An Introduction to 3D Computer Graphics, Stereoscopic Image, and Animation in OpenGL and C/C++

Fore June

Chapter 7 Lighting and Texture

7.1 Importance of Lighting and Local Illumination

Lighting and shading are important features in graphics for making a scene appear more realistic and more understandable. Lighting provides crucial visual cues about the curvature and orientation of surfaces. It also helps viewers perceive a graphics scene having three-dimensionality. Figure 7-1 shows a lit and an unlit sphere drawn in the same way. We can see that the lit sphere gives us a 3D perception but the unlit sphere simply looks like a 2D disk.

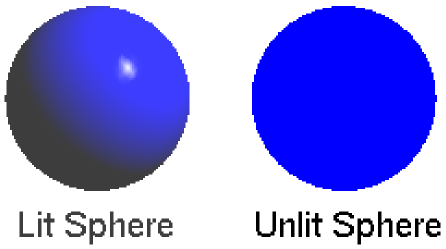

Figure 7-1 A Lit Sphere and an Unlit Sphere

Shading refers to the process of making colors and brightness vary smoothly across a surface. Gouraud shading and Phong shading are the two most popular shading methods that make use of interpolation to create a smooth appearance to surfaces. We shall discuss a local model of illumination and shading called **Phong lighting model**, which is the simplest and by far the most popular lighting and shading model. This model gives good shading and illumination and we can implement it efficiently in hardware or software. This is also the model that OpenGL has used. Here, we mainly discuss the lighting features and do not address the shading details.

The Phong lighting model (also called Phong illumination or Phong reflection model) is considered as a local model of lighting because it only considers the effects of a light source shining directly on a surface and then being reflected directly to the viewpoint; second bounces are ignored. In general, a local lighting model only considers the light property and direction, the viewer's position, and the object material properties. It considers only the first bounce of the light ray but ignores any secondary reflections, which are light rays that are reflected for more than once by surfaces before reaching the viewpoint. Nor does a basic local model consider shadows created by light. Actually, the Phong model is not based on real physics but on empirical graphical experience. The model consists of the following features:

1. All light sources are modeled as point light sources.
2. Light is composed of red (R), green (G), and blue (B) colors.
3. Light reflection intensities can be calculated independently using the principle of superposition for each light source and for each of the 3 color components (R, G, B). Therefore, we describe a source through a three-component intensity or illumination vector

$$\mathbf{I} = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{7.1}$$

Each of the components of $\mathbf{I}$ in (7.1) is the intensity of the independent red, green, and blue components.

4. There are three distinct kinds of light or illumination that contribute to the computation of the final illumination of an object:
   - **Ambient Light**: light that arrives equally from all directions. We use this to model the kind of light that has been scattered so much by its environment that we cannot tell its original source direction. Therefore, ambient light shines uniformly on a surface regardless of its orientation. The position of an ambient light source is meaningless.
   - **Diffuse Light**: light from a point source that will be reflected diffusely. We use this to model the kind of light that is reflected evenly in all directions away from the surface. (Of course, in reality this depends on the surface, not the light itself. As we mentioned earlier, this model is not based on real physics but on graphical experience.)
   - **Specular Light**: light from a point source that will be reflected specularly. We use this to model the kind of light that is reflected in a mirror-like fashion, the way that a light ray reflected from a shinny surface.
5. The model also assigns each surface material properties, which can be one of the four kinds:
   - Materials with **ambient reflection properties** reflect ambient light.
   - Materials with **diffuse reflection properties** reflect diffuse light.
   - Materials with **specular reflection properties** reflect specular light.
   - Materials with **emissive properties** emits light. The emissivity of a surface controls how much light the surface emits in the absence of any incident light. Note that light emitted from a surface does not act as light source that illuminates other surfaces. It only affects the color of the object seen by the observer.

Note that if an object only possesses ambient reflection properties, it only reflects ambient light even if light sources of other types (diffuse and specular) exist in the environment. In general, we assign the four material properties to an object and the final illumination at a point observed by the viewer is the sum of ambient, diffuse, and specular reflections plus the light emitted by the object.

7.2 Phong Reflection Model

Figure 7-2 shows the fundamental vectors of the Phong lighting model. As shown in the figure, a point on the surface is illuminated by a point light source and viewed from some viewpoint (eye position). To calculate the illumination at the point, we need to know the vectors, $\mathbf{n}$, $\mathbf{v}$, and $\mathbf{l}$, where

$$\begin{aligned} \mathbf{n} &= \text{unit normal vector to the surface} \\ \mathbf{v} &= \text{unit vector of viewpoint (eye) direction} \\ \mathbf{l} &= \text{unit vector of point light source direction} \end{aligned} \tag{7.2}$$

These are unit vectors. That is, $|\mathbf{n}| = |\mathbf{v}| = |\mathbf{l}| = 1$. These three vectors along with the attributes of the light source and the properties of the surface material are used by the Phong model to determine the amount (intensity) of light reaching the eye.

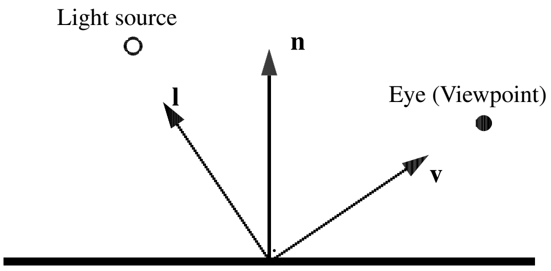

Figure 7-2 Phong Lighting Vectors

7.2.1 Specular Reflection

In the specular reflection model, the amount of light that can be received by an observer depends on her position of observation (viewpoint). If the surface is perfectly smooth, mirror-like reflections occur and the eye can see the reflected light only if it is positioned at a point along perfect-reflection direction. But for non-perfect surface we still can see specular highlight when the eye moves a little bit away from the perfect reflection (mirror) direction. The larger the deviation of the view angle from the mirror direction, the less light we can see. Phong model makes use of this observation to derive a formula to calculate the specular reflection. Figure 7-3 shows the setup for calculating the specular reflection. In addition to the unit vectors $\mathbf{n}$, $\mathbf{v}$, and $\mathbf{l}$ above, we have the following quantities:

$$\begin{aligned}
\theta &= \text{angle of incidence} = \text{perfect reflection angle}\\
\mathbf{r} &= \text{unit vector of direction of perfect reflection, coplanar with } \mathbf{l} \text{ and } \mathbf{n}\\
\phi &= \text{deviation of view angle from perfect reflection direction}\\
|\mathbf{r}| &= |\mathbf{n}| = |\mathbf{v}| = |\mathbf{l}| = 1\\
\mathbf{l}\cdot\mathbf{n} &= \mathbf{r}\cdot\mathbf{n} = \cos\theta\\
I_s^{in} &= \text{incidence specular light intensity}\\
I_s &= \text{reflected specular light intensity}
\end{aligned} \tag{7.3}$$

