Chapter 3

Drawing and Transforming Triangles

Chapter 2, "Your First Step with WebGL," explained the basic approach to drawing WebGL graphics. You saw how to retrieve the WebGL context and clear a `<canvas>` in preparation for drawing your 2D/3DCG. You then explored the roles and features of the vertex and fragment shaders and how to actually draw graphics with them. With this basic structure in mind, you then constructed several sample programs that drew simple shapes composed of points on the screen.

This chapter builds on those basics by exploring how to draw more complex shapes and how to manipulate those shapes in 3D space. In particular, this chapter looks at

- The critical role of triangles in 3DCG and WebGL's support for drawing triangles
- Using multiple triangles to draw other basic shapes
- Basic transformations that move, rotate, and scale triangles using simple equations
- How matrix operations make transformations simple

By the end of this chapter, you will have a comprehensive understanding of WebGL's support for drawing basic shapes and how to use matrix operations to manipulate those shapes. Chapter 4, "More Transformations and Basic Animation," then builds on this knowledge to explore simple animations.

Drawing Multiple Points

As you are probably aware, 3D models are actually made from a simple building block: the humble triangle. For example, looking at the frog in Figure 3.1, the figure on the right side shows the triangles used to make up the shape, and in particular the three vertices that make up one triangle of the head. So, although this game character has a complex shape, its basic components are the same as a simple one, except of course for many more triangles and their associated vertices. By using smaller and smaller triangles, and therefore more and more vertices, you can create more complex or smoother objects. Typically, a complex shape or game character will consist of tens of thousands of triangles and their associated vertices. Thus, multiple vertices used to make up triangles are pivotal for drawing 3D objects.

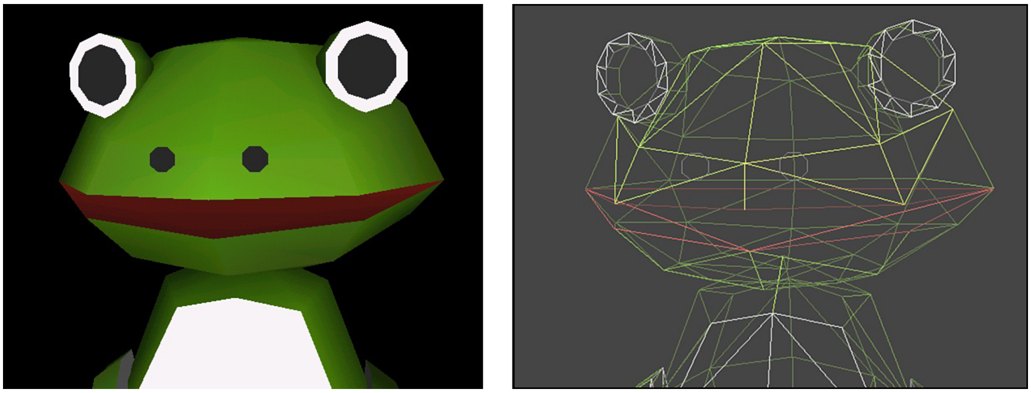


Figure 3.1 Complex characters are also constructed from multiple triangles

In this section, you explore the process of drawing shapes using multiple vertices. However, to keep things simple, you'll continue to use 2D shapes, because the technique to deal with multiple vertices for a 2D shape is the same as dealing with them for a 3D object. Essentially, if you can master these techniques for 2D shapes, you can easily understand the examples in the rest of this book that use the same techniques for 3D objects.

As an example of handling multiple vertices, let's create a program, `MultiPoint`, that draws three red points on the screen; remember, three points or vertices make up the triangle. Figure 3.2 shows a screenshot from `Multipoint`.

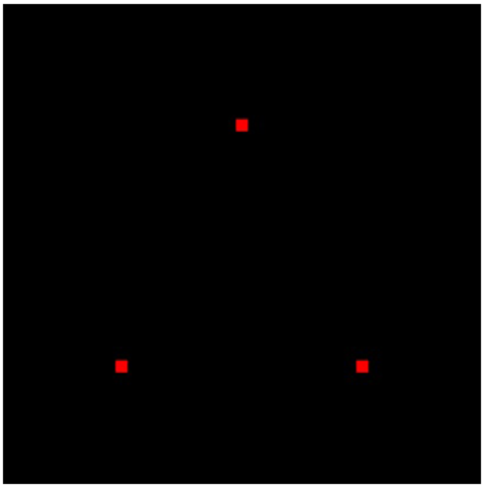


Figure 3.2 `MultiPoint`

In the previous chapter, you created a sample program, `ClickedPoints`, that drew multiple points based on mouse clicks. `ClickedPoints` stored the position of the points in a JavaScript array (`g_points[]`) and used the `gl.drawArrays()` method to draw each point (Listing 3.1). To draw multiple points, you used a loop that iterated through the array, drawing each point in turn by passing one vertex at a time to the shader.

Listing 3.1 Drawing Multiple Points as Shown in ClickedPoints.js (Chapter 2)

```
for(var i = 0; i<len; i+=2) {
  // Pass the position of a point to a_Position variable
  gl.vertexAttrib3f(a_Position, g_points[i], g_points[i+1], 0.0);

  // Draw a point
  gl.drawArrays(gl.POINTS, 0, 1);
}
```

Obviously, this method is useful only for single points. For shapes that use multiple vertices, you need a way to simultaneously pass multiple vertices to the vertex shader so that you can draw shapes constructed from multiple vertices, such as triangles, rectangles, and cubes.

WebGL provides a convenient way to pass multiple vertices and uses something called a **buffer object** to do so. A buffer object is a memory area that can store multiple vertices in the WebGL system. It is used both as a staging area for the vertex data and a way to simultaneously pass the vertices to a vertex shader.

Let's examine a sample program before explaining the buffer object so you can get a feel for the processing flow.

Sample Program (MultiPoint.js)

The processing flowchart for `MultiPoint.js` (see Figure 3.3) is basically the same as for `ClickedPoints.js` (Listing 2.7) and `ColoredPoints.js` (Listing 2.8), which you saw in Chapter 2. The only difference is a new step, setting up the positions of vertices, which is added to the previous flow.

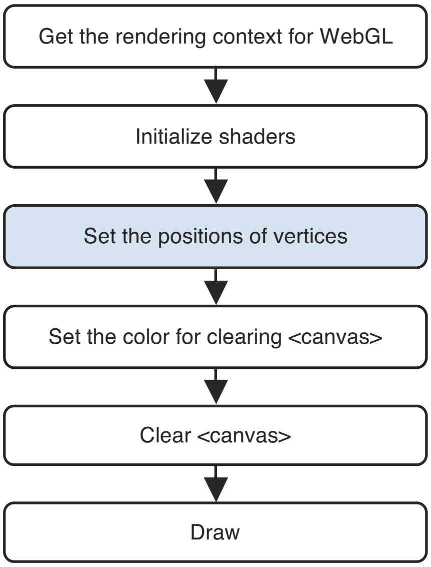

Get the rendering context for WebGL
Initialize shaders
Set the positions of vertices
Set the color for clearing <canvas>
Clear <canvas>
Draw

Figure 3.3 Processing flowchart for MultiPoints.js

This step is implemented at line 34, the function `initVertexBuffers()`, in Listing 3.2.

Listing 3.2 MultiPoint.js

```
1 // MultiPoint.js
2 // Vertex shader program
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'void main() {\n' +
6   '  gl_Position = a_Position;\n' +
7   '  gl_PointSize = 10.0;\n' +
8   '}\n';
9
10 // Fragment shader program
    ...
15
16 function main() {
    ...
```

```
  // Get the rendering context for WebGL
  var gl = getWebGLContext(canvas);
     ...
  // Initialize shaders
  if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
     ...
  }

  // Set the positions of vertices
  var n = initVertexBuffers(gl);
  if (n < 0) {
    console.log('Failed to set the positions of the vertices');
    return;
  }

  // Set the color for clearing <canvas>
     ...
  // Clear <canvas>
...
  // Draw three points
  gl.drawArrays(gl.POINTS, 0, n); // n is 3
}

function initVertexBuffers(gl) {
  var vertices = new Float32Array([
    0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
  ]);
  var n = 3; // The number of vertices

  // Create a buffer object
  var vertexBuffer = gl.createBuffer();
  if (!vertexBuffer) {
    console.log('Failed to create the buffer object ');
    return -1;
  }

  // Bind the buffer object to target
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  // Write date into the buffer object
  gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

  var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
 ...
  // Assign the buffer object to a_Position variable
  gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
```

```
75
76   // Enable the assignment to a_Position variable
77   gl.enableVertexAttribArray(a_Position);
78
79   return n;
80 }
```

The new function `initVertexBuffers()` is defined at line 50 and used at line 34 to set up the vertex buffer object. The function stores multiple vertices in the buffer object and then completes the preparations for passing it to a vertex shader:

```
33 // Set the positions of vertices
34  var n = initVertexBuffers(gl);
```

The return value of this function is the number of vertices being drawn, stored in the variable `n`. Note that in case of error, `n` is negative.

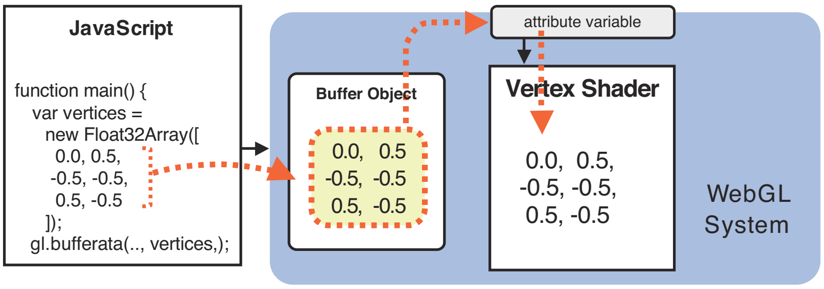
As in the previous examples, the drawing operation is carried out using a single call to `gl.drawArrays()` at Line 48. This is similar to `ClickedPoints.js` except that `n` is passed as the third argument of `gl.drawArrays()` rather than the value 1:

```
46   // Draw three points
47   gl.drawArrays(gl.POINTS, 0, n); // n is 3
```

Because you are using a buffer object to pass multiple vertices to a vertex shader in `initVertexBuffers()`, you need to specify the number of vertices in the object as the third parameter of `gl.drawArrays()` so that WebGL then knows to draw a shape using all the vertices in the buffer object.

