

Chapter 8 Rendering Faces for Visual Realism.

Goals of the Chapter

- To add realism to drawings of 3D scenes.
- To examine ways to determine how light reflects off of surfaces.
- To render polygonal meshes that are bathed in light.
- To see how to make a polygonal mesh object appear smooth.
- To develop a simple hidden surface removal using a depth-buffer.
- To develop methods for adding textures to the surfaces of objects.
- To add shadows of objects to a scene.

Preview.

Section 8.1 motivates the need for enhancing the realism of pictures of 3D objects. Section 8.2 introduces various shading models used in computer graphics, and develops tools for computing the ambient, diffuse, and specular light contributions to an object's color. It also describes how to set up light sources in OpenGL, how to describe the material properties of surfaces, and how the OpenGL graphics pipeline operates when rendering polygonal meshes.

Section 8.3 focuses on rendering objects modeled as polygon meshes. Flat shading, as well as Gouraud and Phong shading, are described. Section 8.4 develops a simple hidden surface removal technique based on a depth buffer. Proper hidden surface removal greatly improves the realism of pictures.

Section 8.5 develops methods for "painting" texture onto the surface of an object, to make it appear to be made of a real material such as brick or wood, or to wrap a label or picture of a friend around it. Procedural textures, which create texture through a routine, are also described. The thorny issue of proper interpolation of texture is developed in detail. Section 8.5.4 presents a complete program that uses OpenGL to add texture to objects. The next sections discuss mapping texture onto curved surfaces, bump mapping, and environment mapping, providing more tools for making a 3D scene appear real.

Section 8.6 describes two techniques for adding shadows to pictures. The chapter finishes with a number of Case Studies that delve deeper into some of these topics, and urges the reader to experiment with them.

8.1. Introduction.

In previous chapters we have developed tools for modeling mesh objects, and for manipulating a camera to view them and make pictures. Now we want to add tools to make these objects and others look visually interesting, or realistic, or both. Some examples in Chapter 5 invoked a number of OpenGL functions to produce shiny teapots and spheres apparently bathed in light, but none of the underlying theory of how this is done was examined. Here we rectify this, and develop the lore of **rendering** a picture of the objects of interest. This is the business of *computing* how each pixel of a picture should look. Much of it is based on a **shading model**, which attempts to model how light that emanates from light sources would interact with objects in a scene. Due to practical limitations one usually doesn't try to simulate all of the physical principles of light scattering and reflection; this is very complicated and would lead to very slow algorithms. But a number of approximate models have been invented that do a good job and produce various levels of realism.

We start by describing a hierarchy of techniques that provide increasing levels of realism, in order to show the basic issues involved. Then we examine how to incorporate each technique in an application, and also how to use OpenGL to do much of the hard work for us.

At the bottom of the hierarchy, offering the lowest level of realism, is a **wire-frame** rendering. Figure 8.1 shows a flurry of 540 cubes as wire-frames. Only the edges of each object are drawn, and you can see right through an object. It can be difficult to see what's what. (A stereo view would help a little.)

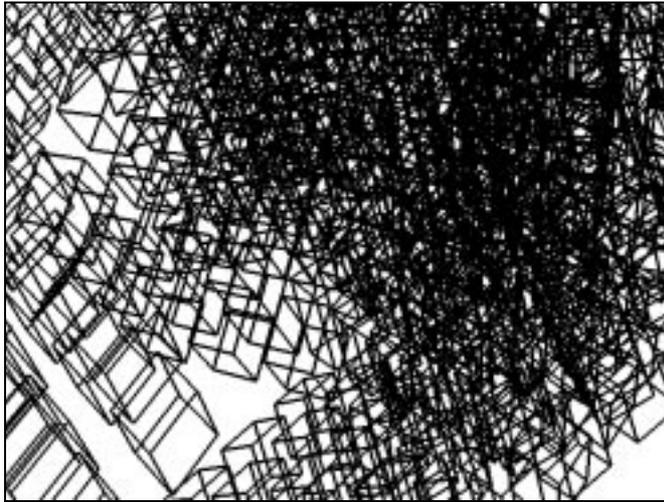


Figure 8.1. A wire-frame rendering of a scene.

Figure 8.2 makes a significant improvement by not drawing any edges that lie behind a face. We can call this a “wire-frame with hidden surface removal” rendering. Even though only edges are drawn the objects now look solid and it is easy to tell where one stops and the next begins. Notice that some edges simply end abruptly as they slip behind a face.

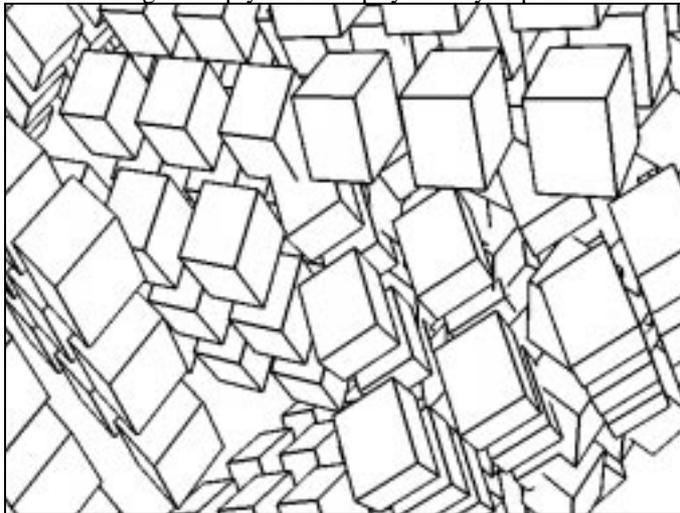


Figure 8.2. Wire-frame view with hidden surfaces removed.

(For the curious: This picture was made using OpenGL with its depth buffer enabled. For each mesh object the faces were drawn in white using `drawMesh()`, and then the edges were drawn in black using `drawEdges()`. Both routines were discussed in Chapter 6.)

The next step in the hierarchy produces pictures where objects appear to be “in a scene”, illuminated by some light sources. Different parts of the object reflect different amounts of light depending on the properties of the surfaces involved, and on the positions of the sources and the camera’s eye. This requires computing the brightness or color of each fragment rather than having the user choose it. The computation requires the use of some shading model that determines the proper amount of light that is reflected from each fragment.

Figure 8.3 shows a scene modeled with polygonal meshes: a buckyball rests atop two cylinders, and the column rests on a floor. Part a shows the wire-frame version, and part b shows a shaded version (with hidden surfaces removed). Those faces aimed toward the light source appear brighter than those aimed away from the source. This picture shows **flat shading**: a calculation of how much light is scattered from each face is computed at a single point, so all points on a face are rendered with the same gray level.



Figure 8.3. A mesh approximation shaded with a shading model. a). wire-frame view b). flat shading,

The next step up is of course to use color. Plate 22 shows the same scene where the objects are given different colors.

In Chapter 6 we discussed building a mesh approximation to a smoothly curved object. A picture of such an object ought to reflect this smoothness, showing the smooth “underlying surface” rather than the individual polygons. Figure 8.4 show the scene rendered using **smooth shading**. (Plate 23 shows the colored version.) Here different points of a face are drawn with different gray levels found through an interpolation scheme known as **Gouraud shading**. The variation in gray levels is much smoother, and the edges of polygons disappear, giving the impression of a smooth, rather than a faceted, surface. We examine Gouraud shading in Section 8.3.



Figure 8.4. The scene rendered with smooth shading.

Highlights can be added to make objects look shiny. Figure 8.5 show the scene with **specular light** components added. (Plate 24 shows the colored version.) The shinier an object is, the more localized are its specular highlights, which often make an object appear to be made from plastic.



Figure 8.5. Adding specular highlights.

Another effect that improves the realism of a picture is shadowing. Figure 8.6 show the scene above with shadows properly rendered, where one object casts a shadow onto a neighboring object. We discuss how to do this in Section 8.6.

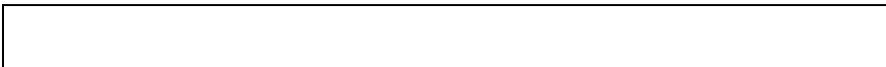


Figure 8.6. The scene rendered with shadows.

Adding texture to an object can produce a big step in realism. Figure 8.7 (and Plate 25) show the scene with different textures “painted” on each surface. These textures can make the various surfaces appear to made of some material such as wood, marble, or copper. And images can be “wrapped around” an object like a decal.



Figure 8.7. Mapping textures onto surfaces.

There are additional techniques that improve realism. In Chapter 14 we study ray tracing in depth. Although raytracing is a computationally expensive approach, it is easy to program, and produces pictures that show proper shadows, mirror-like reflections, and the passage of light through transparent objects.

In this chapter we describe a number of methods for rendering scenes with greater realism. We first look at the classical lighting models used in computer graphics that make an object appear bathed in light from some light sources, and see how to draw a polygonal mesh so that it appears to have a smoothly curved surface. We then examine a particular hidden surface removal method - the one that OpenGL uses - and see how it is incorporated into the rendering process. (Chapter 13 examines a number of other hidden surface removal methods.) We then examine techniques for drawing shadows that one object casts upon another, and for adding texture to each surface to make it appear to be made of some particular material, or to have some image painted on it. We also examine chrome mapping and environment mapping to see how to make a local scene appear to be imbedded in a more global scene.

8.2. Introduction to Shading Models.

The mechanism of light reflection from an actual surface is very complicated, and it depends on many factors. Some of these are geometric, such as the relative directions of the light source, the

observer's eye, and the normal to the surface. Others are related to the characteristics of the surface, such as its roughness, and color of the surface.

A shading model dictates how light is scattered or reflected from a surface. We shall examine some simple shading models here, focusing on achromatic light. **Achromatic** light has brightness but no color; it is only a shade of gray. Hence it is described by a single value: intensity. We shall see how to calculate the intensity of the light reaching the eye of the camera from each portion of the object. We then extend the ideas to include colored lights and colored objects. The computations are almost identical to those for achromatic light, except that separate intensities of red, green, and blue components are calculated.

A shading model frequently used in graphics supposes that two types of light sources illuminate the objects in a scene: point light sources and **ambient** light. These light sources “shine” on the various surfaces of the objects, and the incident light interacts with the surface in three different ways:

- Some is absorbed by the surface and is converted to heat;
- Some is reflected from the surface;
- Some is transmitted into the interior of the object, as in the case of a piece of glass.

If all incident light is absorbed, the object appears black and is known as a **black body**. If all is transmitted, the object is visible only through the effects of refraction, which we shall discuss in Chapter 14.

Here we focus on the part of the light that is reflected or scattered from the surface. Some amount of this reflected light travels in just the right direction to reach the eye, causing the object to be seen. The fraction that travels to the eye is highly dependent on the geometry of the situation. We assume that there are two types of reflection of incident light: diffuse scattering and specular reflection.

- **Diffuse scattering**, occurs when some of the incident light slightly penetrates the surface and is re-radiated uniformly in all directions. Scattered light interacts strongly with the surface, and so its color is usually affected by the nature of the surface material.
- **Specular reflections** are more mirror-like and are highly directional: Incident light does not penetrate the object but instead is reflected directly from its outer surface. This gives rise to highlights and makes the surface look shiny. In the simplest model for specular light the reflected light has the same color as the incident light. This tends to make the material look like plastic. In a more complex model the color of the specular light varies over the highlight, providing a better approximation to the shininess of metal surfaces. We discuss both models for specular reflections.

Most surfaces produce some combination of the two types of reflection, depending on surface characteristics such as roughness and type of material. We say that the total light reflected from the surface in a certain direction is the sum of the diffuse component and the specular component. For each surface point of interest we compute the size of each component that reaches the eye. Algorithms are developed next that accomplish this.

8.2.1. Geometric Ingredients for Finding Reflected Light.

*On the outside grows the furside, on the inside grows the skinside;
So the furside is the outside, and the skinside is the inside.*
Herbert George Ponting, The Sleeping Bag

We need to find three vectors in order to compute the diffuse and specular components. Figure 8.8 shows the three principal vectors required to find the amount of light that reaches the eye from a point P .



Figure 8.8. Important directions in computing reflected light.

1. The normal vector, \mathbf{m} , to the surface at P .

2. The vector \mathbf{v} from P to the viewer's eye.
3. The vector \mathbf{s} from P to the light source.

The angles between these three vectors form the basis for computing light intensities. These angles are normally calculated using world coordinates, because some transformations (such as the perspective transformation) do not preserve angles.

Each face of a mesh object has two sides. If the object is solid one is usually the “inside” and one is the “outside”. The eye can then see only the outside (unless the eye is inside the object!), and it is this side for which we must compute light contributions. But for some objects, such as the open box of Figure 8.9, the eye might be able to see the inside of the lid. It depends on the angle between the normal to that side, \mathbf{m}_2 , and the vector to the eye, \mathbf{v} . If the angle is less than 90° this side is visible. Since the cosine of that angle is proportional to the dot product $\mathbf{v} \cdot \mathbf{m}_2$, the eye can see this side only if $\mathbf{v} \cdot \mathbf{m}_2 > 0$.



Figure 8.9. Light computations are made for one side of each face.

We shall develop the shading model for a given side of a face. If that side of the face is “turned away” from the eye there is normally no light contribution. In an actual application the rendering algorithm must be told whether to compute light contributions from one side or both sides of a given face. We shall see that OpenGL supports this.

8.2.2. Computing the Diffuse Component.

Suppose that light falls from a point source onto one side of a facet (a small piece of a surface). A fraction of it is re-radiated diffusely in all directions from this side. Some fraction of the re-radiated part reaches the eye, with intensity w denoted by I_d . How does I_d depend on the directions \mathbf{m} , \mathbf{v} , and \mathbf{s} ?

Because the scattering is uniform in all directions, the orientation of the facet, F , relative to the eye is not significant. Therefore, I_d is independent of the angle between \mathbf{m} and \mathbf{v} (unless $\mathbf{v} \cdot \mathbf{m} < 0$, whereupon I_d is zero.) On the other hand, the amount of light that illuminates the facet *does* depend on the orientation of the facet relative to the point source: It is proportional to the area of the facet that it sees, that is, the area subtended by a facet.

Figure 8.10a shows in cross section a point source illuminating a facet S when \mathbf{m} is aligned with \mathbf{s} . In Figure 8.10b the facet is turned partially away from the light source through angle θ . The area subtended is now only $\cos(\theta)$ as much as before, so that the brightness of S is reduced by this same factor. This relationship between brightness and surface orientation is often called **Lambert's law**. Notice that for θ near 0 , brightness varies only slightly with angle, because the cosine changes slowly there. As θ approaches 90° , however, the brightness falls rapidly to 0 .

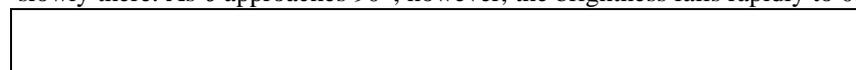


Figure 8.10. The brightness depends on the area subtended.

Now we know that $\cos(\theta)$ is the dot product between normalized versions of \mathbf{s} and \mathbf{m} . We can therefore adopt as the strength of the diffuse component:

$$I_d = I_s \rho_d \frac{\mathbf{s} \cdot \mathbf{m}}{|\mathbf{s}| |\mathbf{m}|}$$

In this equation, I_s is the intensity of the light source, and ρ_d is the **diffuse reflection coefficient**. Note that if the facet is aimed away from the eye this dot product is negative and we want I_d to evaluate to 0 . So a more precise computation of the diffuse component is:

$$I_d = I_s \rho_d \max\left(\frac{\mathbf{s} \cdot \mathbf{m}}{|\mathbf{s}| |\mathbf{m}|}, 0\right) \quad (8.1)$$

This max term might be implemented in code (using the `Vector3` methods `dot()` and `length()` - see Appendix 3) by:

```
double tmp = s.dot(m); // form the dot product
double value = (tmp<0) ? 0 : tmp/(s.length() * m.length());
```

Figure 8.11 shows how a sphere appears when it reflects diffuse light, for six reflection coefficients: 0, 0.2, 0.4, 0.6, 0.8, and 1. In each case the source intensity is 1.0 and the background intensity is set to 0.4. Note that the sphere is totally black when ρ_d is 0.0, and the shadow in its bottom half (where the dot product above is negative) is also black.

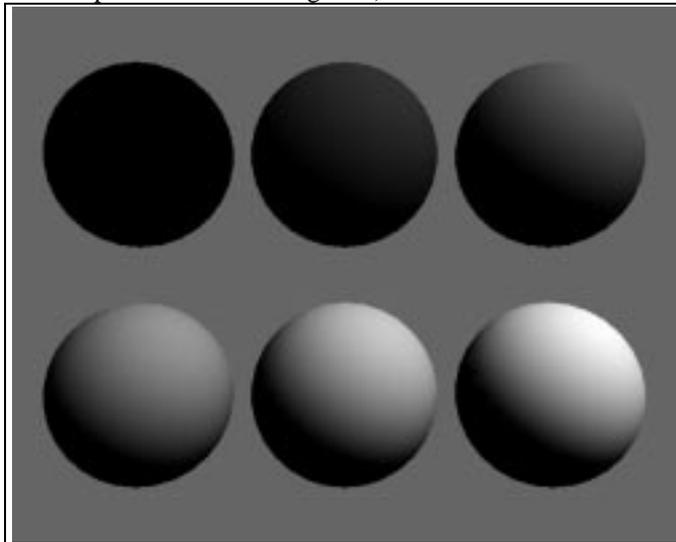


Figure 8.11. Spheres with various reflection coefficients shaded with diffuse light. (file: fig8.11.bmp)

In reality the mechanism behind diffuse reflection is much more complex than the simple model we have adopted here. The reflection coefficient ρ_d depends on the wavelength (color) of the incident light, the angle θ , and various physical properties of the surface. But for simplicity and to reduce computation time, these effects are usually suppressed when rendering images. A “reasonable” value for ρ_d is chosen for each surface, sometimes by trial and error according to the realism observed in the resulting image.

In some shading models the effect of distance is also included, although it is somewhat controversial. The light intensity falling on facet S in Figure 8.10 from the point source is known to fall off as the inverse square of the distance between S and the source. But experiments have shown that using this law yields pictures with exaggerated depth effects. (What is more, it is sometimes convenient to model light sources as if they lie “at infinity”. Using an inverse square law in such a case would quench the light entirely!) The problem is thought to be in the model: We model light sources as point sources for simplicity, but most scenes are actually illuminated by additional reflections from the surroundings, which are difficult to model. (These effects are lumped together into an ambient light component.) It is not surprising, therefore, that strict adherence to a physical law based on an unrealistic model can lead to unrealistic results.

