Chapter 7
Color Systems

After reading this chapter, you'll be able to do the following:

- Specifying RGBA or color-index mode for application.
- Specifying desired colors for drawing objects.
- Using smooth shading to draw a single polygon with more than one color.

7.1 Introduction

The goal of almost all OpenGL applications is to draw color pictures in a window on the screen. The window is a rectangular array of pixels, each of which contains and displays its own color. Thus, in a sense, the point of all the calculations performed by an OpenGL implementation - calculations that take into account OpenGL commands, state information, and values of parameters - is to determine the final color of every pixel that's to be drawn in the window. This chapter explains the commands for specifying colors and how OpenGL interprets them in the following major sections:

- Color perception discusses how the eye perceives color.
- Computer color describes the relationship between pixels on a computer monitor and their colors; it also defines the two display modes, RGBA and color index.
- RGBA versus color-index mode explains how the two display modes use graphics hardware and how to decide which mode to use.
- Specifying a color and a shading model describes the OpenGL commands you use to specify the desired color or shading model.

This chapter can be organized as follows: section 2 discusses color perception. In Section 3, we describe color display mode. Section 4 presents the choosing between RGBA and color-index mode. Specifying a color and a shading model is described in Section 5.

7.2 Color perception

Physically, light is composed of photons - tiny particles of light, each traveling along its own path, and each vibrating at its own frequency (or wavelength, or energy - any one of frequency, wavelength, or energy determines the others). A photon is completely characterized by its position, direction, and frequency/wavelength/energy. Photons with wavelengths ranging from about 390 nanometers (nm) (violet) and 720 nm (red) cover the colors of the visible spectrum, forming the colors of a rainbow (violet, indigo, blue, green, yellow, orange, red).

However, your eyes perceive lots of colors that aren't in the rainbow - white, black, brown, and pink, for example. How does this happen?

What your eye actually sees is a mixture of photons of different frequencies. Real light sources are characterized by the distribution of photon frequencies they emit. Ideal white light consists of an equal amount of light of all frequencies. Laser light is usually very pure, and all photons have almost identical frequencies (and direction and phase, as well). Light from a sodium-vapor lamp has more light in the yellow frequency. Light from most stars in space has a distribution that depends heavily on their temperatures (black-body radiation). The frequency distribution of light from most sources in your immediate environment is more complicated.

The human eye perceives color when certain cells in the retina (called cone cells, or just cones) become excited after being struck by photons. The three different kinds of cone cells respond best to three different wavelengths of light: one type of cone cell responds best to red light, one type to green, and the other to blue. (A person who is color-blind is usually missing one or more types of cone cells.) When a given mixture of photons enters the eye, the cone cells in the retina register different degrees of excitation depending on their types, and if a different mixture of photons comes in that happens to excite the three types of cone cells to the same degrees, its color is indistinguishable from that of the first mixture.

Since each color is recorded by the eye as the levels of excitation of the cone cells by the incoming photons, the eye can perceive colors that aren't in the spectrum produced by a prism or rainbow. For example, if you send a mixture of red and blue photons so that both the red and blue cones in the retina are excited, your eye sees it as magenta, which isn't in the spectrum. Other combinations give browns, turquoises, and mauves, none of which appear in the color spectrum.

A computer-graphics monitor emulates visible colors by lighting pixels with a combination of red, green, and blue light in proportions that excite the red-, green-, and blue-sensitive cones in the retina in such a way that it matches the excitation levels generated by the

photon mix it's trying to emulate. If humans had more types of cone cells, some that were yellow-sensitive for example, color monitors would probably have a yellow gun as well, and we'd use RGBY (red, green, blue, yellow) quadruples to specify colors. And if everyone were color-blind in the same way, this chapter would be simpler.
To display a particular color, the monitor sends the right amounts of red, green, and blue light to appropriately stimulate the different types of cone cells in your eye. A color monitor can send different proportions of red, green, and blue to each of the pixels, and the eye sees a million or so pinpoints of light, each with its own color.
This section considers only how the eye perceives combinations of photons that enter it. The situation for light bouncing off materials and entering the eye is even more complex - white light bouncing off a red ball will appear red, or yellow light shining through blue glass appears almost black, for example.

7.2.1. Computer color

On a color computer screen, the hardware causes each pixel on the screen to emit different amounts of red, green, and blue light. These are called the R, G, and B values. They're often packed together (sometimes with a fourth value, called alpha, or A), and the packed value is called the RGB (or RGBA) value. The color information at each pixel can be stored either in RGBA mode, in which the R, G, B, and possibly A values are kept for each pixel, or in color-index mode, in which a single number (called the color index) is stored for each pixel. Each color index defines a particular set of R, G, and B values.
In color-index mode, you might want to alter the values in the color map. Since color maps are controlled by the window system, there are no OpenGL commands to do this. All the examples in this book initialize the color-display mode at the time the window is opened by using routines from the GLUT library. There is a great deal of variation among the different graphics hardware platforms in both the size of the pixel array and the number of colors that can be displayed at each pixel. On any graphics system, each pixel has the same amount of memory for storing its color, and all the memory for all the pixels is

called the color buffer. The size of a buffer is usually measured in bits, so an 8-bit buffer could store 8 bits of data (256 possible different colors) for each pixel. The size of the possible buffers varies from machine to machine.

The R, G, and B values can range from 0.0 (none) to 1.0 (full intensity). For example, R = 0.0, G = 0.0, and B = 1.0 represents the brightest possible blue. If R, G, and B are all 0.0, the pixel is black; if all are 1.0, the pixel is drawn in the brightest white that can be displayed on the screen. Blending green and blue creates shades of cyan. Blue and red combine for magenta. Red and green create yellow. To help you create the colors you want from the R, G, and B components looks at the color cube shown in Plate 12. The axes of this cube represent intensities of red, blue, and green. A black-and-white version of the cube is shown in Fig.7-1.

