Lights and Lighting

Prerequisites

An understanding of color at the level of the discussion of the chapter on color in these notes, and some observation of the way lights work in creating the images of the world around you.

Introduction

There are two ways to think of how we see things. The first is that things have an intrinsic color and we simply see the color that they are. The color is set by defining a color in RGB space and simply instructing the graphics system to draw everything with that color until a new color is chosen. This approach is synthetic and somewhat simpleminded, because it’s clear that things in the real world don’t behave like this; they have a color, but the color they have is strongly influenced by the light that illuminates them, so it’s somehow wrong not to take the light into account in presenting a scene.

The second way to think of how we see things is to realize that seeing involves light that reaches us because of physical interaction between light and the objects we see. Light sources emit energy that reaches the objects and is then re-sent to us in ways that involve the color of the light and the physical properties of the objects. In this chapter we will discuss how this way of thinking about light allows us to model how we see the world, and how computer graphics distills that into fairly simple definitions and relatively straightforward computations to produce images that seem to have a relationship to the lights and materials present in the scene.

A first step in developing graphics that depends on lights and materials is to create a model of the nature of light. In the simplified but workable model of light that we will use, there are three fundamental components of light: the ambient, diffuse, and specular components. We can think of each of these as follows:

- **ambient**: light that is present in the scene because of the overall illumination in the space. This can include light that has bounced off objects in the space and that is thus independent of any particular light.
- **diffuse**: light that comes directly from a particular light source to an object, where it is then sent directly to the viewer. Normal diffuse light radiates back a subset of the wavelengths it receives that depends on the material that makes up an object, with the effect of creating the color of the object.
- **specular**: light that comes directly from a particular light source to an object, where it is then reflected directly to the viewer because the object reflects, not absorbs, the light. This light is generally the color of the light source, not the object that reflects the light.

All three of these components contribute to the light at every wavelength, and the graphics system models this by applying them to the RGB components separately. The sum of these light components is the light that is actually seen from an object.

Just as the light is modeled to create an image, the materials of the objects that make up the scene are modeled in terms of how they contribute to the light. Each object will have a set of properties that defines how it responds to each of the three components of light. The ambient property will define the behavior (essentially, the color) of the object in ambient light, the diffuse property in diffuse light, and the specular property in specular light. As we noted above, realistic lighting tends to assume that objects behave the same in ambient and diffuse light, and that objects simply take on the light color in specular light, but because of the kind of calculations that are done to display a lighted image, it is as easy to treat each of the light and material properties separately.
Thus in order to use lights in a scene, you must define your lights in terms of the three kinds of color they provide, and you must define objects not in simple terms of their color, but in terms of their material properties. This will be different from the process we saw in the earlier module on color but the changes will be something you can handle without difficulty. The OpenGL API has its own way to specify the three components of light and the three components of materials, and we will also discuss this below when we talk about implementing lighting for your work.

Definitions

Ambient, diffuse, and specular light

Ambient light is light that comes from no apparent source but is simply present in a scene. This is the light you would find in portions of a scene that are not in direct light from any of the lights in the scene, such as the light on the underside of an object, for example. Ambient light can come from each individual light source, as well as from an overall ambient light value, and you should plan for each individual light to contribute to the overall brightness of a scene by contributing something to the ambient portion of the light. The amount of diffuse light reflected by an object is given simply by \( A = L_A \cdot C_A \) for a constant \( C_A \) that depends on the material of the object and the ambient light \( L_A \) present in the scene, where the light \( L_A \) and constant \( C_A \) are to be thought of as RGB triples, not simple constants, and the calculation is to yield another RGB value. Ambient light is usually fairly low-level if you want to emphasize the effect of the lights (you might think of this as a night effect) or fairly high-level if you want to see everything in the scene with a fairly uniform light (this would be a brightly-lit effect). If you want to emphasize shapes, use a fairly low ambient light.