The realism of most pictures is enhanced rather little by the introduction of a distance term. Some approaches force the intensity to be inversely proportional to the distance between the eye and the object, but this is not based on physical principles. It is interesting to experiment with such effects, and OpenGL provides some control over this effect, as we see in Section 8.2.9, but we don't include a distance term in the following development.

8.2.3. Specular Reflection.

Real objects do not scatter light uniformly in all directions, and so a specular component is added to the shading model. Specular reflection causes highlights, which can add significantly to the realism of a picture when objects are shiny. In this section we discuss a simple model for the behavior of specular light due to Phong [Phong 1975]. It is easy to apply and OpenGL supports a good approximation to it. Highlights generated by Phong specular light give an object a “plastic-like”

appearance, so the Phong model is good when you intend the object to be made of shiny plastic or glass. The Phong model is less successful with objects that are supposed to have a shiny metallic surface, although you can roughly approximate them with OpenGL by careful choices of certain color parameters, as we shall see. More advanced models of specular light have been developed that do a better job of modeling shiny metals. These are not supported directly by OpenGL's rendering process, so we defer a detailed discussion of them to Chapter 14 on ray tracing.

Figure 8.12a shows a situation where light from a source impinges on a surface and is reflected in different directions. In the **Phong model** we discuss here, the amount of light reflected is greatest in the direction of perfect mirror reflection, \mathbf{r} , where the angle of incidence θ equals the angle of reflection. This is the direction in which all light would travel if the surface were a perfect mirror. At other near-by angles the amount of light reflected diminishes rapidly, as indicated by the relative lengths of the reflected vectors. Part b shows this in terms of a "beam pattern" familiar in radar circles. The distance from P to the beam envelope shows the relative strength of the light scattered in that direction.

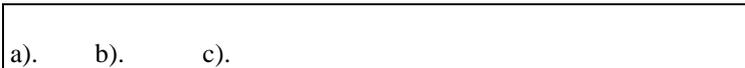


Figure 8.12. Specular reflection from a shiny surface.

Part c shows how to quantify this beam pattern effect. We know from Chapter 5 that the direction \mathbf{r} of perfect reflection depends on both \mathbf{s} and the normal vector \mathbf{m} to the surface, according to:

$$\mathbf{r} = -\mathbf{s} + 2 \frac{(\mathbf{s} \cdot \mathbf{m})}{|\mathbf{m}|^2} \mathbf{m} \quad (\text{the mirror-reflection direction}) \quad (8.2)$$

For surfaces that are shiny but not true mirrors, the amount of light reflected falls off as the angle ϕ between \mathbf{r} and \mathbf{v} increases. The actual amount of falloff is a complicated function of ϕ , but in the Phong model it is said to vary as some power f of the cosine of ϕ , that is, according to $(\cos(\phi))^f$, in which f is chosen experimentally and usually lies between 1 and 200.

Figure 8.13 shows how this intensity function varies with ϕ for different values of f . As f increases, the reflection becomes more mirror-like and is more highly concentrated along the direction \mathbf{r} . A perfect mirror could be modeled using $f = \infty$, but pure reflections are usually handled in a different manner, as described in Chapter 14.



Figure 8.13. Falloff of specular light with angle.

Using the equivalence of $\cos(\phi)$ and the dot product between \mathbf{r} and \mathbf{v} (after they are normalized), the contribution I_{sp} due to specular reflection is modeled by

$$I_{sp} = I_s \rho_s \left(\frac{\mathbf{r} \cdot \mathbf{v}}{|\mathbf{r}| |\mathbf{v}|} \right)^f \quad (8.3)$$

where the new term ρ_s is the **specular reflection coefficient**. Like most other coefficients in the shading model, it is usually determined experimentally. (As with the diffuse term, if the dot product $\mathbf{r} \cdot \mathbf{v}$ is found to be negative, I_{sp} is set to zero.)

A boost in efficiency using the "halfway vector". It can be expensive to compute the specular term in Equation 8.3, since it requires first finding vector \mathbf{r} and normalizing it. In practice an alternate term, apparently first described by Blinn [blinn77], is used to speed up computation. Instead of using the cosine of the angle between \mathbf{r} and \mathbf{v} , one finds a vector halfway between \mathbf{s} and \mathbf{v} , that is, $\mathbf{h} = \mathbf{s} + \mathbf{v}$, as suggested in Figure 8.14. If the normal to the surface were oriented along \mathbf{h} the viewer would see the brightest specular highlight. Therefore the angle β between \mathbf{m} and \mathbf{h} can be used to measure the falloff of specular intensity that the viewer sees. The angle β is not the same as ϕ (in fact β is twice ϕ if the various vectors are coplanar - see the exercises), but this difference can be compensated for by using a different value of the exponent f . (The specular term is not based on physical principles anyway, so it is at least plausible that our adjustment to it yields acceptable

results.) Thus it is common practice to base the specular term on $\cos(\beta)$ using the dot product of \mathbf{h} and \mathbf{m} :



Figure 8.14. The halfway vector.

$$I_{sp} = I_s \rho_s \max\left(0, \left(\frac{\mathbf{h} \cdot \mathbf{m}}{|\mathbf{h}| |\mathbf{m}|}\right)^f\right) \quad \{\text{adjusted specular term}\} \quad (8.4)$$

Note that with this adjustment the reflection vector \mathbf{r} need not be found, saving computation time. In addition, if both the light source and viewer are very remote then \mathbf{s} and \mathbf{v} are constant over the different faces of an object, so \mathbf{h} need only be computed once.

Figure 8.15 shows a sphere reflecting different amounts of specular light. The reflection coefficient ρ_s varies from top to bottom with values 0.25, 0.5, and 0.75, and the exponent f varies from left to right with values 3, 6, 9, 25, and 200. (The ambient and diffuse reflection coefficients are 0.1 and 0.4 for all spheres.)

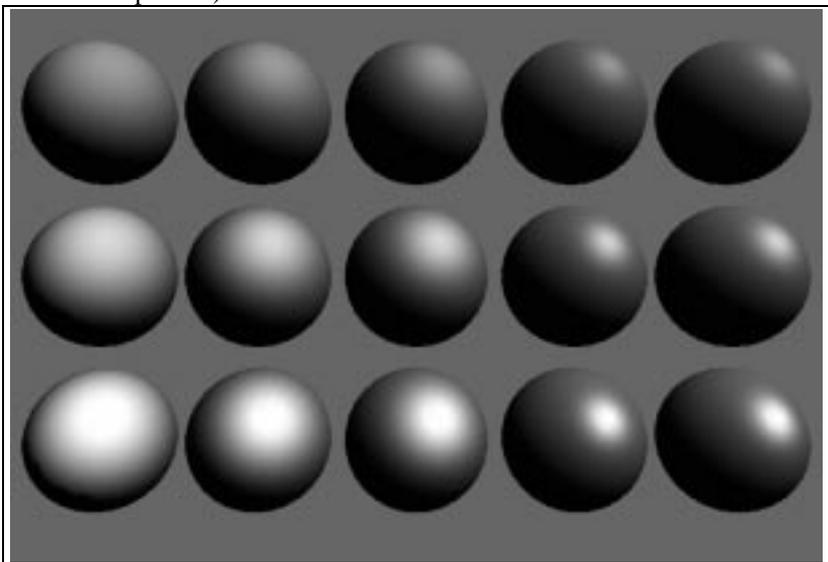


Figure 8.15. Specular reflection from a shiny surface.

The physical mechanism for specularly reflected light is actually much more complicated than the Phong model suggests. A more realistic model makes the specular reflection coefficient dependent on both the wavelength λ (i.e. the color) of the incident light and the angle of incidence θ_i , (the angle between vectors \mathbf{s} and \mathbf{m} in Figure 8.10) and couples it to a “Fresnel term” that describes the physical characteristics of how light reflects off certain classes of surface materials. As mentioned, OpenGL is not organized to include these effects, so we defer further discussion of them until Chapter 14 on ray tracing, where we compute colors on a point by point basis, applying a shading model directly.

Practice Exercises.

8.2.1. Drawing Beam Patterns. Draw beam patterns similar to that in Figure 8.12 for the cases $f=1$, $f=10$, and $f=100$.

8.2.2. On the halfway vector. By examining the geometry displayed in Figure 8.14, show that $\beta = 2\phi$ if the vectors involved are coplanar. Show that this is not so if the vectors are non coplanar. See also [fisher94]

8.2.3. A specular speed up. Schlick [schlick94] has suggested an alternative to the exponentiation required when computing the specular term. Let D denote the dot product $\mathbf{r} \cdot \mathbf{v}/|\mathbf{r}||\mathbf{v}|$ in Equation 8.3. Schlick suggests replacing D^f with $\frac{D}{f - (f-1)D}$, which is faster to compute. Plot these two functions for values of D in $[0,1]$ for various values of f and compare them. Pay particular attention to values of D near 1, since this is where specular highlights are brightest.

8.2.4. The Role of Ambient Light.

The diffuse and specular components of reflected light are found by simplifying the “rules” by which physical light reflects from physical surfaces. The dependence of these components on the relative positions of the eye, model, and light sources greatly improves the realism of a picture over renderings that simply fill a wireframe with a shade.

But our desire for a simple reflection model leaves us with far from perfect renderings of a scene. As an example, shadows are seen to be unrealistically deep and harsh. To soften these shadows, we can add a third light component called “ambient light.”

With only diffuse and specular reflections, any parts of a surface that are shadowed from the point source receive no light and so are drawn black! But this is not our everyday experience. The scenes we observe around us always seem to be bathed in some soft non-directional light. This light arrives by multiple reflections from various objects in the surroundings and from light sources that populate the environment, such as light coming through a window, fluorescent lamps, and the like. But it would be computationally very expensive to model this kind of light precisely.

Ambient Sources and Ambient Reflections

To overcome the problem of totally dark shadows, we imagine that a uniform “background glow” called **ambient light** exists in the environment. This ambient light source is not situated at any particular place, and it spreads in all directions uniformly. The source is assigned an intensity, I_a . Each face in the model is assigned a value for its **ambient reflection coefficient**, ρ_a (often this is the same as the diffuse reflection coefficient, ρ_d), and the term $I_a\rho_a$ is simply added to whatever diffuse and specular light is reaching the eye from each point P on that face. I_a and ρ_a are usually arrived at experimentally, by trying various values and seeing what looks best. Too little ambient light makes shadows appear too deep and harsh; too much makes the picture look washed out and bland.

Figure 8.16 shows the effect of adding various amounts of ambient light to the diffuse light reflected by a sphere. In each case both the diffuse and ambient sources have intensity 1.0, and the diffuse reflection coefficient is 0.4. Moving from left to right the ambient reflection coefficient takes on values 0.0, 0.1, 0.3, 0.5, and 0.7. With only a modest amount of ambient light the harsh shadows on the underside of the sphere are softened and look more realistic. Too much ambient reflection, on the other hand, suppresses the shadows excessively.

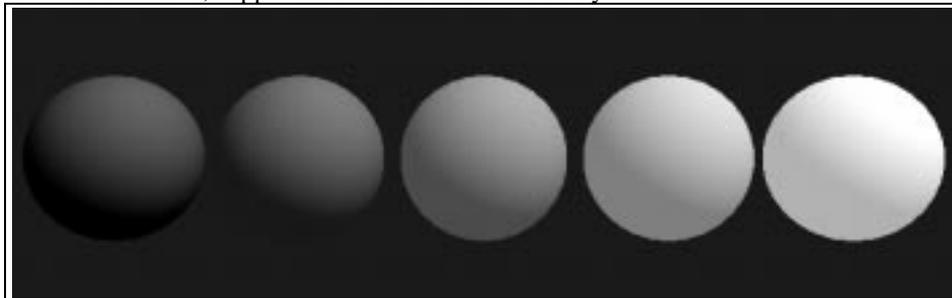


Figure 8.16. On the effect of ambient light.

8.2.5. Combining Light Contributions.

We can now sum the three light contributions - diffuse, specular, and ambient - to form the total amount of light I that reaches the eye from point P :

$$I = I_a\rho_a + I_d\rho_d \times \text{lambert} + I_{sp}\rho_s \times \text{phong}^f \quad (8.5)$$

where we define the values

$$\text{lambert} = \max\left(0, \frac{\mathbf{s} \cdot \mathbf{m}}{|\mathbf{s}||\mathbf{m}|}\right), \text{ and } \text{phong} = \max\left(0, \frac{\mathbf{h} \cdot \mathbf{m}}{|\mathbf{h}||\mathbf{m}|}\right) \quad (8.6)$$

I depends on the various source intensities and reflection coefficients, as well as on the relative positions of the point P , the eye, and the point light source. Here we have given different names, I_d

and I_{sp} , to the intensities of the diffuse and specular components of the light source, because OpenGL allows you to set them individually, as we see later. In practice they usually have the same value.

To gain some insight into the variation of I with the position of P , consider again Figure 8.10. I is computed for different points P on the facet shown. The ambient component shows no variation over the facet; \mathbf{m} is the same for all P on the facet, but the directions of both \mathbf{s} and \mathbf{v} depend on P . (For instance, $\mathbf{s} = S - P$ where S is the location of the light source. How does \mathbf{v} depend on P and the eye?) If the light source is fairly far away (the typical case), \mathbf{s} will change only slightly as P changes, so that the diffuse component will change only slightly for different points P . This is especially true when \mathbf{s} and \mathbf{m} are nearly aligned, as the value of $\cos()$ changes slowly for small angles. For remote light sources, the variation in the direction of the halfway vector \mathbf{h} is also slight as P varies. On the other hand, if the light source is close to the facet, there can be substantial changes in \mathbf{s} and \mathbf{h} as P varies. Then the specular term can change significantly over the facet, and the bright highlight can be confined to a small portion of the facet. This effect is increased when the eye is also close to the facet -causing large changes in the direction of \mathbf{v} - and when the exponent f is very large.

Practice Exercise 8.2.4. Effect of the Eye Distance. Describe how much the various light contributions change as P varies over a facet when a). the eye is far away from the facet and b). when the eye is near the facet.

8.2.6. Adding Color.

It is straightforward to extend this shading model to the case of colored light reflecting from colored surfaces. Again it is an approximation born from simplicity, but it offers reasonable results and is serviceable.

Chapter 12 provides more detail and background on the nature of color, but as we have seen previously colored light can be constructed by adding certain amounts of red, green, and blue light. When dealing with colored sources and surfaces we calculate each color component individually, and simply add them to form the final color of reflected light. So Equation 8.5 is applied three times:

$$\begin{aligned} I_r &= I_{ar}\rho_{ar} + I_{dr}\rho_{dr} \times \text{lambert} + I_{spr}\rho_{sr} \times \text{phong}^f \\ I_g &= I_{ag}\rho_{ag} + I_{dg}\rho_{dg} \times \text{lambert} + I_{spg}\rho_{sg} \times \text{phong}^f \\ I_b &= I_{ab}\rho_{ab} + I_{db}\rho_{db} \times \text{lambert} + I_{spb}\rho_{sb} \times \text{phong}^f \end{aligned} \quad (8.7)$$

(where *lambert* and *phong* are given in Equation 8.6) to compute the red, green, and blue components of reflected light. Note that we say the light sources have three “types” of color: ambient = (I_{ar}, I_{ag}, I_{ab}) , diffuse = (I_{dr}, I_{dg}, I_{db}) , and specular = $(I_{spr}, I_{spg}, I_{spb})$. Usually the diffuse and specular light colors are the same. Note also that the *lambert* and *phong* terms do not depend on which color component is being computed, so they need only be computed once. To pursue this approach we need to define nine reflection coefficients:

ambient reflection coefficients: $\rho_{ar}, \rho_{ag},$ and ρ_{ab} ,
diffuse reflection coefficients: $\rho_{dr}, \rho_{dg},$ and ρ_{db}
specular reflection coefficients: $\rho_{sr}, \rho_{sg},$ and ρ_{sb}

The ambient and diffuse reflection coefficients are based on the color of the surface itself. By “color” of a surface we mean the color that is reflected from it when the illumination is *white* light: a surface is red if it appears red when bathed in white light. If bathed in some other color it can exhibit an entirely different color. The following examples illustrate this.

Example 8.2.1. The color of an object. If we say that the color of a sphere is 30% red, 45% green, and 25% blue, it makes sense to set its ambient and diffuse reflection coefficients to $(0.3K, 0.45K, 0.25K)$, where K is some scaling value that determines the overall fraction of incident light that is reflected from the sphere. Now if it is bathed in white light having equal amounts of red, green, and blue ($I_{sr} = I_{sg} = I_{sb} = I$) the individual diffuse components have intensities $I_r = 0.3 K I, I_g = 0.45 K I, I_b = 0.25 K I$, so as expected we see a color that is 30% red, 45% green, and 25% blue.

Example 8.2.2. A reddish object bathed in greenish light. Suppose a sphere has ambient and diffuse reflection coefficients (0.8, 0.2, 0.1), so it appears mostly red when bathed in white light. We illuminate it with a greenish light $I_s = (0.15, 0.7, 0.15)$. The reflected light is then given by (0.12, 0.14, 0.015), which is a fairly even mix of red and green, and would appear yellowish (as we discuss further in Chapter 12).

The color of specular light. Because specular light is mirror-like, the color of the specular component is often the same as that of the light source. For instance, it is a matter of experience that the specular highlight seen on a glossy red apple when illuminated by a yellow light is yellow rather than red. This is also observed for shiny objects made of plastic-like material. To create specular highlights for a plastic surface the specular reflection coefficients, ρ_{sr} , ρ_{sg} , and ρ_{sb} used in Equation 8.7 are set to the same value, say ρ_s , so that the reflection coefficients are 'gray' in nature and do not alter the color of the incident light. The designer might choose $\rho_s = 0.5$ for a slightly shiny plastic surface, or $\rho_s = 0.9$ for a highly shiny surface.

Objects made of different materials.

A careful selection of reflection coefficients can make an object appear to be made of a specific material such as copper, gold, or pewter, at least approximately. McReynolds and Blythe [mcReynolds97] have suggested using the reflection coefficients given in Figure 8.17. Plate ??? shows several spheres modelled using these coefficients. The spheres do appear to be made of different materials. Note that the specular reflection coefficients have different red, green, and blue components, so the color of specular light is not simply that of the incident light. But McReynolds and Blythe caution users that, because OpenGL's shading algorithm incorporates a Phong specular component, the visual effects are not completely realistic. We shall revisit the issue in Chapter 14 and describe the more realistic Cook-Torrance shading approach..

