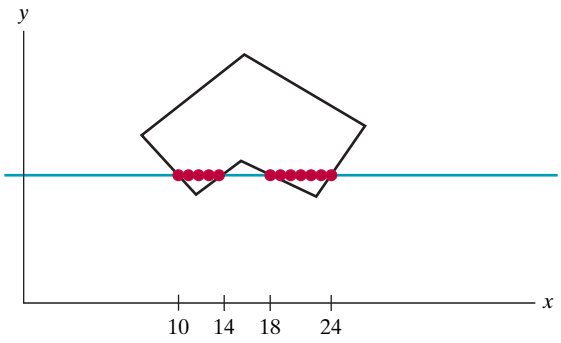


FIGURE 4–20 Interior pixels along a scan line passing through a polygon fill area.

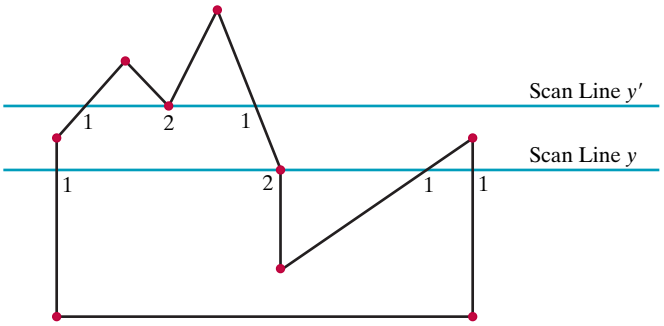


FIGURE 4–21 Intersection points along scan lines that intersect polygon vertices. Scan line y generates an odd number of intersections, but scan line y' generates an even number of intersections that can be paired to identify correctly the interior pixel spans.

identify the interior pixels for scan line y, we must count the vertex intersection as only one point. Thus, as we process scan lines, we need to distinguish between these cases.

We can detect the topological difference between scan line y and scan line y' in Fig. 4-21 by noting the position of the intersecting edges relative to the scan line. For scan line y, the two edges sharing an intersection vertex are on opposite sides of the scan line. But for scan line y', the two intersecting edges are both above the scan line. Thus, a vertex that has adjoining edges on opposite sides of an intersecting scan line should be counted as just one boundary intersection point. We can identify these vertices by tracing around the polygon boundary in either clockwise or counterclockwise order and observing the relative changes in vertex y coordinates as we move from one edge to the next. If the three endpoint y values of two consecutive edges monotonically increase or decrease, we need to count the shared (middle) vertex as a single intersection point for the scan line passing through that vertex. Otherwise, the shared vertex represents a local extremum (minimum or maximum) on the polygon boundary, and the two edge intersections with the scan line passing through that vertex can be added to the intersection list.

One method for implementing the adjustment to the vertex-intersection count is to shorten some polygon edges to split those vertices that should be counted as one intersection. We can process nonhorizontal edges around the polygon boundary in the order specified, either clockwise or counterclockwise. As we process each edge, we can check to determine whether that edge and the next nonhorizontal edge have either monotonically increasing or decreasing endpoint y values. If so, the lower edge can be shortened to ensure that only one intersection point is generated for the scan line going through the common vertex joining the two edges. Figure 4-22 illustrates shortening of an edge. When the endpoint y coordinates of the two edges are increasing, the y value of the upper endpoint for the current edge is decreased by 1, as in Fig. 4-22 (a). When the endpoint y values are monotonically decreasing, as in Fig. 4-22 (b), we decrease the y coordinate of the upper endpoint of the edge following the current edge.

Typically, certain properties of one part of a scene are related in some way to the properties in other parts of the scene, and these **coherence properties** can be used in computer-graphics algorithms to reduce processing. Coherence methods

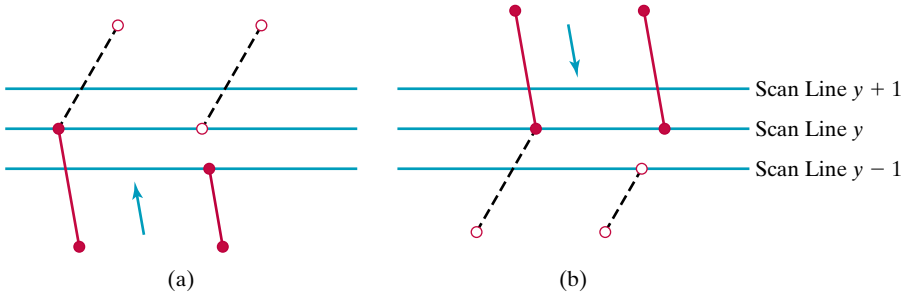


FIGURE 4–22 Adjusting endpoint y values for a polygon, as we process edges in order around the polygon perimeter. The edge currently being processed is indicated as a solid line. In (a), the y coordinate of the upper endpoint of the current edge is decreased by 1. In (b), the y coordinate of the upper endpoint of the next edge is decreased by 1.

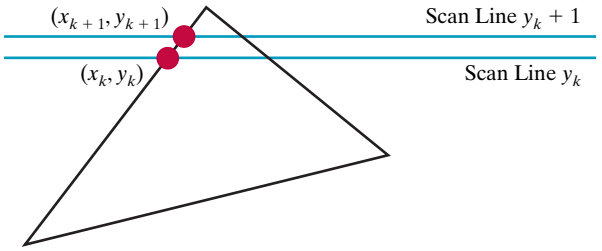


FIGURE 4–23 Two successive scan lines intersecting a polygon boundary.

often involve incremental calculations applied along a single scan line or between successive scan lines. For example, in determining fill-area edge intersections, we can set up incremental coordinate calculations along any edge by exploiting the fact that the slope of the edge is constant from one scan line to the next. Figure 4-23 shows two successive scan lines crossing the left edge of a triangle. The slope of this edge can be expressed in terms of the scan-line intersection coordinates:

$$m = \frac{y_{k+1} - y_k}{x_{k+1} - x_k} \tag{4-6}$$

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

$$y_{k+1} - y_k = 1 \tag{4-7}$$

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

$$x_{k+1} = x_k + \frac{1}{m} \tag{4-8}$$

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

An obvious parallel implementation of the fill algorithm is to assign each scan line that crosses the polygon to a separate processor. Edge intersection calculations are then performed independently. Along an edge with slope m, the intersection

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

x_k value for scan line k above the initial scan line can be calculated as

$$x_k = x_0 + \frac{k}{m} \tag{4-9}$$

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

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

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

$$x_{k+1} = x_k + \frac{\Delta x}{\Delta y} \tag{4-10}$$

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

As an example of this integer-incrementing scheme, suppose we have an edge with slope $m = \frac{7}{3}$. At the initial scan line, we set the counter to 0 and the counter increment to 3. As we move up to the next three scan lines along this edge, the counter is successively assigned the values 3, 6, and 9. On the third scan line above the initial scan line, the counter now has a value greater than 7. So we increment the x intersection coordinate by 1, and reset the counter to the value $9 - 7 = 2$. We continue determining the scan-line intersections in this way until we reach the upper endpoint of the edge. Similar calculations are carried out to obtain intersections for edges with negative slopes.

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

To efficiently perform a polygon fill, we can first store the polygon boundary in a *sorted edge table* that contains all the information necessary to process the scan lines efficiently. Proceeding around the edges in either a clockwise or a counterclockwise order, we can use a bucket sort to store the edges, sorted on the smallest y value of each edge, in the correct scan-line positions. Only nonhorizontal edges are entered into the sorted edge table. As the edges are processed, we can also shorten certain edges to resolve the vertex-intersection question. Each entry in

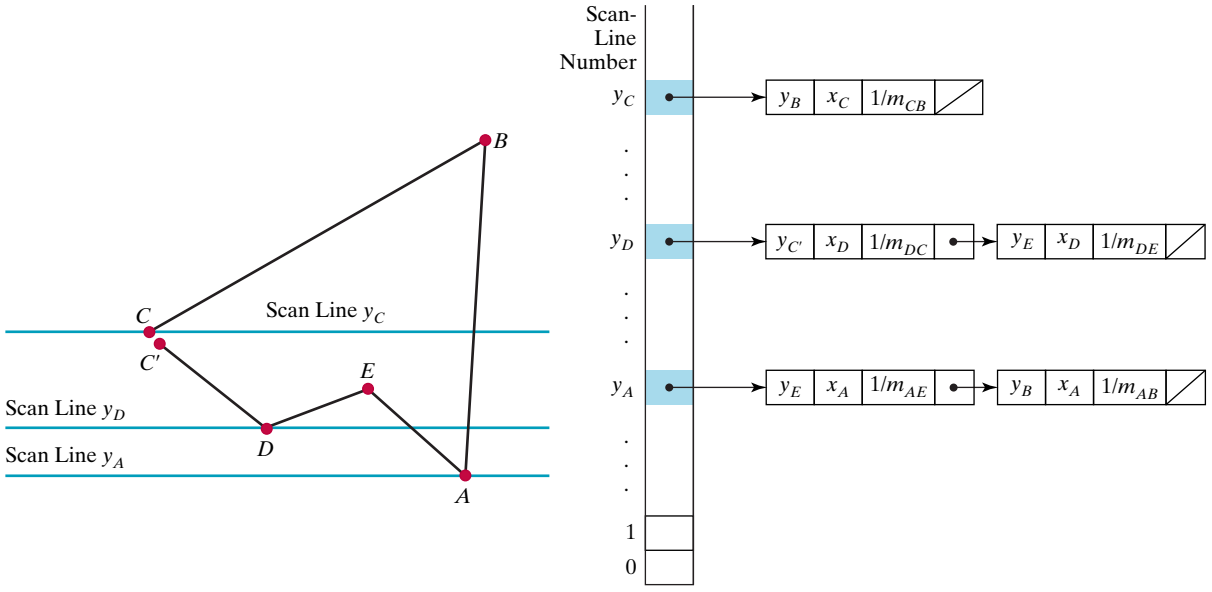


FIGURE 4–24 A polygon and its sorted edge table, with edge $\overline{\mathbf{DC}}$ shortened by one unit in the y direction.

the table for a particular scan line contains the maximum y value for that edge, the x-intercept value (at the lower vertex) for the edge, and the inverse slope of the edge. For each scan line, the edges are in sorted order from left to right. Figure 4-24 shows a polygon and the associated sorted edge table.

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

Implementation of edge-intersection calculations can be facilitated by storing Δx and Δy values in the sorted edge list. Also, to ensure that we correctly fill the interior of specified polygons, we can apply the considerations discussed in Section 3-13. For each scan line, we fill in the pixel spans for each pair of x intercepts starting from the leftmost x intercept value and ending at one position before the rightmost x intercept. And each polygon edge can be shortened by one unit in the y direction at the top endpoint. These measures also guarantee that pixels in adjacent polygons will not overlap.

4-11 SCAN-LINE FILL OF CONVEX POLYGONS

When we apply a scan-line fill procedure to a convex polygon, there can be no more than a single interior span for each screen scan line. So we need to process the polygon edges only until we have found two boundary intersections for each scan line crossing the polygon interior.

