Specifying Colors in OpenGL

OpenGL has two color modes — the RGBA mode we have already seen, and color index mode.

In RGBA mode, a color is specified by three intensities (for the Red, Green, and Blue components of the color) and optionally a fourth value, Alpha, which controls transparency.

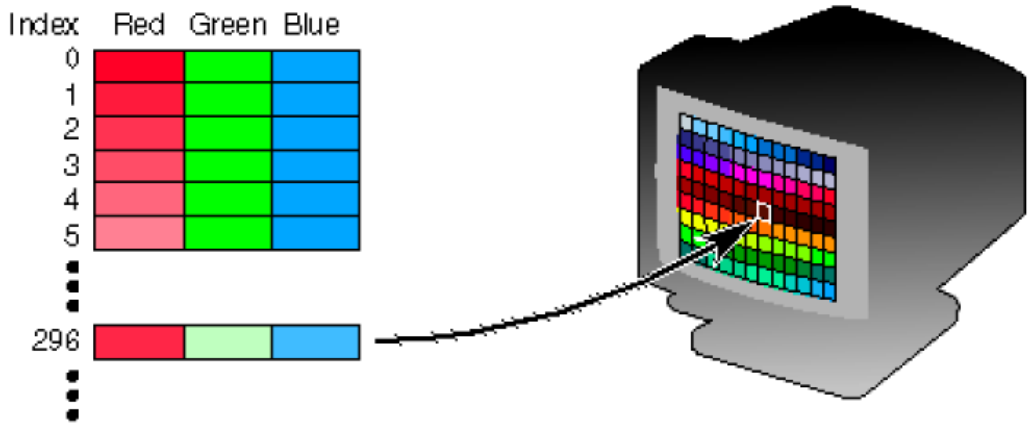
The function `glColor4f(red, green, blue, alpha)`

maps available red, green, and blue intensities onto (0.0, 1.0) where 0.0 means that the color component is absent ((0.0, 0.0, 0.0) is black) and 1.0 is a saturated color ((1.0, 1.0, 1.0) is white.)

The number of bits used to represent each color depends upon the graphics card. Current graphics cards have several Mbytes of memory, and use 24 or 32 bits for color (24-bit: 8 bits per color; 32-bit: 8 bits per color + 8 bits padding or transparency). The term bitplane refers to an image of single-bit values. Thus, a system with 24 bits of color has 24 *bitplanes*.

Not long ago, 8 to 16 bits of color was usual, permitting only 256 to 64K individual colors. In order to increase the range of colors displayed, a color lookup table was often used. In the table, colors would be represented by, $m-$bit entries (e.g. $m = 24$). The length of this table was 2^n, with colors selected

by an n-bit index.



For a system with 8 bitplanes, $2^8 = 256$ different colors could be displayed simultaneously. These 256 colors could be selected from a set of 2^m colors. For $m = 24$ this gives $2^24 \approx 16$ million colours. The color table could be altered to give different palettes of colors.

OpenGL supports this mode, called color index mode.

The command `glIndexf(cindex)` sets the current color index to `cindex`.

Analogous to `glClearColor()` there is a corresponding `glClearIndex(clearindex)` which sets the clearing color to `clearindex`.

OpenGL has no function to set a color lookup table, but `GLUT` has the function
`glutSetColor(GLint index, GLfloat red, GLfloat green, GLfloat blue)` which sets the color at `index` to the intensities specified by `red`, `green`, and `blue`.

The function `glGetIntegerv()` with arguments `GL_RED_BITS`, `GL_GREEN_BITS`, `GL_BLUE_BITS`, `GL_ALPHA_BITS`, and `GL_INDEX_BITS` allows you to determine the number of bits for the identified entity.

Color-index mode does not support transparency.

Choosing between color modes

In general, RGBA mode provides more flexibility, and is fully supported by current hardware.

Color index mode may be useful when:

- porting or modifying an existing program using color-index mode
- only a small number of bitplanes are available (e.g., 8 bits — possibly 3 for red, 3 for green, and 2 for blue, giving a very small range for each color)
- drawing layers or color-map animation are used.

Shading models

OpenGL has two shading models; primitives are drawn with a single color (flat shading) or with a smooth color variation from vertex to vertex (smooth shading, or Gouraud shading).

The shading mode is specified with `glShadeModel(mode)` where `mode` is either `GL_SMOOTH` or `GL_FLAT`.

In RGBA mode, the color variation is obtained by interpolating the RGB values. The interpolation is on each of the R, G, and B components, and is in the horizontal and vertical directions.

In color index mode, the index values are interpolated, so the color table must be loaded with a set of smoothly changing colors.

For flat shading, the color of a single vertex determines the color of the polygon.

For the situation where lighting elements are present, a more elaborate shading algorithm can be used. (This will be discussed later.)