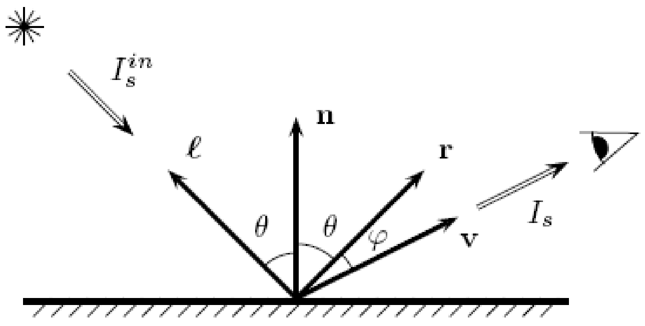

Figure 7-3 Setup for Calculating Specular Reflection

The reflection direction is determined by the incident direction and the normal to the surface. The colplanar condition implies that we can express **r** as a linear combination of **l** and **n**:

$$\mathbf{r} = a\mathbf{l} + b\mathbf{n} \tag{7.4}$$

where a and b are constants which can be determined by taking the dot product of $\mathbf{n}$ and Equation (7.4). This gives

$$\begin{aligned} & \mathbf{n} \cdot \mathbf{r} = a\mathbf{n} \cdot \mathbf{l} + b\mathbf{n} \cdot \mathbf{n} \\ \Rightarrow \quad & \cos\theta = a\cos\theta + b \end{aligned} \tag{7.5}$$

Squaring (7.4), we obtain

$$\begin{aligned} & \mathbf{r} \cdot \mathbf{r} = a^2 + 2ab\mathbf{l} \cdot \mathbf{n} + b^2 \\ \Rightarrow \quad & 1 = a^2 + 2ab\cos\theta + b^2 \end{aligned} \tag{7.6}$$

Solving (7.5) and (7.6), we obtain

$$\begin{aligned} & a = -1 \\ & b = 2\cos\theta = 2\mathbf{l} \cdot \mathbf{n} \end{aligned} \tag{7.7}$$

Therefore, the reflection direction is given by

$$\boxed{\mathbf{r} = 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n} - \mathbf{l}} \tag{7.8}$$

The Phong model calculates the specular reflection using an empirical formula which can be expressed in the following form:

$$\begin{aligned} I_s &= c_s I_s^{in}(\mathbf{r} \cdot \mathbf{v})^f \\ &= c_s I_s^{in}(\cos\phi)^f \end{aligned} \tag{7.9}$$

where c_s is a constant called the *specular reflectivity coefficient* and the exponent f is a value that can be adjusted empirically on an ad hoc basis to achieve desired lighting effect. The exponent f is ≥ 0, and values in the range 50 to 100 are typically used for shinny surfaces. The larger the exponent factor f, the narrower the beam of specularly reflected light becomes. When $f \rightarrow \infty$, (7.9) is reduced to mirror-like perfect reflection. Figure 7-4 shows a general specular reflection.

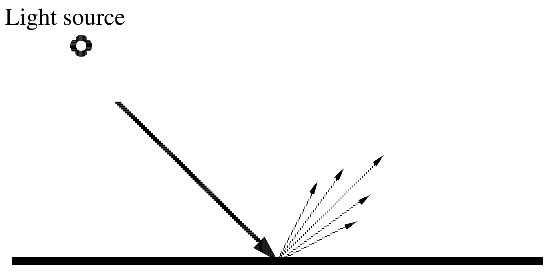

Figure 7-4 Specular Reflection

Note that the light intensity I is composed of any of the three components, red, green, and blue and each component operates independently. We can think of the quantity I_s in Equation (7.9) as an element of the set $\{R_s, G_s, B_s\}$. That is, $I_s \in \{R_s, G_s, B_s\}$.

Therefore, (7.9) actually represents three equations, one for each color component. For example,

$$R_s = c_s^R R_s^{in}(\cos\phi)^f \tag{7.10}$$

where R_s denotes the brightness of the reflected red color component, R_s^{in} is the incident brightness of the red component, and c_s^R is the specular reflectivity coefficient for the red color component.

In general, the reflectivity coefficient c_s depends on the surface and the wavelength of the of the light. We may need to use a different c_s for each of the red, green, and blue components. The material property of an object is defined by the reflectivity coefficients. A perfectly red ball would have $c_s^R = 1, c_s^G = 0$, and $c_s^B = 0$, which yields $R_s = R_s^{in}, G_s = 0, B_s = 0$. It reflects all the incoming red light and absorbs all the green and blue light that strikes it.

Figure 7-5 shows two spheres with specular properties exposed to specular light from the same direction. The two spheres have different exponent factors f. The one with a lot larger f has much more concentrated reflection.

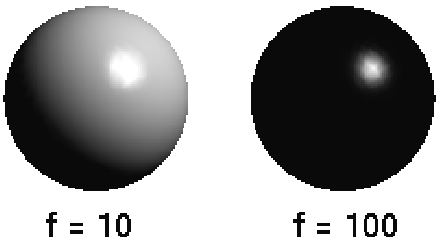

Figure 7-5 Spheres with Same Specular Material Properties but Different Exponents f Lit by the Same Specular Light

7.2.2 Diffuse Reflection

In the model of diffuse reflection, a surface receives illumination from a light source and reflects equally in all direction. The eye position does not matter and has no effect on the amount of light it receives from reflection. However, the amount of reflection depends on the incident angle θ of the light.

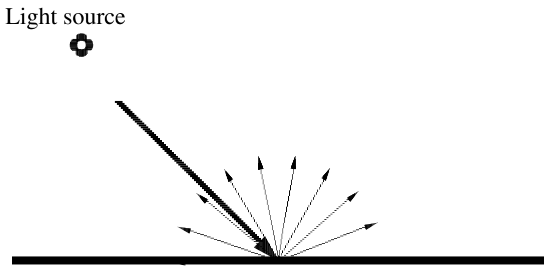

Figure 7-6 Diffuse Reflection

The intensity of reflected light is calculated based on Lamberts law, which says that if a light beam is incident on a small surface at an angle θ to the surface normal, the radiant energy that the small surface receives is equal to the product of the incoming light intensity and cos θ. Therefore, the reflected diffuse light intensity is given by

$$I_d = c_d I_d^{in} \cos\theta \tag{7.11}$$

where cos $\theta = \mathbf{l} \cdot \mathbf{n}$, I_d^{in} is the incident diffuse light intensity and c_d is a constant called the *diffuse reflectivity coefficient*. Maximum reflection occurs when $\theta = 0$, meaning that the incoming light is incident normally on the surface. The illuminated sphere on the left side of Figure 7-7 below is an example of diffuse reflection; the sphere is illuminated by diffuse light only. Again we can assume that $I_d \in \{R_d, G_d, B_d\}$, and $c_d \in \{c_d^R, c_d^G, c_d^B\}$.

7.2.3 Ambient Reflection and Emissivity

Ambient light is the illumination of an object caused by multiple reflections from other surfaces. The Phong model assumes ambient light is uniform in the environment. It is a very rough approximation of global illumination. In the model, the amount of ambient light incident on each object is a constant for all surfaces in the scene and the amount of ambient light reflected by an object is independent of the object's position or orientation. Surface properties are used to determine how much ambient light is reflected. The intensity of the reflected light is given by

$$I_a = c_a I_a^{in} \tag{7.12}$$

where I_a^{in} is the incident ambient light intensity and c_a is a constant called the *ambient reflectivity coefficient*; $I_a \in \{R_a, B_a, G_a\}$ and $c_a \in \{c_a^R, c_a^G, c_a^B\}$.

In addition, a surface can also be given an emissive intensity constant $\mathbf{I}_e$, which is equal to the light emitted by the surface.

