The depth buffer was briefly introduced in Chapter 7. The z values of fragments (which are normalized to a value between 0.0 and 1.0) are written into the buffer. For example, say there are two triangles on top of each other and you draw from the triangle on top. First, the z value of the triangle on top is written into the depth buffer. Then, when the triangle on bottom is drawn, the hidden surface removal function compares the z value of its fragment that is going to be drawn, with the z value already written in the depth buffer. Then only when the z value of the fragment that is going to be drawn is smaller than the existing value in the buffer (that is, when it's closer to the eye point) will the fragment be drawn into the color buffer. This approach ensures that hidden surface removal is achieved. Therefore, after drawing, the z value of the fragment of the surface that can be seen from the eye point is left in the depth buffer.

Opaque objects are drawn into the color buffer in the correct order by removing the hidden surfaces in the processing of steps 1 and 2, and the z value that represents the order is written in the depth buffer. Transparent objects are drawn into the color buffer using that z value in steps 3, 4, and 5, so the hidden surfaces of the transparent objects behind the opaque objects will be removed. This results in the correct image being shown where both objects coexist.

Switching Shaders

The sample programs in this book draw using a single vertex shader and a single fragment shader. If all objects can be drawn with the same shaders, there is no problem. However, if you want to change the drawing method for each object, you need to add significant complexity to the shaders to achieve multiple effects. A solution is to prepare more than one shader and then switch between these shaders as required. Here, you construct a sample program, `ProgramObject`, which draws a cube colored with a single color and another cube with a texture image. Figure 10.16 shows a screen shot.
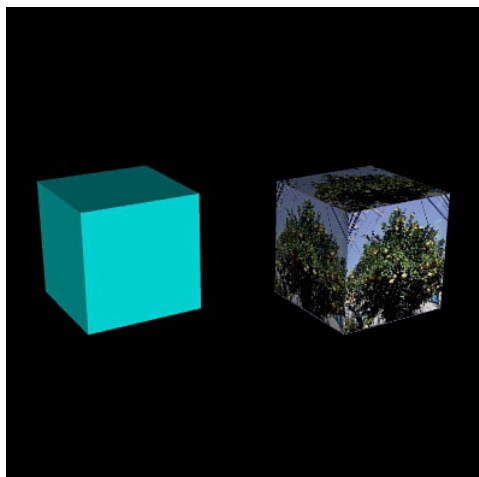

Figure 10.16   A screen shot of ProgramObject

This program is also an example of the shading of an object with a texture image.

How to Implement Switching Shaders

The shaders can be switched easily by creating program objects, as explained in Chapter 8, "Lighting Objects," and switching them before drawing. Switching is carried out using the function gl.useProgram(). Because you are explicitly manipulating shader objects, you cannot use the convenience function initShaders(). However, you can use the function createProgram() in cuon-utils.js, which is called from initShaders().

The following is the processing flow of the sample program. It performs the same procedure twice, so it looks long, but the essential code is simple:

1. Prepare the shaders to draw an object shaded with a single color.

2. Prepare the shaders to draw an object with a texture image.

3. Create a program object that has the shaders from step 1 with createProgram().

4. Create a program object that has the shaders from step 2 with createProgram().

5. Specify the program object created by step 3 with gl.useProgram().

6. Enable the buffer object after assigning it to the attribute variables.

7. Draw a cube (drawn in a single color).

8. Specify the program object created in step 4 using gl.useProgram().

9. Enable the buffer object after assigning it to the attribute variables.

10. Draw a cube (texture is pasted).

Now let's look at the sample program.

Sample Program (ProgramObject.js)

The key program code for steps 1 to 4 is shown in Listing 10.11. Two types of vertex shader and fragment shader are prepared: SOLID_VSHADER_SOURCE (line 3) and SOLID_FSHADER_SOURCE (line 19) to draw an object in a single color, and TEXTURE_VSHADER_SOURCE (line 29) and TEXTURE_FSHADER_SOURCE (line 46) to draw an object with a texture image. Because the focus here is on how to switch the program objects, the contents of the shaders are omitted.