The general polygon scan-line algorithm discussed in the preceding section can be simplified considerably for convex-polygon fill. We again use coordinate extents to determine which edges cross a scan line. Intersection calculations with these edges then determine the interior pixel span for that scan line, where any vertex crossing is counted as a single boundary intersection point. When a scan

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

line intersects a single vertex (at an apex, for example), we plot only that point. Some graphics packages further restrict fill areas to be triangles. This makes filling even easier, because each triangle has just three edges to process.

4-12 SCAN-LINE FILL FOR REGIONS WITH CURVED BOUNDARIES

Since an area with curved boundaries is described with nonlinear equations, a scan-line fill generally takes more time than a polygon scan-line fill. We can use the same general approach detailed in Section 4-10, but the boundary intersection calculations are performed with curve equations. And the slope of the boundary is continuously changing, so we cannot use the straightforward incremental calculations that are possible with straight-line edges.

For simple curves such as circles or ellipses, we can apply fill methods similar to those for convex polygons. Each scan line crossing a circle or ellipse interior has just two boundary intersections. And we can determine these two intersection points along the boundary of a circle or an ellipse using the incremental calculations in the midpoint method. Then we simply fill in the horizontal pixel spans from one intersection point to the other. Symmetries between quadrants (and between octants for circles) are used to reduce the boundary calculations.

FIGURE 4–25 Interior fill of an elliptical arc.

Similar methods can be used to generate a fill area for a curve section. For example, an area bounded by an elliptical arc and a straight line section (Fig. 4-25) can be filled using a combination of curve and line procedures. Symmetries and incremental calculations are exploited whenever possible to reduce computations.

Filling other curve areas can involve considerably more processing. We could use similar incremental methods in combination with numerical techniques to determine the scan-line intersections, but usually such curve boundaries are approximated with straight-line segments.

4-13 FILL METHODS FOR AREAS WITH IRREGULAR BOUNDARIES

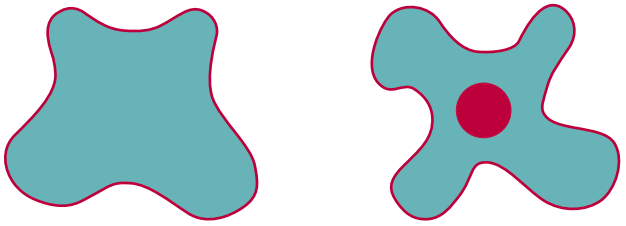
Another approach for filling a specified area is to start at an inside position and "paint" the interior, point by point, out to the boundary. This is a particularly useful technique for filling areas with irregular borders, such as a design created with a paint program. Generally, these methods require an input starting position inside the area to be filled and some color information about either the boundary or the interior.

We can fill irregular regions with a single color or with a color pattern. For a pattern fill, we overlay a color mask, as discussed in Section 4-9. As each pixel within the region is processed, its color is determined by the corresponding values in the overlaid pattern.

Boundary-Fill Algorithm

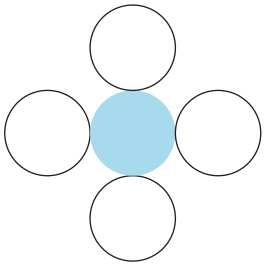
If the boundary of some region is specified in a single color, we can fill the interior of this region, pixel by pixel, until the boundary color is encountered. This method,

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

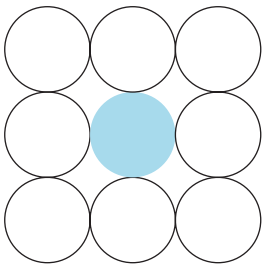


(a) (b)

FIGURE 4–26 Example color boundaries for a boundary-fill procedure.



(a)



(b)

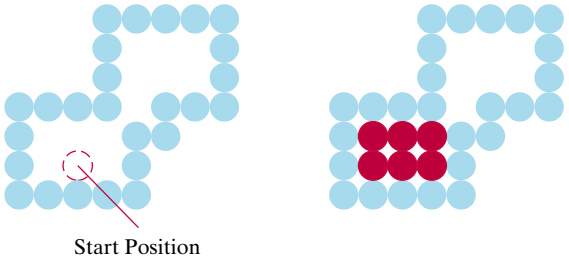
FIGURE 4–27 Fill methods applied to a 4-connected area (a) and to an 8-connected area (b). Hollow circles represent pixels to be tested from the current test position, shown as a solid color.

called the **boundary-fill algorithm,** is employed in interactive painting packages, where interior points are easily selected. Using a graphics tablet or other interactive device, an artist or designer can sketch a figure outline, select a fill color from a color menu, specify the area boundary color, and pick an interior point. The figure interior is then painted in the fill color. Both inner and outer boundaries can be set up to define an area for boundary fill, and Fig. 4-26 illustrates examples for specifying color regions.

Basically, a boundary-fill algorithm starts from an interior point (x, y) and tests the color of neighboring positions. If a tested position is not displayed in the boundary color, its color is changed to the fill color and its neighbors are tested. This procedure continues until all pixels are processed up to the designated boundary color for the area.

Figure 4-27 shows two methods for processing neighboring pixels from a current test position. In Fig. 4-27(a), four neighboring points are tested. These are the pixel positions that are right, left, above, and below the current pixel. Areas filled by this method are called **4-connected.** The second method, shown in Fig. 4-27(b), is used to fill more complex figures. Here the set of neighboring positions to be tested includes the four diagonal pixels, as well as those in the cardinal directions. Fill methods using this approach are called **8-connected.** An 8-connected boundary-fill algorithm would correctly fill the interior of the area defined in Fig. 4-28, but a 4-connected boundary-fill algorithm would only fill part of that region.

The following procedure illustrates a recursive method for painting a 4-connected area with a solid color, specified in parameter `fillColor`, up to a boundary color specified with parameter `borderColor`. We can extend this



(a) (b)

FIGURE 4–28 The area defined within the color boundary (a) is only partially filled in (b) using a 4-connected boundary-fill algorithm.

procedure to fill an 8-connected region by including four additional statements to test the diagonal positions $(x \pm 1, y \pm 1)$.

```
void boundaryFill4 (int x, int y, int fillColor, int borderColor)
{
   int interiorColor;

   /* Set current color to fillColor, then perform following oprations. */
   getPixel (x, y, interiorColor);
   if ((interiorColor != borderColor) && (interiorColor != fillColor)) {
      setPixel (x, y);    // Set color of pixel to fillColor.
      boundaryFill4 (x + 1, y , fillColor, borderColor);
      boundaryFill4 (x - 1, y , fillColor, borderColor);
      boundaryFill4 (x , y + 1, fillColor, borderColor);
      boundaryFill4 (x , y - 1, fillColor, borderColor)
   }
}
```

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

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

An example of how pixel spans could be filled using this approach is illustrated for the 4-connected fill region in Figure 4-29. In this example, we first process scan lines successively from the start line to the top boundary. After all upper scan lines are processed, we fill in the pixel spans on the remaining scan lines in order down to the bottom boundary. The leftmost pixel position for each horizontal span is located and stacked, in left to right order across successive scan lines, as shown in Fig. 4-29. In (a) of this figure, the initial span has been filled, and starting positions 1 and 2 for spans on the next scan lines (below and above) are stacked. In Fig. 4-29(b), position 2 has been unstacked and processed to produce the filled span shown, and the starting pixel (position 3) for the single span on the next scan line has been stacked. After position 3 is processed, the filled spans and stacked positions are as shown in Fig. 4-29(c). And Fig. 4-29(d) shows the filled pixels after processing all spans in the upper right of the specified area. Position 5 is next processed, and spans are filled in the upper left of the region; then position 4 is picked up to continue the processing for the lower scan lines.

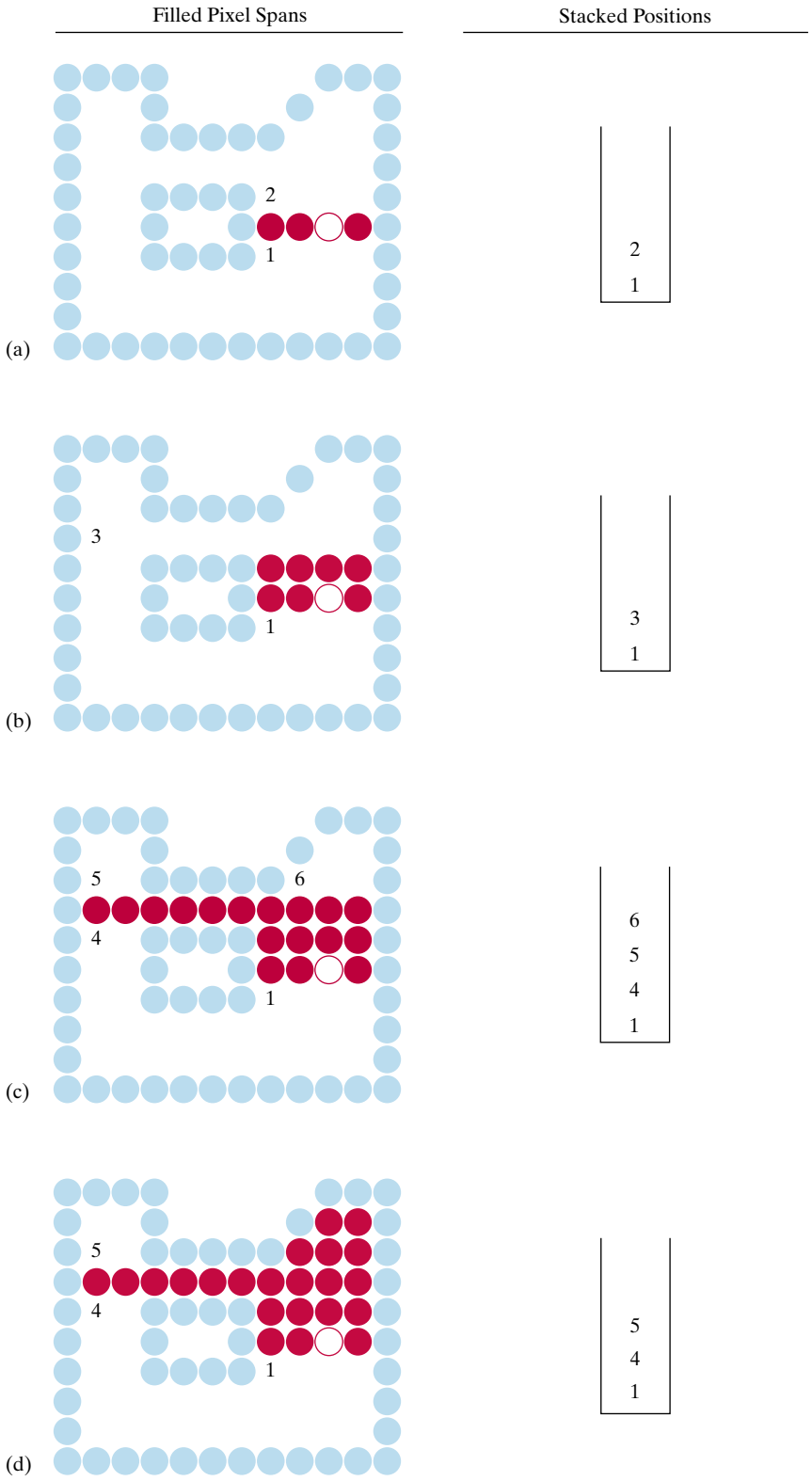


FIGURE 4–29 Boundary fill across pixel spans for a 4-connected area: (a) Initial scan line with a filled pixel span, showing the position of the initial point (hollow) and the stacked positions for pixel spans on adjacent scan lines. (b) Filled pixel span on the first scan line above the initial scan line and the current contents of the stack. (c) Filled pixel spans on the first two scan lines above the initial scan line and the current contents of the stack. (d) Completed pixel spans for the upper-right portion of the defined region and the remaining stacked positions to be processed.