Using Buffer Objects

As indicated earlier, a buffer object is a mechanism provided by the WebGL system that provides a memory area allocated in the system (see Figure 3.4) that holds the vertices you want to draw. By creating a buffer object and then writing the vertices to the object, you can pass multiple vertices to a vertex shader through one of its attribute variables.



```
function main() {
  var vertices =
    new Float32Array([
      0.0, 0.5,
     -0.5, -0.5,
      0.5, -0.5
    ]);
  gl.bufferata(.., vertices,);
```

Figure 3.4 Passing multiple vertices to a vertex shader by using a buffer object

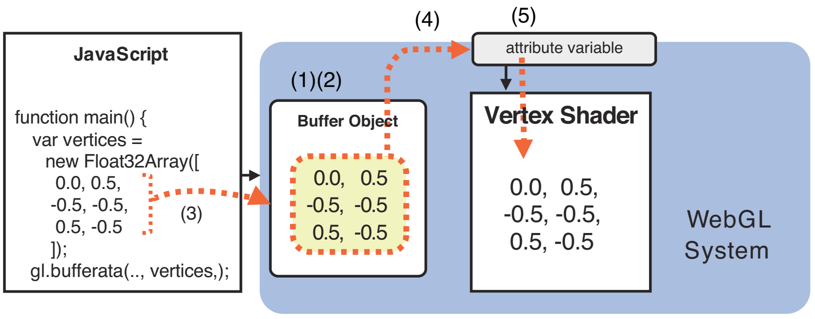
In the sample program, the data (vertex coordinates) written into a buffer object is defined as a special JavaScript array (`Float32Array`) as follows. We will explain this special array in detail later, but for now you can think of it as a normal array:

```
var vertices = new Float32Array([
  0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
]);
```

There are five steps needed to pass multiple data values to a vertex shader through a buffer object. Because WebGL uses a similar approach when dealing with other objects such as texture objects (Chapter 4) and framebuffer objects (Chapter 8, “Lighting Objects”), let’s explore these in detail so you will be able to apply the knowledge later:

1. Create a buffer object (`gl.createBuffer()`).
2. Bind the buffer object to a target (`gl.bindBuffer()`).
3. Write data into the buffer object (`gl.bufferData()`).
4. Assign the buffer object to an attribute variable (`gl.vertexAttribPointer()`).
5. Enable assignment (`gl.enableVertexAttribArray()`).

Figure 3.5 illustrates the five steps.



```
function main() {
  var vertices =
    new Float32Array([
     0.0, 0.5,
    -0.5, -0.5,
     0.5, -0.5
    ]);
  gl.bufferata(.., vertices,);
```

Figure 3.5 The five steps to pass multiple data values to a vertex shader using a buffer object

The code performing the steps in the sample program in Listing 3.2 is as follows:

```
// Create a buffer object                                   <- (1)
var vertexBuffer = gl.createBuffer();
if (!vertexBuffer) {
  console.log('Failed to create a buffer object');
  return -1;
}

// Bind the buffer object to a target                       <- (2)
```

```
64    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
65    // Write date into the buffer object                          <- (3)
66    gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
67
68    var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
     ...
73    // Assign the buffer object to a_Position variable            <- (4)
74    gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
75
76    // Enable the assignment to a_Position variable               <- (5)
77    gl.enableVertexAttribArray(a_Position);
```

Let's start with the first three steps (1–3), from creating a buffer object to writing data (vertex coordinates in this example) to the buffer, explaining the methods used within each step.

Create a Buffer Object (gl.createBuffer())

Before you can use a buffer object, you obviously need to create the buffer object. This is the first step, and it's carried out at line 57:

```
57    var vertexBuffer = gl.createBuffer();
```

You use the `gl.createBuffer()` method to create a buffer object within the WebGL system. Figure 3.6 shows the internal state of the WebGL system. The upper part of the figure shows the state before executing the method, and the lower part is after execution. As you can see, when the method is executed, it results in a single buffer object being created in the WebGL system. The keywords `gl.ARRAY_BUFFER` and `gl.ELEMENT_ARRAY_BUFFER` in the figure will be explained in the next section, so you can ignore them for now.

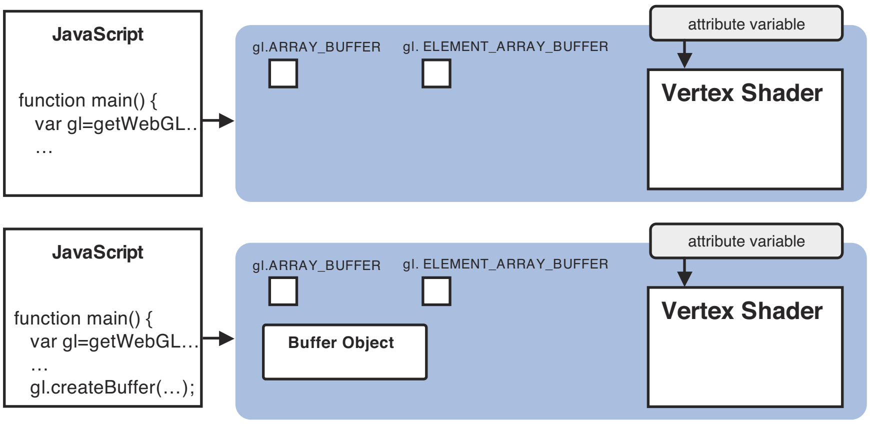


Figure 3.6 Create a buffer object

The following shows the specification of `gl.createBuffer()`.

`gl.createBuffer``()`

Create a buffer object.

Return value	non-null	The newly created buffer object.
	null	Failed to create a buffer object.
Errors	None	

The corresponding method `gl.deleteBuffer()` deletes the buffer object created by `gl.createBuffer()`.

`gl.deleteBuffer``(buffer)`

Delete the buffer object specified by *buffer*.

Parameters	buffer	Specifies the buffer object to be deleted.
Return Value	None	
Errors	None	

Bind a Buffer Object to a Target (gl.bindBuffer())

After creating a buffer object, the second step is to bind it to a "target." The target tells WebGL what type of data the buffer object contains, allowing it to deal with the contents correctly. This binding process is carried out at line 64 as follows:

```
64 gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
```

The specification of `gl.bindBuffer()` is as follows.

`gl.bindBuffer(target, buffer)`

Enable the buffer object specified by *buffer* and bind it to the *target*.

Parameters Target can be one of the following:

`gl.ARRAY_BUFFER`	Specifies that the buffer object contains vertex data.

	`gl.ELEMENT_ARRAY_BUFFER`	Specifies that the buffer object contains index values pointing to vertex data. (See Chapter 6, "The OpenGL ES Shading Language [GLSL ES].)
	buffer	Specifies the buffer object created by a previous call to `gl.createBuffer()`. When `null` is specified, binding to the *target* is disabled.
Return Value	None	
Errors	INVALID_ENUM	*target* is none of the above values. In this case, the current binding is maintained.

In the sample program in this section, `gl.ARRAY_BUFFER` is specified as the *target* to store vertex data (positions) in the buffer object. After executing line 64, the internal state in the WebGL system changes, as shown in Figure 3.7.

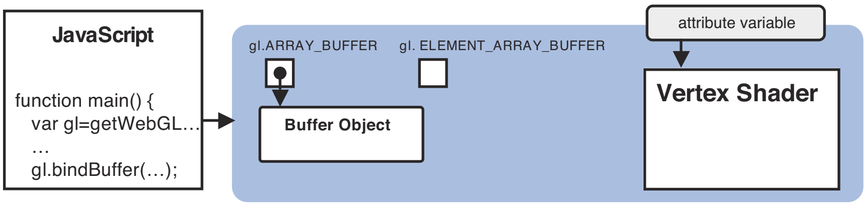


Figure 3.7 Bind a buffer object to a target

The next step is to write data into the buffer object. Note that because you won't be using the `gl.ELEMENT_ARRAY_BUFFER` until Chapter 6, it'll be removed from the following figures for clarity.

Write Data into a Buffer Object (gl.bufferData())

Step 3 allocates storage and writes data to the buffer. You use `gl.bufferData()` to do this, as shown at line 66:

```
66    gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
```

This method writes the data specified by the second parameter (`vertices`) into the buffer object bound to the first parameter (`gl.ARRAY_BUFFER`). After executing line 66, the internal state of the WebGL system changes, as shown in Figure 3.8.

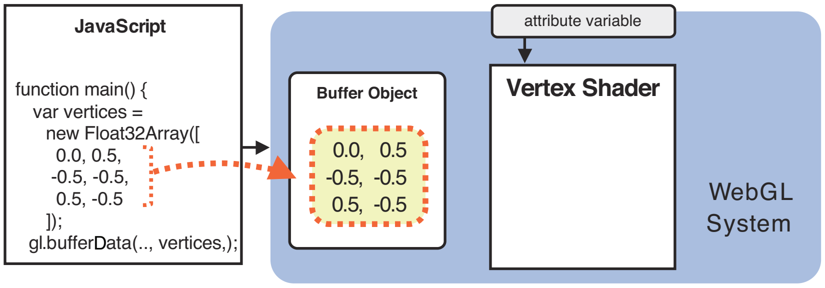


Figure 3.8 Allocate storage and write data into a buffer object

You can see in this figure that the vertex data defined in your JavaScript program is written to the buffer object bound to `gl.ARRAY_BUFFER`. The following table shows the specification of `gl.bufferData()`.

`gl.bufferData(target, data, usage)`

Allocate storage and write the data specified by *data* to the buffer object bound to *target*.

Parameters	target	Specifies `gl.ARRAY_BUFFER` or `gl.ELEMENT_ARRAY_BUFFER`.	
	data	Specifies the data to be written to the buffer object (typed array; see the next section).	
	usage	Specifies a hint about how the program is going to use the data stored in the buffer object. This hint helps WebGL optimize performance but will not stop your program from working if you get it wrong.	
		`gl.STATIC_DRAW`	The buffer object data will be specified once and used many times to draw shapes.
		`gl.STREAM_DRAW`	The buffer object data will be specified once and used a few times to draw shapes.
		`gl.DYNAMIC_DRAW`	The buffer object data will be specified repeatedly and used many times to draw shapes.
Return value	None		
Errors	INVALID_ENUM	*target* is none of the preceding constants	

Now, let us examine what data is passed to the buffer object using `gl.bufferData()`. This method uses the special array `vertices` mentioned earlier to pass data to the vertex shader. The array is created at line 51 using the `new` operator with the data arranged as <x coordinate and y coordinate of the first vertex>, <x coordinate and y coordinate of the second vertex>, and so on:

```
51   var vertices = new Float32Array([
52     0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
53   ]);
54   var n = 3; // The number of vertices
```

As you can see in the preceding code snippet, you are using the `Float32Array` object instead of the more usual JavaScript `Array` object to store the data. This is because the standard array in JavaScript is a general-purpose data structure able to hold both numeric data and strings but isn't optimized for large quantities of data of the same type, such as `vertices`. To address this issue, the typed array, of which one example is `Float32Array`, has been introduced.

Typed Arrays

WebGL often deals with large quantities of data of the same type, such as vertex coordinates and colors, for drawing 3D objects. For optimization purposes, a special type of array (**typed array**) has been introduced for each data type. Because the type of data in the array is known in advance, it can be handled efficiently.

`Float32Array` at line 51 is an example of a typed array and is generally used to store vertex coordinates or colors. It's important to remember that a typed array is expected by WebGL and is needed for many operations, such as the second parameter *data* of `gl.bufferData()`.

Table 3.1 shows the different typed arrays available. The third column shows the corresponding data type in C as a reference for those of you familiar with the C language.

