```
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.

In this section, you construct a sample program that reads a 3D model constructed using a 3D modeling tool. For this example, we use the Blender[6] modeling tool, which is a popular tool with a free version available. Blender is able to export 3D model files using the well-known OBJ format, which is text based and easy to read, understand, and parse. OBJ is a geometry definition file format originally developed by Wavefront Technologies. This file format is open and has been adopted by other 3D graphics vendors. Although this means it is reasonably well known and used, it also means that there are a number of variations in the format. To simplify the example code, we have made a number of assumptions, such as not using textures. However, the example gives you a good understanding of how to read model data into your programs and provides a basis for you to begin experimentation. The approach taken in the example code is designed to be reasonably generic and can be used for other text-based formats.

Start Blender and create a cube like that shown in Figure 10.25. The color of one face of this cube is orange, and the other faces are red. Then export the model to a file named `cube.obj`. (You can find an example of it in the `resources` directory with the sample programs.) Let's take a look at `cube.obj`, which, because it is a text file, can be opened with a simple text editor.

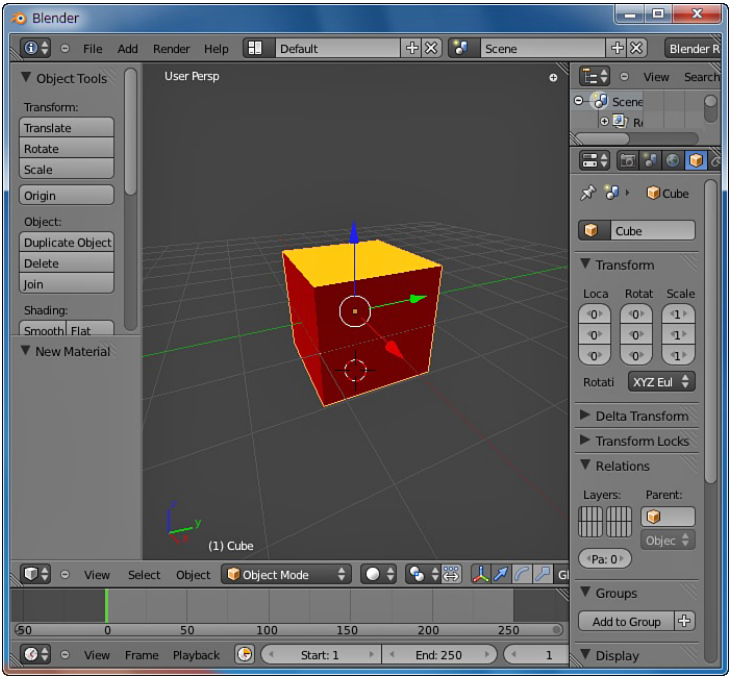

Figure 10.25 Blender, 3D modeling tool

6. www.blender.org/

Figure 10.26 shows the contents of `cube.obj`. Line numbers have been added to help with the explanation and would not normally be in the file.

