# Display Shadows

Chapter 8 explained shading, which is one of the phenomena when light hits an object. We briefly mentioned shadowing, another phenomena, but didn't explain how to implement it. Let's take a look at that now. There are several methods to realize shadowing, but we will explain a method that uses a **shadow map** (depth map). This method is quite expressive and used in a variety of computer graphics situations and even in special effects in movies.

## How to Implement Shadows

The shadow map method is based on the idea that the sun cannot see the shadow of objects. Essentially, it works by considering the viewer's eye point to be at the same position as the light source and determining what can be seen from that point. All the objects you can see would appear to be in the light. Anything behind those objects would be in shadow. With this method, you can use the distance to the objects (in fact, you will use the z value, which is the depth value) from the light source to judge whether the objects are visible. As you can see in Figure 10.21, where there are two points on the same line, P1 and P2, P2 is in the shadow because the distance from the light source to P2 is longer than P1.

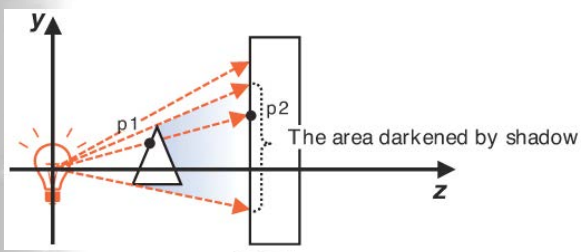


**Figure 10.21** Theory of shadow map

You need two pairs of shaders for this process: (1) a pair of shaders that calculate the distance from the light source to the objects, and (2) a pair of shaders that draws the shadow using the calculated distance. Then you need a method to pass the distance data from the light source calculated in the first pair of shaders to the second pair of shaders. You can use a texture image for this purpose. This texture image is called the **shadow map**, so this method is called **shadow mapping**. The shadow mapping technique consists of the following two processes:

1. Move the eye point to the position of the light source and draw objects from there. Because the fragments drawn from the position are hit by the light, you write the distances from the light source to each fragment in the texture image (shadow map).
2. Move the eye point back to the position from which you want to view the objects and draw them from there. Compare the distance from the light source to the fragments drawn in this step and the distance recorded in the shadow map from step

1. If the former distance is greater, you can draw the fragment as in shadow (in the darker color).

You will use the framebuffer object in step 1 to save the distance in the texture image. Therefore, the configurations of the framebuffer object used here is the same as that of `FramebufferObject.js` in Figure 10.20. You also need to switch pairs of shaders between steps 1 and 2 using the technique you learned in the section "Switching Shaders," earlier in this chapter. Now let's take a look at the sample program `Shadow`. Figure 10.22 shows a screen shot where you can see a shadow of the red triangle cast onto the slanted white rectangle.

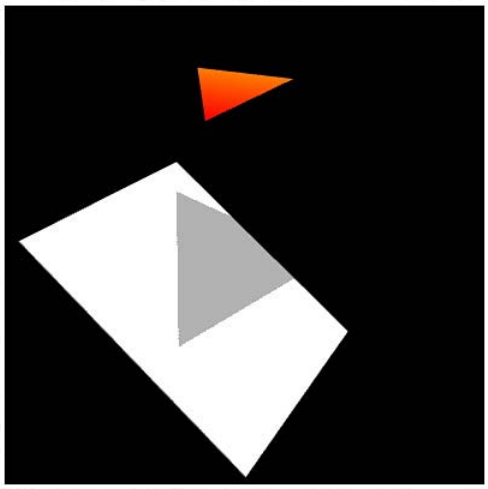

**Figure 10.22** Shadow

## Sample Program (Shadow.js)

The key aspects of shadowing take place in the shaders, which are shown in Listing 10.15.