Table 3.1 Typed Arrays Used in WebGL

Typed Array	Number of Bytes per Element	Description (C Types)
`Int8Array`	1	8-bit signed integer (signed char)
`Uint8Array`	1	8-bit unsigned integer (unsigned char)
`Int16Array`	2	16-bit signed integer (signed short)
`Uint16Array`	2	16-bit unsigned integer (unsigned short)
`Int32Array`	4	32-bit signed integer (signed int)
`Uint32Array`	4	32-bit unsigned integer (unsigned int)
`Float32Array`	4	32-bit floating point number (float)
`Float64Array`	8	64-bit floating point number (double)

Like JavaScript, these typed arrays have a set of methods, a property, and a constant available that are shown in Table 3.2. Note that, unlike the standard `Array` object in JavaScript, the methods `push()` and `pop()` are not supported.

Table 3.2 Methods, Property, Constant of Typed Arrays

Methods, Properties, and Constants	Description
`get(index)`	Get the *index*-th element
`set(index, value)`	Set *value* to the *index*-th element
`set(array, offset)`	Set the elements of *array* from *offset*-th element
`length`	The length of the array
`BYTES_PER_ELEMENT`	The number of bytes per element in the array

Just like standard arrays, the `new` operator creates a typed array and is passed the array data. For example, to create `Float32Array` vertices, you could pass the array `[0.0, 0.5, -0.5, -0.5, 0.5, -0.5]`, which represents a set of vertices. Note that the only way to create a typed array is by using the `new` operator. Unlike the `Array` object, the `[]` operator is not supported:

```
var vertices = new Float32Array([
  0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
]);
```

In addition, just like a normal JavaScript array, an empty typed array can be created by specifying the number of elements of the array as an argument. For example:

```
var vertices = new Float32Array(4);
```

With that, you've completed the first three steps of the process to set up and use a buffer (that is, creating a buffer object in the WebGL system, binding the buffer object to a target, and then writing data into the buffer object). Let's now look at how to actually use the buffer, which takes place in steps 4 and 5 of the process.

Assign the Buffer Object to an Attribute Variable (gl.vertexAttribPointer())

As explained in Chapter 2, you can use `gl.vertexAttrib[1234]f()` to assign data to an attribute variable. However, these methods can only be used to assign a single data value to an attribute variable. What you need here is a way to assign an array of values—the vertices in this case—to an attribute variable.

`gl.vertexAttribPointer()` solves this problem and can be used to assign a buffer object (actually a reference or handle to the buffer object) to an attribute variable. This can be seen at line 74 when you assign a buffer object to the attribute variable `a_Position`:

```
74   gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
```

The specification of `gl.vertexAttribPointer()` is as follows.

`gl.vertexAttribPointer(location, size, type, normalized, stride, offset)`

Assign the buffer object bound to `gl.ARRAY_BUFFER` to the attribute variable specified by *location*.

Parameters	location	Specifies the storage location of an attribute variable.
	size	Specifies the number of components per vertex in the buffer object (valid values are 1 to 4). If *size* is less than the number of components required by the attribute variable, the missing components are automatically supplied just like `gl.vertexAttrib[1234]f()`. For example, if *size* is 1, the second and third components will be set to 0, and the fourth component will be set to 1.
	type	Specifies the data format using one of the following:

`gl.UNSIGNED_BYTE`	unsigned byte	for `Uint8Array`
`gl.SHORT`	signed short integer	for `Int16Array`
`gl.UNSIGNED_SHORT`	unsigned short integer	for `Uint16Array`
`gl.INT`	signed integer	for `Int32Array`
`gl.UNSIGNED_INT`	unsigned integer	for `Uint32Array`
`gl.FLOAT`	floating point number	for `Float32Array`

	normalized	Either `true` or `false` to indicate whether nonfloating data should be normalized to [0, 1] or [–1, 1].
	stride	Specifies the number of bytes between different vertex data elements, or zero for default stride (see Chapter 4).
	offset	Specifies the offset (in bytes) in a buffer object to indicate what number-th byte the vertex data is stored from. If the data is stored from the beginning, *offset* is 0.
Return value	None	
Errors	INVALID_OPERATION	There is no current program object.
	INVALID_VALUE	*location* is greater than or equal to the maximum number of attribute variables (8, by default). *stride* or *offset* is a negative value.

So, after executing this fourth step, the preparations are nearly completed in the WebGL system for using the buffer object at the attribute variable specified by *location*. As you can see in Figure 3.9, although the buffer object has been assigned to the attribute variable, WebGL requires a final step to "enable" the assignment and make the final connection.

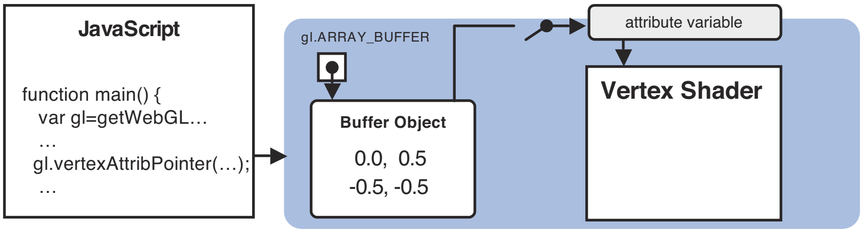


Figure 3.9 Assign a buffer object to an attribute variable

The fifth and final step is to enable the assignment of the buffer object to the attribute variable.

Enable the Assignment to an Attribute Variable (gl.enableVertexAttribArray())

To make it possible to access a buffer object in a vertex shader, we need to enable the assignment of the buffer object to an attribute variable by using `gl.enableVertexAttrib-Array()` as shown in line 77:

```
77    gl.enableVertexAttribArray(a_Position);
```

The following shows the specification of `gl.enableVertexAttribArray()`. Note that we are using the method to handle a buffer even though the method name suggests it's only for use with "vertex arrays." This is not a problem and is simply a legacy from OpenGL.

`gl.enableVertexAttribArray(location)`

Enable the assignment of a buffer object to the attribute variable specified by *location*.

Parameters	location	Specifies the storage location of an attribute variable.
Return value	None	
Errors	INVALID_VALUE	*location* is greater than or equal to the maximum number of attribute variables (8 by default).

When you execute `gl.enableVertexAttribArray()` specifying an attribute variable that has been assigned a buffer object, the assignment is enabled, and the unconnected line is connected as shown in Figure 3.10.

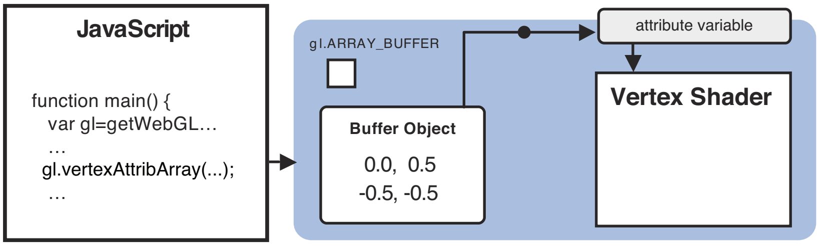


Figure 3.10 Enable the assignment of a buffer object to an attribute variable

You can also break this assignment (disable it) using the method `gl.disableVertexAttribArray()`.

`gl.disableVertexAttribArray(location)`

Disable the assignment of a buffer object to the attribute variable specified by *location*.

Parameters	location	Specifies the storage location of an attribute variable.
Return Value	None	
Errors	INVALID_VALUE	*location* is greater than or equal to the maximum number of attribute variables (8 by default).

Now, everything is set! All you need to do is run the vertex shader, which draws the points using the vertex coordinates specified in the buffer object. As in Chapter 2, you will use the method `gl.drawArrays`, but because you are drawing multiple points, you will actually use the second and third parameters of `gl.drawArrays()`.

Note that after enabling the assignment, you can no longer use `gl.vertexAttrib[1234]f()` to assign data to the attribute variable. You have to explicitly disable the assignment of a buffer object. You can't use both methods simultaneously.

The Second and Third Parameters of gl.drawArrays()

Before entering into a detailed explanation of these parameters, let's take a look at the specification of `gl.drawArrays()` that was introduced in Chapter 2. Following is a recap of the method with only the relevant parts of the specification shown.

`gl.drawArrays(mode, first, count)`

Execute a vertex shader to draw shapes specified by the *mode* parameter.

Parameters	mode	Specifies the type of shape to be drawn. The following symbolic constants are accepted: `gl.POINTS`, `gl.LINES`, `gl.LINE_STRIP`, `gl.LINE_LOOP`, `gl.TRIANGLES`, `gl.TRIANGLE_STRIP`, and `gl.TRIANGLE_FAN`.
	first	Specifies what number-th vertex is used to draw from (integer).
	count	Specifies the number of vertices to be used (integer).

In the sample program this method is used as follows:

```
47   gl.drawArrays(gl.POINTS, 0, n); // n is 3
```

As in the previous examples, because you are simply drawing three points, the first parameter is still `gl.POINTS`. The second parameter *first* is set to 0 because you want to draw from the first coordinate in the buffer. The third parameter *count* is set to 3 because you want to draw three points (in line 47, `n` is 3).

When your program runs line 47, it actually causes the vertex shader to be executed *count* (three) times, sequentially passing the vertex coordinates stored in the buffer object via the attribute variable into the shader (Figure 3.11).

Note that for each execution of the vertex shader, 0.0 and 1.0 are automatically supplied to the z and w components of `a_Position` because `a_Position` requires four components (`vec4`) and you are supplying only two.

Remember that at line 74, the second parameter *size* of `gl.vertexAttribPointer()` is set to 2. As just discussed, the second parameter indicates how many coordinates per vertex are specified in the buffer object and, because you are only specifying the x and y coordinates in the buffer, you set the size value to 2:

```
74   gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
```

After drawing all points, the content of the color buffer is automatically displayed in the browser (bottom of Figure 3.11), resulting in our three red points, as shown in Figure 3.2.