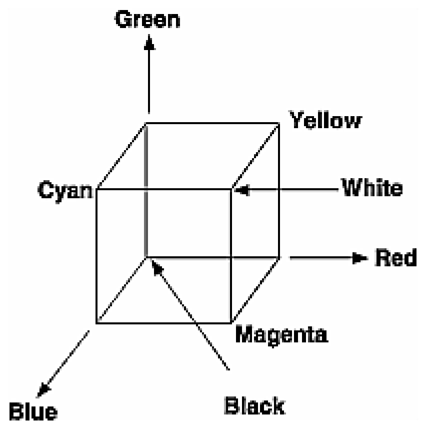


Fig.7-1: The color cube in black and white.

The commands to specify a color for an object (in this case, a point) can be as simple as this:

```
glColor3f (1.0, 0.0, 0.0);/* the current RGB color is red:
*/
                    /* full red, no green, no blue. */
glBegin (GL_POINTS);
    glVertex3fv (point_array);
```

```
glEnd ();
```

In certain modes (for example, if lighting or texturing calculations are performed), the assigned color might go through other operations before arriving in the framebuffer as a value representing a color for a pixel. In fact, the color of a pixel is determined by a lengthy sequence of operations.

Early in a program's execution, the color-display mode is set to either RGBA mode or color-index mode. Once the color-display mode is initialized, it can't be changed. As the program executes, a color (either a color index or an RGBA value) is determined on a per-vertex basis for each geometric primitive. This color is either a color you've explicitly specified for a vertex or, if lighting is enabled, is determined from the interaction of the transformation matrices with the surface normals and other material properties. In other words, a red ball with a blue light shining on it looks different from the same ball with no light on it. After the relevant lighting calculations are performed, the chosen shading model is applied. As explained in "Specifying a Color and a Shading Model," you can choose flat or smooth shading, each of which has different effects on the eventual color of a pixel.

7.2.2. RGBA versus color-index mode

In either color-index or RGBA mode, a certain amount of color data is stored at each pixel. This amount is determined by the number of bitplanes in the framebuffer. A bitplane contains 1 bit of data for each pixel. If there are 8color bitplanes, there are 8 color bits per pixel, and hence 28 = 256 different values or colors that can be stored at the pixel.

Bitplanes are often divided evenly into storage for R, G, and B components (that is, a 24-bitplane system devotes 8 bits each to red, green, and blue), but this isn't always true. To find out the number of bitplanes available on your system for red, green, blue, alpha, or color-index values, use glGetIntegerv() with GL_RED_BITS, GL_GREEN_BITS, GL_BLUE_BITS,
GL_ALPHA_BITS, and GL_INDEX_BITS.

Note: Color intensities on most computer screens aren't perceived as linear by the human eye. Consider colors consisting of just a red component, with green and blue set to zero. As the intensity varies from 0.0 (off) to 1.0 (full on), the number of electrons striking the pixels increases, but the question is, does 0.5 look like halfway between 0.0 and 1.0? To test this, write a program that draws alternate pixels in a checkerboard pattern to intensities 0.0 and 1.0, and compare it with a region drawn solidly in color 0.5. From a reasonable distance from the screen, the two regions should appear to have the same intensity. If they look noticeably different, you need to use whatever correction mechanism is provided on your particular system. For example, many systems have adjusted intensities so that 0.5 appears to be halfway between 0.0 and 1.0. The mapping generally used is an exponential one, with the exponent referred to as gamma (hence the term gamma correction). Using the same gamma for the red, green, and blue components gives pretty good results, but three different gamma values might give slightly better results.

7.2.3. RGBA display mode

In RGBA mode, the hardware sets aside a certain number of bitplanes for each of the R, G, B, and A components (not necessarily the same number for each component) as shown in Fig.7-2. The R, G, and B values are typically stored as integers rather than floating-point numbers, and they're scaled to the number of available bits for storage and retrieval. For example, if a system has 8 bits available for the R component, integers between 0 and 255 can be stored; thus, 0, 1, 2, ..., 255 in the bitplanes would correspond to R values of 0/255 = 0.0, 1/255, 2/255, ..., 255/255 = 1.0. Regardless of the number of bitplanes, 0.0 specifies the minimum intensity, and 1.0 specifies the maximum intensity.

Note: The alpha value (the A in RGBA) has no direct effect on the color displayed on the screen. It can be used for many things, including blending and transparency, and it can have an effect on the values of R, G, and B that are written.

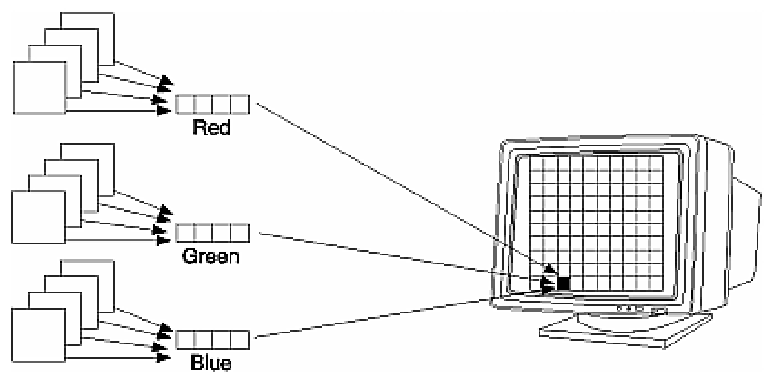


Fig.7-2:RGB Values from the Bitplanes.

The number of distinct colors that can be displayed at a single pixel depends on the number of bitplanes and the capacity of the hardware to interpret those bitplanes. The number of distinct colors can't exceed 2n, where n is the number of bitplanes. Thus, a machine with 24 bitplanes for RGB can display up to 16.77 million distinct colors.

7.2.4. Dithering

Some graphics hardware uses dithering to increase the number of apparent colors. Dithering is the technique of using combinations of some colors to create the effect of other colors. To illustrate how dithering works, suppose your system has only 1 bit each for R, G, and B and thus can display only eight colors: black, white, red, blue, green, yellow, cyan, and magenta. To display a pink region, the hardware can fill the region in a checkerboard manner, alternating red and white pixels. If your eye is far enough away from the screen that it can't distinguish individual pixels, the region appears pink - the average of red and white. Redder pinks can be achieved by filling a higher proportion of the pixels with red, whiter pinks would use more white pixels, and so on.

