Chapter 8

Lighting Objects

This chapter focuses on lighting objects, looking at different light sources and their effects on the 3D scene. Lighting is essential if you want to create realistic 3D scenes because it helps to give the scene a sense of depth.

The following key points are discussed in this chapter:

- Shading, shadows, and different types of light sources including point, directional, and ambient
- Reflection of light in the 3D scene and the two main types: diffuse and ambient
- The details of shading and how to implement the effect of light to make objects, such as the pure white cube in the previous chapter, look three-dimensional

By the end of this chapter, you will have all the knowledge you need to create lighted 3D scenes populated with both simple and complex 3D objects.

Lighting 3D Objects

When light hits an object in the real world, part of the light is reflected by the surface of the object. Only after this reflected light enters your eyes can you see the object and distinguish its color. For example, a white box reflects white light which, when it enters your eyes, allows you to tell that the box is white.

In the real world, two important phenomena occur when light hits an object (see Figure 8.1):

- Depending on the light source and direction, surface color is shaded.
- Depending on the light source and direction, objects "cast" shadows on the ground or the floor.

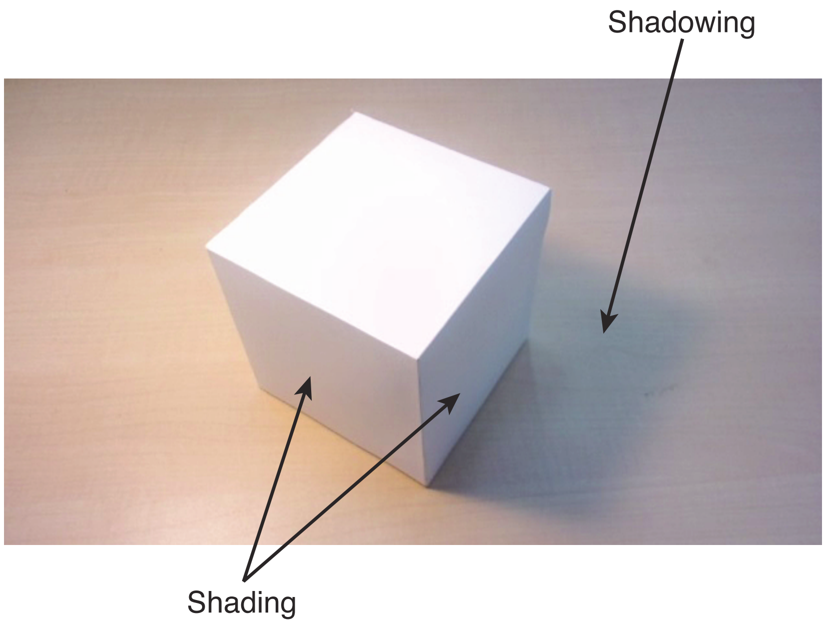

Figure 8.1 Shading and shadowing

In the real world, you usually notice shadows, but you quite often don't notice shading, which gives 3D objects their feeling of depth. Shading is subtle but always present. As shown in Figure 8.1, even surfaces of a pure white cube are distinguishable because each surface is shaded differently by light. As you can see, the surfaces hit by more light are brighter, and the surfaces hit by less light are darker, or more shaded. These differences allow you to distinguish each surface and ensure that the cube looks cubic.

In 3D graphics, the term **shading**[1] is used to describe the process that re-creates this phenomenon where the colors differ from surface to surface due to light. The other phenomenon, that the shadow of an object falls on the floor or ground, is re-created using a process called **shadowing**. This section discusses shading. Shadowing is discussed in Chapter 10, which focuses on a set of useful techniques that build on your basic knowledge of WebGL.

[1] Shading is so critical to 3D graphics that the core language, GLSL ES, is a shader language, the OpenGL ES Shading Language. The original purpose of shaders was to re-create the phenomena of shading.

When discussing shading, you need to consider two things:

- The type of light source that is emitting light
- How the light is reflected from surfaces of an object and enters the eye

Before we begin to program, let's look at different types of light sources and how light is reflected from different surfaces.

Types of Light Source

When light illuminates an object, a light source emits the light. In the real world, light sources are divided into two main categories: **directional light**, which is something like the sun that emits light naturally, and **point light**, which is something like a light bulb that emits light artificially. In addition, there is **ambient light** that represents indirect light (that is, light emitted from all light sources and reflected by walls or other objects (see Figure 8.2). In 3D graphics, there are additional types of light sources. For example, there is a spot light representing flashlights, headlamps, and so on. However, in this book, we don't address these more specialized light sources. Refer to the book *OpenGL ES 2.0 Programming Guide* for further information on these specialized light sources.

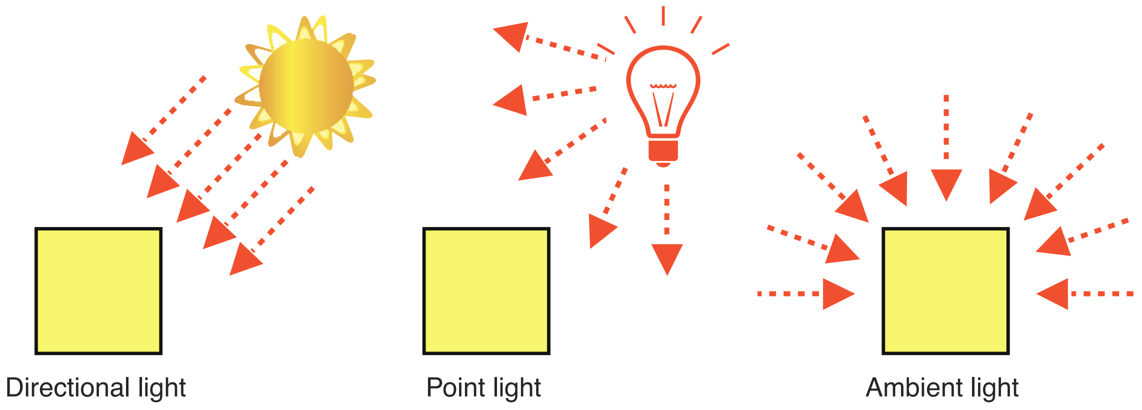

Figure 8.2 Directional light, point light, and ambient light

Focusing on the three main types of light source covered in this book:

Directional light: A directional light represents a light source whose light rays are parallel. It is a model of light whose source is considered to be at an infinite distance, such as the sun. Because of the distance travelled, the rays are effectively parallel by the time they reach the earth. This light source is considered the simplest, and because its rays are parallel can be specified using only direction and color.

Point light: A point light represents a light source that emits light in all directions from one single point. It is a model of light that can be used to represent light bulbs, lamps, flames, and so on. This light source is specified by its position and color.[2] However, the light direction is determined from the position of the light source and the position at which the light strikes a surface. As such, its direction can change considerably within the scene.

Ambient light: Ambient light (indirect light) is a model of light that is emitted from the other light source (directional or point), reflected by other objects such as walls, and reaches objects indirectly. It represents light that illuminates an object from all directions and has the same intensity.[3] For example, if you open the refrigerator door at night, the entire kitchen becomes slightly lighter. This is the effect of the ambient light. Ambient light does not have position and direction and is specified only by its color.

Now that you know the types of light sources that illuminate objects, let's discuss how light is reflected by the surface of an object and then enters the eye.

Types of Reflected Light

How light is reflected by the surface of an object and thus what color the surface will become is determined by two things: the type of the light and the type of surface of the object. Information about the type of light includes its color and direction. Information about the surface includes its color and orientation.

When calculating reflection from a surface, there are two main types: **diffuse reflection** and **environment** (or **ambient**) **reflection**. The remainder of this section describes how to calculate the color due to reflection using the two pieces of information described earlier. There is a little bit of math to be considered, but it's not complicated.

Diffuse Reflection

Diffuse reflection is the reflection of light from a directional light or a point light. In diffuse reflection, the light is reflected (scattered) equally in all directions from where it hits (see Figure 8.3). If a surface is perfectly smooth like a mirror, all incoming light is reflected; however, most surfaces are rough like paper, rock, or plastic. In such cases, the light is scattered in random directions from the rough surface. Diffuse reflection is a model of this phenomenon.

[2] This type of light actually attenuates; that is, it is strong near the source and becomes weaker farther from the source. For the sake of simplicity of the description and sample programs, light is treated as nonattenuating in this book. For attenuation, please refer to the book *OpenGL ES 2.0 Programming Guide*.

[3] In fact, ambient light is the combination of light emitted from light sources and reflected by various surfaces. It is approximated in this way because it would otherwise need complicated calculations to take into account all the many light sources and how and where they are reflected.

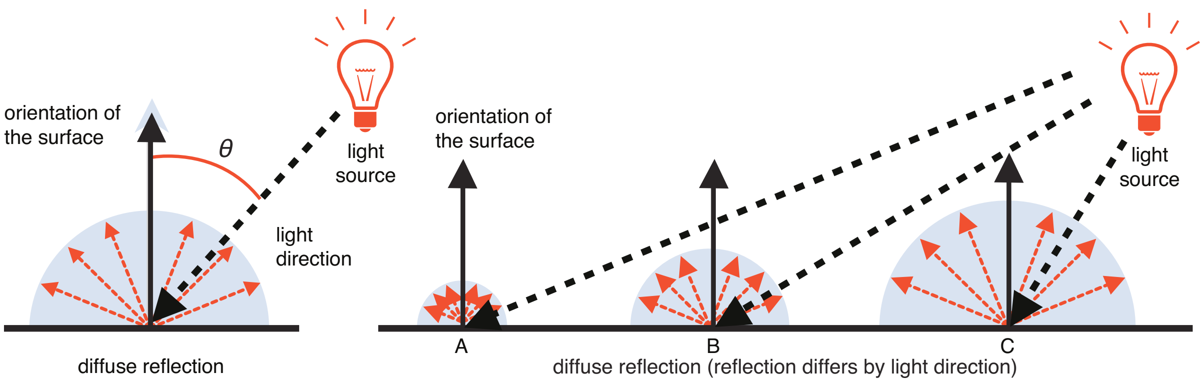
orientation of the surface
θ
light source
light direction
diffuse reflection
orientation of the surface
light source
A
B
C
diffuse reflection (reflection differs by light direction)

Figure 8.3 Diffuse reflection

In diffuse reflection, the color of the surface is determined by the color and the direction of light and the base color and orientation of the surface. The angle between the light direction and the orientation of the surface is defined by the angle formed by the light direction and the direction "perpendicular" to the surface. Calling this angle θ, the surface color by diffuse reflection is calculated using the following formula.