```
1  # Blender v2.60 (sub 0) OBJ File: ''
2  # www.blender.org
3  mtllib cube.mtl
4  o Cube
5  v 1.000000 -1.000000 -1.000000
6  v 1.000000 -1.000000 1.000000
7  v -1.000000 -1.000000 1.000000
8  v -1.000000 -1.000000 -1.000000
9  v 1.000000 1.000000 -1.000000
0  v 1.000000 1.000000 1.000001
1  v -1.000000 1.000000 1.000000
2  v -1.000000 1.000000 -1.000000
3  usemtl Material
4  f 1 2 3 4
5  f 5 8 7 6
6  f 2 6 7 3
7  f 3 7 8 4
8  f 5 1 4 8
9  usemtl Material.001
0  f 1 5 6 2
```

Figure 10.26 cube.obj

Once the model file has been created by the modeling tool, your program needs to read the data and store it in the same type of data structures that you've used before. The following steps are required:

1. Prepare the array (`vertices`) of type `Float32Array` and read the vertex coordinates of the model from the file into the array.
2. Prepare the array (`colors`) of type `Float32Array` and read the colors of the model from the file into the array.
3. Prepare the array (`normals`) of type `Float32Array` and read the normals of the model form the file into the array.
4. Prepare the array (`indices`) of type `Uint16Array` (or `Uint8Array`) and read the indices of the vertices that specify the triangles that make up the model from the file into the array.
5. Write the data read during steps 1 through 4 into the buffer object and then draw the model using `gl.drawElements()`.

So in this case, you read the data described in `cube.obj` (shown in Figure 10.26) in the appropriate arrays and then draw the model in step 5. Reading data from the file requires understanding the format of the file `cube.obj` (referred to as the OBJ file).

The OBJ File Format

An OBJ file is made up of several sections,[7] including vertex positions, face definitions, and material definitions. There may be multiple vertices, normals, and faces within their sections:

- Lines beginning with a hash character (#) are comments. Lines 1 and 2 in Figure 10.26 are comments generated by Blender describing its version number and origin. The remaining lines define the 3D model.
- Line 3 references an external materials file. The OBJ format maintains the material information of the model in an external material file called an MTL file.

  `mtllib` <external mtl filename>

  specifies that the materials file is `cube.mtl`.
- Line 4 specifies the named object in the following format:

  <object name>

  This sample program does not use this information.
- Lines 5 to 12 define vertex positions in the following format using (x,y,z[,w]) coordinates, where w is optional and defaults to 1.0.

  `v` x y z [w]

  In this example, it has eight vertices because the model is a standard cube.
- Lines 13 to 20 specify a material and the faces that use the material. Line 13 specifies the material name, as defined in the MTL file referenced at line 4, and the specific material using the following format:

  `usemtl` <material name>
- The following lines, 14 to 18, define faces of the model and the material to be applied to them. Faces are defined using lists of vertex, texture, and normal indices.

  `f` v1 v2 v3 v4 ...

  v1, v2, v3, ... are the vertex indices starting from 1 and matching the corresponding vertex elements of a previously defined vertex list. This sample program handles vertex and normals. Figure 10.26 does not contain normals, but if a face has a normal, the following format would be used:

  `f` v1//vn1 v2//vn2 v3//vn3 ...

  vn1, vn2, vn3, ... are the normal indices starting from 1.

[7] See http://en.wikipedia.org/wiki/Wavefront_.obj_file

The MTL File Format

The MTL file may define multiple materials. Figure 10.27 shows `cube.mtl`.

```
 1 # Blender MTL File: ''
 2 # Material Count: 2
 3 newmtl Material
 4 Ka 0.000000 0.000000 0.000000
 5 Kd 1.000000 0.000000 0.000000
 6 Ks 0.000000 0.000000 0.000000
 7 Ns 96.078431
 8 Ni 1.000000
 9 d 1.000000
10 illum 0
11 newmtl Material.001
12 Ka 0.000000 0.000000 0.000000
13 Kd 1.000000 0.450000 0.000000
14 Ks 0.000000 0.000000 0.000000
15 Ns 96.078431
16 Ni 1.000000
17 d 1.000000
18 illum 0
```

Figure 10.27 cube.mtl

- Lines 1 and 2 are comments that Blender generates.
- Each new material (from line 3) starts with the `newmtl` command:

  `newmtl <material name>`

  This is the material name that is used in the OBJ file.
- Lines 4 to 6 define the ambient, diffuse, and specular color using `Ka`, `Kd`, and `Ks`, respectively. Color definitions are in RGB format, where each component is between 0 and 1. This sample program uses only diffuse color.
- Line 7 specifies the weight of the specular color using `Ns`. Line 8 specifies the optical density for the surface using `Ni`. Line 9 specifies transparency using `d`. Line 10 specifies illumination models using `illum`. The sample program does not use this item of information.

Given this understanding of the structure of the OBJ and MTL files, you have to extract the vertex coordinates, colors, normals, and indices describing a face from the file, write them into the buffer objects, and draw with `gl.drawElements()`. The OBJ file may not have the information on normals, but you can calculate them from the vertex coordinates that make up a face by using a "cross product."[8]

Let's look at the sample program.

[8] If the vertices of a triangle are v0, v1, and v2, the vector of v0 and v1 is (x1, y1, z1), and the vector of v1 and v2 is (x2, y2, z2), then the cross product is defined as (y1*z2 – z1*y2, z1*x2 – x1*z2, x1*y2 – y1*z2). The result will be the normal for the triangle. (See the book *3D Math Primer for Graphics and Game Development*.)

Sample Program (OBJViewer.js)

The basic steps are as follows: (1) prepare an empty buffer object, (2) read an OBJ file (an MTL file), (3) parse it, (4) write the results into the buffer object, and (5) draw. These steps are implemented as shown in Listing 10.18.

Listing 10.18 OBJViewer.js

