Chapter 2

Your First Step with WebGL

As explained in Chapter 1, "Overview of WebGL," WebGL applications use both HTML and JavaScript to create and draw 3D graphics on the screen. To do this, WebGL utilizes the new `<canvas>` element, introduced in HTML5, which defines a drawing area on a web page. Without WebGL, the `<canvas>` element only allows you to draw two-dimensional graphics using JavaScript. With WebGL, you can use the same element for drawing three-dimensional graphics.

This chapter explains the `<canvas>` element and the core functions of WebGL by taking you, step-by-step, through the construction of several example programs. Each example is written in JavaScript and uses WebGL to display and interact with a simple shape on a web page. Because of this, these JavaScript programs are referred to as **WebGL applications**.

The example WebGL applications will highlight some key points, including:

- How WebGL uses the `<canvas>` element and how to draw on it
- The linkage between HTML and WebGL using JavaScript
- Simple WebGL drawing functions
- The role of shader programs within WebGL

By the end of this chapter, you will understand how to write and execute basic WebGL applications and how to draw simple 2D shapes. You will use this knowledge to explore further the basics of WebGL in Chapters 3, "Drawing and Transforming Triangles," 4, "More Transformations and Basic Animation," and 5, "Using Colors and Texture Images."

What Is a Canvas?

Before HTML5, if you wanted to display an image in a web page, the only native HTML approach was to use the `<img>` tag. This tag, although a convenient tool, is restricted to still images and

doesn't allow you to dynamically draw and display the image on the fly. This is one of the reasons that non-native solutions such as Flash Player have been used.

However, HTML5, by introducing the `<canvas>` tag, has changed all that, offering a convenient way to draw computer graphics dynamically using JavaScript.

In a similar manner to the way artists use paint canvases, the `<canvas>` tag defines a drawing area on a web page. Then, rather than using brush and paints, you can use JavaScript to draw anything you want in the area. You can draw points, lines, rectangles, circles, and so on by using JavaScript methods provided for `<canvas>`. Figure 2.1 shows an example of a drawing tool that uses the `<canvas>` tag.

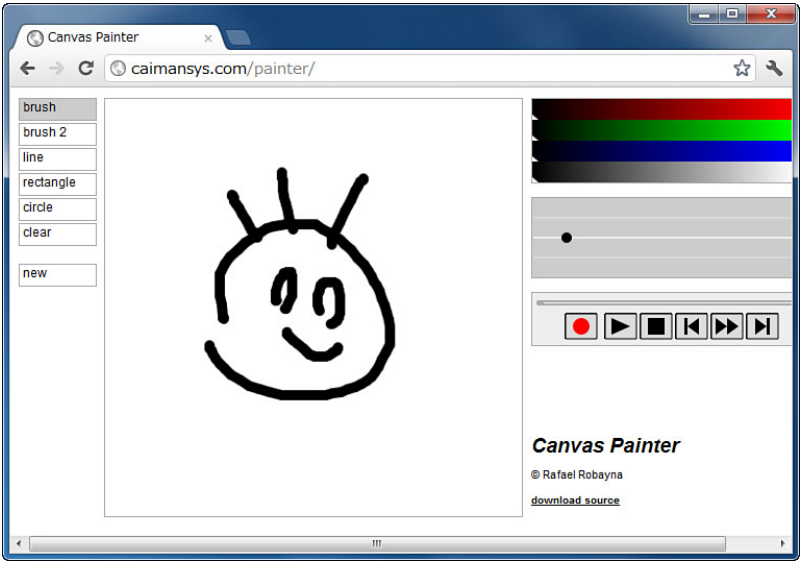

Figure 2.1 A drawing tool using the <canvas> element (http://caimansys.com/painter/)

This drawing tool runs within a web page and allows you to interactively draw lines, rectangles, and circles and even change their colors.

Although you won't be creating anything as sophisticated just yet, let's look at the core functions of `<canvas>` by using a sample program, `DrawRectangle`, which draws a filled blue rectangle on a web page. Figure 2.2 shows `DrawRectangle` when it's loaded into a browser.