Equation 8.1

$$\langle \textit{surface color by diffuse reflection} \rangle = \langle \textit{light color} \rangle \times \langle \textit{base color of surface} \rangle \times \cos\theta$$

where <*light color*> is the color of light emitted from a directional light or a point light. Multiplication with the <*base color of the surface*> is performed for each RGB component of the color. Because light by diffuse reflection is scattered equally in all directions from where it hits, the intensity of the reflected light at a certain position is the same from any angle (see Figure 8.4).

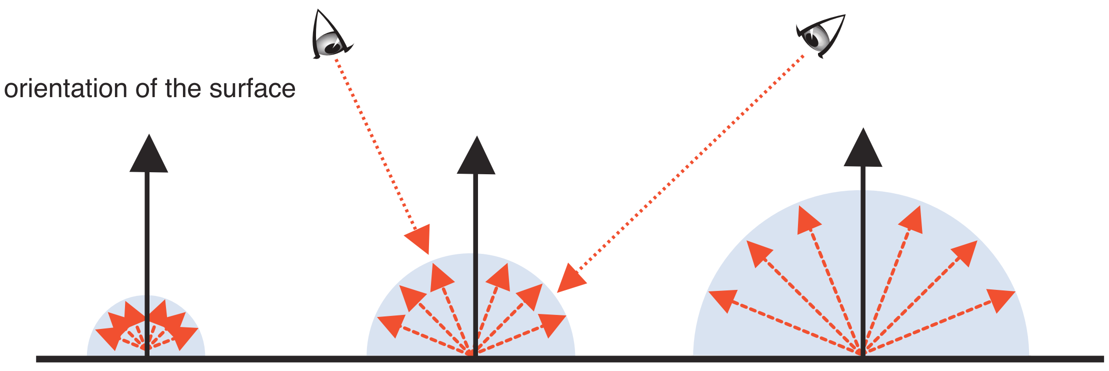

Figure 8.4 The intensity of light at a given position is the same from any angle

Ambient Reflection

Ambient reflection is the reflection of light from another light source. In ambient reflection, the light is reflected at the same angle as its incoming angle. Because an ambient light illuminates an object equally from all directions with the same intensity, its brightness is the same at any position (see Figure 8.5). It can be approximated as follows.

Equation 8.2

$$\langle \textit{surface color by ambient reflection} \rangle = \langle \textit{light color} \rangle \times \langle \textit{base color of surface} \rangle$$

where <*light color*> is the color of light emitted from other light source.

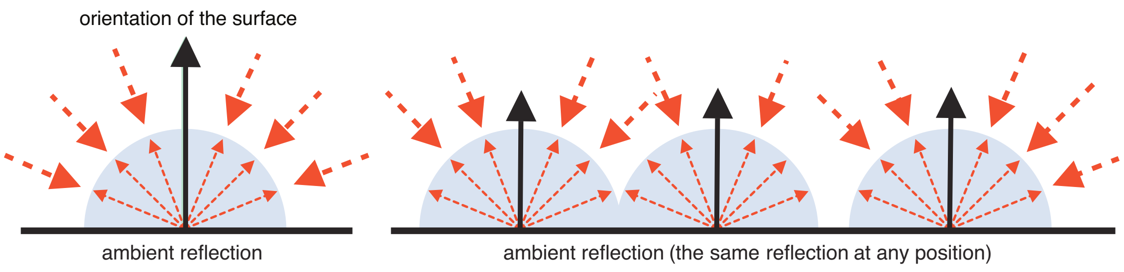
orientation of the surface
ambient reflection
ambient reflection (the same reflection at any position)

Figure 8.5 Ambient reflection

When both diffuse reflection and ambient reflection are present, the color of the surface is calculated by adding, as follows.

Equation 8.3

$$\langle \textit{surface color by diffuse and ambient reflection} \rangle = \langle \textit{surface color by diffuse reflection} \rangle + \langle \textit{surface color by ambient reflection} \rangle$$

Note that it is not required to always use both light sources, or use the formulas exactly as mentioned here. You are free to modify each formula to achieve the effect you require when showing the object.

Now let's construct some sample programs that perform shading (shading and coloring the surfaces of an object by placing a light source at an appropriate position). First let's try to implement shading due to directional light and its diffuse reflection.

Shading Due to Directional Light and Its Diffuse Reflection

As described in the previous section, surface color is determined by light direction and the orientation of the surface it strikes when considering diffuse reflection. The calculation of the color due to directional light is easy because its direction is constant. The formula for calculating the color of a surface by diffuse reflection (Equation 8.1) is shown again here:

$$\langle \textit{surface color by diffuse reflection} \rangle = \langle \textit{light color} \rangle \times \langle \textit{base color of surface} \rangle \times \cos\theta$$

The following three pieces of information are used:

- The color of the light source (directional light)
- The base color of the surface
- The angle (θ) between the light and the surface

The color of a light source may be white, such as sunlight, or other colors, such as the orange of lighting in road tunnels. As you know, it can be represented by RGB. White light such as sunlight has an RGB value of (1.0, 1.0, 1.0). The base color of a surface means the color that the surface was originally defined to have, such as red or blue. To calculate the color of a surface, you need to apply the formula for each of the three RGB components; the calculation is performed three times.

For example, assume that the light emitted from a light source is white (1.0, 1.0, 1.0), and the base color of the surface is red (1.0, 0.0, 0.0). From Equation 8.1, when θ is 0.0 (that is, when the light hits perpendicularly), cos θ becomes 1.0. Because the R component of the light source is 1.0, the R component of the base surface color is 1.0, and the cos θ is 1.0, the R component of the surface color by diffuse reflection is calculated as follows:

R = 1.0 * 1.0 * 1.0 = 1.0

The G and B components are also calculated in the same way, as follows:

G = 1.0 * 0.0 * 1.0 = 0.0

B = 1.0 * 0.0 * 1.0 = 0.0

From these calculations, when white light hits perpendicularly on a red surface, the surface color by diffuse reflection turns out to be (1.0, 0.0, 0.0), or red. This is consistent with real-world experience. Conversely, when the color of the light source is red and the base color of a surface is white, the result is the same.

Let's now consider the case when θ is 90 degrees, or when the light does not hit the surface at all. From your real-world experience, you know that in this case the surface will appear black. Let's validate this. Because cos θ is 0 when θ is 90 degrees, and anything multiplied by zero is zero, the result of the formula is 0 for R, G, and B; that is, the surface color becomes (0.0, 0.0, 0.0), or black, as expected. Equally, when θ is 60 degrees, you'd expect that a small amount of light falling on a red surface would result in a darker red color, and because cos θ is 0.5, the surface color is (0.5, 0.0, 0.0), which is dark red, as expected.

These simple examples have given you a good idea of how to calculate surface color due to diffuse reflection. To allow you to factor in directional light, let's transform the preceding formula to make it easy to handle so you can then explore how to draw a cube lit by directional light.

Calculating Diffuse Reflection Using the Light Direction and the Orientation of a Surface

In the previous examples, an arbitrary value for θ was chosen. However, typically it is complicated to get the angle θ between the light direction and the orientation of a surface. For example, when creating a model, the angle at which light hits each surface cannot be determined in advance. In contrast, the orientation of each surface can be determined regardless of where light hits from. Because the light direction is also determined when its light source is determined, it seems convenient to try to use these two pieces of information.

Fortunately, mathematics tells us that cos θ is derived by calculating the dot product of the light direction and the orientation of a surface. Because the dot product is so often used, GLSL ES provides a function to calculate it.[4] (More details can be found in Appendix B, "Built-In Functions of GLSL ES 1.0.") When representing the dot product by "·", cos θ is defined as follows:

$$\cos\theta = \langle \textit{light direction} \rangle \bullet \langle \textit{orientation of a surface} \rangle$$

From this, Equation 8.1 can be transformed as following Equation 8.4:

Equation 8.4

$$\begin{aligned}&\langle \textit{surface color by diffuse reflection} \rangle = \\ &\quad \langle \textit{light color} \rangle \times \langle \textit{base color of surface} \rangle \times \\ &\quad\quad (\langle \textit{light direction} \rangle \bullet \langle \textit{orientation of a surface} \rangle)\end{aligned}$$

Here, there are two points to be considered: the length of the vector and the light direction. First, the length of vectors that represent light direction and orientation of the surface, such as (2.0, 2.0, 1.0), must be 1.0,[5] or the color of the surface may become too dark or bright. Adjusting the components of a vector so that its length becomes 1.0 is called **normalization**.[6] GLSL ES provides functions for normalizing vectors that you can use directly.

The second point to consider concerns the light direction for the reflected light. The light direction is the opposite direction from that which the light rays travel (see Figure 8.6).

