OpenGL Lighting Model (taken from the Redbook)

When you look at a physical surface, your eye's perception of the color depends on the distribution of photon energies that arrive and trigger your cone cells. Those photons come from a light source or combination of sources, some of which are absorbed and some of which are reflected by the surface. In addition, different surfaces may have very different properties- some are shiny and preferentially reflect light in certain directions, while others scatter incoming light equally in all directions. Most surfaces are somewhere in between.

OpenGL approximates light and lighting as if light can be broken into red, green, and blue components. Thus, the color of a light source is characterized by the amounts of red, green, and blue light it emits, and the material of a surface is characterized by the percentages of the incoming red, green, and blue components that are reflected in various directions. The OpenGL lighting equations are just approximations but ones that work fairly well and can be computed relatively quickly. If you desire a more accurate (or just different) lighting model, you have to do your own calculations in software. Such software can be enormously complex, as a few hours of reading any optics textbook should convince you.

In the OpenGL lighting model, the light in a scene comes from several light sources that can be individually turned on and off. Some light comes from a particular direction or position, and some light is generally scattered about the scene. For example, when you turn on a light bulb in a room, most of the light comes from the bulb, but some light comes after bouncing off one, two, three, or more walls. This bounced light (called *ambient* light) is assumed to be so scattered that there is no way to tell its original direction, but it disappears if a particular light source is turned off.

Finally, there might be a general ambient light in the scene that comes from no particular source, as if it had been scattered so many times that its original source is impossible to determine.

In the OpenGL model, the light sources have effects only when there are surfaces that absorb and reflect light. Each surface is assumed to be composed of a material with various properties. A material might emit its own light (such as headlights on an automobile), it might scatter some incoming light in all directions, and it might reflect some portion of the incoming light in a preferential direction (such as a mirror or other shiny surface).

The OpenGL lighting model considers the lighting to be divided into four independent components: ambient, diffuse, specular, and emissive. All four components are computed independently and then added together.

Ambient, **Diffuse**, **Specular, and Emissive Light**

Ambient illumination is light that's been scattered so much by the environment that its direction is impossible to determine- it seems to come from all directions. Backlighting in a room has a large ambient component, since most of the light that reaches your eye has first bounced off many surfaces. A spotlight outdoors has a tiny ambient component; most of the light travels in the same direction, and since you're outdoors, very little of the light reaches your eye after bouncing off other objects. When ambient light strikes a surface, it's scattered equally in all directions.

The diffuse component is the light that comes from one direction, so it's brighter if it comes squarely down on a surface than if it barely glances off the surface. Once it hits a surface, however, it's scattered equally in all directions, so it appears equally bright, no matter where the eye is located. Any light coming from a particular position or direction probably has a diffuse component.

Specular light comes from a particular direction, and it tends to bounce off the surface in a preferred direction. A well-collimated laser beam bouncing off a high-quality mirror produces almost 100 percent specular reflection. Shiny metal or plastic has a high specular component, and chalk or carpet has almost none. You can think of specularity as shininess.

In addition to ambient, diffuse, and specular colors, materials may have an *emissive* color, which simulates light originating from an object. In the OpenGL lighting model, the emissive color of a surface adds intensity to the object, but is unaffected by any light sources. Also, the emissive color does not introduce any additional light into the overall scene.

Although a light source delivers a single distribution of frequencies, the ambient, diffuse, and specular components might be different. For example, if you have a white light in a room with red walls, the scattered light tends to be red, although the light directly striking objects is white. OpenGL allows you to set the red, green, and blue values for each component of light independently.

Material Colors

The OpenGL lighting model makes the approximation that a material's color depends on the percentages of the incoming red, green, and blue light it reflects., For example, a perfectly red ball reflects all the incoming red light and absorbs all the green and blue light that strikes it. If you view such a ball in white light (composed of equal amounts of red, green, and

blue light), all the red is reflected, and you see a red ball. If the ball is viewed in pure red light, it also appears to be red. If, however, the red ball is viewed in pure green light, it appears black (all the green is absorbed, and there's no incoming red, so no light is reflected).