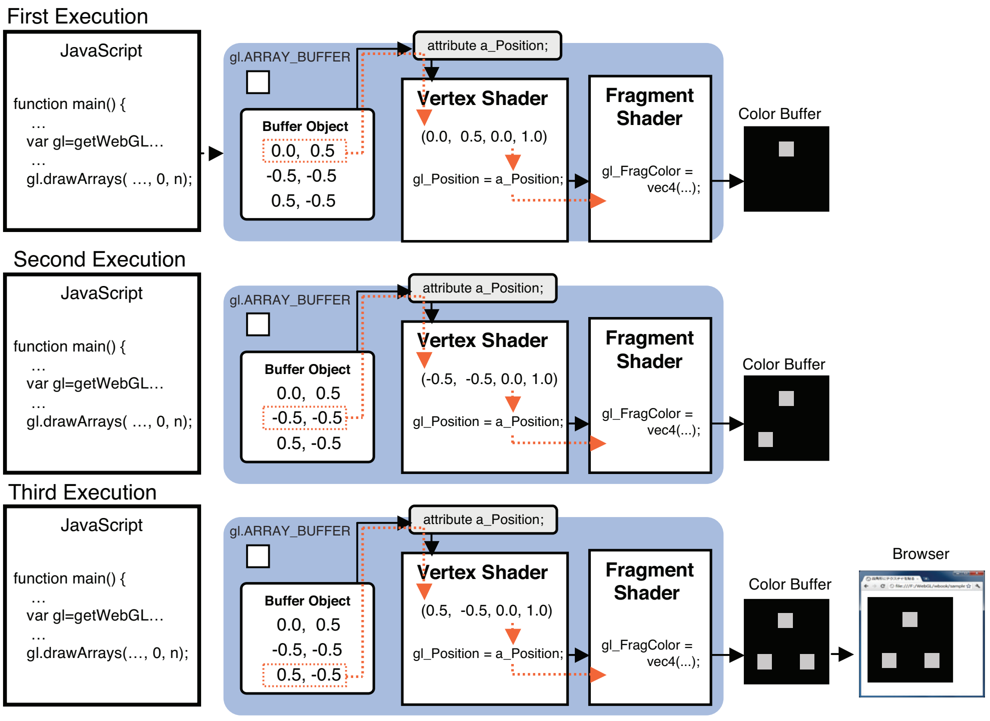


Figure 3.11 How the data in a buffer object is passed to a vertex shader during execution

Experimenting with the Sample Program

Let's experiment with the sample program to better understand how `gl.drawArrays()` works by modifying the second and third parameters. First, let's specify 1 as the third argument for *count* at line 47 instead of our variable `n` (set to 3) as follows:

```
47  gl.drawArrays(gl.POINTS, 0, 1);
```

In this case, the vertex shader is executed only once, and a single point is drawn using the first vertex in the buffer object.

If you now specify 1 as the second argument, only the second vertex is used to draw a point. This is because you are telling WebGL that you want to start drawing from the second vertex and you only want to draw one vertex. So again, you will see only a single point, although this time it is the second vertex coordinates that are shown in the browser:

```
47  gl.drawArrays(gl.POINTS, 1, 1);
```

This gives you a quick feel for the role of the parameters *first* and *count*. However, what will be happen if you change the first parameter *mode*? The next section explores the first parameter in more detail.

Hello Triangle

Now that you've learned the basic techniques to pass multiple vertex coordinates to a vertex shader, let's try to draw other shapes using multiple vertex coordinates. This section uses a sample program `HelloTriangle`, which draws a single 2D triangle. Figure 3.12 shows a screenshot of `HelloTriangle`.

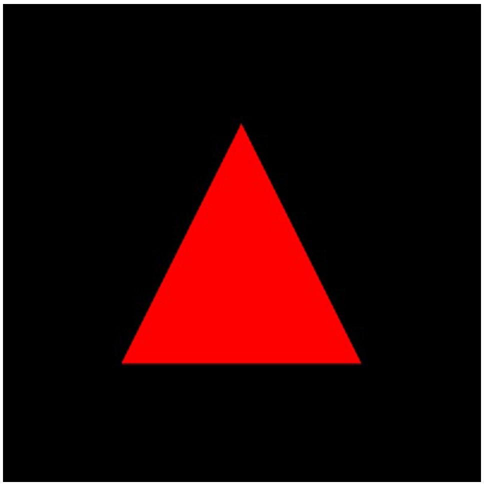


Figure 3.12 HelloTriangle

Sample Program (HelloTriangle.js)

Listing 3.3 shows `HelloTriangle.js`, which is almost identical to `MultiPoint.js` used in the previous section with two critical differences.

Listing 3.3 HelloTriangle.js

```
// HelloTriangle.js
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'void main() {\n' +
  '  gl_Position = a_Position;\n' +
  '}\n';

```

```
 9 // Fragment shader program
10 var FSHADER_SOURCE =
11   'void main() {\n' +
12   '  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n' +
13   '}\n';
14
15 function main() {
    ...
19   // Get the rendering context for WebGL
20   var gl = getWebGLContext(canvas);
     ...
26   // Initialize shaders
27   if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
     ...
30   }
31
32   // Set the positions of vertices
33   var n = initVertexBuffers(gl);
...
39   // Set the color for clearing <canvas>
     ...
45   // Draw a triangle
46   gl.drawArrays(gl.TRIANGLES, 0, n);
47 }
48
49 function initVertexBuffers(gl) {
50   var vertices = new Float32Array([
51     0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
52   ]);
53   var n = 3; // The number of vertices
     ...
78   return n;
79 }
```

The two differences from `MultiPoint.js` are

- The line to specify the size of a point `gl_PointSize = 10.0`; has been removed from the vertex shader. This line only has an effect when you are drawing a point.
- The first parameter of `gl.drawArrays()` has been changed from `gl.POINTS` to `gl.TRIANGLES` at line 46.

The first parameter, *mode,* of `gl.drawArrays()` is powerful and provides the ability to draw various shapes. Let's take a look.

Basic Shapes

By changing the argument we use for the first parameter, *mode,* of `gl.drawArrays()`, we can change the meaning of line 46 into "execute the vertex shader three times (`n` is 3), and draw a triangle using the three vertices in the buffer, starting from the first vertex coordinate":

```
46   gl.drawArrays(gl.TRIANGLES, 0, n);
```

In this case, the three vertices in the buffer object are no longer individual points, but become three vertices of a triangle.

The WebGL method `gl.drawArrays()` is both powerful and flexible, allowing you to specify seven different types of basic shapes as the first argument. These are explained in more detail in Table 3.3. Note that v0, v1, v2 ... indicates the vertices specified in a buffer object. The order of vertices affects the drawing of the shape.

The shapes in the table are the only ones that WebGL can draw directly, but they are the basics needed to construct complex 3D graphics. (Remember the frog at the start of this chapter.)

Table 3.3 Basic Shapes Available in WebGL

Basic Shape	**Mode**	**Description**
Points	`gl.POINTS`	A series of points. The points are drawn at v0, v1, v2 ...
Line segments	`gl.LINES`	A series of unconnected line segments. The individual lines are drawn between vertices given by (v0, v1), (v2, v3), (v4, v5)... If the number of vertices is odd, the last one is ignored.
Line strips	`gl.LINE_STRIP`	A series of connected line segments. The line segments are drawn between vertices given by (v0, v1), (v1, v2), (v2, v3), ... The first vertex becomes the start point of the first line, the second vertex becomes the end point of the first line and the start point of the second line, and so on. The *i*-th (*i*> 1) vertex becomes the start point of the *i*-th line and the end point of the *i-1*-th line. (The last vertex becomes the end point of the last line.)
Line loops	`gl.LINE_LOOP`	A series of connected line segments. In addition to the lines drawn by `gl.LINE_STRIP`, the line between the last vertex and the first vertex is drawn. The line segments drawn are (v0, v1), (v1, v2), ..., and (v*n*, v0). v*n* is the last vertex.
Triangles	`gl.TRIANGLES`	A series of separate triangles. The triangles given by vertices (v0, v1, v2), (v3, v4, v5), ... are drawn. If the number of vertices is not a multiple of 3, the remaining vertices are ignored.

Basic Shape	Mode	Description
Triangle strips	`gl.TRIANGLE_STRIP`	A series of connected triangles in strip fashion. The first three vertices form the first triangle and the second triangle is formed from the next vertex and one of the sides of the first triangle. The triangles are drawn given by (v0, v1, v2), (v2, v1, v3), (v2, v3, v4) ... (Pay attention to the order of vertices.)
Triangle fans	`gl.TRIANGLE_FAN`	A series of connected triangles sharing the first vertex in fan-like fashion. The first three vertices form the first triangle and the second triangle is formed from the next vertex, one of the sides of the first triangle, and the first vertex. The triangles are drawn given by (v0, v1, v2), (v0, v2, v3), (v0, v3, v4), ...

Figure 3.13 shows these basic shapes.

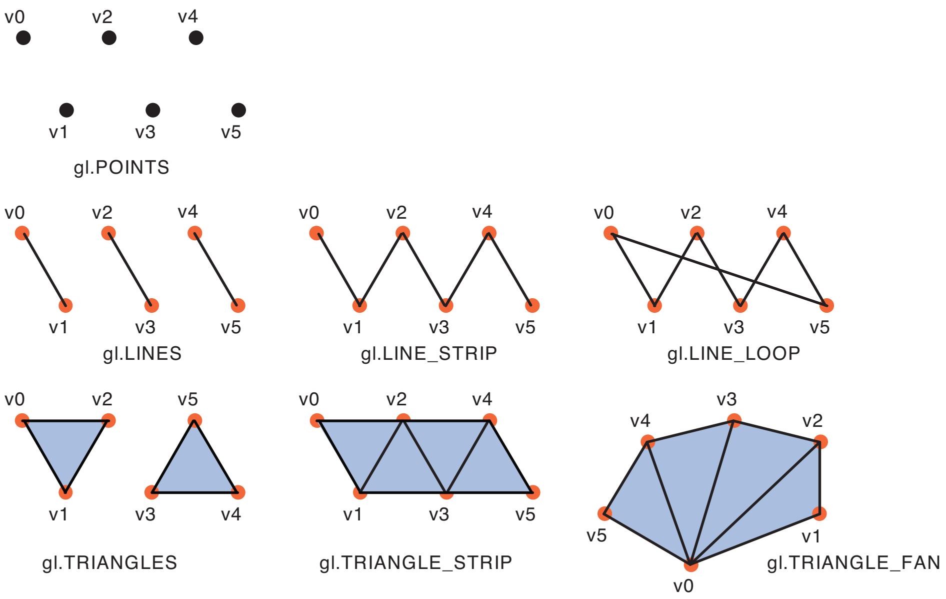


Figure 3.13 Basic shapes available in WebGL

As you can see from the figure, WebGL can draw only three types of shapes: a point, a line, and a triangle. However, as explained at the beginning of this chapter, spheres to cubes to 3D monsters to humanoid characters in a game can be constructed from small triangles. Therefore, you can use these basic shapes to draw anything.

Experimenting with the Sample Program

To examine what will happen when using `gl.LINES`, `gl.LINE_STRIP`, and `gl.LINE_LOOP`, let's change the first argument of `gl.drawArrays()` as shown next. The name of each sample program is `HelloTriangle_LINES`, `HelloTriangle_LINE_STRIP`, and `HelloTriangle_LINE_LOOP`, respectively:

```
gl.drawArrays(gl.LINES, 0, n);
gl.drawArrays(gl.LINE_STRIP, 0, n);
gl.drawArrays(gl.LINE_LOOP, 0, n);
```

Figure 3.14 shows a screenshot of each program.