[4] Mathematically, the dot product of two vectors *n* and *l* is written as follows:
$n \bullet 1 = |n| \text{ x } |1| \text{ x } \cos\theta$
where | | means the length of the vector. From this equation, you can see that when the lengths of *n* and *l* are 1.0, the dot product is equal to cos θ. If *n* is (n_x, n_y, n_z) and *l* is (l_x, l_y, l_z), then $n_l = n_x * l_x + n_y * l_y + n_z * l_z$ from the law of cosines.

[5] If the components of the vector n are (n_x, n_y, n_z), its length is as follows:
$$\text{length of n} = |\text{n}| = \sqrt{n_x^2 + n_y^2 + n_z^2}$$

[6] Normalized n is $(n_x/m, n_y/m, n_z/m)$, where m is the length of n. |n| = sqrt(9) = 3. The vector (2.0, 2.0, 1.0) above is normalized into (2.0/3.0, 2.0/3.0, 1.0/3.0).

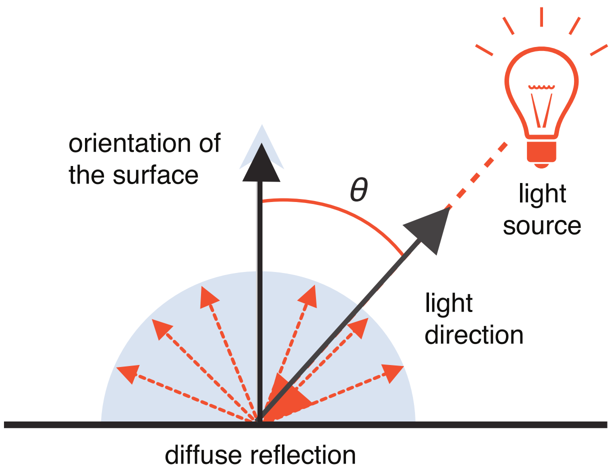

Figure 8.6 The light direction is from the reflecting surface to the light source

Because we aren't using an angle to specify the orientation of the surface, we need another mechanism to do that. The solution is to use normal vectors.

The Orientation of a Surface: What Is the Normal?

The orientation of a surface is specified by the direction perpendicular to the surface and is called a **normal** or a **normal vector**. This direction is represented by a triple number, which is the direction of a line from the origin (0, 0, 0) to (n_x, n_y, n_z) specified as the normal. For example, the direction of the normal (1, 0, 0) is the positive direction of the x-axis, and the direction of the normal (0, 0, 1) is the positive direction of the z-axis. When considering surfaces and their normals, two properties are important for our discussion.

A Surface Has Two Normals

Because a surface has a front face and a back face, each side has its own normal; that is, the surface has two normals. For example, the surface perpendicular to the z-axis has a front face that is facing toward the positive direction of the z-axis and a back face that is facing the negative direction of the z-axis, as shown in Figure 8.7. Their normals are (0, 0, 1) and (0, 0, –1), respectively.

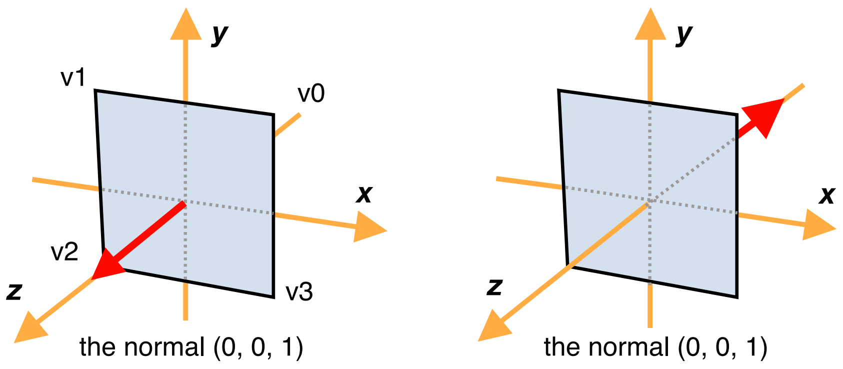

Figure 8.7 Normals

In 3D graphics, these two faces are distinguished by the order in which the vertices are specified when drawing the surface. When you draw a surface specifying vertices in the order[7] v0, v1, v2, and v3, the front face is the one whose vertices are arranged in a clockwise fashion when you look along the direction of the normal of the face (same as the right-handed rule determining the positive direction of rotation in Chapter 3, "Drawing and Transforming Triangles"). So in Figure 8.7, the front face has the normal (0, 0, –1) as in the right side of the figure.

The Same Orientation Has the Same Normal

Because a normal just represents direction, surfaces with the same orientation have the same normal regardless of the position of the surfaces.

If there is more than one surface with the same orientation placed at different positions, the normals of these surfaces are identical. For example, the normals of a surface perpendicular to the z-axis, whose center is placed at (10, 98, 9), are still (0, 0, 1) and (0, 0, –1). They are the same as when it is positioned at the origin (see Figure 8.8).

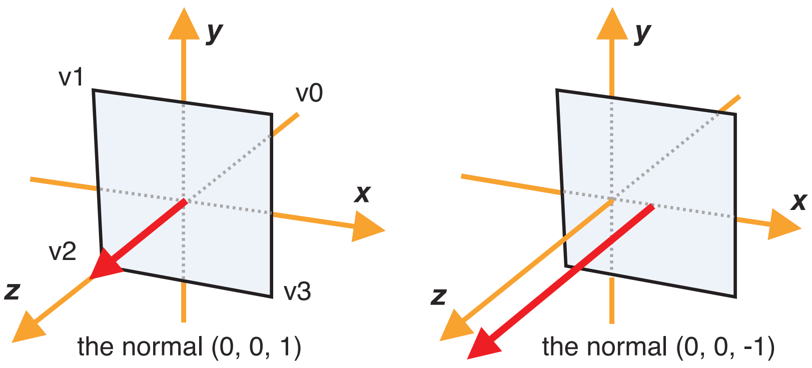

Figure 8.8 If the orientation of the surface is the same, the normal is identical regardless of its position

The left side of Figure 8.9 shows the normals that are used in the sample programs in this section. Normals are labeled using, for example "n(0, 1, 0)" as in this figure.

[7] Actually, this surface is composed of two triangles: a triangle drawn in the order v0, v1, and v2, and a triangle drawn in the order v0, v2, and v3.

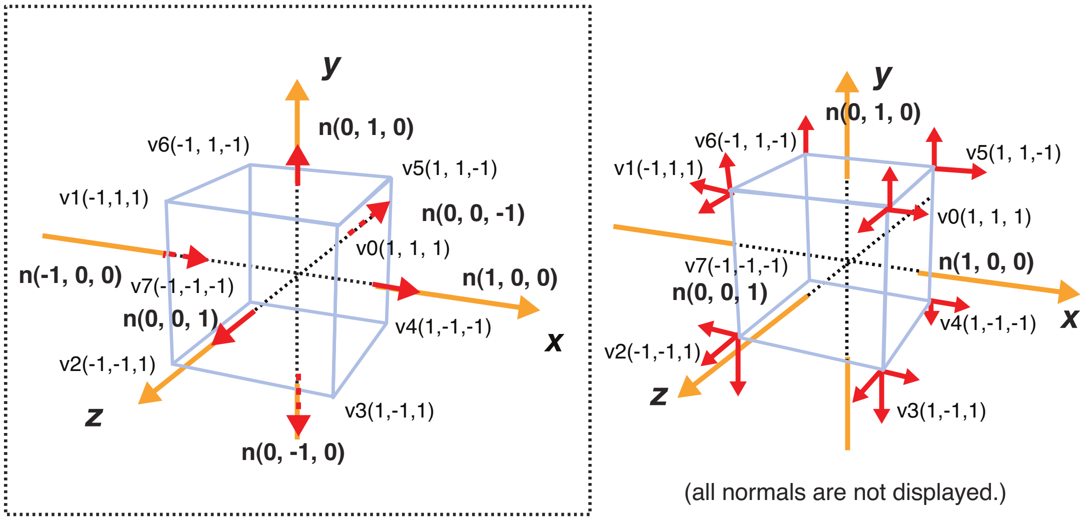

Figure 8.9 Normals of the surfaces of a cube

Once you have calculated the normals for a surface, the next task is to pass that data to the shader programs. In the previous chapter, you passed color data for a surface to the shader as "per-vertex data." You can pass normal data using the same approach: as per-vertex data stored in a buffer object. In this section, as shown in Figure 8.9 (right side), the normal data is specified for each vertex, and in this case there are three normals per vertex, just as there are three color data specified per vertex.[8]

Now let's construct a sample program `LightedCube` that displays a red cube lit by a white directional light. The result is shown in Figure 8.10.

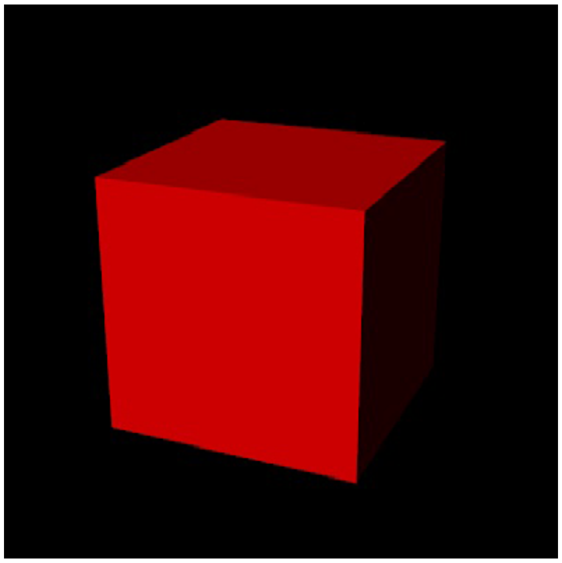

Figure 8.10 LightedCube

[8] Cubes or cuboids are simple but special objects whose three surfaces are connected perpendicularly. They have three different normals per vertex. On the other hand, smooth objects such as game characters have one normal per vertex.