Listing 10.11 ProgramObject (Process for Steps 1 to 4)

```
 1  // ProgramObject.js
 2  // Vertex shader for single color drawing                              <- (1)
 3  var SOLID_VSHADER_SOURCE =
      ...
18  // Fragment shader for single color drawing
```

```
19   var SOLID_FSHADER_SOURCE =
       ...
28   // Vertex shader for texture drawing                              <- (2)
29   var TEXTURE_VSHADER_SOURCE =
       ...
45   // Fragment shader for texture drawing
46   var TEXTURE_FSHADER_SOURCE =
       ...
58   function main() {
       ...
69     // Initialize shaders
70     var solidProgram = createProgram(gl, SOLID_VSHADER_SOURCE,
                                            ➥SOLID_FSHADER_SOURCE);   <- (3)
71     var texProgram = createProgram(gl, TEXTURE_VSHADER_SOURCE,
                                            ➥TEXTURE_FSHADER_SOURCE); <- (4)
       ...
77     // Get the variables in the program object for single color drawing
78     solidProgram.a_Position = gl.getAttribLocation(solidProgram, 'a_Position');
79     solidProgram.a_Normal = gl.getAttribLocation(solidProgram, 'a_Normal');
       ...
83     // Get the storage location of attribute/uniform variables
84     texProgram.a_Position = gl.getAttribLocation(texProgram, 'a_Position');
85     texProgram.a_Normal = gl.getAttribLocation(texProgram, 'a_Normal');
       ...
89     texProgram.u_Sampler = gl.getUniformLocation(texProgram, 'u_Sampler');
       ...
99     // Set vertex information
100    var cube = initVertexBuffers(gl, solidProgram);
       ...
106    // Set texture
107    var texture = initTextures(gl, texProgram);
       ...
122    // Start drawing
123    var currentAngle = 0.0; // Current rotation angle (degrees)
124    var tick = function() {
125      currentAngle = animate(currentAngle);  // Update rotation angle
       ...
128      // Draw a cube in single color
129      drawSolidCube(gl, solidProgram, cube, -2.0, currentAngle, viewProjMatrix);
130      // Draw a cube with texture
131      drawTexCube(gl, texProgram, cube, texture, 2.0, currentAngle,
                                                      ➥viewProjMatrix);
132
133      window.requestAnimationFrame(tick, canvas);
```

```
134     };
135     tick();
136   }
137
138   function initVertexBuffers(gl, program) {
      ...
148    var vertices = new Float32Array([   // Vertex coordinates
149     1.0, 1.0, 1.0, -1.0, 1.0, 1.0, -1.0,-1.0, 1.0,  1.0,-1.0, 1.0,
150     1.0, 1.0, 1.0,  1.0,-1.0, 1.0,  1.0,-1.0,-1.0,  1.0, 1.0,-1.0,
      ...
154     1.0,-1.0,-1.0, -1.0,-1.0,-1.0, -1.0, 1.0,-1.0,  1.0, 1.0,-1.0
155     ]);
156
157     var normals = new Float32Array([   // Normal
      ...
164     ]);
165
166     var texCoords = new Float32Array([ // Texture coordinates
      ...
173     ]);
174
175     var indices = new Uint8Array([     // Indices for vertices
      ...
182     ]);
183
184     var o = new Object(); // Use Object to return buffer objects
185
186     // Write vertex information to buffer object
187     o.vertexBuffer = initArrayBufferForLaterUse(gl, vertices, 3, gl.FLOAT);
188     o.normalBuffer = initArrayBufferForLaterUse(gl, normals, 3, gl.FLOAT);
189     o.texCoordBuffer = initArrayBufferForLaterUse(gl, texCoords, 2, gl.FLOAT);
190     o.indexBuffer = initElementArrayBufferForLaterUse(gl, indices,
                                                  ➥gl.UNSIGNED_BYTE);
      ...
193     o.numIndices = indices.length;
      ...
199     return o;
200   }
```

Starting with the `main()` function in JavaScript, you first create a program object for
each shader with `createProgram()` at lines 70 and 71. The arguments of the `createPro-`
`gram()` are the same as the `initShaders()`, and the return value is the program object.
You save each program object in `solidProgram` and `texProgram`. Then you retrieve the
storage location of the attribute and uniform variables for each shader at lines 78 to 89.
You will store them in the corresponding properties of the program object, as you did in

`MultiJointModel_segment.js`. Again, you leverage JavaScript's ability to freely append a new property of any type to an object.

The vertex information is then stored in the buffer object by `initVertexBuffers()` at line 100. You need (1) vertex coordinates, (2) the normals, and (3) indices for the shader to draw objects in a single color. In addition, for the shader to draw objects with a texture image, you need the texture coordinates. The processing in `initVertexBuffers()` handles this and binds the correct buffer object to the corresponding attribute variables when the program object is switched.

`initVertexBuffers()` prepares the vertex coordinates from line 148, normals from line 157, texture coordinates from line 166, and index arrays from line 175. Line 184 creates object (`o`) of type `Object`. Then you store the buffer object to the property of the object (lines 187 to 190). You can maintain each buffer object as a global variable, but that introduces too many variables and makes it hard to understand the program. By using properties, you can more conveniently manage all four buffer objects using one object `o`.[4]

You use `initArrayBufferForLaterUse()`, explained in `MultiJointModel_segment.js`, to create each buffer object. This function writes vertex information into the buffer object but does not assign it to the attribute variables. You use the buffer object name as its property name to make it easier to understand. Line 199 returns the object `o` as the return value.

Once back in `main()` in JavaScript, the texture image is set up in `initTextures()` at line 107, and then everything is ready to allow you to draw the two cube objects. First, you draw a single color cube using `drawSolidCube()` at line 129, and then you draw a cube with a texture image by using `drawTexCube()` at line 131. Listing 10.12 shows the latter half of the steps, steps 5 through 10.

