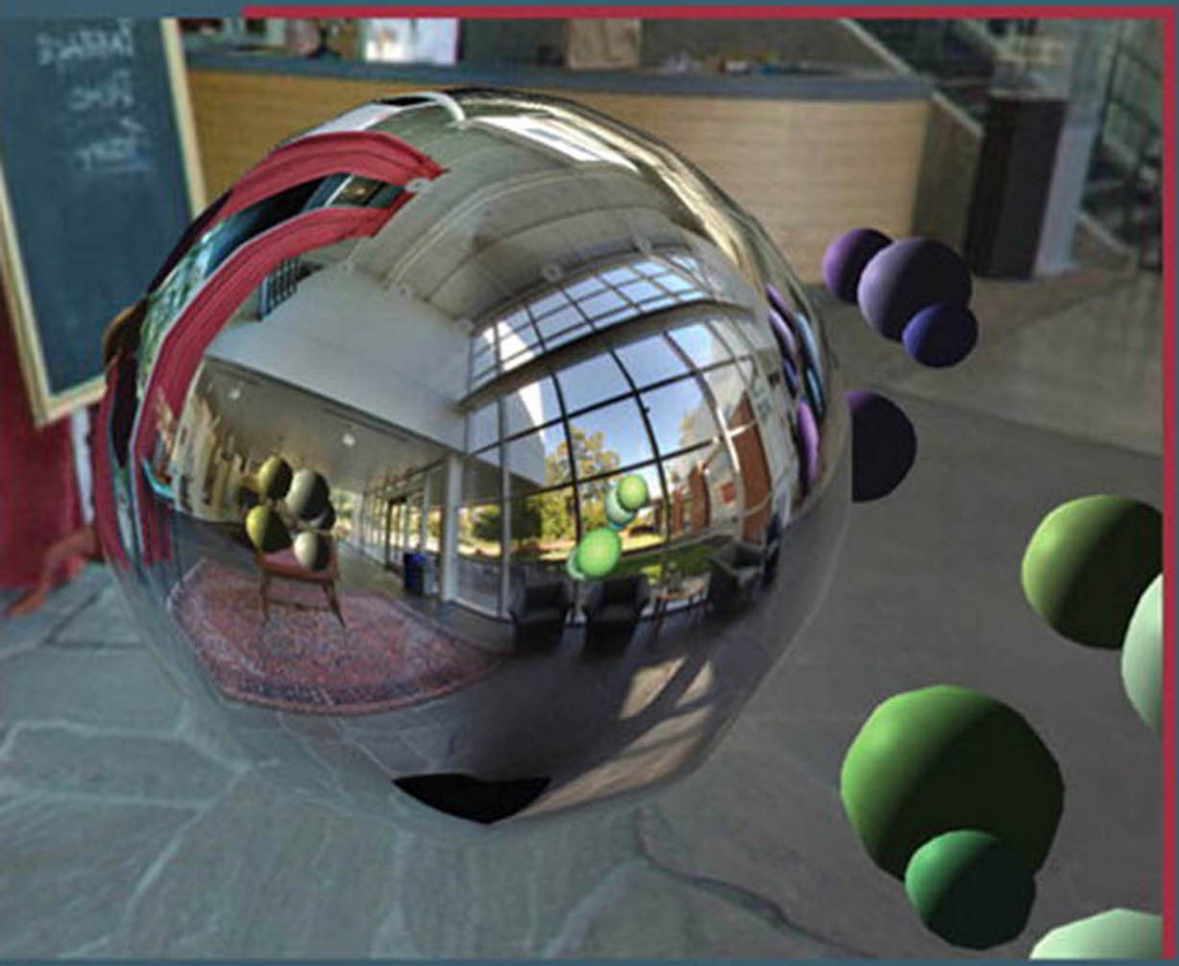
WebGL® Programming Guide

Interactive 3D Graphics Programming with WebGL



Kouichi Matsuda ▪ Rodger Lea

Praise for
WebGL Programming Guide

"WebGL provides one of the final features for creating applications that deliver 'the desktop application experience' in a web browser, and the *WebGL Programming Guide* leads the way in creating those applications. Its coverage of all aspects of using WebGL—JavaScript, OpenGL ES, and fundamental graphics techniques—delivers a thorough education on everything you need to get going. Web-based applications are the wave of the future, and this book will get you ahead of the curve!"