Like lights, materials have different ambient, diffuse, and specular colors, which determine the ambient, diffuse, and specular reflectances of the material. A material's ambient reflectance is combined with the ambient component of each incoming light source, the diffuse reflectance with the light's diffuse component, and similarly for the specular reflectance and component. Ambient and diffuse reflectances define the color of the material and are typically similar if not identical. Specular reflectance is usually white or gray, so that specular highlights end up being the color of the light source's specular intensity. If you think of a white light shining on a shiny red plastic sphere, most of the sphere appears red, but the shiny highlight is white.

RGB Values for Lights and Materials

The color components specified for lights mean something different than for materials. For a light, the numbers correspond to a percentage of full intensity for each color. If the R, G, and B values for a light's color are all 1.0, the light is the brightest possible white. If the values are 0.5, the color is still white, but only at half intensity, so it appears gray. If R = G = 1 and B = 0 (full red **and** green with no blue), the light appears yellow.

For materials, the numbers correspond to the reflected proportions of those colors. So if R = 1, G = 0.5, and B = 0 for a material, that material reflects all the incoming red light, half the incoming green light, and none of the incoming blue light. In other words, if an OpenGL light has components (LR, LG, LB), and a material has corresponding components (MR, MG, MB), then, ignoring all other reflectivity effects, the light that arrives at the eye is given by (LR * MR, LG * MG, LB * MB).

Similarly, if you have two lights that send (R1, G1, B1) and (R2, G2, B2) to the eye, OpenGL adds the components, giving (R1 + R2, G1 + G2, B1 + B2). If any of the sums is greater than 1 (corresponding to a color brighter than the equipment can display), the component is clamped to 1.

These are the steps required to add lighting to your scene.

1. Define normal vectors for each vertex of every object. These normals determine the orientation of the object relative to the light sources.
2. Create, select, and position one or more light sources.
3. Create and select a *lighting model,* which defines the level of global ambient light and the effective location of the viewpoint (for the purposes of lighting calculations).
4. Define material properties for the objects in the scene.

Define Normal Vectors for Each Vertex of Every Object

An object's normals determine its orientation relative to the light sources. For each vertex, OpenGL uses the assigned normal to determine how much light that particular vertex receives from each light source. In this example, the normals for the sphere are defined as part of the **glutSolidSphere( )** routine.

For proper lighting, surface normals must be of unit length. You must also be careful that the modelview transformation matrix does not scale the surface normal, so that the resulting normal is no longer of unit length. To ensure that normals are of unit length, you may need to call **glEnable( )** with GL_NORMALIZE or GL_RESCALE_NORMAL as a parameter.

GL_RESCALE_NORMAL causes each component in a surface normal to be multiplied by the same value, determined from the modelview transformation matrix. Therefore, it works correctly only if the normal was scaled uniformly and was a unit-length vector to begin with. GL_NORMALIZE is a more thorough operation than GL _RESCALE_NORMAL. When GL_NORMALIZE is enabled, the length of the normal vector is calculated, and then each component of the normal is divided by the calculated length. This operation guarantees that the resulting normal is of unit length, but may be more expensive than simply rescaling normals

Creating Light Sources

Light sources have several properties, such as color, position, and direction. The following sections explain how to control these properties and what the resulting light looks like. The command used to specify all properties of lights is **glLight*().**

It takes three arguments: to identify the light whose property is being specified, the property, and the desired value for that property.

void glLight{if}(GLenum *light, Glenum pname, TYPE param);*
void glLight{if}v(GLenum light, *GLenum pname, TYPE * param);*

Creates the light specified by *light*, which can be GL_LIGHT0, GL_LIGHT1, ... , or GL_LIGHT7. The characteristic of the light being set is defined by *pname*, which specifies a named parameter (see Table 5-1). *param* indicates the values to which the *pname* characteristic is set; it's a pointer to a group of values if the vector version is used, or the value itself if the nonvector version is used. The nonvector version can be used to set only single-valued light characteristics.