Material	ambient: $\rho_{ar}, \rho_{ag}, \rho_{ab}$	diffuse: $\rho_{dr}, \rho_{dg}, \rho_{db}$	specular: $\rho_{sr}, \rho_{sg}, \rho_{sb}$	exponent: f
Black	0.0	0.01	0.50	32
Plastic	0.0	0.01	0.50	
	0.0	0.01	0.50	
Brass	0.329412 0.223529 0.027451	0.780392 0.568627 0.113725	0.992157 0.941176 0.807843	27.8974
Bronze	0.2125 0.1275 0.054	0.714 0.4284 0.18144	0.393548 0.271906 0.166721	25.6
Chrome	0.25 0.25 0.25	0.4 0.4 0.4	0.774597 0.774597 0.774597	76.8
Copper	0.19125 0.0735 0.0225	0.7038 0.27048 0.0828	0.256777 0.137622 0.086014	12.8
Gold	0.24725 0.1995 0.0745	0.75164 0.60648 0.22648	0.628281 0.555802 0.366065	51.2
Pewter	0.10588 0.058824 0.113725	0.427451 0.470588 0.541176	0.3333 0.3333 0.521569	9.84615
Silver	0.19225 0.19225 0.19225	0.50754 0.50754 0.50754	0.508273 0.508273 0.508273	51.2
Polished Silver	0.23125 0.23125 0.23125	0.2775 0.2775 0.2775	0.773911 0.773911 0.773911	89.6

Figure 8.17. Parameters for common materials [mcReynolds97].

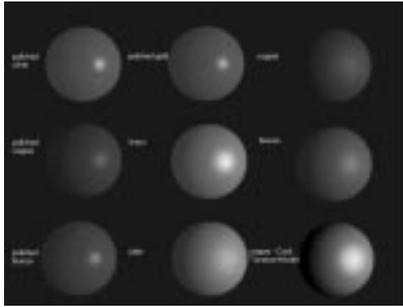


Plate 26. Shiny spheres made of different materials.

8.2.7. Shading and the Graphics pipeline.

At which step in the graphics pipeline is shading performed? And how is it done? Figure 8.18 shows the pipeline again. The key idea is that the vertices of a mesh are sent down the pipeline along with their associated vertex normals, and all shading calculations are done on *vertices*. (Recall that the `draw()` method in the `Mesh` class sends a vertex normal along with each vertex, as in Figure 6.15.)



Figure 8.18. The graphics pipeline revisited.

The figure shows a triangle with vertices v_0 , v_1 , and v_2 being rendered. Vertex v_1 has the normal vector \mathbf{m}_1 associated with it. These quantities are sent down the pipeline with calls such as:

```
glBegin(GL_POLYGON);
    for(int i = 0; i < 3; i++)
    {
        glNormal3f(norm[i].x, norm[i].y, norm[i].z);
        glVertex3f(pt[i].x, pt[i].y, pt[i].z);
    }
glEnd();
```

The Call to `glNormal3f()` sets the “current normal vector”, which is applied to all vertices subsequently sent using `glVertex3f()`. It remains current until changed with another call to `glNormal3f()`. For this code example a new normal is associated with each vertex.

The vertices are transformed by the modelview matrix, M , effectively expressing them in camera (eye) coordinates. The normal vectors are also transformed, but vectors transform differently from points. As shown in Section 6.5.3, transforming points of a surface by a matrix M causes the normal \mathbf{m} at any point to become the normal $M^T\mathbf{m}$ on the transformed surface, where M^T is the transpose of the inverse of M . OpenGL automatically performs this calculation on normal vectors.

As we discuss in the next section OpenGL allows you to specify various light sources and their locations. Lights are objects too, and the light source positions are also transformed by the modelview matrix.

So all quantities end up after the modelview transformation being expressed in camera coordinates. At this point the shading model of Equation 8.7 is applied, and a color is “attached” to each vertex. The computation of this color requires knowledge of vectors \mathbf{m} , \mathbf{s} , and \mathbf{v} , but these are all available at this point in the pipeline. (Convince yourself of this).

Progressing farther down the pipeline, the pseudodepth term is created and the vertices are passed through the perspective transformation. The color information tags along with each vertex. The clipping step is performed in homogeneous coordinates as described earlier. This may alter some of the vertices. Figure 8.19 shows the case where vertex v_1 of the triangle is clipped off, and two new vertices, a and b , are created. The triangle becomes a quadrilateral. The color at each of the new vertices must be computed, since it is needed in the actual rendering step.

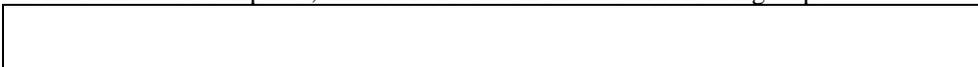


Figure 8.19. Clipping a polygon against the (warped) view volume.

The color at each new vertex is usually found by interpolation. For instance, suppose that the color at v_0 is (r_0, g_0, b_0) and the color at v_1 is (r_1, g_1, b_1) . If the point a is 40% of the way from v_0 to v_1 the color associated with a is a blend of 60% of (r_0, g_0, b_0) and 40% of (r_1, g_1, b_1) . This is expressed as

$$\text{color at point } a = (\text{lerp}(r_0, r_1, 0.4), \text{lerp}(g_0, g_1, 0.4), \text{lerp}(b_0, b_1, 0.4)) \quad (8.8)$$

where we use the convenient function *lerp()* (short for “linear interpolation” - recall “tweening” in Section 4.5.3) defined by:

$$\text{lerp}(G, H, f) = G + (H - G)f \quad (8.9)$$

Its value lies at fraction f of the way from G to H .¹

The vertices are finally passed through the viewport transformation where they are mapped into screen coordinates (along with pseudodepth which now varies between 0 and 1). The quadrilateral is then rendered (with hidden surface removal), as suggested in Figure 8.19. We shall say much more about the actual rendering step.

8.2.8. Using Light Sources in OpenGL.

OpenGL provides a number of functions for setting up and using light sources, as well as for specifying the surface properties of materials. It can be daunting to absorb all of the many possible variations and details, so we describe the basics here. In this section we discuss how to establish different kinds of light sources in a scene. In the next section we look at ways to characterize the reflective properties of the surfaces of an object.

Creating a light source.

OpenGL allows you to define up to eight sources, which are referred to through names `GL_LIGHT0`, `GL_LIGHT1`, etc. Each source is invested with various properties, and must be enabled. Each property has a default value. For example, to create a source located at $(3, 6, 5)$ in world coordinates, use²:

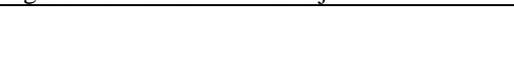
```
GLfloat myLightPosition[] = {3.0, 6.0, 5.0, 1.0};
glLightfv(GL_LIGHT0, GL_POSITION, myLightPosition);
glEnable(GL_LIGHTING); // enable
glEnable(GL_LIGHT0); // enable this particular source
```

The array `myLightPosition[]` (use any name you wish for this array) specifies the location of the light source, and is passed to `glLightfv()` along with the name `GL_LIGHT0` to attach it to the particular source `GL_LIGHT0`.

Some sources, such as a desk lamp, are “in the scene”, whereas others, like the sun, are infinitely remote. OpenGL allows you to create both types by using homogeneous coordinates to specify light position:

- $(x, y, z, 1)$: a local light source at the position (x, y, z)
- $(x, y, z, 0)$: a vector to an infinitely remote light source in the direction (x, y, z)

Figure 8.20 shows a local source positioned at $(0, 3, 3, 1)$ and a remote source “located” along vector $(3, 3, 0, 0)$. Infinitely remote light sources are often called “**directional**”. There are computational advantages to using directional light sources, since the direction \mathbf{s} in the calculations of diffuse and specular reflections is *constant* for all vertices in the scene. But directional light sources are not always the correct choice: some visual effects are properly achieved only when a light source is close to an object.



¹ In Section 8.5 we discuss replacing linear interpolation by “hyperbolic interpolation” as a more accurate way to form the colors at the new vertices formed by clipping.

² Here and elsewhere the type `float` would most likely serve as well as `GLfloat`. But using `GLfloat` makes your code more portable to other OpenGL environments.

Figure 8.20. A local source and an infinitely remote source.

You can also spell out different colors for a light source. OpenGL allows you to assign a different color to three “types of light” that a source emits: ambient, diffuse, and specular. It may seem strange to say that a source emits ambient light. It is still treated as in Equation 8.7: a global omnidirectional light that bathes the entire scene. The advantage of attaching it to a light source is that it can be turned on and off as an application proceeds. (OpenGL also offers a truly ambient light, not associated with any source, as we discuss later in connection with “lighting models”.)

Arrays are defined to hold the colors emitted by light sources, and they are passed to `glLightfv()`.

```
GLfloat amb0[] = {0.2, 0.4, 0.6, 1.0}; // define some colors
GLfloat diff0[] = {0.8, 0.9, 0.5, 1.0};
GLfloat spec0[] = { 1.0, 0.8, 1.0, 1.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, amb0); // attach them to LIGHT0
glLightfv(GL_LIGHT0, GL_DIFFUSE, diff0);
glLightfv(GL_LIGHT0, GL_SPECULAR, spec0);
```

Colors are specified in so-called **RGBA** format, meaning red, green, blue, and “alpha”. The alpha value is sometimes used for blending two colors on the screen. We discuss it in Chapter 10. For our purposes here it is normally 1.0.

Light sources have various default values. For all sources:

default ambient = (0, 0, 0, 1); ← dimmest possible: black

For light source `LIGHT0`:

default diffuse = (1, 1, 1, 1); ← brightest possible: white

default specular = (1, 1, 1, 1); ← brightest possible: white

whereas for the other sources the diffuse and specular values have defaults of black.

Spotlights.

Light sources are *point sources* by default, meaning that they emit light uniformly in all directions. But OpenGL allows you to make them into spotlights, so they emit light in a restricted set of directions. Figure 8.21 shows a spotlight aimed in direction \mathbf{d} , with a “cutoff angle” of α . No light is seen at points lying outside the cutoff cone. For vertices such as P that lie inside the cone, the amount of light reaching P is attenuated by the factor $\cos^\epsilon(\beta)$ where β is the angle between \mathbf{d} and a line from the source to P , and ϵ is an exponent chosen by the user to give the desired fall-off of light with angle.

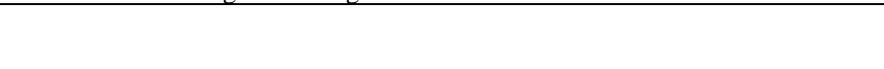


Figure 8.21. Properties of a spotlight.

The parameters for a spotlight are set using `glLightf()` to set a single value, and `glLightfv()` to set a vector:

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0); // a cutoff angle of 45°
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 4.0); // ε = 4.0
GLfloat dir[] = {2.0, 1.0, -4.0}; // the spotlight's direction
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, dir);
```

The default values for these parameters are $\mathbf{d} = (0,0,-1)$, $\alpha = 180^\circ$, and $\epsilon = 0$, which makes a source an omni-directional point source.

Attenuation of light with distance.

OpenGL also allows you to specify how rapidly light diminishes with distance from a source. Although we have downplayed the importance of this dependence, it can be interesting to

experiment with different fall-off rates, and to fine tune a picture. OpenGL attenuates the strength of a positional³ light source by the following attenuation factor:

$$atten = \frac{1}{k_c + k_l D + k_q D^2} \quad (8.11)$$

where k_c , k_l , and k_q are coefficients and D is the distance between the light's position and the vertex in question. This expression is rich enough to allow you to model any combination of constant, linear, and quadratic (inverse square law) dependence on distance from a source. These parameters are controlled by function calls:

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
```

and similarly for `GL_LINEAR_ATTENUATION`, and `GL_QUADRATIC_ATTENUATION`. The default values are $k_c = 1$, $k_l = 0$, and $k_q = 0$, which eliminate any attenuation.

Lighting model.

OpenGL allows three parameters to be set that specify general rules for applying the shading model. These parameters are passed to variations of the function `glLightModel`.

a). **The color of global ambient light.** You can establish a global ambient light source in a scene that is independent of any particular source. To create this light, specify its color using:

```
GLfloat amb[] = {0.2, 0.3, 0.1, 1.0};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, amb);
```

This sets the ambient source to the color (0.2, 0.3, 0.1). The default value is (0.2, 0.2, 0.2, 1.0), so this ambient light is always present unless you purposely alter it. This makes objects in a scene visible even if you have not invoked any of the lighting functions.

b). **Is the viewpoint local or remote?** OpenGL computes specular reflections using the “halfway vector” $\mathbf{h} = \mathbf{s} + \mathbf{v}$ described in Section 8.2.3. The true directions \mathbf{s} and \mathbf{v} are normally different at each vertex in a mesh (visualize this). If the light source is directional then \mathbf{s} is constant, but \mathbf{v} still varies from vertex to vertex. Rendering speed is increased if \mathbf{v} is made constant for all vertices. This is the default: OpenGL uses $\mathbf{v} = (0, 0, 1)$, which points along the positive z-axis in camera coordinates. You can force the pipeline to compute the true value of \mathbf{v} for each vertex by executing:

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

c). **Are both sides of a polygon shaded properly?** Each polygonal face in a model has two sides. When modeling we tend to think of them as the “inside” and “outside” surfaces. The convention is to list the vertices of a face in counter-clockwise (CCW) order as seen from outside the object. Most mesh objects represent solids that enclose space, so there is a well defined inside and outside. For such objects the camera can only see the outside surface of each face (assuming the camera is not inside the object!). With proper hidden surface removal the inside surface of each face is hidden from the eye by some closer face.

OpenGL has no notion of “inside” and “outside.” It can only distinguish between “front faces” and “back faces”. A face is a **front face** if its vertices are listed in counter-clockwise (CCW) order as seen by the eye⁴. Figure 8.22a shows the eye viewing a cube, which we presume was modeled using the CCW ordering convention. Arrows indicate the order in which the vertices of each face are passed to OpenGL (in a `glBegin(GL_POLYGON);...; glEnd()` block). For a space-enclosing object all faces that are visible to the eye are therefore front faces, and OpenGL draws them

³ This attenuation factor is disabled for directional light sources, since they are infinitely remote.

⁴ You can reverse this sense with `glFrontFace(GL_CW)`, which decrees that a face is a front face only if its vertices are listed in clock-wise order. The default is `glFrontFace(GL_CCW)`.

properly with the correct shading. OpenGL also draws the back faces⁵, but they are ultimately hidden by closer front faces.



Figure 8.22. OpenGL's definition of a front face.

Things are different in part b, which shows a box with a face removed. Again arrows indicate the order in which vertices of a face are sent down the pipeline. Now three of the visible faces are back faces. By default OpenGL does not shade these properly. To coerce OpenGL to do proper shading of back faces, use:

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

Then OpenGL reverses the normal vectors of any back-face so that they point toward the viewer, and it performs shading computations properly. Replace `GL_TRUE` with `GL_FALSE` (the default) to turn off this facility.

Note: Faces drawn by OpenGL do not cast shadows, so the back faces receive the same light from a source even though there may be some other face between them and the source.

Moving light sources.

Recall that light sources pass through the modelview matrix just as vertices do. Therefore lights can be repositioned by suitable uses of `glRotated()` and `glTranslated()`. The array `position` specified using `glLightfv(GL_LIGHT0, GL_POSITION, position)` is modified by the modelview matrix in effect at the time `glLightfv()` is called. So to modify the light position with transformations, and independently move the camera, embed the light positioning command in a push/pop pair, as in:

```
void display()
{
    GLfloat position[] = {2, 1, 3, 1}; //initial light position

    <.. clear color and depth buffers ..>
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glPushMatrix();
    glRotated(...); // move the light
    glTranslated(...);
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glPopMatrix();

    gluLookAt(...); // set the camera position
    <.. draw the object ..>
    glutSwapBuffers();
}
```

On the other hand, to have the light move with the camera, use:

```
GLfloat pos[] = {0,0,0,1};
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glLightfv(GL_LIGHT0, GL_POSITION, position); // light at (0,0,0)
gluLookAt(...); // move the light and the camera
<.. draw the object ..>
```

This establishes the light to be positioned at the eye (like a minor's lamp), and the light moves with the camera.

8.2.9. Working with Material Properties in OpenGL.

⁵ You can improve performance by instructing OpenGL to skip rendering of back faces, with `glCullFace(GL_BACK); glEnable(GL_CULL_FACE);`

You can see the effect of a light source only when light reflects off an object's surface. OpenGL provides ways to specify the various reflection coefficients that appear in Equation 8.7. They are set with variations of the function `glMaterialfv`, and they can be specified individually for front faces and back faces (see the discussion concerning Figure 8.22). For instance,

```
GLfloat myDiffuse[] = {0.8, 0.2, 0.0, 1.0};
glMaterialfv(GL_FRONT, GL_DIFFUSE, myDiffuse);
```

sets the diffuse reflection coefficient $(\rho_{dr}, \rho_{dg}, \rho_{db}) = (0.8, 0.2, 0.0)$ for all subsequently specified front faces. Reflection coefficients are specified as a 4-tuple in RBGA format, just like a color. The first parameter of `glMaterialfv()` can take on values:

`GL_FRONT`: set the reflection coefficient for front faces
`GL_BACK`: set it for back faces
`GL_FRONT_AND_BACK`: set it for both front and back faces

The second parameter can take on values:

`GL_AMBIENT`: set the ambient reflection coefficients
`GL_DIFFUSE`: set the diffuse reflection coefficients
`GL_SPECULAR`: set the specular reflection coefficients
`GL_AMBIENT_AND_DIFFUSE`: set both the ambient and diffuse reflection coefficients to the same values. This is for convenience, since the ambient and diffuse coefficients are so often chosen to be the same.
`GL_EMISSION`: set the emissive color of the surface.

The last choice sets the **emissive color** of a face, causing it to “glow” in the specified color, independent of any light source.

Putting it all together.

We now extend Equation 8.7 to include the additional contributions that OpenGL actually calculates. The total red component is given by:

$$I_r = e_r + I_{mr} \rho_{ar} + \sum_i \text{atten}_i \times \text{spot}_i \times (I_{ar}^i \rho_{ar} + I_{dr}^i \rho_{dr} \times \text{lambert}_i + I_{spr}^i \rho_{sr} \times \text{phong}_i^f) \quad (8.12)$$

Expressions for the green and blue components are similar. The emissive light is e_r , and I_{mr} is the global ambient light introduced in the lighting model. The summation denotes that the ambient, diffuse, and specular contributions of all light sources are summed. For the i -th source atten_i is the attenuation factor as in Equation 8.10, spot_i is the spotlight factor (see Figure 8.21), and lambert_i and phong_i are the familiar diffuse and specular dot products. All of these terms must be recalculated for each source.

Note: If I_r turns out to have a value larger than 1.0, OpenGL clamps it to 1.0: the brightest any light component can be is 1.0.