```
 1 // OBJViewer.js (
   ...
28 function main() {
   ...
40   if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
41     console.log('Failed to initialize shaders.');
42     return;
43   }
   ...
49   // Get the storage locations of attribute and uniform variables
50   var program = gl.program;
51   program.a_Position = gl.getAttribLocation(program, 'a_Position');
52   program.a_Normal = gl.getAttribLocation(program, 'a_Normal');
53   program.a_Color = gl.getAttribLocation(program, 'a_Color');
   ...
63   // Prepare empty buffer objects for vertex coordinates, colors, and normals
64   var model = initVertexBuffers(gl, program);
   ...
75   // Start reading the OBJ file
76   readOBJFile('../resources/cube.obj', gl, model, 60, true);
   ...
81     draw(gl, gl.program, currentAngle, viewProjMatrix, model);
   ...
85 }
86
87 // Create a buffer object and perform the initial configuration
88 function initVertexBuffers(gl, program) {
89   var o = new Object();
90   o.vertexBuffer = createEmptyArrayBuffer(gl, program.a_Position, 3, gl.FLOAT);
91   o.normalBuffer = createEmptyArrayBuffer(gl, program.a_Normal, 3, gl.FLOAT);
92   o.colorBuffer = createEmptyArrayBuffer(gl, program.a_Color, 4, gl.FLOAT);
93   o.indexBuffer = gl.createBuffer();
   ...
98   return o;
99 }
100
101 // Create a buffer object, assign it to attribute variables, and enable the
                                                                    ➥assignment
```