Parameter Name	**Default Values**	**Meaning**
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	ambient intensity of light
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0) or	diffuse intensity of light (default for light 0 is white;
	(0.0, 0.0, 0.0, 1.0)	for other lights, black)
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0) or (0.0, 0.0, 0-0, 1-0)	specular intensity of light (default for light 0 is white; for other lights, black)
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	*(x, y, z, w)* position of light
Gl,_SPOT_DIRECTION	(0.0, 0.0, -1.0)	*(x, y, z)* direction of spotlight
GL_SPOT_EXPONENT	0.0	spotlight exponent
GL_SPOT_CUTOFF	180.0	spotlight cutoff angle
GL_CONSTANT_ATTENUATION	1.0	constant attenuation factor
GL_LINEAR_ATTENUATION	0.0	linear attenuation factor
GL_QUADRATIC_ATTENUATION	0.0	quadratic attenuation factor

Table 5-1 Default Values for *pname* Parameter of glLight*()

Note: The default values listed for GL_DIFFUSE and GL_SPECULAR in Table 5-1 differ from GL_LIGHT0 to other lights (GL_LIGHTI, GL_LIGHT2, etc.). For GL_LIGHT0, the default value.is (1.0, 1.0, 1.0, 1.0) for both GL_DIFFUSE and GL_SPECULAR. For other lights, the default value is (0.0, 0.0, 0.0, 1.0) for the same light source properties.

Example 5-2 shows how to use gLLight*()

Example 5-2 Defining Colors and Position for a Light Source

```
GLfloat light_ambient[] = ( 0.0, 0.0, 0.0, 1.0};
GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0};
GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0};
GLfloat light_position[] = ( 1.0, 1.0, 1.0, 0.0};

glLightfv (GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv (GL_LIGHT0, GL_SPECULAR, light_specular);
glLightfv (GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv (GL_LIGHT0, GL_POSITION, light_position);
```

Spotlights

As previously mentioned, you can have a positional light source act as a spotlight-that is, by restricting the shape of the light it emits to a cone. To create a spotlight, you need to determine the spread of the cone of light you desire. (Remember that since spotlights are positional lights, you also have to locate them where you want them. Again, note that nothing prevents you from creating a directional spotlight, but it won't give you the result you want.) To specify the angle between the axis of the cone and a ray along the edge of the cone, use the GL_SPOT_CUTOFF parameter. The angle of the cone at the apex is then twice this value.

Note that no light is emitted beyond the edges of the cone. By default, the spotlight feature is disabled because the GL_SPOT_CUTOFF parameter is 180.0. This value means that light is emitted in all directions (the angle at the cone's apex is 360 degrees, so it isn't a cone at all). The value for GL_SPOT_CUTOFF is restricted to the range [0.0, 90.0] (unless it has the special value of 180.0). The following line sets the cutoff parameter to 45 degrees:

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);
```

You also need to specify a spotlight's direction, which determines the axis of the cone of light:

```
GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
```

The direction is specified in object coordinates. By default, the direction is (0.0, 0.0, -1.0), so if you don't explicitly set the value of GL_SPOT_DIRECTION, the light points down the negative z-axis. Also, keep in mind that a spotlight's direction is transformed by the modelview matrix just as though it were a normal vector, and the result is stored in eye coordinates. (See "Controlling a Light's Position and Direction" for more information about such transformations.)

In addition to the spotlight's cutoff angle and direction, there are two ways you can control the intensity distribution of the light within the cone. First you can set the attenuation factor described earlier, which is multiplied by the light's intensity. You can also set the GL_SPOT_EXPONENT parameter, which by default is zero, to control how concentrated the light is. The light's intensity is highest in the center of the cone. It's attenuated toward the edges of the cone by the cosine of the angle between the direction of the light and the direction from the light to the vertex being lit, raised to the power of the spot exponent. Thus, higher spot exponents result in a more focused light source.