8.2.10. Shading of Scenes Specified by SDL.

The scene description language SDL introduced in Chapter 5 supports the loading of material properties into objects, so that they can be shaded properly. For instance,

```
light 3 4 5 .8 .8 .8 ! bright white light at (3, 4, 5)
background 1 1 1 ! white background
globalAmbient .2 .2 .2 ! a dark gray global ambient light
ambient .2 .6 0
diffuse .8 .2 1 ! red material
specular 1 1 1 ! bright specular spots - the color of the source
exponent 20 !set the Phong exponent
scale 4 4 4 sphere
```

describes a scene containing a sphere with material properties (see Equation 8.7):

- ambient reflection coefficients: $(\rho_{ar}, \rho_{ag}, \rho_{ab}) = (.2, 0.6, 0)$,
- diffuse reflection coefficients: $(\rho_{dr}, \rho_{dg}, \rho_{db}) = (0.8, 0.2, 1.0)$,
- specular reflection coefficients: $(\rho_{sr}, \rho_{sg}, \rho_{sb}) = (1.0, 1.0, 1.0)$
- and Phong exponent $f = 20$.

The light source is given a color of $(0.8, 0.8, 0.8)$ for both its diffuse and specular components. There is a global ambient term $(I_{ar}, I_{ag}, I_{ab}) = (0.2, 0.2, 0.2)$.

The current material properties are loaded into each object's `mtl` field at the time it is created (see the end of `Scene::getObject()` in `Shape.cpp` of Appendix 4). When an object draws itself using its `drawOpenGL()` method, it first passes its material properties to OpenGL (see `Shape::tellMaterialsGL()`), so that at the moment it is actually drawn OpenGL has these properties in its current state.

In Chapter 14 when raytracing we shall use each object's material field in a similar way to acquire the material properties and do proper shading.

8.3. Flat Shading and Smooth Shading.

Different objects require different shading effects. In Chapter 6 we modeled a variety of shapes using polygonal meshes. For some, like the barn or buckyball, we want to see the individual faces in a picture, but for others, like the sphere or chess pawn, we want to see the "underlying" surface that the faces approximate.

In the modeling process we attached a normal vector to each vertex of each face. If a certain face is to appear as a distinct polygon we attach the *same* normal vector to all of its vertices; the normal vector chosen is the normal direction to the plane of the face. On the other hand, if the face is supposed to approximate an underlying surface we attach to each vertex the normal to the underlying surface at that point.

We examine now how the normal vector information at each vertex is used to perform different kinds of shading. The main distinction is between a shading method that accentuates the individual polygons (flat shading) and a method that blends the faces to de-emphasize the edges between them (smooth shading). There are two kinds of smooth shading, called Gouraud and Phong shading, and we shall discuss both.

For both kinds of shading the vertices are passed down the graphics pipeline, shading calculations are performed to attach a color to each vertex, and ultimately the vertices of the face are converted to screen coordinates and the face is "painted" pixel by pixel with the appropriate color.

Painting a Face.

The face is colored using a polygon-fill routine. Filling a polygon is very simple, although fine tuning the fill algorithm for highest efficiency can get complex. (See Chapter 10.) Here we look at the basics, focusing on how the color of each pixel is set.

A polygon-fill routine is sometimes called a **tiler**, because it moves over the polygon pixel by pixel, coloring each pixel as appropriate, as one would lay down tiles on a parquet floor. Specifically, the pixels in a polygon are visited in a regular order, usually scan-line by scan-line from the bottom to the top of the polygon, and across each scan-line from left to right.

We assume here that the polygons of interest are *convex*. A tiler designed to fill only convex polygons can be made highly efficient, since at each scan-line there is a single unbroken "run" of pixels that lie inside the polygon. Most implementations of OpenGL exploit this and always fill convex polygons correctly, but do not guarantee to fill non-convex polygons properly. See the exercises for more thoughts on convexity.

Figure 8.23 shows an example where the face is a convex quadrilateral. The screen coordinates of each vertex are noted. The lowest and highest points on the face are y_{bot} and y_{top} , respectively. The tiler first fills in the row at $y = y_{\text{bot}}$ (in this case a single pixel), then the one at $y_{\text{bot}} + 1$, etc. At each scan-line, say y_s in the figure, there is a leftmost pixel, x_{left} , and a rightmost, x_{right} . The tiler moves from x_{left} to x_{right} , placing the desired color in each pixel. So the tiler is implemented as a simple double loop:



Figure 8.23. Filling a polygonal face with color.

```
for (int y = y_bott; y <= y_top; y++)          // for each scan-line
{
    <.. find x_left and x_right ..>
    for (int x = x_left; x <= x_right; x++)// fill across the scan-line
    {
        <.. find the color c for this pixel ..>
        <.. put c into the pixel at (x, y) ..>
    }
}
```

(We shall see later how hidden surface removal is easily accomplished within this double loop as well.) The principal difference between flat and smooth shading is the manner in which the color *c* is determined at each pixel.

8.3.1. Flat Shading.

When a face is flat (like the roof of a barn) and the light sources are quite distant the diffuse light component varies little over different points on the roof (the *lambert* term in Equation 8.6 is nearly the same at each vertex of the face). In such cases it is reasonable to use the same color for every pixel “covered” by the face. OpenGL offers a rendering mode in which the entire face is drawn with the same color. Although a color is passed down the pipeline as part of each vertex of the face, the painting algorithm uses only one of them (usually that of the first vertex in the face). So the command above, <find the color *c* for this pixel>, is not inside the loops but instead appears just prior to the loops, setting *c* to the color of one of the vertices. (Using the same color for every pixel tends to make flat shading quite fast.)

Flat shading is established in OpenGL using:

```
glShadeModel (GL_FLAT);
```

Figure 8.24 shows a buckyball and a sphere rendered using flat shading. The individual faces are clearly visible on both objects. The sphere is modeled as a smooth object, but no smoothing is taking place in the rendering, since the color of an entire face is set to that of only one vertex.

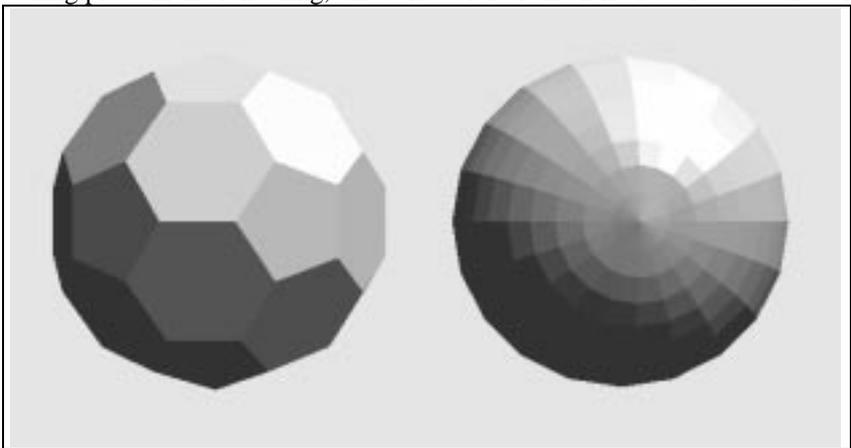


Figure 8.24. Two meshes rendered using flat shading.

Edges between faces actually appear more pronounced than they “are”, due to a phenomenon in the eye known as **lateral inhibition**, first described by Ernst Mach⁶. When there is a discontinuity in intensity across an object the eye manufactures a **Mach band** at the discontinuity, and a vivid edge

⁶ Ernst Mach (1838-1916), an Austrian physicist, whose early work strongly influenced the theory of relativity.

is seen (as discussed further in the exercises). This exaggerates the polygonal “look” of mesh objects rendered with flat shading.

Specular highlights are rendered poorly with flat shading, again because an entire face is filled with a color that was computed at only one vertex. If there happens to be a large specular component at the representative vertex, that brightness is drawn uniformly over the entire face. If a specular highlight doesn’t fall on the representative point, it is missed entirely. For this reason, there is little incentive for including the specular reflection component in the shading computation.

8.3.2. Smooth Shading.

Smooth shading attempts to de-emphasize edges between faces by computing colors at more points on each face. There are two principal types of smooth shading, called Gouraud shading and Phong shading [gouraud71, phong75]. OpenGL does only Gouraud shading, but we describe both of them.

Gouraud shading computes a different value of c for each pixel. For the scanline at y_s (in figure 8.23) it finds the color at the leftmost pixel, $color_{left}$ by linear interpolation of the colors at the top and bottom of the left edge⁷. For the scan-line at y_s the color at the top is $color_4$ and that at the bottom is $color_1$, so $color_{left}$ would be calculated as (recall Equation 8.9):

$$color_{left} = lerp(color_1, color_4, f) \quad (8.13)$$

where fraction f , given by

$$f = \frac{y_s - y_{bott}}{y_4 - y_{bott}}$$

varies between 0 and 1 as y_s varies from y_{bott} to y_4 . Note that Equation 8.13 involves three calculations since each color quantity has a red, green, and blue component.

Similarly $color_{right}$ is found by interpolating the colors at the top and bottom of the right edge. The tiler then fills across the scanline, linearly interpolating between $color_{left}$ and $color_{right}$ to obtain the color at pixel x :

$$c(x) = lerp(color_{left}, color_{right}, \frac{x - x_{left}}{x_{right} - x_{left}}) \quad (8.14)$$

To increase efficiency this color is computed incrementally at each pixel. That is, there is a constant difference between $c(x+1)$ and $c(x)$, so

$$c(x+1) = c(x) + \frac{color_{right} - color_{left}}{x_{right} - x_{left}} \quad (8.15)$$

The increment is calculated only once outside of the innermost loop. In terms of code this looks like:

```
for (int y = Ybott; y <= Ytop; y++)          // for each scan-line
{
    <.. find xleft and xright ..>
    <.. find colorleft and colorright ..>
    colorinc = (colorright - colorleft) / (xright - xleft);
    for (int x = xleft, c = colorleft; x <= xright; x++, c+=colorinc)
        <.. put c into the pixel at (x, y) ..>
}
```

⁷ We shall see later that, although colors are usually interpolated *linearly* as we do here, better results can be obtained by using so-called *hyperbolic interpolation*. For Gouraud shading the distinction is minor; for texture mapping it is crucial.

Gouraud shading is modestly more expensive computationally than flat shading. Gouraud shading is established in OpenGL using:

```
glShadeModel (GL_SMOOTH) ;
```

Figure 8.25 shows a buckyball and a sphere rendered using Gouraud shading. The buckyball looks the same as when it was flat shaded in Figure 8.24, because the same color is associated with each vertex of a face, so interpolation changes nothing. But the sphere looks much smoother. There are no abrupt jumps in color between neighboring faces. The edges of the faces (and the Mach bands) are gone, replaced by a smoothly varying color across the object. Along the silhouette, however, you can still see the bounding edges of individual faces.

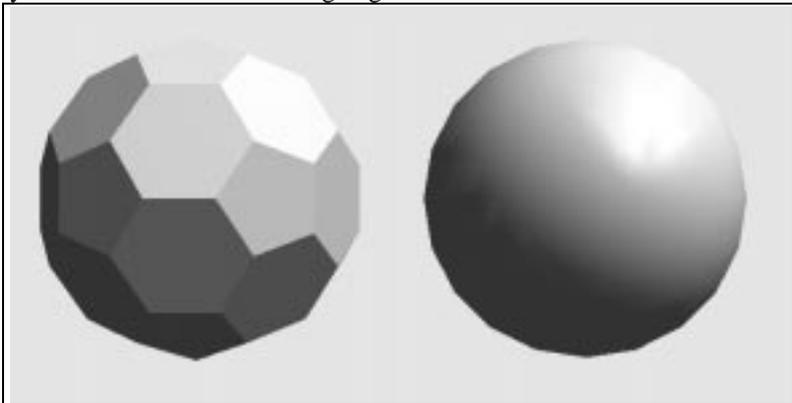


Figure 8.25. Two meshes rendered using smooth shading. (file: fig8.25.bmp)

Why do the edges disappear with this technique? Figure 8.26a shows two faces, F and F' , that share an edge. When rendering F the colors c_L and c_R are used, and when rendering F' the colors c_L' and c_R' are used. But since c_R equals c_L' there is an abrupt change in color at the edge along the scanline.

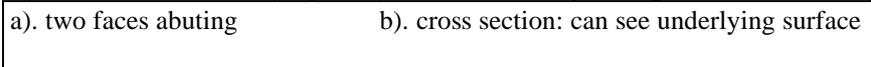


Figure 8.26. Continuity of color across a polygon edge.

Figure 8.26b suggests how this technique reveals the “underlying” surface approximated by the mesh. The polygonal surface is shown in cross section, with vertices V_1, V_2 , etc. marked. The imaginary smooth surface that the mesh supposedly represents is suggested as well. Properly computed vertex normals $\mathbf{m}_1, \mathbf{m}_2$, etc. point perpendicularly to this imaginary surface, so the normal for “correct” shading is being used at each vertex, and the color thereby found is correct. The color is then made to vary smoothly between vertices, not following any physical law but rather a simple mathematical one.

Because colors are formed by interpolating rather than computing colors at every pixel, Gouraud shading does not picture highlights well. Therefore, when Gouraud shading is used, one normally suppresses the specular component of intensity in Equation 8.12. Highlights are better reproduced using Phong shading, discussed next.

Phong Shading.

Greater realism can be achieved - particularly with regard to highlights on shiny objects - by a better approximation of the normal vector to the face at each pixel. This type of shading is called **Phong shading**, after its inventor Phong Bui-tuong [phong75].

When computing Phong shading we find the normal vector *at each point* on the face and we apply the shading model there to find the color. We compute the normal vector at each pixel by interpolating the normal vectors at the vertices of the polygon.

Figure 8.27 shows a projected face, with the normal vectors $\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3$, and \mathbf{m}_4 indicated at the four vertices. For the scan-line y_s as shown the vectors \mathbf{m}_{left} and \mathbf{m}_{right} are found by linear interpolation. For instance, \mathbf{m}_{left} is found as



Figure 8.27. Interpolating normals.

$$\mathbf{m}_{left} = \text{lerp}(\mathbf{m}_4, \mathbf{m}_3, \frac{y_s - y_4}{y_3 - y_4})$$

This interpolated vector must be normalized to unit length before its use in the shading formula. Once \mathbf{m}_{left} and \mathbf{m}_{right} are known, they are interpolated to form a normal vector at each x along the scan-line. This vector, once normalized, is used in the shading calculation to form the color at that pixel.

Figure 8.28 shows an object rendered using Gouraud shading and Phong shading. Because the direction of the normal vector varies smoothly from point to point and more closely approximates that of an underlying smooth surface, the production of specular highlights is much more faithful than with Gouraud shading, and more realistic renderings are produced.

1st Ed. Figure 15.25

Figure 8.28. Comparison of Gouraud and Phong shading (Courtesy of Bishop and Weimar 1986).

The principal drawback of Phong shading is its speed: a great deal more computation is required per pixel, so that Phong shading can take 6 to 8 times longer than Gouraud shading to perform. A number of approaches have been taken to speed up the process [bishop86, claussen90].

OpenGL is not set up to do Phong shading, since it applies the shading model once per vertex right after the modelview transformation, and normal vector information is not passed to the rendering stage following the perspective transformation and perspective divide. We will see in Section 8.5, however, that an approximation to Phong shading can be created by mapping a “highlight” texture onto an object using the environment mapping technique.

Practice Exercises.

8.3.1. Filling your face. Fill in details of how the polygon fill algorithm operates for the polygon with vertices $(x, y) = (23, 137), (120, 204), (200, 100), (100, 25)$, for scan lines $y = 136, y = 137$, and $y = 138$. Specifically write the values of x_{left} and x_{right} in each case.

8.3.2. Clipped convex polygons are still convex. Develop a proof that if a convex polygon is clipped against the camera’s view volume, the clipped polygon is still convex.

8.3.3. Retaining edges with Gouraud Shading. In some cases we may want to show specific creases and edges in the model. Discuss how this can be controlled by the choice of the vertex normal vectors. For instance, to retain the edge between faces F and F' in Figure 8.26, what should the vertex normals be? Other tricks and issues can be found in the references [e.g. Rogers85].

8.3.4. Faster Phong shading with fence shading. To increase the speed of Phong shading Behrens [behrens94] suggests interpolating normal vectors between vertices to get \mathbf{m}_L and \mathbf{m}_R in the usual way at each scan line, but then computing colors only at these left and right pixels, interpolating them along a scan line as in Gouraud shading. This so-called “fence shading” speeds up rendering dramatically, but does less well in rendering highlights than true Phong shading. Describe general directions for the vertex normals $\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3$, and \mathbf{m}_4 in Figure 8.27 such that

- Fence shading produces the same highlights as Phong shading;
- Fence shading produces very different highlights than does Phong shading.

8.3.5. The Phong shading algorithm. Make the necessary changes to the tiling code to incorporate Phong shading. Assume the vertex normal vectors are available for each face. Also discuss how Phong shading can be approximated by OpenGL’s smooth shading algorithm. Hint: increase the number of faces in the model.

8.4. Adding Hidden Surface Removal

It is very simple to incorporate hidden surface removal in the rendering process above if enough memory is available to have a “depth buffer” (also called a “z-buffer”). Because it fits so easily into the rendering mechanisms we are discussing, we include it here. Other (more efficient and less memory-hungry) hidden surface removal algorithms are described in Chapter 13.

8.4.1. The Depth Buffer Approach.

The depth buffer (or z-buffer) algorithm is one of the simplest and most easily implemented hidden surface removal methods. Its principal limitations are that it requires a large amount of memory,

and that it often renders an object that is later obscured by a nearer object (so time spent rendering the first object is wasted).

Figure 8.29 shows a depth buffer associated with the frame buffer. For every pixel $p[i][j]$ on the display the depth buffer stores a b bit quantity $d[i][j]$. The value of b is usually in the range of 12 to 30 bits.



Figure 8.29. Conceptual view of the depth buffer.

During the rendering process the depth buffer value $d[i][j]$ contains the pseudodepth of the closest object encountered (so far) at that pixel. As the tiler proceeds pixel by pixel across a scan-line filling the current face, it tests whether the pseudodepth of the current face is less than the depth $d[i][j]$ stored in the depth buffer at that point. If so the color of the closer surface replaces the color $p[i][j]$ and this smaller pseudodepth replaces the old value in $d[i][j]$. Faces can be drawn in any order. If a remote face is drawn first some of the pixels that show the face will later be replaced by the colors of a nearer face. The time spent rendering the more remote face is therefore wasted. Note that this algorithm works for objects of any shape including curved surfaces, because it finds the closest surface based on a point-by-point test.