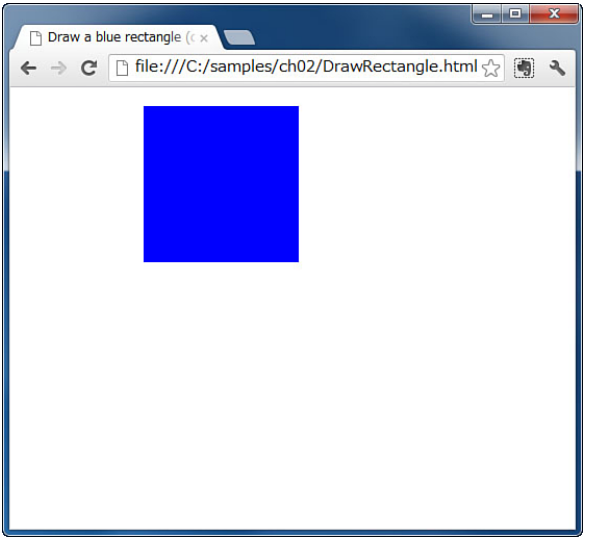

Figure 2.2 DrawRectangle

Using the <canvas> Tag

Let's look at how `DrawRectangle` works and explain how the `<canvas>` tag is used in the HTML file. Listing 2.1 shows `DrawingTriangle.html`. Note that all HTML files in this book are written in HTML5.

Listing 2.1 DrawRectangle.html

```
 1  <!DOCTYPE html>
 2  <html lang="en">
 3    <head>
 4      <meta charset="utf-8" />
 5      <title>Draw a blue rectangle (canvas version)</title>
 6    </head>
 7
 8    <body onload="main()">
 9     <canvas id="example" width="400" height="400">
10     Please use a browser that supports "canvas"
11     </canvas>
12     <script src="DrawRectangle.js"></script>
13    </body>
14  </html>
```

The <canvas> tag is defined at line 9. This defines the drawing area as a 400 × 400 pixel region on a web page using the width and height attributes in the tag. The canvas is given an identifier using the id attribute, which will be used later:

```
<canvas id="example" width="400" height="400"></canvas>
```

By default, the canvas is invisible (actually transparent) until you draw something into it, which we'll do with JavaScript in a moment. That's all you need to do in the HTML file to prepare a <canvas> that the WebGL program can use. However, one thing to note is that this line only works in a <canvas>-enabled browser. However, browsers that don't support the <canvas> tag will ignore this line, and nothing will be displayed on the screen. To handle this, you can display an error message by adding the message into the tag as follows:

```
 9     <canvas id="example" width="400" height="400">
10     Please use a browser that supports "canvas"
11     </canvas>
```

To draw into the canvas, you need some associated JavaScript code that performs the drawing operations. You can include that JavaScript code in the HTML or write it as a separate JavaScript file. In our examples, we use the second approach because it makes the code easier to read. Whichever approach you take, you need to tell the browser where the JavaScript code starts. Line 8 does that by telling the browser that when it loads the separate JavaScript code it should use the function main() as the entry point for the JavaScript program. This is specified for the <body> element using its onload attribute that tells the browser to execute the JavaScript function main() after it loads the <body> element:

```
8    <body onload="main()">
```

Line 12 tells the browser to import the JavaScript file DrawRectangle.js in which the function main() is defined:

```
12     <script src="DrawRectangle.js"></script>
```

For clarity, all sample programs in this book use the same filename for both the HTML file and the associated JavaScript file, which is imported in the HTML file (see Figure 2.3).

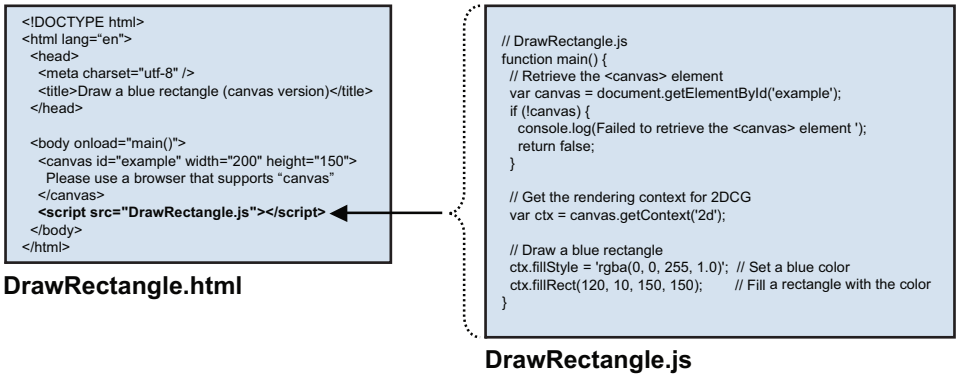

Figure 2.3 DrawRectangle.html and DrawRectangle.js

DrawRectangle.js

`DrawRectangle.js` is a JavaScript program that draws a blue rectangle on the drawing area defined by the `<canvas>` element (see Listing 2.2). It has only 16 lines, which consist of the three steps required to draw two-dimensional computer graphics (2D graphics) on the canvas:

1. Retrieve the `<canvas>` element.
2. Request the rendering "context" for the 2D graphics from the element.
3. Draw the 2D graphics using the methods that the context supports.

These three steps are the same whether you are drawing a 2D or a 3D graphic; here, you are drawing a simple 2D rectangle using standard JavaScript. If you were drawing a 3D graphic using WebGL, then the rendering context in step (2) at line 11 would be for a 3D rendering context; however, the high-level process would be the same.

Listing 2.2 DrawRectangle.js

```
 1  // DrawRectangle.js
 2  function main() {
 3    // Retrieve <canvas> element                                  <- (1)
 4    var canvas = document.getElementById('example');
 5    if (!canvas) {
 6      console.log('Failed to retrieve the <canvas> element');
 7      return;
 8    }
 9
10   // Get the rendering context for 2DCG                          <- (2)
11    var ctx = canvas.getContext('2d');
```

```
12
13   // Draw a blue rectangle                                          <- (3)
14   ctx.fillStyle = 'rgba(0, 0, 255, 1.0)';  // Set a blue color
15   ctx.fillRect(120, 10, 150, 150); // Fill a rectangle with the color
16  }
```

Let us look at each step in order.

Retrieve the <canvas> Element

To draw something on a `<canvas>`, you must first retrieve the `<canvas>` element from the HTML file in the JavaScript program. You can get the element by using the method `document.getElementById()`, as shown at line 4. This method has a single parameter, which is the string specified in the attribute `id` in the `<canvas>` tag in our HTML file. In this case, the string is `'example'` and it was defined back in `DrawRectangle.html` at line 9 (refer to Listing 2.1).

If the return value of this method is not `null`, you have successfully retrieved the element. However, if it is `null`, you have failed to retrieve the element. You can check for this condition using a simple `if` statement like that shown at line 5. In case of error, line 6 is executed. It uses the method `console.log()` to display the parameter as a string in the browser's console.

Note In Chrome, you can show the console by going to Tools, JavaScript Console or pressing Ctrl+Shift+J (see Figure 2.4); in Firefox, you can show it by going to Tools, Web Developer, Web Console or pressing Ctrl+Shift+K.

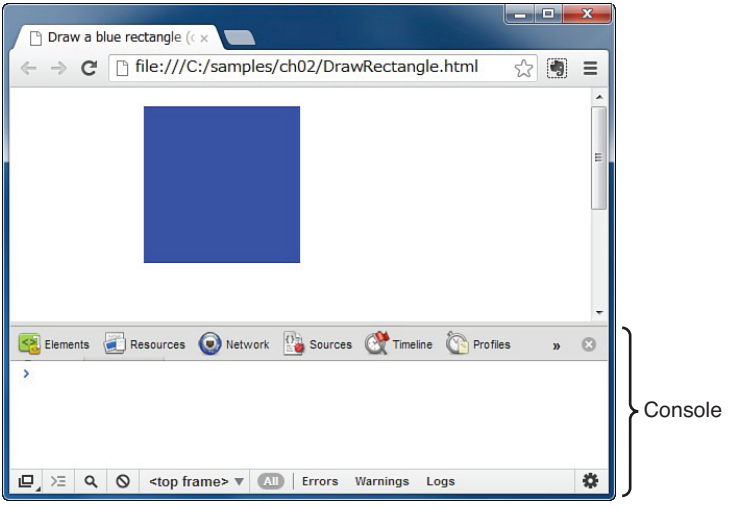

Figure 2.4 Console in Chrome

Get the Rendering Context for the 2D Graphics by Using the Element

Because the `<canvas>` is designed to be flexible and supports both 2D and 3D, it does not provide drawing methods directly and instead provides a mechanism called a **context**, which supports the actual drawing features. Line 11 gets that context:

```
11    var ctx = canvas.getContext('2d');
```

The method `canvas.getContext()` has a parameter that specifies which type of drawing features you want to use. In this example you want to draw a 2D shape, so you must specify 2d (case sensitive).