Sample Program (LightedCube.js)

The sample program is shown in Listing 8.1. It is based on `ColoredCube` from the previous chapter, so the basic processing flow of this program is the same as `ColoredCube`.

As you can see from Listing 8.1, the vertex shader has been significantly modified so that it calculates Equation 8.4. In addition, the normal data is added in `initVertexBuffers()` defined at line 89, so that they can be passed to the variable `a_Normal`. The fragment shader is the same as in `ColoredCube`, and unmodified. It is reproduced so that you can see that no fragment processing is needed.

Listing 8.1 LightedCube.js

```
 1 // LightedCube.js
 2 // Vertex shader program
 3 var VSHADER_SOURCE =
 4   'attribute vec4 a_Position;\n' +
 5   'attribute vec4 a_Color;\n' +
 6   'attribute vec4 a_Normal;\n' +       // Normal
 7   'uniform mat4 u_MvpMatrix;\n' +
 8   'uniform vec3 u_LightColor;\n' +     // Light color
 9   'uniform vec3 u_LightDirection;\n' + // world coordinate, normalized
10   'varying vec4 v_Color;\n' +
11   'void main() {\n' +
12   '  gl_Position = u_MvpMatrix * a_Position ;\n' +
13      // Make the length of the normal 1.0
14   '  vec3 normal = normalize(vec3(a_Normal));\n' +
15      // Dot product of light direction and orientation of a surface
16   '  float nDotL = max(dot(u_LightDirection, normal), 0.0);\n' +
17      // Calculate the color due to diffuse reflection
18   '  vec3 diffuse = u_LightColor * vec3(a_Color) * nDotL;\n' +
19   '  v_Color = vec4(diffuse, a_Color.a);\n' +
20   '}\n';
21
22 // Fragment shader program
   ...
28   'void main() {\n' +
29   '  gl_FragColor = v_Color;\n' +
30   '}\n';
31
32 function main() {
   ...
49   // Set the vertex coordinates, the color, and the normal
50   var n = initVertexBuffers(gl);
   ...
```

```
  var u_MvpMatrix = gl.getUniformLocation(gl.program, 'u_MvpMatrix');
  var u_LightColor = gl.getUniformLocation(gl.program, 'u_LightColor');
  var u_LightDirection = gl.getUniformLocation(gl.program, 'u_LightDirection');
...
  // Set the light color (white)
  gl.uniform3f(u_LightColor, 1.0, 1.0, 1.0);
  // Set the light direction (in the world coordinate)
  var lightDirection = new Vector3([0.5, 3.0, 4.0]);
  lightDirection.normalize();     // Normalize
  gl.uniform3fv(u_LightDirection, lightDirection.elements);

  // Calculate the view projection matrix
  var mvpMatrix = new Matrix4();   // Model view projection matrix
  mvpMatrix.setPerspective(30, canvas.width/canvas.height, 1, 100);
  mvpMatrix.lookAt(3, 3, 7, 0, 0, 0, 0, 1, 0);
  // Pass the model view projection matrix to the variable u_MvpMatrix
  gl.uniformMatrix4fv(u_MvpMatrix, false, mvpMatrix.elements);
...
  gl.drawElements(gl.TRIANGLES, n, gl.UNSIGNED_BYTE, 0);// Draw a cube
}

function initVertexBuffers(gl) {
...
  var vertices = new Float32Array([ // Vertices
     1.0, 1.0, 1.0,  -1.0, 1.0, 1.0,  -1.0,-1.0, 1.0,   1.0,-1.0, 1.0,
     1.0, 1.0, 1.0,   1.0,-1.0, 1.0,   1.0,-1.0,-1.0,   1.0, 1.0,-1.0,
...
     1.0,-1.0,-1.0,  -1.0,-1.0,-1.0,  -1.0, 1.0,-1.0,   1.0, 1.0,-1.0
  ]);
...

  var normals = new Float32Array([ // Normals
    0.0, 0.0, 1.0,   0.0, 0.0, 1.0,   0.0, 0.0, 1.0,   0.0, 0.0, 1.0,
    1.0, 0.0, 0.0,   1.0, 0.0, 0.0,   1.0, 0.0, 0.0,   1.0, 0.0, 0.0,
...
    0.0, 0.0,-1.0,   0.0, 0.0,-1.0,   0.0, 0.0,-1.0,   0.0, 0.0,-1.0
  ]);
...
  if(!initArrayBuffer(gl,'a_Normal', normals, 3, gl.FLOAT)) return -1;
...
  return indices.length;
}
```

As a reminder, here is the calculation that the vertex shader performs (Equation 8.4):

$$\begin{aligned}&\langle \textit{surface color by diffuse reflection} \rangle = \\ &\quad \langle \textit{light color} \rangle \times \langle \textit{base color of surface} \rangle \times \\ &\quad\quad (\langle \textit{light direction} \rangle \bullet \langle \textit{orientation of a surface} \rangle)\end{aligned}$$

You can see that four pieces of information are needed to calculate this equation: (1) light color, (2) a surface base color, (3) light direction, and (4) surface orientation. In addition, <*light direction*> and <*surface orientation*> must be normalized (1.0 in length).

Processing in the Vertex Shader

From the four pieces of information necessary for Equation 8.4, the base color of a surface is passed as `a_Color` at line 5 in the following code, and the surface orientation is passed as `a_Normal` at line 6. The light color is passed using `u_LightColor` at line 8, and the light direction is passed as `u_LightDirection` at line 9. You should note that only `u_LightDirection` is passed in the world coordinate[9] system and has been normalized in the JavaScript code for ease of handling. This avoids the overhead of normalizing it every time it's used in the vertex shader:

```
 4  'attribute vec4 a_Position;\n' +
 5  'attribute vec4 a_Color;\n' +              <-(2) surface base color
 6  'attribute vec4 a_Normal;\n' +   // Normal  <-(4) surface orientation
 7  'uniform mat4 u_MvpMatrix;\n' +
 8  'uniform vec3 u_LightColor;\n' +   // Light color                  <-(1)
 9  'uniform vec3 u_LightDirection;\n' + // world coordinate,normalized <-(3)
10  'varying vec4 v_Color;\n' +
11  'void main() {\n' +
12  '  gl_Position = u_MvpMatrix * a_Position ;\n' +
13     // Make the length of the normal 1.0
14  '  vec3 normal = normalize(vec3(a_Normal));\n' +
15     // Dot product of light direction and orientation of a surface
16  '  float nDotL = max(dot(u_LightDirection, normal), 0.0);\n' +
17     // Calculate the color due to diffuse reflection
18  '  vec3 diffuse = u_LightColor * vec3(a_Color) * nDotL;\n' +
19  '  v_Color = vec4(diffuse, a_Color.a);\n' +
20  '}\n';
```

Once the necessary information is available, you can carry out the calculation. First, the vertex shader normalizes the vector at line 14. Technically, because the normal used in this sample program is 1.0 in length, this process is not necessary. However, it is good practice, so it is performed here:

9 In this book, the light effect with shading is calculated in the world coordinate system (see Appendix G, "World Coordinate System Versus Local Coordinate System") because it is simpler to program and more intuitive with respect to the light direction. It is also safe to calculate it in the view coordinate system but more complex.

```
14   '  vec3 normal = normalize(vec3(a_Normal));\n' +
```

Although `a_Normal` is of type `vec4`, a normal represents a direction and uses only the x, y, and z components. So you extract these components with `.xyz` and then normalize. If you pass the normal using a type `vec3`, this process is not necessary. However, it is passed as a type `vec4` in this code because a `vec4` will be needed when we extend the code for the next example. We will explain the details in a later sample program. As you can see, GLSL ES provides `normalize()`, a built-in function to normalize a vector specified as its argument. In the program, the normalized normal is stored in the variable `normal` for use later.

Next, you need to calculate the dot product $\langle \textit{light direction} \rangle \bullet \langle \textit{surface orientation} \rangle$ from Equation 8.4. The light direction is stored in `u_LightDirection`. Because it is already normalized, you can use it as is. The orientation of the surface is the `normal` that was normalized at line 14. The dot product "·" can then be calculated using the built-in function `dot()`, which again is provided by GLSL ES and returns the dot product of the two vectors specified as its arguments. That is, calling `dot(u_LightDirection, normal)` performs $\langle \textit{light direction} \rangle \bullet \langle \textit{surface orientation} \rangle$. This calculation is performed at line 16.

```
16   '  float nDotL = max(dot(u_LightDirection, normal), 0.0);\n' +
```

Once the dot product is calculated, if the result is positive, it is assigned to `nDotL`. If it is negative then 0.0 is assigned. The function `max()` used here is a GLSL ES built-in function that returns the greater value from its two arguments.

A negative dot product means that θ in cos θ is more than 90 degrees. Because θ is the angle between the light direction and the surface orientation, a value of θ greater than 90 degrees means that light hits the surface on its back face (see Figure 8.11). This is the same as no light hitting the front face, so 0.0 is assigned to `nDotL`.