The array $d[][]$ is initially loaded with value 1.0, the greatest pseudodepth value possible. The frame buffer is initially loaded with the background color.

Finding the pseudodepth at each pixel.

We need a rapid way to compute the pseudodepth at each pixel. Recall that each vertex $P = (P_x, P_y, P_z)$ of a face is sent down the graphics pipeline, and passes through various transformations. The information available for each vertex after the viewport transformation is the 3-tuple that is a scaled and shifted version of (see Equation 7.2)

$$(x, y, z) = \left(\frac{P_x}{-P_z}, \frac{P_y}{-P_z}, \frac{aP_z + b}{-P_z} \right)$$

The third component is pseudodepth. Constants a and b have been chosen so that the third component equals 0 if P lies in the near plane, and 1 if P lies in the far plane. For highest efficiency we would like to compute it at each pixel incrementally, which implies using linear interpolation as we did for color in Equation 8.15.

Figure 8.30 shows a face being filled along scanline y . The pseudodepth values at various points are marked. The pseudodepths $d_1, d_2, d_3,$ and d_4 at the vertices are known. We want to calculate d_{left} at scan-line y_s as $\text{lerp}(d_1, d_4, f)$ for fraction $f = (y_s - y_1)/(y_4 - y_1)$, and similarly d_{right} as $\text{lerp}(d_2, d_3, h)$ for the appropriate h . And we want to find the pseudodepth d at each pixel (x, y) along the scan-line as $\text{lerp}(d_{\text{left}}, d_{\text{right}}, k)$ for the appropriate k . (What are the values of h and k ?) The question is whether this calculation produces the “true” pseudodepth of the corresponding point on the 3D face.



Figure 8.30. Incremental computation of pseudodepth.

The answer is that it works correctly. We prove this later after developing some additional algebraic artillery, but the key idea is that the original 3D face is flat, and perspective projection preserves flatness, so pseudodepth varies linearly with the projected x and y coordinates. (See Exercise 8.5.2.)

Figure 8.31 shows the nearly trivial additions to the Gouraud shading tiling algorithm that accomplish hidden surface removal. Values of d_{left} and d_{right} are found (incrementally) for each scan-line, along with d_{inc} which is used in the innermost loop. For each pixel d is found, a single comparison is made, and an update of $d[i][j]$ is made if the current face is found to be closest.

```
for (int y = Y_bott; y <= Y_top; y++) // for each scan-line
{
    <.. find x_left and x_right ..>
    <.. find d_left and d_right, and d_inc ..>
    <.. find color_left and color_right, and color_inc ..>
```

```

for (int x = xleft, c = colorleft, d = dleft; x <= xright; x++, c+=colorinc, d+= dinc)
    if(d < d[x][y])
    {
        <.. put c into the pixel at (x, y) ..>
        d[x][y] = d; // update the closest depth
    }
}

```

Figure 8.31. Doing depth computations incrementally.

Depth compression at greater distances.

Recall from Example 7.4.4 that the pseudodepth of a point does not vary linearly with actual depth from the eye, but instead approaches an asymptote. This means that small changes in true depth map into extremely small changes in pseudodepth when the depth is large. Since only a limited number of bits are used to represent pseudodepth, two nearby values can easily map into the same value, which can lead to errors in the comparison $d < d[x][y]$. Using a larger number of bits to represent pseudodepth helps, but this requires more memory. It helps a little to place the near plane as far away from the eye as possible.

OpenGL supports a depth buffer, and uses the algorithm described above to do hidden surface removal. You must instruct OpenGL to create a depth buffer when it initializes the display mode:

```
glutInitDisplayMode(GLUT_DEPTH | GLUT_RGB);
```

and enable depth testing with

```
glEnable(GL_DEPTH_TEST);
```

Then each time a new picture is to be created the depth buffer must be initialized using:

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT); // clear screen
```

Practice Exercises.

8.4.1. The increments. Fill in details of how d_{left} , d_{right} , and d are found from the pseudodepth values known at the polygon vertices.

8.4.2. Coding depth values. Suppose b bits are allocated for each element in the depth buffer. These b bits must record values of pseudodepth between 0 and 1. A value between 0 and 1 can be expressed in binary in the form $.d_1d_2d_3\dots d_b$ where d_i is 0 or 1. For instance, a pseudodepth of 0.75 would be coded as .1100000000... Is this a good use of the b bits? Discuss alternatives.

8.4.3. Reducing the Size of the Depth Buffer. If there is not enough memory to implement a full depth buffer, one can generate the picture in pieces. A depth buffer is established for only a fraction of the scan lines, and the algorithm is repeated for each fraction. For instance, in a 512-by-512 display, one can allocate memory for a depth buffer of only 64 scan lines and do the algorithm eight times. Each time the entire face list is scanned, depths are computed for faces covering the scan lines involved, and comparisons are made with the reigning depths so far. Having to scan the face list eight times, of course, makes the algorithm operate more slowly. Suppose that a scene involves F faces, and each face covers on the average L scanlines. Estimate how much more time it takes to use the depth buffer method when memory is allocated for only $n\text{Rows}/N$ scanlines.

8.4.4. A single scanline depth buffer. The fragmentation of the frame buffer of the previous exercise can be taken to the extreme where the depth buffer records depths for only *one* scan line. It appears to require more computation, as each face is “brought in fresh” to the process many times, once for each scan line. Discuss how the algorithm is modified for this case, and estimate how much longer it takes to perform than when a full-screen depth buffer is used.

8.5. Adding Texture to Faces.

I found Rome a city of bricks and left it a city of marble.
Augustus Caesar, from Suetonius

The realism of an image is greatly enhanced by adding surface texture to the various faces of a mesh object. Figure 8.32 shows some examples. In part a) images have been “pasted onto” each of the faces of a box. In part b) a label has been wrapped around a cylindrical can, and the wall behind the can appears to be made of bricks. In part c) a table has a wood-grain surface, and the

floor is tiled with decorative tiles. The picture on the wall contains an image pasted inside the frame.

a). box b). beer can c). wood table – screen shots

Figure 8.32. Examples of texture mapped onto surfaces.

The basic technique begins with some texture function in “**texture space**” such as that shown in Figure 8.33a. Texture space is traditionally marked off by parameters named s and t . The texture is a function $texture(s, t)$ which produces a color or intensity value for each value of s and t between 0 and 1.

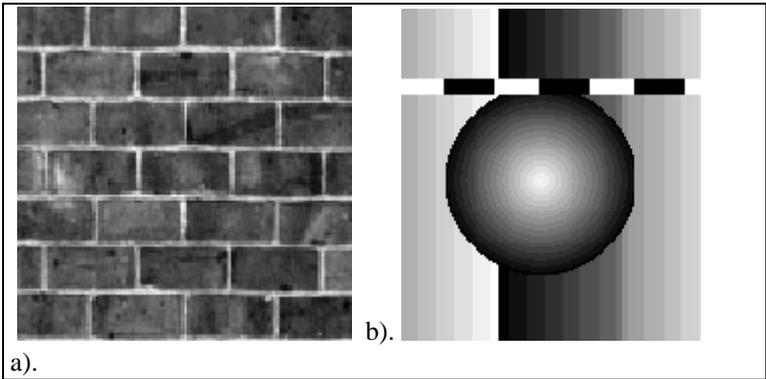


Figure 8.33. Examples of textures. a). image texture, b). procedural texture.

There are numerous sources of textures. The most common are bitmaps and computed functions.

• **Bitmap textures.**

Textures are often formed from bitmap representations of images (such as a digitized photo, clip art, or an image computed previously in some program). Such a texture consists of an array, say `txtr[c][r]`, of color values (often called **texels**). If the array has C columns and R rows, the indices c and r vary from 0 to $C-1$ and $R-1$, respectively. In the simplest case the function $texture(s, t)$ provides “samples” into this array as in

```
Color3 texture(float s, float t)
{
    return txtr[(int)(s * C)][(int)(t * R)];
}
```

where `Color3` holds an RGB triple. For example, if $R = 400$ and $C = 600$, then $texture(0.261, 0.783)$ evaluates to `txtr[156][313]`. Note that a variation of s from 0 to 1 encompasses 600 pixels, whereas the same variation in t encompasses 400 pixels. To avoid distortion during rendering this texture must be mapped onto a rectangle with aspect ratio 6/4.

• **Procedural textures.**

Alternatively we can define a texture by a mathematical function or procedure. For instance, the “sphere” shape that appears in Figure 8.33b could be generated by the function

```
float fakeSphere(float s, float t)
{
    float r = sqrt((s-0.5)*(s-0.5)+(t-0.5)*(t-0.5));
    if(r < 0.3) return 1 - r/0.3; // sphere intensity
    else return 0.2; // dark background
}
```

that varies from 1 (white) at the center to 0 (black) at the edges of the apparent sphere. Another example that mimics a checkerboard is examined in the exercises. Anything that can be computed can provide a texture: smooth blends and swirls of color, the Mandelbrot set, wireframe drawings of solids, etc.

We see later that the value $texture(s, t)$ can be used in a variety of ways: it can be used as the color of the face itself as if the face is “glowing”; it can be used as a reflection coefficient to “modulate” the amount of light reflected from the face; it can be used to alter the normal vector to the surface to give it a “bumpy” appearance.

Practice Exercise 8.5.1. The classic checkerboard texture. Figure 8.34 shows a checkerboard consisting of 4 by 5 squares with brightness levels that alternate between 0 (for black) and 1 (for white).

- Write the function `float texture(float s, float t)` for this texture. (See also Exercise 2.3.1.)
- Write `texture()` for the case where there are M rows and N columns in the checkerboard.
- Repeat part b for the case where the checkerboard is rotated 40° relative to the s and t axes.



Figure 8.34. A classic checkerboard pattern.

With a texture function in hand, the next step is to map it properly onto the desired surface, and then to view it with a camera. Figure 8.35 shows an example that illustrates the overall problem. Here a single example of texture is mapped onto three different objects: a planar polygon, a cylinder, and a sphere. For each object there is some transformation, say T_w (for “texture to world”) that maps texture (s, t) values to points (x, y, z) on the object’s surface. The camera takes a snapshot of the scene from some angle, producing the view shown. We call the transformation from points in 3D to points on the screen T_{ws} (“from world to screen”), so a point (x, y, z) on a surface is “seen” at pixel location $(sx, sy) = T_{ws}(x, y, z)$. So overall, the value (s^*, t^*) on the texture finally arrives at pixel $(sx, sy) = T_{ws}(T_w(s^*, t^*))$.



Figure 8.35. Drawing texture on several object shapes.

The rendering process actually goes the other way: for each pixel at (sx, sy) there is a sequence of questions:

- What is the closest surface “seen” at (sx, sy) ? This determines which texture is relevant.
- To what point (x, y, z) on this surface does (sx, sy) correspond?
- To which texture coordinate pair (s, t) does this point (x, y, z) correspond?

So we need the inverse transformation, something like $(s, t) = T_w^{-1}(T_{ws}^{-1}(sx, sy))$, that reports (s, t) coordinates given pixel coordinates. This inverse transformation can be hard to obtain or easy to obtain, depending on the surface shapes.

8.5.1. Pasting the Texture onto a Flat Surface.

We first examine the most important case: mapping texture onto a flat surface. This is a modeling task. In Section 8.5.2 we tackle the viewing task to see how the texture is actually rendered. We then discuss mapping textures onto more complicated surface shapes.

Pasting Texture onto a Flat Face.

Since texture space itself is flat, it is simplest to paste texture onto a flat surface. Figure 8.36 shows a texture image mapped to a portion of a planar polygon F . We must specify how to associate points on the texture with points on F . In OpenGL we associate a point in texture space $P_i = (s_i, t_i)$ with each vertex V_i of the face using the function `glTexCoord2f()`. The function `glTexCoord2f(s, t)` sets the “current texture coordinates” to (s, t) , and they are attached to subsequently defined vertices. Normally each call to `glVertex3f()` is preceded by a call to `glTexCoord2f()`, so each vertex “gets” a new pair of texture coordinates. For example, to define a quadrilateral face and to “position” a texture on it, we send OpenGL four texture coordinates and the four 3D points, as in:

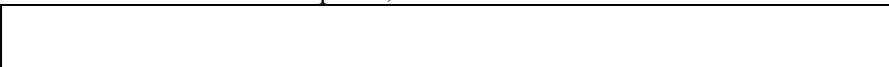


Figure 8.36. Mapping texture onto a planar polygon.

```
glBegin(GL_QUADS); // define a quadrilateral face
```

```

    glTexCoord2f(0.0, 0.0); glVertex3f(1.0, 2.5, 1.5);
    glTexCoord2f(0.0, 0.6); glVertex3f(1.0, 3.7, 1.5);
    glTexCoord2f(0.8, 0.6); glVertex3f(2.0, 3.7, 1.5);
    glTexCoord2f(0.8, 0.0); glVertex3f(2.0, 2.5, 1.5);
glEnd();

```

Attaching a P_i to each V_i is equivalent to prescribing a polygon P in texture space that has the same number of vertices as F . Usually P has the same shape as F as well: then the portion of the texture that lies inside P is pasted without distortion onto the whole of F . When P and F have the same shape the mapping is clearly affine: it is a scaling, possibly accompanied by a rotation and a translation.

Figure 8.37 shows the very common case where the four corners of the texture square are associated with the four corners of a rectangle. (The texture coordinates (s, t) associated with each corner are noted on the 3D face.) In this example the texture is a 640 by 480 pixel bitmap, and it is pasted onto a rectangle with aspect ratio 640/480, so it appears without distortion. (Note that the texture coordinates s and t still vary from 0 to 1.) Figure 8.38 shows the use of texture coordinates that “tile” the texture, making it repeat. To do this some texture coordinates that lie outside of the interval $[0,1]$ are used. When the renderer encounters a value of s and t outside of the unit square such as $s = 2.67$ it ignores the integral part and uses only the fractional part 0.67. Thus the point on a face that requires $(s, t) = (2.6, 3.77)$ is textured with $texture(0.6, 0.77)$. By default OpenGL tiles texture this way. It may be set to “clamp” texture values instead, if desired; see the exercises.



Figure 8.37. Mapping a square to a rectangle.



Figure 8.38. Producing repeated textures.

Thus a coordinate pair (s, t) is sent down the pipeline along with each vertex of the face. As we describe in the next section, the notion is that points inside F will be filled with texture values lying inside P , by finding the internal coordinate values (s, t) using interpolation. This interpolation process is described in the next section.

Adding texture coordinates to Mesh objects.

Recall from Figure 6.13 that a mesh object has three lists: the vertex, normal vector, and face lists. We must add to this a “texture coordinate” list, that stores the coordinates (s, t_i) to be associated with various vertices. We can add an array of elements of the type:

```
class TxtrCoord{public: float s, t;};
```

to hold all of the coordinate pairs of interest for the mesh. There are several different ways to treat texture for an object, and each has implications for how texture information is organized in the model. The two most important are:

1. The mesh object consists of a small number of flat faces, and a different texture is to be applied to each. Here each face has only a single normal vector but its own list of texture coordinates. So the data associated with each face would be:
 - the number of vertices in the face;
 - the index of the normal vector to the face;
 - a list of indices of the vertices;
 - a list of indices of the texture coordinates;
2. The mesh represents a smooth underlying object, and a single texture is to be “wrapped” around it (or a portion of it). Here each vertex has associated with it a specific normal vector and a particular texture coordinate pair. A single index into the vertex/normals/texture lists is used for each vertex. The data associated with each face would then be:
 - the number of vertices in the face;

- a list of indices of the vertices;

The exercises take a further look at the required data structures for these types of meshes.

8.5.2. Rendering the Texture.

Rendering texture in a face F is similar to Gouraud shading: the renderer moves across the face pixel by pixel. For each pixel it must determine the corresponding texture coordinates (s, t) , access the texture, and set the pixel to the proper texture color. We shall see that finding the coordinates (s, t) must be done very carefully.

Figure 8.39 shows the camera taking a snapshot of face F with texture pasted onto it, and the rendering in progress. Scanline y is being filled from x_{left} to x_{right} . For each x along this scanline we must compute the correct position (shown as $P(x, y)$) on the face, and from this obtain the correct position (s^*, t^*) within the texture.



Figure 8.39. Rendering the face in a camera snapshot.

Having set up the texture to object mapping, we know the texture coordinates at each of the vertices of F , as suggested in Figure 8.40. The natural thing is to compute $(s_{\text{left}}, t_{\text{left}})$ and $(s_{\text{right}}, t_{\text{right}})$ for each scanline in a rapid incremental fashion and to interpolate between these values moving across the scanline. But we must be careful: simple increments from s_{left} to s_{right} as we march across scanline y from x_{left} to x_{right} won't work, since equal steps across a projected face do *not* correspond to equal steps across the 3D face.



Figure 8.40. Incremental calculation of texture coordinates.

Figure 8.41 illustrates the problem. Part a shows face F viewed so that its left edge is closer to the viewer than its right edge. Part b shows the projection F' of this face on the screen. At scan-line $y = 170$ we mark points equally spaced across F' , suggesting the positions of successive pixels on the face. The corresponding positions of these marks on the actual face are shown in part a. They are seen to be more closely spaced at the farther end of F . This is simply the effect of perspective foreshortening.

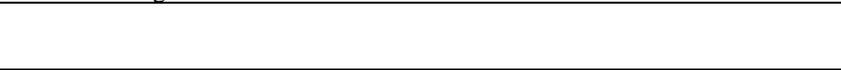


Figure 8.41. Spacing of samples with linear interpolation.

If we use simple linear interpolation and take equally spaced steps in s and t to compute texture coordinates, we “sample” into the texture at the wrong spots, and a distorted image results. Figure 8.42 shows what happens with a simple checkerboard texture mapped onto a rectangle. Linear interpolation is used in part a, producing palpable distortion in the texture. This distortion is particularly disturbing in an animation where the polygon is rotating, as the texture appears to warp and stretch dynamically. Correct interpolation is used in part b, and the checkerboard looks as it should. In an animation this texture would appear to be firmly attached to the moving or rotating face.

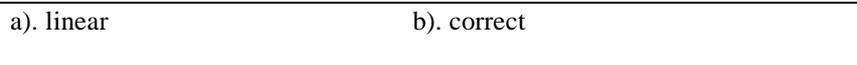


Figure 8.42. Images formed using linear interpolation and correct interpolation.

Several approaches have appeared in the literature that develop the proper interpolation method. Heckbert and Moreton [heckbert91] and Blinn [blinn92] describe an elegant development based on the general nature of affine and projective mappings. Segal et al [segal92] arrive at the same result using a more algebraic derivation based on the parametric representation for a line segment. We follow the latter approach here.

Figure 8.43 shows the situation to be analyzed. We know that affine and projective transformations preserve straightness, so line L_e in eye space projects to line L_s in screen space, and similarly the texels we wish to draw on line L_s lie along the line L_t in texture space that maps

to L_e . The key question is this: if we move in equal steps across L_s on the screen how should we step across texels along L_t in texture space?

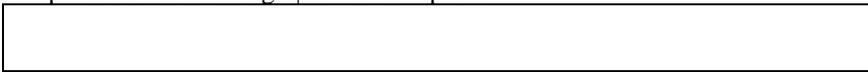


Figure 8.43. Lines in one space map to lines in another.

We develop a general result next that summarizes how interpolation works: it all has to do with the effect of perspective division. Then we relate the general result to the transformations performed in the graphics pipeline, and see precisely where extra steps must be taken to do proper mapping of texture.

Figure 8.44 shows the line AB in 3D being transformed into the line ab in 3D by matrix M . (M might represent an affine transformation, or a more general perspective transformation.) A maps to a , B maps to b . Consider the point $R(g)$ that lies fraction g of the way between A and B . It maps to some point $r(f)$ that lies fraction f of the way from a to b . The fractions f and g are *not* the same as we shall see. The question is, as f varies from 0 to 1 how exactly does g vary? That is, how does motion along ab correspond to motion along AB ?

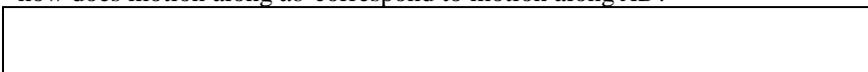


Figure 8.44. How does motion along corresponding lines operate?

Deriving how g and f are related.

We denote the homogeneous coordinate version of a by \tilde{a} , and name its components $\tilde{a} = (a_1, a_2, a_3, a_4)$. (We use subscripts 1,2,3, and 4 instead of x, y , etc. to prevent ambiguity, since there are so many different “ x, y, z ” spaces.) So point a is found from \tilde{a} by perspective division: $a = (\frac{a_1}{a_4}, \frac{a_2}{a_4}, \frac{a_3}{a_4})$. Since M maps $A = (A_1, A_2, A_3)$ to a we know $\tilde{a} = M(A, 1)^T$ where $(A, 1)^T$ is the column vector with components A_1, A_2, A_3 , and 1. Similarly, $\tilde{b} = M(B, 1)^T$. (Check each of these relations carefully.) Now using $lerp()$ notation to keep things succinct, we have defined $R(g) = lerp(A, B, g)$, which maps to $M(lerp(A, B, g), 1)^T = lerp(\tilde{a}, \tilde{b}, g) = (lerp(a_1, b_1, g), lerp(a_2, b_2, g), lerp(a_3, b_3, g), lerp(a_4, b_4, g))$. (Check these, too.) This is the homogeneous coordinate version $\tilde{r}(f)$ of the point $r(f)$. We recover the actual components of $r(f)$ by perspective division. For simplicity write just the first component $r_1(f)$, which is:

$$r_1(f) = \frac{lerp(a_1, b_1, g)}{lerp(a_4, b_4, g)} \tag{8.16}$$

But since by definition $r(f) = lerp(a, b, f)$ we have another expression for the first component $r_1(f)$:

$$r_1(f) = lerp(\frac{a_1}{a_4}, \frac{b_1}{b_4}, f) \tag{8.17}$$

Expressions (what are they?) for $r_2(f)$ and $r_3(f)$ follow similarly. Equate these two versions of $r_1(f)$ and do a little algebra to obtain the desired relationship between f and g :

$$g = \frac{f}{lerp(\frac{b_4}{a_4}, 1, f)} \tag{8.18}$$

Therefore the point $R(g)$ maps to $r(f)$, but g and f aren't the same fraction. g matches at $f = 0$ and at $f = 1$, but its growth with f is tempered by a denominator that depends on the ratio b_4/a_4 . If a_4 equals b_4 then g is identical to f (check this). Figure 8.45 shows how g varies with f , for different values of b_4/a_4 .



Figure 8.45. How g depends on f .

We can go the final step and show where the point $R(g)$ is on the 3D face that maps into $r(f)$. Simply use Equation 8.17 in $R(g) = A(1-g)+Bg$ and simplify algebraically (check this out) to obtain for the first component:

$$R_1 = \frac{\text{lerp}\left(\frac{A_1}{a_4}, \frac{B_1}{b_4}, f\right)}{\text{lerp}\left(\frac{1}{a_4}, \frac{1}{b_4}, f\right)} \quad (8.19)$$

with similar expressions resulting for the components R_2 and R_3 (which have the *same* denominator as R_1). This is a key result. It tells which 3D point (R_1, R_2, R_3) corresponds (in eye coordinates) to a given point that lies (fraction f of the way) between two given points a and b in screen coordinates. So any quantity (such as texture) that is “attached” to vertices of the 3D face and varies linearly between them will behave the same way.

The two cases of interest for the transformation with matrix M are:

- The transformation is affine;
- The transformation is the perspective transformation.

a). When the transformation is affine then a_4 and b_4 are both 1 (why?), so the formulas above simplify immediately. The fractions f and g become identical, and R_1 above becomes $\text{lerp}(A_1, B_1, f)$. We can summarize this as:

Fact: If M is *affine*, equal steps along the line ab do correspond to equal steps along the line AB .

b). When M represents the perspective transformation from eye coordinates to clip coordinates the fourth components a_4 and b_4 are no longer 1. We developed the matrix M in Chapter 7. Its basic form, given in Equation 7.10, is:

$$M = \begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & c & d \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

where c and d are constants that make pseudodepth work properly. What is $M(A,1)^T$ for this matrix? It's $\tilde{a} = (NA_1, NA_2, cA_3 + d, -A_3)$, the crucial part being that $a_4 = -A_3$. This is the position of the point along the z -axis in camera coordinates, that is the depth of the point in front of the eye.

So the relative sizes of a_4 and b_4 lie at the heart of perspective foreshortening of a line segment: they report the “depths” of A and B , respectively, along the camera’s viewplane normal. If A and B have the same depth (i.e. they lie in a plane parallel to the camera’s viewplane), there is no perspective distortion along the segment, so g and f are indeed the same. Figure 8.46 shows in cross section how rays from the eye through evenly spaced spots (those with equal increments in f) on the viewplane correspond to unevenly spaced spots on the original face in 3D. For the case shown A is closer than B , causing $a_4 < b_4$, so the g -increments grow in size moving across the face from A to B .



Figure 8.46. The values of a_4 and b_4 are related to the depths of points.

Rendering incrementally.

We now put these ingredients together and find the proper texture coordinates (s, t) at each point on the face being rendered. Figure 8.47 shows a face of the barn being rendered. The left edge of the face has endpoints a and b . The face extends from x_{left} to x_{right} across scan-line y . We need to find appropriate texture coordinates $(s_{\text{left}}, t_{\text{left}})$ and $(s_{\text{right}}, t_{\text{right}})$ to attach to x_{left} and x_{right} , respectively, which we can then interpolate across the scan-line. Consider finding $s_{\text{left}}(y)$, the value of s_{left} at scan-line y .

We know that texture coordinate s_A is attached to point a , and s_B is attached to point b , since these values have been passed down the pipeline along with the vertices A and B . If the scan-line at y is fraction f of the way between y_{bott} and y_{top} (so that $f = (y - y_{\text{bott}})/(y_{\text{top}} - y_{\text{bott}})$), then we know from Equation 8.19 that the proper texture coordinate to use is:

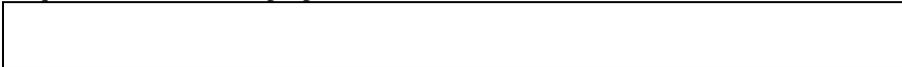


Figure 8.47. Rendering the texture on a face.

$$s_{\text{left}}(y) = \frac{\text{lerp}\left(\frac{s_A}{a_4}, \frac{s_B}{b_4}, f\right)}{\text{lerp}\left(\frac{1}{a_4}, \frac{1}{b_4}, f\right)} \quad (8.20)$$

and similarly for t_{left} . Notice that s_{left} and t_{left} have the same denominator: a linear interpolation between values $1/a_4$ and $1/b_4$. The numerator terms are linear interpolations of texture coordinates which have been divided by a_4 and b_4 . This is sometimes called “rational linear” rendering [heckbert91] or “hyperbolic interpolation” [blinn92]. To calculate (s, t) efficiently as f advances we need to store values of s_A/a_4 , s_B/b_4 , t_A/a_4 , t_B/b_4 , $1/a_4$, and $1/b_4$, as these don’t change from pixel to pixel. Both the numerator and denominator terms can be found incrementally for each y , just as we did for Gouraud shading (see Equation 8.15). But to find s_{left} and t_{left} we must still perform an explicit division at each value of y .

The pair $(s_{\text{right}}, t_{\text{right}})$ is calculated in a similar fashion. They have denominators that are based on values of a'_4 and b'_4 that arise from the projected points a' and b' .

Once $(s_{\text{left}}, t_{\text{left}})$ and $(s_{\text{right}}, t_{\text{right}})$ have been found the scan-line can be filled. For each x from x_{left} to x_{right} the values s and t are found, again by hyperbolic interpolation. (what is the expression for s at x ?)

Implications for the graphics pipeline.

What are the implications of having to use hyperbolic interpolation to render texture properly? And does the clipping step need any refinement? As we shall see, we must send certain additional information down the pipeline, and calculate slightly different quantities than supposed so far.

Figure 8.48 shows a refinement of the pipeline. Various points are labeled with the information that is available at that point. Each vertex V is associated with a texture pair (s, t) as well as a vertex normal. The vertex is transformed by the modelview matrix (and the normal is multiplied by the inverse transpose of this matrix), producing vertex $A = (A_1, A_2, A_3)$ and a normal \mathbf{n}' in eye coordinates. Shading calculations are done using this normal, producing the color $\mathbf{c} = (c_r, c_g, c_b)$. The texture coordinates (s_A, t_A) (which are the same as (s, t)) are still attached to A . Vertex A then undergoes the perspective transformation, producing $\tilde{a} = (a_1, a_2, a_3, a_4)$. The texture coordinates and color \mathbf{c} are not altered.



Figure 8.48. Refinement of the graphics pipeline to include hyperbolic interpolation.

Now clipping against the view volume is done, as discussed in Chapter 7. As the figure suggests, this can cause some vertices to disappear and others to be formed. When a vertex such as D is created we must determine its position (d_1, d_2, d_3, d_4) and attach to it the appropriate color and texture point. By the nature of the clipping algorithm the position components d_i are formed by linear interpolation: $d_i = \text{lerp}(a_i, b_i, t)$, for $i = 1, \dots, 4$, for some t . Notice that the fourth component d_4 is also formed this way. It is natural to use linear interpolation here also to form both the color components and the texture coordinates. (The rationale for this is discussed in the exercises.) Therefore after clipping the face still consists of a number of vertices, and to each is attached a color and a texture point. For point A the information is stored in the array $(a_1, a_2, a_3, a_4, s_A, t_A, \mathbf{c}, 1)$. A final term of 1 has been appended: we will use it in the next step.

Now perspective division is done. Since for hyperbolic interpolation we need terms such as s_A/a_4 and $1/a_4$ (see Equation 8.20) we divide *every* item in the array that we wish to interpolate hyperbolically by a_4 to obtain the array $(x, y, z, 1, s_A/a_4, t_A/a_4, \mathbf{c}, 1/a_4)$. (We could also divide the color components in order to obtain slightly more realistic Gouraud shading. See the exercises.) The

first three, $(x, y, z) = (a_1/a_4, a_2/a_4, a_3/a_4)$ report the position of the point in normalized device coordinates. The third component is pseudodepth. The first two components are scaled and shifted by the viewport transformation. To simplify notation we shall continue to call the screen coordinate point (x, y, z) .

So finally the renderer receives the array $(x, y, z, 1, s_A/a_4, t_A/a_4, \mathbf{c}, 1/a_4)$ for each vertex of the face to be rendered. Now it is simple to render texture using hyperbolic interpolation as in Equation 8.20: the required values s_A/a_4 and $1/a_4$ are available for each vertex.

Practice exercises.

8.5.1. Data structures for mesh models with textures. Discuss the specific data types needed to represent mesh objects in the two cases:

- a). a different texture is to be applied to each face;
- b). a single texture is to be “wrapped” around the entire mesh.

Draw templates for the two data types required, and for each show example data in the various arrays when the mesh holds a cube.

8.5.2. Pseudodepth calculations are correct. Show that it is correct, as claimed in Section 8.4, to use linear (rather than hyperbolic) interpolation when finding pseudodepth. Assume point A projects to a , and B projects to b . With linear interpolation we compute pseudodepth at the projected point $lerp(a, b, f)$ as the third component of this point. This is the correct thing to do only if the resulting value equals the true pseudodepth of the point that $lerp(A, B, g)$ (for the appropriate g) projects to. Show that it is in fact correct. Hint: Apply Equations 8.16 and 8.17 to the third component of the point being projected.

8.5.3. Wrapping and clamping textures in OpenGL. To make the pattern “wrap” or “tile” in the s direction use: `glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)`. Similarly use `GL_TEXTURE_WRAP_T` for wrapping in the t -direction. This is actually the default, so you needn't do this explicitly. To turn off tiling replace `GL_REPEAT` with `GL_CLAMP`. Refer to the OpenGL documentation for more details, and experiment with different OpenGL settings to see their effect.

8.5.4. Rationale for linear interpolation of texture during clipping. New vertices are often created when a face is clipped against the view volume. We must assign texture coordinates to each vertex. Suppose a new vertex V is formed that is fraction f of the way from vertex A to vertex B on a face. Further suppose that A is assigned texture coordinates (s_A, t_A) , and similarly for B . Argue why, if a texture is considered as “pasted” onto a flat face, it makes sense to assign texture coordinates $(lerp(s_A, s_B, f), lerp(t_A, t_B, f))$ to V .

8.5.5. Computational burden of hyperbolic interpolation. Compare the amount of computation required to perform hyperbolic interpolation versus linear interpolation of texture coordinates. Assume multiplication and division each require 10 times as much time as addition and subtraction.

8.5.3. What does the texture modulate?

How are the values in a texture map “applied” in the rendering calculation? We examine three common ways to use such values in order to achieve different visual effects. We do it for the simple case of the gray scale intensity calculation of Equation 8.5. For full color the same calculations are applied individually for the red, green, and blue components.

1). Create a glowing object.

This is the simplest method computationally. The visible intensity I is set equal to the texture value at each spot:

$$I = texture(s, t)$$

(or to some constant multiple of it). So the object appears to emit light or glow: lower texture values emit less light and higher texture values emit more light. No additional lighting calculations need be done.

(For colored light the red, green, and blue components are set separately: for instance, the red component is $I_r = texture_r(s, t)$.)

To cause OpenGL to do this type of texturing, specify:

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
```

2). Paint the texture by modulating the reflection coefficient.

We noted earlier that the color of an object is the color of its diffuse light component (when bathed in white light). Therefore we can make the texture appear to be painted onto the surface by varying the diffuse reflection coefficient, and perhaps the ambient reflection coefficient as well. We say that the texture function “modulates” the value of the reflection coefficient from point to point. Thus we replace Equation 8.5 with:

$$I = texture(s, t)[I_a \rho_a + I_d \rho_d \times lambert] + I_s \rho_s \times phong^f$$

for appropriate values of s and t . Since Phong specular reflections are the color of the source rather than the object, highlights do not depend on the texture.

To cause OpenGL to do this type of texturing, specify:

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

3). Simulate roughness by Bump Mapping.

Bump mapping is a technique developed by Blinn [blinn78] to give a surface a wrinkled (like a raisin) or dimpled (like an orange) appearance without struggling to model each dimple itself. Here the texture function is used to perturb the surface normal vector, which causes perturbations in the amount of diffuse and specular light. Figure 8.49 shows one example, and Plate ??? shows another. One problem associated with bump mapping is that since the model itself does not contain the dimples, the object’s silhouette doesn’t show dimples either, but is perfectly smooth along each face. In addition, the corner between two adjacent faces is also visible in the silhouette. This can be seen in the example.

screen shot – bump mapping on a buckyball, showing some (smooth) edges in silhouette

Figure 8.49. An apparently dimpled surface, achieved by bump mapping.

The goal is to make a scalar function $texture(s, t)$ perturb the normal vector at each spot in a controlled fashion. In addition, the perturbation should depend only on the surface shape and the texture itself, and not on the orientation of the object or position of the eye. If it depended on orientation, the dimples would change as the object moved in an animation, contrary to the desired effect.

Figure 8.50 shows in cross section how bump mapping works. Suppose the surface is represented parametrically by the function $P(u, v)$, and has unit normal vector $\mathbf{m}(u, v)$. Suppose further that the 3D point at (u^*, v^*) corresponds to the texture at (u^*, v^*) . Blinn’s method simulates perturbing the position of the true surface in the direction of the normal vector by an amount proportional to $texture(u^*, v^*)$:

a). b).

Figure 8.50. On the nature of bump mapping.

$$P'(u^*, v^*) = P(u^*, v^*) + texture(u^*, v^*) \mathbf{m}(u^*, v^*) \tag{8.21}$$

as shown in Figure 8.50a, which adds undulations and wrinkles in the surface. This perturbed surface has a new normal vector $\mathbf{m}'(u^*, v^*)$ at each point. The idea is to use this perturbed normal as if it were “attached” to the original unperturbed surface at each point, as shown in Figure 8.50b. Blinn shows that a good approximation to the $\mathbf{m}'(u^*, v^*)$ (before normalization) is given by:

$$\mathbf{m}'(u^*, v^*) = \mathbf{m}(u^*, v^*) + \mathbf{d}(u^*, v^*) \tag{8.22}$$

where the perturbation vector \mathbf{d} is given by

$$\mathbf{d}(u^*, v^*) = (\mathbf{m} \times P_v) texture_u - (\mathbf{m} \times P_u) texture_v$$

⁸ Use either GL_REPLACE or GL_DECAL.

where $texture_u$ and $texture_v$ are partial derivatives of the texture function with respect to u and v respectively. Further, P_u and P_v are partial derivative of $P(u, v)$ with respect to u and v , respectively. All functions are evaluated at (u^*, v^*) . Derivations of this result may also be found in [watt2, miller98]. Note that the perturbation function depends only on the partial derivatives of $texture()$, not on $texture()$ itself.

If a mathematical expression is available for $texture()$ you can form its partial derivatives analytically. For example, $texture()$ might undulate in two directions by combining sinewaves, as in: $texture(u, v) = \sin(au)\sin(bv)$ for some constants a and b . If the texture comes instead from an image array, linear interpolation can be used to evaluate it at (u^*, v^*) , and finite differences can be used to approximate the partial derivatives.