**Listing 10.15** Shadow.js (Shader part)

```
1  // Shadow.js
2  // Vertex shader program to generate a shadow map
3  var SHADOW_VSHADER_SOURCE =
     ...
6    'void main() {\n' +
7    '  gl_Position = u_MvpMatrix * a_Position;\n' +
8    '}\n';
9
10 // Fragment shader program for creating a shadow map
11 var SHADOW_FSHADER_SOURCE =
```

```
    ...
15  'void main() {\n' +
16  '  gl_FragColor = vec4(gl_FragCoord.z, 0.0, 0.0, 0.0);\n' +          <-(1)
17  '}\n';
18
19  // Vertex shader program for regular drawing
20  var VSHADER_SOURCE =
    ...
23  'uniform mat4 u_MvpMatrix;\n' +
24  'uniform mat4 u_MvpMatrixFromLight;\n' +
25  'varying vec4 v_PositionFromLight;\n' +
26  'varying vec4 v_Color;\n' +
27  'void main() {\n' +
28  '  gl_Position = u_MvpMatrix * a_Position;\n' +
29  '  v_PositionFromLight = u_MvpMatrixFromLight * a_Position;\n' +
30  '  v_Color = a_Color;\n' +
31  '}\n';
32
33  // Fragment shader program for regular drawing
34  var FSHADER_SOURCE =
    ...
38  'uniform sampler2D u_ShadowMap;\n' +
39  'varying vec4 v_PositionFromLight;\n' +
40  'varying vec4 v_Color;\n' +
41  'void main() {\n' +
42  '  vec3 shadowCoord =(v_PositionFromLight.xyz/v_PositionFromLight.w)
                                                          ➥/ 2.0 + 0.5;\n' +
43  '  vec4 rgbaDepth = texture2D(u_ShadowMap, shadowCoord.xy);\n' +
44  '  float depth = rgbaDepth.r;\n' +  // Retrieve the z value from R
45  '  float visibility = (shadowCoord.z > depth + 0.005) ? 0.7:1.0;\n'+   <-(2)
46  '  gl_FragColor = vec4(v_Color.rgb * visibility, v_Color.a);\n' +
47  '}\n';
```

Step 1 is performed in the shader responsible for the shadow map, defined from lines 3 to 17. You just switch the drawing destination to the framebuffer object, pass a model view projection matrix in which an eye point is located at a light source to `u_MvpMatrix`, and draw the objects. This results in the distance from the light source to the fragments being written into the texture map (shadow map) attached to the framebuffer object. The vertex shader at line 7 just multiplies the model view projection matrix by the vertex coordinates to calculate this distance. The fragment shader is more complex and needs to calculate the distance from the light source to the drawn fragments. For this purpose, you can utilize the built-in variable `gl_FragCoord` of the fragment shader used in Chapter 5.

`gl_FragCoord` is a `vec4` type built-in variable that contains the coordinates of each fragment. `gl_FragCoord.x` and `gl_FragCoord.y` represents the position of the fragment on the

screen, and `gl_FragCoord.z` contains the normalized z value in the range of [0, 1]. This is calculated using `(gl_Position.z/gl.Position.w)/2.0+0.5`. (See Section 2.12 of *OpenGL ES 2.0 specification* for further details.) `gl_FragCoord.z` is specified in the range of 0.0 to 1.0, with 0.0 representing the fragments on the near clipping plane and 1.0 representing those on the far clipping plane. This value is written into the R (red) component value (any component could be used) in the shadow map at line 16.

```
16    '  gl_FragColor = vec4(gl_FragCoord.z, 0.0, 0.0, 0.0);\n'  +    <-(1)
```

Subsequently, the z value for each fragment drawn from the eye point placed at the light source is written into the shadow map. This shadow map is passed to `u_ShadowMap` at line 38.

For step 2, you need to draw the objects again after resetting the drawing destination to the color buffer and moving the eye point to its original position. After drawing the objects, you decide a fragment color by comparing the z value of the fragment with that stored in the shadow map. This is done in the normal shaders from lines 20 to 47. `u_MvpMatrix` is the model view projection matrix where the eye point is placed at the original position and `uMvpMatrixFromLight`, which was used to create the shadow map, is the model view projection matrix where the eye point is moved to the light source. The main task of the vertex shader defined at line 20 is calculating the coordinates of each fragment from the light source and passing them to the fragment shader (line 29) to obtain the z value of each fragment from the light source.

