Chapter 8: Lighting and Shading

This chapter introduces the next level beyond color for creating improved computer graphics images. It focuses on two related areas that give improved representations of how images look: lighting that is based on a simple model of the interaction of lights and surfaces, and shading that is based on a simple model of how color changes across a graphics surface primitive. The lighting model is based on three simple components of light that model indirect, direct, and reflective lighting, and introduces both the idea of a light and of the light properties of an object’s materials. The shading model is based on the idea of averaging the color of the vertices of a polygon to get the color of the interior of the polygon. Between them, you will be able to make images that look much more realistic than the images you could make with simple color.

This chapter needs you to have an understanding of color at the level of the earlier chapter on color, an observation of the way lights work in the world around you, and an understanding of the concept of polygons and of simple averaging across a polygon.

Lighting

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 simplistic, but it’s very easy to create images this way because we must simply set a color when we define and draw our geometry. In fact, when objects don’t have an intrinsic color but we use their color to represent something else, as we did sometimes in the science examples in an earlier chapter, then this is an appropriate approach to color for images.

However, it’s clear that things in the real world don’t simply have a color that you see; the color you see is strongly influenced by the lighting conditions in which you see them, so it’s inaccurate not to take the light into account in presenting a scene. So the second way to think of how we see things is to realize that we see things because of light that reaches us after a physical interaction between light in the world and the objects we see. Light sources emit energy that reaches the objects and the energy is then re-sent to us by reflections that involve both the color of the light and the physical properties of the objects. In computer graphics, this approach to lighting that considers light that comes only from lights in the scene is called the local illumination model. It is handled by having the graphics programmer define the lights in the scene. In this chapter we will discuss how this approach to 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 with the local illumination model depends on modeling 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 comes from an object that reflects a subset of the wavelengths it receives and 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 the light without interacting with it.
and giving the light the color of the object. This light is generally the color of the light source, not the object that reflects the light.

These three components are computed based on properties of the lights, properties of the materials that are in your scene, and the geometric relationship between the lights and the objects. All three of these components contribute to the overall light as defined by the RGB light components, and the graphics system models this by applying them to the RGB components separately. The sum of these three 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, our simple models of realistic lighting tend 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.

So 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 and will require more thought as you model the objects in your scenes, but the changes will be something you can handle without difficulty. Graphics APIs have their own ways 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 from a single light 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. If you have multiple lights, the total diffuse light is \( A = \sum L_A \cdot C_A \), where the sum is over all the lights. If you want to emphasize the effect of the lights, you will probably want to have your ambient light at a fairly low level (so you get the effect of lights at night or in a dim room) or you can use a fairly high ambient level if you want to see everything in the scene with a reasonably uniform light. If you want to emphasize shapes, use a fairly low ambient light so the shading can bring out the variations in the surface.

Diffuse light comes from specific light sources and is reflected by the surface of the object at a particular wavelength depending on properties of the object’s material. The general model for diffuse light, used by OpenGL and other APIs, is based on the idea of brightness, or light energy per unit area. A light emits a certain amount of energy per unit area in the direction it is shining, and when this falls on a surface, the intensity of light seen on the surface is proportional to the surface area illuminated by the unit area of the light. As the diagram in Figure 8.1 shows, one unit of area as seen by the light, or one unit of area perpendicular to the light direction, illuminates an
area of $1/\cos(\Theta)$ in the surface. So if we have $L_D$ light energy per unit of area in the light direction, we have $L_D \cos(\Theta)$ units of light energy per unit of area on the surface if the cosine is positive. As the angle of incidence of the light on the surface decreases, the amount of light at the surface becomes reduced, 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.

Now that we know the amount of diffuse light energy per unit of surface, how does that appear to the eye? Diffuse light is reflected from a surface according to Lambert’s law that states that the amount of light reflected in a given direction is proportional to the cosine of the angle from the surface in that direction. If the amount of diffuse light per unit surface is $D$ and the unit vector from the surface to your eye is named $E$, then the amount of energy reflected from one unit of surface to your eye is $D \cos(\Theta) = D \cdot (E \cdot N)$. But the unit of surface area is not seen by your eye to be a unit of area; its area is $\cos(\Theta) = E \cdot N$. So the intensity of light you perceive at the surface is the ratio of the energy your eye receives and the area your eye sees, and this intensity is simply $D$—which is the same, no matter where your eye is. Thus the location of the eye point does not participate in the diffuse light computation, which fits our observation things do not change color as the angle between their normals and our eye direction does not change as we move around.

Based on the discussions above, we now have the diffuse lighting calculation that computes the intensity of diffuse light as

$$D = L_D \cdot C_D \cdot \cos(\Theta) = L_D \cdot C_D \cdot (L \cdot N)$$

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. Again, if you have several lights, the diffuse light is

$$D = \sum L_D \cdot C_D \cdot (L \cdot N)$$

where the sum is over all the lights (including a different light vector $L$ for each light). Our use of the dot product instead of the cosine assumes that our normal vector $N$ and light vector $L$ are of unit length, as we discussed when we introduced the dot product. This computation is done separately for each light source and each object, because it depends on the angle from the object to the light.

Diffuse light interacts with the objects it illuminates in a way that produces the color we see in the objects. The object does not reflect all the light that hits it; rather, it absorbs certain wavelengths (or colors) and reflects the others. The color we see in an object, then, comes from the light that is reflected instead of being absorbed, and it is this behavior that we will specify when we define the diffuse property of materials for our scenes.