The illuminated sphere on the right side of Figure 7-7 is an example of ambient reflection; the sphere is illuminated by ambient light only.

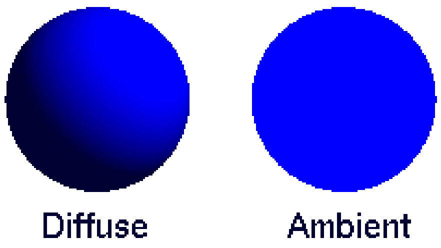

Figure 7-7 Diffuse and Ambient Reflections

7.2.4 Phong Reflection

Putting all these together, we obtain the Phong model which states that the total intensity I of the reflected light of a particular wavelength is the sum of all of the above reflected intensities. That is,

$$\begin{aligned} I &= I_a + I_d + I_s + I_e \\ &= c_a I_a^{in} + c_d I_d^{in}(\mathbf{l} \cdot \mathbf{n}) + c_s I_s^{in}(\mathbf{r} \cdot \mathbf{v})^f + I_e \end{aligned} \tag{7.13}$$

To write a single equation that incorporates all three wavelengths (red, green, blue) at once, we use boldface variables to denote a 3-tuple as we do for denoting a vector and use the symbol $\odot$ to denote component-wise multiplication on 3-tuples. That is,

$$\mathbf{I_s} = \begin{pmatrix} R_s \\ G_s \\ B_s \end{pmatrix} \qquad \mathbf{c_s} = \begin{pmatrix} c_s^R \\ c_s^G \\ c_s^B \end{pmatrix}$$

and

$$\mathbf{c_s} \odot \mathbf{I_s} = \begin{pmatrix} c_s^R \\ c_s^G \\ c_s^B \end{pmatrix} \odot \begin{pmatrix} R_s \\ G_s \\ B_s \end{pmatrix} = \begin{pmatrix} c_s^R R_s \\ c_s^G G_s \\ c_s^B B_s \end{pmatrix} \tag{7.14}$$

and so on. We can rewrite (7.13) as

$$\begin{aligned} \mathbf{I} &= \mathbf{I}_a + \mathbf{I}_d + \mathbf{I}_s + \mathbf{I}_e \\ &= \mathbf{c}_a \odot \mathbf{I}_a^{in} + \mathbf{c}_d \odot \mathbf{I}_d^{in}(\mathbf{l} \cdot \mathbf{n}) + \mathbf{c}_s \odot \mathbf{I}_s^{in}(\mathbf{r} \cdot \mathbf{v})^f + \mathbf{I}_e \end{aligned} \tag{7.15}$$

Equation (7.15) is for one light source. If there are multiple light sources in the scene, we simply sum the contributions from all the light sources. Suppose there are L light sources and the incident light direction at a point on a surface of light number i is l_i and the perfect reflection direction is $\mathbf{r}_i$. The i-th light also has an intensity $\mathbf{I}^{in,i}$ that represents the intensity of the light reaching the observed point on the surface. This intensity may be moderated by the surface from the light and other factors such as spotlight effects. Moreover, if $\mathbf{n} \cdot \mathbf{l}_i \leq 0$, the light is not shining from above the surface; in this case, we set $\mathbf{I}^{in,i}$ to 0. The overall illumination at the point is given by

$$\mathbf{I} = \mathbf{c}_a \odot \mathbf{I}_a^{in} + \mathbf{c}_d \odot \sum_{i=1}^{L} \mathbf{I}_d^{in,i}(\mathbf{l}_i \cdot \mathbf{n}) + \mathbf{c}_s \odot \sum_{i=1}^{L} \mathbf{I}_s^{in,i}(\mathbf{r}_i \cdot \mathbf{v})^f + \mathbf{I}_e \tag{7.16}$$

Figure 7-8 shows a sphere rendered with Phong illumination. Two light sources have been used.

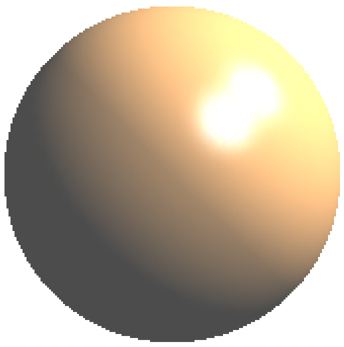

Figure 7-8 Phong Illumination

7.3 OpenGL Lighting

OpenGL implements the full Phong lighting model and supports all material properties, including the ambient, diffuse, and specular reflectivity coefficients and emissivity. We may assign ambient, diffuse and specular intensities and some special effects such as spotlighting and distance attenuation independently to each light source.

The reflectivity coefficient of a surface determines the reflectance of the surface; its 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.

By default, Phong lighting is disabled in OpenGL and a vertex color is set by a **glColor***() command. However, after we have enabled lighting, the **glColor***() commands will have no effect on the scene. We can use the following command to enable Phong lighting:

```
glEnable ( GL_LIGHTING );
```

In general, we can create up to eight point light sources in a graphics scene of OpenGL using commands of **glLight***(). The light sources are named GL_LIGHT0, GL_LIGHT1, GL_LIGHT2, and so on. Each light source must be enabled explicitly using **glEnable**(). For example, we can "turn on" or enable light sources 0 and 1 (GL_LIGHT0 and GL_LIGHT1) by the commands,

```
glEnable( GL_LIGHT0 );
glEnable( GL_LIGHT1 );
```

To turn light source 1 off but leave light 0 on, we can disable only light source 1 using the command

```
glDisable( GL_LIGHT1 );
```

OpenGL provides the options of rendering one side or both sides of a polygon. A polygon consists of a front face and a back face. In general, the face that faces the exterior of an object is the front face and the one that faces the interior is the back face. By default, OpenGL does not light the back faces. To tell OpenGL to light the back faces too, we can use the command

glLightModel*i* (GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);

where *i* can be $0, 1, ..., , 7$, denoting the *i*-th light. The functions **glLightModel***() set the lighting model parameters. The first parameter specifies a single-valued lighting model parameter, which can be

```
GL_LIGHT_MODEL_LOCAL_VIEWER
GL_LIGHT_MODEL_COLOR_CONTROL
GL_LIGHT_MODEL_TWO_SIDE
```

The second parameter passes the appropriate value to the function for the specified lighting model.

In summary, to set up lighting properly in OpenGL, we need to perform the following tasks:

1. Set light properties.
2. Enable or disable lighting.
3. Set surface material properties.
4. Provide correct surface normals.
5. Set light model properties.