With this technique, there are no pink pixels. The only way to achieve the effect of "pinkness" is to cover a region consisting of multiple pixels - you can't dither a single pixel. If you specify an RGB value for an unavailable color and fill a polygon, the hardware fills the pixels in the interior of the polygon with a mixture of nearby colors whose average appears to your eye to be the color you want. (Remember, though, that if you're reading pixel information out of the frame buffer, you get the actual red and white pixel values, since there aren't any pink ones.

Fig.7-3 illustrates some simple dithering of black and white pixels to make shades of gray. From left to right, the 4 X 4 patterns at the top represent dithering patterns for 50 percent, 19 percent, and 69 percent gray. Under each pattern, you can see repeated reduced copies of each pattern, but these black and white squares are still bigger than most pixels. If you look at them from across the room, you can see that they blur together and appear as three levels of gray.

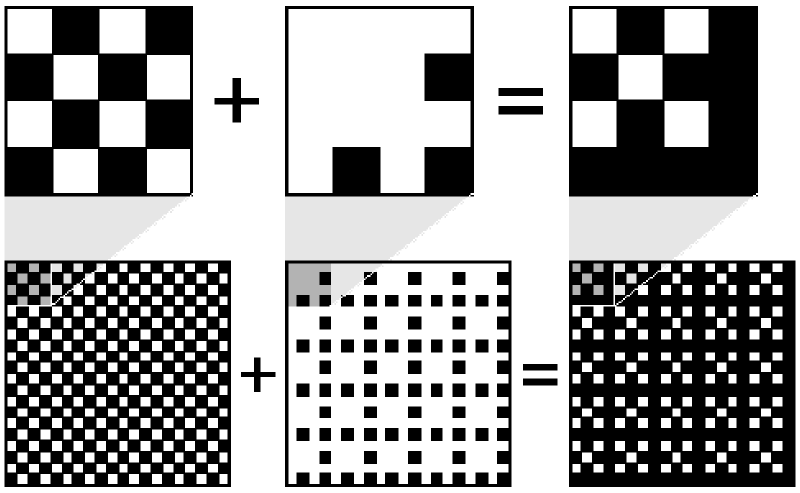


Fig.7-3: Dithering Black and White to Create Gray.

With about 8 bits each of R, G, and B, you can get a fairly high-quality image without dithering. Just because your machine has 24 color bitplanes, however, doesn't mean that dithering won't be desirable. For example, if you are running in double-buffer mode, the bitplanes might be divided into two sets of twelve, so there are really only 4 bits each per R, G, and B component. Without dithering, 4-bit-per-component color can give less than satisfactory results in many situations. You enable or disable dithering by passing GL_DITHER to glEnable() or glDisable(). Note that dithering, unlike many other features, is enabled by default.

7.3 Color-index display mode

With color-index mode, OpenGL uses a color map (or lookup table in Fig.7-4), which is similar to using a palette to mix paints to prepare for a paint-by-number scene. A painter's palette provides spaces to mix paints together; similarly, a computer's color map provides indices where the primary red, green, and blue values can be mixed, as shown in Fig.7-4.

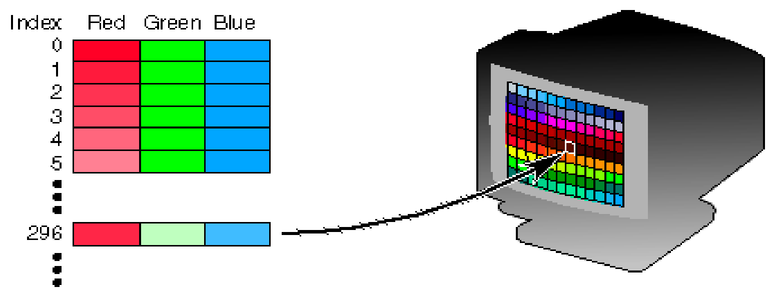


Fig.7-4: A Color Map.

A painter filling in a paint-by-number scene chooses a color from the color palette and fills the corresponding numbered regions with that color. A computer stores the color index in the bitplanes for each pixel. Then those bitplane values reference the color map, and the screen is painted with the corresponding red, green, and blue values from the color map, as shown in Fig. 7-5.

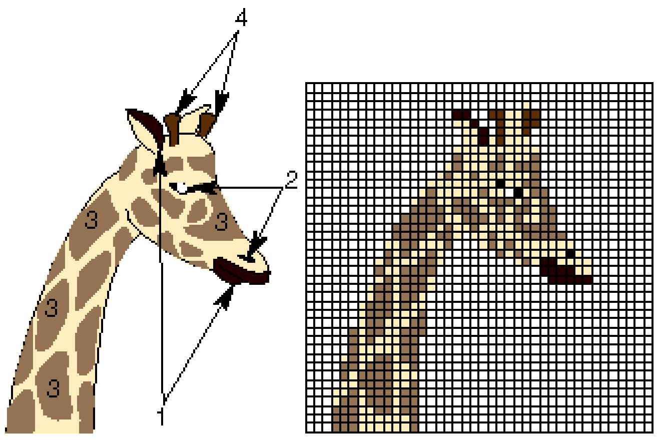


Fig.7-5: Using a Color Map to Paint a Picture.

In color-index mode, the number of simultaneously available colors is limited by the size of the color map and the number of bitplanes available. The size of the color map is determined by the amount of hardware dedicated to it. The size of the color map is always a power of 2, and typical sizes range from 256 (28) to 4096 (212), where the exponent is the number of bitplanes being used. If there are 2n indices in the color map and m available bitplanes, the number of usable entries is the smaller of 2n and 2m.
With RGBA mode, each pixel's color is independent of other pixels. However, in color-index mode, each pixel with the same index stored in its bitplanes shares the same color-map location. If the contents of an entry in the color map change, then all pixels of that color index change their color.

7.4 Choosing between RGBA and color-index mode

You should base your decision to use RGBA or color-index mode on what hardware is available and on what your application needs. For most systems, more colors can be simultaneously represented with RGBA mode than with color-index mode. Also, for several effects, such as shading, lighting, texture mapping, and fog, RGBA provides more flexibility than color-index mode.