**Specular** light is a surface phenomenon that produces bright highlights on shiny surfaces. Specular light depends 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 reflected according to Lambert’s law as diffuse light is, but is reflected with the angle of incidence equal to the angle of reflection, as illustrated in Figure 8.2. 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 called the *specularity coefficient* which gives smaller, brighter highlights as it increases and makes the material seem increasingly shiny, as shown in the three successive figures of Figure 8.3. The computation of the reflection vector was described in the chapter on mathematics in computer graphics, and this reflection vector is calculated as \( R = 2(N \cdot L)N - L \) when we take into effect the different direction of the light vector \( L \) and use the names in Figure 8.2.

The specular light seen on an object in the image is computed by the equations

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

for a light’s specularity value \( L_S \) and the object’s specular coefficient \( C_s \). Again, we assume the eye vector \( E \) and reflection vector \( R \) have unit length in order to use the dot product for the cosine. The specular light depends on the angle between the eye and the light reflection, because it is light that is reflected directly from the surface of an object in a mirror-like way. Because it describes the extent to which the surface acts shiny, the specular coefficient is also called the *shininess* coefficient, and larger values give shinier materials. Shininess is a relative term, because even rather shiny materials may have small surface irregularities that keep the reflection from being perfectly mirror-like. Note the visual effect of increasing the specularity coefficient: the highlight gets smaller and more focused—that is, the sphere looks shinier and more polished. Thus the specular or shininess coefficient is part of the definition of a material. This produces a fairly good model of shininess, because for relatively large values of \( N \) (for example, \( N \) near or above 30) the function \( \cos^N(\Theta) \) has value very near one if the angle \( \Theta \) is small, and drops off quickly as the angle increases, with the speed of the dropoff increasing as the power is increased. 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.

Specular light behaves quite differently from diffuse light in the way it interacts with objects. We generally assume that no light is absorbed in the specular process so that the color of the specular highlight is the same as the color of the light, although it may be possible to specify a material that behaves differently.

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.

Later in this 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 do a lighting computation could give us a color for each vertex, which would require us to determine a separate normal 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 when we discuss shading later in this 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 most graphics APIs with very specialized programming. We will discuss a simple approach to shadows based on texture mapping later in the book.
Surface normals

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 a previous chapter, we saw that the cross product of two vectors is another vector that is perpendicular to both of them. If we have a polygon (always considered here to be convex), we can calculate a vector parallel to any edge by calculating the difference of the endpoints of the edge. If we do this for two adjacent edges, we get two vectors that lie in the polygon’s plane, so their cross product is a vector perpendicular to both edges, and hence perpendicular to the plane, and we have created a normal to the polygon. When we do this, it is a very good idea to calculate the length of this normal and divide this length into each component of the vector in order to get a unit vector because most APIs’ lighting model assumes that our normals have this property.

It is also possible to compute normals from analytic considerations if we have the appropriate modeling to work with. If we have an object whose surface is defined by differentiable parametric equations, then we can take appropriate derivatives of those equations to get two directions for the tangent plane to the surface, and take their cross product to get the normal.

In both cases, we say we have the “normal” but in fact, a surface normal vector could go either way from the surface, either “up” or “down”. We need to treat surface normals consistently in order to get consistent results for our lighting. For the approach that takes the cross product of edges, you get this consistency by using a consistent orientation for the vertices of a polygon, and this is usually done by treating the vertices, and thus the edges, as proceeding in a counterclockwise direction. Another way to say this is to take any interior point and the line from the interior point to the first vertex. Then the angle from that line to the line from the point to any other vertex will increase as the vertex index increases. The orientation for analytic normals is simpler. If we have the x- and y-direction vectors in the tangent plane to the surface, we recall the right-hand rule for cross products and always take the order (x-direction)x(y-direction) in order to get the z-direction, which is the normal.

Because a polygon is a plane figure, it has two sides which we will call the front side and back side. If we take the conventions above for the normal, then the normal will always point out of a polyhedron, will always point up on a surface defined as a function of two variables, or will always point toward the side of a polygon in which the vertices appear to be in counterclockwise order. This side towards which the normal points is called the front side, and this distinction is important in material considerations below as well as other graphical computations.

Light properties

As we noted at the beginning of this chapter, we will focus on a local illumination model for our discussion of lighting. Because all the lighting comes from lights you provide, lights are critical components of your modeling work in defining an image. In the chapter on the principles of
modeling, we saw how to include lights in your scene graph for an image. 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.

**Light color**

Light color is usually seen as having a single value, which like all colors can be expressed in terms of the RGB or RGBA color model. Because the lighting model separates ambient, diffuse, and specular components, however, some graphics APIs allow you to specify the ambient, diffuse, and specular colors of a light separately. You will want to look at your API carefully to see what model it uses for colors and for the lighting model components.

**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). This represents the point where the light is placed, represented in homogeneous coordinates. If the fourth coordinate is zero, which is invalid for actual coordinates, the light will be treated as a directional light as described below. 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 8.4. 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 8.4: spotlight direction and cutoff](image-url)
In more detail, if a spotlight has position $P$, dropoff $d$, direction $D$, and cutoff $\Theta$, the light energy at a point $Q$ in a direction $V$ from the spotlight would then be zero if the absolute value of the dot product $(Q-P) \cdot D$ were larger than $\cos(\Theta)$, or would be multiplied by $((Q-P) \cdot D)^d$ if the absolute value of the dot product is less than $\cos(\Theta)$.