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

Flood-Fill Algorithm

Sometimes we want to fill in (or recolor) an area that is not defined within a single color boundary. Figure 4-30 shows an area bordered by several different color regions. We can paint such areas by replacing a specified interior color instead of searching for a particular boundary color. This fill procedure is called a **flood-fill algorithm.** We start from a specified interior point (x, y) and reassign all pixel values that are currently set to a given interior color with the desired fill color. If the area we want to paint has more than one interior color, we can first reassign pixel values so that all interior points have the same color. Using either a 4-connected or 8-connected approach, we then step through pixel positions until all interior points have been repainted. The following procedure flood fills a 4-connected region recursively, starting from the input position.

FIGURE 4-30 An area defined within multiple color boundaries.

```
void floodFill4 (int x, int y, int fillColor, int interiorColor)
{
   int color;

   /* Set current color to fillColor, then perform following operations. */
   getPixel (x, y, color);
   if (color = interiorColor) {
      setPixel (x, y);    // Set color of pixel to fillColor.
      floodFill4 (x + 1, y, fillColor, interiorColor);
      floodFill4 (x - 1, y, fillColor, interiorColor);
      floodFill4 (x, y + 1, fillColor, interiorColor);
      floodFill4 (x, y - 1, fillColor, interiorColor)
   }
}
```

We can modify the above procedure to reduce the storage requirements of the stack by filling horizontal pixel spans, as discussed for the boundary-fill algorithm. In this approach, we stack only the beginning positions for those pixel spans having the value `interiorColor`. The steps in this modified flood-fill algorithm are similar to those illustrated in Fig. 4-29 for a boundary fill. Starting at the first position of each span, the pixel values are replaced until a value other than `interiorColor` is encountered.

4-14 OpenGL FILL-AREA ATTRIBUTE FUNCTIONS

In the OpenGL graphics package, fill-area routines are available for convex polygons only. We generate displays of filled convex polygons in four steps:

(1) Define a fill pattern.
(2) Invoke the polygon-fill routine.
(3) Activate the polygon-fill feature of OpenGL.
(4) Describe the polygons to be filled.

A polygon fill pattern is displayed up to and including the polygon edges. Thus, there are no boundary lines around the fill region unless we specifically add them to the display.

In addition to specifying a fill pattern for a polygon interior, there are a number of other options available. One option is to display a hollow polygon, where no interior color or pattern is applied and only the edges are generated. A hollow polygon is equivalent to the display of a closed polyline primitive. Another option is to show the polygon vertices, with no interior fill and no edges. Also, we designate different attributes for the front and back faces of a polygon fill area.

OpenGL Fill-Pattern Function

By default, a convex polygon is displayed as a solid-color region, using the current color setting. To fill the polygon with a pattern in OpenGL, we use a 32-bit by 32-bit mask. A value of 1 in the mask indicates that the corresponding pixel is to be set to the current color, and a 0 leaves the value of that frame-buffer position unchanged. The fill pattern is specified in unsigned bytes using the OpenGL data type `GLubyte`, just as we did with the `glBitmap` function. We define a bit pattern with hexadecimal values as, for example,

```
GLubyte fillPattern [ ] = {
    0xff, 0x00, 0xff, 0x00, ... };
```

The bits must be specified starting with the bottom row of the pattern, and continuing up to the topmost row (32) of the pattern, as we did with `bitShape` in Section 3-19. This pattern is replicated across the entire area of the display window, starting at the lower-left window corner, and specified polygons are filled where the pattern overlaps those polygons (Fig. 4-31).

Once we have set a mask, we can establish it as the current fill pattern with the function

```
glPolygonStipple (fillPattern);
```

Next, we need to enable the fill routines before we specify the vertices for the polygons that are to be filled with the current pattern. We do this with the statement

```
glEnable (GL_POLYGON_STIPPLE);
```

Similarly, we turn off pattern filling with

```
glDisable (GL_POLYGON_STIPPLE);
```

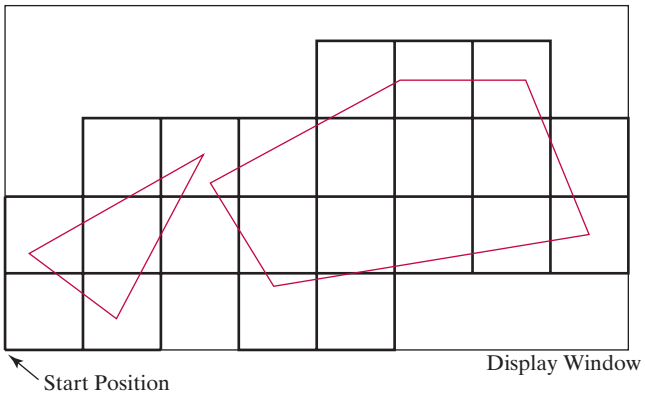


FIGURE 4-31 Tiling a rectangular fill pattern across a display window to fill two convex polygons.

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

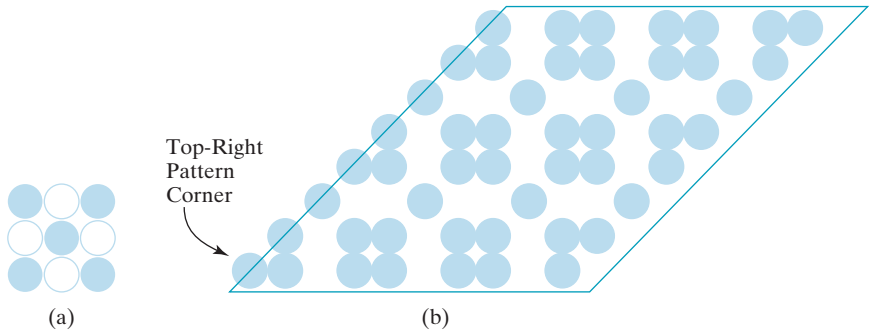


FIGURE 4–32 A 3 by 3 bit pattern (a) superimposed on a parallelogram to produce the fill area in (b), where the top-right corner of the pattern coincides with the lower-left corner of the parallelogram.

Figure 4-32 illustrates how a 3 by 3 bit pattern, repeated over a 32 by 32 bit mask, might be applied to fill a parallelogram.

OpenGL Texture and Interpolation Patterns

Another method for filling polygons is to use texture patterns, as discussed in Chapter 10. This can produce fill patterns that simulate the surface appearance of wood, brick, brushed steel, or some other material. Also, we can obtain an interpolation coloring of a polygon interior just as we did with the line primitive. To do this, we assign different colors to polygon vertices. Interpolation fill of a polygon interior is used to produce realistic displays of shaded surfaces under various lighting conditions.

As an example of an interpolation fill, the following code segment assigns either a blue, red, or green color to each of the three vertices of a triangle. The polygon fill is then a linear interpolation of the colors at the vertices.

```
glShadeModel (GL_SMOOTH);

glBegin (GL_TRIANGLES);
   glColor3f (0.0, 0.0, 1.0);
   glVertex2i (50, 50);
   glColor3f (1.0, 0.0, 0.0);
   glVertex2i (150, 50);
   glColor3f (0.0, 1.0, 0.0);
   glVertex2i (75, 150);
glEnd ( );
```

Of course, if a single color is set for the triangle as a whole, the polygon is filled with that one color. And if we change the argument in the `glShadeModel` function to `GL_FLAT` in this example, the polygon is filled with the last color specified (green). The value `GL_SMOOTH` is the default shading, but we can include that specification to remind us that the polygon is to be filled as an interpolation of the vertex colors.

OpenGL Wire-Frame Methods

We can also choose to show only polygon edges. This produces a wire-frame or hollow display of the polygon. Or we could display a polygon by only plotting a set of points at the vertex positions. These options are selected with the

function

```
glPolygonMode (face, displayMode);
```

We use parameter `face` to designate which face of the polygon we want to show as edges only or vertices only. This parameter is then assigned either `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`. Then, if we want only the polygon edges displayed for our selection, we assign the constant `GL_LINE` to parameter `displayMode`. To plot only the polygon vertex points, we assign the constant `GL_POINT` to parameter `displayMode`. A third option is `GL_FILL`. But this is the default display mode, so we usually only invoke `glPolygonMode` when we want to set attributes for the polygon edges or vertices.

Another option is to display a polygon with both an interior fill and a different color or pattern for its edges (or for its vertices). This is accomplished by specifying the polygon twice: once with parameter `displayMode` set to `GL_FILL` and then again with `displayMode` set to `GL_LINE` (or `GL_POINT`). For example, the following code section fills a polygon interior with a green color, and then the edges are assigned a red color.

```
glColor3f (0.0, 1.0, 0.0);
\* Invoke polygon-generating routine. */

glColor3f (1.0, 0.0, 0.0);
glPolygonMode (GL_FRONT, GL_LINE);
\* Invoke polygon-generating routine again. */
```

For a three-dimensional polygon (one that does not have all vertices in the xy plane), this method for displaying the edges of a filled polygon may produce gaps along the edges. This effect, sometimes referred to as **stitching,** is caused by differences between calculations in the scan-line fill algorithm and calculations in the edge line-drawing algorithm. As the interior of a three-dimensional polygon is filled, the depth value (distance from the xy plane) is calculated for each (x, y) position. But this depth value at an edge of the polygon is often not exactly the same as the depth value calculated by the line-drawing algorithm for the same (x, y) position. Therefore, when visibility tests are made, the interior fill color could be used instead of an edge color to display some points along the boundary of a polygon.