The fragment shader uses the coordinates to calculate the z value (line 42). As mentioned, the shadow map contains the value of `(gl_Position.z/gl.Position.w)/2.0+0.5`. So you could simply calculate the z value to compare with the value in the shadow map by `(v_PositionFromLight.z/v_PositionFromLight.w)/2.0+0.5`. However, because you need to get the texel value from the shadow map, line 42 performs the following extra calculation using the same operation. To compare to the value in the shadow map, you need to get the texel value from the shadow map whose texture coordinates correspond to the coordinates `(v_PositionFromLight.x, v_PositionFromLight.y)`. As you know, `v_PositionFromLight.x` and `v_PositionFromLight.y` are the x and y coordinates in the WebGL coordinate system (see Figure 2.18 in Chapter 2), and they range from –1.0 to 1.0. On the other hand, the texture coordinates s and t in the shadow map range from 0.0 to 1.0 (see Figure 5.20 in Chapter 5). So, you need to convert the x and y coordinates to the s and t coordinates. You can also do this with the same expression to calculate the z value. That is:

The texture coordinate `s` is `(v_PositionFromLight.x/v_PositionFromLight.w)/2.0 + 0.5`.

The texture coordinate `t` is `(v_PositionFromLight.y/v_PositionFromLight.w)/2.0 + 0.5`.

See also Section 2.12 of the *OpenGL ES 2.0 specification*[5] for further details about this calculation. These are carried out using the same type of calculation and can be achieved in one line, as shown at line 42:

[5] www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf

```
42    '  vec3 shadowCoord =(v_PositionFromLight.xyz/v_PositionFromLight.w)
                                                          ➥/ 2.0 + 0.5;\n' +
43    '  vec4 rgbaDepth = texture2D(u_ShadowMap, shadowCoord.xy);\n' +
44    '  float depth = rgbaDepth.r;\n' +  // Retrieve the z value from R
```

You retrieve the value from the shadow map at lines 43 and 44. Only the `R` value is retrieved using `rgbaDepth.r` at line 44 because you wrote it into the R component at line 16. Line 45 checks whether that fragment is in the shadow. When the position of the fragment is determined to be greater than the depth (that is, `shadowCoord.z > depth`), a value of 0.7 is stored in `visibility`. The `visibility` is used at line 46 to draw the shadow with a darker color:

```
45    '  float visibility = (shadowCoord.z > depth + 0.005) ? 0.7:1.0;\n'+
46    '  gl_FragColor = vec4(v_Color.rgb * visibility, v_Color.a);\n' +
```

Line 45 adds a small offset of 0.005 to the depth value. To understand why this is needed, try running the sample program without this number. You will see a striped pattern as shown in Figure 10.23, referred to as the **Mach band.**

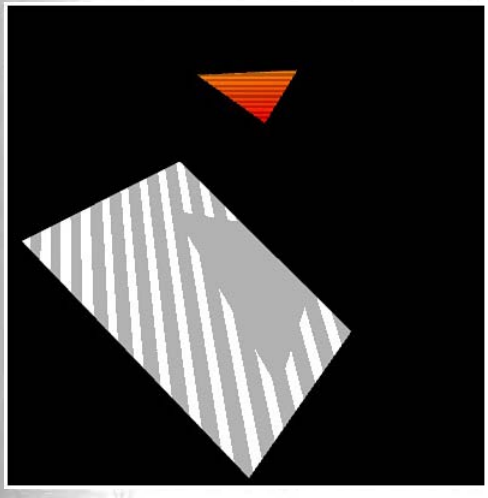

**Figure 10.23** Striped pattern

The value of 0.005 is added to suppress the stripe pattern. The stripe pattern occurs because of the precision of the numbers you can store in the RGBA components. It's a little complex, but it's worth understanding because this problem occurs elsewhere in 3D graphics. The z value of the shadow map is stored in the R component of RGBA in the texture map, which is an 8-bit number. This means that the precision of R is lower than its comparison target (`shadowCoord.z`), which is of type `float`. For example, let the z value simply be 0.1234567. If you represent the value using 8 bits, in other words using 256

possibilities, you can represent the value in a precision of 1/256 (=0.0390625). So you can represent 0.1234567 as follows:

```
0.1234567 / (1 / 256) = 31.6049152
```

Numbers below the decimal point cannot be used in 8 bits, so only 31 can be stored in 8 bits. When you divide 31 by 256, you obtain 0.12109375 which, as you can see, is smaller than the original value (0.1234567). This means that even if the fragment is at the same position, its z value stored in the shadow map becomes smaller than its z value in `shadowCoord.z`. As a result, the z value in `shadowCoord.z` becomes larger than that in the shadow map according to the position of the fragment resulting in the stripe patterns. Because this happens because the precision of the R value is 1/256 (=0.00390625), by adding a small offset, such as 0.005, to the R value, you can stop the stripe pattern from appearing. Note that any offset greater than 1/256 will work; 0.005 was chosen because it is 1/256 plus a small margin.

