Chapter 2

Attributes of geometric primitives

2.1 Overview

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.

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 the line.

Another approach is to maintain a system list of current attribute values. Separate functions are then included in the graphics package for setting 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.

2.1 Overview

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

2.1 Overview

OpenGL state variables

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 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.

2.1 Overview

OpenGL state variables (continued)

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 the `glBegin/glEnd` pairs, along with the coordinate values, so that parameter settings can vary from one coordinate position to another.

2.2 Color

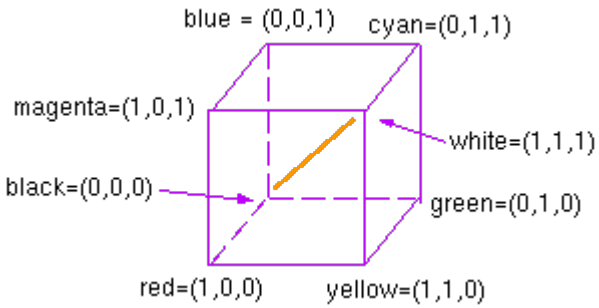
The way colors are specified, usually depends on the technology used. For example, for printers the CMY color model is suitable due to the subtractive combination of colors.

Raster displays are typically based on different colors being added together. For example, combining red, green, and blue results in the color white when added together. Hence, these types use an additive color model with fundamental colors being red, green, and blue. Thus, the RGB color model is very suitable for raster displays.

2.2 Color

The RGB color space

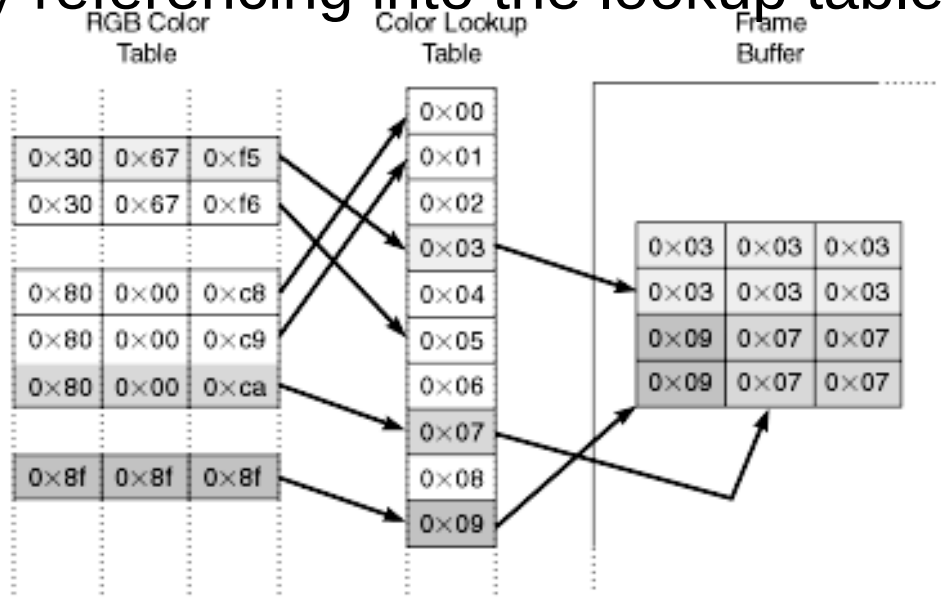
The RGB color space can be visualized as a cube with the three fundamental colors in three corners, maximizing the distances between one another. In the remaining corners, we find black, white, cyan, magenta, and yellow.



2.2 Color

Color lookup tables

We could also specify colors using a color lookup table. There, a table of colors is stored containing a subset of colors available in the RGB color model. Then, colors are specified by referencing into the lookup table.



2.2 Color

Color lookup tables

Usually, the reference to the color in the lookup table uses less memory compared to storing the real RGB value. Therefore, using a color lookup table can reduce the memory requirements for storing the frame buffer in the graphics memory. Since memory is quite affordable nowadays, specifying the RGB values directly is the more common approach now.

2.2 Color

OpenGL color functions

In our first example, a few OpenGL color functions were introduced. 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 segments. Also, we set the color display mode to RGB with the statement:

```
glutInitDisplayMode (GLUT_SINGLE,
                     GLUT_RGB);
```

Instead of `GLUT_RGB`, we could specify `GLUT_RGBA` to enable the alpha channel (transparency). This allows us to blend between the color of a primitive and a background color, for example.

2.2 Color

OpenGL color functions (continued)

Colors are usually specified using their red, green, and blue components of that color. A fourth component can be used to specify the alpha coefficient used for color blending. 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 where a value of 1.0 (or 255 when using integer) results in an opaque setting.

2.2 Color

OpenGL color functions (continued)

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. Typical suffix codes for the numerical data types are `b` (byte), `i` (integer), `s` (short), `f` (float), and `d` (double). Floating point values for the color components are in the range from 0.0 (zero intensity) to 1.0 (full intensity). When using integer numbers, the values range from 0 to 255.

2.2 Color

OpenGL color functions (continued)

The default color is (1.0, 1.0, 1.0, 1.0), i.e. white with full opacity.