The result of this call, the context, is stored in the variable `ctx` ready for use. Note, for brevity we haven't checked error conditions, which is something you should always do in your own programs.

Draw the 2D Graphics Using the Methods Supported by the Context

Now that we have a drawing context, let's look at the code for drawing a blue rectangle, which is a two-step process. First, set the color to be used when drawing. Second, draw (or fill) a rectangle with the color.

Lines 14 and 15 handle these steps:

```
13   // Draw a blue rectangle                          <- (3)
14   ctx.fillStyle = 'rgba(0, 0, 255, 1.0)';  // Set color to blue
15   ctx.fillRect(120, 10, 150, 150); // Fill a rectangle with the color
```

The `rgba` in the string `rgba(0, 0, 255, 1.0)` on line 14 indicate r (red), g (green), b (blue), and a (alpha: transparency), with each RGB parameter taking a value from 0 (minimum value) to 255 (maximum value) and the alpha parameter from 0.0 (transparent) to 1.0 (opaque). In general, computer systems represent a color by using a combination of red, green, and blue (light's three primary colors), which is referred to as **RGB format**. When alpha (transparency) is added, the format is called **RGBA format**.

Line 15 then uses the `fillStyle` property to specify the fill color when drawing the rectangle. However, before going into the details of the arguments on line 15, let's look at the coordinate system used by the `<canvas>` element (see Figure 2.5).

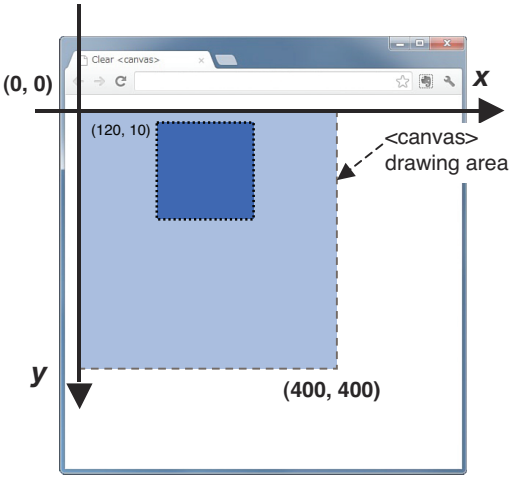

Figure 2.5 The coordinate system of <canvas>

As you can see in the figure, the coordinate system of the `<canvas>` element has the horizontal direction as the x-axis (right-direction is positive) and the vertical direction as the y-axis (down-direction is positive). Note that the origin is located at the upper-left corner and the down direction of the y-axis is positive. The rectangle drawn with a dashed line is the original `<canvas>` element in our HTML file (refer to Listing 2.1), which we specified as being 400 by 400 pixels. The dotted line is the rectangle that the sample program draws.

When we use `ctx.fillRect()` to draw a rectangle, the first and second parameters of this method are the position of the upper-left corner of the rectangle within the `<canvas>`, and the third and fourth parameters are the width and height of the rectangle (in pixels):

```
15   ctx.fillRects(120, 10, 150, 150);// Fill a rectangle with the color
```

After loading `DrawRectangle.html` into your browser, you will see the rectangle that was shown in Figure 2.2.

So far, we've only looked at 2D graphics. However, WebGL also utilizes the same `<canvas>` element to draw 3D graphics on a web page, so let's now enter into the WebGL world.

The World's Shortest WebGL Program: Clear Drawing Area

Let's start by constructing the world's shortest WebGL program, `HelloCanvas`, which simply clears the drawing area specified by a `<canvas>` tag. Figure 2.6 shows the result of loading the program, which clears (by filling with black) the rectangular area defined by a `<canvas>`.