```
102 function createEmptyArrayBuffer(gl, a_attribute, num, type) {
103   var buffer =  gl.createBuffer();  // Create a buffer object
    ...
108   gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
109   gl.vertexAttribPointer(a_attribute, num, type, false, 0, 0);
110   gl.enableVertexAttribArray(a_attribute);  // Enable the assignment
111
112   return buffer;
113 }
114
115 // Read a file
116 function readOBJFile(fileName, gl, model, scale, reverse) {
117   var request = new XMLHttpRequest();
118
119   request.onreadystatechange = function() {
120     if (request.readyState === 4 && request.status !== 404) {
121       onReadOBJFile(request.responseText, fileName, gl, model, scale, reverse);
122     }
123   }
124   request.open('GET', fileName, true); // Create a request to get file
125   request.send();                      // Send the request
126 }
127
128 var g_objDoc = null;      // The information of OBJ file
129 var g_drawingInfo = null; // The information for drawing 3D model
130
131 // OBJ file has been read
132 function onReadOBJFile(fileString, fileName, gl, o, scale, reverse) {
133   var objDoc = new OBJDoc(fileName);  // Create a OBJDoc object
134   var result = objDoc.parse(fileString, scale, reverse);
135   if (!result) {
136     g_objDoc = null; g_drawingInfo = null;
137     console.log("OBJ file parsing error.");
138     return;
139   }
140   g_objDoc = objDoc;
141 }
```

Within the JavaScript, the processing in `initVertexBuffers()`, called at line 64, has been changed. The function simply prepares an empty buffer object for the vertex coordinates, colors, and normals for the 3D model to be displayed. After parsing the OBJ file, the information corresponding to each buffer object will be written in the object.

The initVertexBuffers() function at line 88 creates the appropriate empty buffer objects at lines 90 to 92 using createEmptyArrayBuffer() and assigns them to an attribute variable. This function is defined at line 102 and, as you can see, creates a buffer object (line 103), assigns it to an attribute variable (line 109), and enables the assignment (line 110), but it does not write the data. After storing these buffer objects to model at line 64, the preparations of the buffer object are completed. The next step is to read the OBJ file contents into this buffer, which takes place at line 76 using readOBJFile(). The first argument is the location of the file (URL), the second one is gl, and the third one is the Object object (model) that packages the buffer objects. The tasks carried out by this function are similar to those when loading a texture image using the Image object and are shown here:

(2.1) Create an XMLHttpRequest object (line 117).

(2.2) Register the event handler to be called when the loading of the file is completed (line 119).

(2.3) Create a request to acquire the file using the open() method (line 124).

(2.4) Send the request to acquire the file (line 125).

Line 117 creates the XMLHttpRequest object, which sends an HTTP request to a web server. Line 119 is the registration of the event handler that will be called after the browser has loaded the file. Line 124 creates the request to acquire the file using the open() method. Because you are requesting a file, the first argument is GET, and the second one is the URL for the file. The last one specifies whether or not the request is asynchronous. Finally, line 125 uses the send() method to send the request to the web server to get the file.[9]

Once the browser has loaded the file, the event handler at line 119 is called. Line 120 checks for any errors returned by the load request. If the readyState property is 4, it indicates that the loading process is completed. However, if the readyState is not 4 and the status property is 404, it indicates that the specified file does not exist. The 404 error is the same as "404 Not Found," which is displayed when you try to display a web page that does not exist. When the file has been loaded successfully, onReadOBJFile() is called, which is defined at line 132 and takes five arguments. The first argument, responseText, contains the contents of the loaded file as one string. An OBJDoc object is created at line 133, which will be used, via the parse() method, to extract the results in a form that WebGL can easily use. The details will be explained next. Line 140 assigns the objDoc, which contains the parsing result in g_objDoc for rendering the model later.

[9] Note: When you want to run the sample programs that use external files in Chrome from your local disk, you should add the option --allow-file-access-from-files to Chrome. This is for security reasons. Chrome, by default, does not allow access to local files such as ../resources/cube.obj. For Firefox, the equivalent parameter, set via account:config, is security.fileuri.strict_origin_policy, which should be set to false. Remember to set it back as you open a security loophole if local file access is enabled.

User-Defined Object

Before proceeding to the explanation of the remaining code of `OBJViewer.js`, you need to understand how to create your own (user-defined) objects in JavaScript. `OBJViewer.js` uses user-defined objects to parse an OBJ file. In JavaScript, you can create user-defined objects which, once created, are treated in the same way as built-in objects like `Array` and `Date`.

The following is the `StringParser` object used in `OBJViewer.js`. The key aspects are how to define a **constructor** to create a user-defined object and how to add methods to the object. The constructor is a special method that is called when creating an object with `new`. The following is the constructor for the `StringParser` object:

```
595  // Constructor
596  var StringParser = function(str) {
597    this.str;   // Store the string specified by the argument
598    this.index; // Position in the string to be processed
599    this.init(str);
600  }
```

You can define the constructor with the anonymous function (see Chapter 2). Its parameter is the one that will be specified when creating the object with `new`. Lines 597 and 598 are the declaration of properties that can be used for this new object type, similar to properties like the length property of Array. You can define the property by writing the keyword `this` followed by `.` and the property name. Line 599 then calls `init()`, an initialization method that has been defined for this user-defined object.

Let's take a look at `init()`. You can add a method to the object by writing the method name after the keyword `prototype`. The body of the method is also defined using an anonymous function:

```
601  // Initialize StringParser object
602  StringParser.prototype.init = function(str) {
603    this.str = str;
604    this.index = 0;
605  }
```

What is convenient here is that you can access the property that is defined in the constructor from the method. The `this.str` at line 603 refers to `this.str` defined at line 597 in the constructor. The `this.index` at line 604 refers to `this.index` at line 598 in the constructor. Let's try using this `StringParse` object:

```
var sp = new StringParser('Tomorrow is another day.');
alert(sp.str);        // "Tomorrow is another day." is displayed.
sp.str = 'Quo Vadis'; // The content of str is changed to "Quo Vadis".
alert(sp.str);        // "Quo Vadis" is displayed
sp.init('Cinderella, tonight?');
alert(sp.str);        // "Cinderella, tonight?" is displayed
```

Let's look at another method, `skipDelimiters()`, that skips the delimiters (tab, space, (,), or ") in a string:

```
608  StringParser.prototype.skipDelimiters = function() {
609    for(var i = this.index, len = this.str.length; i < len; i++) {
610      var c = this.str.charAt(i);
611      // Skip TAB, Space, (, ), and "
612      if (c == '\t'|| c == ' ' || c == '(' || c == ')' || c == '"') continue;
613      break;
614    }
615    this.index = i;
616  }
```

The `charAt()` method at line 610 is supported by the `String` object that manages a string and retrieves the character specified by the argument from the string.

Now let's look at the parser code in `OBJViewer.js`.

Sample Program (Parser Code in OBJViewer.js)

`OBJViewer.js` parses the content of an OBJ file line by line and converts it to the structure shown in Figure 10.28. Each box in Figure 10.28 is a user-defined object. Although the parser code in `OBJViewer.js` looks quite complex, the core parsing process is simple. The complexity comes because it is repeated several times. Let's take a look at the core processing, which once you understand will allow you to understand the whole process.

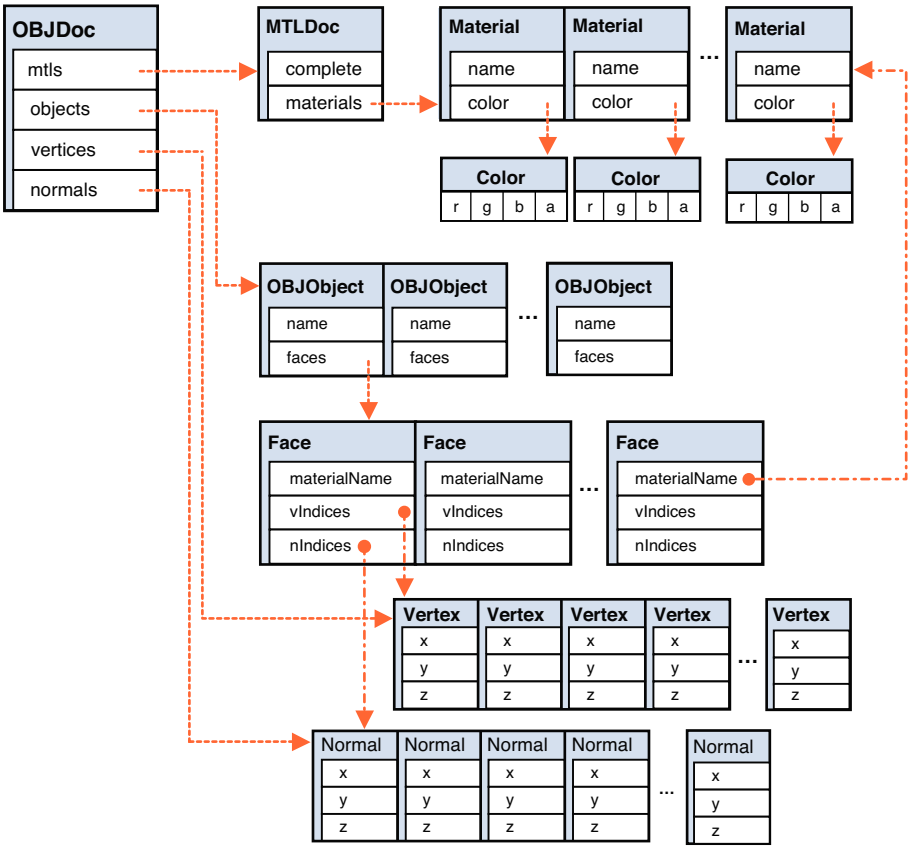

Figure 10.28 The internal structure after parsing an OBJ file

Listing 10.19 shows the basic code of `OBJViewer.js`.

Listing 10.19 OBJViewer.js (Parser Part)