Next, let's look at the JavaScript program that passes the data to the shader (see Listing 10.16) with a focus on the type of transformation matrices passed. To draw a shadow clearly, the size of a texture map for the offscreen rendering defined at line 49 is larger than that of the `<canvas>`.

**Listing 10.16** Shadow.js (JavaScript Part)

```
49  var OFFSCREEN_WIDTH = 1024, OFFSCREEN_HEIGHT = 1024;
50  var LIGHT_X = 0, LIGHT_Y = 7, LIGHT_Z = 2;
51
52  function main() {
      ...
63    // Initialize shaders for generating a shadow map
64    var shadowProgram = createProgram(gl, SHADOW_VSHADER_SOURCE,
                                                    ➥SHADOW_FSHADER_SOURCE);
      ...
72    // Initialize shaders for regular drawing
73    var normalProgram = createProgram(gl, VSHADER_SOURCE, FSHADER_SOURCE);
      ...
85    // Set vertex information
86    var triangle = initVertexBuffersForTriangle(gl);
87    var plane = initVertexBuffersForPlane(gl);
      ...
93    // Initialize a framebuffer object (FBO)
94    var fbo = initFramebufferObject(gl);
      ...
99    gl.activeTexture(gl.TEXTURE0); // Set a texture object to the texture unit
100   gl.bindTexture(gl.TEXTURE_2D, fbo.texture);
      ...
106   var viewProjMatrixFromLight = new Matrix4(); // For the shadow map
```

```
107   viewProjMatrixFromLight.setPerspective(70.0,
                              ➥OFFSCREEN_WIDTH/OFFSCREEN_HEIGHT, 1.0, 100.0);
108   viewProjMatrixFromLight.lookAt(LIGHT_X, LIGHT_Y, LIGHT_Z, 0.0, 0.0, 0.0, 0.0,
                                                                  ➥1.0, 0.0);
109
110   var viewProjMatrix = new Matrix4(); // For regular drawing
111   viewProjMatrix.setPerspective(45, canvas.width/canvas.height, 1.0, 100.0);
112   viewProjMatrix.lookAt(0.0, 7.0, 9.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
113
114   var currentAngle = 0.0; // Current rotation angle [degrees]
115   var mvpMatrixFromLight_t = new Matrix4();  // For triangle
116   var mvpMatrixFromLight_p = new Matrix4();  // For plane
117   var tick = function() {
118     currentAngle = animate(currentAngle);
119     // Change the drawing destination to FBO
120     gl.bindFramebuffer(gl.FRAMEBUFFER, fbo);
      ...
124     gl.useProgram(shadowProgram); // For generating a shadow map
125     // Draw the triangle and the plane (for generating a shadow map)
126     drawTriangle(gl, shadowProgram, triangle, currentAngle,
                                                  ➥viewProjMatrixFromLight);
127     mvpMatrixFromLight_t.set(g_mvpMatrix);  // Used later
128     drawPlane(gl, shadowProgram, plane, viewProjMatrixFromLight);
129     mvpMatrixFromLight_p.set(g_mvpMatrix);  // Used later
130     // Change the drawing destination to color buffer
131     gl.bindFramebuffer(gl.FRAMEBUFFER, null);
      ...
135     gl.useProgram(normalProgram); // For regular drawing
136     gl.uniform1i(normalProgram.u_ShadowMap, 0); // Pass gl.TEXTURE0
137     // Draw the triangle and plane (for regular drawing)
138     gl.uniformMatrix4fv(normalProgram.u_MvpMatrixFromLight, false,
                                              ➥mvpMatrixFromLight_t.elements);
139     drawTriangle(gl, normalProgram, triangle, currentAngle, viewProjMatrix);
140     gl.uniformMatrix4fv(normalProgram.u_MvpMatrixFromLight, false,
                                              ➥mvpMatrixFromLight_p.elements);
141     drawPlane(gl, normalProgram, plane, viewProjMatrix);
142
143     window.requestAnimationFrame(tick, canvas);
144   };
145   tick();
146 }
```