Dave Shreiner, Coauthor of *The OpenGL Programming Guide, Eighth Edition*; Series Editor, *OpenGL Library* (Addison Wesley)

"HTML5 is evolving the Web into a highly capable application platform supporting beautiful, engaging, and fully interactive applications that run portably across many diverse systems. WebGL is a vital part of HTML5, as it enables web programmers to access the full power and functionality of state-of-the-art 3D graphics acceleration. WebGL has been designed to run securely on any web-capable system and will unleash a new wave of developer innovation in connected 3D web-content, applications, and user interfaces. This book will enable web developers to fully understand this new wave of web functionality and leverage the exciting opportunities it creates."

Neil Trevett, Vice President Mobile Content, NVIDIA; President, The Khronos Group

"With clear explanations supported by beautiful 3D renderings, this book does wonders in transforming a complex topic into something approachable and appealing. Even without denying the sophistication of WebGL, it is an accessible resource that beginners should consider picking up before anything else."

Evan Burchard, Author, *Web Game Developer's Cookbook* (Addison Wesley)

"Both authors have a strong OpenGL background and transfer this knowledge nicely over to WebGL, resulting in an excellent guide for beginners as well as advanced readers."

Daniel Haehn, Research Software Developer, Boston Children's Hospital

"*WebGL Programming Guide* provides a straightforward and easy-to-follow look at the mechanics of building 3D applications for the Web without relying on bulky libraries or wrappers. A great resource for developers seeking an introduction to 3D development concepts mixed with cutting-edge web technology."

Brandon Jones, Software Engineer, Google

Chapter 5

Using Colors and Texture Images

The previous chapters explained the key concepts underlying the foundations of WebGL through the use of examples based on 2D shapes. This approach has given you a good understanding of how to deal with single color geometric shapes in WebGL. Building on these basics, you now delve a little further into WebGL by exploring the following three subjects:

- Passing other data such as color information to the vertex shader
- The conversion from a shape to fragments that takes place between the vertex shader and the fragment shader, which is known as the **rasterization process**
- Mapping images (or textures) onto the surfaces of a shape or object

This is the final chapter that focuses on the key functionalities of WebGL. After reading this chapter, you will understand the techniques and mechanism for using colors and textures in WebGL and will have mastered enough WebGL to allow you to create sophisticated 3D scenes.

Passing Other Types of Information to Vertex Shaders

In the previous sample programs, a single buffer object was created first, the vertex coordinates were stored in it, and then it was passed to the vertex shader. However, beside coordinates, vertices involved in 3D graphics often need other types of information such as color information or point size. For example, let us take a look at a program you used in Chapter 3, "Drawing and Transforming Triangles," which draws three points: `MultiPoint.js`. In the shader, in addition to the vertex coordinates, you provided the point size as extra information. However, the point size was a fixed value and set in the shader rather than passed from outside:

```
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'
```

Line 6 assigns the vertex coordinates to `gl_Position`, and line 7 assigns a fixed point size of 10.0 to `gl_PointSize`. If you now wanted to modify the size of that point from your JavaScript program, you would need a way to pass the point size with the vertex coordinates.

Let's look at an example, `MultiAttributeSize`, whose goal is to draw three points of different sizes: 10.0, 20.0, and 30.0, respectively (see Figure 5.1).

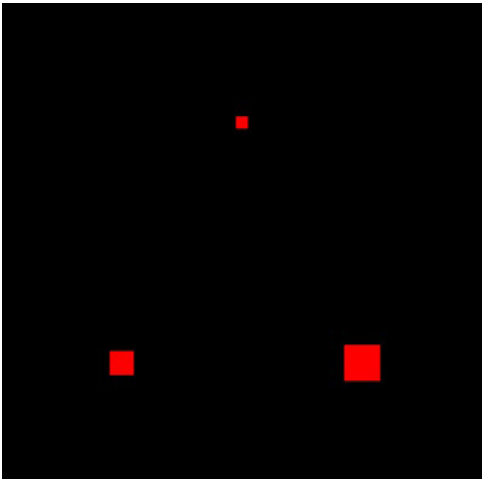


Figure 5.1 MultiAttributeSize

In the previous chapter, you carried out the following steps to pass the vertex coordinates:

1. Create a buffer object.
2. Bind the buffer object to the target.
3. Write the coordinate data into the buffer object.
4. Assign the buffer object to the attribute variable.
5. Enable the assignment.