You might prefer to use color-index mode in the following cases:
If you're porting an existing application that makes significant use of color-index mode, it might be easier to not change to RGBA mode.
If you have a small number of bitplanes available, RGBA mode may produce noticeably coarse shades of colors. For example, if you have only 8 bitplanes, in RGBA mode, you may have only 3 bits for red, 3 bits for green, and 2 bits for blue. You'd only have 8 (23) shades of red and green, and only 4 shades of blue. The gradients between color shades are likely to be very obvious.
In this situation, if you have limited shading requirements, you can use the color lookup table to load more shades of colors. For example, if you need only shades of blue, you can use color-index mode and store up to 256 (28) shades of blue in the color-lookup table, which is much better than the 4 shades you would have in RGBA mode. Of course, this example would use up your entire color-lookup table, so you would have no shades of red, green, or other combined colors.
Color-index mode can be useful for various tricks, such as color-map animation and drawing in layers. In general, use RGBA mode wherever possible. It works with texture mapping and works better with lighting, shading, fog, antialiasing, and blending.

Changing between display modes

In the best of all possible worlds, you might want to avoid making a choice between RGBA and color-index display mode. For example, you may want to use color-index mode for a color-map animation effect and then, when needed, immediately change the scene to RGBA mode for texture mapping.
Or similarly, you may desire to switch between single and double buffering. For example, you may have very few bitplanes; let's say 8 bitplanes. In single-buffer mode, you'll have 256 (28) colors, but if you are using double-buffer mode to eliminate flickering from your animated program, you may only have 16 (24) colors. Perhaps you want to draw a moving object without flicker and are willing to sacrifice colors for using double-buffer mode (maybe the object is moving so fast that the viewer won't notice the details). But when the object comes to rest, you will want to draw it in single-buffer mode so

that you can use more colors. Unfortunately, most window systems won't allow an easy switch. For example, with the X Window System, the color-display mode is an attribute of the X Visual. An X Visual must be specified before the window is created. Once it is specified, it cannot be changed for the life of the window. After you create a window with a double-buffered, RGBA display mode, you're stuck with it. A tricky solution to this problem is to create more than one window, each with a different display mode. Then you must control the visibility of the windows (for example, mapping or unmapping an X Window, or managing or unmanaging a Motif or Athena widget) and draw the object into the appropriate, visible window.

7.5 Specifying a color and a shading model

OpenGL maintains a current color (in RGBA mode) and a current color index (in color-index mode). Unless you're using a more complicated coloring model such as lighting or texture mapping, each object is drawn using the current color (or color index). Look at the following pseudocode sequence:

```
set_color(RED);
draw_item(A);
draw_item(B);
set_color(GREEN);
set_color(BLUE);
draw_item(C);
```

Items A and B are drawn in red, and item C is drawn in blue. The fourth line, which sets the current color to green, has no effect (except to waste a bit of time). With no lighting or texturing, when the current color is set, all items drawn afterward are drawn in that color until the current color is changed to something else.

7.5.1. Specifying a color in RGBA mode

In RGBA mode, use the **glColor*()** command to select a current color.

void **glColor3**{b s i f d ub us ui} (TYPEr, TYPEg, TYPEb);
void **glColor4**{b s i f d ub us ui} (TYPEr, TYPEg, TYPEb, TYPEa);

void **glColor3**{b s i f d ub us ui}**v** (const TYPE*v);
void **glColor4**{b s i f d ub us ui}**v** (const TYPE*v);
This command can have up to three suffixes, which differentiate variations of the parameters accepted. The first suffix is either 3 or 4, to indicate whether you supply an alpha value in addition to the red, green, and blue values. If you don't supply an alpha value, it's automatically set to 1.0. The second suffix indicates the data type for parameters: byte, short, integer, float, double, unsigned byte, unsigned short, or unsigned integer. The third suffix is an optional **v**, which indicates that the argument is a pointer to an array of values of the given data type.

Suffix	Data Type	Minimum Value	Min Value Maps	Max Value	Max Value Maps
b	1-byte integer	-128	-1.0	127	1.0
s	2-byte integer	-32,768	-1.0	32,767	1.0
i	4-byte integer	2,147,483,648	-1.0	2,147,483,647	1.0
ub	unsigned 1-byte integer	0	0.0	255	1.0
us	unsigned 2-byte integer	0	0.0	65,535	1.0
ui	unsigned 4-byte integer	0	0.0	4,294,967,295	1.0

Table 4-1: Converting color values to floating-point numbers.

For the versions of **glColor*()** that accept floating-point data types, the values should typically range between 0.0 and 1.0, the minimum and maximum values that can be stored in the framebuffer. Unsigned-integer color components, when specified, are linearly mapped to floating-point values such that the largest represent table value maps to 1.0 (full intensity), and zero maps to 0.0 (zero intensity). Signed-integer color components, when specified, are linearly mapped to floating-point values such that the most positive represent table value maps to 1.0, and the most negative represent table value maps to -1.0 (see Table 4-1). Neither floating-point nor signed-integer values are clamped to the range [0, 1] before updating the current color or current lighting material parameters. After lighting calculations, resulting color values outside the range [0, 1] are clamped to the range [0, 1] before they are interpolated or written into a color buffer. Even if lighting is disabled, the color components are clamped before rasterization.

7.5.2. Specifying a color in color-index mode

In color-index mode, use the **glIndex*()** command to select a single-valued color index as the current color index.
void **glIndex**{sifd ub}(TYPE c);
void **glIndex**{sifd ub}**v**(const TYPE *c);
The first suffix for this command indicates the data type for parameters: short, integer, float, double, or unsigned byte. The second, optional suffix is **v**, which indicates that the argument is an array of values of the given data type (the array contains only one value). **glClearColor()** is the clear color. For color-index mode, there is a corresponding **glClearIndex()**.
void **glClearIndex**(GLfloat cindex);
Sets the current clearing color in color-index mode. In a color-index mode window, a call to **glClear**(GL_COLOR_BUFFER_BIT) will use cindex to clear the buffer. The default clearing index is 0.0.
Note: OpenGL does not have any routines to load values into the color-lookup table. Window systems typically already have such operations. GLUT has the routine **glutSetColor()** to call the window-system specific commands.

