be retrieved as a string using the standard DOM API. I won't explain the API functions that are used, but here is a function that takes the *id* of the script element as its parameter and returns a string containing the text from inside the element:

```
function getTextContent( elementID ) {
    var element = document.getElementById(elementID);
    var node = element.firstChild;
    var str = "";
    while (node) {
        if (node.nodeType == 3) // this is a text node
            str += node.textContent;
        node = node.nextSibling;
    }
    return str;
}
```

For the above example, I would call *getTextContent*("vshader") and use the return value as the source code for the vertex shader. The sample programs for this chapter use this technique. For fragment shaders, I will use the *type* "x-shader/x-fragment", but in fact the value of *type* doesn't matter as long as it is something that web browsers will not recognize as a legitimate scripting language. (By the way, JavaScript `<script>` elements can use "text/javascript" as the *type*, but that is the default value and is usually omitted.)

7.1.2 Introducing glMatrix

Transformations are essential to computer graphics. The WebGL API does not provide any functions for working with transformations. In Section 6.5, we used a simple JavaScript library to represent modeling transformations in 2D. Things get more complex in three dimensions. For 3D graphics with WebGL, the JavaScript side will usually have to create both a modelview transform and a projection transform, and it will have to apply rotation, scaling, and translation to the modelview matrix, all without help from WebGL. Doing so is much easier if you have a JavaScript library to do the work. One commonly used library is ***glMatrix***, a free JavaScript library for vector and matrix math written by Brandon Jones and Colin MacKenzie IV. It is available from http://glmatrix.net. The download from that web site includes a JavaScript file webgl/gl-matrix-min.js that defines the library. According to its license, this file can be freely used and distributed. You can find a copy in the *source* folder in the web site download of this book. My examples use version 2.3 of the library. Note that *gl-matrix-min.js* is a "minified" JavaScript file, which is not meant to be human-readable. To learn about the library, see the documentation on the web site (currently at http://glmatrix.net/docs/ as I write this), or read the source files in the download. (You can also read the full source for version 2.2, in human-readable form including comments, in the file webgl/gl-matrix.js, which can be found in the directory *source/webgl* in the web-site download of this book.)

The *glMatrix* API can be made can be made available for use on a web page with a script element such as

```
<script src="gl-matrix-min.js"></script>
```

This assumes that *gl-matrix-min.js* is in the same directory as the web page.

The *glMatrix* library defines what it calls "classes" named *vec2*, *vec3*, and *vec4* for working with vectors of 2, 3, and 4 numbers. It defines *mat3* for working with 3-by-3 matrices and *mat4* for 4-by-4 matrices. The names should not be confused with the GLSL types of the same names; *glMatrix* in entirely on the JavaScript side. However, a *glMatrix* *mat4* can be passed to a shader program to specify the value of a GLSL *mat4*, and similarly for the other vector and matrix types.

Each *glMatrix* class defines a set of functions for working with vectors and matrices. In fact, however, although the documentation uses the term "class," *glMatrix* is not object-oriented. Its classes are really just JavaScript objects, and the functions in its classes are what would be called static methods in Java. Vectors and matrices are represented in *glMatrix* as arrays, and the functions in classes like *vec4* and *mat4* simply operate on those arrays. There are no objects of type *vec4* or *mat4* as such, just arrays of length 4 or 16 respectively. The arrays can be either ordinary JavaScript arrays or typed arrays of type *Float32Array*. If you let *glMatrix* create the arrays for you, they will be *Float32Arrays*, but all *glMatrix* functions will work with either kind of array. For example, if the *glMatrix* documentation says that a parameter should be of type *vec3*, it is OK to pass either a *Float32Array* or a regular JavaScript array of three numbers as the value of that parameter.

Note that it is also the case that either kind of array can be used in WebGL functions such as *glUniform3fv*() and *glUniformMatrix4fv*(). *glMatrix* is designed to work with those functions. For example, a *mat4* in *glMatrix* is an array of length 16 that holds the elements of a 4-by-4 array in column-major order, the same format that is used by *glUniformMatrix4fv*.

* * *

Each *glMatrix* class has a *create*() function which creates an array of the appropriate length and fills it with default values. For example,

```
transform = mat4.create();
```

sets *transform* to be a new *Float32Array* of length 16, initialized to represent the identity matrix. Similarly,

```
vector = vec3.create();
```

creates a *Float32Array* of length 3, filled with zeros. Each class also has a function *clone*(x) that creates a copy of its parameter x. For example:

```
saveTransform = mat4.clone(modelview);
```

Most other functions do **not** create new arrays. Instead, they modify the contents of their first parameter. For example, *mat4.multiply*(A,B,C) will modify A so that it holds the matrix product of B and C. Each parameter must be a *mat4* (that is, an array of length 16) that already exists. It is OK for some of the arrays to be the same. For example, *mat4.multiply*(A,A,B) has the effect of multiplying A times B and modifying A so that it contains the answer.