```
214 // OBJDoc object
215 // Constructor
216 var OBJDoc = function(fileName) {
217   this.fileName = fileName;
218   this.mtls = new Array(0);      // Initialize the property for MTL
219   this.objects = new Array(0);   // Initialize the property for Object
220   this.vertices = new Array(0);  // Initialize the property for Vertex
221   this.normals = new Array(0);   // Initialize the property for Normal
222 }
223
224 // Parsing the OBJ file
225 OBJDoc.prototype.parse = function(fileString, scale, reverseNormal) {
```

```
226   var lines = fileString.split('\n');  // Break up into lines
227   lines.push(null); // Append null
228   var index = 0;    // Initialize index of line
229
230   var currentObject = null;
231   var currentMaterialName = "";
232
233   // Parse line by line
234   var line;         // A string in the line to be parsed
235   var sp = new StringParser();  // Create StringParser
236   while ((line = lines[index++]) != null) {
237     sp.init(line);                  // init StringParser
238     var command = sp.getWord();     // Get command
239     if(command == null)     continue;  // check null command
240
241     switch(command){
242     case '#':
243       continue;  // Skip comments
244     case 'mtllib':     // Read Material chunk
245       var path = this.parseMtllib(sp, this.fileName);
246       var mtl = new MTLDoc();   // Create MTL instance
247       this.mtls.push(mtl);
248       var request = new XMLHttpRequest();
249       request.onreadystatechange = function() {
250         if (request.readyState == 4) {
251           if (request.status != 404) {
252             onReadMTLFile(request.responseText, mtl);
253           }else{
254             mtl.complete = true;
255           }
256         }
257       }
258       request.open('GET', path, true); // Create a request to get file
259       request.send();                  // Send the request
260       continue; // Go to the next line
261     case 'o':
262     case 'g':   // Read Object name
263       var object = this.parseObjectName(sp);
264       this.objects.push(object);
265       currentObject = object;
266       continue; // Go to the next line
267     case 'v':   // Read vertex
268       var vertex = this.parseVertex(sp, scale);
269       this.vertices.push(vertex);
270       continue; // Go to the next line
```

```
271     case 'vn':   // Read normal
272       var normal = this.parseNormal(sp);
273       this.normals.push(normal);
274       continue; // Go to the next line
275     case 'usemtl': // Read Material name
276       currentMaterialName = this.parseUsemtl(sp);
277       continue; // Go to the next line
278     case 'f': // Read face
279       var face = this.parseFace(sp, currentMaterialName, this.vertices,
                                                                    ➥reverse);
280       currentObject.addFace(face);
281       continue; // Go to the next line
282     }
283   }
284
285   return true;
286 }
```

Lines 216 to 222 define the constructor for the `OBJDoc` object, which consists of five properties that will be parsed and set up. The actual parsing is done in the `parse()` method at line 225. The content of the OBJ file is passed as one string to the argument *fileString* of the `parse()` method and then split into manageable pieces using the `split()` method. This method splits a string into pieces delimited by the characters specified as the argument. As you can see at line 226, the argument specifies "\n" (new line), so each line is stored in `this.lines` as an array. `null` is appended at the end of the array at line 227 to make it easy to find the end of the array. `this.index` indicates how many lines have been parsed and is initialized to 0 at line 228.

You have already seen the `StringParser` object, which is created at line 235, in the previous section. This object is used for parsing the content of the line.

Now you are ready to start parsing the OBJ file. Each line is stored in `line` using `this.lines[this.index++]` at line 236. Line 237 writes the line to `sp` (`StringParser`). Line 238 gets the first word of the line using `sp.getWord()` and stores it in `command`. You use the methods shown in Table 10.3, where "word" in the table indicates a string surrounded by a delimiter (tab, space, (,), or ").

Table 10.3 Method that StringParser Supports

Method	Description
`StringParser.init(str)`	Initialize `StringParser` to be able to parse *str.*
`StringParser.getWord()`	Get a word.
`StringParser.skipToNext-Word()`	Skip to the beginning of the next word.

Method	Description
`StringParser.getInt()`	Get a word and convert it to an integer number.
`StringParser.getFloat()`	Get a word and convert it to a floating point number.

The `switch` statement at line 241 checks the command to determine how to process the following lines in the OBJ file.

If the command is `#` (line 242), the line is a comment. Line 243 skips it using `continue`.

If the command is `mtllib` (line 241), the line is a reference to an MTL file. Line 245 generates the path to the file. Line 246 creates an MTLDoc object for storing the material information in the MTL file, and line 247 stores it in `this.mtls`. Then lines 248 to 259 read the file in the same way that you read an OBJ file. The MTL file is parsed by `onReadMTLfile()`, which is called when it is loaded.

If the command is `o` (line 261) or `g` (line 262), it indicates a named object or group. Line 263 parses the line and returns the results in `OBJObject`. This object is stored in `this.objects` at line 264 and `currentObject`.

If the command is `v`, the line is a vertex position. Line 268 parses (x, y, z) and returns the result in `Vertex` object. This object is stored in `this.vertices` at line 269.

If the command is `f`, it indicates that the line is a face definition. Line 279 parses it and returns the result in the `Face` object. This object is stored in the `currentObject`. Let's take a look at `parseVertex()`, which is shown in Listing 10.20.

Listing 10.20 OBJViewer.js (parseVertex())