Diffuse light comes from particular light sources and is absorbed by the surface of the object, then radiated back in a particular wavelength depending on properties of the object’s material. The OpenGL model for diffuse light is based on the concept that the intensity of light seen on a surface is proportional to the amount of light falling on a unit of the surface area. This is proportional to the cosine of the angle between the surface normal and the light direction as long as that cosine is positive, and zero otherwise, as illustrated in the diagram in Figure 7.1. As the angle of incidence of the light on the surface decreases, the amount of light reflected from the surface becomes dimmer, going to zero when the light is parallel to the surface. Because it is impossible to talk about “negative light,” we replace any negative value of the cosine with zero, which eliminates diffuse light on surfaces facing away from the light. The diffuse lighting calculation computes the amount of diffuse light as

\[
D = L_D \cdot C_D \cdot \cos (N \cdot L)
\]

for the value of the diffuse light \( L_D \) from each light source and the ambient property of the material \( C_D \), which shows why we must have surface normals in order to calculate diffuse light. This computation is done separately for each light source and each object, because it depends on the angle from the object to the light. It is important to note that diffuse light is independent of the point from which we view the material, and thus the location of the eye point does not participate in

![Figure 7.1: diffuse lighting](image)
the diffuse light computation. This is obvious from looking at the world around us, because things
do not change color as the angle between their normals and our eye direction does not change as
we move around. The lighting model supports this behavior, as is seen by noting that for the unit
area in Figure 7.1, the geometric size of that area as seen from any point decreases in exactly the
same way the diffuse light diminishes as the angle $\Theta$ grows from 0 to $\pi/2$, so the brightness of the
surface (the light per unit area) remains the same.

Specular light is a surface phenomenon that provides shiny highlights. These depend on the
smoothness and electromagnetic properties of the surface, so smooth metallic objects (for example)
reflect light well. The energy in specular light is not absorbed by the object and re-radiated, but is
reflected with the angle of incidence equal to the angle of reflection, as illustrated in the left-hand
diagram of Figure 7.2. Such light may have a small amount of "spread" as it leaves the object,
depending on the shininess of the object, so the standard model for specular light allows you to
define the shininess of an object to control that spread. Shininess is controlled by a parameter

\[
\text{Figure 7.2: specular lighting}
\]

which gives smaller, brighter highlights as it increases, as shown in the three successive figures of
Figure 7.3.

\[
\text{Figure 7.3: specular highlights with shininess coefficients 20, 50, and 80}
\]

(left, center, and right), respectively

The specular light on an object in the image is given by

\[
S = L_S \cdot C_S \cdot \cos^N (E \cdot R)
\]

for a light’s specularity $L_S$ and the object’s specular coefficient $C_S$. In addition, the computation
involves the value of the shininess coefficient $N$ that depends on the light and on the material.
Note the visual effect of increasing the shininess coefficient: the highlight gets smaller and more
focused — that is, the sphere looks shinier and more polished. The specular light computation is
done separately for each light and each object, because it depends on the angle from the object to
the light (as well as the angle to the eye point). This calculation depends fundamentally on both the
direction from the object to the light and the direction of the object to the eye, so you should expect
to see specular light move as objects, lights, or your eye point moves.

Because both diffuse and specular lighting need to have normals to the surface at each vertex, we
need to remind ourselves how we get normals to a surface. One way to do this is analytically; we
know that the normal to a sphere at any given point is in the direction from the center of the sphere
to the point, so we need only know the center and the point to calculate the normal. In other cases,
such as surfaces that are the graph of a function, it is possible to calculate the directional derivatives
for the function at a point and take the cross product of these derivatives, because the derivatives
define the tangent plane to the surface at the point. But sometimes we must calculate the normal
from the coordinates of the polygon, and this calculation was described in the discussion of
mathematical fundamentals by taking the cross product of two adjacent edges of the polygon in the
direction the edges are oriented.

So with the mechanics of computing these three light values in hand, we consider the constants that
appeared in the calculations above. The ambient constant is the product of the ambient light
component and the ambient material component, each calculated for the red, green, and blue parts
respectively. Similarly the diffuse and specular constants are the products of their respective light
and material components. Thus a white light and any color of material will produce the color of the
material; a red light and a red material will produce a red color; but a red light and a blue material
will produce a black color, because there is no blue light to go with the blue material and there is no
red material to go with the red light. The final light at any point is the sum of these three parts: the
ambient, diffuse, and specular values, each computed for all three RGB components. If any
component has a final value larger than one, it is clamped to have value 1.