Let's look at the `main()` function from line 52 in the JavaScript program. Line 64 initializes the shaders for generating the shadow map. Line 73 initializes the shaders for normal drawing. Lines 86 and 87, which set up the vertex information and

`initFramebufferObject()` at line 94, are the same as the `FramebufferObject.js`. Line 94 prepares a framebuffer object, which contains the texture object for a shadow map. Lines 99 and 100 enable texture unit 0 and bind it to the target. This texture unit is passed to `u_ShadowMap` in the shaders for normal drawing.

Lines 106 to 108 prepare a view projection matrix to generate a shadow map. The key point is that the first three arguments (that is, the position of an eye point) at line 108 are specified as the position of the light source. Lines 110 to 112 prepare the view projection matrix from the eye point where you want to view the scene.

Finally, you draw the triangle and plane using all the preceding information. First you generate the shadow map, so you switch the drawing destination to the framebuffer object at line 120. You draw the objects by using the shaders for generating a shadow map (`shadowProgram`) at lines 126 and 128. You should note that lines 127 and 129 save the model view projection matrices from the light source. Then the shadow map is generated, and you use it to draw shadows with the code from line 135. Line 136 passes the map to the fragment shader. Lines 138 and 140 pass the model view projection matrices saved at line 127 and 129, respectively, to `u_MvpMatrixFromLight`.

## Increasing Precision

Although you've successfully calculated the shadow and drawn the scene with the shadow included, the example code is only able to handle situations in which the light source is close to the object. To see this, let's change the y coordinate of the light source position to 40:

```
50  var LIGHT_X = 0, LIGHT_Y = 40, LIGHT_Z = 2;
```

If you run the modified sample program, you can see that the shadow is not displayed—as in the left side of Figure 10.24. Obviously, you want the shadow to be displayed correctly, as in the figure on the right.

The reason the shadow is no longer displayed when the distance from the light source to the object is increased is that the value of `gl_FragCoord.z` could not be stored in the R component of the texture map because it has only an 8-bit precision. A simple solution to this problem is to use not just the R component but the B, G, and A components. In other words, you save the value separately in 4 bytes. There is a routine procedure to do this, so let's see the sample program. Only the fragment shader is changed.

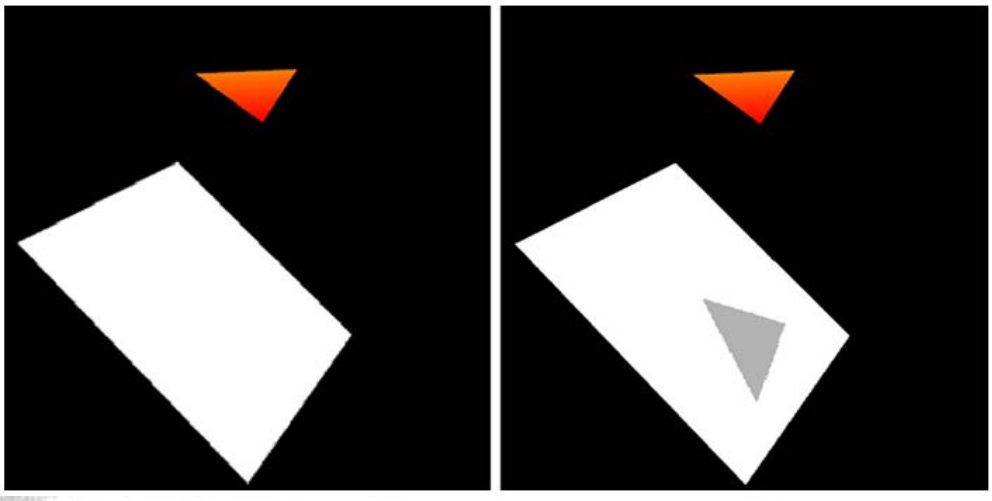

**Figure 10.24** The shadow is not displayed

## Sample Program (Shadow_highp.js)

Listing 10.17 shows the fragment shader of `Shadow_highp.js`. You can see that the processing to handle the z value is more complex than that in `Shadow.js`.