One way to eliminate the gaps along displayed edges of a three-dimensional polygon is to shift the depth values calculated by the fill routine so that they do not overlap with the edge depth values for that polygon. We do this with the following two OpenGL functions.

```
glEnable (GL_POLYGON_OFFSET_FILL);
glPolygonOffset (factor1, factor2);
```

The first function activates the offset routine for scan-line filling, and the second function is used to set a couple of floating-point values `factor1` and `factor2` that are used to calculate the amount of depth offset. The calculation for this depth offset is

$$\text{depthOffset} = \text{factor1} \cdot \text{maxSlope} + \text{factor2} \cdot \text{const} \qquad (4\text{-}11)$$

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

where maxSlope is the maximum slope of the polygon and const is an implementation constant. For a polygon in the xy plane, the slope is 0. Otherwise, the maximum slope is calculated as the change in depth of the polygon divided by either the change in x or the change in y. A typical value for the two factors is either 0.75 or 1.0, although some experimentation with the factor values is often necessary to produce good results. As an example of assigning values to offset factors, we can modify the previous code segment as follows:

```
glColor3f (0.0, 1.0, 0.0);
glEnable (GL_POLYGON_OFFSET_FILL);
glPolygonOffset (1.0, 1.0);
\* Invoke polygon-generating routine. */
glDisable (GL_POLYGON_OFFSET_FILL);

glColor3f (1.0, 0.0, 0.0);
glPolygonMode (GL_FRONT, GL_LINE);
\* Invoke polygon-generating routine again. */
```

Now the interior fill of the polygon is pushed a little farther away in depth, so that it does not interfere with the depth values of its edges. It is also possible to implement this method by applying the offset to the line-drawing algorithm, by changing the argument of the `glEnable` function to `GL_POLYGON_OFFSET_LINE`. In this case, we want to use negative factors to bring the edge depth values closer. And if we just wanted to display different color points at the vertex positions, instead of highlighted edges, the argument in the `glEnable` function would be `GL_POLYGON_OFFSET_POINT`.

Another method for eliminating the stitching effect along polygon edges is to use the OpenGL stencil buffer to limit the polygon interior filling so that it does not overlap the edges. But this approach is more complicated and generally slower, so the polygon depth-offset method is preferred.

To display a concave polygon using OpenGL routines, we must first split it into a set of convex polygons. We typically divide a concave polygon into a set of triangles, using the methods described in Section 3-15. Then we could display the concave polygon as a fill region by filling the triangles. Similarly, if we want to show only the polygon vertices, we plot the triangle vertices. But to display the original concave polygon in a wire-frame form, we cannot just set the display mode to `GL_LINE`, because that would show all the triangle edges that are interior to the original concave polygon (Fig. 4-33).

Fortunately, OpenGL provides a mechanism that allows us to eliminate selected edges from a wire-frame display. Each polygon vertex is stored with a one-bit flag that indicates whether or not that vertex is connected to the next vertex by a boundary edge. So all we need do is set that bit flag to "off" and the edge

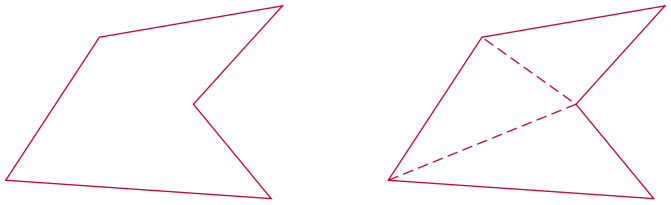


FIGURE 4–33 Dividing a concave polygon (a) into a set of triangles (b) produces triangle edges (dashed) that are interior to the original polygon.

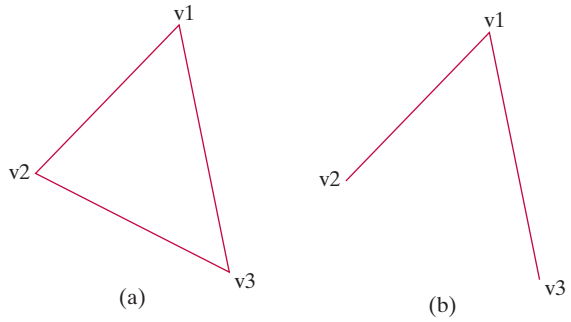


FIGURE 4-34 The triangle in (a) can be displayed as in (b) by setting the edge flag for vertex v2 to the value GL_FALSE, assuming that the vertices are specified in a counterclockwise order.

following that vertex will not be displayed. We set this flag for an edge with the following function.

```
glEdgeFlag (flag);
```

To indicate that a vertex does not precede a boundary edge, we assign the OpenGL constant GL_FALSE to parameter flag. This applies to all subsequently specified vertices until the next call to glEdgeFlag is made. The OpenGL constant GL_TRUE turns the edge flag back on again, which is the default. Function glEdgeFlag can be placed between glBegin/glEnd pairs. As an illustration of the use of an edge flag, the following code displays only two edges of the defined triangle (Fig. 4-34).

```
glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);

glBegin (GL_POLYGON);
   glVertex3fv (v1);
   glEdgeFlag (GL_FALSE);
   glVertex3fv (v2);
   glEdgeFlag (GL_TRUE);
   glVertex3fv (v3);
glEnd ( );
```

Polygon edge flags can also be specified in an array that could be combined or associated with a vertex array (Sections 3-17 and 4-3). The statements for creating an array of edge flags are

```
glEnableClientState (GL_EDGE_FLAG_ARRAY);

glEdgeFlagPointer (offset, edgeFlagArray);
```

Parameter offset indicates the number of bytes between the values for the edge flags in the array edgeFlagArray. The default value for parameter offset is 0.

OpenGL Front-Face Function

Although, by default, the ordering of polygon vertices controls the identification of front and back faces, we can independently label selected surfaces in a scene as front or back with the function

```
glFrontFace (vertexOrder);
```

If we set parameter vertexOrder to the OpenGL constant GL_CW, then a subsequently defined polygon with a clockwise ordering for its vertices is considered

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

to be front facing. This OpenGL feature can be used to swap faces of a polygon for which we have specified vertices in a clockwise order. The constant `GL_CCW` labels a counterclockwise ordering of polygon vertices as front facing, which is the default ordering.

4-15 CHARACTER ATTRIBUTES

We control the appearance of displayed characters with attributes such as font, size, color, and orientation. In many packages, attributes can be set both for entire character strings (text) and for individual characters that can be used for special purposes such as plotting a data graph.

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

Color settings for displayed text can be stored in the system attribute list and used by the procedures that generate character definitions in the frame buffer. When a character string is to be displayed, the current color is used to set pixel values in the frame buffer corresponding to the character shapes and positions.

We could adjust text size by scaling the overall dimensions (height and width) of characters or by scaling only the height or the width. Character size (height) is specified by printers and compositors in *points,* where 1 point is about 0.035146 centimeters (or 0.013837 inch, which is approximately $\frac{1}{72}$ inch). For example, the characters in this book are set in a 10-point font. Point measurements specify the size of the *body* of a character (Fig. 4-35), but different fonts with the same point specifications can have different character sizes, depending on the design of the typeface. The distance between the *bottomline* and the *topline* of the character body is the same for all characters in a particular size and typeface, but the body width may vary. *Proportionally spaced fonts* assign a smaller body width to narrow characters such as i, j, l, and f compared to broad characters such as *W* or *M*. *Character height* is defined as the distance between the *baseline* and the *capline* of characters. Kerned characters, such as f and j in Fig. 4-35, typically extend beyond the character body limits, and letters with descenders (g, j, p, q, y) extend below the baseline. Each character is positioned within the character body by a font designer in such a way that suitable spacing is attained along and between print lines when text is displayed with character bodies touching.

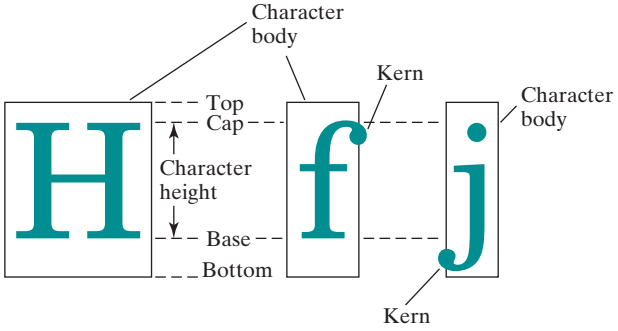


FIGURE 4–35 Examples of character bodies.

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

Height 1

Height 2

Height 3

FIGURE 4–36 Text strings displayed with different character-height settings and a constant width-to-height ratio.

width 0.5

width 1.0

width 2.0

FIGURE 4–37 Text strings displayed with varying sizes for the character widths and a fixed height.

Spacing 0.0

Spacing 0.5

Spacing 1.0

FIGURE 4–38 Text strings displayed with different character-spacing values.

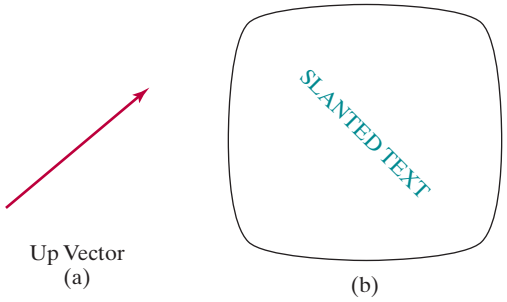


FIGURE 4–39 Direction of the up vector (a) controls the orientation of displayed text (b).

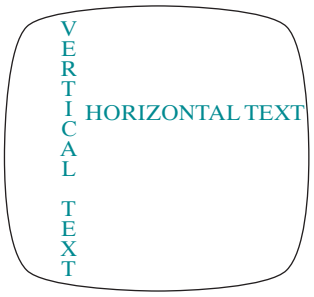


FIGURE 4–40 Text-path attributes can be set to produce horizontal or vertical arrangements of character strings.

Sometimes, text size is adjusted without changing the width-to-height ratio of characters. Figure 4-36 shows a character string displayed with three different character heights, while maintaining the ratio of width to height. Examples of text displayed with a constant height and varying widths are given in Fig. 4-37.

Spacing between characters is another attribute that can often be assigned to a character string. Figure 4-38 shows a character string displayed with three different settings for the intercharacter spacing.

The orientation for a character string can be set according to the direction of a **character up vector.** Text is then displayed so that the orientation of characters from baseline to capline is in the direction of the up vector. For example, with the direction of the up vector at 45°, text would be displayed as shown in Fig. 4-39. A procedure for orienting text could rotate characters so that the sides of character bodies, from baseline to capline, are aligned with the up vector. The rotated character shapes are then scan converted into the frame buffer.

It is useful in many applications to be able to arrange character strings vertically or horizontally. Examples of this are given in Fig. 4-40. We could also arrange the characters in a text string so that the string is displayed forward or backward. Examples of text displayed with these options are shown in Fig. 4-41. A procedure for implementing text-path orientation adjusts the position of the individual characters in the frame buffer according to the option selected.

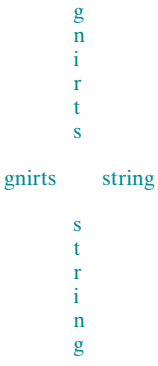


FIGURE 4–41 A text string displayed with the four text-path options: left, right, up, and down.

Character strings could also be oriented using a combination of up-vector and text-path specifications to produce slanted text. Fig. 4-42 shows the directions

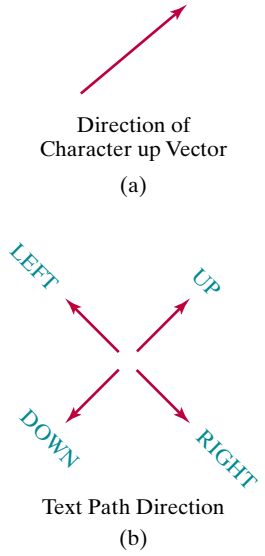


FIGURE 4–42 An up-vector specification (a) and associated directions for the text path (b).