**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 $d$ from the light, multiplying the light energy by $k/d^2$, 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 a point 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 simply 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.
Materials

Lighting involves both the specification of the lights in a scene and the light-related properties of the objects in the scene. If you want to use lighting in creating a scene, you must specify both of these: the properties of your lights, and the properties of the materials of your objects. In this section we discuss the nature of material specifications. Implementing lighting for a scene 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 material properties: $C_A$, the reflectivity of the material in ambient light, $C_D$, the reflectivity of material in diffuse light; $C_S$, the reflectivity of the material in specular light; and $N$, the shininess coefficient. The three reflectivity terms involve color, so they are specified in RGB terms, while the shininess coefficient is the exponent of the dot produce $R \cdot E$ in the specular component. These all take part in the computations of the color of the object from the lighting model, and are called the material specifications. 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. The API may have some other kinds of behavior for materials, such as specifications of the front and back sides of a material separately; this is the case for OpenGL, but it may vary between APIs.

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.

Shading

Shading is the process of computing the color for the components of a scene. It is usually done by calculating the effect of light on each object in a scene to create an effective lighted presentation. The shading process is thus based on the physics of light, and the most detailed kinds of shading computation can involve deep subtleties of the behavior of light, including the way light scatters from various kinds of materials with various details of surface treatments. Considerable research has been done in those areas and any genuinely realistic rendering must take a number of surface details into account.

Most graphics APIs do not have the capability to do these detailed kinds of computation. The usual beginning API such as OpenGL supports two shading models for polygons: flat shading and smooth shading. You may choose either, but smooth shading is usually more pleasing and can be somewhat more realistic. Unless there is a sound reason to use flat shading in order to represent data or other communication concepts more accurately, you will probably want to use smooth shading for your images. We will briefly discuss just a bit more sophisticated kinds of shading, even though the beginning API cannot directly support them.

Definitions

Flat shading of a polygon presents each polygon with a single color. This effect is computed by assuming that each polygon is strictly planar and all the points on the polygon have exactly the same kind of lighting treatment. The term flat can be taken to mean that the color is flat (does not vary) across the polygon, or that the polygon is colored as though it is flat (planar) and thus does
not change color as it is lighted. This is the effect you will get if you simply set a color for the polygon and do not use a lighting model (the color is flat), or if you use lighting and materials models and then display the polygon with a single normal vector (the polygon is flat). This single normal allows you only a single lighting computation for the entire polygon, so the polygon is presented with only one color.

**Smooth shading** of a polygon displays the pixels in the polygon with smoothly-changing colors across the surface of the polygon. This requires that you provide information that defines a separate color for each vertex of your polygon, because the smooth color change is computed by interpolating the vertex colors across the interior of the triangle with the standard kind of interpolation we saw in the graphics pipeline discussion. The interpolation is done in screen space after the vertices’ position has been set by the projection, so the purely linear calculations can easily be done in graphics cards. This per-vertex color can be provided by your model directly, but it is often produced by per-vertex lighting computations. In order to compute the color for each vertex separately you must define a separate normal vector for each vertex of the polygon so that the lighting model will produce different colors at each vertex.

Every graphics API will support flat shading in a consistent way, but different graphics APIs may treat smooth shading somewhat differently, so it is important for you to understand how your particular API handles this. The simplest smooth shading is done by calculating color at the vertices of each polygon and then interpolating the colors smoothly across the polygon. If the polygon is a triangle, you may recall that every point in the triangle is a convex combination of the vertices, so you may simply use that same convex combination of the vertex colors. As computer graphics becomes more sophisticated, however, we will see more complex kinds of polygon shading in graphics APIs so that the determination of colors for the pixels in a polygon will become increasingly flexible.

**Examples of flat and smooth shading**

We have seen many examples of polygons earlier in these notes, but we have not been careful to distinguish between whether they were presented with flat and smooth shading. Figure 8.5 shows see two different images of the same relatively coarsely-defined function surface with flat shading (left) and with smooth shading (right), to illustrate the difference. Clearly the smooth-shaded image is much cleaner, but there are still some areas where the triangles change direction very

![Figure 8.5: a surface with flat shading (left) and the same surface with smooth shading (right)](image-url)
quickly and the boundaries between the triangles still show as color variations in the smoothly-
shaded image. Smooth shading is very nice—probably nicer than flat shading in many
applications—but it isn't perfect.

The computation for this smooth shading uses simple polygon interpolation in screen space.
Because each vertex has its own normal, the lighting model computes a different color for each
vertex. The interpolation then calculates colors for each pixel in the polygon that vary smoothly
across the polygon interior, providing a smooth color graduation across the polygon. This
interpolation is called Gouraud shading and is one of the standard techniques for creating images.
It is quick to compute but because it only depends on colors at the polygon vertices, it can miss
lighting effects within polygons. Visually, it is susceptible to showing the color of a vertex more
strongly along an edge of a polygon than a genuinely smooth shading would suggest, as you can
see in the right-hand image in Figure 9.5. Other kinds of interpolation are possible that do not
show some of these problems, though they are not often provided by a graphics API, and one of
these is discussed below.

An interesting experiment to help you understand the properties of shaded surfaces is to consider
the relationship between smooth shading and the resolution of the display grid. In principle, you
should be able to use fairly fine grid with flat shading or a much coarser grid with smooth shading
to achieve similar results. You should define a particular grid size and flat shading, and try to find
the smaller grid that would give a similar image with smooth shading. Figure 8.6 is an example of
this experiment. The surface still shows a small amount of the faceting of flat shading but avoids
much of the problem with quickly-varying surface directions of a coarse smooth shading. It is
probably superior in many ways to the smooth-shaded polygon of Figure 8.5. It may be either
faster or slower than the original smooth shading, depending on the efficiency of the polygon
interpolation in the graphics pipeline. This is an example of a very useful experimental approach to
computer graphics: if you have several different ways to approximate an effect, it can be very
useful to try all of them that make sense and see which works better, both for effect and for speed,
in a particular application!