Examples:

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

float colorArray[3] = { 0.0, 1.0, 1.0};
glColor3fv (colorArray);

glColor4i (0, 255, 255, 255);
```

2.2 Color

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, for example when using anti-aliasing or using transparency.

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 **destination color** and the color of the second object is the **source color**.

2.2 Color

OpenGL color blending (continued)

To apply color blending in an application, we first need to activate this OpenGL feature using the following function:

```
glEnable (GL_BLEND);
```

To turn off color blending, we can use:

```
glDisable (GL_BLEND);
```

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

2.2 Color

OpenGL color functions (continued)

We then need to specify how to combine the two colors using this function:

```
glBlendFunc (s, d);
```

The parameters `s` and `d` specify how the source and destination colors are to be combined using the formula:

$$s \cdot src\_color + d \cdot destination\_color$$

As factors, the following OpenGL constants can be used:

`GL_ZERO, GL_ONE, GL_DST_ALPHA, GL_SRC_ALPHA, GL_ONE_MINUS_DST_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_COLOR, GL_SRC_COLOR`

The default setting is `glBlendFunc (GL_ONE, GLZERO).`

2.2 Color

OpenGL color arrays

When using vertex arrays, colors can also be specified in an array just like the vertices. Similarly to the vertices, we need to activate the color-array features of OpenGL:

```
glEnableClientState (GL_COLOR_ARRAY);
```

Then, 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 using alpha values or not.

2.2 Color

OpenGL color arrays (continued)

The example in chapter 1 then changes to:

```
GLfloat vertices[] = { … };
GLfloat colors[] = { … };
glEnableClientState (GL_VERTEX_ARRAY);
glEnableClientState (GL_COLOR_ARRAY);
glVertexPointer (3, GL_FLOAT, 0, vertices);
glColorPointer (3, GL_FLOAT, 0, colors);
glDrawArrays (GL_TRIANGLE_STRIP, 0, 10);
```

2.2 Color

OpenGL color arrays (continued)

We could even store both colors and vertices in one interleaved array. Each of the pointers would then reference the single interleaved array, with an appropriate offset value and slightly different start pointer to the array:

```
GLfloat interleaved[] = { … };
glEnableClientState (GL_VERTEX_ARRAY);
glEnableClientState (GL_COLOR_ARRAY);
glVertexPointer (3, GL_FLOAT, 6 * sizeof
                 (GLfloat), &(interleaved[3]));
glColorPointer (3, GL_FLOAT, 6 * sizeof
                (GLfloat), interleaved);
glDrawArrays (GL_TRIANGLE_STRIP, 0, 10);
```

2.2 Color

OpenGL color arrays (continued)

To reduce the number of function calls even further, OpenGL provides a function in which we can specify all the vertex and color arrays at once. Therefore, we can replace the two calls referring to the arrays with this one:

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

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 interleaved are to be interleaved with the color for each vertex listed before the coordinates. This function also automatically enables both vertex and color arrays.

2.3 Attributes

OpenGL point attributes

Besides changing the color of a point, we can also specify the point size. We set the size for an OpenGL point with:

```
glPointSize (size);
```

Parameter `size` is assigned a positive floating point value, which is rounded to an integer (unless the point is to be anti-aliased). A point size of 1.0 (the default value) displays a single pixel, and a point size of 2.0 displays a 2 by 2 pixel array.

2.3 Attributes

OpenGL line attributes

Similar to the point size, the line width can be changed using the following OpenGL 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 the standard (default) width of 1.0. When using anti-aliasing, fractional widths are possible as well.

2.3 Attributes

OpenGL line attributes (continued)

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 dashed and dots. We set a current display style for lines with this 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 in the bit-pattern denotes an “on” 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 which produces a solid line.

2.3 Attributes

OpenGL line attributes (continued)

As an example of specifying a line style, the following function call results in dashed lines:

```
glLineStipple (1, 0x00FF);
```

The first half of this pattern (eight pixels) switches those pixels off, while the second half results in visible pixels. 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.

2.3 Attributes

OpenGL line attributes (continued)

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.

2.3 Attributes

OpenGL line attributes (continued)

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

```
glEnable (GL_LINE_STIPPLE);
```

Without enabling this feature, lines would still appear as solid lines, even though a pattern was provided.

To disable the use of a pattern we can issue the following function call:

```
glDisable (GL_LINE_STIPPLE);
```

2.4 Fill attributes

OpenGL fill attributes

By default, a polygon is displayed as a solid-color region, using the current color setting. To fill the polygon with a pattern in OpenGL, a 32-bit by 32-bit bit mask has to be specified similar to defining a line-style:

```
GLubyte pattern []
 = { 0xff, 0x00, 0xff, 0x00, …};
```

Once we have set the mask, we can establish it as the current fill pattern:

```
glPolygonStipple (pattern);
```

2.4 Fill attributes

OpenGL fill attributes (continued)

Since OpenGL is a state machine, 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 the pattern filling with

```
glDisable (GL_POLYGON_STIPPLE);
```

Then, we can specify the polygons by using one of OpenGL's drawing methods.