When you have multiple lights, they are treated additively — the ambient light in the scene is the
sum of any overall ambient light for the entire scene plus the ambient lights of the individual lights,
the diffuse light in the scene is the sum of the diffuse lights of the individual lights, and the
specular light in the scene is the sum of the diffuse lights of the individual lights. As above, if
these sums exceed one in any one component, the value is clamped to unity.

As we saw above, you need to calculate normals to the surface in order to compute diffuse and
specular light. This is often done by defining normal vectors to the surface in the specifications of
the geometry of a scene to allow the lighting computation to be carried out. Processes for
computing normals were described in the early chapter on mathematical fundamentals. These can
involve analysis of the nature of the object, so you can sometimes compute exact normals (for
example, if you are displaying a sphere, the normal at any point has the same direction as the
radius vector). If an analytic calculation is not available, normals to a polygonal face of an object
can be computed by calculating cross products of the edges of the polygon. However, it is not
enough merely to specify a normal; you need to have unit normals, normal vectors that are exactly
one unit long (usually called normalized vectors). It can be awkward to scale the normals yourself,
and doing this when you define your geometry may not even be enough because scaling or other
computations can change the length of the normals. In many cases, your graphics API may
provide a way to define that all normals are to be normalized before they are used.

In the next chapter we will discuss shading models, but here we need to note that all our lighting
computations assume that we are calculating the light at a single vertex on a model. If we choose
to do this calculation at only one point on each polygon, we can only get a single color for the
polygon, which leads to the kind of lighting called flat shading. If we wanted to do smooth
shading, which can give a much more realistic kind of image, we would need to determine a
separate normal for each vertex so that the lighting computation could give us a color for each
vertex. If the vertex is part of several polygons and we want to calculate a normal for the vertex
that we can use for all the polygons, we can calculate a separate normal based on each of the
polygons and then average them to get the normal for the vertex. The individual colors for the vertices are then used to calculate colors for all the points in the polygon, as is discussed in more detail in the next chapter.

Note that none of our light computation handles shadows, however, because shadows depend on the light that reaches the surface, which is a very different question from the way light is reflected from the surface. Shadows are difficult and are handled in OpenGL with very specialized programming which we will not cover in these notes.

Use of materials

As we said earlier, lighting involves two parts: both the specification of the lighting properties of the objects in the scene, and the specification of the lights in a scene. If you want to use lighting in creating a scene, you must specify both of these. Here we discuss material specifications, and we follow this by discussing light properties, but implementing lighting involves putting these all together as is discussed in the example at the end of this chapter.

As we saw above, each object participates in determining the reflected light that makes up its color when it is displayed. In the discussion of the three components of light, we saw four constants $C_A$, $C_D$, $C_S$, and $N$ that are part of the computations of the light. The first three of these constants have separate RGB components and together the four constants identify the way a material interacts with light, so they are often called the set of definitions of the material. They need to be defined for each object in your scene in order to allow the lighting calculations to be carried out. Your graphics API will allow you to see these as part of your modeling work; they should be considered as part of the appearance information you would include in a shape node in your scene graph.

All the discussion of lighting above assumed that an object is reflective, but an object can also be emissive — that is, send out light of its own. Such a light simply adds to the light of the object but does not add extra light to the scene, allowing you to define a bright spot to present something like an actual light in the scene. This is managed by defining a material to have an emissive light property, and the final lighting calculations for this material adds the components of the light emission to the other lighting components when the object’s color is computed.

Light properties

Lights are critical components of your modeling work in defining an image, as we saw in the discussion of lights in the scene graph in the modeling chapter. Along with the location of each light, which is directly supported by the scene graph, you will want to define other aspects of the light, and these are discussed in this section.

Your graphics API allows you to define a number of properties for a light. Typically, these can include its position or its direction, its color, how it is attenuated (diminished) over distance, and whether it is an omnidirectional light or a spotlight. We will cover these properties lightly here but will not go into depth on them all, but the properties of position and color are critical. The other properties are primarily useful if you are trying to achieve a particular kind of effect in your scene. The position and color properties are illustrated in the example at the end of this chapter.