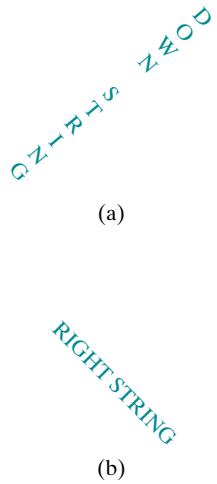


FIGURE 4–43 The 45° up vector in Fig. 4-42 produces the display (a) for a *down* path and the display (b) for a *right* path.

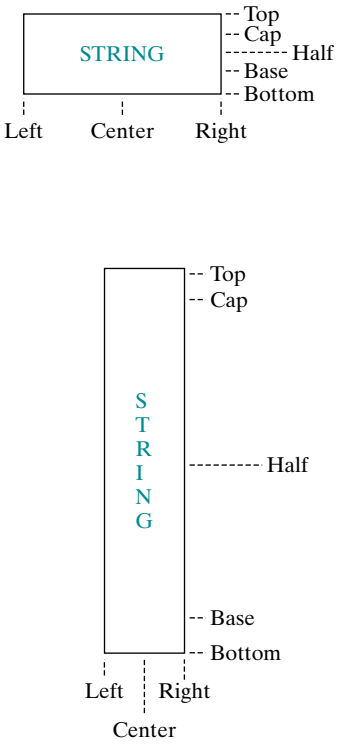


FIGURE 4–44 Character alignments for horizontal and vertical strings.

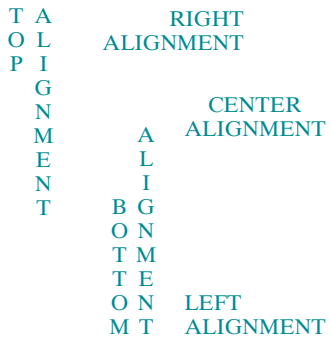


FIGURE 4–45 Character-string alignments.

of character strings generated by various text path settings for a 45° up vector. Examples of character strings generated for text-path values *down* and *right* with this up vector are illustrated in Fig. 4-43.

Another possible attribute for character strings is alignment. This attribute specifies how text is to be displayed with respect to a reference position. For example, individual characters could be aligned according to the base lines or the character centers. Figure 4-44 illustrates typical character positions for horizontal and vertical alignments. String alignments are also possible, and Fig. 4-45 shows common alignment positions for horizontal and vertical text labels.

In some graphics packages, a text-precision attribute is also available. This parameter specifies the amount of detail and the particular processing options that are to be used with a text string. For a low-precision text string, many attribute selections, such as text path, are ignored, and faster procedures are used for processing the characters through the viewing pipeline.

Finally, a library of text-processing routines often supplies a set of special characters, such as a small circle or cross, which are useful in various applications. Most often these characters are used as marker symbols in network layouts or in graphing data sets. The attributes for these marker symbols are typically *color* and *size*.

4-16 OpenGL CHARACTER-ATTRIBUTE FUNCTIONS

We have two methods for displaying characters with the OpenGL package. Either we can design a font set using the bitmap functions in the core library, or we can invoke the GLUT character-generation routines. The GLUT library contains functions for displaying predefined bitmap and stroke character sets. Therefore, the character attributes we can set are those that apply to either bitmaps or line segments.

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

For either bitmap or outline fonts, the display color is determined by the current color state. In general, the spacing and size of characters is determined by the font designation, such as `GLUT_BITMAP_9_BY_15` and `GLUT_STROKE_MONO_ROMAN`. But we can also set the line width and line type for the outline fonts. We specify the width for a line with the `glLineWidth` function, and we select a line type with the `glLineStipple` function. The GLUT stroke fonts will then be displayed using the current values we specified for the OpenGL line-width and line-type attributes.

We can accomplish some other text-display characteristics using the transformation functions described in Chapter 5. The transformation routines allow us to scale, position, and rotate the GLUT stroke characters in either two-dimensional space or three-dimensional space. In addition, the three-dimensional viewing transformations (Chapter 7) can be used to generate other display effects.

4-17 ANTIALIASING

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

An example of the effects of undersampling is shown in Fig. 4-46. To avoid losing information from such periodic objects, we need to set the sampling frequency to at least twice that of the highest frequency occurring in the object, referred to as the **Nyquist sampling frequency** (or Nyquist sampling rate) f_s:

$$f_s = 2 f_{\max} \tag{4-12}$$

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

$$\Delta x_s = \frac{\Delta x_{\text{cycle}}}{2} \tag{4-13}$$

where $\Delta x_{\text{cycle}} = 1/f_{\max}$. In Fig. 4-46, our sampling interval is one and one-half

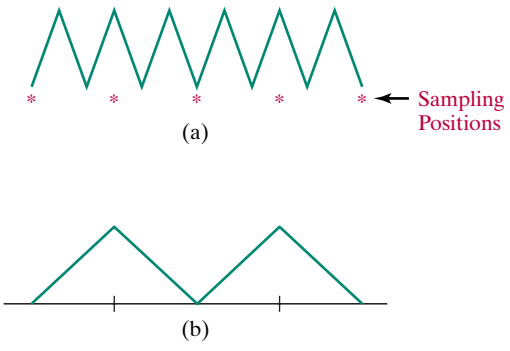


FIGURE 4-46 Sampling the periodic shape in (a) at the indicated positions produces the aliased lower-frequency representation in (b).

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

times the cycle interval, so the sampling interval is at least three times too large. If we want to recover all the object information for this example, we need to cut the sampling interval down to one-third the size shown in the figure.

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

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

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

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

Raster objects can also be antialiased by shifting the display location of pixel areas. This technique, called **pixel phasing,** is applied by "micropositioning" the electron beam in relation to object geometry. For example, pixel positions along a straight-line segment can be moved closer to the defined line path to smooth out the raster stair-step effect.

Supersampling Straight-Line Segments

We can perform supersampling in several ways. For a straight-line segment, we can divide each pixel into a number of subpixels and count the number of subpixels that overlap the line path. The intensity level for each pixel is then set to a value that is proportional to this subpixel count. An example of this method is given in Fig. 4-47. Each square pixel area is divided into nine equal-sized square subpixels, and the shaded regions show the subpixels that would be selected by Bresenham's algorithm. This scheme provides for three intensity settings above zero, since the maximum number of subpixels that can be selected within any pixel is three. For this example, the pixel at position (10, 20) is set to the

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

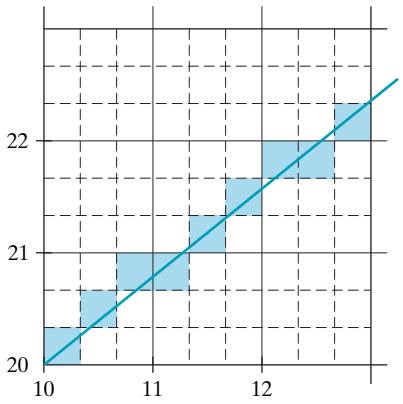


FIGURE 4–47 Supersampling subpixel positions along a straight-line segment whose left endpoint is at screen coordinates (10, 20).

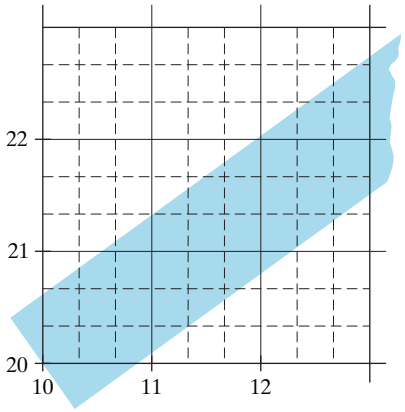


FIGURE 4–48 Supersampling subpixel positions in relation to the interior of a line of finite width.

maximum intensity (level 3); pixels at (11, 21) and (12, 21) are each set to the next highest intensity (level 2); and pixels at (11, 20) and (12, 22) are each set to the lowest intensity above zero (level 1). Thus the line intensity is spread out over a greater number of pixels to smooth the original jagged effect. This procedure displays a somewhat blurred line in the vicinity of the stair steps (between horizontal runs). If we want to use more intensity levels to antialiase the line with this method, we increase the number of sampling positions across each pixel. Sixteen subpixels gives us four intensity levels above zero; twenty-five subpixels gives us five levels; and so on.

In the supersampling example of Fig. 4-47, we considered pixel areas of finite size, but we treated the line as a mathematical entity with zero width. Actually, displayed lines have a width approximately equal to that of a pixel. If we take the finite width of the line into account, we can perform supersampling by setting pixel intensity proportional to the number of subpixels inside the polygon representing the line area. A subpixel can be considered to be inside the line if its lower left corner is inside the polygon boundaries. An advantage of this supersampling procedure is that the number of possible intensity levels for each pixel is equal to the total number of subpixels within the pixel area. For the example in Fig. 4-47, we can represent this line with finite width by positioning the polygon boundaries parallel to the line path as in Fig. 4-48. And each pixel can now be set to one of nine possible brightness levels above zero.

Another advantage of supersampling with a finite-width line is that the total line intensity is distributed over more pixels. In Fig. 4-48, we now have the pixel at grid position (10, 21) turned on (at intensity level 2), and we also pick up contributions from pixels immediately below and immediately to the left of position (10, 21). Also, if we have a color display, we can extend the method to take background colors into account. A particular line might cross several different color areas, and we can average subpixel intensities to obtain pixel color settings. For instance, if five subpixels within a particular pixel area are determined to be inside the boundaries for a red line and the remaining four subpixels fall within a blue background area, we can calculate the color for this pixel as

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

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

The trade-off for these gains from supersampling a finite-width line is that identifying interior subpixels requires more calculations than simply determining which subpixels are along the line path. Also, we need to take into account the positioning of the line boundaries in relation to the line path. This positioning depends on the slope of the line. For a 45° line, the line path is centered on the polygon area; but for either a horizontal or a vertical line, we want the line path to be one of the polygon boundaries. As an example, a horizontal line passing through grid coordinates (10, 20) could be represented as the polygon bounded by horizontal grid lines $y = 20$ and $y = 21$. Similarly, the polygon representing a vertical line through (10, 20) can have vertical boundaries along grid lines $x = 10$ and $x = 11$. For lines with slope $|m| < 1$, the mathematical line path is positioned proportionately closer to the lower polygon boundary; and for lines with slope $|m| > 1$, the line path is placed closer to the upper polygon boundary.

Subpixel Weighting Masks

Supersampling algorithms are often implemented by giving more weight to subpixels near the center of a pixel area, since we would expect these subpixels to be more important in determining the overall intensity of a pixel. For the 3 by 3 pixel subdivisions we have considered so far, a weighting scheme as in Fig. 4-49 could be used. The center subpixel here is weighted four times that of the corner subpixels and twice that of the remaining subpixels. Intensities calculated for each of the nine subpixels would then be averaged so that the center subpixel is weighted by a factor of $\frac{1}{4}$; the top, bottom, and side subpixels are each weighted by a factor of $\frac{1}{8}$; and the corner subpixels are each weighted by a factor of $\frac{1}{16}$. An array of values specifying the relative importance of subpixels is usually referred to as a *weighting mask*. Similar masks can be set up for larger subpixel grids. Also, these masks are often extended to include contributions from subpixels belonging to neighboring pixels, so that intensities can be averaged with adjacent pixels to provide a smoother intensity variation between pixels.