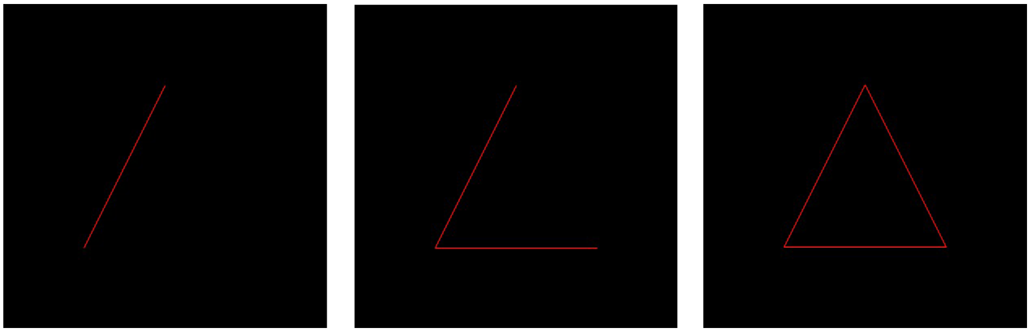


Figure 3.14 gl.LINES, gl.LINE_STRIP, and gl.LINE_LOOP

As you can see, `gl.LINES` draws a line using the first two vertices and does not use the last vertex, whereas `gl.LINE_STRIP` draws two lines using the first three vertices. Finally, `gl.LINE_LOOP` draws the lines in the same manner as `gl.LINE_STRIP` but then “loops” between the last vertex and the first vertex and makes a triangle.

Hello Rectangle (HelloQuad)

Let's use this basic way of drawing triangles to draw a rectangle. The name of the sample program is `HelloQuad`, and Figure 3.15 shows a screenshot when it's loaded into your browser.

Figure 3.16 shows the vertices of the rectangle. Of course, the number of vertices is four because it is a rectangle. As explained in the previous section, WebGL cannot draw a rectangle directly, so you need to divide the rectangle into two triangles (v0, v1, v2) and (v2, v1, v3) and then draw each one using `gl.TRIANGLES`, `gl.TRIANGLE_STRIP`, or `gl.TRIANGLE_FAN`. In this example, you'll use `gl.TRIANGLE_STRIP` because it only requires you to specify four vertices. If you were to use `gl.TRIANGLES`, you would need to specify a total of six.

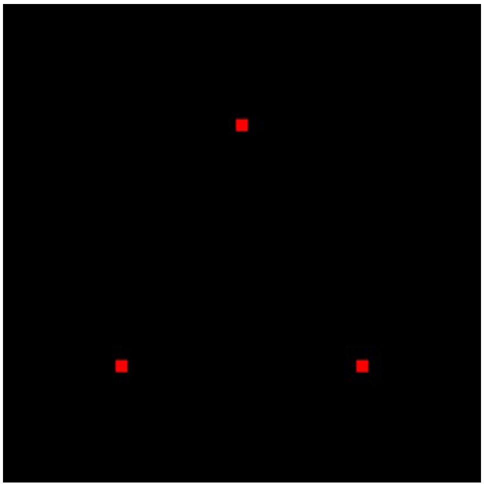


Figure 3.15 HelloQuad

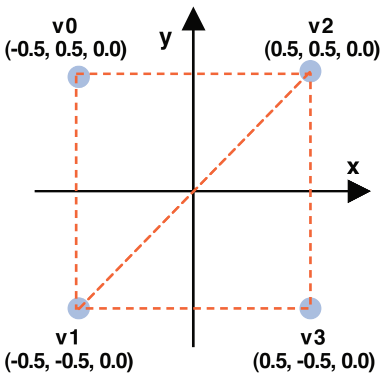


Figure 3.16 The four vertex coordinates of the rectangle

Basing the example on `HelloTriangle.js`, you need to add an extra vertex coordinate at line 50. Pay attention to the order of vertices; otherwise, the draw command will not execute correctly:

```
50   var vertices = new Float32Array([
51     -0.5, 0.5,   -0.5, -0.5,   0.5, 0.5,   0.5, -0.5
52   ]);
```

Because you've added a fourth vertex, you need to change the number of vertices from 3 to 4 at line 53:

```
53   var n = 4; // The number of vertices
```

Then, by modifying line 46 as follows, your program will draw a rectangle in the browser:

```
46   gl.drawArrays(gl.TRIANGLE_STRIP, 0, n);
```

Experimenting with the Sample Program

Now that you have a feel for how to use `gl.TRIANGLE_STRIP`, let's change the first parameter of `gl.drawArrays()` to `gl.TRIANGLE_FAN`. The name of the sample program is `HelloQuad_FAN`:

```
46   gl.drawArrays(gl.TRIANGLE_FAN, 0, n);
```

Figure 3.17 show a screenshot of `HelloQuad_FAN`. In this case, we can see the ribbon-like shape on the screen.

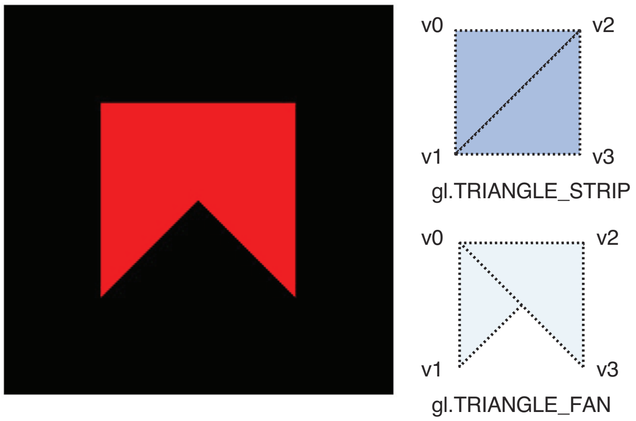


Figure 3.17 HelloQuad_FAN

Looking at the order of vertices and the triangles drawn by `gl.TRIANGLE_FAN` shown on the right side of Figure 3.17, you can see why the result became a ribbon-like shape. Essentially, `gl.TRIANGLE_FAN` causes WebGL to draw a second triangle that shares the first vertex (v0), and this second triangle overlaps the first, creating the ribbon-like effect.

Moving, Rotating, and Scaling

Now that you understand the basics of drawing shapes like triangles and rectangles, let's take another step and try to move (translate), rotate, and scale the triangle and display the results on the screen. These operations are called **transformations (affine transformations)**. This section introduces some math to explain each transformation and help you to understand how each operation can be realized. However, when you write your own programs, you don't need the math; instead, you can use one of several convenient libraries, explained in the next section, that handle the math for you.

If you find reading this section and in particular the math too much on first read, it's okay to skip it and return later. Or, if you already know that transformations can be written using a matrix, you can skip this section as well.

First, let's write a sample program, `TranslatedTriangle`, that moves a triangle 0.5 units to the right and 0.5 units up. You can use the triangle you drew in the previous section. The right direction means the positive direction of the x-axis, and the up direction means the positive direction of the y-axis. (See the coordinate system in Chapter 2.) Figure 3.18 shows `TranslatedTriangle`.

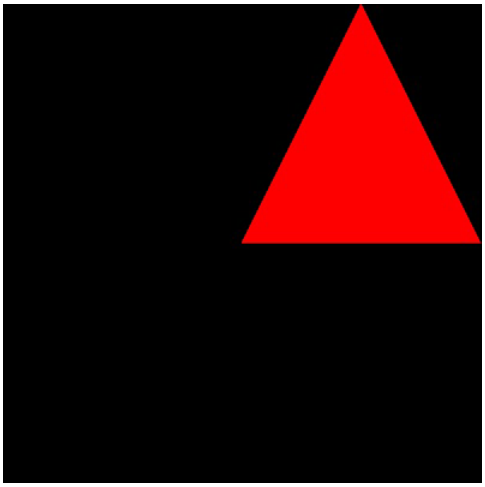


Figure 3.18 TranslatedTriangle

Translation

Let us examine what kind of operations you need to apply to each vertex coordinate of a shape to translate (move) the shape. Essentially, you just need to add a translation distance for each direction (x and y) to each component of the coordinates. Looking at Figure 3.19, the goal is to translate the point p (x, y, z) to the point p' (x', y', z'), so the translation distance for the x, y, and z direction is `Tx`, `Ty`, and `Tz`, respectively. In this figure, `Tz` is 0.

To determine the coordinates of p', you simply add the T values, as shown in Equation 3.1.

Equation 3.1

$$x' = x + Tx$$

$$y' = y + Ty$$

$$z' = z + Tz$$

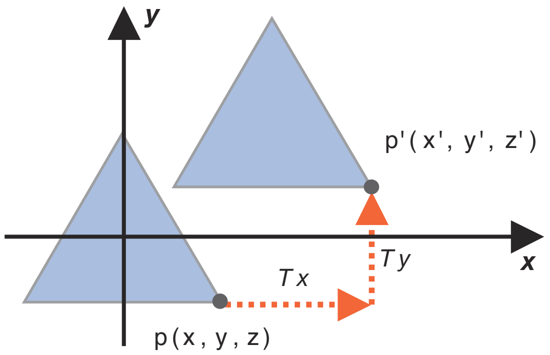


Figure 3.19 Calculating translation distances

These simple equations can be implemented in a WebGL program just by adding each constant value to each vertex coordinate. You've probably realized already that because they are a **per-vertex operation**, you need to implement the operations in a vertex shader. Conversely, they clearly aren't a per-fragment operation, so you don't need to worry about the fragment shader.

Once you understand this explanation, implementation is easy. You need to pass the translation distances `Tx`, `Ty`, and `Tz` to the vertex shader, apply Equation 3.1 using the distances, and then assign the result to `gl_Position`. Let's look at a sample program that does this.

Sample Program (TranslatedTriangle.js)

Listing 3.4 shows `TranslatedTriangle.js`, in which the vertex shader is partially modified to carry out the translation operation. However, the fragment shader is the same as in `HelloTriangle.js` in the previous section. To support the modification to the vertex shader, some extra code is added to the `main()` function in the JavaScript.

Listing 3.4 TranslatedTriangle.js

```
// TranslatedTriangle.js
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'uniform vec4 u_Translation;\n' +
  'void main() {\n' +
  '  gl_Position = a_Position + u_Translation;\n' +
  '}\n';

// Fragment shader program
    ...
```

```
16 // The translation distance for x, y, and z direction
17 var Tx = 0.5, Ty = 0.5, Tz = 0.0;
18
19 function main() {
    ...
23   // Get the rendering context for WebGL
24   var gl = getWebGLContext(canvas);
    ...
30   // Initialize shaders
31   if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
    ...
34  }
35
36   // Set the positions of vertices
37   var n = initVertexBuffers(gl);
    ...
43   // Pass the translation distance to the vertex shader
44   var u_Translation = gl.getUniformLocation(gl.program, 'u_Translation');
    ...
49   gl.uniform4f(u_Translation, Tx, Ty, Tz, 0.0);
50
51   // Set the color for clearing <canvas>
    ...
57   // Draw a triangle
58   gl.drawArrays(gl.TRIANGLES, 0, n);
59 }
60
61 function initVertexBuffers(gl) {
62   var vertices = new Float32Array([
63     0.0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
64   ]);
65   var n = 3; // The number of vertices
    ...
90   return n;
93 }
```

First, let's examine `main()` in JavaScript. Line 17 defines the variables for each translation distance of Equation 3.1:

```
17 var Tx = 0.5, Ty = 0.5, Tz = 0.0;
```

Because `Tx`, `Ty`, and `Tz` are fixed (uniform) values for all vertices, you use the uniform variable `u_Translation` to pass them to a vertex shader. Line 44 retrieves the storage location of the uniform variable, and line 49 assigns the data to the variable:

```
44   var u_Translation = gl.getUniformLocation(gl.program, 'u_Translation');
    ...
49   gl.uniform4f(u_Translation, Tx, Ty, Tz, 0.0);
```

Note that `gl.uniform4f()` requires a homogenous coordinate, so we supply a fourth argument (w) of 0.0. This will be explained in more detail later in this section.