![Figure 8.6: a flat-shaded image with three times as many subdivisions in each direction as the
previous figure](image)

**Calculating per-vertex normals**

The difference between the programming for these two parts of Figure 8.5 is that the image with
flat-shaded polygons uses only one normal per polygon (calculated by computing the cross product
of two edges of each triangle or by computing an analytic normal for any point in the polygon), while the image with smooth-shaded polygons uses a separate normal per vertex (in this case, calculated by using analytic processes to determine the exact value of the normal at the vertex). It can take a bit more work to compute the normal at each vertex instead of only once per polygon, but that is the price one must pay for smooth shading.

There are a number of ways you may calculate the normal for a particular vertex of a model. You may use an interpolation technique, in which you compute a weighted average of the normals of each of the polygons that includes the vertex, or you may use an analytic computation. The choice of technique will depend on the information you have for your model.

**Averaging polygon normals**

In the interpolation technique, you can calculate the normal \( N \) at a vertex by computing the weighted average of the normals for all the polygons that meet at the vertex as

\[
N = \left( \sum a_i N_i \right) / \left( \sum a_i \right),
\]

with the sum taken over all indices \( i \) of polygons \( P_i \) that include this vertex, where each polygon \( P_i \) has a normal \( N_i \) and has weight \( a_i \) at the vertex in question. Each weight can be calculated as you wish to represent the importance of the polygon in determining the normal to the vertex. A common approach is to let the weight be the angle of the polygon at the vertex. This angle \( a_i \) can be calculated easily as the inverse cosine of the dot product of the two edges of the polygon \( P_i \) that meet at the vertex, as described in the chapter on mathematics for modeling. As suggested there, the angle can be retained with the vertex information for the polygon and used in cases such as this for calculating the per-vertex normal.

**Analytic computations**

In the example of Figure 8.5, an analytic approach to computing the normal \( N \) at each vertex was possible as described in the previous chapter because the surface was defined by \( f(x,y) = 0.3 \cos(x^2 + y^2 + t) \), an easy to use, closed-form equation. In the smooth-shaded example, we were able to calculate the vertex normals by using the analytic directional derivatives at each vertex: the derivatives in the \( x \) and \( y \) directions respectively are

\[
fx(x,y) = \frac{\partial f}{\partial x} = -0.6 \cdot x \cdot \sin(x^2 + y^2 + t) \quad \text{and} \quad fy(x,y) = \frac{\partial f}{\partial y} = -0.6 \cdot y \cdot \sin(x^2 + y^2 + t).
\]

These are used to calculate the tangent vectors in these directions, and the cross products of those tangent vectors were computed to get the vertex normal. This is shown in the code sample at the end of this chapter.

In general, if you have a function \( f \) with partial derivatives \( fx \) and \( fy \), as we have above, the tangent vectors to the surface in the \( x \) and \( y \) directions are \( <1, 0, fx> \) and \( <0, 1, fy> \), respectively. Taking the cross product through determinants as we discussed in the chapter on mathematics for modeling, we see that a normal vector to the surface at a point is \( <-fx, -fy, 1> \). Normalizing this, either through direct computation or by enabling GL_NORMALIZE, completes the calculation of the surface normal.

It can also be possible to get exact normals from other kinds of models; we saw in an early chapter in these notes that the normals to a sphere are simply the radius vectors for the sphere, so a purely geometric model may also have exactly-defined normals. In general, when models permit you to carry out analytic or geometric calculations for normals, these will be more exact and will give you better results than using an interpolation technique.
Other shading models

You cannot and must not assume that the smooth shading model of a simply API such as OpenGL is an accurate representation of smooth surfaces. It assumes that the surface of the polygon varies uniformly, it only includes per-vertex information in calculating colors across the polygon, and it relies on a linear behavior of the RGB color space that is not accurate, as you saw when we talked about colors. Like many of the features of any computer graphics system, it approximates a reality, but there are better ways to achieve the effect of smooth surfaces. For example, there is a shading model called Phong shading that requires the computation of one normal per vertex and uses the interpolated values of the normals themselves to compute the color at each pixel in the polygon, instead of simply interpolating the vertex colors. Interpolating the normals is much more complicated than interpolating colors, because the uniformly-spaced pixels in screen space do not come from uniformly-spaced points in 3D eye space or 3D model space; the perspective projection involves a division by the Z-coordinate of the point in eye space. This makes normal interpolation more complex—and much slower—than color interpolation and takes it out of the range of simple graphics APIs. However, the Phong shading model behaves like a genuinely smooth surface across the polygon, including picking up specular highlights within the polygon and behaving smoothly along the edges of the polygon. The details of how Gouraud and Phong shading operate are discussed in any graphics textbook. We encourage you to read them as an excellent example of the use of interpolation as a basis for many computer graphics processes.

The Phong shading model assumes that normals change smoothly across the polygon, but another shading model is based on controlling the normals across the polygon. Like the texture map that we describe later and that creates effects that change across a surface and are independent of the colors at each vertex, we may create a mapping that alters the normals in the polygon so the shading model can create the effect of a bumpy surface. This is called a bump map, and like Phong shading the normal for each individual pixel is computed separately. Here the pixel normal is computed as the normal from Phong shading plus a normal computed from the bump map by the gradient of the color. The color of each individual pixel is then computed from the standard lighting model. Figure 8.7 shows an example of the effect of a particular bump map. Note that the bump map itself is defined simply a 2D image where the height of each point is defined by the color; this is called a height field. Elevations or distances are sometimes presented in this way, for example in terrain modeling.