7.5.3. Specifying a shading model

A line or a filled polygon primitive can be drawn with a single color (flat shading) or with many different colors (smooth shading, also called Gouraud shading). You specify the desired shading technique with **glShadeModel()**.

void **glShadeModel** (GLenum mode);

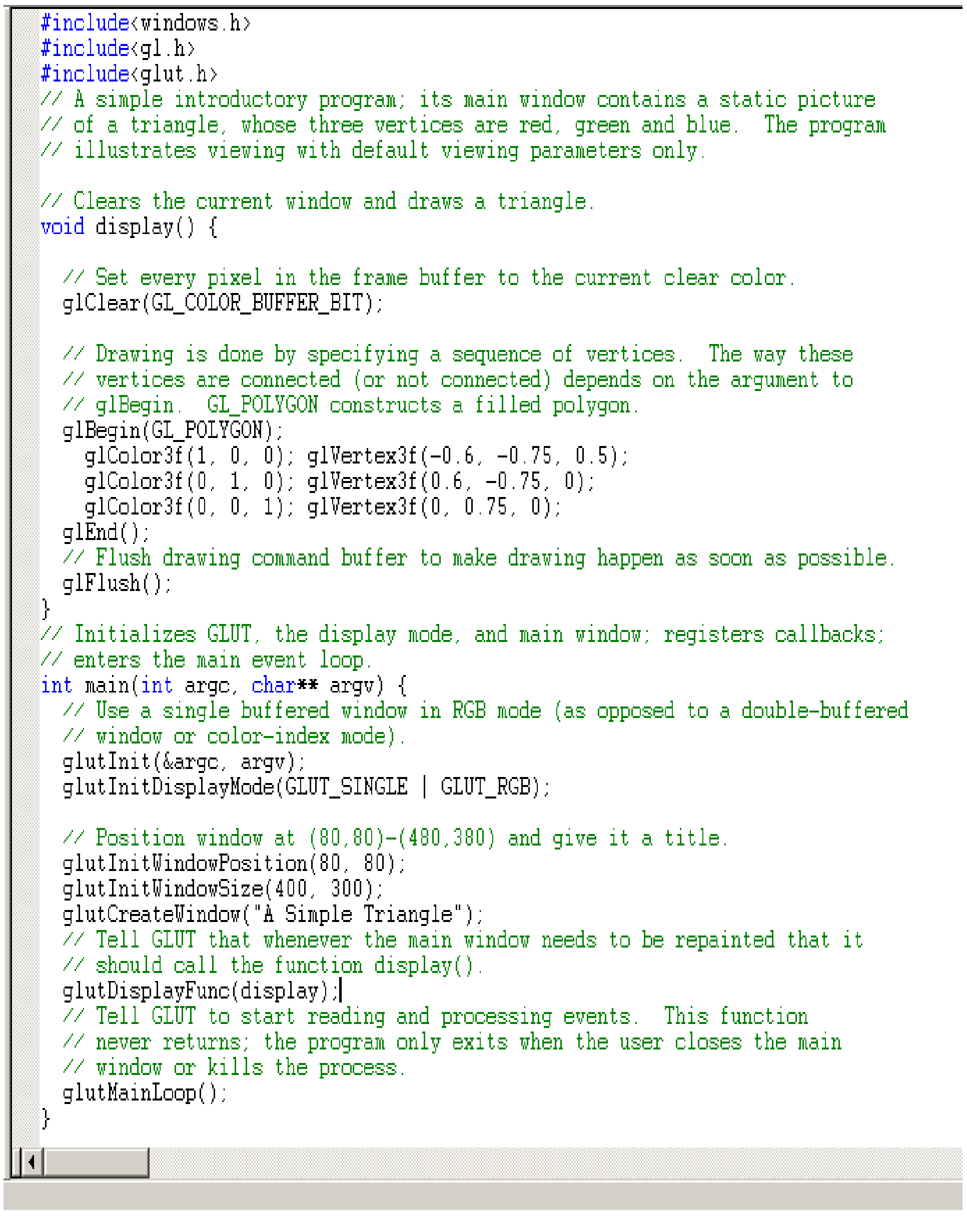
The mode parameter can be either GL_SMOOTH (the default) or GL_FLAT. With flat shading, the color of one particular vertex of an independent primitive is duplicated across all the primitive's vertices to render that primitive. With smooth shading, the color at each vertex is treated individually. For a line primitive, the colors along the line segment are interpolated between the vertex colors. For a polygon primitive, the colors for the interior of the polygon are interpolated between the vertex colors. Example 4-1 draws a smooth-shaded triangle.

Example 4-1: Drawing a smooth-shaded triangle

A simple introductory program; its main window contains a static picture of a triangle, whose three vertices are red, green and blue. The program illustrates viewing with default viewing parameters only.

With smooth shading, neighboring pixels have slightly different color values. In RGBA mode, adjacent pixels with slightly different values look similar, so the color changes across a polygon appear gradual. In color-index mode, adjacent pixels may reference different locations in the color-index table, which may not have similar colors at all. Adjacent color-index entries may contain wildly different colors, so a smooth-shaded polygon in color-index mode can look psychedelic. To avoid this problem, you have to create a color ramp of smoothly changing colors among a contiguous set of indices in the color map. Remember that loading colors into a color map is performed through your window system rather than OpenGL. If you use GLUT, you can use glutSetColor() to load a single index in the color map with specified red, green, and blue values.