There are functions for multiplying a matrix by standard transformations such as scaling and rotation. For example if A and B are *mat4s* and v is a *vec3*, then *mat4.translate*(A,B,v) makes A equal to the product of B and the matrix that represents translation by the vector v. In practice, we will use such operations mostly on a matrix that represents the modelview transformation. So, suppose that we have a *mat4* named *modelview* that holds the current modelview transform. To apply a translation by a vector `[dx,dy,dz]`, we can say

```
mat4.translate( modelview, modelview, [dx,dy,dz] );
```

This is equivalent to calling *glTranslatef*(*dx,dy,dz*) in OpenGL. That is, if we draw some geometry after this statement, using *modelview* as the modelview transformation, then the geometry will first be translated by `[dx,dy,dz]` and then will be transformed by whatever was the previous value of *modelview*. Note the use of a vector to specify the translation in this command, rather than three separate parameters; this is typical of *glMatrix*. To apply a scaling transformation with scale factors *sx*, *sy*, and *sz*, use

```
mat4.scale( modelview, modelview, [sx,sy,sz] );
```

For rotation, *glMatrix* has four functions, including three for the common cases of rotation about the x, y, or z axis. The fourth rotation function specifies the axis of rotation as the line from (0,0,0) to a point (*dx,dy,dz*). This is equivalent to *glRotatef(angle,dx,dy,dz)* Unfortunately, the angle of rotation in these functions is specified in radians rather than in degrees:

```
mat4.rotateX( modelview, modelview, radians );
mat4.rotateY( modelview, modelview, radians );
mat4.rotateZ( modelview, modelview, radians );
mat4.rotate( modelview, modelview, radians, [dx,dy,dz] );
```

These function allow us to do all the basic modeling and viewing transformations that we need for 3D graphics. To do hierarchical graphics, we also need to save and restore the transformation as we traverse the scene graph. For that, we need a stack. We can use a regular JavaScript array, which already has *push* and *pop* operations. So, we can create the stack as an empty array:

```
var matrixStack = [];
```

We can then push a copy of the current modelview matrix onto the stack by saying

```
matrixStack.push( mat4.clone(modelview) );
```

and we can remove a matrix from the stack and set it to be the current modelview matrix with

```
modelview = matrixStack.pop();
```

These operations are equivalent to *glPushMatrix*() and *glPopMatrix*() in OpenGL.

* * *

The starting point for the modelview transform is usually a viewing transform. In OpenGL, the function *gluLookAt* is often used to set up the viewing transformation (Subsection 3.3.4). The *glMatrix* library has a "lookAt" function to do the same thing:

```
mat4.lookAt( modelview, [eyex,eyey,eyez], [refx,refy,refz], [upx,upy,upz] );
```

Note that this function uses three *vec3's* in place of the nine separate parameters in *gluLookAt*, and it places the result in its first parameter instead of in a global variable. This function call is actually equivalent to the two OpenGL commands

```
glLoadIdentity();
gluLookAt( eyex,eyey,eyez,refx,refy,refz,upx,upy,upz );
```

So, you don't have to set *modelview* equal to the identity matrix before calling *mat4.lookAt*, as you would usually do in OpenGL. However, you do have to create the *modelview* matrix at some point before using *mat4.lookAt*, such as by calling

```
var modelview = mat4.create();
```

If you do want to set an existing *mat4* to the identity matrix, you can do so with the *mat4.identity* function. For example,

```
mat4.identity( modelview );
```

You could use this as a starting point if you wanted to compose the view transformation out of basic scale, rotate, and translate transformations.

Similarly, *glMatrix* has functions for setting up projection transformations. It has functions equivalent to *glOrtho*, *glFrustum*, and *gluPerspective* (Subsection 3.3.3), except that the field-of-view angle in *mat4.perspective* is given in radians rather than degrees:

```
mat4.ortho( projection, left, right, bottom, top, near, far );

mat4.frustum( projection, left, right, bottom, top, near, far );

mat4.perspective( projection, fovyInRadians, aspect, near, far );
```

As with the modelview transformation, you do not need to load *projection* with the identity before calling one of these functions, but you must create *projection* as a *mat4* (or an array of length 16).