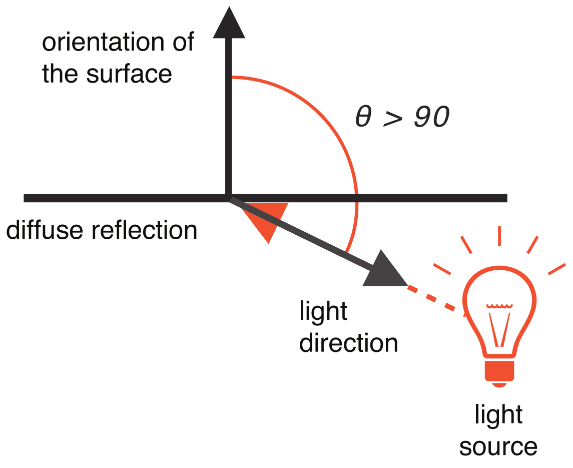

Figure 8.11 A normal and light in case θ is greater than 90 degrees

Now that the preparation is completed, you can calculate Equation 8.4. This is performed at line 18, which is a direct implementation of Equation 8.4. `a_Color`, which is of type `vec4` and holds the RGBA values, is converted to a `vec3` (`.rgb`) because its transparency (alpha value) is not used in lighting.

In fact, transparency of an object's surface has a significant effect on the color of the surface. However, because the calculation of the light passing through an object is complicated, we ignore transparency and don't use the alpha value in this program:

```
18   '  vec3 diffuse = u_LightColor * vec3(a_Color) * nDotL;\n' +
```

Once calculated, the result, `diffuse`, is assigned to the varying variable `v_Color` at line 19. Because `v_Color` is of type `vec4`, `diffuse` is also converted to `vec4` with `1.0`:

```
19   '  v_Color = vec4(diffuse, 1.0);\n' +
```

The result of the processing steps above is that a color, depending on the direction of the vertex's normal, is calculated, passed to the fragment shader, and assigned to `gl_FragColor`. In this case, because you use a directional light, vertices that make up the same surface are the same color, so each surface will be a solid color.

That completes the vertex shader code. Let's now take a look at how the JavaScript program passes the data needed for Equation 8.4 to the vertex shader.

Processing in the JavaScript Program

The light color (`u_LightColor`) and the light direction (`u_LightDirection`) are passed to the vertex shader from the JavaScript program. Because the light color is white (1.0, 1.0, 1.0), it is simply written to `u_LightColor` using `gl.uniform3f()`:

```
69   // set the Light color (white)
70   gl.uniform3f(u_LightColor, 1.0, 1.0, 1.0);
```

The next step is to set up the light direction, which must be passed after normalization, as discussed before. You can normalize it with the `normalize()` function for `Vector3` objects that is provided in `cuon-matrix.js`. Usage is simple: Create the `Vector3` object that specifies the vector you want to normalize as its argument (line 72), and invoke the `normalize()` method on the object. Note that the notation in JavaScript is different from that of GLSL ES:

```
71   // Set the light direction (in the world coordinate)
72   var lightDirection = new Vector3([0.5, 3.0, 4.0]);
73   lightDirection.normalize();     // Normalize
74   gl.uniform3fv(u_LightDirection, lightDirection.elements);
```

The result is stored in the `elements` property of the object in an array of type `Float32Array` and then assigned to `u_LightDirection` using `gl.uniform3fv()` (line 74).

Finally, the normal data is written in `initVertexBuffers()`, defined at line 89. Actual normal data is stored in the array `normals` at line 118 per vertex along with the color data, as in `ColoredCube.js`. Data is assigned to `a_Normal` in the vertex shader by invoking `initArrayBuffer()` at line 140:

```
140 if(!initArrayBuffer(gl, 'a_Normal', normals, 3, gl.FLOAT)) return -1;
```

initArrayBuffer(), which was also used in ColoredCube, assigns the array specified by the third argument (normals) to the attribute variable that has the name specified by the second argument (a_Normal).

Add Shading Due to Ambient Light

Although at this stage you have successfully added lighting to the scene, as you can see from Figure 8.9, when you run LightedCube, the cube is a little different from the box in the real world. In particular, the surface on the opposite side of the light source appears almost black and not clearly visible. You can see this problem more clearly if you animate the cube. Try the sample program LightedCube_animation (see Figure 8.12) to see the problem more clearly.

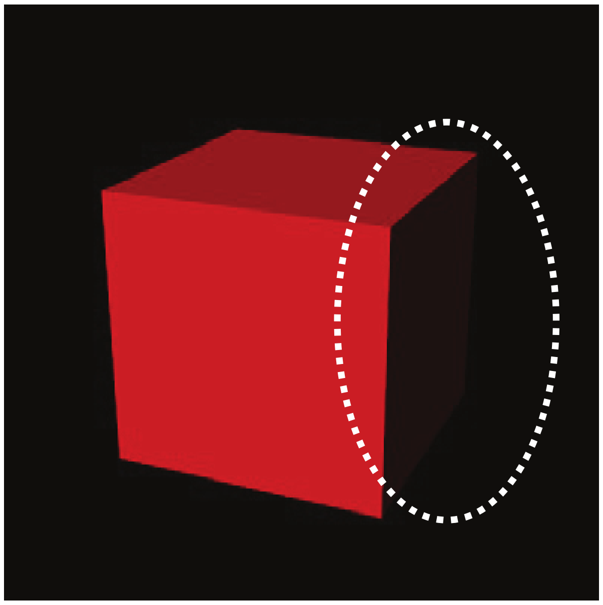

Figure 8.12 The result of LightedCube_animation

Although the scene is correctly lit as the result of Equation 8.4, our real-world experiences tells us that something isn't right. It is unusual to see such a sharp effect because, in the real world, surfaces such as the back face of the cube are also lit by diffuse or reflected light. The ambient light described in the previous section represents this indirect light and can be used to make the scene more lifelike. Let's add that to the scene and see if the effect is more realistic. Because ambient light models the light that hits an object from all directions with constant intensity, the surface color due to the reflection is determined only by the light color and the base color of the surface. The formula that calculates this was shown as Equation 8.2. Let's see it again:

$$\langle \textit{surface color by ambient reflection} \rangle = \\ \langle \textit{light color} \rangle \times \langle \textit{base color of surface} \rangle$$

Let's try to add the color due to ambient light described by this formula to the sample program `LightedCube`. To do this, use Equation 8.3 shown here:

$$\langle \textit{surface color by diffuse and ambient reflection} \rangle = \\ \langle \textit{surface color by diffuse reflection} \rangle + \langle \textit{surface color by ambient reflection} \rangle$$

Ambient light is weak because it is the light reflected by other objects like the walls. For example, if the ambient light color is (0.2, 0.2, 0.2) and the base color of a surface is red, or (1.0, 0.0, 0.0), then, from Equation 8.2, the surface color due to the ambient light is (0.2, 0.0, 0.0). For example, if there is a white box in a blue room—that is, the base color of the surface is (1.0, 1.0, 1.0) and the ambient light is (0.0, 0.0, 0.2)—the color becomes slightly blue (0.0, 0.0, 0.2).

Let's implement the effect of ambient reflection in the sample program `LightedCube_ambient`, which results in the cube shown in Figure 8.13. You can see that the surface that the light does not directly hit is now also slightly colored and more closely resembles the cube in the real world.

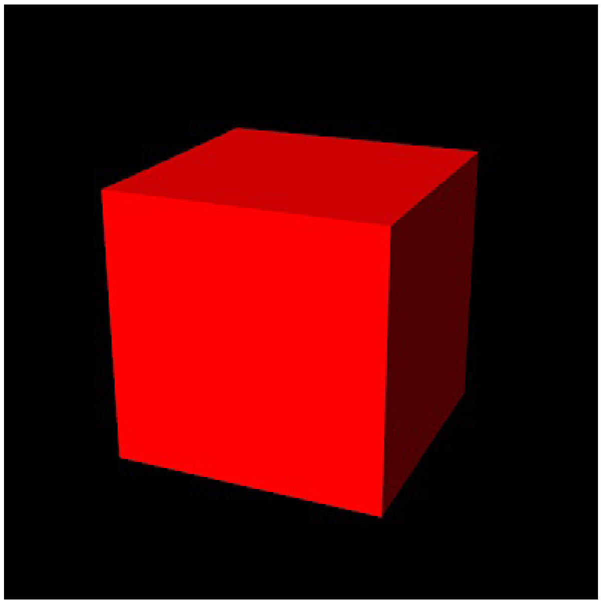

Figure 8.13 LightedCube_ambient

Sample Program (LightedCube_ambient.js)

Listing 8.2 illustrates the sample program. Because it is almost the same as `LightedCube`, only the modified parts are shown.

Listing 8.2 LightedCube_ambient.js

```
// LightedCube_ambient.js
// Vertex shader program
 ...
 8   'uniform vec3 u_LightColor;\n' +     // Light color
 9   'uniform vec3 u_LightDirection;\n' + // World coordinate, normalized
10   'uniform vec3 u_AmbientLight;\n' +   // Color of an ambient light
11   'varying vec4 v_Color;\n' +
12   'void main() {\n' +
  ...
16      // The dot product of the light direction and the normal
17   '  float nDotL = max(dot(lightDirection, normal), 0.0);\n' +
18      // Calculate the color due to diffuse reflection
19   '  vec3 diffuse = u_LightColor * a_Color.rgb * nDotL;\n' +
20      // Calculate the color due to ambient reflection
21   '  vec3 ambient = u_AmbientLight * a_Color.rgb;\n' +
22      // Add surface colors due to diffuse and ambient reflection
23   '  v_Color = vec4(diffuse + ambient, a_Color.a);\n' +
24   '}\n';
   ...
36 function main() {
   ...
64   // Get the storage locations of uniform variables and so on
   ...
68   var u_AmbientLight = gl.getUniformLocation(gl.program, 'u_AmbientLight');
   ...
80   // Set the ambient light
81   gl.uniform3f(u_AmbientLight, 0.2, 0.2, 0.2);
   ...
95 }
```

`u_AmbientLight` at line 10 is added to the vertex shader to pass in the color of ambient light. After Equation 8.2 is calculated using it and the base color of the surface (`a_Color`), the result is stored in the variable ambient (line 21). Now that both `diffuse` and `ambient` are determined, the surface color is calculated at line 23 using Equation 8.3. The result is passed to `v_Color`, just like in `LightedCube`, and the surface is painted with this color.