If you now wanted to pass several items of vertex information to the vertex shader through buffer objects, you could just apply the same steps to all the items of information associated with a vertex. Let's look at a sample program that uses multiple buffers to do just that.

Sample Program (MultiAttributeSize.js)

`MultiAttributeSize.js` is shown in Listing 5.1. The fragment shader is basically the same as in `MultiPoint.js`, so let's omit it this time. The vertex shader is also similar, apart from the fact that you add a new attribute variable that specifies the point size. The numbers 1 through 5 on the right of the listing note the five steps previously outlined.

Listing 5.1 MultiAttributeSize.js

```
 1 // MultiAttributeSize.js
 2 // Vertex shader program
 3 var VSHADER_SOURCE =
 4   'attribute vec4 a_Position;\n' +
 5   'attribute float a_PointSize;\n' +
 6   'void main() {\n' +
 7   '  gl_Position = a_Position;\n' +
 8   '  gl_PointSize = a_PointSize;\n' +
 9   '}\n';
   ...
17 function main() {
   ...
34   // Set the vertex information
35   var n = initVertexBuffers(gl);
   ...
47   // Draw three points
48   gl.drawArrays(gl.POINTS, 0, n);
49 }
50
51 function initVertexBuffers(gl) {
52   var vertices = new Float32Array([
53     0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
54   ]);
55   var n = 3;
56
57   var sizes = new Float32Array([
58     10.0, 20.0, 30.0  // Point sizes
59 ]);
60
61   // Create a buffer object
62   var vertexBuffer = gl.createBuffer();                          <-(1)
63   var sizeBuffer = gl.createBuffer();                            <-(1')
   ...
69   // Write vertex coordinates to the buffer object and enable it
70   gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);                  <-(2)
71   gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);      <-(3)
72   var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
```

```
    ...
77   gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);       <-(4)
78   gl.enableVertexAttribArray(a_Position);                             <-(5)
79
80   // Write point sizes to the buffer object and enable it
81   gl.bindBuffer(gl.ARRAY_BUFFER, sizeBuffer);                         <-(2')
82   gl.bufferData(gl.ARRAY_BUFFER, sizes, gl.STATIC_DRAW);              <-(3')
83   var a_PointSize = gl.getAttribLocation(gl.program, 'a_PointSize');
    ...
88   gl.vertexAttribPointer(a_PointSize, 1, gl.FLOAT, false, 0, 0);      <-(4')
89   gl.enableVertexAttribArray(a_PointSize);                            <-(5')
    ...
94   return n;
95 }
```

First of all, let us examine the vertex shader in Listing 5.1. As you can see, the attribute variable `a_PointSize`, which receives the point size from the JavaScript program, has been added. This variable, declared at line 5 as a `float`, is then assigned to `gl_PointSize` at line 8. No other changes are necessary for the vertex shader, but you will need a slight modification to the process in `initVertexBuffers()` so it can handle several buffer objects. Let us take a more detailed look at it.

Create Multiple Buffer Objects

The function `initVertexBuffers()` starts at line 51, and the vertex coordinates are defined from lines 52 to 54. The point sizes are then specified at line 57 using the array `sizes`:

```
57   var sizes = new Float32Array([
58     10.0, 20.0, 30.0  // Point sizes
59   ]);
```

A buffer object is created at line 62 for the vertex data, and at line 63 another buffer object (`sizeBuffer`) is created for storing the array of "point sizes."

From lines 70 to 78, the program binds the buffer object for the vertex coordinates, writes the data, and finally assigns and enables the attribute variables associated with the buffer object. These tasks are the same as those described in the previous sample programs.

Lines 80 to 89 are new additions for handling the different point sizes. However, the steps are the same as for a vertex buffer. Bind the buffer object for the point sizes (`sizeBuffer`) to the target (line 81), write the data (line 82), assign the buffer object to the attribute variable `a_PointSize` (line 88), and enable it.

Once these steps in `initVertexBuffers()` are completed, the internal state of the WebGL system looks like Figure 5.2. You can see that the two separate buffer objects are created and then assigned to the two separate attribute variables.