Positional lights

When we want a light that works as if it were located within your scene, you will want your light to have an actual position in the scene. To define a light that has position, you will set the position as a four-tuple of values whose fourth component is non-zero (typically, you will set this to be
1.0). The first three values are then the position of the light and all lighting calculations are done with the light direction from an object set to the vector from the light position to the object.

**Spotlights**

Unless you specify otherwise, a positional light will shine in all directions. If you want a light that shines only in a specific direction, you can define the light to be a spotlight that has not only a position, but also other properties such as a direction, a cutoff, and a dropoff exponent, as you will see from the basic model for a spotlight shown in Figure 7.4. The direction is simply a 3D vector that is taken to be parallel to the light direction, the cutoff is assumed to be a value between 0.0 and 90.0 that represents half the spread of the spotlight and determines whether the light is focused tightly or spread broadly (a smaller cutoff represents a more focused light), and the dropoff exponent controls how much the intensity drops off between the centerline of the spotlight and the intensity at the edge.

![Figure 7.4: spotlight direction and cutoff](image)

**Attenuation**

The physics of light tells us that the energy from a light source on a unit surface diminishes as the square of the distance from the light source from the surface. This diminishing is called attenuation, and computer graphics can model that behavior in a number of ways. An accurate model would deal with the way energy diffuses as light spreads out from a source which would lead to a light that diminishes as the square of the distance from the light, and the graphics system would diminish the intensity of the light accordingly. However, the human perceptual system is more nearly logarithmic than linear in the way we see light, so we do not recognize this kind of diminishing light as realistic, and we probably would need to use an attenuation that drops off more slowly. Your graphics API will probably give you some options in modeling attenuation.

**Directional lights**

Up to now, we have talked about lights as being in the scene at a specific position. When such lights are used, the lighting model takes the light direction at any point as the direction from the light position to that point. However, if we were looking for an effect like sunlight, we want light that comes from the same direction at all points in the scene. In effect, we want to have a light at infinity. If your graphics API supports directional lights, there will be a way to specify that the light is directional instead of positional and that defines the direction from which the light will be received.
Positioning and moving lights

Positional lights can be critical components of a scene, because they determine how shapes and contours can be seen. As we noted in the chapter on modeling, lights are simply another part of the model of your scene and affected by all the transformations present in the modelview matrix when the light position is defined. A summary of the concepts from the scene graph will help remind us of the issues here.

- If the light is to be at a fixed place in the scene, then it is at the top level of the scene graph and you can define its position immediately after you set the eye point. This will create a position for the light that is independent of the eye position or of any other modeling in the scene.
- If the light is to be at a fixed place relative to the eye point, then you need to define the light position and other properties before you define the eye position. The light position and properties are then modified by the transformations that set the eye point, but not by any subsequent modeling transformations in the scene.
- If the light is to be at a fixed place relative to an object in the scene, then you define the light position as a branch of the group node in the scene graph that defines the object. Anything that affects the object will then be done above that group node, and will affect the light in the same way as it does the rest of the object.
- If the light is to move around in the scene on its own, then the light is a content node of the scene graph and the various properties of the light are defined as that node is set.

The summary of this modeling is that a positional light is treated simply as another part of the modeling process and is managed in the same way as any other object would be.

Lights and materials in OpenGL

Several times above we suggested that a graphics API would have facilities to support several of the lighting issues we discussed. Here we will outline the OpenGL support for lighting and materials so you can use these capabilities in your work. In some of these we will use the form of the function that takes separate R, G, and B parameters (or separate X, Y, and Z coordinates), such as 

```glLightf(light, name, set_of_values)```

while in others we will use the vector form that takes 3-dimensional vectors for colors and points, but in some cases we will use the vector form such as 

```glLightfv(light, name, vector_values)```

and you may use whichever form fits your particular design and code best.