7.3.1 Light Properties

Light sources have a number of properties, such as color, position, and direction. We can use the functions **glLight***() to specify all properties of lights. The functions take three arguments to identify the light source, the property, and the desired value for that property:

> void **glLight**{if} (GLenum *light_source*, GLenum *pname*, TYPE *param*);
> void **glLight**{if**v**} (GLenum *light_source*, GLenum *pname*, TYPE **param*);
>
> The functions create the light specified by *light_source*, which can be GL_LIGHT0, GL_LIGHT1, ... , or GL_LIGHT7. The characteristic of the light being set is specified by *pname*, which is a named parameter as shown in Table 7-1. The parameter *param* indicates the values to which the *pname* characteristic is assigned; it can be a pointer to a group of values if the vector version is used, or the value itself if the nonvector version is used. We can use the nonvector version to set only single-valued light characteristics.
>
> Example:

```
float light[] = { 0.5, 0.5, 0.8 };
float light_position[] = { 1.0, 1.0, 1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_DIFFUSE, light );
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

Table 7-1 Default Values for *pname* Parameter of **glLight***()

pname	**Default Value**	**Meaning**
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	ambient RGBA intensity of light
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	diffuse RGBA intensity of light
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	specular RGBA intensity of light
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

The default values listed for GL_DIFFUSE and GL_SPECULAR in Table 7-1 apply only to GL_LIGHT0. For other lights, the default values are $(0.0, 0.0, 0.0, 1.0)$ for both GL_DIFFUSE and GL_SPECULAR.

7.3.2 Types of Light

OpenGL supports two types of lights:

1. point light source at a fixed location, and

2. directional light (location of source at infinite distance).

To specify which type of light we are going to use, we assign a nonzero or a zero value to the fourth component (w) of the light position in homogeneous coordinates. If $w = 0$, we specify a directional light (an infinite-distance light source). If $w \neq 0$, we specify a finite point source located at $(x/w, y/w, z/w)$.

The following code section is an example indicating that light source 0 is a finite point source located at $(2, 1, 0.5)$ and light source 1 is a directional light which shines along the direction $(-4, -2, -1)$ (i.e. from the point $(4, 2, 1)$ to the origin $(0, 0, 0)$).

```
float x = 4, y = 2, z = 1, w = 2;
float light_position0[4] = { x, y, z, w };
glLightfv(GL_LIGHT0, GL_POSITION, light_position0);
w = 0;
float light_position1[4] = { x, y, z, w };
glLightfv(GL_LIGHT1, GL_POSITION, light_position1);
```

7.3.3 Surface Material Properties

We use OpenGL functions **glMaterial***() to set the surface material properties. We can use the functions to specify the ambient, diffuse and specular reflectivity coefficients and the emissive intensity. Note again that an ambient surface interacts with ambient light, a diffuse surface interacts with diffuse light, and so on. For example, to set the diffuse reflectivity coefficients, we may use a code similar to the following:

```
float r = 0.8, g = 0.6, b = 0.5, a = 1.0;
float color[4] = {r, g, b, a};
glMaterialfv( GL_FRONT, GL_DIFFUSE, color );
```

In this example, the function **glMaterialfv**() has three parameters. The first parameter GL_FRONT indicates that we are handling the front faces of polygons. If we want to set the back faces or both front and back faces, we can use the parameters GL_BACK and GL_FRONT_AND_BACK respectively. The second parameter GL_DIFFUSE indicates that we are setting diffuse reflectivity coefficients. Other valid parameters for this argument include GL_AMBIENT, GL_SPECULAR, GL_EMISSION, and GL_AMBIENT_AND_DIFFUSE. The third parameter, $color$, is an array containing the reflectivity coefficients for the red, green, blue and alpha components.

The color components specified for materials have different meaning from that of lights. 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 their values are all the same but less than 1, the light is still white but appears gray. If $R = G = 0$ and $B = 1$, the light is blue.

For materials, the numbers are reflectivity coefficients, corresponding to the reflected proportions of those colors. Therefore, if $R = 1$, and $G = B = 0$ for a material, that material reflects only 100% of all incoming red light, but absorbs all green, and blue light. For example, if an OpenGL light has components $(1, 0.9, 0.8)$, and a material has components $(0.6, 0.5, 0.4)$, then the light that arrives at the eye is $(1 \times 0.6, 0.9 \times 0.5, 0.8 \times 0.4) = (0.6, 0.45, 0.36)$.

If we have two lights L_1 and L_2 with components,

$$L_1 = \begin{pmatrix} R_1 \\ G_1 \\ B_1 \end{pmatrix} \qquad L_2 = \begin{pmatrix} R_2 \\ G_2 \\ B_2 \end{pmatrix}$$

shining at the same point. OpenGL adds the components, giving

$$L = L_1 + L_2 = \begin{pmatrix} R_1 + R_2 \\ G_1 + G_2 \\ B_1 + B_2 \end{pmatrix} \tag{7.17}$$

If any of the sums is larger than 1, OpenGL clamps it to 1.

7.3.4 Surface Normals

We need to provide correct normals for OpenGL to produce 'correct' lighting. Equations (7.13) and (7.8) show that the diffuse and specular reflections depend on the normal at the point where light is incident on the surface. In general, we associate a normal to each vertex of the polygons that make up the surface similar to the following code where (xn, yn, zn) specifies the normal at the point (x, y, z):

```
glBegin ( ... );
  glNormal3f ( xn, yn, zn );
  glVertex3f ( x, y, z );
  ---
glEnd()
```

The normals we provide need to have a unit length. We can use "**glEnable** (GL_NORMALIZE);" to have OpenGL normalize all the normals. When we use the "glut" utilities to create a graphics object such as **glutSolidSphere**(), the normals at various points of the object have been calculated by the called functions.

7.3.5 Attenuation and Spotlighting

We can create some special lighting effects using the distance **attenuation** and **spotlighting** features supported by OpenGL. We know that for real-world point-source lights, the intensity of light decreases as distance from the light increases. Since a directional light is infinitely far away, it does not make sense to attenuate its intensity over distance, so attenuation is disabled for a directional light. However, we can attenuate a positional light. OpenGL provides functions to attenuate a light source by multiplying the contribution of that source by an attenuation factor ρ:

$$\rho = \frac{1}{k_c + k_l d + k_q d^2} \tag{7.18}$$

where k_c, k_l, and k_q are the *constant attenuation factor*, the *linear attenuation factor*, and the *quadratic attenuation factor* respectively; d is the distance from the light source to the incident point of a surface. An incident intensity value is multiplied by the distance attenuation factor before being used in the Phong lighting calculations. By default, $k_c = 1.0, k_l = k_q = 0.0$. We can assign different values to these parameters using the function **glLightf**(). For example, the code

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5);
```

will set $k_c = 2.0, k_l = 1.0, k_q = 0.5$ for light 0; if $d = 2$, the attenuation factor will be

$$\rho = \frac{1}{2.0 + 1.0 \times 2 + 0.5 \times 2^2} = \frac{1}{2.0 + 2.0 + 2.0} = 0.17$$

We can make use of the spotlight effect to make a positional light act as a narrow beam of light like that shown in Figure 7-9. A spot light source usually has the following characteristics:

1. It is a point light source that emits light only within a specified volume of a cone.
2. The cone is parameterized by direction and cutoff angle (see Figure 7-9).
3. Light is parameterized by position and intensity.
4. An exponent parameter is used to control the intensity distribution within the cone and how fast the light intensity attenuates from the central axis of the spotlight.
5. Intensity falls off with the square of the distance from the source.