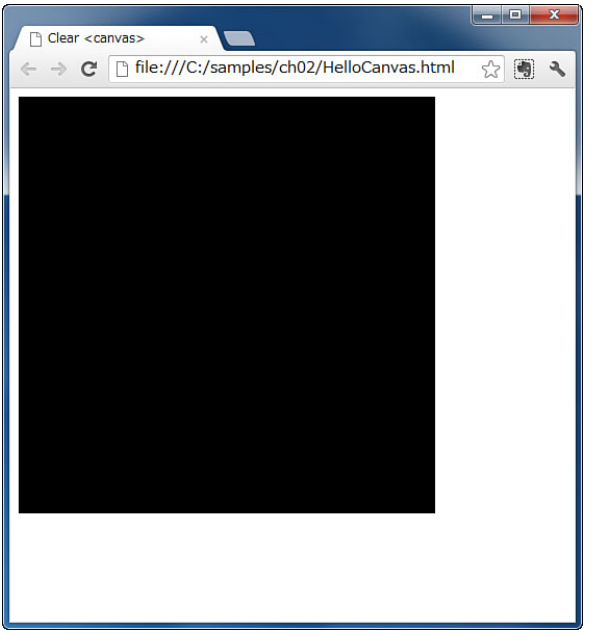

Figure 2.6 HelloCanvas

The HTML File (HelloCanvas.html)

Take a look at `HelloCanvas.html`, as shown in Figure 2.7). Its structure is simple and starts by defining the drawing area using the `<canvas>` element at line 9 and then importing `HelloCanvas.js` (the WebGL program) at line 16.

Lines 13 to 15 import several other JavaScript files, which provide useful convenience functions that help WebGL programming. These will be explained in more detail later. For now, just think of them as libraries.

```
 1 <!DOCTYPE html>
 2 <html lang=”en”>
 3  <head>
 4    <meta charset="utf-8" />
 5    <title>Clear canvas</title>
 6  </head>
 7
 8  <body onload="main()">
 9    <canvas id="webgl" width="400" height="400">
10     Please use the browser supporting "canvas"
11    </canvas>
12
13    <script src="../lib/webgl-utils.js"></script>
14    <script src="../lib/webgl-debug.js"></script>
15    <script src="../lib/cuon-utils.js"></script>
16    <script src="HelloCanvas.js"></script>
17  </body>
18 </html>
```

<canvas> into which WebGL draws shapes

JavaScript files containing convenient functions for WebGL

JavaScript file drawing shapes into the <canvas>

Figure 2.7 HelloCanvas.html

You've set up the canvas (line 9) and then imported the `HelloCanvas` JavaScript file (line 16), which actually uses WebGL commands to access the canvas and draw your first 3D program. Let us look at the WebGL program defined in `HelloCanvas.js`.

JavaScript Program (HelloCanvas.js)

`HelloCanvas.js` (see Listing 2.3) has only 18 lines, including comments and error handling, and follows the same steps as explained for 2D graphics: retrieve the `<canvas>` element, get its rendering context, and then begin drawing.

Listing 2.3 HelloCanvas.js

```
 1 // HelloCanvas.js
 2 function main() {
 3   // Retrieve <canvas> element
 4   var canvas = document.getElementById('webgl');
 5
 6   // Get the rendering context for WebGL
 7   var gl = getWebGLContext(canvas);
 8   if (!gl) {
 9     console.log('Failed to get the rendering context for WebGL');
10     return;
11   }
```

```
12
13   // Specify the color for clearing <canvas>
14   gl.clearColor(0.0, 0.0, 0.0, 1.0);
15
16   // Clear <canvas>
17   gl.clear(gl.COLOR_BUFFER_BIT);
18 }
```

As in the previous example, there is only one function, `main()`, which is the link between the HTML and the JavaScript and set at `<body>` element using its `onload` attribute (line 8) in `HelloCanvas.html` (refer to Figure 2.7).

Figure 2.8 shows the processing flow of the `main()` function of our WebGL program and consists of four steps, which are discussed individually next.