As you can see, this program simply adds `ambient` at line 23, causing the whole cube to become brighter. This implements the effect of the ambient light hitting an object equally from all directions.

The examples so far have been able to handle static objects. However, because objects are likely to move within a scene, or the viewpoint changes, you have to be able to handle such transformations. As you will recall from Chapter 4, "More Transformations and Basic Animation," an object can be translated, scaled, or rotated using coordinate transformations. These transformations may also change the normal direction and require a recalculation of lighting as the scene changes. Let's take a look at how to achieve that.

Lighting the Translated-Rotated Object

The program `LightedTranslatedRotatedCube` uses a directional light source to light a cube that is rotated 90 degrees clockwise around the z-axis and translated 0.9 units in the y-axis direction. A part from a directional light as described in the previous section, the sample, `LightedCube_ambient`, uses diffuse reflection and ambient reflection and rotates and translates the cube. The result is shown in Figure 8.14.

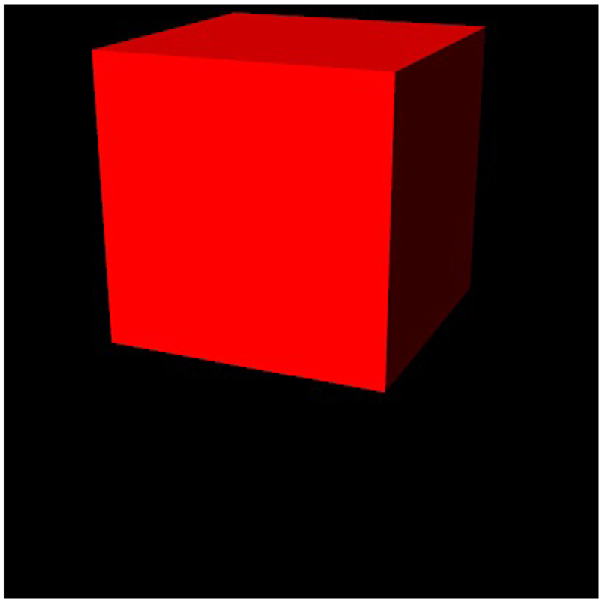

Figure 8.14 LightedTranslatedRotatedCube

You saw in the previous section that the normal direction may change when coordinate transformations are applied. Figure 8.15 shows some examples of that. The leftmost figure in Figure 8.15 shows the cube used in this sample program looking along the negative direction of the z-axis. The only normal (1, 0, 0), which is toward the positive direction of the x-axis, is shown. Let's perform some coordinate transforms on this figure, which are the three figures on the right.

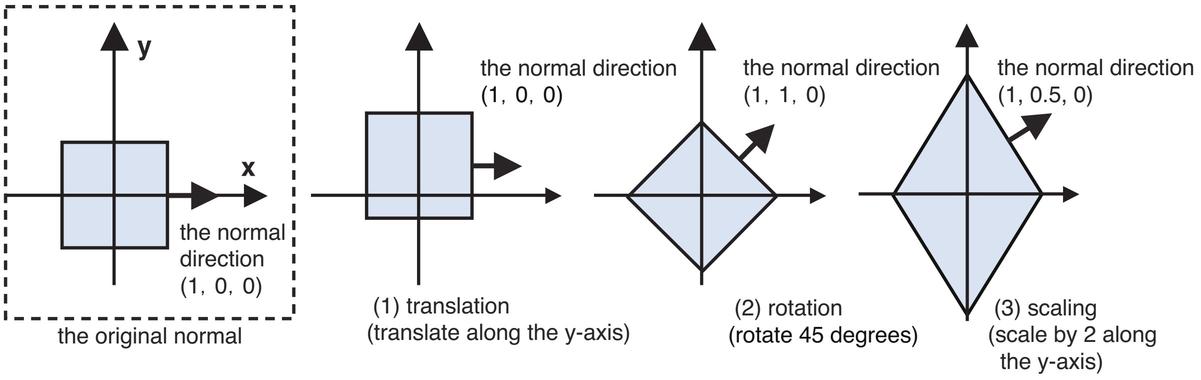

Figure 8.15 The changes of the normal direction due to coordinate transformations

You can see the following from Figure 8.15:

- The normal direction is **not changed** by a translation because the orientation of the object does not change.
- The normal direction is **changed** by a rotation according to the orientation of the object.
- Scaling has a more complicated effect on the normal. As you can see, the object in the rightmost figure is rotated i and then scaled two times only in the y-axis. In this case, the normal direction is **changed** because the orientation of the surface changes. On the other hand, if an object is scaled equally in all axes the normal direction is **not changed.** Finally, **even if** an object is scaled unequally, the normal direction may **not change**. For example, when the leftmost figure (the original normal) is scaled two times only in the y-axis direction, the normal direction does not change.

Obviously, the calculation of the normal under various transformations is complex, particularly when dealing with scaling. However, a mathematical technique can help.

The Magic Matrix: Inverse Transpose Matrix

As described in Chapter 4, the matrix that performs a coordinate transformation on an object is called a model matrix. The normal direction can be calculated by multiplying the normal by the **inverse transpose matrix** of a model matrix. The inverse transpose matrix is the matrix that transposes the inverse of a matrix.

The inverse of the matrix M is the matrix R, where both R*M and M*R become the identity matrix. The term **transpose** means the operation that exchanges rows and columns of a matrix. The details of this are explained in Appendix E, "The Inverse Transpose Matrix." For our purposes, it can be summarized simply using the following rule:

Rule: You can calculate the normal direction if you multiply the normal by the inverse transpose of the model matrix.

The inverse transpose matrix is calculated as follows:

1. Invert the original matrix.
2. Transpose the resulting matrix.

This can be carried out using convenient methods supported by the `Matrix4` object (see Table 8.1).

Table 8.1 Matrix4 Methods for an Inverse Transpose Matrix

Method	**Description**
`Matrix4.setInverseOf(m)`	Calculates the inverse of the matrix stored in *m* and stores the result in the `Matrix4` object, where *m* is a `Matrix4` object
`Matrix4.transpose()`	Transposes the matrix stored in the `Matrix4` object and writes the result back into the `Matrix4` object

Assuming that a model matrix is stored in `modelMatrix`, which is a `Matrix4` object, the following code snippet will get its inverse transpose matrix. The result is stored in the variable named `normalMatrix`, because it performs the coordinate transformation of a normal:

```
Matrix4 normalMatrix = new Matrix4();
// Calculate the model matrix
...
// Calculate the matrix to transform normal according to the model matrix
normalMatrix.setInverseOf(modelMatrix);
normalMatrix.transpose();
```

Now let's see the program `LightedTranslatedRotatedCube.js` that lights the cube, which is rotated 90 degrees clockwise around the z-axis and translated 0.9 along the y-axis, all using directional light. You'll use the cube that was transformed by the model matrix in `LightedCube_ambient` from the previous section.

Sample Program (LightedTranslatedRotatedCube.js)

Listing 8.3 shows the sample program. The changes from `LightedCube_ambient` are that `u_NormalMatrix` is added (line 8) to pass the matrix for coordinate transformation of the normal to the vertex shader, and the normal is transformed at line 16 using this matrix. `u_NormalMatrix` is calculated within the JavaScript.

Listing 8.3 LightedTranslatedRotatedCube.js

```
 1 // LightedTranslatedRotatedCube.js
 2 // Vertex shader program
 3 var VSHADER_SOURCE =
   ...
 6   'attribute vec4 a_Normal;\n' +
 7   'uniform mat4 u_MvpMatrix;\n' +
 8   'uniform mat4 u_NormalMatrix;\n'+    // Transformation matrix of normal
 9   'uniform vec3 u_LightColor;\n' +     // Light color
10   'uniform vec3 u_LightDirection;\n' + // World coordinate, normalized
11   'uniform vec3 u_AmbientLight;\n' +   // Ambient light color
12   'varying vec4 v_Color;\n' +
  'void main() {\n' +
  '  gl_Position = u_MvpMatrix * a_Position;\n' +
     // Recalculate normal with normal matrix and make its length 1.0
  '  vec3 normal = normalize(vec3(u_NormalMatrix * a_Normal));\n' +
     // The dot product of the light direction and the normal
  '  float nDotL = max(dot(u_LightDirection, normal), 0.0);\n' +
     // Calculate the color due to diffuse reflection
  '  vec3 diffuse = u_LightColor * a_Color.rgb * nDotL;\n' +
     // Calculate the color due to ambient reflection
  '  vec3 ambient = u_AmbientLight * a_Color.rgb;\n' +
     // Add the surface colors due to diffuse and ambient reflection
  '  v_Color = vec4(diffuse + ambient, a_Color.a);\n' +
  '}\n';
...
function main() {
...
  // Get the storage locations of uniform variables and so on
  var u_MvpMatrix = gl.getUniformLocation(gl.program, 'u_MvpMatrix');
  var u_NormalMatrix = gl.getUniformLocation(gl.program, 'u_NormalMatrix');
...
  var modelMatrix = new Matrix4();  // Model matrix
  var mvpMatrix = new Matrix4();    // Model view projection matrix
  var normalMatrix = new Matrix4(); // Transformation matrix for normal

  // Calculate the model matrix
  modelMatrix.setTranslate(0, 1, 0); // Translate to y-axis direction
  modelMatrix.rotate(90, 0, 0, 1);   // Rotate around the z-axis
  // Calculate the view projection matrix
  mvpMatrix.setPerspective(30, canvas.width/canvas.height, 1, 100);
  mvpMatrix.lookAt(-7, 2.5, 6, 0, 0, 0, 0, 1, 0);
  mvpMatrix.multiply(modelMatrix);
  // Pass the model view projection matrix to u_MvpMatrix
  gl.uniformMatrix4fv(u_MvpMatrix, false, mvpMatrix.elements);

  // Calculate matrix to transform normal based on the model matrix
  normalMatrix.setInverseOf(modelMatrix);
  normalMatrix.transpose();
  // Pass the transformation matrix for normal to u_NormalMatrix
  gl.uniformMatrix4fv(u_NormalMatrix, false, normalMatrix.elements);
...
}
```