![Figure 8.7: a bump map defined as a height field, left, and the bump map applied to a specular surface](image)

The shading models that have been described so far are all based on the simple lighting model of the previous chapter, which assumes that light behavior is uniform in all directions from the surface normal (that is, the lighting is isotropic). However, there are some materials where the lighting parameters differ depending on the angle around the normal. Such materials include examples such as brushed metals or the non-label surface of a CD, and the shading for these materials is called anisotropic because the reflections are different in different directions. Here the
simple role in lighting calculations of the angle from the normal of the diffuse reflection and the angle from the reflected light in the specular reflection, are replaced by a more complex function called the bidirectional reflection distribution function (or BRDF) that depends typically on both the latitude $\Theta$ and longitude $\Phi$ angle of the eye and of the light from the point being lighted: $\rho(\Theta_e, \Phi_e,\Theta_l, \Phi_l)$. The BRDF may also take into account behaviors that differ for different wavelengths (or different colors) of light. The lighting calculations for such materials, then, may involve much more complex kinds of computation than the standard isotropic model and are beyond the scope of simple graphics APIs, but you will find this kind of shading in some professional graphics tools. Figure 8.8 shows the effect on a red sphere of applying flat, smooth, and Phong shading, and an anisotropic shading.

Note that the flat-shaded sphere shows each facet clearly, the smooth-shaded sphere shows some facet edges and the specular reflection is not quite smoothly distributed over the surface, while the facet edges and the specular reflection in the Phong shaded sphere are quite a bit smoother and less broken up.

![Figure 8.8: a sphere presented with flat shading (left), smooth shading (second), Phong shading (third) and an anisotropic shading (right).](image)

**Vertex and pixel shaders**

One of the recent advances in shading that is somewhat ahead of most graphics APIs is providing a shading language with which the programmer can define shading at a much more detailed level than we have discussed so far. By allowing you to define a programmatic way to shade both vertices and individual pixels, it is possible to develop anisotropic shaders, motion blur, bump mapping, and many other very sophisticated kinds of effects. There is ongoing research in making this accessible to the programmer through vertex shading languages that could be part of a graphics API, and in the last chapter we saw how they can fit into the structure of the rendering pipeline. We look forward to a great deal of development in this direction in the near future.

**Lighting, Shading, and Materials in Scene Graphs**

When we introduced scene graphs in the chapter on principles of modeling, we included some concepts that we could not fully implement until now. These included the concept of lights and the idea of including both geometry and appearance information in the terminal nodes of a scene graph. In this section we will expand on these ideas and show how they can be included in the code you will generate from your scene graph design.

Lights are placed in a scene in the same way that geometry is, and in fact one might simply see a light as a piece of geometry that happens to emit light. A light has not only a position but an orientation as well, as we have seen with directional light. Other information must be stored about the light, such as the kind of light; the ambient, diffuse, and specular colors of the light; and, as appropriate, other parameters for the kind of light. These must be included with the light node in the scene graph, and might be considered the appearance part of the light node.
The appearance part of a terminal node must include the kind of information needed for appropriate rendering to be done. If lighting is not to be used, then this information is probably limited to simple color and shading information. The color might be defined per vertex or per polygon, depending on the need of the image to be produced. However, if lighting is to be used, then more comprehensive information is needed, including a full material definition for the geometry and the kind of shading to be used in rendering the object. In case the geometry is described with several different kinds of materials, it may be necessary to break up the geometry node into several smaller geometry nodes, one for each kind of material. This may increase the number of terminal nodes in the scene graph, but does not really increase the complexity of the model very much.

The resulting scene graph will now contain not only the viewpoint, the lights, and the geometry, but will also include all the definitions needed to define the lights and the appearance of the geometry in a way that takes advantage of all the tools offered in this chapter.

It is possible that you may want some parts of your image to be presented with lighting while other parts are presented with directly-specified colors. This can be done by including the presence or absence of lighting as one aspect of the appearance node of your geometry. The graphics API can then be directed to use or not use a lighting model according to this aspect.

**Global illumination**

There is another approach to lighting that takes a more realistic view than the local illumination model. In any actual scene, the light that does not come from direct lights is not simply an ambient value; it is reflected from every object, every surface in the scene. Lighting that accounts for this uses a *global illumination model*, so-called because light is computed globally for the entire scene independent of the viewpoint instead of being computed for each polygon separately in a way that depends on the viewpoint, as we did earlier in this chapter.

Global illumination models include *radiosity*, which assumes that every surface can emit light and calculates the emission in a sequence of steps that converges to an eventual stable light. In the first step, light sources emit their light and any other source is zero. The light energy arriving at each surface is calculated and stored with the surface, and in the next step, that energy is emitted according to properties of the material. This step is repeated until the difference between energies at each point from one step to the next is small, and one assumes that the emission from each point is then known. When the scene is displayed, each point is given a color consistent with the energy emitted at that point. In practice, the scene is not handled as a set of points, however; every object in the scene is subdivided into a number of patches having a simple geometry, and the calculations are done on a per-patch basis.

Another global illumination approach is called *photon mapping*. In this approach the light energy is modeled by emitting photons from each light source and tracing them as they hit various surfaces in the scene. As a photon hits a surface it is accumulated to the surface and, may be emitted from the surface based on the properties of the surface and on a probability calculation. This is done until a sufficient (and surprisingly small) number of photons have been traced through the scene, and then the overall illumination is derived from the accumulated results. You can see that this is quite similar in some ways to the radiosity concept but the computation is quite different.

Because of the way these global illumination processes work, there is no question of shading models with them. Any unique kinds of shading comes out in the way the light energy arriving at a surface is passed on to the next surface.
One of the advantages of global illumination is that once you have computed the light at each point in the scene, displaying the scene can be done very rapidly. This is a good example of an asymmetric process, because you can put in a large amount of processing to create the lighting for the model, but once you have done so, you need not do much processing to display the model. This has a number of interesting and useful applications but is not yet supported by basic graphics APIs—though it may be in the future.