7.1.3 Transforming Coordinates

Of course, the point of making a projection and a modelview transformation is to use them to transform coordinates while drawing primitives. In WebGL, the transformation is usually done in the vertex shader. The coordinates for a primitive are specified in object coordinates. They are multiplied by the modelview transformation to covert them into eye coordinates and then by the projection matrix to covert them to the final clip coordinates that are actually used for drawing the primitive. Alternatively, the modelview and projection matrices can be multiplied together to get a matrix that represents the combined transformation; object coordinates can then be multiplied by that matrix to transform them directly into clip coordinates.

In the shader program, coordinate transforms are ususally represented as GLSL uniform variables of type *mat4*. The shader program can use either separate projection and modelview matrices or a combined matrix (or both). Sometimes, a separate modelview transform matrix is required, because certain lighting calculations are done in eye coordinates, but here is a minimal vertex shader that uses a combined matrix:

```
attribute vec3 a_coords;           // (x,y,z) object coordinates of vertex.
uniform mat4 modelviewProjection;  // Combined transformation matrix.
void main() {
    vec4 coords = vec4(a_coords,1.0);   // Add 1.0 for the w-coordinate.
    gl_Position = modelviewProjection * coords;  // Transform the coordinates.
}
```

This shader is from the sample program webgl/glmatrix-cube-unlit.html. That program lets the user view a colored cube, using just basic color with no lighting applied. The user can select either an orthographic or a perspective projection and can rotate the cube using the keyboard. The rotation is applied as a modeling transformation consisting of separate rotations about the x-, y-, and z-axes. For transformation matrices on the JavaScript side, the program uses the *mat4* class from the *glMatrix* library to represent the projection, modelview, and combined transformation matrices:

```
var projection = mat4.create();  // projection matrix
var modelview = mat4.create();   // modelview matrix
var modelviewProjection = mat4.create();  // combined matrix
```

Only *modelviewProjection* corresponds to a shader variable. The location of that variable in the shader program is obtained during initialization using

```
u_modelviewProjection = gl.getUniformLocation(prog, "modelviewProjection");
```

The transformation matrices are computed in the *draw*() function, using functions from the *glMatrix* *mat4* class. The value for *modelviewProjection* is sent to the shader program using *gl.uniformMatrix4fv* before the primitives that make up the cube are drawn. Here is the code that does it:

```
/* Set the value of projection to represent the projection transformation */

if (document.getElementById("persproj").checked) {
     mat4.perspective(projection, Math.PI/5, 1, 4, 8);
}
else {
     mat4.ortho(projection, -2, 2, -2, 2, 4, 8);
}

/* Set the value of modelview to represent the viewing transform. */

mat4.lookAt(modelview, [2,2,6], [0,0,0], [0,1,0]);

/* Apply the modeling tranformation to modelview. */

mat4.rotateX(modelview, modelview, rotateX);
mat4.rotateY(modelview, modelview, rotateY);
mat4.rotateZ(modelview, modelview, rotateZ);

/* Multiply the projection matrix times the modelview matrix to give the
   combined transformation matrix, and send that to the shader program. */

mat4.multiply( modelviewProjection, projection, modelview );
gl.uniformMatrix4fv(u_modelviewProjection, false, modelviewProjection );
```

If separate modelview and projection matrices are used in the shader program, then the modelview matrix can be applied to transform object coordinates to eye coordinates, and the projection can then be applied to the eye coordinates to compute *gl_Position*. Here is a minimal vertex shader that does that:

```
attribute vec3 a_coords;  // (x,y,z) object coordinates of vertex.
uniform mat4 modelview;   // Modelview transformation.
uniform mat4 projection;  // Projection transformation
void main() {
    vec4 coords = vec4(a_coords,1.0);      // Add 1.0 for w-coordinate.
    vec4 eyeCoords = modelview * coords;   // Apply modelview transform.
    gl_Position = projection * eyeCoords;  // Apply projection transform.
}
```

7.1.4 Transforming Normals

Normal vectors are essential for lighting calculations (Subsection 4.1.3). When a surface is transformed in some way, it seems that the normal vectors to that surface will also change. However, that is not true if the transformation is a translation. A normal vector tells what direction a surface is facing. Translating the surface does not change the direction in which the surface is facing, so the nomal vector remains the same. Remember that a vector doesn't have a position, just a length and a direction. So it doesn't even make sense to talk about moving or translating a vector.