Now, let's take a look at the vertex shader that uses this translation data. As you can see, the uniform variable `u_Translation` in the shader, to which the translation distances are passed, is defined as type `vec4` at line 5. This is because you want to add the components of `u_Translation` to the vertex coordinates passed to `a_Position` (as defined by Equation 3.1) and then assign the result to the variable `gl_Position`, which has type `vec4`. Remember, per Chapter 2, that the assignment operation in GLSL ES is only allowed between variables of the same types:

```
4   'attribute vec4 a_Position;\n' +
5   'uniform vec4 u_Translation;\n' +
6   'void main() {\n' +
7   '  gl_Position = a_Position + u_Translation;\n' +
8   '}\n';
```

After these preparations have been completed, the rest of tasks are straightforward. To calculate Equation 3.1 within the vertex shader, you just add each translation distance (Tx, Ty, Tz) passed in `u_Translation` to each vertex coordinate (x, y, z) passed in `a_Position`.

Because both variables are of type `vec4`, you can use the `+` operator, which will actually add the four components simultaneously (see Figure 3.20). This easy addition of vectors is a feature of GLSL ES and will be explained in more detail in Chapter 6.

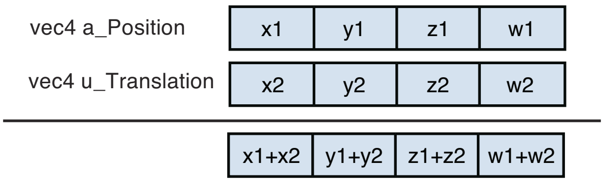


Figure 3.20 Addition of vec4 variables

Now, we'll return to the fourth element, (w), of the vector. As explained in Chapter 2, you need to specify the homogeneous coordinate to `gl_Position`, which is a four-dimensional coordinate. If the last component of the homogeneous coordinate is 1.0, the coordinate indicates the same position as the three-dimensional coordinate. In this case, because the last component is `w1+w2` to ensure that `w1+w2` is 1.0, you need to specify 0.0 to the value of w (the fourth parameter of `gl.uniform4f()`).

Finally, at line 58, `gl.drawArrays(gl.TRIANGLES, 0, n)` executes the vertex shader. For each execution, the following three steps are performed:

1. Each vertex coordinate set is passed to `a_Position`.
2. `u_Translation` is added to `a_Position`.
3. The result is assigned to `gl_Position`.

Once executed, you've achieved your goal because each vertex coordinate set is modified (translated), and then the translated shape (in this case, a triangle) is displayed on the screen. If you now load `TranslatedTriangle.html` into your browser, you will see the translated triangle.

Now that you've mastered translation (moving), the next step is to look at rotation. The basic approach to realize rotation is the same as translation, requiring you to manipulate the vertex coordinates in the vertex shader.

Rotation

Rotation is a little more complex than translation because you have to specify multiple items of information. The following three items are required:

- Rotation axis (the axis the shape will be rotated around)
- Rotation direction (the direction: clockwise or counterclockwise)
- Rotation angle (the number of degrees the shape will be rotated through)

In this section, to simplify the explanation, you can assume that the rotation is performed around the `z`-axis, in a counterclockwise direction, and for β degrees. You can use the same approach to implement other rotations around the x-axis or y-axis.

In the rotation, if β is positive, the rotation is performed in a counterclockwise direction around the rotation axis looking at the shape toward the negative direction of the z-axis (see Figure 3.21); this is called **positive rotation**. Just as for the coordinate system, your hand can define the direction of rotation. If you take your right hand and have your thumb follow the direction of the rotation axis, your fingers show the direction of rotation. This is called the **right-hand-rule rotation**. As we discussed in Chapter 2, it's the default we are using for WebGL in this book.

Now let's find the expression to calculate the rotation in the same way that you did for translation. As shown in Figure 3.22, we assume that the point p' (x', y', z') is the β degree rotated point of p (x, y, z) around the z-axis. Because the rotation is around the z-axis, the z coordinate does not change, and you can ignore it for now. The explanation is a little mathematical, so let's take it a step at a time.

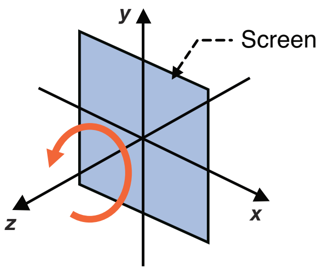


Figure 3.21 Positive rotation around the z-axis

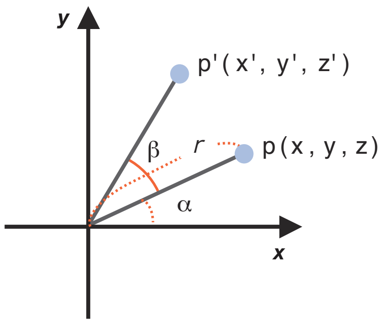


Figure 3.22 Calculating rotation around the z-axis

In Figure 3.22, r is the distance from the origin to the point p, and α is the rotation angle from the x-axis to the point. You can use these items of information to represent the coordinates of p, as shown in Equation 3.2.

Equation 3.2

$$x = r \cos \alpha$$

$$y = r \sin \alpha$$

Similarly, you can find the coordinate of p' by using r, α, and β as follows:

$$x' = r \cos (\alpha + \beta)$$

$$y' = r \sin (\alpha + \beta)$$

Then you can use the addition theorem of trigonometric functions[1] to get the following:

$$x' = r(\cos\alpha\cos\beta - \sin\alpha\sin\beta)$$

$$y' = r(\sin\alpha\cos\beta + \cos\alpha\sin\beta)$$

Finally, you get the following expressions (Equation 3.3) by assigning Equation 3.2 to the previous expressions and removing `r` and α.

Equation 3.3

$$x' = x\cos\beta - y\sin\beta$$

$$y' = x\sin\beta + y\cos\beta$$

$$z' = z$$

So by passing the values of sin β and cos β to the vertex shader and then calculating Equation 3.3 in the shader, you get the coordinates of the rotated point. To calculate sin β and cos β, you can use the methods of the JavaScript `Math` object.

Let's look at a sample program, `RotatedTriangle`, which rotates a triangle around the z-axis, in a counterclockwise direction, by 90 degrees. Figure 3.23 shows `RotatedTriangle`.

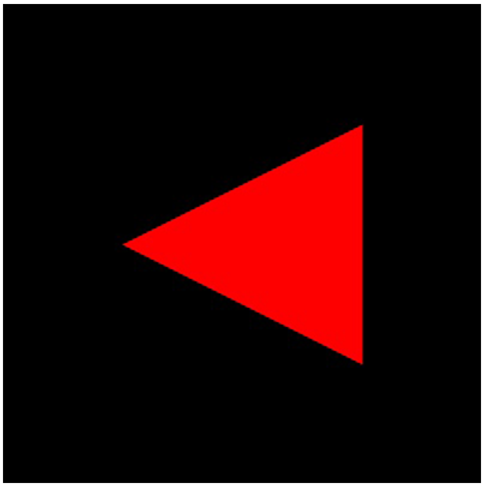


Figure 3.23 RotatedTriangle

[1] $\sin(a \pm \mathbf{b}) = \sin a \cos b \mp \cos a \sin b$
$\cos(a \pm b) = \cos a \cos b \mp \sin a \sin b$

Sample Program (RotatedTriangle.js)

Listing 3.5 shows `RotatedTriangle.js` which, in a similar manner to `TranslatedTriangle.js`, modifies the vertex shader to carry out the rotation operation. The fragment shader is the same as in `TranslatedTriangle.js` and, as usual, is not shown. Again, to support the shader modification, several processing steps are added to `main()` in the JavaScript program. Additionally, Equation 3.3 is added in the comments from lines 4 to 6 to remind you of the calculation needed.

Listing 3.5 RotatedTriangle.js

```
 1  // RotatedTriangle.js
 2  // Vertex shader program
 3  var VSHADER_SOURCE =
 4   // x' = x cos b - y sin b
 5   // y' = x sin b + y cos b                          Equation 3.3
 6   // z' = z
 7   'attribute vec4 a_Position;\n' +
 8   'uniform float u_CosB, u_SinB;\n' +
 9   'void main() {\n' +
10   '  gl_Position.x = a_Position.x * u_CosB - a_Position.y *u_SinB;\n'+
11  '  gl_Position.y = a_Position.x * u_SinB + a_Position.y * u_CosB;\n'+
12   '  gl_Position.z = a_Position.z;\n' +
13   '  gl_Position.w = 1.0;\n' +
14   '}\n';
15
16  // Fragment shader program
     ...
22  // Rotation angle
23  var ANGLE = 90.0;
24
25  function main() {
      ...
42    // Set the positions of vertices
43    var n = initVertexBuffers(gl);
      ...
49    // Pass the data required to rotate the shape to the vertex shader
50    var radian = Math.PI * ANGLE / 180.0; // Convert to radians
51    var cosB = Math.cos(radian);
52    var sinB = Math.sin(radian);
53
54    var u_CosB = gl.getUniformLocation(gl.program, 'u_CosB');
55    var u_SinB = gl.getUniformLocation(gl.program, 'u_SinB');
      ...
60   gl.uniform1f(u_CosB, cosB);
61   gl.uniform1f(u_SinB, sinB);
62
63    // Set the color for clearing <canvas>
       ...
69    // Draw a triangle
70    gl.drawArrays(gl.TRIANGLES, 0, n);
71  }
72
73  function initVertexBuffers(gl) {
74    var vertices = new Float32Array([
75      0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
76    ]);
77    var n = 3; // The number of vertices
       ...
105    return n;
106  }
```

Let's look at the vertex shader, which is straightforward:

```
 2  // Vertex shader program
 3  var VSHADER_SOURCE =
 4   // x' = x cos b - y sin b
 5   // y' = x sin b + y cos b
 6   // z' = z
 7   'attribute vec4 a_Position;\n' +
 8   'uniform float u_CosB, u_SinB;\n' +
 9   'void main() {\n' +
10   '  gl_Position.x = a_Position.x * u_CosB - a_Position.y * u_SinB;\n'+
11   '  gl_Position.y = a_Position.x * u_SinB + a_Position.y * u_CosB;\n'+
12   '  gl_Position.z = a_Position.z;\n' +
13   '  gl_Position.w = 1.0;\n' +
14   '}\n';
```

Because the goal is to rotate the triangle by 90 degrees, the sine and cosine of 90 need to be calculated. Line 8 defines two uniform variables for receiving these values, which are calculated in the JavaScript program and then passed to the vertex shader.

You could pass the rotation angle to the vertex shader and then calculate the values of sine and cosine in the shader. However, because they are identical for all vertices, it is more efficient to do it once in the JavaScript.