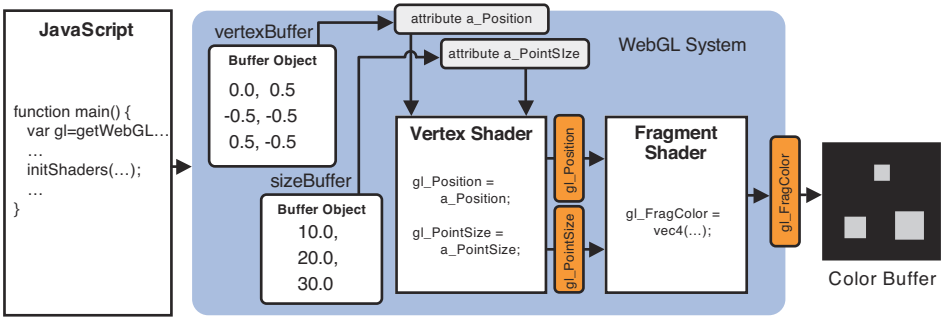


Figure 5.2 Using two buffer objects to pass data to a vertex shader

In this situation, when `gl.drawArrays()` at line 48 is executed, all the data stored inside the buffer objects is sequentially passed to each attribute variable in the order it was stored inside the buffer objects. By assigning this data to `gl_Position` at line 7 and `gl_PointSize` at line 8, respectively (the vertex shader's program in Figure 5.2), you are now able to draw different size objects located at different positions.

By creating a buffer object for each type of data in this way and then allocating it to the attribute variables, you can pass several pieces of information about each vertex to the vertex shader. Other types of information that can be passed include color, texture coordinates (described in this chapter), and normals (see Chapter 7), as well as point size.

The gl.vertexAttribPointer() Stride and Offset Parameters

Although multiple buffer objects are a great way to handle small amounts of data, in a complicated 3D object with many thousands of vertices, you can imagine that managing all the associated vertex data is an extremely difficult task. For example, imagine needing to manually check each of these arrays when the total count of `MultiAttributeSize.js`'s vertices and sizes reaches 1000.[1] However, WebGL allows the vertex coordinates and the size to be bundled into a single component and provides mechanisms to access the different data types. For example, you can group the vertex and size data in the following way (refer to Listing 5.2), often referred to as **interleaving**.

Listing 5.2 An Array Containing Multiple Items of Vertex Information

```
var verticesSizes = new Float32Array([
  // Vertex coordinates and size of a point
    0.0,  0.5, 10.0,   // The 1st point
   -0.5, -0.5, 20.0,  // The 2nd point
    0.5, -0.5, 30.0    // The 3rd point
]);
```

[1] In practice, because modeling tools that create 3D models actually generate this data, there is no necessity to either manually input them or visually check their consistency. The use of modeling tools and the data they generate will be discussed in Chapter 10.

As just described, once you have stored several types of information pertaining to the vertex in a single buffer object, you need a mechanism to access these different data elements. You can use the fifth (*stride*) and sixth (*offset*) arguments of `gl.vertexAttribPointer()` to do this, as shown in the example that follows.

Sample Program (MultiAttributeSize_Interleaved.js)

Let's construct a sample program, `MultiAttributeSize_Interleaved`, which passes multiple data to the vertex shader, just like `MultiAttributeSize.js` (refer to Listing 5.1), except that it bundles the data into a single array or buffer. Listing 5.3 shows the program in which the vertex shader and the fragment shader are the same as in `MultiAttributeSize.js`.

Listing 5.3 MultiAttributeSize_Interleaved.js

```
 1 // MultiAttributeSize_Interleaved.js
 2 // Vertex shader program
 3 var VSHADER_SOURCE =
 4   'attribute vec4 a_Position;\n' +
 5   'attribute float a_PointSize;\n' +
 6   'void main() {\n' +
 7   '  gl_Position = a_Position;\n' +
 8   '  gl_PointSize = a_PointSize;\n' +
 9   '}\n';
   ...
17 function main() {
   ...
34   // Set vertex coordinates and point sizes
35   var n = initVertexBuffers(gl);
   ...
48   gl.drawArrays(gl.POINTS, 0, n);
49 }
50
51 function initVertexBuffers(gl) {
52   var verticesSizes = new Float32Array([
53     // Vertex coordinates and size of a point
54      0.0,  0.5,  10.0,  // The 1st vertex
55     -0.5, -0.5,  20.0,  // The 2nd vertex
56      0.5, -0.5,  30.0   // The 3rd vertex
57   ]);
58   var n = 3;
59
60   // Create a buffer object
61   var vertexSizeBuffer = gl.createBuffer();
   ...
```