The program:



```
#include<windows.h>
#include<gl.h>
#include<glut.h>
// A simple introductory program; its main window contains a static picture
// of a triangle, whose three vertices are red, green and blue.  The program
// illustrates viewing with default viewing parameters only.

// Clears the current window and draws a triangle.
void display() {

  // Set every pixel in the frame buffer to the current clear color.
  glClear(GL_COLOR_BUFFER_BIT);

  // Drawing is done by specifying a sequence of vertices.  The way these
  // vertices are connected (or not connected) depends on the argument to
  // glBegin.  GL_POLYGON constructs a filled polygon.
  glBegin(GL_POLYGON);
    glColor3f(1, 0, 0); glVertex3f(-0.6, -0.75, 0.5);
    glColor3f(0, 1, 0); glVertex3f(0.6, -0.75, 0);
    glColor3f(0, 0, 1); glVertex3f(0, 0.75, 0);
  glEnd();
  // Flush drawing command buffer to make drawing happen as soon as possible.
  glFlush();
}
// Initializes GLUT, the display mode, and main window; registers callbacks;
// enters the main event loop.
int main(int argc, char** argv) {
  // Use a single buffered window in RGB mode (as opposed to a double-buffered
  // window or color-index mode).
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

  // Position window at (80,80)-(480,380) and give it a title.
  glutInitWindowPosition(80, 80);
  glutInitWindowSize(400, 300);
  glutCreateWindow("A Simple Triangle");
  // Tell GLUT that whenever the main window needs to be repainted that it
  // should call the function display().
  glutDisplayFunc(display);
  // Tell GLUT to start reading and processing events.  This function
  // never returns; the program only exits when the user closes the main
  // window or kills the process.
  glutMainLoop();
}
```

Output:

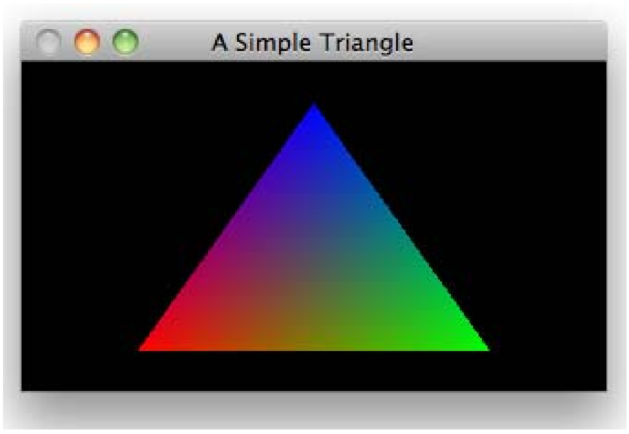


Fig.7-6: The output: a smooth-shaded triangle.

The first argument for **glutSetColor()** is the index, and the others are the red, green, and blue values. To load thirty-two contiguous color indices (from color index 16 to 47) with slightly differing shades of yellow, you might call

```
for (i = 0; i < 32; i++) {
   glutSetColor  (16+i,  1.0*(i/32.0),  1.0*(i/32.0),
0.0);
}
```

Now, if you render smooth-shaded polygons that use only the colors from index 16 to 47, those polygons have gradually differing shades of yellow. With flat shading, the color of a single vertex defines the color of an entire primitive. For a line segment, the color of the line is the current color when the second (ending) vertex is specified. For a polygon, the color used is the one that's in effect when a particular vertex is specified, as shown in Table 7-2. The table counts vertices and polygons starting from 1. OpenGL follows these rules consistently, but the best way to avoid uncertainty about how a flat-shaded primitive will be drawn is to specify only one color for the primitive.

Type of Polygon	Vertex Used to Select the Color for the i_{th} Polygon
single polygon	1
triangle strip	i+2
triangle fan	i+2
independent triangle	3i
quad strip	2i+2
independent quad	4i

Table 7-2: OpenGL selects a color for the i_{th} flat-shaded polygon.

Bibliography

1- Foley, van Dam, et al., “Computer Graphics: Principles and Practice. Reading”, MA: Addison-Wesley Developers Press, 1990.
2- The OpenGL Programmer's Guide (the Redbook), Addison-Wesley
3- F. S. Hill, Jr. and S. Kelley, "Computer Graphics using OpenGL", 3nd edition, Prentice Hall 2001.
4- A. Watt, "3D Computer Graphics", 3rd edition, Addison Wesley – Pearson Education, 2000.
5- E Angel, "Interactive Computer Graphics: A top-Down approach with OpenGL", 3rd edition, Addison Wesley, 2003.
6- P. Egerton & W. Hall, "Computer Graphics: Mathematical First Steps", Prentice Hall - Pearson Education, 1999.
7- R. Hearn and M.P. Baker, "Computer Graphics with OpenGL", 3rd ed, Prentice Hall - Pearson Education, 2004.

Exercises

1- What is difference between RGB and RGBA color?
2- How OpenGL selects a color for the i^{th} flat-shaded polygon?
3- How do you change display mode?
4- What is the computer color?
5- Drawing a smooth-shaded rectangular?
6- How do you specifying a color in RGBA mode?
7- How do you specifying a color and a shading model?
8- What colors look the brightest to the human eye? Which are the least bright?
9- Name at least one limitation of the RGB color space.
10- What is an example of an additive color space?
11- How much dynamic range does an average computer monitor have? How much can real world scenes have? (Just an orders of magnitude answer here is fine)
12- How can we display high dynamic range images on a regular display?
13- Copy, compile and run the following the program.

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include "glut.h"
void display() {
      glClear(GL_COLOR_BUFFER_BIT);
      glBegin(GL_POLYGON);
            glVertex2f(-0.75, 0.5);
            glVertex2f(0.75, 0.5);
            glVertex2f(0.75, -0.5);
            glVertex2f(-0.75, -0.5);
      glEnd();
      glFlush();
  }