**Local illumination**

In contrast to global illumination models, where the energy reflected from every surface is taken into account, local illumination models assume that light energy comes only from light sources. This is the approach taken by OpenGL, where the light at any point is only that accounted for by the ambient, diffuse, and specular light components described earlier in this chapter. This is handled by defining the properties of lights and of materials relative to these three components, as we describe in this section.

*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, pname, 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, pname, vector_values)`, and you may use whichever form fits your particular design and code best.

As is often the case in OpenGL, there are particular names that you must use for some of these values. Lights must be named `GL_LIGHT0` through `GL_LIGHT7` for standard OpenGL (some implementations may allow more lights, but eight possible lights is standard), and the parameter name `pname` must be one of the available light parameters

- `GL_AMBIENT`
- `GL_DIFFUSE`
- `GL_SPECULAR`
- `GL_POSITION`
- `GL_SPOT_DIRECTION`
- `GL_SPOT_EXPONENT`
- `GL_SPOT_CUTOFF`
- `GL_CONSTANT_ATTENUATION`
- `GL_LINEAR_ATTENUATION`
- `GL_QUADRATIC_ATTENUATION`

In this section we will discuss the properties of OpenGL lights that lead to these parameters.

**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 is whether your scene will use one-sided or two-sided lighting, which is chosen with the function

```c
glLightModel[0|1](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

\[
gltLightModel[f|i](GL\_LIGHT\_MODEL\_LOCAL\_VIEWER, value)
\]

with a value of 0 meaning that the view direction is parallel to the \(Z\)-axis and non-zero that the view direction 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 \(GL\_LIGHT0 \ldots GL\_LIGHT7\), and you create them by defining their properties with the \(glLight\*(\ldots)\) 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 \(GL\_LIGHT0\) by the following position definition and definition of the first of the three lights in the three-light example

\[
\begin{align*}
gllightfv(GL\_LIGHT0, GL\_POSITION, light\_pos0 ); & \quad // \text{ light } 0 \\
gllightfv(GL\_LIGHT0, GL\_AMBIENT, amb\_color0 ); \\
gllightfv(GL\_LIGHT0, GL\_DIFFUSE, diff\_col0 ); \\
gllightfv(GL\_LIGHT0, GL\_SPECULAR, spec\_col0 );
\end{align*}
\]

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.

\[
\begin{align*}
\text{GLfloat light\_pos0 = \{ ..., ..., ... \};} \\
\text{GLfloat diff\_col0 = \{ ..., ..., ... \};}
\end{align*}
\]

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 \(gl\Enable(GL\_NORMALIZE)\) 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

\[
\text{gl\Enable(GL\_LIGHTING); \quad // so lighting models are used}
\]

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 below:

\[
\begin{align*}
gl\Enable(GL\_LIGHT0); & \quad // \text{ use LIGHT0} \\
gl\Enable(GL\_LIGHT1); & \quad // \text{ and LIGHT1} \\
gl\Enable(GL\_LIGHT2); & \quad // \text{ and LIGHT2}
\end{align*}
\]

Lights may also be disabled with the \(gldisable(\ldots)\) 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:

```
gllightModelf(GL_LIGHT_MODEL_AMBIENT, r, g, b, a)
```

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:

```
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° (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 = 1/(A_c + A_l * D + A_q * 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 $A_q$ (think of constant, linear, and quadratic attenuation terms) are 1.0, 0.0, and 0.0 respectively. The actual values of the attenuation constants can be set by the functions

```
glLightf(GL_*_ATENUATION, value)
```

functions, where the wildcard * is to be replaced by one of the three symbolic constants identified 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

```
glLightf(light, GL_POSITION, 10.0, 10.0, 10.0, 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.
OpenGL takes advantage of the two-sided nature of polygons that we described earlier in this chapter, and allows you to specify that for your material you are lighting the front side of each polygon, the back side of each polygon (refer to the earlier discussion for the concept of front and back sides), or both the front and back sides. You do this by specifying your materials with the parameters GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK. 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

\[
glMaterial[i|f][v](\text{face}, \text{parametername}, \text{value})
\]

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.

```c
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 \( P[0], P[1], \text{and } P[2] \), compute its normal, and use the calculated normal, we would see code something like this:

```c
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 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. For example, it can be used to determine whether you’re looking at a front or back face, and it is used in the formulas earlier in the chapter to compute both the diffuse light and the specular light for a polygon.

Code for the example

Defining the light colors and positions in the initialization function:

```c
GLfloat light_pos0[4]={ 0.0, 10.0, 2.0, 1.0 }; //up y-axis
GLfloat light_col0[4]={ 1.0, 0.0, 0.0, 1.0 }; //light is red
GLfloat amb_color0[4]={ 0.3, 0.0, 0.0, 1.0 }; //ambient color
GLfloat light_pos1[4]={ 5.0, -5.0, 2.0, 1.0 }; //lower right
GLfloat light_col1[4]={ 0.0, 1.0, 0.0, 1.0 }; //light is green
GLfloat amb_color1[4]={ 0.0, 0.3, 0.0, 1.0 }; //ambient color
GLfloat light_pos2[4]={-5.0, 5.0, 2.0, 1.0 }; //lower left
GLfloat light_col2[4]={ 0.0, 0.0, 1.0, 1.0 }; //light is blue
GLfloat amb_color2[4]={ 0.0, 0.0, 0.3, 1.0 }; //ambient color
```

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 );
```
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 );

// light 2

// two-sided

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 8.9 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 from the ambient light; to change this you would need to reduce the ambient light
and increase the brightness of the three 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. We will see an example of this when we describe how to create a texture map of shadows by calculating a view from the light point. 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 (anisotropic) 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. This topic is explored in more detail in the chapter on visual communication.