8.5.4. A Texturing Example using OpenGL.

To illustrate how to invoke the texturing tools that OpenGL provides, we show an application that displays a rotating cube having different images painted on its six sides. Figure 8.51 shows a snapshot from the animation created by this program.

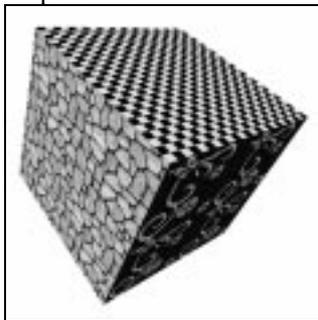


Figure 8.51. The textured cube generated by the example code.

The code for the application is shown in Figure 8.52. It uses a number of OpenGL functions to establish the six textures and to attach them to the walls of the cube. There are many variations of the parameters shown here that one could use to map textures. The version shown works well, but careful adjustment of some parameters (using the OpenGL documentation as a guide) can improve the images or increase performance. We discuss only the basics of the key routines.

One of the first tasks when adding texture to pictures is to create a **pixmap** of the texture in memory. OpenGL uses textures that are stored in “pixel maps”, or **pixmaps** for short. These are discussed in depth in Chapter 10, and the class `RGBpixmap` is developed that provides tools for creating and manipulating **pixmaps**. Here we view a **pixmap** as a simple array of pixel values, each pixel value being a triple of bytes to hold the red, green, and blue color values:

```
class RGB{ // holds a color triple - each with 256 possible intensities
public: unsigned char r,g,b;
};
```

The `RGBpixmap` class stores the number of rows and columns in the **pixmap**, as well as the address of the first pixel in memory:

```
class RGBpixmap{
public:
    int nRows, nCols; // dimensions of the pixmap
    RGB* pixel; // array of pixels
    int readBMPFile(char * fname); // read BMP file into this pixmap
    void makeCheckerboard();
    void setTexture(GLuint textureName);
};
```

We show it here as having only three methods that we need for mapping textures. Other methods and details are discussed in Chapter 10. The method `readBMPFile()` reads a BMP file⁹ and stores the pixel values in its `Pixmap` object; it is detailed in Appendix 3. The other two methods are discussed next.

Our example OpenGL application will use six textures. To create them we first make an `RGBPixmap` object for each:

```
RGBPixmap pix[6]; // create six (empty) pixmaps
```

and then load the desired texture image into each one. Finally each one is passed to OpenGL to define a texture.

1). Making a procedural texture.

We first create a checkerboard texture using the method `makeCheckerboard()`. The checkerboard pattern is familiar and easy to create, and its geometric regularity makes it a good texture for testing correctness. The application generates a checkerboard `Pixmap` in `pix[0]` using: `pix[0].makeCheckerboard()`.

The method itself follows:

```
void RGBPixmap:: makeCheckerboard()
{ // make checkerboard patten
  nRows = nCols = 64;
  pixel = new RGB[3 * nRows * nCols];
  if(!pixel){cout << "out of memory!";return;}
  long count = 0;
  for(int i = 0; i < nRows; i++)
    for(int j = 0; j < nCols; j++)
    {
      int c = (((i/8) + (j/8)) %2) * 255; 10
      pixel[count].r = c; // red
      pixel[count].g = c; // green
      pixel[count++].b = 0; // blue
    }
}
```

It creates a 64 by 64 pixel array, where each pixel is an RGB triple. OpenGL requires that texture pixel maps have a width and height that are both some power of two. The pixel map is laid out in memory as one long array of bytes: row by row from bottom to top, left to right across a row. Here each pixel is loaded with the value $(c, c, 0)$, where c jumps back and forth between 0 and 255 every 8 pixels. (We used a similar “jumping” method in Exercise 2.3.1.) The two colors of the checkerboard are black: $(0,0,0)$, and yellow: $(255,255,0)$. The function returns the address of the first pixel of the `Pixmap`, which is later passed to `glTexImage2D()` to create the actual texture for OpenGL.

Once the pixel map has been formed, we must bind it to a unique integer “name” so that it can be referred to in OpenGL without ambiguity. We arbitrarily assign the names 2001, 2002, ..., 2006 to our six textures in this example¹¹. The texture is created by making certain calls to OpenGL, which we encapsulate in the method:

```
void RGBPixmap :: setTexture(GLuint textureName)
{
  glBindTexture(GL_TEXTURE_2D,textureName);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,GL_NEAREST);
}
```

⁹ This is a standard device-independent image file format from Microsoft. Many images are available on the internet in BMP format, and tools are readily available on the internet to convert other image formats to BMP files.

¹⁰ A faster way that uses C++’s bit manipulation operators is $c = ((i \& 8) ^ (j \& 8)) * 255;$

¹¹ To avoid overlap in (integer) names in an application that uses many textures, it is better to let OpenGL supply unique names for textures using `glGenTextures()`. If we need six unique names we can build an array to hold them: `GLuint name[6];` and then call `glGenTextures(6,name)`. OpenGL places six heretofore unused integers in `name[0], ..., name[5]`, and we subsequently refer to the i -th texture using `name[i]`.

Suppose that we want to wrap a label about a circular cylinder, as suggested in Figure 8.53a. It's natural to think in terms of cylindrical coordinates. The label is to extend from θ_a to θ_b in azimuth and from z_a to z_b along the z -axis. The cylinder is modeled as a polygonal mesh, so its walls are rectangular strips as shown in part b. For vertex V_i of each face we must find suitable texture coordinates (s, t) , so that the correct "slice" of the texture is mapped onto the face.



Figure 8.53. Wrapping a label around a cylinder.

The geometry is simple enough here that a solution is straightforward. There is a direct linear relationship between (s, t) and the azimuth and height (θ, z) of a point on the cylinder's surface:

$$s = \frac{\theta - \theta_a}{\theta_b - \theta_a}, \quad t = \frac{z - z_a}{z_b - z_a} \quad (8.23)$$

So if there are N faces around the cylinder, the i -th face has left edge at azimuth $\theta_i = 2\pi i/N$, and its upper left vertex has texture coordinates $(s_i, t_i) = ((2\pi i/N - \theta_a)/(\theta_b - \theta_a), 1)$. Texture coordinates for the other three vertices follow in a similar fashion. This association between (s, t) and the vertices of each face is easily put in a loop in the modeling routine (see the exercises).

Things get more complicated when the object isn't a simple cylinder. We see next how to map texture onto a more general surface of revolution.

Example 8.5.2. "Shrink wrapping" a label onto a Surface of Revolution.

Recall from Chapter 6 that a surface of revolution is defined by a profile curve $(x(v), z(v))^{12}$ as shown in Figure 8.54a, and the resulting surface - here a vase - is given parametrically by $P(u, v) = (x(v) \cos u, x(v) \sin u, z(v))$. The shape is modeled as a collection of faces with sides along contours of constant u and v (see Figure 8.54b). So a given face F_i has four vertices $P(u_i, v_i)$, $P(u_{i+1}, v_i)$, $P(u_i, v_{i+1})$, and $P(u_{i+1}, v_{i+1})$. We need to find the appropriate (s, t) coordinates for each of these vertices.

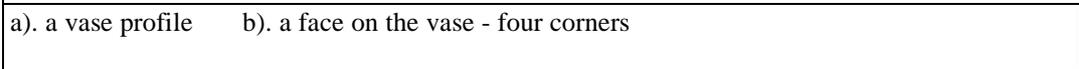


Figure 8.54. Wrapping a label around a vase.

One natural approach is to proceed as above and to make s and t vary linearly with u and v in the manner of Equation 8.23. This is equivalent to wrapping the texture about an imaginary rubber cylinder that encloses the vase (see Figure 8.55a), and then letting the cylinder collapse, so that each texture point slides radially (and horizontally) until it hits the surface of the vase. This method is called "shrink wrapping" by Bier and Sloane [bier86], who discuss several possible ways to map texture onto different classes of shapes. They view shrink wrapping in terms of the imaginary cylinder's normal vector (see Figure 8.55b): texture point P_i is associated with the object point V_i that lies along the normal from P_i .



Figure 8.55. Shrink wrapping texture onto the vase.

Shrink wrapping works well for cylinder-like objects, although the texture pattern will be distorted if the profile curve has a complicated shape.

Bier and Sloane suggest some alternate ways to associate texture points on the imaginary cylinder with vertices on the object. Figure 8.56 shows two other possibilities.

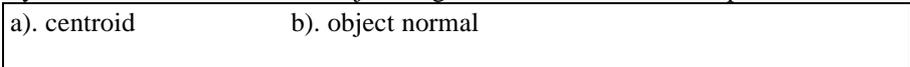


Figure 8.56. Alternative mappings from the imaginary cylinder to the object.

¹² We revert to calling the parameters u and v in the parametric representation of the shape, since we are using s and t for the texture coordinates.

In part a) a line is drawn from the object's centroid C , through the vertex V_i , to its intersection with the cylinder P_i . And in part b) the normal vector to the object's surface at V_i is used: P_i is at the intersection of this normal from V_i with the cylinder. Notice that these three ways to associate texture points with object points can lead to very different results depending on the shape of the object (see the exercises). The designer must choose the most suitable method based on the object's shape and the nature of the texture image being mapped. (What would be appropriate for a chess pawn?)

Example 8.5.3. Mapping texture onto a sphere.

It was easy to wrap a texture rectangle around a cylinder: topologically a cylinder can be sliced open and laid flat without distortion. A sphere is a different matter. As all map makers know, there is no way to show accurate details of the entire globe on a flat piece of paper: if you slice open a sphere and lay it flat some parts always suffer serious stretching. (Try to imagine a checkerboard mapped over an entire sphere!)

It's not hard to paste a rectangular texture image onto a *portion* of a sphere, however. To map the texture square to the portion lying between azimuth θ_a to θ_b and latitude ϕ_a to ϕ_b just map linearly as in Equation 8.23: if vertex V_i lies a (θ_i, ϕ_i) associate it with texture coordinates $(s_i, t_i) = ((\theta_i - \theta_a)/(\theta_b - \theta_a), (\phi_i - \phi_a)/(\phi_b - \phi_a))$. Figure 8.57 shows an image pasted onto a band around a sphere. Only a small amount of distortion is seen.

a). texture on portion of sphere b). 8 maps onto 8 octants

Figure 8.57. Mapping texture onto a sphere.

Figure 8.57b shows how one might cover an entire sphere with texture: map eight triangular texture maps onto the eight octants of the sphere.

Example 8.5.4. Mapping texture to sphere-like objects.

We discussed adding texture to cylinder-like objects above. But some objects are more sphere-like than cylinder-like. Figure 8.58a shows the buckyball, whose faces are pentagons and hexagons. One could devise a number of pentagonal and hexagonal textures and manually paste one of each face, but for some scenes it may be desirable to wrap the whole buckyball in a single texture.

a). buckyball b). three mapping methods

Figure 8.58. Sphere-like objects.

It is natural to surround a sphere-like object with an imaginary sphere (rather than a cylinder) that has texture pasted to it, and use one of the association methods discussed above. Figure 8.58b shows the buckyball surrounded by such a sphere in cross section. The three ways of associating texture points P_i with object vertices V_i are sketched:

- object-centroid: P_i is on a line from the centroid C through vertex V_i ;
- object-normal: P_i is the intersection of a ray from V_i in the direction of the face normal;
- sphere-normal: V_i is the intersection of a ray from P_i in the direction of the normal to the sphere at P_i .

(Question: Are the object-centroid and sphere-normal methods the same if the centroid of the object coincides with the center of the sphere?) The object centroid method is most likely the best, and it is easy to implement. As Bier and Sloane argue, the other two methods usually produce unacceptable final renderings.

Bier and Sloane also discuss using an imaginary box rather than a sphere to surround the object in question. Figure 8.59a shows the six faces of a cube spread out over a texture image, and part b) shows the texture wrapped about the cube, which in turn encloses an object. Vertices on the object can be associated with texture points in the three ways discussed above: the object-centroid and cube-normal are probably the best choices.

a). texture on 6 faces of box b). wrapping texture onto

Figure 8.59. Using an enclosing box.

Practice exercises.

8.5.7. How to associate P_i and V_i . Surface of revolution S shown in Figure 8.60 consists of a sphere resting on a cylinder. The object is surrounded by an imaginary cylinder having a checkerboard texture pasted on it. Sketch how the texture will look for each of the following methods of associating texture points to vertices:

- a). shrink wrapping;
- b). object centroid;
- c). object normal;

Figure 8.60. A surface of revolution surrounded by an imaginary cylinder.

8.5.8. Wrap a texture onto a torus. A torus can be viewed as a cylinder that “bends” around and closes on itself. The torus shown in Figure 8.61 has the parametric representation given by $P(u, v) = ((D + A \cos(v)) \cos(u), (D + A \cos(v)) \sin(u), A \sin(v))$. Suppose you decide to polygonalize the torus by taking vertices based on the samples $u_i = 2\pi i/N$ and $v_j = 2\pi j/M$, and you wish to wrap some texture from the unit texture space around this torus. Write code that generates, for each of the faces, each vertex and its associated texture coordinates (s, t) .

Figure 8.61. Wrapping texture about a torus.

8.5.6. Reflection mapping.

The class of techniques known as “reflection mapping” can significantly improve the realism of pictures, particularly in animations. The basic idea is to see reflections in an object that suggest the “world” surrounding that object.

The two main types of reflection mapping are called “chrome mapping” and “environment mapping.” In the case of **chrome mapping** a rough and usually blurry image that suggests the surrounding environment is reflected in the object, as you would see in a surface coated with chrome. Television commercials abound with animations of shiny letters and logos flying around in space, where the chrome map includes occasional spotlights for dramatic effect. Figure 8.62 offers an example. Part a) shows the chrome texture, and part b) shows it reflecting in the shiny object. The reflection provides a rough suggestion of the world surrounding the object.

- | | |
|----------------------------------|-------------------------------|
| a). chrome map
(screen shots) | b). scene with chrome mapping |
|----------------------------------|-------------------------------|

Figure 8.62. Example of chrome mapping.

In the case of **environment mapping** (first introduced by Blinn and Newell [blinn 76]) a recognizable image of the surrounding environment is seen reflected in the object. We get valuable visual cues from such reflections, particularly when the object moves about. Everyone has seen the classic photographs of an astronaut walking on the moon with the moonscape reflected in his face mask. And in the movies you sometimes see close-ups of a character's reflective dark glasses, in which the world about her is reflected. Figure 8.63 shows two examples where a cafeteria is reflected in a sphere and a torus. The cafeteria texture is wrapped about a large sphere that surrounds the object, so that the texture coordinates (s, t) correspond to azimuth and latitude about the enclosing sphere.



Figure 8.63. Example of environment mapping (courtesy of Haeberli and Segal).

Figure 8.64 shows the use of a surrounding cube rather than a sphere. Part a) shows the map, consisting of six images of various views of the interior walls, floor, and ceiling of a room. Part b) shows a shiny object reflecting different parts of the room. The use of an enclosing cube was introduced by Greene [greene 86], and generally produces less distorted reflections than are seen with an enclosing sphere. The six maps can be generated by rendering six separate images from the point of view of the object (with the object itself removed, of course). For each image a synthetic camera is set up and the appropriate window is set. Alternatively, the textures can be digitized from photos taken by a real camera that looks in the six principal directions inside an actual room or scene.



Figure 8.64. Environment mapping based on a surrounding cube.

Chrome and environment mapping differ most dramatically from normal texture mapping in an animation when the shiny object is moving. The reflected image will “flow” over the moving object, whereas a normal texture map will be attached to the object and move with it. And if a shiny sphere rotates about a fixed spot a normal texture map spins with the sphere, but a reflection map stays fixed.

How is environment mapping done? What you see at point P on the shiny object is what has arrived at P from the environment in just the right direction to reflect into your eye. To find that direction trace a ray from the eye to P , and determine the direction of the reflected ray. Trace this ray to find where it hits the texture (on the enclosing cube or sphere). Figure 8.65 shows a ray emanating from the eye to point P . If the direction of this ray is \mathbf{u} and the unit normal at P is \mathbf{m} , we know from Equation 8.2 that the reflected ray has direction $\mathbf{r} = \mathbf{u} - 2(\mathbf{u} \cdot \mathbf{m})\mathbf{m}$. The reflected ray moves in direction \mathbf{r} until it hits the hypothetical surface with its attached texture. It is easiest computationally to suppose that the shiny object is centered in, and much smaller than, the enclosing cube or sphere. Then the reflected ray emanates approximately from the object’s center, and its direction \mathbf{r} can be used directly to index into the texture.



Figure 8.65. Finding the direction of the reflected ray.