In OpenGL, properties of a spot light are specified by setting light parameters

GL_SPOT_DIRECTION
GL_SPOT_EXPONENT
GL_SPOT_CUTOFF

The following is a sample code of setting these parameters:

```
float direction[3] = {x, y, z};
glLightfv( GL_LIGHT1, GL_SPOT_DIRECTION, direction );
const float theta = 60;
glLightf( GL_LIGHT1, GL_SPOT_CUTOFF, theta );
const float e = 2.0;
glLightf( GL_LIGHT1, GL_SPOT_EXPONENT, e );
```

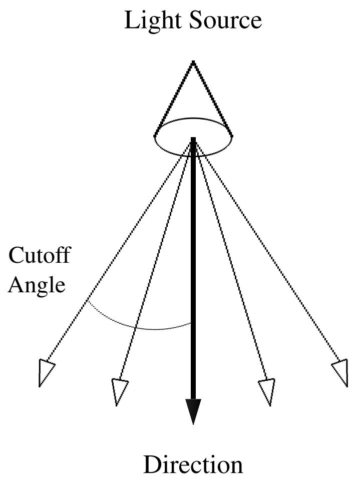

Figure 7-9 Spot Light

Example 7-1

This example presents the code that shows the basic steps of lighting objects in a scene. The following is the code of this example that displays a sphere illuminated by two light sources, as shown earlier in Figure 7-8.

```
void init(void)
{
   //Material properties: reflectivity coefficients
   GLfloat mat_specular[] = { 0.6, .8, 1, 1.0 };
   GLfloat mat_ambient[] = { 0.1, 0.1, 0.1, 1.0 };
   GLfloat mat_diffuse[] = { 0.6, 0.4, 0.2, 1.0 };
   GLfloat mat_shininess0[] = { 20.0};
   GLfloat mat_shininess1[] = { 100.0};
   //Light properties
   GLfloat light_position0[] = {1.0, 1.0, 1.0, 0.0};//directional light
   GLfloat light0[] = { 1, 1, 1 };  //light source 0
   GLfloat light1[] = {1, 1, 1 };   //light source 1
   GLfloat light_position1[] = {1.0, 1.0, 0.0, 0.0};//directional light

   glClearColor (1.0, 1.0, 1.0, 0.0);
   glShadeModel (GL_SMOOTH);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess0);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);

   glLightfv(GL_LIGHT0, GL_POSITION, light_position0);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, light0 );
   glLightfv(GL_LIGHT0, GL_AMBIENT, light0 );
   glLightfv(GL_LIGHT0, GL_SPECULAR, light0 );

   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess1);
   glLightfv(GL_LIGHT1, GL_DIFFUSE, light1 );
   glLightfv(GL_LIGHT1, GL_AMBIENT, light1 );
   glLightfv(GL_LIGHT1, GL_POSITION, light_position1);
   glLightfv(GL_LIGHT1, GL_SPECULAR, light1 );

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_LIGHT1);
   glEnable(GL_DEPTH_TEST);
}

void display(void)
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glutSolidSphere (0.5, 64, 64);

   glFlush ();
}
```

7.4 Texture Mapping

Texture mapping is the mapping of a separately defined graphics image, a picture, or a pattern to a surface. The technique helps us to combine pixels with geometric objects to generate complex images without building large geometric models. For example, we can apply texture mapping to ‘glue’ an image of a brick wall to a polygon and to draw the entire wall as a single polygon. Texture mapping ensures that all the right things happen as the polygon is transformed and rendered. A texture map can hold any kind of information that affects the appearance of a surface. The map is simply a table consisting of data that will be mapped to various pixel locations. We can think of texture mapping as a table lookup process that retrieves the information affecting a particular point on the surface as it is rendered. Figure 7-10 shows an example of texture mapping; a two-dimensional world map is mapped onto the surface of a sphere.

Texture mapping does not require intensive computing. Consequently, people like to use texture maps in real-time rendering such as video games and animations to produce special visual effects. We can apply texture maps at various situations in constructing a graphics scene:

1. A texture map can hold colors to be applied to a surface in *replace* or *decal* mode. In this case, the colors of the texture map simply overwrite the original colors of the object. Lighting should not be turned on as the texture data will overwrite the data generated by any lighting effects.
2. After applying a lighting model to a scene, a texture map can hold color attributes such as color intensities and transparency to change the the surface appearance of the objects in the scene. The texture map attributes are blended with, or modulate, the surface colors generated from the lighting model.
3. A texture map can hold the parameters of a lighting model such as reflectivity coefficients and normals. We can use the texture map values to modify the surface properties that are input to the lighting model.

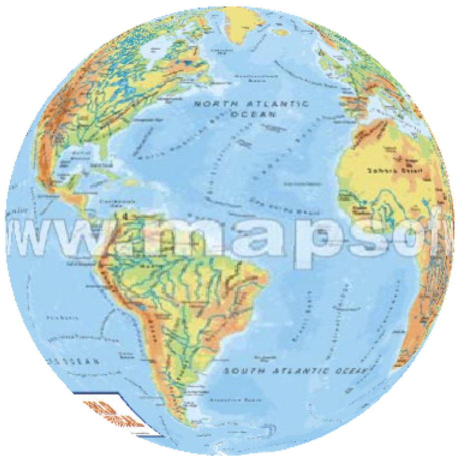

Figure 7-10 A World Map is Mapped onto the Surface of a Sphere

Actually, texture mapping is just one of the general techniques that apply discrete data for surface rendering. We can classify these techniques into three major types:

1. **Texture mapping** uses a pattern (or texture) to determine the color of a fragment. It can be 1, 2, 3, or 4 dimensional. Figure 7-10 above shows an example of this mapping.
2. **Bump mapping** distorts the normal vectors during a shading process to make the surface appear to have small variations in shape like the bumps on a real orange.

Figure 7-11 below shows an orange without bump mapping and another one with bump mapping which makes the orange look a lot more realistic.

3. **Environment mapping** (also called reflection mapping) stores the image of the environment surrounding of the rendered object as a texture image and then maps the image to the object surface. Figure 7-12 shows an example of environment mapping.

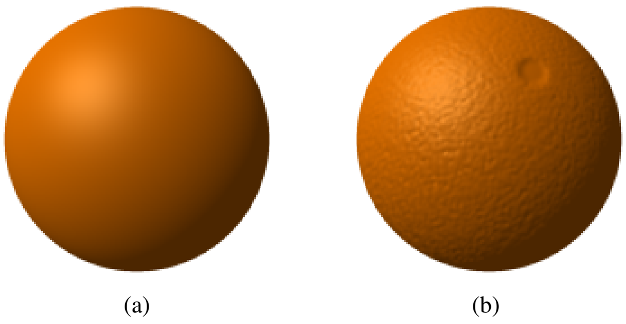

Figure 7-11 Rendering an Orange (a) Without Bump Mapping, and (b) With Bump Mapping

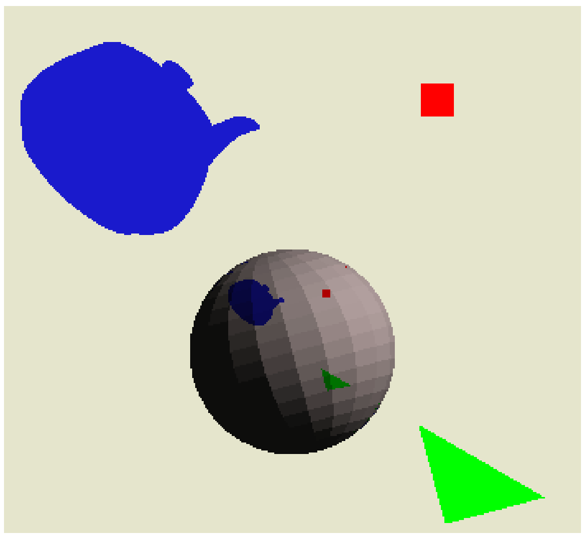

Figure 7-12 Example of Environment Mapping

7.4.1 Two Dimensional Texture Mapping

Just like calling a picture element a pixel, we call a texture element a **texel**. We denote the texture coordinates of a texel as (s,t) and describe the texture pattern by $T(s,t)$. $T(s,t)$ here represents the attributes of the texture at the texel coordinates (s,t). A texture map associates a texel with each point on a geometric object that is itself mapped to screen coordinates for display. If homogeneous coordinates (x,y,z,w) are used to specify a point of an object, then a texture map is described by

$$\begin{aligned} x &= x(s,t) \\ y &= y(s,t) \\ z &= z(s,t) \\ w &= w(s,t) \end{aligned} \tag{7.19}$$

Conversely, given a point (x, y, z, w) of the object, the texel coordinates are obtained by the inverse functions:

$$\begin{aligned} s &= s(x,y,z,w) \\ t &= t(x,y,z,w) \end{aligned} \tag{7.20}$$

For parametric (u, v) surfaces, we need an additional function to map from (u, v) to (x, y, z, w). We also need the mapping from (u, v) to (s, t) and its inverse.