**Shading example**

The two issues in using OpenGL shading are the selection of the shading model and the specification of a color at each vertex, either explicitly with the `glColor*(...)` function or by setting a normal per vertex with the `glNormal*(...)` function. The default shading model for OpenGL is smooth, for example, but you will not get the visual effect of smooth shading unless you specify the appropriate normals for your model, as described below. OpenGL allows you to select the shading model with the `glShadeModel` function, and the only values of its single parameter are the symbolic parameters `GL_SMOOTH` and `GL_FLAT`. You may use the `glShadeModel` function to switch back and forth between smooth and flat shading any time you wish.

In the sample code below, we set up smooth shading by the approach of defining a separate normal vector at each vertex. To begin, we will use the following function call in the `init()` function to ensure that we automatically normalize all our normals in order to avoid having to do this computation ourselves:

```c
glEnable(GL_NORMALIZE);//make unit normals after transforms
```

We use the analytic nature of the surface to generate the normals for each vertex. We compute the partial derivatives \( \frac{\partial f}{\partial x} \) and \( \frac{\partial f}{\partial y} \) for the function in order to get tangent vectors at each vertex:

```c
#define f(x,y)  0.3*cos(x*x+y*y+t) // original function
#define fx(x,y) -0.6*x*sin(x*x+y*y+t)// partial derivative in x
#define fy(x,y) -0.6*y*sin(x*x+y*y+t)// partial derivative in y
```

In the display function, we first compute the values of \( x \) and \( y \) with the functions that compute the grid points in our domain, here called \( XX(i) \) and \( YY(j) \), and then we do the following computation for each triangle in the surface, using an inline cross product operation. We are careful to compute the triangle surface normal as (X–partial cross Y–partial), in that order, so the right-hand rule for cross products gives the correct direction for it. We could have also used the analytic form for the cross product and avoided the calculation below, but the approach here is more general and can be used with we do not have analytic forms available.

```c
glBegin(GL_POLYGON);
    x = XX(i);
    y = YY(j);
    vec1[0] = 1.0;
    vec1[1] = 0.0;
    vec1[2] = fx(x,y); // partial in X-Z plane
```
vec2[0] = 0.0;
vec2[1] = 1.0;
vec2[2] = fy(x, y); // partial in Y-Z plane
glNormal3fv(Normal);
glVertex3f(XX(i), YY(j), vertices[i][j]);
... // do similar code two more times for each vertex of the
... // triangle
glEnd();

This would probably be handled more efficiently by setting up the vec1 and vec2 vectors as above and then calling a utility function to calculate the cross product.

Summary

This chapter has shown you how to create images using the Phong lighting model for vertex color and using simple linear color averaging for creating smoothly-changing color across polygons. This gives you a significant advantage in dealing with the appearance part of an image and lets you make much more interesting and informative images for your audience. From here, the only simple extensions to appearance is the texture mapping that is presented in the next chapter.

Questions

This set of questions covers your recognition of issues in lighting and shading as you see them in your environment. These will help you see the different kinds of light used in the OpenGL simple local lighting model, and will also help you understand some of the limitations of this model.

1. In your environment, identify examples of objects that show only ambient light, that show diffuse light, and that show specular light. Note the relationship of these objects to direct light sources, and draw conclusions about the relationships that give only ambient light, that give both ambient and diffuse light, and that give specular light. Observe the specular light and see whether it has the color of the object or the color of the light source.

2. In your environment, identify objects that show high, moderate, and low specularity. What seems to be the property of the materials of these objects that makes them show specular light?

3. In your environment, find examples of positional lights and directional lights, and discuss how that would affect your choice of these two kinds of lights in a scene.

4. In your environment, select some objects that seem to be made of different materials and identify the ambient, diffuse, and specular properties of each. Try to define each of these materials in terms of the OpenGL material functions.

5. In your environment, find examples where the ambient lighting seems to be different in different places. What are the contributing factors that make this so? What does this say about the accuracy of local lighting models as compared with global lighting models?

Exercises

This set of exercises asks you to calculate some of things that are important in modeling or lighting, and then often to draw conclusions from your calculations. The calculations should be straightforward based on information from the chapter.
6. If you have a triangle whose vertices are \(V_0 = (\ , \ , \), \(V_1 = (\ , \ , \), and \(V_2 = (\ , \ , \), in that sequence, calculate the appropriate edge vectors and then calculate the unit normal to this triangle based on a cross product of the edge vectors. Re-do the calculations if the two vertices \(V_0\) and \(V_1\) are switched in the sequence.

7. If you have a surface given by a function of two variables, \(<\text{fill in function here}>\), calculate the partial derivatives in \(u\) and \(v\), and for the point \(P = (\ , \ , \)\), calculate the two directional tangents and, using the cross product, calculate the unit normal to the surface at \(P\).

8. For some particular surfaces, the surface normal is particularly easy to compute. Show how you can easily calculate the normals for a plane, a sphere, or a cylinder.

9. Going back to question 1 above, switch the values of vertices \(V_0\) and \(V_2\) and do the calculations again. What do you observe about relation between the new unit normal you have just computed and the unit normal you computed originally?

10. We saw in question 6 that the orientation of a polygon defines the direction of a surface normal, and the direction of the surface normal affects the sign of the dot product in the diffuse and specular equations. For the normals of question 6, assume that you have a light at point \(L = (\ , \ , \)\), and calculate the diffuse and specular light components for each of the two normals. Which orientation makes sense, and what does that tell you about the way you define the sequences of vertices for your geometry?