```
302 OBJDoc.prototype.parseVertex = function(sp, scale) {
303   var x = sp.getFloat() * scale;
304   var y = sp.getFloat() * scale;
305   var z = sp.getFloat() * scale;
306   return (new Vertex(x, y, z));
307 }
```

Line 303 retrieves the x value from the line using `sp.getFloat()`. A scaling factor is applied when the model is too small or large. After retrieving the three coordinates, line 306 creates a `Vertex` object using x, y, and z and returns it.

Once the OBJ file and MTL files have been fully parsed, the arrays for the vertex coordinates, colors, normals, and indices are created from the structure shown in Figure 10.28. Then `onReadComplete()` is called to write them into the buffer object (see Listing 10.21).

Listing 10.21 OBJViewer.js (onReadComplete())

```
176 // OBJ File has been read completely
177 function onReadComplete(gl, model, objDoc) {
178   // Acquire the vertex coordinates and colors from OBJ file
179   var drawingInfo = objDoc.getDrawingInfo();
180
181   // Write date into the buffer object
182   gl.bindBuffer(gl.ARRAY_BUFFER, model.vertexBuffer);
183   gl.bufferData(gl.ARRAY_BUFFER, drawingInfo.vertices,gl.STATIC_DRAW);
184
185   gl.bindBuffer(gl.ARRAY_BUFFER, model.normalBuffer);
186   gl.bufferData(gl.ARRAY_BUFFER, drawingInfo.normals, gl.STATIC_DRAW);
187
188   gl.bindBuffer(gl.ARRAY_BUFFER, model.colorBuffer);
189   gl.bufferData(gl.ARRAY_BUFFER, drawingInfo.colors, gl.STATIC_DRAW);
190
191   // Write the indices to the buffer object
192   gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, model.indexBuffer);
193   gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, drawingInfo.indices, gl.STATIC_DRAW);
194
195   return drawingInfo;
196 }
```

This method is straightforward and starts at Line 178, which retrieves the drawing information from `objDoc` that contains the results from parsing the OBJ file. Lines 183, 186, 189, and 193 write vertices, normals, colors, and indices into the respective buffer objects.

The function `getDrawingInfo()` at line 451 retrieves the vertices, normals, colors, and indices from the `objDoc` and is shown in Listing 10.22.

Listing 10.22 OBJViewer.js (Retrieving the Drawing Information)