The `smooth` program shows a simple example of smooth shading. Here is the relavent code from `smooth.c`.

```
void init(void) {
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_SMOOTH);
}
void triangle(void)
{
   glBegin (GL_TRIANGLES);
   glColor3f (1.0, 0.0, 0.0);
   glVertex2f (5.0, 5.0);
   glColor3f (0.0, 1.0, 0.0);
   glVertex2f (25.0, 5.0);
   glColor3f (0.0, 0.0, 1.0);
   glVertex2f (5.0, 25.0);
   glEnd();
}
void display(void) {
   glClear (GL_COLOR_BUFFER_BIT);
   triangle ();
   glFlush ();
}
```

Transparency

So far, we have ignored the `alpha` term.

The natural interpretation of `alpha` is opacity. When `alpha` is large, the polygon is opaque; when small, it is translucent.

For `alpha` to have any meaning, blending must be set.

This is done with `glEnable(GL_BLEND)`

Next, the way the blending is to be done is specified. Basically, the colors of the incoming polygon (the source) are combined with the currently stored pixel value (the destination) to give a resultant pixel color as follows:

$(R_s S_r + R_d D_r, G_s S_g + G_d D_g, B_s S_b + B_d D_b, A_s S_a + A_d D_a)$

where (S_r, S_g, S_b) and (D_r, D_g, D_b) are blending factors for the source and destination red, green, and blue components, respectively.

Blending factors have values between 0 and 1, and the resulting pixel values are clamped to [0,1].

The blending factors are set with the function

`glBlendFunc(GLenum sfactor, GLenum dfactor)`

The following table gives values for `sfactor` and `dfactor`:

Constant	`factor`	blend factor
`GL_ZERO`*	`s` or `d`	$(0, 0, 0, 0)$
`GL_ONE`*	`s` or `d`	$(1, 1, 1, 1)$
`GL_DST_COLOR`	`s`	(R_d, G_d, B_d, A_d)
`GL_SRC_COLOR`	`d`	(R_s, G_s, B_s, A_s)
`GL_ONE_MINUS_DST_COLOR`	`s`	$(1, 1, 1, 1) - (R_d, G_d, B_d, A_d)$
`GL_ONE_MINUS_SRC_COLOR`	`d`	$(1, 1, 1, 1) - (R_s, G_s, B_s, A_s)$
`GL_SRC_ALPHA`*	`s` or `d`	(A_s, A_s, A_s, A_s)
`GL_DST_ALPHA`	`s` or `d`	(A_d, A_d, A_d, A_d)
`GL_ONE_MINUS_SRC_ALPHA`*	`s` or `d`	$(1, 1, 1, 1) - (A_s, A_s, A_s, A_s)$
`GL_ONE_MINUS_DST_ALPHA`	`s` or `d`	$(1, 1, 1, 1) - (A_d, A_d, A_d, A_d)$
`GL_SRC_ALPHA_SATURATE`	`s`	$(f, f, f, 1)$, $f = \min(A_s, 1 - A_d)$

* — The most commonly used factors.

Blending can be disabled with `glDisable(GL_BLEND)`

The same effect can be achieved by using `GL_ONE` for the source and `GL_ZERO` for the destination.

(In fact, this is the default setting when blending is enabled!)

The program `glSandBox` is useful for illustrating blending.

Sample Uses of Blending:

- To draw a picture composed half of one image and half of another, equally blended, first set the source factor to `GL_ONE` and the destination factor to `GL_ZERO`, and draw the first image. Then set the source factor to `GL_SRC_ALPHA` and destination factor to `GL_ONE_MINUS_SRC_ALPHA`, and draw the second image with `alpha` equal to 0.5.
  This probably is the most common blending operation.

- The blending functions using the source or destination colors `GL_DST_COLOR` or `GL_ONE_MINUS_DST_COLOR` for the source factor and `GL_SRC_COLOR` or `GL_ONE_MINUS_SRC_COLOR` for the destination factor effectively allow modification of each color component individually.

  This operation is equivalent to applying a simple filter for example, multiplying the red component by 80 percent, the green component by 40 percent, and the blue component by 72 percent would simulate viewing the scene through a photographic filter that blocks 20 percent of red light, 60 percent of green, and 28 percent of blue.

The program `alpha.c` shows a simple application of blending. Two intersecting triangle are drawn, and the overlap is blended.

The code to setup blending is:

```
static void init(void)
{
   glEnable (GL_BLEND);
   glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
   glShadeModel (GL_FLAT);
   glClearColor (0.0, 0.0, 0.0, 0.0);
}
```

Note that only the state of the system is changed; the drawing functions are used normally.

The program allows the user to toggle which of the two overlapping triangles is on top.

The code to draw one triangle is:

```
static void drawLeftTriangle(void)
{
   /* draw yellow triangle on LHS of screen */
   glBegin (GL_TRIANGLES);
      glColor4f(1.0, 1.0, 0.0, 0.75);
      glVertex3f(0.1, 0.9, 0.0);
      glVertex3f(0.1, 0.1, 0.0);
      glVertex3f(0.7, 0.5, 0.0);
   glEnd();
}
```

Similar code draws the right (cyan) triangle, also at an alpha value of 0.75.