Specifying and defining lights

When you begin to plan your scene and are designing your lighting, you may need to define your light model with the `glLightModel(...)` function. This will allow you to define some fundamental properties of your lighting. Perhaps the most important use of this function is defining if your scene will use one-sided or two-sided lighting, which is chosen with the function

```glLightModel[f|i](GL_LIGHT_MODEL_TWO_SIDE, value)```

where `[f|i]` means that you use either the letter `f` or the letter `i` to indicate whether the parameter value is real or integer. If the (real or integer) value of the numeric parameter is 0, one-sided lighting is used and only the front side of your material is lighted; if the value is non-zero, both front and back sides of your material are lighted. Other uses of the function include setting a global ambient light, discussed below, and choosing whether specular calculations are done by assuming the view direction is parallel to the Z-axis or the view direction is towards the eye point. This is determined by the function

```glLightModel[f|i](GL_LIGHT_MODEL_LOCAL_VIEWER, value)```

with a value of 0 meaning that the view direction is parallel to the Z-axis and non-zero that it is toward the origin. The default value is 0.
OpenGL allows you to define up to eight lights for any scene. These lights have the symbolic names \texttt{GL\_LIGHT0} ... \texttt{GL\_LIGHT7}, and you create them by defining their properties with the \texttt{glLight\*}(...) functions before they are available for use. You define the position and color of your lights (including their ambient, specular, and diffuse contributions) as illustrated for the light \texttt{GL\_LIGHT0} by the following position definition and definition of the first of the three lights in the three-light example

\begin{verbatim}
    glLightfv(GL_LIGHT0, GL_POSITION, light_pos0 ); // light 0
    glLightfv(GL_LIGHT0, GL_AMBIENT,    amb_color0 );
    glLightfv(GL_LIGHT0, GL_DIFFUSE,     diff_col0 );
    glLightfv(GL_LIGHT0, GL_SPECULAR,    spec_col0 );
\end{verbatim}

Here we use a light position and specific light colors for the specular, diffuse, and ambient colors that we must define in separate statements such as those below.

\begin{verbatim}
    GLfloat light_pos0 = { ..., ..., ... };
    GLfloat diff_col0  = { ..., ..., ... };
\end{verbatim}

In principle, both of these vectors are four-dimensional, with the fourth value in the position vector being a homogeneous coordinate value and with the fourth value of the color vector being the alpha value for the light. We have not used homogeneous coordinates to describe our modeling, but they are not critical for us. We have used alpha values for colors, of course, but the default value for alpha in a color is 1.0 and unless you want your light to interact with your blending design somehow, we suggest that you use that value for the alpha component of light colors, which you can do by simply using RGB-only light definitions as we do in the example at the end of this chapter.

As we noted earlier in this chapter, you must define normals to your objects’ surfaces for lighting to work successfully. Because the lighting calculations involve cosines that are calculated with dot products with the normal vector, however, you must make sure that your normal vectors are all of unit length. You can ensure that this is the case by enabling automatic normalization with the function call \texttt{glEnable(GL\_NORMALIZE)} before any geometry is specified in your display function.

Before any light is available to your scene, the overall lighting operation must be enabled and then each of the individual lights to be used must also be enabled. This is an easy process in OpenGL. First, you must specify that you will be using lighting models by invoking the standard enable function

\begin{verbatim}
    glEnable(GL_LIGHTING);   // so lighting models are used
\end{verbatim}

Then you must identify the lights you will be using by invoking an enable function for each light, as illustrated by the following setup of all three lights for the three-light case of the example below:

\begin{verbatim}
    glEnable(GL_LIGHT0);     // use LIGHT0
    glEnable(GL_LIGHT1);     // and LIGHT1
    glEnable(GL_LIGHT2);     // and LIGHT2
\end{verbatim}

Lights may also be disabled with the \texttt{glDisable(...) function, so you may choose when to have a particular light active and when to have it inactive in an animation or when carrying out a particular display that may be chosen, say, by a user interaction.

In addition to the ambient light that is contributed to your scene from each of the individual lights’ ambient components, you may define an overall ambient light for the scene that is independent of any particular light. This is done with the function:

\begin{verbatim}
    glLightModelf(GL\_LIGHT\_MODEL\_AMBIENT, r, g, b, a)
\end{verbatim}

and the value of this light is added into the overall ambient lighting computation.