1	2	1
2	4	2
1	2	1

FIGURE 4–49 Relative weights for a grid of 3 by 3 subpixels.

Area Sampling Straight-Line Segments

We perform area sampling for a straight line by setting pixel intensity proportional to the area of overlap of the pixel with the finite-width line. The line can be treated as a rectangle, and the section of the line area between two adjacent vertical (or two adjacent horizontal) screen grid lines is then a trapezoid. Overlap areas for pixels are calculated by determining how much of the trapezoid overlaps each pixel in that column (or row). In Fig. 4-48, the pixel with screen grid coordinates (10, 20) is about 90 percent covered by the line area, so its intensity would be set to 90 percent of the maximum intensity. Similarly, the pixel at (10, 21) would be set to an intensity of about 15 percent of maximum. A method for estimating pixel overlap areas is illustrated by the supersampling example in Fig. 4-48. The total number of subpixels within the line boundaries is approximately equal to the overlap area, and this estimation is improved by using finer subpixel grids.

Filtering Techniques

A more accurate method for antialiasing lines is to use **filtering** techniques. The method is similar to applying a weighted pixel mask, but now we imagine a continuous *weighting surface* (or *filter function*) covering the pixel. Figure 4-50 shows

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

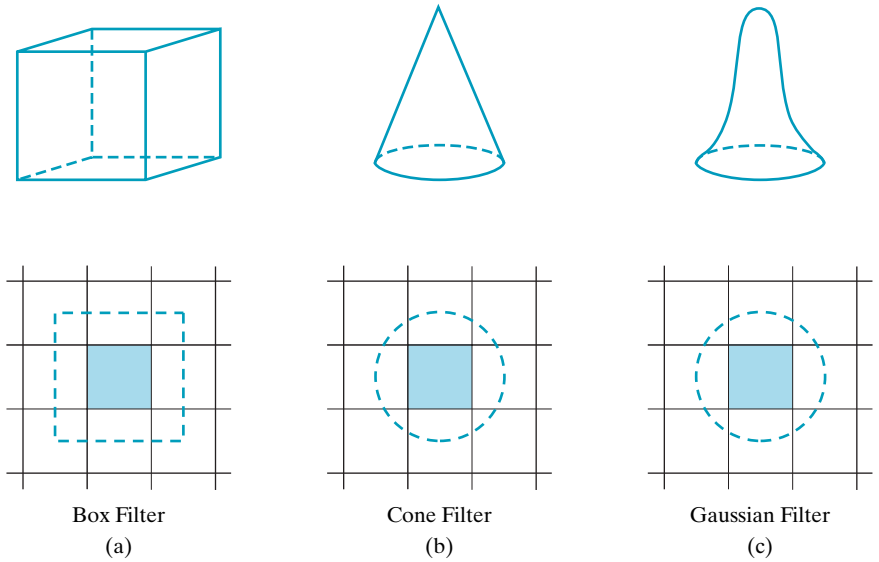


FIGURE 4–50 Common filter functions used to antialias line paths. The volume of each filter is normalized to 1.0, and the height gives the relative weight at any subpixel position.

examples of rectangular, conical, and Gaussian filter functions. Methods for applying the filter function are similar to those for applying a weighting mask, but now we integrate over the pixel surface to obtain the weighted average intensity. To reduce computation, table lookups are commonly used to evaluate the integrals.

Pixel Phasing

On raster systems that can address subpixel positions within the screen grid, pixel phasing can be used to antialias objects. A line display is smoothed with this technique by moving (micropositioning) pixel positions closer to the line path. Systems incorporating *pixel phasing* are designed so that the electron beam can be shifted by a fraction of a pixel diameter. The electron beam is typically shifted by $\frac{1}{4}$, $\frac{1}{2}$, or $\frac{3}{4}$ of a pixel diameter to plot points closer to the true path of a line or object edge. Some systems also allow the size of individual pixels to be adjusted as an additional means for distributing intensities. Figure 4-51 illustrates the antialiasing effects of pixel phasing on a variety of line paths.

Compensating for Line-Intensity Differences

Antialiasing a line to soften the stair-step effect also compensates for another raster effect, illustrated in Fig. 4-52. Both lines are plotted with the same number of pixels, yet the diagonal line is longer than the horizontal line by a factor of $\sqrt{2}$. For example, if the horizontal line had a length of 10 centimeters, the diagonal line would have a length of more than 14 centimeters. The visual effect of this is that the diagonal line appears less bright than the horizontal line, since the diagonal line is displayed with a lower intensity per unit length. A line-drawing algorithm could be adapted to compensate for this effect by adjusting the intensity of each line according to its slope. Horizontal and vertical lines would be displayed with the lowest intensity, while 45° lines would be given the highest intensity. But if antialiasing techniques are applied to a display, intensities are automatically compensated. When the finite width of a line is taken into account, pixel intensities are adjusted so that the line displays a total intensity proportional to its length.

hearn-50265; ISBN: 0-13-015390-7 book July 30, 2003 14:45

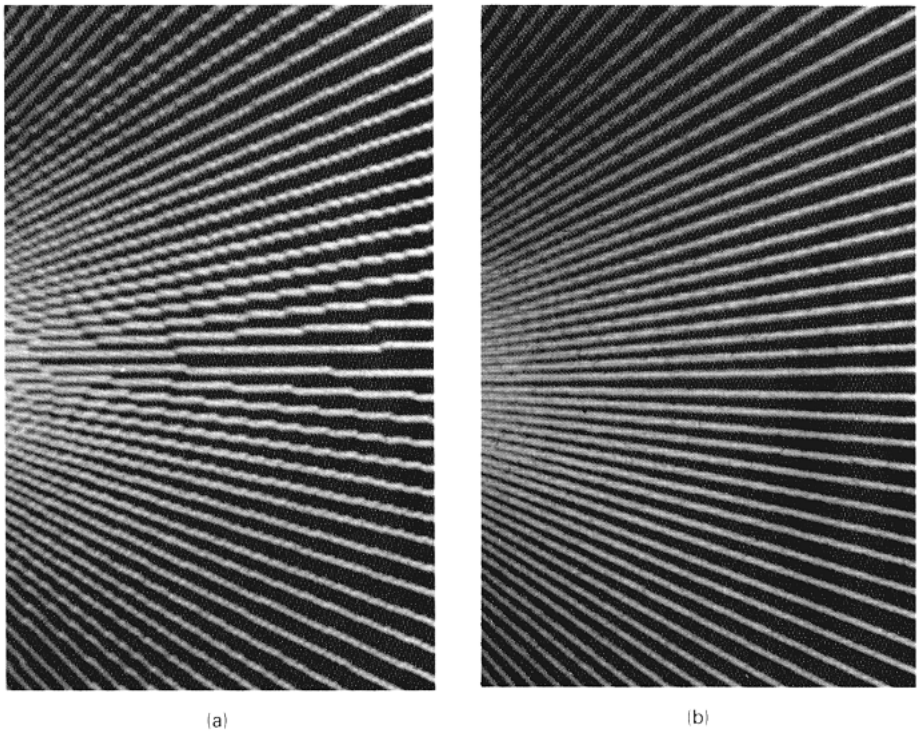


FIGURE 4–51 Jagged lines (a), plotted on the Merlin 9200 system, are smoothed (b) with an antialiasing technique called pixel phasing. This technique increases the number of addressable points on the system from 768 by 576 to 3072 by 2304. (*Courtesy of Peritek Corp.*)

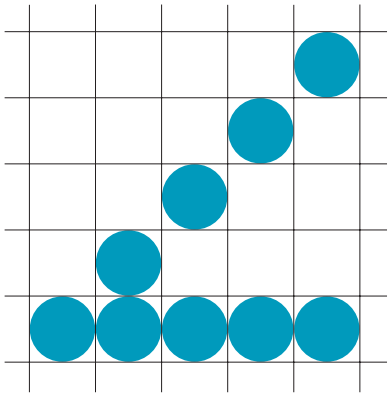


FIGURE 4–52 Unequal length lines displayed with the same number of pixels in each line.

Antialiasing Area Boundaries

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

If system capabilities permit the repositioning of pixels, we could smooth area boundaries by shifting pixel positions closer to the boundary. Other methods adjust pixel intensity at a boundary position according to the percent of the pixel area that is interior to the object. In Fig. 4-53, the pixel at position (x, y) has about half its area inside the polygon boundary. Therefore, the intensity at that position would be adjusted to one-half its assigned value. At the next position $(x+1, y+1)$

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

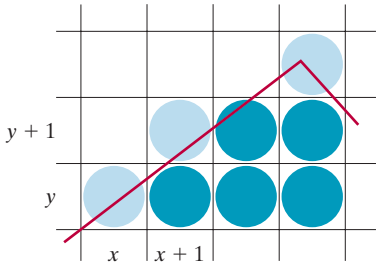


FIGURE 4–53 Adjusting pixel intensities along an area boundary.

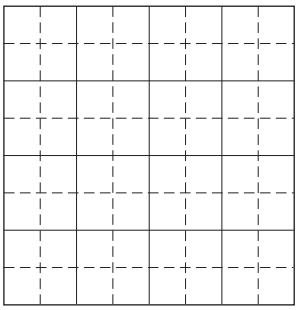


FIGURE 4–54 A 4 by 4 pixel section of a raster display subdivided into an 8 by 8 grid.

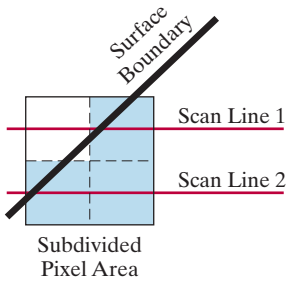


FIGURE 4–55 A subdivided pixel area with three subdivisions inside an object boundary line.

along the boundary, the intensity is adjusted to about one-third the assigned value for that point. Similar adjustments, based on the percent of pixel area coverage, are applied to the other intensity values around the boundary.

Supersampling methods can be applied by determining the number of subpixels that are in the interior of an object. A partitioning scheme with four subareas per pixel is shown in Fig. 4-54. The original 4 by 4 grid of pixels is turned into an 8 by 8 grid, and we now process eight scan lines across this grid instead of four. Figure 4-55 shows one of the pixel areas in this grid that overlaps an object boundary. Along the two scan lines, we determine that three of the subpixel areas are inside the boundary. So we set the pixel intensity at 75 percent of its maximum value.

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

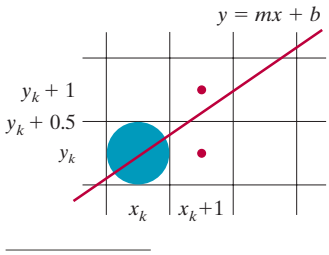


FIGURE 4–56 Boundary edge of a fill area passing through a pixel grid section.

