Lights and Lighting

Prerequisites

An understanding of color at the level of the discussion of the earlier module in this set, and some enlightened observation of the way lights work in the world around you.

Introduction

OpenGL allows you to define lights in your scene and to have the lights provide illumination in a way that approximates the way light works in the real world. You can also define your objects to have properties that simulate the light behavior of real materials. Objects and lights have properties that model those found in these things in the real world, and lights interact with objects in a way that emulates (though simply) the real world. This chapter begins to examine how this works.

Part of the way that lighting gives your scene a bit more realism is to model the nature of light in simple ways. In this model there are basically three components of lighting: ambient, diffuse, and specular. 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, 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. This light is the color of the light source, not the object that reflects the light.

The sum of these light components is the light that is actually seen, and the lighting calculations take all three into account in a way that depends on the material being illuminated. Thus in order to use lights in a scene, you must define your 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.

Definitions

Ambient, diffuse, and specular light

Ambient light 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. This would be the light on the underside of an object, for example. The amount of diffuse light is given simply by \( A = C_A \) for a constant \( C_A \) that depends on the light and material. 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.

Ambient light can come from each individual light source, as well as from an overall ambient light value. When you define your light model, you can set

\[
glLightModelf(GL\_LIGHT\_MODEL\_AMBIENT, r, g, b, a)\]

to define an ambient light that is added into the overall lighting computation separately from the ambient contribution from each individual light.

Diffuse light is absorbed by the surface of the object and is 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 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 diffuse light as \( D = C_D \cdot \cos(N \cdot L) \) for some constant \( C_D \) that depends on the light and material, which shows why we must have surface normals in order to calculate diffuse light. Note that diffuse light is independent of the point from which we view the material; this is obvious from looking at the world around us and the lighting model supports that property, though it is a little involved to demonstrate.

![Figure 7.1: diffuse lighting](image)

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 OpenGL model for specular light allows you to define the shininess of an object to control that spread. Shininess is controlled by a parameter (called \text{GL_SHININESS}, naturally) which gives smaller, brighter highlights as it increases, as shown in the three successive figures of Figure 7.3. The end result is that the contribution of the specular light is given by \( S = C_S \cdot \cos^N (E \cdot R) \) for some constant \( C_S \) that depends on the light and material.

![Figure 7.2: the specular lighting diagram](image)
Figure 7.3: specular highlights with shininess coefficients 20, 50, and 80 (left, center, and right), respectively

So with the mechanics of 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 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 using the function `glNormal*(...)` to specify a normal vector at a point. However, this is not enough — 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 this may not even be enough because scaling or other computations can change the length of the normals. You can ensure that your normals are normalized by enabling normalization by setting `glEnable(GL_NORMALIZE)` at the top of your program.

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.

Use of material specifications

In order for OpenGL to model the way a light interacts with an object, the object must be defined in 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 must also 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 will have both a front side (the side that is in the direction the normal is pointing) and a back side. The lighting model can treat only the front side or can treat both the front and back sides of your polygons; the function

\[ \text{glLightModel*(GL_LIGHT_MODEL_TWO_SIDE, value)} \]

specifies two-sided lighting if the value parameter is non-zero. 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 make them 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.

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**

OpenGL allows you to define a number of properties for a light: its position, its color, how it is attenuated (diminished) over distance, and whether it is an omnidirectional light or a spotlight. We will not cover all these properties here, but the properties of position and color are critical. The other properties are primarily useful if you are trying to achieve particular realistic effects in your scene. The position and color properties are illustrated in the example at the end of this section.

**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 a direction, a cutoff, and an exponent.

![Figure 7.4: spotlight direction and cutoff](image)
The direction is simply a 3D vector, the cutoff is a real value between 0.0 and 90.0 that represents half the spread of the spotlight (so a smaller cutoff represents a more focused light), and the exponent controls the difference between the intensity at the center of the spotlight and the intensity at the edge. The spotlight properties are set with functions as shown in the following example:

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

**Attenuation**

In the real world, the energy from a light source on a unit surface diminishes as the light source is farther from the surface. This diminishing is called attenuation, and computer graphics can model that behavior accurately or can treat attenuation in other ways. Initially, attenuation is not set, but you can set it with `glLightf(GL_*_ATTENUATION, value)`.

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, then the light value is multiplied by the attenuation factor

$$A = 1/(A_C + A_L*D + A_Q*D^2)$$

As we noted, the default value of the attenuation coefficients are 1, 0, and 0, respectively.

**Moving lights**

Positional lights can be critical components of a scene, because they determine how shapes and contours can be seen. Because lights are affected by all the transformations present in the modelview matrix when the light position is defined, we must be very careful where that definition is done.

- If the light is to be at a fixed place in the scene, then you define the light position immediately after you set the modelview matrix to be the identity. This creates an absolute position independent of the eye position or any modeling in the scene.
- If the light is to be at a fixed place relative to the eye point, then you define the light position immediately after you set your eye point with the `gluLookAt(...)` function. The light position is then modified by the eyepoint transformation, 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 if it were part of that object. As we described when we talked about modeling, the eye location is set after the object’s overall position and orientation were defined.
- If the light is to move around in the scene on its own, then the light position is set after whatever modeling is needed to achieve the motion.

So 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.

**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. This is done by
specifying the fourth component in the light 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.

Enabling lights

This is an easy process in OpenGL. First, you must specify that you will be using lighting models by invoking the standard enable function

```c
glEnable(GL_LIGHTING);   // so lighting models are used
```

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

```c
glEnable(GL_LIGHT0);     // we'll use LIGHT0
glEnable(GL_LIGHT1);     // ... and LIGHT1
glEnable(GL_LIGHT2);     // ... and LIGHT2
```

Finally, you must define the position and color of your lights (including their ambient, specular, and diffuse contributions) as illustrated by the following, also from the definition of one of the lights in the three-light example

```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 );
```

Note that in this case we use the light color for the specular and diffuse components but use an overall ambient color that is independent of this particular light.

An example

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. When you use a colored light, 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. 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.

However, what if you shine colored lights on a white surface? 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) and shine lights of different colors on it, we should see it reflect these different colors. In the three-lightcube example we provide with this module, we define three lights that shine from three different directions on a white cube, and you can rotate the cube around to expose each face to one or more of the three lights. This will let you see all the lights on various faces and to experiment with the reflection properties they have. You can also 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 examples

Example: Lights of all three primary colors applied to a white surface

This example illustrates how lights interact with a surface. We have a white cube with fairly standard surface properties, and this cube is in a space with three colored lights (that are red, green, and blue, naturally). The three lights are in the space between the viewer and the cube, so initially they all light the single face of the cube that is visible. However, as the viewer rotates the cube, the motion can obscure one or more of the lights on faces seen by the viewer, so the faces will seem to change color. These colors are caused by having the faces illuminated by only some of the lights.

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 );

GLfloat mat_shininess[] = { 50.0 };```

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

glLightfv(GL_LIGHT2, GL_POSITION, light_pos2 ); // light 2
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:

```c
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, recalling that higher values of shininess will create more focused and smaller specular highlights on the object. Note that this example doesn’t deal with the material’s back side because the object is closed and all the back side of the material is invisible.
glMaterialfv(GL_FRONT, GL_AMBIENT, white);
glMaterialfv(GL_FRONT, GL_DIFFUSE, white);
glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, mat_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.

![Image of the white cube viewed with three colored 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 systems, but it lacks some very important things that might let you achieve some kinds of
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.

Example Code
threeelightcube.c

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.