The remaining properties of lights that we discussed earlier in this chapter are also straightforward to set in OpenGL. If you want a particular light to be a spotlight, you will need to set the direction, cutoff, and dropoff properties that we described earlier in this chapter, as well as the standard
position property. These additional properties are set with the `glLightf*(...)` functions as follows:

```c
    glLightf(light, GL_SPOT_DIRECTION, -1.0, -1.0, -1.0);
    glLightf(light, GL_SPOT_CUTOFF, 30.0);
    glLightf(light, GL_SPOT_EXPONENT, 2.0);
```

If you do not specify the spotlight cutoff and exponent, these are 180 degrees (which means that the light really isn’t a spotlight at all) and the exponent is 0. If you do set the spotlight cutoff, the value is limited to lie between 0 and 90, as we described earlier.

Attenuation is not modeled realistically by OpenGL, but is set up in a way that can make it useful. There are three components to attenuation: constant, linear, and quadratic. The value of each is set separately as noted above with the symbolic constants `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION`, and `GL_QUADRATIC_ATTENUATION`. If these three attenuation coefficients are \( A_C \), \( A_L \), and \( A_Q \), respectively, and the distance of the light from the surface is \( D \), then the light value is multiplied by the attenuation factor

\[
A = \frac{1}{(A_C + A_L \times D + A_Q \times D^2)}
\]

where \( D \) is the distance between the light position and the vertex where the light is calculated. The default values for \( A_C \), \( A_L \), and \( A_Q \) are 1.0, 0.0, and 0.0 respectively. The actual values of the attenuation constants can be set by the `glLightf(GL_*_ATTENUATION, value)` functions, where the wildcard is to be replaced by one of the three symbolic constants mentioned above.

A directional light is specified by setting the fourth component in its position to be zero. The direction of the light is set by the first three components, and these are transformed by the modelview matrix. Such lights cannot have any attenuation properties but otherwise work just like any other light: its direction is used in any diffuse and specular light computations but no distance is ever calculated. An example of the way a directional light is defined would be

```c
    glLightf(light, GL_POSITION, 10.0, 10.0, 10.0, 0.);
```

### Defining materials

In order for OpenGL to model the way a light interacts with an object, the object must be defined in terms of the way it handles ambient, diffuse, and specular light. This means that you must define the color of the object in ambient light and the color in diffuse light. (No, we can’t think of any cases where these would be different, but we can’t rule out the possibility that this might be used somehow.) You do not define the color of the object in specular light, because specular light is the color of the light instead of the color of the object, but you must define the way the material handles the specular light, which really means how shiny the object is and what color the shininess will be. All these definitions are handled by the `GL_MATERIAL*` function.

Recall that any polygon has two sides, which we will call the **front** side and **back** side. The difference between these is the direction of the normal to the polygon, with the front side being the side toward which the normal points. Because the normal can represent the direction of the cross product of two polygon edges in the order in which edges go around the polygon, and because of the right-hand rule for determining the direction of the cross product, we can avoid the reference to the polygon normal and simply note that the front side is the side from which the edges of the polygon are in counterclockwise order (or the side for which the angles from an interior point of the polygon to the vertices are in increasing order).

If you use two-sided lighting, when you specify the properties for your material, you must specify them for both the front side and the back side of the material. You can choose to make these properties the same by defining your material with the parameter `GL_FRONT_AND_BACK` instead of defining `GL_FRONT` and `GL_BACK` separately. This will allow you to use separate colors for
the front side and back side of an object, for example, and make it clear which side is being seen in case the object is not closed.

To allow us to define an object’s material properties we have the `glMaterial*(...)` function family. These functions have the general form

```
glMaterial[i|f][v](face, parametername, value)
```

and can take either integer or real parameter values ([i|f]) in either individual or vector ([v]) form. The parameter `face` is a symbolic name that must be one of `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`. The value of `parametername` is a symbolic name whose values can include `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_EMISSION`, `GL_SHININESS`, or `GL_AMBIENT_AND_DIFFUSE`. Finally, the `value` parameter is either a single number, a set of numbers, or a vector that sets the value the symbolic parameter is to have in the OpenGL system.