A parametric surface is described by two parameters, u, and v. That is, the position of a point P on the surface is a function of u and v:

$$P(u,v) = \begin{pmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{pmatrix} \tag{7.21}$$

A simple mapping that is commonly used is the **linear mapping**; we map a point on the texture $T(s, t)$ to a point $P(u, v)$ on the parametric surface. This mapping is given by

$$\begin{aligned} u &= as + bt + c \\ v &= ds + et + f \end{aligned} \tag{7.22}$$

where a, b, c, d, e, and f are constants. The mapping is invertible if $ae \neq bd$.

Linear mapping makes it easy to map a texture to a group of parametric surface patches. Suppose the rectangle in st space specified by (s_{min}, t_{min}) and (s_{max}, t_{max}) is mapped to the one in uv space specified by (u_{min}, v_{min}) and (u_{max}, v_{max}) as shown in Figure 7-13.

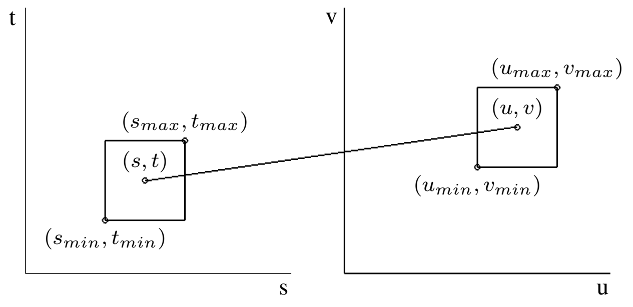

Figure 7-13 Mapping Texture to Surface Patches

The mapping is given by the following linear equations:

$$\begin{aligned} u &= u_{min} + \frac{s - s_{min}}{s_{max} - s_{min}}(u_{max} - u_{min}) \\ v &= v_{min} + \frac{t - t_{min}}{t_{max} - t_{min}}(v_{max} - v_{min}) \end{aligned} \tag{7.23}$$

Example 7-2 Mapping Texture to Surface of Revolution

The surface generated by revolving a profile (curve) around the z-axis can be expressed in the following parametric form:

$$P(u,v) = \begin{pmatrix} r(u)cos(v) \\ r(u)sin(v) \\ z(u) \end{pmatrix} \tag{7.24}$$

where v is the angle of revolution and r is the radius of revolution at a certain height (z value) which depends on the parameter u. We can set u lie in the range $[0,1]$, and v in $[0,2\pi]$. That is,

$$\begin{aligned} 0 \leq u \leq 1 \\ 0 \leq v \leq 2\pi \end{aligned} \tag{7.25}$$

Suppose the texture coordinates are normalized so that $0 \leq s \leq 1$ and $0 \leq t \leq 1$. Then the mapping between (u,v) and (s,t) is given by

$$\begin{aligned} s &= u \\ t &= \frac{v}{2\pi} \end{aligned} \tag{7.26}$$

Example 7-3 Mapping Texture to Sphere

We can define a sphere parametrically by

$$P(\theta,\phi) = \begin{pmatrix} r\text{sin}\,\theta\text{cos}\,\phi \\ r\text{sin}\,\theta\text{sin}\,\phi \\ r\text{cos}\,\theta \end{pmatrix} \tag{7.27}$$

where

$$\begin{aligned} &r = radius \\ &\theta = u = \text{angle from } z\text{-axis } (0 \leq \theta \leq \pi) \\ &\phi = v = \text{angle from } x\text{-axis } (0 \leq \phi \leq 2\pi) \end{aligned} \tag{7.28}$$

The texture mapping is

$$\begin{aligned} s &= \frac{\phi}{2\pi} \\ t &= \frac{\theta}{\pi} \end{aligned} \tag{7.29}$$

Given a point (x,y,z), we can obtain (s,t) from

$$\begin{aligned} t &= \frac{1}{\pi}cos^{-1}\frac{z}{r} \\ s &= \frac{1}{2\pi}cos^{-1}\frac{x}{rsin(t\pi)} \end{aligned} \tag{7.30}$$

The following code section shows this texture mapping. The function takes the coordinates x,y, and z, as well as the radius r as inputs; it then calculates the corresponding texture coordinates s and t and returns their values.

```
const double PI = 3.141592654;
const double TWOPI = 6.283185308;

void
```

```
sphereMap(double x,double y,double z,double r,double *s,double *t)
{
   *t = acos (z / r ) / PI;
   if (y >= 0)
      *s = acos ( x / ( r * sin(PI*(*t)) ) ) / TWOPI;
   else
      *s = ( PI + acos ( x / (r * sin(PI*(*t))) ) ) / TWOPI;
}
```

7.4.2 OpenGL Texture Mapping

To use texture mapping in OpenGL, we need to perform the following steps.

1. **Enable texture mapping**.
   We need to enable texturing before we can apply texture to any object. We usually think of a texture as a 2D image but it can also be 1D. Texturing is enabled or disabled by the functions **glEnable**() or **glDisable**() with symbolic constants GL_TEXTURE_1D or GL_TEXTURE_2D for one or two dimensional texturing respectively:

   glEnable (GL_TEXTURE_1D);
   glEnable (GL_TEXTURE_2D);
   glDisable (GL_TEXTURE_1D);
   glDisable (GL_TEXTURE_2D);

   If both are enabled, 2D texturing is used.

2. **Specify an image or a pattern to be used as texture**.
   Before applying the texture to an object, we need to specify the image or pattern that will be used as texture. In the process, we use the function **glGenTextures**() to generate texture names. We then use **glBindTexture**() to bind a named texture to a texturing target. The function **glBindTexture**() lets us create or use a named texture. When a texture is bound to a target, the previously bound target is automatically released. We can switch between different textures we want using this function. This essentially chooses what texture we are working with.
   The following example code illustrates this step.

   ```
   GLuint handles[2];
   glGenTextures(2, handles);

   glBindTexture(GL_TEXTURE_2D, handles[0]);
   //Initialize texture parameters; load a texture with glTexImage2D

   glBindTexture(GL_TEXTURE_2D, handles[1]);
   //Initialize texture parameters and load another texture
   ```

3. **Indicate how the texture is to be applied to each pixel**.
   The data describing a texture may consist of one, two, three, or four elements per texel, representing anything from a modulation constant to an (R, G, B, A) quadruple. We can choose any of four possible functions for computing the final RGBA value from the fragment color and the texture-image data:

   (a) **decal** mode – the texture color is the final color; the texture is painted on top of the fragment.

