As you can see, an unlit sphere looks no different from a two-dimensional disk. This demonstrates how critical the interaction between objects and light is in creating a three-dimensional scene.

With OpenGL, you can manipulate the lighting and objects in a scene to create many different kinds of effects. This chapter begins with a primer on hidden-surface removal. Then it explains how to control the lighting in a scene, discusses the OpenGL conceptual model of lighting, and describes in detail how to set the numerous illumination parameters to achieve certain effects. Toward the end of the chapter, the mathematical computations that determine how lighting affects color are presented.

This chapter contains the following major sections:

- "A Hidden-Surface Removal Survival Kit" describes the basics of removing hidden surfaces from view.
- "Real-World and OpenGL Lighting" explains in general terms how light behaves in the world and how OpenGL models this behavior.
- "A Simple Example: Rendering a Lit Sphere" introduces the OpenGL lighting facility by presenting a short program that renders a lit sphere.
- "Creating Light Sources" explains how to define and position light sources.
- "Selecting a Lighting Model" discusses the elements of a lighting model and how to specify them.
- "Defining Material Properties" explains how to describe the properties of objects so that they interact with light in a desired way.
- "The Mathematics of Lighting" presents the mathematical calculations used by OpenGL to determine the effect of lights in a scene.
- "Lighting in Color-Index Mode" discusses the differences between using RGBA mode and color-index mode for lighting.

A Hidden-Surface Removal Survival Kit

With this section, you begin to draw shaded, three-dimensional objects, in earnest. With shaded polygons, it becomes very important to draw the objects that are closer to our viewing position and to eliminate objects obscured by others nearer to the eye.

When you draw a scene composed of three-dimensional objects, some of them might obscure all or parts of others. Changing your viewpoint can change the obscuring relationship. For example, if you view the scene from the opposite direction, any object that was previously in front of another is now behind it. To

draw a realistic scene, these obscuring relationships must be maintained. Suppose your code works like this:

```
while (1) {
   get_viewing_point_from_mouse_position();
   glClear(GL_COLOR_BUFFER_BIT);
   draw_3d_object_A();
   draw_3d_object_B();
}
```

For some mouse positions, object A might obscure object B. For others, the reverse may hold. If nothing special is done, the preceding code always draws object B second (and thus on top of object A) no matter what viewing position is selected. In a worst case scenario, if objects A and B intersect one another so that part of object A obscures object B and part of B obscures A, changing the drawing order does not provide a solution.

The elimination of parts of solid objects that are obscured by others is called *hidden-surface removal*. (Hidden-line removal, which does the same job for objects represented as wireframe skeletons, is a bit trickier and isn't discussed here. See "Hidden-Line Removal" in Chapter 14 for details.) The easiest way to achieve hidden-surface removal is to use the depth buffer (sometimes called a z-buffer). (Also see Chapter 10.)

A depth buffer works by associating a depth, or distance, from the view plane (usually the near clipping plane), with each pixel on the window. Initially, the depth values for all pixels are set to the largest possible distance (usually the far clipping plane) using the **glClear()** command with GL_DEPTH_BUFFER_BIT. Then the objects in the scene are drawn in any order.

Graphical calculations in hardware or software convert each surface that's drawn to a set of pixels on the window where the surface will appear if it isn't obscured by something else. In addition, the distance from the view plane is computed. With depth buffering enabled, before each pixel is drawn a comparison is done with the depth value already stored at the pixel. If the new pixel is closer than (in front of) what's there, the new pixel's color and depth values replace those that are currently written into the pixel. If the new pixel's depth is greater than what's currently there, the new pixel is obscured, and the color and depth information for the incoming pixel is discarded.

To use depth buffering, you need to enable depth buffering. This has to be done only once. Before drawing, each time you draw the scene, you need to clear the depth buffer and then draw the objects in the scene in any order.

To convert the preceding code example so that it performs hidden-surface removal, modify it to the following:

```
glutInitDisplayMode (GLUT_DEPTH | .... );
glEnable(GL_DEPTH_TEST);
...
while (1) {
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   get_viewing_point_from_mouse_position();
   draw_3d_object_A();
```

```
    draw_3d_object_B();
}
```

The argument to **glClear()** clears both the depth and color buffers.

Depth-buffer testing can affect the performance of your application. Since information is discarded rather than used for drawing, hidden-surface removal can increase your performance slightly. However, the implementation of your depth buffer probably has the greatest effect on performance. A "software" depth buffer (implemented with processor memory) may be much slower than one implemented with a specialized hardware depth buffer.

Real-World and OpenGL Lighting

When you look at a physical surface, your eye's perception of the color depends on the distribution of photon energies that arrive and trigger your cone cells. (See "Color Perception" in Chapter 4.) Those photons come from a light source or combination of sources, some of which are absorbed and some of which are reflected by the surface. In addition, different surfaces may have very different properties - some are shiny and preferentially reflect light in certain directions, while others scatter incoming light equally in all directions. Most surfaces are somewhere in between.

OpenGL approximates light and lighting as if light can be broken into red, green, and blue components. Thus, the color of light sources is characterized by the amount of red, green, and blue light they emit, and the material of surfaces is characterized by the percentage of the incoming red, green, and blue components that is reflected in various directions. The OpenGL lighting equations are just an approximation but one that works fairly well and can be computed relatively quickly. If you desire a more accurate (or just different) lighting model, you have to do your own calculations in software. Such software can be enormously complex, as a few hours of reading any optics textbook should convince you.

In the OpenGL lighting model, the light in a scene comes from several light sources that can be individually turned on and off. Some light comes from a particular direction or position, and some light is generally scattered about the scene. For example, when you turn on a light bulb in a room, most of the light comes from the bulb, but some light comes after bouncing off one, two, three, or more walls. This bounced light (called ambient) is assumed to be so scattered that there is no way to tell its original direction, but it disappears if a particular light source is turned off.

Finally, there might be a general ambient light in the scene that comes from no particular source, as if it had been scattered so many times that its original source is impossible to determine.

In the OpenGL model, the light sources have an effect only when there are surfaces that absorb and reflect light. Each surface is assumed to be composed of a material with various properties. A material might emit its own light (like headlights on an automobile), it might scatter some incoming light in all directions, and it might reflect some portion of the incoming light in a preferential direction like a mirror or other shiny surface.

The OpenGL lighting model considers the lighting to be divided into four independent components: emissive, ambient, diffuse, and specular. All four components are computed independently and then