```
GLfloat lightl_ambient[] = {0.2, O.2, 0.2, 1.0};
GLfloat lightl_diffuse[] = { 1.0, 1.0, 1.0, 1.0};
GLfloat lightl_specular[] = { 1.0, 1.0, 1.0, 1.0};
GLfloat lightl_position[] = { -2.0, 2.0, 1.0, 1.0};
GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };
glLightfv(GL_LIGHT1, GL_AMBIENT, lightl_ambient);
glLightfv(GL_LIGHT1, GL_DIFFUSE, lightl_diffuse);
glLightfv(GL_LIGHT1, GL_SPECULAR, lightl_specular);
glLightfv(GL_LIGHT1, GL_POSITION, lightl_position);
glLightf(GL_LIGHT1, GL_CONSTANT_ATTENUATION, 1.5);
glLightf(GL_LIGHT1, GL_LINEAR_ATTENUATION, 0.5);
glLightf(GL_LIGHT1, GL_QUADRATIC_ATTENUATION, 0.2);
glLightf(GL_LIGHT1, GL_SPOT_CUTOFF, 45.0);
glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, spot_direction);
glLightf(GL_LIGHT1, GL_SPOT_EXPONENT, 2.0);
glEnable(GL_LIGHT1);
```

Controlling a Light's Position and Direction

OpenGL treats the position and direction of a light source just as it treats the position of a geometric primitive. In other words, a light source is subject to the same matrix transformations as a primitive. More specifically, when glLight*( ) is called to specify the position or the direction of a light source, the position or direction is transformed by the current modelview matrix and stored in eye coordinates. This means you can manipulate a light source's position or direction by changing the contents of the modelview matrix. (The projection matrix has no effect on a light's position or direction.) This section explains how to achieve the following three different effects by changing the point in the program at which the light position is set, relative to modeling or viewing transformations:

• A light position that remains fixed

• A light that moves around a stationary object

• A light that moves along with the viewpoint

Keeping the Light Stationary

To achieve a stationary light, you need to set the light position after whatever viewing and/or modeling transformation you use. Example 5-4 shows how the relevant code from the init() and reshape() routines might look.

Example 5-4 Stationary Light Source

```
glViewport(0, 0, (GLsizei) w, (GLsizei) h);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (w <= h)
    glOrtho(-1.5, 1.5, -1.5*h/w, 1.5*h/w, -10.0, 10.0);
else
    glOrtho(-1.5*w/h, 1.5*w/h, -1.5, 1.5, -10.0, 10.0);
gLMatrixMode(GL_MODELVIEW);
glLoadIdentity();

/* later in init() */
GLfloat light_position[] = {1.0, 1.0, 1.0, 1.0 };
glLightfv(GL_LIGHT0, GL_POSITION, position);
```

As you can see, the viewport and projection matrices are established first. Then, the identity matrix is loaded as the modelview matrix, after which the light position is set. Since the identity matrix is used, the originally specified light position (1.0, 1.0, 1.0) isn't changed by being multiplied by the modelview matrix. Then, since neither the light position nor the modelview matrix is modified after this point, the direction of the light remains (1.0, 1.0, 1.0).

Independently Moving the Light

Now suppose you want to rotate or translate the light position so that the light moves relative to a stationary object. One way to do this is to set the light position after the modeling transformation, which is itself changed specifically to modify the light position. You can begin with the same series of calls in **init( )** early in the program. Then you need to perform the desired modeling transformation (on the modelview stack) and reset the light position, probably in **display().** Example 5-5 shows what **display()** might be.

Example 5-5 Independently Moving Light Source

```
static Gldouble spin;
void display (void)
{
       Glfloat light_position [] = { 0.0, 0.0, 1.5, 1.0 };
       glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       glPushMatrix();
              gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
              glPushMatrix();
                     glRotated (spin, 1.0, 0.0, 0.0);
                     glLightfv (GL_LIGHT0, GL_POSITION, light_position);
              glPopMatrix();
              glutSolidTorus (0.275, 0.85, 8, 15);
       glPopMatrix();
       glFlush();
}
```

spin is a global variable and is probably controlled by an input device. **display()** causes the scene to be redrawn with the light rotated *spin* degrees around a stationary torus. Note the two pairs of **glPushMatrix()** and **glPopMatrix()** calls, which are used to isolate the viewing and modeling transformations, all of which occur on the modelview stack. Since in Example 5-5 the viewpoint remains constant, the current matrix is pushed down the stack and then the desired viewing transformation is loaded with **gluLookAt( )**. The matrix stack is pushed again before the modeling transformation **glRotated()** is specified. Then the light position is set in the new, rotated coordinate system so that the light itself appears

to be rotated from its previous position. (Remember that the light position is stored in eye coordinates, which are obtained after transformation by the modelview matrix.) After the rotated matrix is popped off the stack, the torus is drawn. Example 5-6 is a program that rotates a light source around an object. When the left mouse button is pressed, the light position rotates an additional 30 degrees. A small, unlit, wireframe cube is drawn to represent the position of the light in the scene.

Example 5-6 Moving a Light with Modeling Transformations: movelight.c

```
#include <GL/glut.h>
#include <stdlib.h>

static int spin = 0;

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_SMOOTH);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
}
/* Here is where the light position is reset after the modeling
   transformation (glRotated) is called. This places the
   light at a new position in world coordinates. The cube
   represents the position of the light.
*/
void display(void)
{