   (b) **replace** mode – it is a variant of the decal mode.
   (c) **modulate** mode – use texture to modulate or scale the fragment's color; this technique is useful for combining the effects of lighting with texturing.
   (d) **constant** mode – a constant color is blended with that of the fragment, based on the texture value.

4. **Draw the scene, supplying both texture and geometric coordinates**.
   We need to specify both texture coordinates and geometric coordinates of the object so that we can "glue" the texture image onto the object surface in the way we want. For example, in 2-D texture mapping, the texture coordinates are in the range [0.0, 1.0]. The object coordinates can be anything. We may associate a texture coordinate with a geometric coordinates using the commands **glTexCoord2f**() and **glVertex3f**(). For example, the commands

   glTexCoord2f(0.0, 0.0);
   glVertex3f(-2.0, -2.0, 0.0);

   associate the texture coordinates $(0, 0)$ with the geometric coordinates $(-2.0, -2.0, 0.0)$.

5. **Delete the texture from memory when it is no longer used**.
   After we have finished texturing, we have to delete the texture from memory using the function **glDeleteTextures**() so that the allocated memory will be freed for other uses. For example, the command

   glDeleteTextures(2, *handles*);

   frees two textures specified by the array *handles*.

7.4.3 Texture Programming Examples

In this section, we present some simple programming examples to illustrate the usage of texture in OpenGL.

Example 7-4 Mapping Texture to Quads

In this example, the 2D texture image is generated by the program; it is the same checker board image used in Example 6-1 of Chapter 6. The program applies this texture to two quads independently, which are rendered using perspective projection as shown in Figure 7-14. The following is the code that gives this output display.

```
//checker_tex.cpp
const int width = 256, height = 256;
unsigned char board[width][height][4];
GLuint handle;

void getBoard(void)
{
  int  c;

  for ( int i = 0; i < width; i++ ) {
    for ( int j = 0; j < height; j++ ){//start from lower-left corner
      c = ((((i&16)==0)^((j&16))==0))*255;//0 or 255-->black or white
      board[i][j][0] =  c; //red component
      board[i][j][1] =  c; //green component
      board[i][j][2] =  c; //blue  component
      board[i][j][3] =  255; //alpha
    }
  }
}

void init(void)
{
  glClearColor (0.8, 0.8, 1.0, 1.0);
  glShadeModel(GL_FLAT);
  glEnable(GL_DEPTH_TEST);
  getBoard();

  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

  glGenTextures(1, &handle);
  glBindTexture(GL_TEXTURE_2D, handle);

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height,
               0, GL_RGBA, GL_UNSIGNED_BYTE, board);
}

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  glFrustum (-2.5, 2.5, -2.5, 2.5, 5, 20.0);
  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt ( 0.0, 0.0, 5.5, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

  glEnable(GL_TEXTURE_2D);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
  glBindTexture(GL_TEXTURE_2D, handle);
  glBegin(GL_QUADS);   //draw two quads
    glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -2.0, 0.0);
    glTexCoord2f(1.0, 0.0); glVertex3f( 0.0, -2.0, 0.0);
    glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 0.0, 0.0);
    glTexCoord2f(0.0, 1.0); glVertex3f(-2.0, 0.0, 0.0);

    glTexCoord2f(0.0, 0.0); glVertex3f(0.5, -2.0, 0.0);
    glTexCoord2f(1.0, 0.0); glVertex3f( 2.5, -1.5, 0.0);
    glTexCoord2f(1.0, 1.0); glVertex3f( 2.5, 0.5, 0.0);
    glTexCoord2f(0.0, 1.0); glVertex3f( 0.5, 0.0, 0.0);
  glEnd();
  glFlush();
}
```

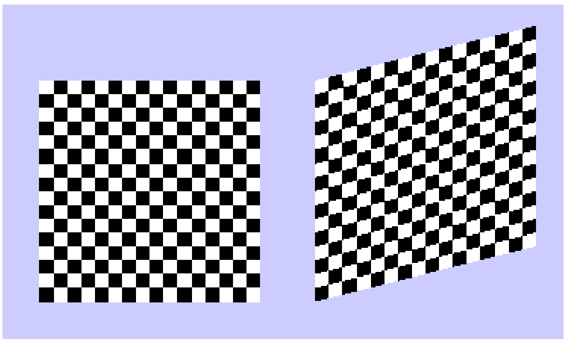

Figure 7-14 Texture Mapping of Example 7-4

In this example, we use the routine **getBoard**() to create the texture image, and perform all the texture-mapping initialization in the routine **init**(). The functions **glGenTextures**() and **glBindTexture**() name and create a texture object for a texture image. The texture map is specified by **glTexImage2D**(), whose input parameters include the size of the image, type of the image, location of the image, and other properties of it. The four calls to **glTexParameter***() specify how the texture is to be wrapped and how the colors are to be filtered if there is no exact match between pixels in the texture and pixels on the screen. In the routine **display**(), texturing is turned on by **glEnable** (GL_TEXTURE_2D), and **glTexEnvf**() sets the drawing mode to GL_DECAL so that the textured polygons are drawn using the colors from the texture map. We then draw two quads with the texture by specifying the texture coordinates along with vertex coordinates. The function **glTexCoord2f**() sets the current texture coordinates that will associate with vertices specified by subsequent **glvertex***() commands until **glTexCoord2f**() is called again.

Example 7-5 Mapping Texture to a Sphere

Figure 7-10 above shows that a world map is mapped onto the surface of a sphere. The following code section is the code that does this mapping, which maps a 2D image onto a sphere. It is a more detailed implementation of Example 7-3 that explains the principles and presents the code for mapping of texture to a sphere. The texture image used in the mapping is given by the file "../data/earth.png". The loading of this image and the initialization of the texture features are all done in the routine **init**(); to load the image, **init**() calls **makeTexImage**() that in turn calls **loadImageRGBA**() of the imagio library that we have discussed in Chapter 6.