**Listing 10.17** Shadow_highp.js

```
 1  // Shadow_highp.js
       ...
10  // Fragment shader program for creating a shadow map
11  var SHADOW_FSHADER_SOURCE =
       ...
15    'void main() {\n' +
16    '  const vec4 bitShift = vec4(1.0, 256.0, 256.0 * 256.0, 256.0 * 256.0 *
                                                                  ➥256.0);\n' +
17    '  const vec4 bitMask = vec4(1.0/256.0, 1.0/256.0, 1.0/256.0, 0.0);\n' +
18    '  vec4 rgbaDepth = fract(gl_FragCoord.z * bitShift);\n' +
19    '  rgbaDepth -= rgbaDepth.gbaa * bitMask;\n' +
20    '  gl_FragColor = rgbaDepth;\n' +
21    '}\n';
       ...
37  // Fragment shader program for regular drawing
38  var FSHADER_SOURCE =
       ...
45    // Recalculate the z value from the rgba
46    'float unpackDepth(const in vec4 rgbaDepth) {\n' +
47    '  const vec4 bitShift = vec4(1.0, 1.0/256.0, 1.0/(256.0 * 256.0),
                                          ➥1.0/(256.0 * 256.0 * 256.0));\n' +
```

```
48    '  float depth = dot(rgbaDepth, bitShift);\n' +
49    '  return depth;\n' +
50    '}\n' +
51    'void main() {\n' +
52    '  vec3 shadowCoord = (v_PositionFromLight.xyz /
                                        ➥v_PositionFromLight.w)/2.0 + 0.5;\n' +
53    '  vec4 rgbaDepth = texture2D(u_ShadowMap, shadowCoord.xy);\n' +
54    '  float depth = unpackDepth(rgbaDepth);\n' +  // Recalculate the z
55    '  float visibility = (shadowCoord.z > depth + 0.0015)? 0.7:1.0;\n'+
56    '  gl_FragColor = vec4(v_Color.rgb * visibility, v_Color.a);\n' +
57    '}\n';
```

The code that splits `gl_FragCoord.z` into 4 bytes (RGBA) is from lines 16 to 19. Because 1 byte can represent up to 1/256, you can store the value greater than 1/256 in R, the value less than 1/256 and greater than 1/(256*256) in G, the value less than 1/(256*256) and greater than 1/(256*256*256) in B, and the rest of value in A. Line 18 calculates each value and stores it in the RGBA components, respectively. It can be written in one line using a `vec4` data type. The function `fract()` is a built-in one that discards numbers below the decimal point for the value specified as its argument. Each value in `vec4`, calculated at line 18, has more precision than 1 byte, so line 19 discards the value that does not fit in 1 byte. By substituting this result to `gl_FragColor` at line 20, you can save the z value using all four components of the RGBA type and achieve higher precision.

`unpackDepth()` at line 54 reads out the z value from the RGBA. This function is defined at line 46. Line 48 performs the following calculation to convert the RGBA value to the original z value. As you can see, the calculation is the same as the inner product, so you use `dot()` at line 48.

$$depth = rgbDepth.r \times 1.0 + \frac{rgbaDepth.g}{256.0} + \frac{rgbaDepth.b}{(256.0 \times 256.0)} + \frac{rgbaDepth.a}{(256.0 \times 256.0 \times 256.0)}$$

Now you have retrieved the distance (z value) successfully, so you just have to draw the shadow by comparing the distance with `shadowCoord.z` at line 55. In this case, 0.0015 is used as the value for adjusting the error (the stripe pattern), instead of 0.005. This is because the precision of the z value stored in the shadow map is a `float` type of `medium` precision (that is, its precision is $2^{-10} = 0.000976563$, as shown in Table 6.15 in Chapter 6). So you add a little margin to it and chose 0.0015 as the value. After that, the shadow can be drawn correctly.

# Load and Display 3D Models

In the previous chapters, you drew 3D objects by specifying their vertex coordinates and color information by hand and stored them in arrays of type `Float32Array` in the JavaScript program. However, as mentioned earlier in the book, in most cases you will actually read the vertex coordinates and color information from 3D model files constructed by a 3D modeling tool.