The name of these uniform variables, `u_CosB` and `u_SinB`, are defined following the naming rule used throughout this book. As you will remember, you use the uniform variable because the values of these variables are uniform (unchanging) per vertex.

As in the previous sample programs, x, y, z, and w are passed in a group to the attribute variable `a_Position` in the vertex shader. To apply Equation 3.3 to x, y, and z, you need to access each component in `a_Position` separately. You can do this easily using the `.` operator, such as `a_Position.x`, `a_Position.y`, and `a_Position.z` (see Figure 3.24 and Chapter 6).

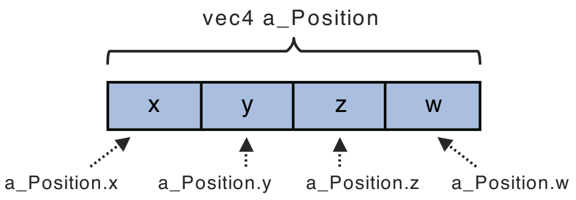


Figure 3.24 Access methods for each component in a vec4

Handily, you can use the same operator to access each component in `gl_Position` to which the vertex coordinate is written, so you can calculate x' = x cos β – y sin β from Equation 3.3 as shown at line 10:

```
10   '  gl_Position.x = a_Position.x * u_CosB - a_Position.y * u_SinB;\n'+
```

Similarly, you can calculate y' as follows:

```
11   '  gl_Position.y = a_Position.x * u_SinB + a_Position.y * u_CosB;\n'+
```

According to Equation 3.3, you just need to assign the original z coordinate to z' directly at line 12. Finally, you need to assign 1.0 to the last component `w`[2]:

```
12   '  gl_Position.z = a_Position.z;\n' +
13   '  gl_Position.w = 1.0;\n' +
```

Now look at `main()` in the JavaScript code, which starts from line 25. This code is mostly the same as in `TranslatedTriangle.js`. The only difference is passing cos β and sin β to the vertex shader. To calculate the sine and cosine of β, you can use the JavaScript `Math.sin()` and `Math.cos()` methods. However, these methods expect parameters in radians, not degrees, so you need to convert from degrees to radians by multiplying the number of degrees by pi and then dividing by 180. You can utilize `Math.PI` as the value of pi as shown at line 50, where the variable `ANGLE` is defined as 90 (degrees) at line 23:

```
50   var radian = Math.PI * ANGLE / 180.0; // Converts degrees to radians
```

[2] In this program, you can also write `gl_Position.w = a_Position.w;` because `a_Position.w` is 1.0.

Once you have the angle in radians, lines 51 and 52 calculate cos β and sin β, and then lines 60 and 61 pass them to the uniform variables in the vertex shader:

```
51    var cosB = Math.cos(radian);
52    var sinB = Math.sin(radian);
53
54    var u_CosB = gl.getUniformLocation(gl.program, 'u_CosB');
55    var u_SinB = gl.getUniformLocation(gl.program, 'u_SinB');
        ...
60    gl.uniform1f(u_CosB, cosB);
61    gl.uniform1f(u_SinB, sinB);
```

When you load this program into your browser, you can see the triangle, rotated through 90 degrees, on the screen. If you specify a negative value to `ANGLE`, you can rotate the triangle in the opposite direction (clockwise). You can also use the same equation. For example, to rotate the triangle in the clockwise direction, you can specify –90 instead of 90 at line 23, and `Math.cos()` and `Math.sin()` will deal with the remaining tasks for you.

For those of you concerned with speed and efficiency, the approach taken here (using two uniform variables to pass the values of cos β and sin β) isn't optimal. To pass the values as a group, you can define the uniform variable as follows:

```
uniform vec2 u_CosBSinB;
```

and then pass the values by:

```
gl.uniform2f(u_CosBSinB,cosB, sinB);
```

Then in the vertex shader, you can access them using `u_CosBSinB.x` and `u_CosBSinB.y`.

Transformation Matrix: Rotation

For simple transformations, you can use mathematical expressions. However, as your needs become more complex, you'll quickly find that applying a series of equations becomes quite complex. For example a "translation after rotation" as shown in Figure 3.25 can be realized by using Equations 3.1 and 3.3 to find the new mathematical expressions for the transformation and then implementing them in a vertex shader.

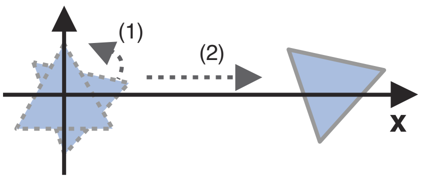


Figure 3.25 Rotate first and then translate a triangle

However, it is time consuming to determine the mathematical expressions every time you need a new set of transformation and then implement them in a vertex shader. Fortunately, there is another tool in the mathematical toolbox, the **transformation matrix**, which is excellent for manipulating computer graphics.

As shown in Figure 3.26, a matrix is a rectangular array of numbers arranged in rows (in the horizontal direction) and columns (in the vertical direction). This notation makes it easy to write the calculations explained in the previous sections. The brackets indicate that these numbers are a group.

$$\begin{bmatrix} 8 & 3 & 0 \\ 4 & 3 & 6 \\ 3 & 2 & 6 \end{bmatrix}$$

Figure 3.26 Example of a matrix

Before explaining the details of how to use a transformation matrix to replace the equations used here, you need to make sure you understand the multiplication of a matrix and a vector. A vector is an object represented by an n-tuple of numbers, such as the vertex coordinates (0.0, 0.5, 1.0).

The multiplication of a matrix and a vector can be written as shown in Equation 3.4. (Although the multiply operator × is often omitted, we explicitly write the operator in this book for clarity.) Here, our new vector (on the left) is the result of multiplying a matrix (in the center) by our original vector (on the right). Note that matrix multiplication is noncommutative. In other words, A × B is not the same as B × A. We discuss this further in Chapter 6.

Equation 3.4

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

This matrix has three rows and three columns and is called a 3×3 matrix. The rightmost part of the equation is a vector composed of x, y, and z. (In the case of a multiplication of a matrix and vector, the vector is written vertically, but it has the same meaning as when it is written horizontally.) This vector has three elements, so it is called a three-dimensional vector. Again, the brackets on both sides of the array of numbers (vector) are also just notation for recognizing that these numbers are a group.

In this case, x', y', and z' are defined using the elements of the matrix and the vector, as shown by Equation 3.5. Note that the multiplication of a matrix and vector can be defined only if the number of columns in a matrix matches the number of rows in a vector.

Equation 3.5

$$x' = ax + by + cz$$

$$y' = dx + ey + fz$$

$$z' = gx + hy + iz$$

Now, to understand how to use a matrix instead of our original equations, let's compare the matrix equations and Equation 3.3 (shown again as Equation 3.6).

Equation 3.6

$$x' = x \cos \beta - y \sin \beta$$

$$y' = x \sin \beta + y \cos \beta$$

$$z' = z$$

For example, compare the equation for x':

$$x' = ax + by + cz$$

$$x' = x \cos \beta - y \sin \beta$$

In this case, if you set a = cos β, b = -sin β, and c = 0, the equations become the same. Similarly, let us compare the equation for y':

$$y' = dx + ey + fz$$

$$y' = x \sin \beta + y \cos \beta$$

In this case, if you set d = sin β, e = cos β, and f = 0, you get the same equation. The last equation about z' is easy. If you set g = 0, h = 0, and i = 1, you get the same equation.

Then, by assigning these results to Equation 3.4, you get Equation 3.7.

Equation 3.7

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\beta & -\sin\beta & 0 \\ \sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

This matrix is called a **transformation matrix** because it "transforms" the right-side vector (x, y, z) to the left-side vector (x', y', z'). The transformation matrix representing a rotation is called a **rotation matrix**.

You can see that the elements of the matrix in Equation 3.7 are an array of coefficients in Equation 3.6. Once you become accustomed to matrix notation, it is easier to write and use matrices than to have to deal with a set of transformation equations.

As you would expect, because matrices are used so often in 3DCG, multiplication of a matrix and a vector is easy to implement in shaders. However, before exploring how, let's quickly look at other types of transformation matrices, and then we will start to use them in shaders.

Transformation Matrix: Translation

Obviously, if we can use a transformation matrix to represent a rotation, we should be able to use it for other types of transformation, such as translation. For example, let us compare the equation for x' in Equation 3.1 to that in Equation 3.5 as follows:

$x' = ax + by + cz$ --- from Equation (3.5)

$x' = x + T_x$ --- from Equation (3.1)

Here, the second equation has the constant term T_x, but the first one does not, meaning that you cannot deal with the second one by using the 3×3 matrix of the first equation. To solve this problem, you can use a 4×4 matrix and the fourth components of the coordinate, which are set to 1 to introduce the constant terms. That is to say, we assume that the coordinates of point p are (x, y, z, 1), and the coordinates of the translated point p (p') are (x', y', z', 1). This gives us Equation 3.8.

Equation 3.8

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

This multiplication is defined as follows:

Equation 3.9

$$x' = ax + by + cz + d$$

$$y' = ex + fy + gz + h$$

$$z' = ix + jy + kz + l$$

$$1 = mx + ny + oz + p$$

From the equation1 = mx + ny + oz + p, it is easy to find that the coefficients are m = 0, n = 0, o = 0, and p = 1. In addition, these equations have the constant terms d, h, and l, which look helpful to deal with Equation 3.1 because it also has constant terms. Let us compare Equation 3.9 and Equation 3.1 (translation), which is reproduced again:

$$x' = x + T_x$$

$$y' = y + T_y$$

$$z' = z + T_z$$

When you compare the x' component of both equations, you can see that a=1, b=0, c=0, and d=T_x. Similarly, when comparing y' from both equations, you find e = 0, f = 1, g = 0, and h = T_y; when comparing z' you see i=0, j=0, k=1, and *l*=T_z. You can use these results to write a matrix that represents a translation, called a **translation matrix**, as shown in Equation 3.10.

Equation 3.10

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Rotation Matrix, Again

At this stage you have successfully created a rotation and a translation matrix, which are equivalent to the two equations you used in the example programs earlier. The final step is to combine these two matrices; however, the rotation matrix (3×3 matrix) and transformation matrix (4×4 matrix) have different numbers of elements. Unfortunately, you cannot combine matrices of different sizes, so you need a mechanism to make them the same size.

To do that, you need to change the rotation matrix (3×3 matrix) into a 4×4 matrix. This is straightforward and requires you to find the coefficient of each equation in Equation 3.9 by comparing it with Equation 3.3. The following shows both equations:

$$x' = x \cos \beta - y \sin \beta$$

$$y' = x \sin \beta + y \cos \beta$$

$$z' = z$$

$$x' = ax + by + cz + d$$

$$y' = ex + fy + gz + h$$

$$z' = ix + iy + kz + l$$

$$1 = mx + ny + oz + p$$

For example, when you compare x' = x cos β – y sin β with x' = ax + by + cz +d, you find a = cos β, b = –sin β, c = 0, and d = 0. In the same way, after comparing in terms of y and z, you get the rotation matrix shown in Equation 3.11:

Equation 3.11

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\beta & -\sin\beta & 0 & 0 \\ \sin\beta & \cos\beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

This allows you to represent both a rotation matrix and translation matrix in the same 4×4 matrix, achieving the original goal!