The processing in the vertex shader is almost the same as in `LightedCube_ambient`. The difference, in line with the preceding rule, is that you multiply `a_Normal` by the inverse transpose of the model matrix at line 16 instead of using it as-is:

```
15    // Recalculate normal with normal matrix and make its length 1.0
16   '  vec3 normal = normalize(vec3(u_NormalMatrix * a_Normal));\n' +
```

Because you passed `a_Normal` as type `vec4`, you can multiply it by `u_NormalMatrix`, which is of type `mat4`. You only need the x, y, and z components of the result of the multiplication, so the result is converted into type `vec3` with `vec3()`. It is also possible to use `.xyz` as before, or write `(u_NormalMatrix * a_Normal).xyz`. However, `vec3()` is used here for simplicity. Now that you understand how the shader calculates the normal direction resulting from the rotation and translation of the object, let's move on to the explanation of the JavaScript program. The key point here is the calculation of the matrix that will be passed to `u_NormalMatrix` in the vertex shader.

`u_NormalMatrix` is the inverse transpose of the model matrix, so the model matrix is first calculated at lines 90 and 91. Because this program rotates an object around the z-axis and translates it in the y-axis direction, you can use the `setTranslate()` and `rotate()` methods of a `Matrix4` object as described in Chapter 4. It is at lines 100 and 101 that the inverse transpose matrix is actually calculated. It is passed to `u_NormalMatrix` in the vertex shader at line 103, in the same way as `mvpMatrix` at line 97. The second argument of `gl.uniformMatrix4f()` specifies whether to transpose the matrix (Chapter 3):

```
 99   // Calculate matrix to transform normal based on the model matrix
100   normalMatrix.setInverseOf(modelMatrix);
101   normalMatrix.transpose();
102   // Pass the normal transformation matrix to u_NormalMatrix
103   gl.uniformMatrix4fv(u_NormalMatrix, false, normalMatrix.elements);
```

When run, the output is similar to Figure 8.14. As you can see, the shading is the same as `LightedCube_ambient` with the cube translated in the y-axis direction. That is because (1) the translation doesn't change the normal direction, (2) neither does the rotation by 90 degrees, because the rotation simply switches the surfaces of the cube, (3) the light direction of the directional light does not change regardless of the position of the object, and (4) diffuse reflection reflects the light in all directions with equal intensity.

You now have a good understanding of the basics of how to implement light and shade in 3D graphics. Let's build on this by exploring another type of light source: the point light.

Using a Point Light Object

In contrast to a directional light, the direction of the light from a point light source differs at each position in the 3D scene (see Figure 8.16). So, when calculating shading, you need to calculate the light direction at the specific position on the surface where the light hits.

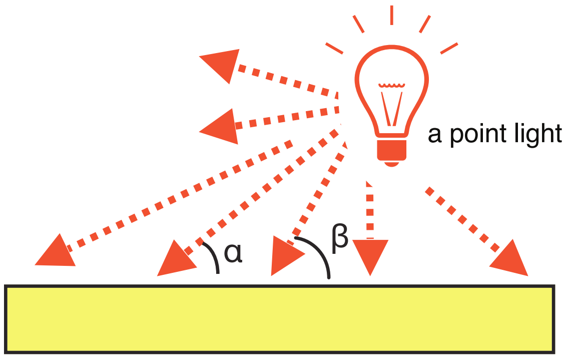

Figure 8.16 The direction of a point light varies by position

In the previous sample programs, you calculated the color at each vertex by passing the normal and the light direction for each vertex. You will use the same approach here, but because the light direction changes, you need to pass the position of the light source and then calculate the light direction at each vertex position.

Here, you construct the sample program `PointLightedCube` that displays a red cube lit with white light from a point light source. We again use diffuse reflection and ambient reflection. The result is shown in Figure 8.17, which is a version of `LightedCube_ambient` from the previous section but now lit with a point light.

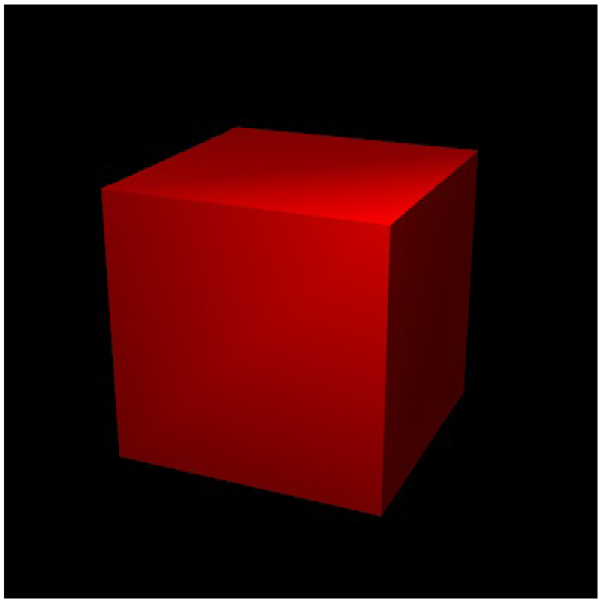

Figure 8.17 PointLightedCube

Sample Program (PointLightedCube.js)

Listing 8.4 shows the sample program in which only the vertex shader is changed from `LightedCube_ambient`. The variable `u_ModelMatrix` for passing the model matrix and the variable `u_LightPosition` representing the light position are added. Note that because you use a point light in this program, you will use the light position instead of the light direction. Also, to make the effect easier to see, we have enlarged the cube.

Listing 8.4 PointLightedCube.js

```
// PointLightedCube.js
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
 ...
  'uniform mat4 u_ModelMatrix;\n' +  // Model matrix
  'uniform mat4 u_NormalMatrix;\n' + // Transformation matrix of normal
  'uniform vec3 u_LightColor;\n' +   // Light color
  'uniform vec3 u_LightPosition;\n' +  // Position of the light source (in the
                                                         ➥world coordinate system)
  'uniform vec3 u_AmbientLight;\n' +   // Ambient light color
  'varying vec4 v_Color;\n' +
  'void main() {\n' +
  '  gl_Position = u_MvpMatrix * a_Position;\n' +
     // Recalculate normal with normal matrix and make its length 1.0
  '  vec3 normal = normalize(vec3(u_NormalMatrix * a_Normal));\n' +
    // Calculate the world coordinate of the vertex
  '  vec4 vertexPosition = u_ModelMatrix * a_Position;\n' +
     // Calculate the light direction and make it 1.0 in length
  '  vec3 lightDirection = normalize(u_LightPosition - vec3(vertexPosition));\n' +
     // The dot product of the light direction and the normal
  '  float nDotL = max(dot( lightDirection, normal), 0.0);\n' +
     // Calculate the color due to diffuse reflection
  '  vec3 diffuse = u_LightColor * a_Color.rgb * nDotL;\n' +
     // Calculate the color due to ambient reflection
  '  vec3 ambient = u_AmbientLight * a_Color.rgb;\n' +
     // Add surface colors due to diffuse and ambient reflection
  '  v_Color = vec4(diffuse + ambient, a_Color.a);\n' +
  '}\n';
 ...
function main() {
 ...
  // Get the storage locations of uniform variables and so on
  var u_ModelMatrix = gl.getUniformLocation(gl.program, 'u_ModelMatrix');
 ...
  var u_LightColor = gl.getUniformLocation(gl.program,'u_LightColor');
  var u_LightPosition = gl.getUniformLocation(gl.program, 'u_LightPosition');
 ...
  // Set the light color (white)
```

```
gl.uniform3f(u_LightColor, 1.0, 1.0, 1.0);
// Set the position of the light source (in the world coordinate)
gl.uniform3f(u_LightPosition, 0.0, 3.0, 4.0);
...
var modelMatrix = new Matrix4();  // Model matrix
var mvpMatrix = new Matrix4();    // Model view projection matrix
var normalMatrix = new Matrix4(); // Transformation matrix for normal

// Calculate the model matrix
modelMatrix.setRotate(90, 0, 1, 0); // Rotate around the y-axis
// Pass the model matrix to u_ModelMatrix
gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);
...
```

The key differences in the processing within the vertex shader are at line 19 and 21. At line 19, you transform the vertex coordinates into world coordinates in order to calculate the light direction at the vertex coordinates. Because a point light emits light in all directions from its position, the light direction at a vertex is the result of subtracting the vertex position from the light source position. Because the light position is passed to the variable `u_LightPosition` using world coordinates at line 11, you also have to convert the vertex coordinates into world coordinates to calculate the light direction. The light direction is then calculated at line 21. Note that it is normalized with `normalize()` so that it will be 1.0 in length. Using the resulting light direction (`lightDirection`), the dot product is calculated at line 23 and then the surface color at each vertex is calculated based on this light direction.