  int main(int argc, char* argv[]) {
        glutInit(&argc, argv);
        glutCreateWindow("Simple OpenGL Program");
        glutDisplayFunc(display);
        glutMainLoop();
    }
```

- Modify the program to do each of the following:
    - **a.** Identify the functionality of each line in the program:
    - **b.** position the window at position (250, 100) with width 200 and height 100
    - **c.** draw a red shape on a blue background
    - **d.** draw a square
    - **e.** draw a triangle
    - **f.** In the glClearColor( ), try changing the first 3 numbers to see that they are R, G and B. (Each color lies in the range 0 to 1).
    - **g.** Change the values in glColor3f( ) to draw in different colors.

14- Experiment program in Eq.(13) to draw multiple shapes, points, lines, line strips, line loops, triangles, triangle strips, triangle fans, quads and quad strips using various colors.

Solved Problems

1- What is the vertex shader?

Solution

The *vertex shader* is a small program running on your graphics card that processes every one of these input vertices individually. This is where the perspective transformation takes place, which projects vertices with a 3D world position onto your 2D screen! It also passes important attributes like color and texture coordinates further down the pipeline.

2- What is the geometry shader?

Solution

The following step, the *geometry shader*, is completely optional and was only recently introduced. Unlike the vertex shader, the geometry shader can output more data than comes in. It takes the primitives from the shape assembly stage as input and can either pass a primitive through down to the rest of the pipeline, modify it first, completely discard it or even replace it with other primitive(s). Since the communication between the GPU and the rest of the PC is relatively slow, this stage can help you reduce the amount of data that needs to be transferred.

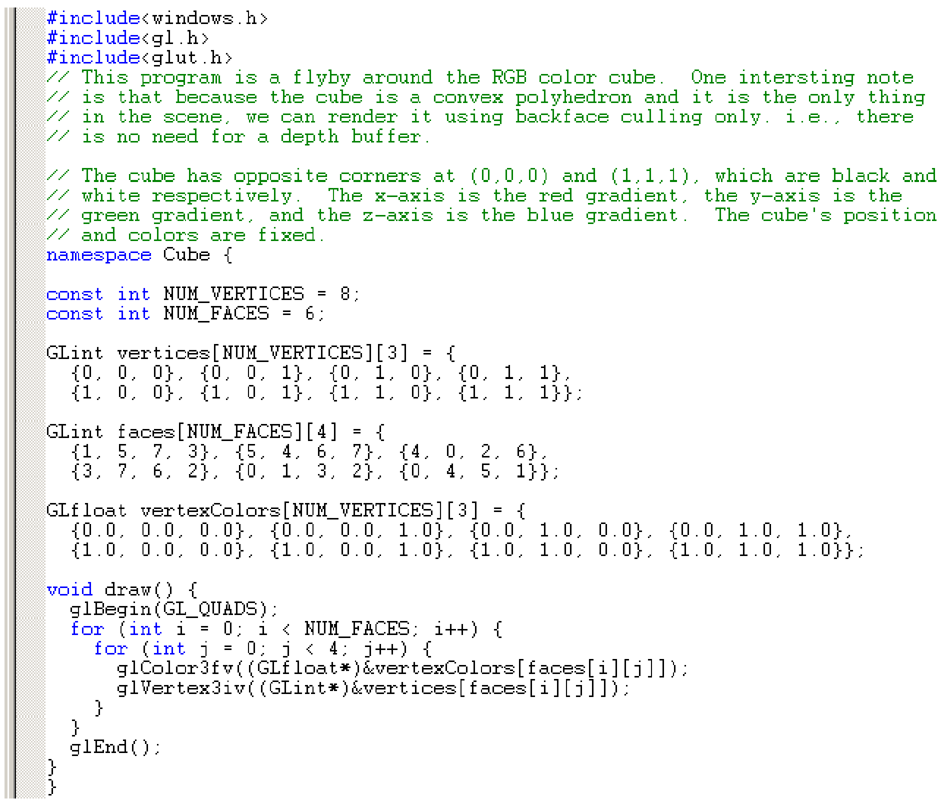
3- What is meant by the fragment shader processes?

Solution

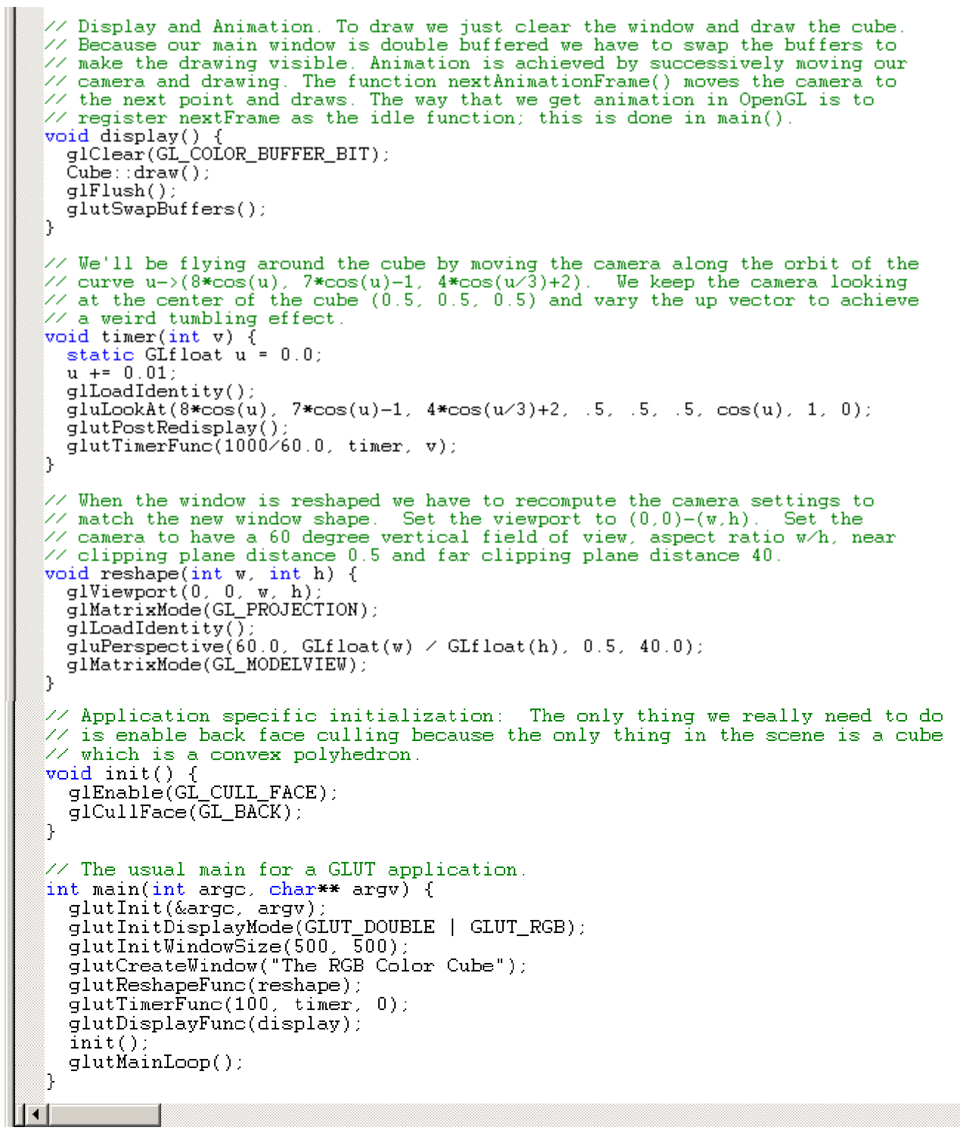
The *fragment shader* processes each individual fragment along with its interpolated attributes and should output the final color. This is usually done by sampling from a texture using the interpolated texture coordinate vertex attributes or simply outputting a color. In more advanced scenarios, there could also be calculations related to lighting and shadowing and special effects in this program. The shader also has the ability to discard certain fragments, which means that a shape will be see-through there.

4- Write an OpenGl program to flyby around the RGB color cube and render it using back face culling only.

The program:



```
#include<windows.h>
#include<gl.h>
#include<glut.h>
// This program is a flyby around the RGB color cube.  One intersting note
// is that because the cube is a convex polyhedron and it is the only thing
// in the scene, we can render it using backface culling only. i.e., there
// is no need for a depth buffer.

// The cube has opposite corners at (0,0,0) and (1,1,1), which are black and
// white respectively.  The x-axis is the red gradient, the y-axis is the
// green gradient, and the z-axis is the blue gradient.  The cube's position
// and colors are fixed.
namespace Cube {

const int NUM_VERTICES = 8;
const int NUM_FACES = 6;

GLint vertices[NUM_VERTICES][3] = {
  {0, 0, 0}, {0, 0, 1}, {0, 1, 0}, {0, 1, 1},
  {1, 0, 0}, {1, 0, 1}, {1, 1, 0}, {1, 1, 1}};

GLint faces[NUM_FACES][4] = {
  {1, 5, 7, 3}, {5, 4, 6, 7}, {4, 0, 2, 6},
  {3, 7, 6, 2}, {0, 1, 3, 2}, {0, 4, 5, 1}};

GLfloat vertexColors[NUM_VERTICES][3] = {
  {0.0, 0.0, 0.0}, {0.0, 0.0, 1.0}, {0.0, 1.0, 0.0}, {0.0, 1.0, 1.0},
  {1.0, 0.0, 0.0}, {1.0, 0.0, 1.0}, {1.0, 1.0, 0.0}, {1.0, 1.0, 1.0}};

void draw() {
  glBegin(GL_QUADS);
  for (int i = 0; i < NUM_FACES; i++) {
    for (int j = 0; j < 4; j++) {
      glColor3fv((GLfloat*)&vertexColors[faces[i][j]]);
      glVertex3iv((GLint*)&vertices[faces[i][j]]);
    }
  }
  glEnd();
}
}
```



```
// Display and Animation. To draw we just clear the window and draw the cube.
// Because our main window is double buffered we have to swap the buffers to
// make the drawing visible. Animation is achieved by successively moving our
// camera and drawing. The function nextAnimationFrame() moves the camera to
// the next point and draws. The way that we get animation in OpenGL is to
// register nextFrame as the idle function; this is done in main().
void display() {
  glClear(GL_COLOR_BUFFER_BIT);
  Cube::draw();
  glFlush();
  glutSwapBuffers();
}

// We'll be flying around the cube by moving the camera along the orbit of the
// curve u->(8*cos(u), 7*cos(u)-1, 4*cos(u/3)+2).  We keep the camera looking
// at the center of the cube (0.5, 0.5, 0.5) and vary the up vector to achieve
// a weird tumbling effect.
void timer(int v) {
  static GLfloat u = 0.0;
  u += 0.01;
  glLoadIdentity();
  gluLookAt(8*cos(u), 7*cos(u)-1, 4*cos(u/3)+2, .5, .5, .5, cos(u), 1, 0);
  glutPostRedisplay();
  glutTimerFunc(1000/60.0, timer, v);
}

// When the window is reshaped we have to recompute the camera settings to
// match the new window shape.  Set the viewport to (0,0)-(w,h).  Set the
// camera to have a 60 degree vertical field of view, aspect ratio w/h, near
// clipping plane distance 0.5 and far clipping plane distance 40.
void reshape(int w, int h) {
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(60.0, GLfloat(w) / GLfloat(h), 0.5, 40.0);
  glMatrixMode(GL_MODELVIEW);
}

// Application specific initialization:  The only thing we really need to do
// is enable back face culling because the only thing in the scene is a cube
// which is a convex polyhedron.
void init() {
  glEnable(GL_CULL_FACE);
  glCullFace(GL_BACK);
}

// The usual main for a GLUT application.
int main(int argc, char** argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
  glutInitWindowSize(500, 500);
  glutCreateWindow("The RGB Color Cube");
  glutReshapeFunc(reshape);
  glutTimerFunc(100, timer, 0);
  glutDisplayFunc(display);
  init();
  glutMainLoop();
}
```

Output:

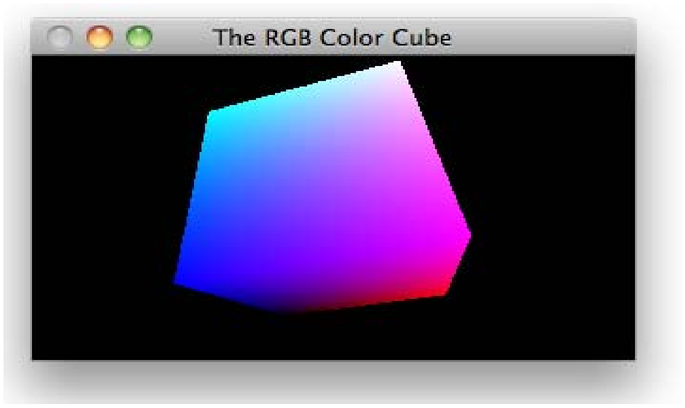


Fig.7-7: The output: RGB color cube.