Sample Program (RotatedTriangle_Matrix.js)

Having constructed a 4×4 rotation matrix, let's go ahead and use this matrix in a WebGL program by rewriting the sample program `RotatedTriangle`, which rotates a triangle 90 degrees around the z-axis in a counterclockwise direction, using the rotation matrix. Listing 3.6 shows `RotatedTriangle_Matrix.js`, whose output will be the same as Figure 3.23 shown earlier.

Listing 3.6 RotatedTriangle_Matrix.js

```
// RotatedTriangle_Matrix.js
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'uniform mat4 u_xformMatrix;\n' +
  'void main() {\n' +
  '  gl_Position = u_xformMatrix * a_Position;\n' +
  '}\n';

// Fragment shader program
...
// Rotation angle
var ANGLE = 90.0;
```

```

function main() {
  ...
  // Set the positions of vertices
  var n = initVertexBuffers(gl);
  ...
  // Create a rotation matrix
  var radian = Math.PI * ANGLE / 180.0; // Convert to radians
  var cosB = Math.cos(radian), sinB = Math.sin(radian);

  // Note: WebGL is column major order
  var xformMatrix = new Float32Array([
     cosB, sinB, 0.0, 0.0,
    -sinB, cosB, 0.0, 0.0,
     0.0,  0.0, 1.0, 0.0,
     0.0,  0.0, 0.0, 1.0
  ]);

  // Pass the rotation matrix to the vertex shader
  var u_xformMatrix = gl.getUniformLocation(gl.program, 'u_xformMatrix');
  ...
  gl.uniformMatrix4fv(u_xformMatrix, false, xformMatrix);

  // Set the color for clearing <canvas>
  ...
  // Draw a triangle
  gl.drawArrays(gl.TRIANGLES, 0, n);
}

function initVertexBuffers(gl) {
  var vertices = new Float32Array([
    0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
  ]);
  var n = 3; // Number of vertices
  ...
  return n;
}
```

First, let us examine the vertex shader:

```
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'uniform mat4 u_xformMatrix;\n' +
6    'void main() {\n' +
7    '  gl_Position = u_xformMatrix * a_Position;\n' +
8    '}\n';
```

At line 7, `u_xformMatrix`, containing the rotation matrix described in Equation 3.11, and `a_Position`, containing the vertex coordinates (this is the right-side vector in Equation 3.11), are multiplied, literally implementing Equation 3.11.

In the sample program `TranslatedTriangle`, you were able to implement the addition of two vectors in one line (`gl_Position = a_Position + u_Translation`). In the same way, a multiplication of a matrix and vector can be written in one line in GLSL ES. This is convenient, allowing the calculation of the four equations (Equation 3.9) in one line. Again, this shows how GLSL ES has been designed specifically for 3D computer graphics by supporting powerful operations like this.

Because the transformation matrix is a 4×4 matrix and GLSL ES requires the data type for all variables, line 5 declares `u_xformMatrix` as type `mat4`. As you would expect, `mat4` is a data type specifically for holding a 4×4 matrix.

Within the main JavaScript program, the rest of the changes just calculate the rotation matrix from Equation 3.11 and then pass it to `u_xformMatrix`. This part starts from line 44:

```
43    // Create a rotation matrix
44    var radian = Math.PI * ANGLE / 180.0; // Convert to radians
45    var cosB = Math.cos(radian), sinB = Math.sin(radian);
46
47    // Note: WebGL is column major order
48    var xformMatrix = new Float32Array([
49       cosB, sinB, 0.0, 0.0,
50      -sinB, cosB, 0.0, 0.0,
51        0.0,  0.0, 1.0, 0.0,
52        0.0,  0.0, 0.0, 1.0
53    ]);
54
55    // Pass the rotation matrix to the vertex shader
     ...
61    gl.uniformMatrix4fv(u_xformMatrix, false, xformMatrix);
```

Lines 44 and 45 calculate the values of cosine and sine, which are required in the rotation matrix. Then line 48 creates the matrix `xformMatrix` using a `Float32Array`. Unlike GLSL ES, because JavaScript does not have a dedicated object for representing a matrix, you need to use the `Float32Array`. One question that arises is in which order you should store the elements of the matrix (which is arranged in rows and columns) in the elements of the array (which is arranged in a line). There are two possible orders: **row major order** and **column major order** (see Figure 3.27).

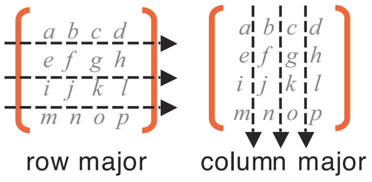


Figure 3.27 Row major order and column major order

WebGL, just like OpenGL, requires you to store the elements of a matrix in the elements of an array in column major order. So, for example, the matrix shown in Figure 3.27 is stored in an array as follows: [*a, e, i, m, b, f, j, n, c, g, k, o, d, h, l, p*]. In the sample program, the rotation matrix is stored in the `Float32Array` in this order in lines 49 to 52.

The array created is then passed to the uniform variable `u_xformMatrix` by using `gl.uniformMatrix4fv()` at line 61. Note that the last letter of this method name is `v`, which indicates that the method can pass multiple data values to the variable.

`gl.uniformMatrix4fv``(location, transpose, array)`

Assign the 4×4 matrix specified by *array* to the uniform variable specified by *location*.

Parameters	location	Specifies the storage location of the uniform variable.
	Transpose	Must be `false` in WebGL.[3]
	array	Specifies an array containing a 4×4 matrix in column major order (typed array).
Return value	None	
Errors	INVALID_OPERATION	There is no current program object.
	INVALID_VALUE	*transpose* is not `false`, or the length of *array* is less than 16.

If you load and run the sample program in your browser, you'll see the rotated triangle. Congratulations! You have successfully learned how to use a transformation matrix to rotate a triangle.

[3] This parameter specifies whether to transpose the matrix or not. The transpose operation, which exchanges the column and row elements of the matrix (see Chapter 7), is not supported by WebGL's implementation of this method and must always be set to `false`.

Reusing the Same Approach for Translation

Now, as you have seen with Equations 3.10 and 3.11, you can represent both a translation and a rotation using the same type of 4×4 matrix. Both equations use the matrices in the form `<new coordinates> = <transformation matrix> * <original coordinates>`. This is coded in the vertex shader as follows:

```
7    '  gl_Position = u_xformMatrix * a_Position;\n' +
```

This means that if you change the elements of the array `xformMatrix` from those of a rotation matrix to those of a translation matrix, you will be able to apply the translation matrix to the triangle to achieve the same result as shown earlier but which used an equation (Figure 3.18).

To do that, change line 17 in `RotatedTriangle_Matrix.js` using the translation distances from the previous example:

```
17  varTx = 0.5, Ty = 0.5, Tz = 0.0;
```

You need to rewrite the code for creating the matrix, remembering that you need to store the elements of the matrix in column major order. Let's keep the same name for the array variable, `xformMatrix`, even though it's now being used to hold a translation matrix, because it reinforces the fact that we are using essentially the same code. Finally, you are not using the variable `ANGLE`, so lines 43 to 45 are commented out:

```
43   // Create a rotation matrix
44   // var radian = Math.PI * ANGLE / 180.0; // Convert to radians
45   // var cosB = Math.cos(radian), sinB = Math.sin(radian);
46
47    // Note: WebGL is column major order
48    var xformMatrix = new Float32Array([
49       1.0, 0.0, 0.0, 0.0,
50       0.0, 1.0, 0.0, 0.0,
51       0.0, 0.0, 1.0, 0.0,
52       Tx, Ty, Tz, 1.0
53    ]);
```

Once you've made the changes, run the modified program, and you will see the same output as shown in Figure 3.18. By using a transformation matrix, you can apply various transformations using the same vertex shader. This is why the transformation matrix is such a convenient and powerful tool for 3D graphics, and it's why we've covered it in detail in this chapter.

Transformation Matrix: Scaling

Finally, let's define the transformation matrix for scaling using the same assumption that the original point is `p` and the point after scaling is `p'`.

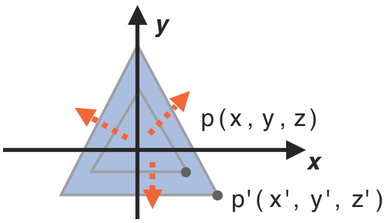


Figure 3.28 A scaling transformation

Assuming the scaling factor for the x-axis, y-axis, and z-axis is S_x, S_y, and S_z respectively, you obtain the following equations:

$$x' = S_x \times x$$

$$y' = S_y \times y$$

$$z' = S_z \times z$$

The following transformation matrix can be obtained by comparing these equations with Equation 3.9.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

As with the previous example, if you store this matrix in `xformMatrix`, you can scale the triangle by using the same vertex shader you used in `RotatedTriangle_Matrix.js`. For example, the following sample program will scale the triangle by a factor of 1.5 in a vertical direction, as shown in Figure 3.29:

```
varSx = 1.0, Sy = 1.5, Sz = 1.0;
 ...
  // Note: WebGL is column major order
  var xformMatrix = new Float32Array([
     Sx,  0.0,  0.0,  0.0,
     0.0,  Sy,  0.0,  0.0,
     0.0,  0.0,  Sz,  0.0,
     0.0,  0.0,  0.0,  1.0
  ]
```

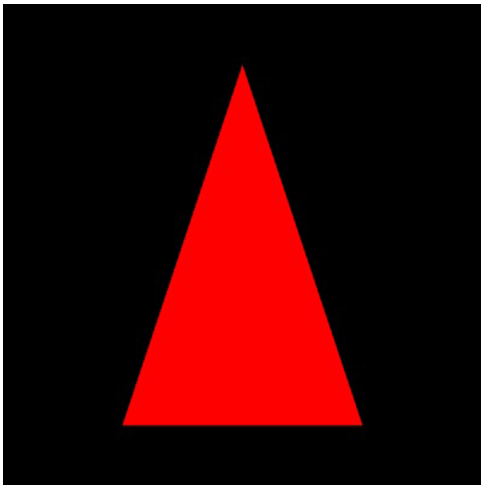


Figure 3.29 Triangle scaled in a vertical direction

Note that if you specify 0.0 to `Sx`, `Sy`, or `Sz`, the scaled size will be 0.0. If you want to keep the original size, specify 1.0 as the scaling factor.

Summary

In this chapter, you explored the process of passing multiple items of information about vertices to a vertex shader, the different types of shapes available to be drawn using that information, and the process of transforming those shapes. The shapes dealt with in this chapter changed from a point to a triangle, but the method of using shaders remained the same, as in the examples in the previous chapter. You were also introduced to matrices and learned how to use transformation matrices to apply translation, rotation, or scaling to 2D shapes. Although it's a little complicated, you should now have a good understanding of the math behind calculating the individual transformation matrices.

In the next chapter, you'll explore more complex transformations but will use a handy library to hide the details, allowing you to focus on the higher-level tasks.

This page intentionally left blank