We first consider the method for a line with slope m in the range from 0 to 1. In Fig. 4-56, a straight-line path is shown on a pixel grid. Assuming that the pixel at position (x_k, y_k) has been plotted, the next pixel nearest the line at $x = x_k + 1$ is either the pixel at y_k or the one at $y_k + 1$. We can determine which pixel is nearer with the calculation

$$y - y_{\text{mid}} = [m(x_k + 1) + b] - (y_k + 0.5) \tag{4-14}$$

This gives the vertical distance from the actual y coordinate on the line to the halfway point between pixels at position y_k and $y_k + 1$. If this difference calculation is negative, the pixel at y_k is closer to the line. If the difference is positive, the pixel at $y_k + 1$ is closer. We can adjust this calculation so that it produces a positive number in the range from 0 to 1 by adding the quantity $1 - m$:

$$p = [m(x_k + 1) + b] - (y_k + 0.5) + (1 - m) \tag{4-15}$$

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

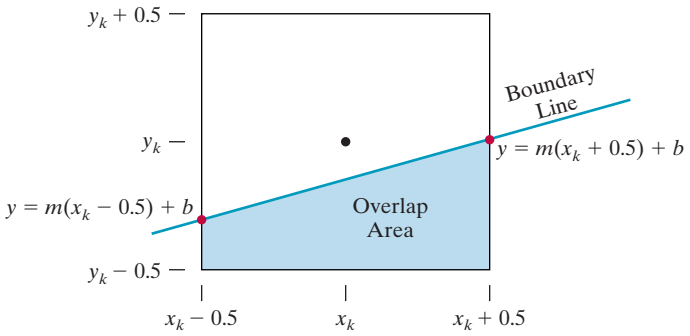


FIGURE 4–57 Overlap area of a pixel rectangle, centered at position (x_k, y_k), with the interior of a polygon fill area.

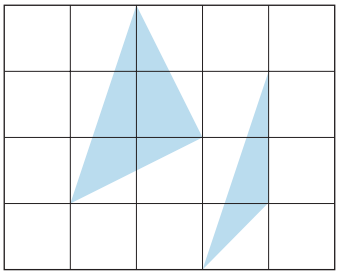


FIGURE 4–58 Polygons with more than one boundary line passing through individual pixel regions.

Parameter p also measures the amount of the current pixel that is overlapped by the area. For the pixel at (x_k, y_k) in Fig. 4-57, the interior part of the pixel has an area that can be calculated as

$$\text{area} = m \cdot x_k + b - y_k + 0.5 \qquad (4\text{-}16)$$

This expression for the overlap area of the pixel at (x_k, y_k) is the same as that for parameter p in Eq. 4-15. Therefore, by evaluating p to determine the next pixel position along the polygon boundary, we also determine the percentage of area coverage for the current pixel.

We can generalize this algorithm to accommodate lines with negative slopes and lines with slopes greater than 1. This calculation for parameter p could then be incorporated into a midpoint line algorithm to locate pixel positions along a polygon edge and, concurrently, adjust pixel intensities along the boundary lines. Also, we can adjust the calculations to reference pixel coordinates at their lower-left coordinates and maintain area proportions, as discussed in Section 3-13.

At polygon vertices and for very skinny polygons, as shown in Fig. 4-58, we have more than one boundary edge passing through a pixel area. For these cases, we need to modify the Pitteway-Watkinson algorithm by processing all edges passing through a pixel and determining the correct interior area.

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

4-18 OpenGL ANTIALIASING FUNCTIONS

We activate the antialiasing routines in OpenGL with the function

```
glEnable (primitiveType);
```

where parameter `primitiveType` is assigned one of the symbolic constant values `GL_POINT_SMOOTH`, `GL_LINE_SMOOTH`, or `GL_POLYGON_SMOOTH`.

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

Assuming we are specifying color values using the RGBA mode, we also need to activate the OpenGL color-blending operations.

```
glEnable (GL_BLEND);
```

Next, we apply the color-blending method described in Section 4-3 using the function

```
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

The smoothing operations are more effective if we use large alpha values in the color specifications for the objects.

Antialiasing can also be applied when we use color tables. However, in this color mode, we must create a `color ramp`, which is a table of color graduations from the background color to the object color. This color ramp is then used to antialias object boundaries.

4-19 OpenGL QUERY FUNCTIONS

We can retrieve current values for any of the state parameters, including attribute settings, using OpenGL **query functions.** These functions copy specified state values into an array, which we can save for later reuse or to check the current state of the system if an error occurs.

For current attribute values we use an appropriate "`glGet`" function, such as

```
glGetBooleanv ( )          glGetFloatv ( )
glGetIntegerv ( )          glGetDoublev ( )
```

In each of the preceding functions, we specify two arguments. The first argument is an OpenGL symbolic constant that identifies an attribute or other state parameter. The second argument is a pointer to an array of the data type indicated by the function name. For instance, we can retrieve the current RGBA floating-point color settings with

```
glGetFloatv (GL_CURRENT_COLOR, colorValues);
```

The current color components are then passed to the array `colorValues`. To obtain the integer values for the current color components, we invoke the `glGetIntegerv` function. In some cases, a type conversion may be necessary to return the specified data type.

Other OpenGL constants, such as `GL_POINT_SIZE`, `GL_LINE_WIDTH`, and `GL_CURRENT_RASTER_POSITION`, can be used in these functions to return current state values. And we could check the range of point sizes or line widths that are supported using the constants `GL_POINT_SIZE_RANGE` and `GL_LINE_WIDTH_RANGE`.

Although we can retrieve and reuse settings for a single attribute with the `glGet` functions, OpenGL provides other functions for saving groups of attributes and reusing their values. We consider the use of these functions for saving current attribute settings in the next section.

There are many other state and system parameters that are often useful to query. For instance, to determine how many bits per pixel are provided in the frame buffer on a particular system, we can ask the system how many bits are

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

available for each individual color component, such as

```
glGetIntegerv (GL_RED_BITS, redBitSize);
```

Here, array `redBitSize` is assigned the number of red bits available in each of the buffers (frame buffer, depth buffer, accumulation buffer, and stencil buffer). Similarly, we can make an inquiry for the other color bits using `GL_GREEN_BITS`, `GL_BLUE_BITS`, `GL_ALPHA_BITS`, or `GL_INDEX_BITS`.

We can also find out whether edge flags have been set, whether a polygon face was tagged as a front face or a back face, and whether the system supports double buffering. And we can inquire whether certain routines, such as color blending, line stippling or antialiasing, have been enabled or disabled.

4-20 OpenGL ATTRIBUTE GROUPS

Attributes and other OpenGL state parameters are arranged in **attribute groups.** Each group contains a set of related state parameters. For instance, the **point-attribute group** contains the size and point-smooth (antialiasing) parameters, and the **line-attribute group** contains the width, stipple status, stipple pattern, stipple repeat counter, and line-smooth status. Similarly, the **polygon-attribute group** contains eleven polygon parameters, such as fill pattern, front-face flag, and polygon-smooth status. Since color is an attribute for all primitives, it has its own attribute group. And some parameters are included in more than one group.

About twenty different attribute groups are available in OpenGL, and all parameters in one or more groups can be saved or reset with a single function. We save all parameters within a specified group using the following command.

```
glPushAttrib (attrGroup);
```

Parameter `attrGroup` is assigned an OpenGL symbolic constant that identifies an attribute group, such as `GL_POINT_BIT`, `GL_LINE_BIT`, or `GL_POLYGON_BIT`. To save color parameters, we use the symbolic constant `GL_CURRENT_BIT`. And we can save all state parameters in all attribute groups with the constant `GL_ALL_ATTRIB_BITS`. The `glPushAttrib` function places all parameters within the specified group onto an **attribute stack.**

We can also save parameters within two or more groups by combining their symbolic constants with a logical `OR` operation. The following statement places all parameters for points, lines, and polygons on the attribute stack.

```
glPushAttrib (GL_POINT_BIT | GL_LINE_BIT | GL_POLYGON_BIT);
```

Once we have saved a group of state parameters, we can reinstate all values on the attribute stack with this function:

```
glPopAttrib ( );
```

No arguments are used in the `glPopAttrib` function since it resets the current state of OpenGL using all values on the stack.

These commands for saving and resetting state parameters use a *server attribute stack*. There is also a *client attribute stack* available in OpenGL for saving and resetting client state parameters. The functions for accessing this stack are

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

`glPushClientAttrib` and `glPopClientAttrib`. Only two client attribute groups are available: one for pixel-storage modes and the other for vertex arrays. Pixel-storage parameters include information such as byte alignment and the type of arrays used to store subimages of a display. Vertex-array parameters give information about the current vertex-array state, such as the enable/disable state of various arrays.

4-21 SUMMARY

Attributes control the display characteristics of graphics primitives. In many graphics systems, attribute values are stored as state variables and primitives are generated using the current attribute values. When we change the value of a state variable, it affects only those primitives defined after the change.

A common attribute for all primitives is color, which is most often specified in terms of RGB (or RGBA) components. The red, green, and blue color values are stored in the frame buffer, and they are used to control the intensity of the three electron guns in an RGB monitor. Color selections can also be made using color-lookup tables. In this case, a color in the frame buffer is indicated as a table index, and the table location at that index stores a particular set of RGB color values. Color tables are useful in data-visualization and image-processing applications, and they can also be used to provide a wide range of colors without requiring a large frame buffer. Often, computer-graphics packages provide options for using either color tables or storing color values directly in the frame buffer.

The basic point attributes are color and size. On raster systems, various point sizes are displayed as square pixel arrays. Line attributes are color, width, and style. Specifications for line width are given in terms of multiples of a standard, one-pixel-wide line. The line-style attributes include solid, dashed, and dotted lines, as well as various brush or pen styles. These attributes can be applied to both straight lines and curves.

Fill-area attributes include a solid-color fill, a fill pattern, or a hollow display that shows only the area boundaries. Various pattern fills can be specified in color arrays, which are then mapped to the interior of the region. Scan-line methods are commonly used to fill polygons, circles, and ellipses. Across each scan line, the interior fill is applied to pixel positions between each pair of boundary intersections, left to right. For polygons, scan-line intersections with vertices can result in an odd number of intersections. This can be resolved by shortening some polygon edges. Scan-line fill algorithms can be simplified if fill areas are restricted to convex polygons. A further simplification is achieved if all fill areas in a scene are triangles. The interior pixels along each scan line are assigned appropriate color values, depending on the fill-attribute specifications. Painting programs generally display fill regions using a boundary-fill method or a flood-fill method. Each of these two fill methods requires an initial interior point. The interior is then painted pixel by pixel from the initial point out to the region boundaries.

Areas can also be filled using color blending. This type of fill has applications in antialiasing and in painting packages. Soft-fill procedures provide a new fill color for a region that has the same variations as the previous fill color. One example of this approach is the linear soft-fill algorithm that assumes that the previous fill was a linear combination of foreground and background colors. This same linear relationship is then determined from the frame buffer settings and used to repaint the area in a new color.

TABLE 4-2

SUMMARY OF OpenGL ATTRIBUTE FUNCTIONS

Function	*Description*
`glutInitDisplayMode`	Select the color mode, which can be either `GLUT_RGB` or `GLUT_INDEX`.
`glColor*`	Specify an RGB or RGBA color.
`glIndex*`	Specify a color using a color-table index.
`glutSetColor (index, r, g, b);`	Load a color into a color-table position.
`glEnable (GL_BLEND);`	Activate color blending.
`glBlendFunc (sFact, dFact);`	Specify factors for color blending.
`glEnableClientState (GL_COLOR_ARRAY);`	Activate color-array features of OpenGL.
`glColorPointer (size, type, stride, array);`	Specify an RGB color array.
`glIndexPointer (type, stride, array);`	Specify a color array using color-index mode.
`glPointSize (size)`	Specify a point size.
`glLineWidth (width);`	Specify a line width.
`glEnable (GL_LINE_STIPPLE);`	Activate line style.
`glEnable (GL_POLYGON_STIPPLE);`	Activate fill style.
`glLineStipple (repeat, pattern);`	Specify a line-style pattern.
`glPolygonStipple (pattern);`	Specify a fill-style pattern.
`glPolygonMode`	Display front or back face as either a set of edges or a set of vertices.
`glEdgeFlag`	Set fill-polygon edge flag to `GL_TRUE` or `GL_FALSE` to determine display status for an edge.
`glFrontFace`	Specify front-face vertex order as either `GL_CCW` or `GL_CW`.
`glEnable`	Activate antialiasing with `GL_POINT_SMOOTH`, `GL_LINE_SMOOTH`, or `GL_POLYGON_SMOOTH`. (Also need to activate color blending.)
`glGet**`	Various query functions, requiring specification of data type, symbolic name of a state parameter, and an array pointer.
`glPushAttrib`	Save all state parameters within a specified attribute group.
`glPopAttrib ( );`	Reinstate all state parameter values that were last saved.

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

Characters can be displayed in different styles (fonts), colors, sizes, spacing, and orientations. To set the orientation of a character string, we can specify a direction for the character up vector and a direction for the text path. In addition, we can set the alignment of a text string in relation to the start coordinate position. Individual characters, called marker symbols, can be used for applications such as plotting data graphs. Marker symbols can be displayed in various sizes and colors using standard characters or special symbols.

Because scan conversion is a digitizing process on raster systems, displayed primitives have a jagged appearance. This is due to the undersampling of information, which rounds coordinate values to pixel positions. We can improve the appearance of raster primitives by applying antialiasing procedures that adjust pixel intensities. One method for doing this is to supersample. That is, we consider each pixel to be composed of subpixels and we calculate the intensity of the subpixels and average the values of all subpixels. We can also weight the subpixel contributions according to position, giving higher weights to the central subpixels. Alternatively, we can perform area sampling and determine the percentage of area coverage for a screen pixel, then set the pixel intensity proportional to this percentage. Another method for antialiasing is to build special hardware configurations that can shift pixel positions.

In OpenGL, attribute values for the primitives are maintained as state variables. An attribute setting remains in effect for all subsequently defined primitives until that attribute value is changed. Changing an attribute value does not affect previously displayed primitives. We can specify colors in OpenGL using either the RGB (RGBA) color mode or the color-index mode, which uses color-table indices to select colors. Also, we can blend color values using the alpha color component. And we can specify values in color arrays that are to be used in conjunction with vertex arrays. In addition to color, OpenGL provides functions for selecting point size, line width, line style, and convex-polygon fill style, as well as providing functions for the display of polygon fill areas as either a set of edges or a set of vertex points. We can also eliminate selected polygon edges from a display, and we can reverse the specification of front and back faces. We can generate text strings in OpenGL using bitmaps or routines that are available in GLUT. Attributes that can be set for the display of GLUT characters include color, font, size, spacing, line width, and line type. The OpenGL library also provides functions to antialias the display of output primitives. We can use query functions to obtain the current value for state variables, and we can also obtain all values within an OpenGL attribute group using a single function.

Table 4-2 summarizes the OpenGL attribute functions discussed in this chapter. Additionally, the table lists some attribute-related functions.

REFERENCES

Soft-fill techniques are given in Fishkin and Barsky (1984). Antialiasing techniques are discussed in Pitteway and Watinson (1980), Crow (1981), Turkowski (1982), Fujimoto and Iwata (1983), Korein and Badler (1983), Kirk and Arvo (1991), and Wu (1991). Gray-scale applications are explored in Crow (1978). Other discussions of attributes and state parameters are available in Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994), and Paeth (1995).

Programming examples using OpenGL attribute functions are given in Woo, Neider, Davis, and Shreiner (1999). A complete listing of OpenGL attribute functions is available in Shreiner (2000), and GLUT character attributes are discussed in Kilgard (1996).

EXERCISES

4-1 Use the `glutSetColor` function to set up a color table for an input set of color values.

4-2 Using vertex and color arrays, set up the description for a scene containing at least six two-dimensional objects.

4-3 Write a program to display the two-dimensional scene description in the previous exercise.

4-4 Using vertex and color arrays, set up the description for a scene containing at least four three-dimensional objects.

4-5 Write a program to display a two-dimensional, gray-scale "cloud" scene, where the cloud shapes are to be described as point patterns on a blue-sky background. The light and dark regions of the clouds are to be modeled using points of varying sizes and interpoint spacing. (For example, a very light region can be modeled with small, widely spaced, light-gray points. Similarly, a dark region can be modeled with larger, more closely spaced, dark-gray points.)

4-6 Modify the program in the previous exercise to display the clouds in red and yellow color patterns as they might be seen at sunrise or at sunset. To achieve a realistic effect, use different shades of red and yellow (and perhaps green) for the points.

4-7 Implement a general line-style function by modifying Bresenham's line-drawing algorithm to display solid, dashed, or dotted lines.

4-8 Implement a line-style function using a midpoint line algorithm to display solid, dashed, or dotted lines.

4-9 Devise a parallel method for implementing a line-style function.

4-10 Devise a parallel method for implementing a line-width function.

4-11 A line specified by two endpoints and a width can be converted to a rectangular polygon with four vertices and then displayed using a scan-line method. Develop an efficient algorithm for computing the four vertices needed to define such a rectangle, with the line endpoints and line width as input parameters.

4-12 Implement a line-width function in a line-drawing program so that any one of three line widths can be displayed.

4-13 Write a program to output a line graph of three data sets defined over the same x-coordinate range. Input to the program is to include the three sets of data values and the labels for the graph. The data sets are to be scaled to fit within a defined coordinate range for a display window. Each data set is to be plotted with a different line style.

4-14 Modify the program in the previous exercise to plot the three data sets in different colors, as well as different line styles.

4-15 Set up an algorithm for displaying thick lines with butt caps, round caps, or projecting square caps. These options can be provided in an option menu.

4-16 Devise an algorithm for displaying thick polylines with a miter join, a round join, or a bevel join. These options can be provided in an option menu.

4-17 Modify the code segments in Section 4-8 for displaying data line plots, so that the line-width parameter is passed to procedure `linePlot`.

4-18 Modify the code segments in Section 4-8 for displaying data line plots, so that the line-style parameter is passed to procedure `linePlot`.

4-19 Complete the program in Section 4-8 for displaying line plots using input values from a data file.

4-20 Complete the program in Section 4-8 for displaying line plots using input values from a data file. In addition, the program should provide labeling for the axes and the coordinates for the display area on the screen. The data sets are to be scaled to fit

hearn-50265; ISBN: 0-13-015390-7 book July 29, 2003 17:6

the coordinate range of the display window, and each plotted line is to be displayed in a different line style, width, and color.

4-21 Implement pen and brush menu options for a line-drawing procedure, including at least two options: round and square shapes.

4-22 Modify a line-drawing algorithm so that the intensity of the output line is set according to its slope. That is, by adjusting pixel intensities according to the value of the slope, all lines are displayed with the same intensity per unit length.

4-23 Define and implement a function for controlling the line style (solid, dashed, dotted) of displayed ellipses.

4-24 Define and implement a function for setting the width of displayed ellipses.

4-25 Write a routine to display a bar graph in any specified screen area. Input is to include the data set, labeling for the coordinate axes, and the coordinates for the screen area. The data set is to be scaled to fit the designated screen area, and the bars are to be displayed in designated colors or patterns.

4-26 Write a procedure to display two data sets defined over the same x-coordinate range, with the data values scaled to fit a specified region of the display screen. The bars for one of the data sets are to be displaced horizontally to produce an overlapping bar pattern for easy comparison of the two sets of data. Use a different color or a different fill pattern for the two sets of bars.

4-27 Devise an algorithm for implementing a color lookup table.

4-28 Suppose you have a system with an 8 inch by 10 inch video screen that can display 100 pixels per inch. If a color lookup table with 64 positions is used with this system, what is the smallest possible size (in bytes) for the frame buffer?

4-29 Consider an RGB raster system that has a 512-by-512 frame buffer with 20 bits per pixel and a color lookup table with 24 bits per pixel. (a) How many distinct gray levels can be displayed with this system? (b) How many distinct colors (including gray levels) can be displayed? (c) How many colors can be displayed at any one time? (d) What is the total memory size? (e) Explain two methods for reducing memory size while maintaining the same color capabilities.

4-30 Modify the scan-line algorithm to apply any specified rectangular fill pattern to a polygon interior, starting from a designated pattern position.

4-31 Write a program to scan convert the interior of a specified ellipse into a solid color.

4-32 Write a procedure to fill the interior of a given ellipse with a specified pattern.

4-33 Write a procedure for filling the interior of any specified set of fill-area vertices, including one with crossing edges, using the nonzero winding number rule to identify interior regions.

4-34 Modify the boundary-fill algorithm for a 4-connected region to avoid excessive stacking by incorporating scan-line methods.

4-35 Write a boundary-fill procedure to fill an 8-connected region.

4-36 Explain how an ellipse displayed with the midpoint method could be properly filled with a boundary-fill algorithm.

4-37 Develop and implement a flood-fill algorithm to fill the interior of any specified area.

4-38 Define and implement a procedure for changing the size of an existing rectangular fill pattern.

4-39 Write a procedure to implement a soft-fill algorithm. Carefully define what the soft-fill algorithm is to accomplish and how colors are to be combined.

4-40 Devise an algorithm for adjusting the height and width of characters defined as rectangular grid patterns.

4-41 Implement routines for setting the character up vector and the text path for controlling the display of character strings.

4-42 Write a program to align text as specified by input values for the alignment parameters.

4-43 Develop procedures for implementing marker attributes (size and color).

4-44 Implement an antialiasing procedure by extending Bresenham's line algorithm to adjust pixel intensities in the vicinity of a line path.

4-45 Implement an antialiasing procedure for the midpoint line algorithm.

4-46 Develop an algorithm for antialiasing elliptical boundaries.

4-47 Modify the scan-line algorithm for area fill to incorporate antialiasing. Use coherence techniques to reduce calculations on successive scan lines.

4-48 Write a program to implement the Pitteway-Watkinson antialiasing algorithm as a scan-line procedure to fill a polygon interior, using the OpenGL point-plotting function.