   GLfloat position[] = {0.0, 0.0, 1.5, 1.0};

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
   glTranslatef(0.0, 0.0, -5.0);
   glPushMatrix();
   glRotated((GLdouble) spin, 1.0, 0.0, 0.0);
   glLightfv(GL_LIGHT0, GL_POSITION, position);
   glTranslated(0.0, 0.0, 1.5);
   glDisable(GL_LIGHTING);
   glColor3f(O.0, 1.0, 1.0);
   glutWireCube(O.1);
   glEnable(GL_LIGHTING);
   glPopMatrix();
   glutSolidTorus(0.275, 0.85, 8, 15);
   glPopMatrix();
   glFlush();
}
void reshape (int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(40.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
void mouse (int button, int state, int x, int y)
{
   switch (button) {
      case GLUT_LEFT_BUTTON:
         if (state == GLUT_DOWN)
            spin = (spin + 30) % 360;
            glutPostRedisplay();
        break;
```

```
     default:
        break;
   }
}
int main(int argc, char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(500, 500);
  glutInitWindowPosition(100, 100);
  glutCreateWindow(argv[0]);
  init();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutMouseFunc(mouse);
  glutMainLoop();
  return 0;
}
```

Moving the Light Source Together with the Viewpoint

To create a light that moves along with the viewpoint, you need to set the light position before the viewing transformation. Then the viewing transformation affects both the light and the viewpoint in the same way. Remember that the light position is stored in eye coordinates, and this is one of the few times when eye coordinates are critical. In Example 5-7, the light position is defined in **init(),** which stores the light position at (0, 0, 0) in eye coordinates. In other words, the light is shining from the lens of the camera.

Example 5-7 Light Source That Moves with the Viewpoint

```
GLfloat light_position[] = { 0.0, 0.0, 0.0, 1.0};
glViewport(0, 0, (GLint) w, (GLint) h);
glMatrixMode(GL_PROJECTION);
glLoadidentity();
gluPerspective(40.0, (GLfloat) w/(GLfloat) h, 1.0, 100.0);
glMatrixMode(GL_MODELVIEW);
glLoadidentity();
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

If the viewpoint is now moved, the light will move along with it, maintaining (0, 0, 0) distance, relative to the eye. In the continuation of Example 5-7, which follows next, the global variables *(ex, ey, ez)* control the position of the viewpoint, and *(upx, upy, upz)* control the value of the up-vector. The **display()** routine that's called from the event loop to redraw the scene might be as follows:

```
static GLdouble ex, ey, ez, upx, upy, upz;
void display(void)
{
    glClear(GL_COLOR_BUFFER_MASK | GL_DEPTH_BUFFER_MASK);
     glPushMatrix();
         gluLookAt(ex, ey, ez, 0.0, 0.0, 0.0, upx, upy, upz);
         glutSolidTorus(0.275,  0.85, 8, 15);
     glPopMatrix();
     glflush();
}
```

When the lit torus is redrawn, both the light position and the viewpoint are moved to the same location. As the values passed to **gluLookAt()** change and the eye moves, the object will never appear dark, because it is always being illuminated from the eye position. Even though you haven't respecified the light position, the light moves because the eye coordinate system has changed.

This method of moving the light can be very useful for simulating the illumination from a miner's hat. Another example would be carrying a candle or lantern. The light position specified by the call to glLightfv(GL_LIGHTi**,** GL_POSITION, position) would be the *x*-, *y*-, and z-distances from the eye position to the illumination source. Then, as the eye position moves, the light will remain the same relative distance away.

Defining Material Properties

You've seen how to create light sources with certain characteristics and how to define the desired lighting model. This section describes how to define the material properties of the objects in the scene: the ambient, diffuse, and specular colors, the shininess, and the color of any emitted light. Most of the material properties are conceptually similar to ones you've already used to create light sources. The mechanism for setting them is similar, except that the command used is called **glMaterial*().**

void glMaterial{if} (GLenum face, *GLenum pname, TYPE param);*
void glMaterial{if}v(GLenum face, *GLenum pname, TYPE *param);*

Specifies a current material property for use in lighting calculations. *face* can be GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK to indicate to which faces of the object the material should be applied. The particular material property being set is identified by *pname,* and the desired values for that property are given by *param,* which is either a pointer to a group of values (if the vector version is used) or the actual value (if the nonvector version is used). The nonvector version works only for setting GL_SHININESS. The possible values for *pname* are shown in Table 5-3. Note that GL_AMBIENT_AND_DIFFUSE allows you to set both the ambient and diffuse material colors simultaneously to the same RGBA value.