```
450 // Retrieve the information for drawing 3D model
451 OBJDoc.prototype.getDrawingInfo = function() {
452   // Create an array for vertex coordinates, normals, colors, and indices
453   var numIndices = 0;
454   for (var i = 0; i < this.objects.length; i++){
455     numIndices += this.objects[i].numIndices;
456   }
457   var numVertices = numIndices;
458   var vertices = new Float32Array(numVertices * 3);
459   var normals = new Float32Array(numVertices * 3);
460   var colors = new Float32Array(numVertices * 4);
461   var indices = new Uint16Array(numIndices);
462
463   // Set vertex, normal, and color
```

```
464   var index_indices = 0;
465   for (var i = 0; i < this.objects.length; i++){
466     var object = this.objects[i];
467     for (var j = 0; j < object.faces.length; j++){
468        var face = object.face[j];
469       var color = this.findColor(face.materialName);
470       var faceNormal = face.normal;
471       for (var k = 0; k < face.vIndices.length; k++){
472         // Set index
473         indices[index_indices] = index_indices;
474         // Copy vertex
475         var vIdx = face.vIndices[k];
476         var vertex = this.vertices[vIdx];
477         vertices[index_indices * 3 + 0] = vertex.x;
478         vertices[index_indices * 3 + 1] = vertex.y;
479         vertices[index_indices * 3 + 2] = vertex.z;
480         // Copy color
481         colors[index_indices * 4 + 0] = color.r;
482         colors[index_indices * 4 + 1] = color.g;
483         colors[index_indices * 4 + 2] = color.b;
484         colors[index_indices * 4 + 3] = color.a;
485         // Copy normal
486         var nIdx = face.nIndices[k];
487         if(nIdx >= 0){
488           var normal =  this.normals[nIdx];
489           normals[index_indices * 3 + 0] = normal.x;
490           normals[index_indices * 3 + 1] = normal.y;
491           normals[index_indices * 3 + 2] = normal.z;
492         }else{
493           normals[index_indices * 3 + 0] = faceNormal.x;
494           normals[index_indices * 3 + 1] = faceNormal.y;
495           normals[index_indices * 3 + 2] = faceNormal.z;
496         }
497         index_indices ++;
498       }
499     }
500   }
501
502   return new DrawingInfo(vertices, normals, colors, indices);
503 };
```

Line 454 calculates the number of indices using a `for` loop. Then lines 458 to 461 create typed arrays for storing vertices, normals, colors, and indices that are assigned to the appropriate buffer objects. The size of each array is determined by the number of indices at line 454.

The program traverses the `OBJObject` objects and its `Face` objects in the order shown in Figure 10.28 and stores the information in the arrays `vertices`, `colors`, and `indices`.

The `for` statement at line 465 loops, extracting each `OBJObject` one by one from the result of the earlier parsing. The `for` statement at line 467 does the same for each `Face` object that makes up the `OBJObject` and performs the following steps for each `Face`:

1. Lines 469 finds the color of the `Face` using `materialName` and stores the color in `color`. Line 468 stores the normal of the face in `faceNomal` for later use.
2. The `for` statement at line 471 loops, extracting vertex indices from the face, storing its vertex position in `vertices` (lines 477 to 479), and storing the r, g, and b components of the color in `colors` (lines 482 to 484). The code from line 486 handles normals. OBJ files may or may not contain normals, so line 487 checks for that. If normals are found in the OBJ file, lines 487 to 489 store them in `normals`. Lines 492 to 494 then store the normals this program generates.

Once you complete these steps for all `OBJObjects`, you are ready to draw. Line 502 returns the information for drawing the model in a `DrawingInfo` object, which manages the vertex information that has to be written in the buffer object, as described previously.

Although this has been, by necessity, a rapid explanation, at this stage you should understand how the contents of the OBJ file can be read in, parsed, and displayed with WebGL. If you want to read multiple model files in a single scene, you would repeat the preceding processes. There are several other models stored as OBJ files in the `resources` directory of the sample programs, which you can look at and experiment with to confirm your understanding (see Figure 10.29).

Figure 10.29 Various 3D models

Handling Lost Context

WebGL uses the underlying graphics hardware, which is a shared resource managed by the operating system. There are several situations where this resource can be "taken away," resulting in information stored within the graphics hardware being lost. These include