11. In the equation for diffuse light, assume a light has unit energy and calculate the energy reflected at different angles from the normal using the diffuse lighting equation. Use the standard angles of \(\pm 30^\circ\), \(\pm 45^\circ\), and \(\pm 60^\circ\). Similarly, calculate the area of a unit square of surface when it is projected onto a plane at different angles to the surface. Then calculate the ratio of the energy to the projected surface area; this should be a constant. Using your calculations and the discussion of diffuse lighting in the chapter, describe why this is so.

12. Based on the specular light equation, write the equation that would let you find the angle \(\Theta\) at which the energy of the specular light would be 50% of the original light energy. How does this equation vary with the specularity coefficient (the shininess value) \(N\)?

13. In exploring the effect of dropoff for spotlights, define spotlight with a cutoff of \(45^\circ\) and constant attenuation, and calculate the light energy over \(5^\circ\) increments from \(0^\circ\) to \(45^\circ\) with dropoff exponents 1, 2, and 4. What does this tell you about the central and edge energies of spotlights?

14. In exploring the effect of attenuation for lights, consider the OpenGL equation for attenuation: 
\[
A = 1/(A_c + A_L \ast D + A_Q \ast D^2) \]
Define a set of points at distances 1, 2, 3, 4, and 5 units from a light, and calculate the light energy at each of these points using values of 1 for each of the lighting coefficients. Calculate the attenuation coefficient \(A\) in three cases: (a) only constant attenuation, \(A_L = A_Q = 0\), (b) only linear attenuation, \(A_c = A_Q = 0\), and (c) only quadratic attenuation, \(A_L = A_J = 0\). What conclusions do you draw about the effect of linear and quadratic attenuation as you observe positional lights in the world around you?

15. For the merry-go-round scene graph defined in the questions in Chapter 2 on modeling, show how you would place a light in the scene graph if the light were (a) at the center of the merry-go-round, (b) on a post at the outside of the base of the merry-go-round, or (c) on top of the head of a horse on the merry-go-round.
16. True or false: for a model with lighting, flat shading is exactly the same as smooth shading if the smooth-shaded model uses the face normal at each vertex. Why is your answer correct?

17. If a triangle is very small, that is, if the number of pixels in the rendered triangle is small, does it really matter whether you use flat or smooth shading for the triangle? Why or why not?

Experiments

18. We have talked about enabling a number of properties or attributes in the discussion of creating lighted, shaded images. In order to really understand what each does, take a program that uses lighting and shading well and systematically disable (rather than enable) each of the properties or attributes to see what happens. For example, disable GL_NORMALIZE to see what happens when you do not have unit normals. Take notes on the results so you can recognize the problems caused when you do not properly enable a property.

19. Consider the gluCylinder you met earlier; our goal is to create a shaded cylinder whose color varies smoothly from one end to the other. In order to do this, represent the cylinder by a set of quads whose vertices are on the two end circles of the cylinder. Define the colors for the two ends of the shaded cylinder, and set smooth shading for your image. For each quad, use the appropriate color for the vertices on each end, and see whether your cylinder does not have the same kind of shading properties you’d expect for a smooth-shaded quad. Try various numbers of quads in order to get the best possible look with the smallest possible number of quads.

Define a hemisphere from scratch using spherical coordinates or any other modeling techniques from Chapter 2 or Chapter 3. Design the hemisphere so you can easily control the resolution of the figure in all dimensions, just as the GLU and GLUT models let you define slices and stacks. This hemisphere will be the basis of the set of questions below.

20. From the question above in which you were asked to try to define material properties that matched some real-world materials, use these material definitions in the image of the hemisphere and see how closely the image approximates the actual appearance of the object. If the approximation is not very good, see if you can identify why this is so and modify the material definitions to make it better.

21. Because we are working with a hemisphere, it is simple to get analytic normals for each vertex. However, a normal for a face is not the same as the normal for a vertex; show how you can get a normal for each face of the hemisphere.

22. Using routine code for displaying polygonal models and using a relatively coarse definition of the hemisphere, compare the effect of flat shading (using the face normal) and the effect of smooth shading (using vertex normals). Compare the effect of using coarsely modeled smooth shading with using flat shading with much higher resolution.

23. For the display of flat shading, choose a vertex normal for each face instead of the face normal, always choosing the same vertex relative to the face, and compare the view you get each way. Why is the vertex normal view less accurate than the face normal view? Is the difference important?

24. Because the display code allows us to rotate the hemisphere and see the inside as well as the outside, we can consider different light models:
   GL_LIGHT_MODEL_TWO_SIDE
   GL_LIGHT_MODEL_LOCAL_VIEWER
and different face parameters for materials:
  GL_FRONT
  GL_BACK
  GL_FRONT_AND_BACK
Try out all or most of these options, and for each, note the effect of the option on the view.

Projects

25. (The simple house) Returning to the simple house we created as a project for Chapter 3, add a few lights within and outside the house and replace the colored walls with a reasonable material description for walls, and show how the previous views of the house now look with lighting and shading.

26. (A scene graph parser) Add light information in the scene graph parser of Chapter 3 so that each light can be placed with appropriate transformations. Also store the properties of each light with the light node so that appropriate lighting information can be generated by the parser.

27. (A scene graph parser) Add appearance information into the scene graph parser of Chapter 3. The code the parser will generate should now include appropriate material definition functions as well as the previous geometry functions. Also include shading information so that each piece of geometry may independently be smooth or flat shaded. See if you can keep track of current material definitions as you parse the graph so that you can avoid generating new materials definitions if they would be redundant with the ones in place at the time the new definitions are read.