Parameter Name	Default Value	Meaning
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	ambient color of material
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	diffuse color of material
GL_AMBIENT_AND_DIFFUSE		ambient and diffuse color of material
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	specular color of material
GL_SHININESS	0.0	specular exponent
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	emissive color of material

Table 5-3 Default Values for *pname* Parameter of glMaterial*()

As discussed in 'Selecting a Lighting Model," you can choose to have lighting calculations performed differently for the front- and back-facing polygons of objects. If the back faces might indeed be seen, you can supply different material properties for the front and back surfaces by using the *face* parameter of glMaterial*().

To give you an idea of the possible effects you can achieve by manipulating material properties, see the teapot program. This program shows the same object drawn with several different sets of material properties. The same light source and lighting model are used for the entire figure.

Note that most of the material properties set with glMaterial*() are (R, G, B, A) colors.

Diffuse and Ambient Reflection

The GL_DIFFUSE and GL_AMBIENT parameters set with **glMaterial*()** affect the colors of the diffuse and ambient light reflected by an object. Diffuse reflectance plays the most important role in determining what you perceive the color of an object to be. Your perception is affected by the color of the incident diffuse light and the angle of the incident light relative to the normal direction. (It's most intense where the incident light falls perpendicular to the surface.) The position of the viewpoint doesn't affect diffuse reflectance at all.

Ambient reflectance affects the overall color of the object. Because diffuse reflectance is brightest where an object is directly illuminated, ambient reflectance is most noticeable where an object receives no direct illumination. An object's total ambient reflectance is affected by the global ambient light and ambient light from individual light sources. Like diffuse reflectance, ambient reflectance isn't affected by the position of the viewpoint.

For real-world objects, diffuse and ambient reflectance are normally the same color. For this reason, OpenGL provides you with a convenient way of assigning the same value to both simultaneously with **glmaterial*():**

```
GLfloat mat_amb_diff [] = { 0.1, 0.5, 0.8, 1.0 };

glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, mat_amb_diff);
```

In this example, the RGBA color (0.1, 0.5, 0.8, 1.0)-a deep blue color-represents the current ambient and diffuse reflectance for both the front- and back-facing polygons.

Specular Reflection

Specular reflection from an object produces highlights. Unlike ambient and diffuse reflection, the amount of specular reflection seen by a viewer does depend on the location of the viewpoint-it is brightest along the direct angle of reflection. To see this, imagine looking at a metallic ball outdoors in the sunlight. As you move your head, the highlight created by the sunlight moves with you to some extent. However, if you move your head too much, you lose the highlight entirely.

OpenGL allows you to set the effect that the material has on reflected light (with GL_SPECULAR) and control the size and brightness of the highlight (with GL_SHININESS). You can assign a number in the range [0.0, 128.0] to GL_SHININESS: the higher the value, the smaller and brighter (more focused) the highlight

Emission

By specifying an RGBA color for GL_EMISSION, you can make an object appear to be giving off light of that color. Since most real-world objects (except lights) don't emit light, you'll probably use this feature mostly to simulate lamps and other light sources in a scene.

Global Ambient Light

In addition to eight lights, there can be other ambient light that's not from any particular source. To specify the intensity of that light, use GL_LIGHT_MODEL_AMBIENT

```
GLfloat lmodel_ambient[] = {0.2, 0.2, 0.2, 1.0};  //the default values
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
```

Local or Infinite Viewpoint

The location of the viewpoint affects the calculations for highlights produced by specular reflectance. More specifically, the intensity of the highlight at a particular vertex depends on the normal at that vertex, the direction from the vertex to the light source, and the direction from the vertex to the viewpoint. With an infinite viewpoint, the direction between it and any vertex in the scene remains constant. A local viewpoint tends to yield more realistic results, but since the direction has to be calculated for each vertex, overall performance is decreased with a local viewpoint. By default, an infinite viewpoint is assumed. To change to a local viewpoint:

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER,  GL_TRUE);
```