If you run this program, you will see a more realistic result, as shown in Figure 8.17. Although this result is more realistic, a closer look reveals an artifact: There are unnatural lines of shade on the cube's surface (see Figure 8.18). You can see this more easily if the cube rotates as it does when you load `PointLightedCube_animation`.

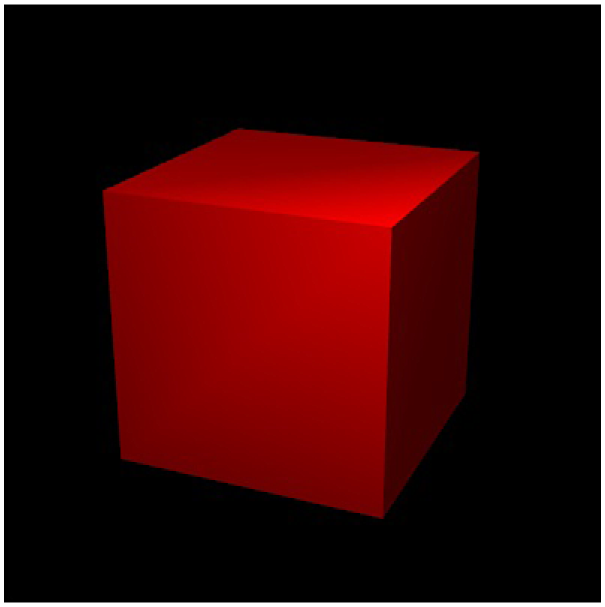

Figure 8.18 The unnatural appearance when processing the point light at each vertex

This comes about because of the interpolation process discussed in Chapter 5, "Using Colors and Texture Images." As you will remember, the WebGL system interpolates the colors between vertices based on the colors you supply at the vertices. However, because the direction of light from a point light source varies by position to shade naturally, you have to calculate the color at every position the light hits instead of just at each vertex. You can see this problem more clearly using a sphere illuminated by a point light, as shown in Figure 8.19.

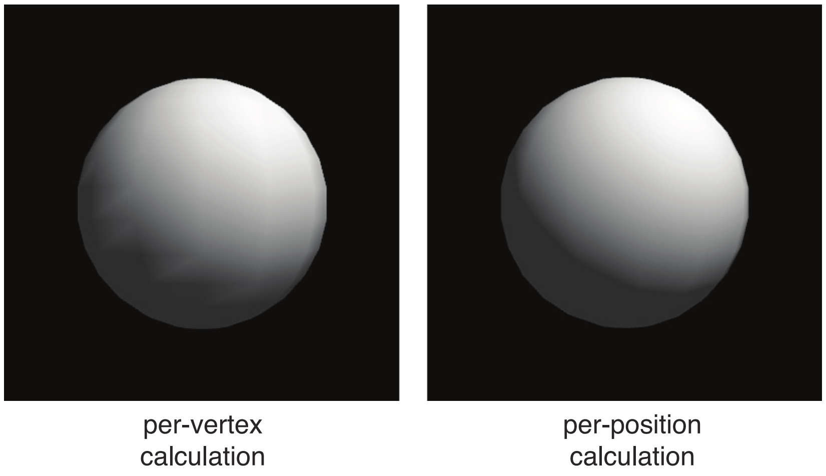

Figure 8.19 The spheres illuminated by a point light

As you can see, the border between the brighter parts and darker parts is unnatural in the left figure. If the effect is hard to see on the page, the left figure is `PointLightedSphere`, and the right is `PointLightedSphere_perFragment`. We will describe how to draw them correctly in the next section.

More Realistic Shading: Calculating the Color per Fragment

At first glance, it may seem daunting to have to calculate the color at every position on a cube surface where the light hits. However, essentially it means calculating the color **per fragment**, so the power of the fragment shader can now be used.

This sample program you will use is `PointLightedCube_perFragment`, and its result is shown in Figure 8.20.

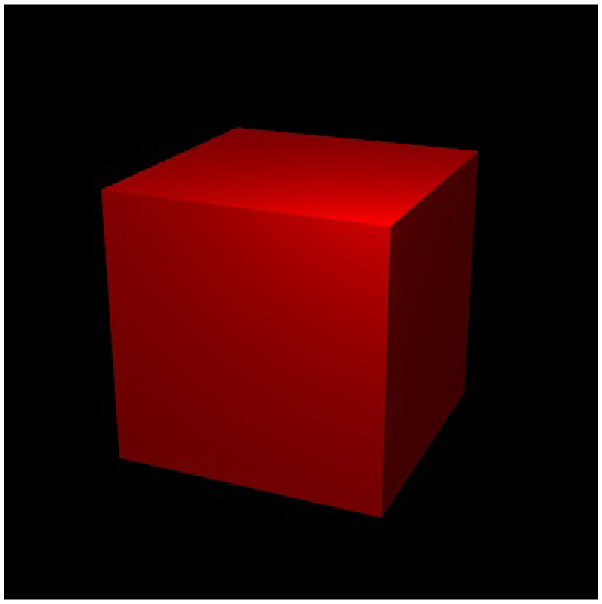

Figure 8.20 PointLightedCube_perFragment

Sample Program (PointLightedCube_perFragment.js)

The sample program, which is based on `PointLightedCube.js`, is shown in Listing 8.5. Only the shader code has been modified and, as you can see, there is less processing in the vertex shader and more processing in the fragment shader.

Listing 8.5 PointLightedCube_perFragment.js

```
// PointLightedCube_perFragment.js
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
 ...
  'uniform mat4 u_ModelMatrix;\n' +  // Model matrix
  'uniform mat4 u_NormalMatrix;\n' + // Transformation matrix of normal
  'varying vec4 v_Color;\n' +
  'varying vec3 v_Normal;\n' +
  'varying vec3 v_Position;\n' +
  'void main() {\n' +
14   '  gl_Position = u_MvpMatrix * a_Position;\n' +
15      // Calculate the vertex position in the world coordinate
16   '  v_Position = vec3(u_ModelMatrix * a_Position);\n' +
17   '  v_Normal = normalize(vec3(u_NormalMatrix * a_Normal));\n' +
18   '  v_Color = a_Color;\n' +
19   '}\n';
20
21 // Fragment shader program
22 var FSHADER_SOURCE =
    ...
26   'uniform vec3 u_LightColor;\n' +     // Light color
27   'uniform vec3 u_LightPosition;\n' +  // Position of the light source
28   'uniform vec3 u_AmbientLight;\n' +   // Ambient light color
29   'varying vec3 v_Normal;\n' +
30   'varying vec3 v_Position;\n' +
31   'varying vec4 v_Color;\n' +
32   'void main() {\n' +
33      // Normalize normal because it's interpolated and not 1.0 (length)
34   '  vec3 normal = normalize(v_Normal);\n' +
35      // Calculate the light direction and make it 1.0 in length
36   '  vec3 lightDirection = normalize(u_LightPosition - v_Position);\n' +
37      // The dot product of the light direction and the normal
38   '  float nDotL = max(dot( lightDirection, normal), 0.0);\n' +
39      // Calculate the final color from diffuse and ambient reflection
40   '  vec3 diffuse = u_LightColor * v_Color.rgb * nDotL;\n' +
41   '  vec3 ambient = u_AmbientLight * v_Color.rgb;\n' +
42   '  gl_FragColor = vec4(diffuse + ambient, v_Color.a);\n' +
43   '}\n';
```

To calculate the color per fragment when light hits, you need (1) the position of the fragment in the world coordinate system and (2) the normal direction at the fragment position. You can utilize interpolation (Chapter 5) to obtain these values per fragment by just calculating them per vertex in the vertex shader and passing them via varying variables to the fragment shader.

These calculations are performed at lines 16 and 17, respectively, in the vertex shader. At line 16, the vertex position in world coordinates is calculated by multiplying each vertex coordinate by the model matrix. After assigning the vertex position to the varying variable `v_Position`, it will be interpolated between vertices and passed to the corresponding variable (`v_Position`) in the fragment shader as the world coordinate of the fragment. The normal calculation at line 17 is carried out for the same purpose.[10] By assigning the result to `v_Normal`, it is also interpolated and passed to the corresponding variable (`v_Normal`) in the fragment shader as the normal of the fragment.

Processing in the fragment shader is the same as that in the vertex shader of `PointLightedCube.js`. First, at line 34, the interpolated normal passed from the vertex shader is normalized. Its length may not be 1.0 anymore because of the interpolation. Next, at line 36, the light direction is calculated and normalized. Using these results, the dot product of the light direction and the normal is calculated at line 38. The colors due to the diffuse reflection and ambient reflection are calculated at lines 40 and 41 and added to get the fragment color, which is assigned to `gl_FragColor` at line 42.

If you have more than one light source, after calculating the color due to diffuse reflection and ambient reflection for each light source, you can obtain the final fragment color by adding all the colors. In other words, you only have to calculate Equation 8.3 as many times as the number of light sources.

Summary

This chapter explored how to light a 3D scene, the different types of light used, and how light is reflected and diffused through the scene. Using this knowledge, you then implemented the effects of different light sources to illuminate a 3D object and examined various shading techniques to improve the realism of the objects. As you have seen, a mastery of lighting is essential to adding realism to 3D scenes, which can appear flat and uninteresting if they're not correctly lit.

[10] In this sample program, this normalization is not necessary because all normals are passed to `a_Normal` with a length of 1.0. However, we normalize them here as good programming practice so the code is more generic.