Below is a short example of setting these values, taken from the example at the end of the chapter.

```
GLfloat shininess[ ]={ 50.0 };
GLfloat white[ ] = { 1.0, 1.0, 1.0, 1.0};
glMaterialfv(GL_FRONT, GL_AMBIENT,   white );
glMaterialfv(GL_FRONT, GL_DIFFUSE,   white );
glMaterialfv(GL_FRONT, GL_SPECULAR,  white );
glMaterialfv(GL_FRONT, GL_SHININESS, shininess );
```

This gives the material a very neutral property that can pick up whatever colors the light should provide for its display.

Most of the parameters and values are familiar from the earlier discussion of the different aspects of the lighting model, but the `GL_AMBIENT_AND_DIFFUSE` parameter is worth pointing out because it is very common to assume that a material has the same properties in both ambient and diffuse light. (Recall that in both cases, the light energy is absorbed by the material and is then re-radiated with the color of the material itself.) This parameter allows you to define both properties to be the same, which supports this assumption.

**Setting up a scene to use lighting**

To define a triangle with vertex points \( P[0], P[1], \) and \( P[2] \), compute its normal, and use the calculated normal, we would see code something like this:

```
glBegin(GL_POLYGON);
   // calculate the normal Norm to the triangle
   calcTriangleNorm(p[0],P[1],P[2],Norm);
   glNormal3fv(Norm);
   glVertex3fv(P[0]);
   glVertex3fv(P[1]);
   glVertex3fv(P[2]);
glEnd();
```

**Using GLU quadric objects**

As we discussed when we introduced the GLU quadric objects in the modeling chapter, the OpenGL system can generate automatic normal vectors for these objects. This is done with the function `gluQuadricNormals(GLUquadric* quad, GLenum normal)` that allows you to set `normal` to either `GLU_FLAT` or `GLU_SMOOTH`, depending on the shading model you want to use for the object.

**An example: lights of all three primary colors applied to a white surface**

Some lighting situations are easy to see — when you put a white light on a colored surface, you see the color of the surface, because the white light contains all the light components and the
surface has the color it reflects among them. Similarly, if you shine a colored light on a white surface, you see the color of the light because only that color is available. When you use a colored light on a colored surface, however, it gets much more complex because a surface can only reflect colors that come to it. So if you shine a (pure) red light on a (pure) green surface you get no reflection at all, and the surface seems black. You don’t see this in the real world because you don’t see lights of pure colors, but it can readily happen in a synthetic scene.

Considering the effect of shining colored lights on a white surface, let’s look at an example. A white surface will reflect all the light that it gets, so if it gets only a red light, it should be able to reflect only red. So if we take a simple shape (say, a cube) in a space with three colored lights (that are red, green, and blue, naturally), we should see it reflect these different colors. In the threelightcube example we discuss below, we define three lights that shine from three different directions on a white cube. If you add code that lets you rotate the cube around to expose each face to one or more of the three lights, you will be able to see all the lights on various faces and to experiment with the reflection properties they have. This may let you see the effect of having two or three lights on one of the faces, as well as seeing a single light. You may also want to move the lights around and re-compile the code to achieve other lighting effects.

There is a significant difference between the cube used in this example and the cube used in the simple lighting example in a previous module. This cube includes not only the vertices of its faces but also information on the normals to each face. (A normal is a vector perpendicular to a surface; we are careful to make all surface normals point away from the object the surface belongs to.) This normal is used for many parts of the lighting computations — to determine whether you’re looking at a front or back face, for example, and to compute both the diffuse light and the specular light for a polygon. We refer you to any standard graphics text for more details.

Code for the example

Defining the light colors and positions in the initialization function:

```c
GLfloat light_pos0[]={ 0.0, 10.0, 2.0, 1.0 }; // light 1: up y-axis
GLfloat light_col0[]={ 1.0, 0.0, 0.0, 1.0 }; // light is red
GLfloat amb_color0[]={ 0.3, 0.0, 0.0, 1.0 }; // even ambiently

GLfloat light_pos1[]={ 5.0, -5.0, 2.0, 1.0 }; // light 2: lower right
GLfloat light_col1[]={ 0.0, 1.0, 0.0, 1.0 }; // light is green
GLfloat amb_color1[]={ 0.0, 0.3, 0.0, 1.0 }; // even ambiently

GLfloat light_pos2[]={ -5.0, 5.0, 2.0, 1.0 }; // light 3: lower left
GLfloat light_col2[]={ 0.0, 0.0, 1.0, 1.0 }; // light is blue
GLfloat amb_color2[]={ 0.0, 0.0, 0.3, 1.0 }; // even ambiently
```