Listing 10.12 ProgramObject.js (Processes for Steps 5 through 10)

```
236  function drawSolidCube(gl, program, o, x, angle, viewProjMatrix) {
237    gl.useProgram(program);    // Tell this program object is used           <-(5)
238
239    // Assign the buffer objects and enable the assignment                   <-(6)
240    initAttributeVariable(gl, program.a_Position, o.vertexBuffer);
241    initAttributeVariable(gl, program.a_Normal, o.normalBuffer);
242    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, o.indexBuffer);
243
244    drawCube(gl, program, o, x, angle, viewProjMatrix);   // Draw            <-(7)
245  }
246
247  function drawTexCube(gl, program, o, texture, x, angle, viewProjMatrix) {
```

---

[4] To keep the explanation simple, the object (`o`) was used. However, it is better programming practice to create a new user-defined type for managing the information about a buffer object and to use it to manage the four buffers.

```
248    gl.useProgram(program);   // Tell this program object is used <-(8)
249
250    // Assign the buffer objects and enable the assignment        <-(9)
251    initAttributeVariable(gl, program.a_Position, o.vertexBuffer);
252    initAttributeVariable(gl, program.a_Normal, o.normalBuffer);
253    initAttributeVariable(gl, program.a_TexCoord, o.texCoordBuffer);
254    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, o.indexBuffer);
255
256    // Bind texture object to texture unit 0
257    gl.activeTexture(gl.TEXTURE0);
258    gl.bindTexture(gl.TEXTURE_2D, texture);
259
260    drawCube(gl, program, o, x, angle, viewProjMatrix); // Draw  <-(10)
261  }
262
263  // Assign the buffer objects and enable the assignment
264  function initAttributeVariable(gl, a_attribute, buffer) {
265    gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
266    gl.vertexAttribPointer(a_attribute, buffer.num, buffer.type, false, 0, 0);
267    gl.enableVertexAttribArray(a_attribute);
268  }
       ...
275  function drawCube(gl, program, o, x, angle, viewProjMatrix) {
276    // Calculate a model matrix
       ...
281    // Calculate transformation matrix for normal
       ...
286    // Calculate a model view projection matrix
       ...
291    gl.drawElements(gl.TRIANGLES, o.numIndices, o.indexBuffer.type, 0);
292  }
```

drawSolidCube() is defined at line 236 and uses gl.useProgram() at line 237 to tell
the WebGL system that you will use the program (program object, solidProgram)
specified by the argument. Then you can draw using solidProgram. The buffer objects
for vertex coordinates and normals are assigned to attribute variables and enabled by
initAttributeVariable() at lines 240 and 241. This function is defined at line 264. Line
242 binds the buffer object for the indices to gl.ELEMENT_ARRAY_BUFFER. With everything
set up, you then call drawCube() at line 244, which uses gl.drawElements() at line 291 to
perform the draw operation.

drawTexCube(), defined at line 247, follows the same steps as drawSolidCube(). Line 253
is added to assign the buffer object for texture coordinates to the attribute variables, and
lines 257 and 258 are added to bind the texture object to the texture unit 0. The actual
drawing is performed in drawCube(), just like drawSolidCube().

Once you've mastered this basic technique, you can use it to switch between any number of shader programs. This way you can use a variety of different drawing effects in a single scene.

Use What You've Drawn as a Texture Image

One simple but powerful technique is to draw some 3D objects and then use the resulting image as a texture image for another 3D object. Essentially, if you can use the content you've drawn as a texture image, you are able to generate images on-the-fly. This means you do not need to download images from the network, and you can apply special effects (such as motion blur and depth of field) before displaying the image. You can also use this technique for shadowing, which will be explained in the next section. Here, you will construct a sample program, `FramebufferObject`, which maps a rotating cube drawn with WebGL to a rectangle as a texture image. Figure 10.17 shows a screen shot.
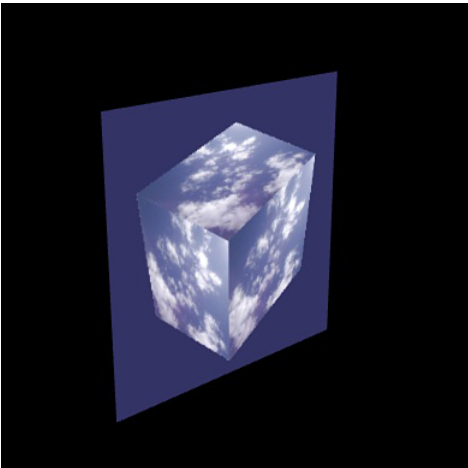

Figure 10.17 FramebufferObject

If you actually run the program, you can see a rotating cube with a texture image of a summer sky pasted to the rectangle as its texture. Significantly, the image of the cube that is pasted on the rectangle is not a movie prepared in advance but a rotating cube drawn by WebGL in real time. This is quite powerful, so let's take a look at what WebGL must do to achieve this.

Framebuffer Object and Renderbuffer Object

By default, the WebGL system draws using a color buffer and, when using the hidden surface removal function, a depth buffer. The final image is kept in the color buffer.

The **framebuffer object** is an alternative mechanism you can use instead of a color buffer or a depth buffer (Figure 10.18). Unlike a color buffer, the content drawn in a framebuffer