The function **createSphere**() creates a sphere composed of quads. Alternatively, one may use triangles to build a sphere (i.e. use GL_TRIANGLE_STRIP for the argument of glBegin() instead of using GL_QUAD_STRIP for it). This function also does the mapping between the texture coordinates and the geometric coordinates as explained in Example 7-3. We also cull the back faces of the polygons using the command **glCullFace** (GL_BACK) so that only the exterior surface of the sphere is rendered.

```
int texImageWidth;
int texImageHeight;
static GLuint handles[6]; //texture names
char maps[][20] = {"../data/earth.png"};

class Point3 {
```

```
public:
  double x;
  double y;
  double z;
  Point3()
  {
    x = y = z = 0.0;
  }

  Point3 ( double x0, double y0, double z0 )
  {
    x = x0; y = y0;    z = z0;
  }
};

//load texture image
GLubyte *makeTexImage( char *loadfile )
{
   int i, j, c, width, height;
   GLubyte *texImage;

   /*
     Only works for .png or .tif images. NULL is returned for errors.
     loadImageRGBA() is from imageio library discussed in Chapter 6.
   */
   texImage = loadImageRGBA( (char *) loadfile, &width, &height);
   texImageWidth = width;
   texImageHeight = height;

   return texImage;
}

void init(void)
{
   glClearColor (1, 1, 1, 0.0);
   glShadeModel(GL_FLAT);

   glEnable(GL_DEPTH_TEST);

   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
   //handles is global
   glGenTextures(1, handles);

   for ( int i = 0; i < 1; ++i ) {
    GLubyte *texImage = makeTexImage( maps[i] );
    if ( !texImage ) {
      printf("\nError reading %s \n", maps[i] );
      continue;
    }
    glBindTexture(GL_TEXTURE_2D, handles[i]); //now we work on handles
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, texImageWidth,
                texImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, texImage);

    delete texImage;    //free memory holding texture image
  }
}
```

```
const double PI = 3.141592654;
const double TWOPI = 6.283185308;
/*
 *    Create a sphere centered at c, with radius r, and precision n
 *    Draw a point for zero radius spheres
 */
void createSphere(Point3 c,double r,int n)
{
   int i,j;
   double phi1, phi2, theta, u, v;
   Point3 p, q;

   if ( r < 0 || n < 0 )
      return;
   if (n < 4 || r <= 0) {
      glBegin(GL_POINTS);
      glVertex3f(c.x,c.y,c.z);
      glEnd();
      return;
   }

   for ( j=0; j < n; j++ ) {
      phi1 = j * TWOPI / n;
      phi2 = (j + 1) * TWOPI / n; //next phi

      glBegin(GL_QUAD_STRIP);
      for ( i=0; i <= n; i++ ) {
         theta = i * PI / n;

        q.x = sin ( theta ) * cos ( phi2 );
        q.y = sin ( theta ) * sin ( phi2 );
        q.z = cos ( theta );
        p.x = c.x + r * q.x;
        p.y = c.y + r * q.y;
        p.z = c.z + r * q.z;

        glNormal3f ( q.x, q.y, q.z );
        u = (double)(j+1) / n; // column
        v = 1 - (double) i / n; // row
        glTexCoord2f( u, v );
        glVertex3f( p.x, p.y, p.z );

        q.x = sin ( theta ) * cos ( phi1 );
        q.y = sin ( theta ) * sin ( phi1 );
        q.z = cos ( theta );
        p.x = c.x + r * q.x;
        p.y = c.y + r * q.y;
        p.z = c.z + r * q.z;

         glNormal3f ( q.x, q.y, q.z );
         u = (double) j / n; // column
         v = 1 - (double) i / n; // row
         glTexCoord2f( j / (double) n, 1 - i / (double) n );
         glVertex3f ( p.x, p.y, p.z );
      }
      glEnd();
   }
}
```

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glEnable(GL_TEXTURE_2D);
   glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
   glEnable( GL_CULL_FACE );
   glCullFace ( GL_BACK );

   glPointSize ( 6 );
   Point3 c;

   createSphere( c, 3.0 , 40 );

   glFlush();
   glDisable(GL_TEXTURE_2D);
}
```

Example 7-6 Mapping Texture to a Cube

In this example, a cube is drawn with texture images. It is similar to Example 7-4, where we "glue" images to quads except that now our images data are loaded from files and the images are "glued" to the six faces of a cube. Again we use **loadImageRGBA**() of **imageio** library to load the images from the files. The following is the code that does the mapping and Figure 7-15 shows the graphics output of the code.

```
int texImageWidth;
int texImageHeight;
static GLuint handles[6];    //texture names

//images for texture maps for 6 faces of cube
char maps[][40] = {"../data/front.png", "../data/back.png",
               "../data/right.png", "../data/left.png",
               "../data/up.png", "../data/down.png" };

//load texture image
GLubyte *makeTexImage( char *loadfile )
{
   int i, j, c, width, height;
   GLubyte *texImage;

   texImage = loadImageRGBA( (char *) loadfile, &width, &height);
   texImageWidth = width;
   texImageHeight = height;

   return texImage;
}

void init(void)
{
   glClearColor (1, 1, 1, 0.0);
   glShadeModel(GL_FLAT);

   glEnable(GL_DEPTH_TEST);

   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
   //handles is global
   glGenTextures(6, handles);
   for ( int i = 0; i < 6; ++i ) {
     GLubyte *texImage = makeTexImage( maps[i] );
     if ( !texImage ) {
       printf("\nError reading %s \n", maps[i] );
       continue;
     }
     glBindTexture(GL_TEXTURE_2D, handles[i]);//now we work on handles
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,GL_NEAREST);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,GL_NEAREST);
     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, texImageWidth,
             texImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,texImage);

     delete texImage;      //free memory holding texture image
   }
   printf("\nPress x, y, z or X, Y, Z to rotate the cube\n");
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glEnable(GL_TEXTURE_2D);
   glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
   float x0 = -1.0, y0 = -1, x1 = 1, y1 = 1, z0 = 1;
   float face[6][4][3] =  {
     {{x0, y0, z0}, {x1, y0, z0}, {x1, y1, z0}, {x0, y1,z0}},  //front
     {{x0, y1, -z0}, {x1, y1, -z0},{x1, y0, -z0},{x0, y0,-z0}},//back
     {{x1, y0, z0}, {x1, y0, -z0}, {x1, y1, -z0}, {x1, y1,z0}},//right
     {{x0, y0, z0}, {x0, y1, z0}, {x0, y1, -z0}, {x0, y0,-z0}},//left
     {{x0, y1, z0}, {x1, y1, z0}, {x1, y1, -z0}, {x0, y1,-z0}},//top
     {{x0, y0, z0}, {x0, y0, -z0}, {x1, y0, -z0}, {x1,y0,z0}}  //bottom
   };
   glEnable( GL_CULL_FACE );
   glCullFace ( GL_BACK );

   glPushMatrix();
   glRotatef( anglex, 1.0, 0.0, 0.0);    //rotate the cube along x-axis
   glRotatef( angley, 0.0, 1.0, 0.0);    //rotate along y-axis
   glRotatef( anglez, 0.0, 0.0, 1.0);    //rotate along z-axis

   for ( int i = 0; i < 6; ++i ) {       //draw cube with texture images
     glBindTexture(GL_TEXTURE_2D, handles[i]);
     glBegin(GL_QUADS);
       glTexCoord2f(0.0, 0.0); glVertex3fv ( face[i][0] );
       glTexCoord2f(1.0, 0.0); glVertex3fv ( face[i][1] );
       glTexCoord2f(1.0, 1.0); glVertex3fv ( face[i][2] );
       glTexCoord2f(0.0, 1.0); glVertex3fv ( face[i][3] );
     glEnd();
   }

   glPopMatrix();
   glFlush();
   glDisable(GL_TEXTURE_2D);
}
```

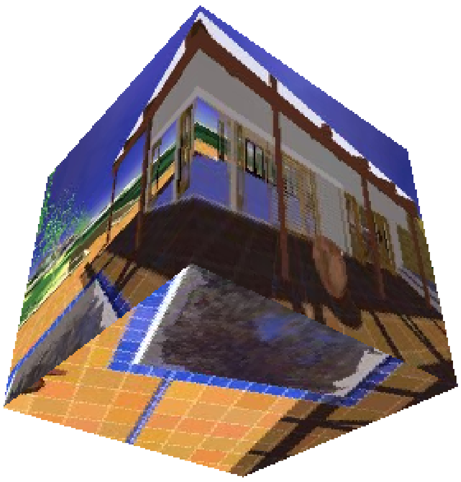

Figure 7-15 Cube Mapping of Example 7-6