Defining the light properties and the lighting model in the initialization function:

```c
glLightfv(GL_LIGHT0, GL_POSITION, light_pos0 ); // light 0
glLightfv(GL_LIGHT0, GL_AMBIENT, amb_color0 );
glLightfv(GL_LIGHT0, GL_SPECULAR, light_col0 );
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_col0 );

/* light 1 */
glLightfv(GL_LIGHT1, GL_POSITION, light_pos1 );
glLightfv(GL_LIGHT1, GL_AMBIENT, amb_color1 );
glLightfv(GL_LIGHT1, GL_SPECULAR, light_col1 );
glLightfv(GL_LIGHT1, GL_DIFFUSE, light_col1 );

/* light 2 */
glLightfv(GL_LIGHT2, GL_POSITION, light_pos2 );
glLightfv(GL_LIGHT2, GL_AMBIENT, amb_color2 );
glLightfv(GL_LIGHT2, GL_SPECULAR, light_col2 );
glLightfv(GL_LIGHT2, GL_DIFFUSE, light_col2 );
```
glLightModeliv(GL_LIGHT_MODEL_TWO_SIDE, &i); // two-sided lighting

Enabling the lights in the initialization function:
- glEnable(GL_LIGHTING);  // so lighting models are used
- glEnable(GL_LIGHT0);    // we'll use LIGHT0
- glEnable(GL_LIGHT1);    // ...  and LIGHT1
- glEnable(GL_LIGHT2);    // ...  and LIGHT2

Defining the material color in the function that draws the surface: we must define the ambient and
diffuse parts of the object’s material specification, as shown below; note that the shininess value
must be an array. Recall that higher values of shininess will create more focused and smaller
specular highlights on the object. That this example doesn’t specify the properties of the material’s
back side because the object is closed and all the back side of the material is invisible.

GLfloat shininess[]={ 50.0 };
glMaterialfv(GL_FRONT, GL_AMBIENT, white );
glMaterialfv(GL_FRONT, GL_DIFFUSE,   white );
glMaterialfv(GL_FRONT, GL_SHININESS, shininess);

Figure 7.5 below shows the cube when it is rotated so one corner points toward the viewer. Here
the ambient light contributed by all three of the lights keeps the colors somewhat muted, but clearly
the red light is above, the green light is below and to the right, and the blue light is below and to
the left of the viewer’s eyepoint. The lights seem to be pastels because each face still gets some of
the other two colors; to change this you would need to change the positions of the lights.

Figure 7.5: the white cube viewed with three colored lights

A word to the wise...

The OpenGL lighting model is essentially the same as the basic lighting model of all standard
graphics APIs, but it lacks some very important things that might let you achieve some particular
effects you would want if you were to try to get genuine realism in your scenes. One of the most
important things lacking in the simple lighting model here is shadows; while OpenGL has
techniques that can allow you to create shadows, they are tricky and require some special effort.
Another important missing part is the kind of “hot” colors that seem to radiate more of a particular
color than they could possibly get in the light they receive, and there is no way to fix this because
of the limited gamut of the phosphors in any computer screen, as described in many textbooks.
Finally, OpenGL does not allow the kind of directional reflection that you would need to model
materials such as brushed aluminum, which can be created on the computer with special
programming. So do not take the OpenGL lighting model as the correct way to do color; take it as
a way that works pretty well and that would take much more effort to do better.
Lighting is a seductive effect because it engages our perceptual system to identify shapes of things. This can be very effective, but beware of applying lighting where your shapes or colors are purely arbitrary and represent abstract concepts. It can be dangerous to infer shapes by lighting where there is no physical reality to the things being displayed.

**Science Examples**

Probably the best example of the use of lights and lighting in a scientific application would occur when you want to emphasize the shape of something by the use of lighting. This could certainly be the case for any surface that represents any mathematical or physical function.