OpenGL provides a tool to perform approximate environment mapping for the case where the texture is wrapped about a large enclosing sphere. It is invoked by setting a mapping mode for both s and t using:

```
glTexGenf(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGenf(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

Now when a vertex P with its unit normal \mathbf{m} is sent down the pipeline, OpenGL calculates a texture coordinate pair (s, t) suitable for indexing into the texture attached to the surrounding sphere. This is done for each vertex of the face on the object, and the face is drawn as always using interpolated texture coordinates (s, t) for points in between the vertices.

How does OpenGL rapidly compute a suitable coordinate pair (s, t) ? As shown in Figure 8.66a it first finds (in eye coordinates) the reflected direction \mathbf{r} (using the formula above), where \mathbf{u} is the unit vector (in eye coordinates) from the eye to the vertex V on the object, and \mathbf{m} is the normal at V .



Figure 8.66. OpenGL’s computation of the texture coordinates.

It then simply uses the expression:

$$(s, t) = \left(\frac{1}{2} \left(\frac{r_x}{p} + 1 \right), \frac{1}{2} \left(\frac{r_y}{p} + 1 \right) \right) \tag{8.24}$$

where p is a mysterious scaling factor $p = \sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$. The derivation of this term is developed in the exercises. We must precompute a texture that shows what you would see of the environment in a perfectly reflecting sphere, from an eye position far removed from the sphere [haeberli93]. This maps the part of the environment that lies in the hemisphere behind the eye into a circle in the middle of the texture, and the part of the environment in the hemisphere in front of the eye into an annulus around this circle (visualize this). This texture must be recomputed if the eye changes position. The pictures in Figure 8.63 were made using this method.

Simulating Highlights using Environment mapping.

Reflection mapping can be used in OpenGL to produce specular highlights on a surface. A texture map is created that has an intense concentrated bright spot. Reflection mapping “paints” this highlight onto the surface, making it appear to be an actual light source situated in the environment. The highlight created can be more concentrated and detailed than those created using the Phong specular term with Gouraud shading. Recall that the Phong term is computed only at the vertices of a face, and it is easy to “miss” a specular highlight that falls between two vertices. With reflection mapping the coordinates (s, t) into the texture are formed at each vertex, and then interpolated in between. So if the coordinates indexed by the vertices happen to surround the bright spot, the spot will be properly rendered inside the face.

Practice Exercise 8.5.9. OpenGL’s computation of texture coordinates for environment mapping. Derive the result in Equation 8.24. Figure 8.66b shows in cross-sectional view the vectors involved (in eye coordinates). The eye is looking from a remote location in the direction $(0,0,1)$. A sphere of radius 1 is positioned on the negative z -axis. Suppose light comes in from direction \mathbf{r} , hitting the sphere at the point (x, y, z) . The normal to the sphere at this point is (x, y, z) , which also must be just right so that light coming along \mathbf{r} is reflected into the direction $(0, 0, 1)$. This means the normal must be half-way between \mathbf{r} and $(0, 0, 1)$, or must be proportional to their sum, so $(x, y, z) = K(r_x, r_y, r_z+1)$ for some K .

- Show that the normal vector has unit length if K is $1/p$, where p is given as in Equation 8.24.
- Show that therefore $(x, y) = (r_x/p, r_y/p)$.
- Suppose for the moment that the texture image extends from -1 to 1 in x and from -1 to 1 in y . Argue why what we want to see reflected at the point (x, y, z) is the value of the texture image at (x, y) .
- Show that if instead the texture uses coordinates from 0 to 1 – as is true with OpenGL – that we want to see at (x, y) the value of the texture image at (s, t) given by Equation 8.24.

8.6. Adding Shadows of Objects.

Shadows make an image much more realistic. From everyday experience the way one object casts a shadow on another object gives important visual cues as to how they are positioned. Figure 8.67 shows two images involving a cube and a sphere suspended above a plane. Shadows are absent in part a, and it is impossible to see how far above the plane the cube and sphere are floating. By contrast, the shadows seen in part b give useful hints as to the positions of the objects. A shadow conveys a lot of information; it’s as if you are getting a second look at the object (from the viewpoint of the light source).

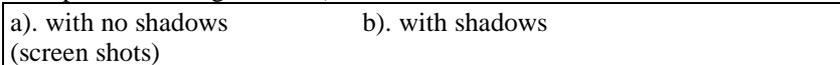


Figure 8.67. The effect on shadows.

In this section we examine two methods for computing shadows: one is based on “painting” shadows as if they were texture, and the other is an adaptation of the depth buffer approach for hidden surface removal. In Chapter 14 we see that a third method arises naturally when raytracing. There are many other techniques, well surveyed in [watt92, crow77, woo90, bergeron86].

8.6.1. Shadows as Texture.

This technique displays shadows that are cast onto a flat surface by a point light source. The problem is to compute the shape of the shadow that is cast. Figure 8.68a shows a box casting a shadow onto the floor. The shape of the shadow is determined by the projections of each of the faces of the box onto the plane of the floor, using the source as the center of projection. In fact the shadow is the union¹³ of the projections of the six faces. Figure 8.68b shows the superposed

¹³ the set theoretic union: A point is in the shadow if it is in one or more of the projections.

projections of two of the faces: the top face projects to *top*' and the front face to *front*'. (Sketch the projections of the other four faces, and see that their union is the required shadow¹⁴.)



Figure 8.68. Computing the shape of a shadow.

This is the key to drawing the shadow. After drawing the plane using ambient, diffuse, and specular light contributions, draw the six projections of the box's faces on the plane using only ambient light. This will draw the shadow in the right shape and color. Finally draw the box. (If the box is near the plane parts of it might obscure portions of the shadow.)

Building the “projected” face:

To make the new face *F'* produced by *F*, project each of its vertices onto the plane in question. We need a way to calculate these vertex positions on the plane. Suppose, as in Figure 8.68a, that the plane passes through point *A* and has normal vector **n**. Consider projecting vertex *V*, producing point *V'*. The mathematics here are familiar: Point *V'* is the point where the ray from the source at *S* through *V* hits the plane. As developed in the exercises, this point is:

$$V' = S + (V - S) \frac{\mathbf{n} \cdot (A - S)}{\mathbf{n} \cdot (V - S)} \tag{8.25}$$

The exercises show how this can be written in homogeneous coordinates as *V* times a matrix, which is handy for rendering engines, like OpenGL, that support convenient matrix multiplication.

Practice Exercises.

8.6.1. Shadow shapes. Suppose a cube is floating above a plane. What is the shape of the cube's shadow if the point source lies a). directly above the top face? b). along a main diagonal of the cube (as in an isometric view)? Sketch shadows for a sphere and for a cylinder floating above a plane for various source positions.

8.6.2. Making the “shadow” face. a). Show that the ray from the source point *S* through vertex *V* hits the plane $\mathbf{n} \cdot (P - A) = 0$ at $t^* = \mathbf{n} \cdot (A - S) / \mathbf{n} \cdot (V - S)$; b). Show that this defines the hit point *V'* as given in Equation 8.25.

8.6.3. It's equivalent to a matrix multiplication. a). Show that the expression for *V'* in Equation 8.25 can be written as a matrix multiplication: $V' = M(V_x, V_y, V_z, 1)^T$, where *M* is a 4 by 4 matrix b). Express the terms of *M* in terms of *A*, *S*, and **n**.

8.6.2. Shadows using a shadow buffer.

A rather different method for drawing shadows uses a variant of the depth buffer that performs hidden surface removal. It uses an auxiliary second depth buffer, called a **shadow buffer**, for each light source. This requires a lot of memory, but this approach is not restricted to casting shadows onto planar surfaces.

The method is based on the principle that any points in the scene that are “hidden” from the light source must be in shadow. On the other hand, if no object lies between a point and the light source the point is not in shadow. The shadow buffer contains a “depth picture” of the scene from the point of view of the light source: each of its elements records the distance from the source to the *closest* object in the associated direction.

Rendering is done in two stages:

1). **Shadow buffer loading.** The shadow buffer is first initialized with 1.0 in each element, the largest pseudodepth possible. Then, using a camera positioned at the light source, each of the faces in the scene is scan converted, but only the pseudodepth of the point on the face is tested. Each element of the shadow buffer keeps track of the smallest pseudodepth seen so far.

To be more specific, Figure 8.69 shows a scene being viewed by the usual “eye camera” as well as a “source camera” located at the light source. Suppose point *P* is on the ray from the source through

¹⁴ You need to form the union of the projections of only the three “front” faces: those facing toward the light source. (Why?)

shadow buffer “pixel” $d[i][j]$, and that point B on the pyramid is also on this ray. If the pyramid is present $d[i][j]$ contains the pseudodepth to B ; if it happens to be absent $d[i][j]$ contains the pseudodepth to P .



Figure 8.69. Using the shadow buffer.

Note that the shadow buffer calculation is independent of the eye position, so in an animation where only the eye moves the shadow buffer is loaded only once. The shadow buffer must be recalculated, however, whenever the objects move relative to the light source.

2). **Render the scene.** Each face in the scene is rendered using the eye camera as usual. Suppose the eye camera “sees” point P through pixel $p[c][r]$. When rendering $p[c][r]$ we must find¹⁵:

- the pseudodepth D from the source to P ;
- the index location $[i][j]$ in the shadow buffer that is to be tested;
- the value $d[i][j]$ stored in the shadow buffer.

If $d[i][j]$ is less than D the point P is in shadow, and $p[c][r]$ is set using only ambient light. Otherwise P is not in shadow and $p[c][r]$ is set using ambient, diffuse, and specular light.

How are these steps done? As described in the exercises, to each point on the eye camera viewplane there corresponds a point on the source camera viewplane¹⁶. For each screen pixel this correspondence is invoked to find the pseudodepth from the source to P as well as the index $[i][j]$ that yields the minimum pseudodepth stored in the shadow buffer.

Practice Exercises.

8.6.4. Finding pseudodepth from the source. Suppose the matrices M_c and M_s map the point P in the scene to the appropriate (3D) spots on the eye camera’s viewplane and the source camera’s viewplane, respectively. a). Describe how to establish a “source camera” and how to find the resulting matrix M_s . b). Find the transformation that, given position (x, y) on the eye camera’s viewplane produces the position (i, j) and pseudodepth on the source camera’s viewplane. c). Once (i, j) are known, how is the index $[i][j]$ and the pseudodepth of P on the source camera determined?

8.6.5. Extended Light sources. We have considered only point light sources in this chapter. Greater realism is provided by modeling extended light sources. As suggested in Figure 8.70a such sources cast more complicated shadows, having an **umbra** within which no light from the source is seen, and a lighter **penumbra** within which a part of the source is visible. In part b) a glowing sphere of radius 2 shines light on a unit cube, thereby casting a shadow on the wall W . Make an accurate sketch of the umbra and penumbra that is observed on the wall. As you might expect, algorithms for rendering shadows due to extended light sources are complex. See [watt92] for a thorough treatment.

a). umbra and penumbra b). example to sketch

Figure 8.70. Umbra and penumbra for extended light sources.

8.7. Summary

Since the beginning of computer graphics there has been a relentless quest for greater realism when rendering 3D scenes. Wireframe views of objects can be drawn very rapidly but are difficult to interpret, particularly if several objects in a scene overlap. Realism is greatly enhanced when the faces are filled with some color and surfaces that should be hidden are removed, but pictures rendered this way still do not give the impression of objects residing in a scene, illuminated by light sources.

¹⁵ Of course, this test is made only if P is closer to the eye than the value stored in the normal depth buffer of the eye camera.

¹⁶ Keep in mind these are 3D points: 2 position coordinates on the viewplane, and pseudodepth.

What is needed is a shading model, that describes how light reflects off a surface depending on the nature of the surface and its orientation to both light sources and the camera's eye. The physics of light reflection is very complex, so programmers have developed a number of approximations and tricks that do an acceptable job most of the time, and are reasonably efficient computationally. The model for the diffuse component is the one most closely based on reality, and becomes extremely complex as more and more ingredients are considered. Specular reflections are not modeled on physical principles at all, but can do an adequate job of recreating highlights on shiny objects. And ambient light is purely an abstraction, a shortcut that avoids dealing with multiple reflections from object to object, and prevents shadows from being too deep.

Even simple shading models involve several parameters such as reflection coefficients, descriptions of a surface's roughness, and the color of light sources. OpenGL provides ways to set many of these parameters. There is little guidance for the designer in choosing the values of these parameters; they are often determined by trial and error until the final rendered picture looks right.

In this chapter we focused on rendering of polygonal mesh models, so the basic task was to render a polygon. Polygonal faces are particularly simple and are described by a modest amount of data, such as vertex positions, vertex normals, surface colors and material. In addition there are highly efficient algorithms for filling a polygonal face with calculated colors, especially if it is known to be convex. And algorithms can capitalize on the flatness of a polygon to interpolate depth in an incremental fashion, making the depth buffer hidden surface removal algorithm simple and efficient.

When a mesh model is supposed to approximate an underlying smooth surface the appearance of a face's edges can be objectionable. Gouraud and Phong shading provide ways to draw a smoothed version of the surface (except along silhouettes). Gouraud shading is very fast but does not reproduce highlights very faithfully; Phong shading produces more realistic renderings but is computationally quite expensive.

The realism of a rendered scene is greatly enhanced by the appearance of texturing on object surfaces. Texturing can make an object appear to be made of some material such as brick or wood, and labels or other figures can be pasted onto surfaces. Texture maps can be used to modulate the amount of light that reflects from an object, or as "bump maps" that give a surface a bumpy appearance. Environment mapping shows the viewer an impression of the environment that surrounds a shiny object, and this can make scenes more realistic, particularly in animations. Texture mapping must be done with care, however, using proper interpolation and antialiasing (as we discuss in Chapter 10).

The chapter closed with a description of some simple methods for producing shadows of objects. This is a complex subject, and many techniques have been developed. The two algorithms described provide simple but partial solutions to the problem.

Greater realism can be attained with more elaborate techniques such as ray tracing and radiosity. Chapter 14 develops the key ideas of these techniques.

8.8. Case Studies.

8.8.1. Case Study 8.1. Creating shaded objects using OpenGL

(Level of Effort: II beyond that of Case Study 7.1). Extend Case Study 7.1 that flies a camera through space looking at various polygonal mesh objects. Extend it by establishing a point light source in the scene, and assigning various material properties to the meshes. Include ambient, diffuse, and specular light components. Provide a keystroke that switches between flat and smooth shading

8.8.2. Case Study 8.2. The Do-it-yourself graphics pipeline.

(Level of Effort: III) Write an application that reads a polygonal mesh model from a file as described in Chapter 6, defines a camera and a point light source, and renders the mesh object using flat shading with ambient and diffuse light contributions. Only gray scale intensities need be computed. For this project do *not* use OpenGL's pipeline; instead create your own. Define modelview, perspective, and viewport matrices. Arrange that vertices can be passed through the

first two matrices, have the shading model applied, followed by perspective division (no clipping need be done) and by the viewport transformation. Each vertex emerges as the array $\{x, y, z, b\}$ where x and y are screen coordinates, z is pseudodepth, and b is the grayscale brightness of the vertex. Use a tool that draws filled polygons to do the actual rendering: if you use OpenGL, use only its 2D drawing (and depth buffer) components. Experiment with different mesh models, camera positions, and light sources to insure that lighting is done properly.

8.8.3. Case Study 8.3. Add Polygon Fill and Depth Buffer HSR.

(Level of Effort: III beyond that needed for Case Study 8.2.) Implement your own depth buffer, and use it in the application of Case Study 8.2. This requires the development of a polygon fill routine as well - see Chapter 10.

8.8.4. Case Study 8.4. Texture Rendering.

(Level of Effort: II beyond that of Case Study 8.1). Enhance the program of Case Study 8.1 so that textures can be painted on the faces of the mesh objects. Assemble a routine that can read a BMP image file and attach it to an OpenGL texture object. Experiment by putting five different image textures and one procedural texture on the sides of a cube, and arranging to have the cube rotate in an animation. Provide a keystroke that lets the user switch between linear interpolation and correct interpolation for rendering textures.

8.8.5. Case Study 8.5. Applying Procedural 3D textures.

(Level of Effort: III) An interesting effect is achieved by making an object appear to be carved out of some solid material, such as wood or marble. Plate ??? shows a (raytraced) vase carved out of marble, and Plate ??? shows a box apparently made of wood. 3D textures are discussed in detail in Chapter 14 in connection with ray tracing, but it is also possible to map "slices" of a 3D texture onto the surfaces of an object, to achieve a convincing effect.

Suppose you have a texture function $B(x, y, z)$ which attaches different intensities or colors to different points in 3D space. For instance $B(x, y, z)$ might represent how "inky" the sea is at position (x, y, z) . As you swim around you encounter a varying inkiness right before your eyes. If you freeze a block of this water and carve some shape out of the block, the surface of the shape will exhibit a varying inkiness. $B()$ can be vector-valued as well: providing three values at each (x, y, z) , which might represent the diffuse reflection coefficients for red, green, and blue light of the material at each point in space. It's not hard to construct interesting functions $B()$:

a). A 3D black and white checkerboard with 125 blocks is formed using:

$$B(x, y, z) = ((\text{int})(5x) + (\text{int})(5y) + (\text{int})(5z)) \% 2 \text{ as } x, y, z \text{ vary from } 0 \text{ to } 1.$$

b). A "color cube" has six different colors at its vertices, with a continuously varying color at points in between. Just use $B(x, y, z) = (x, y, z)$ where $x, y,$ and z vary from 0 to 1. The vertex at $(0, 0, 0)$ is black, that at $(1, 0, 0)$ is red, etc.

c). All of space can be filled with such cubes stacked upon one another using $B(x, y, z) = (\text{fract}(x), \text{fract}(y), \text{fract}(z))$ where $\text{fract}(x)$ is the fractional part of the value x .

Methods for creating wood grain and turbulent marble are discussed in Chapter 14. They can be used here as well.

In the present context we wish to paste such texture onto surfaces. To do this a bitmap is computed using $B()$ for each surface of the object. If the object is a cube, for instance, six different bitmaps are computed, one for each face of the cube. Suppose a certain face of the cube is characterized by the planar surface $P + \mathbf{a}t + \mathbf{b}s$ for s, t in 0 to 1. Then use as texture $B(P_x + a_x t + b_x s, P_y + a_y t + b_y s, P_z + a_z t + b_z s)$. Notice that if there is any coherence to the pattern $B()$ (so nearby points enjoy somewhat the same inkiness or color), then nearby points on adjacent faces of the cube will also have nearly the same color. This makes the object truly look like it is carved out of a single solid material.

Extend Case Study 8.4 to include pasting texture like this onto the faces of a cube and an icosahedron. Use a checkerboard texture, a color cube texture, and a wood grain texture (as described in Chapter 14).

Form a sequence of images of a textured cube, where the cube moves slightly through the material from frame to frame. The object will appear to “slide through” the texture in which it is imbedded. This gives a very different effect from an object moving with its texture attached. Experiment with such animations.

8.8.6. Case Study 8.6. Drawing Shadows.

(Level of Effort:III) Extend the program of Case Study 8.1 to produce shadows. Make one of the objects in the scene a flat planar surface, on which is seen shadows of other objects. Experiment with the “projected faces” approach. If time permits, develop as well the shadow buffer approach.

8.8.7. Case Study 8.7. Extending SDL to Include Texturing.

(Level of Effort:III) The SDL scene description language does not yet include a means to specify the texture that one wants applied to each face of an object. The keyword `texture` is currently in SDL, but does nothing when encountered in a file. Do a careful study of the code in the Scene and Shape classes, available on the book’s internet site, and design an approach that permits a syntax such as

```
texture giraffe.bmp p1 p2 p3 p4
```

to create a texture from a stored image (here `giraffe.bmp`) and paste it onto certain faces of subsequently defined objects. Determine how many parameters `texture` should require, and how they should be used. Extend `drawOpenGL()` for two or three shapes so that it properly pastes such texture onto the objects in question.

8.9. For Further Reading

Jim Blinn’s two JIM BLINN’S CORNER books: A TRIP DOWN THE GRAPHICS PIPELINE [blinn96] and DIRTY PIXELS [blinn98] offer several articles that lucidly explain the issues of drawing shadows and the hyperbolic interpolation used in rendering texture. Heckbert’s “Survey of Texture Mapping” [heckbert86] gives many interesting insights into this difficult topic. The papers “Fast Shadows and Lighting Effects Using Texture Mapping” by Segal et al [segal92] and “Texture Mapping as a Fundamental Drawing Primitive” by Haeberli and Segal [haeberli93] (also available on-line: <http://www.sgi.com/grafica/texmap/>) provide excellent background and context.