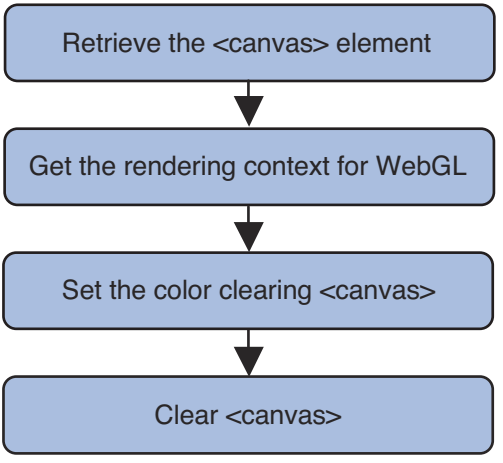

Figure 2.8 The processing flow of the main() function

Retrieve the <canvas> Element

First, `main()` retrieves the `<canvas>` element from the HTML file. As explained in `DrawRectangle.js`, it uses the `document.getElementById()` method specifying `webgl` as the argument. Looking back at `HelloCanvas.html` (refer to Figure 2.7), you can see that attribute `id` is set at the `<canvas>` tag at line 9:

```
9    <canvas id="webgl" width="400" height="400">
```

The return value of this method is stored in the `canvas` variable.

Get the Rendering Context for WebGL

In the next step, the program uses the variable `canvas` to get the rendering context for WebGL. Normally, we would use `canvas.getContext()` as described earlier to get the rendering context for WebGL. However, because the argument specified in `canvas.getContext()` varies between browsers,[1] we have written a special function `getWebGLContext()` to hide the differences between the browsers:

```
7        var gl = getWebGLContext(canvas);
```

This is one of the convenience functions mentioned earlier that was written specially for this book and is defined in `cuon-utils.js`, which is imported at line 15 in `HelloCanvas.html`. The functions defined in the file become available by specifying the path to the file in the attribute `src` in the `<script>` tag and loading the file. The following is the specification of `getWebGLContext()`.

`getWebGLContext(element [, debug])`

Get the rendering context for WebGL, set the debug setting for WebGL, and display any error message in the browser console in case of error.

Parameters	element	Specifies `<canvas>` element to be queried.
	debug (optional)	Default is `true`. When set to `true`, JavaScript errors are displayed in the console. Note: Turn off after debugging; otherwise, performance is affected.
Return value	non-null	The rendering context for WebGL.
	null	WebGL is not available.

The processing flow to retrieve the `<canvas>` element and use the element to get the rendering context is the same as in `DrawRectangle.js` shown earlier, where the rendering context was used to draw 2D graphics on the `<canvas>`.

In a similar way, WebGL uses the rendering context returned by `getWebGLContext()` to draw on the `<canvas>`. However, now the context is for 3D rather than 2D, so 3D (that is, WebGL) methods are available. The program stores the context in the variable `gl` at line 7. You can use any name for the variable. We have intentionally used `gl` throughout this book, because it aligns the names of the WebGL-related methods to that of OpenGL ES 2.0, which is the base specification of WebGL. For example, `gl.clearColor()` at line 14 corresponds to `glClearColor()` in OpenGL ES 2.0 or OpenGL:

```
14   gl.clearColor(0.0, 0.0, 0.0, 1.0);
```

[1] Although most browsers are settling on "experimental-webgl" for this argument, not all do. Additionally, over time, this will evolve to plain 'webgl,' so we have chosen to hide this.

This book explains all WebGL-related methods assuming that the rendering context is held in the variable `gl`.

Once you have the rendering context for WebGL, the next step is to use the context to set the color for clearing the drawing area specified by the `<canvas>`.

Set the Color for Clearing the <canvas>

In the previous section, `DrawRectangle.js` set the drawing color before drawing the rectangle. In a similar way, with WebGL you need to set the color before actually clearing the drawing area. Line 14 uses `gl.clearColor()` to set the color in RGBA format.

`gl.clearColor``(red, green, blue, alpha)`

Specify the clear color for a drawing area:

Parameters	red	Specifies the red value (from 0.0 to 1.0).
	green	Specifies the green value (from 0.0 to 1.0).
	blue	Specifies the blue value (from 0.0 to 1.0).
	alpha	Specifies an alpha (transparency) value (from 0.0 to 1.0). 0.0 means transparent and 1.0 means opaque.
	If any of the values of these parameters is less than 0.0 or more than 1.0, it is truncated into 0.0 or 1.0, respectively.	
Return value	None	
Errors[2]	None	

The sample program calls `gl.clearColor`(0.0, 0.0, 0.0, 1.0) at line 14, so black is specified as the clear color. The followings are examples that specify other colors:

```
(1.0, 0.0, 0.0, 1.0)     red
(0.0, 1.0, 0.0, 1.0)     green
(0.0, 0.0, 1.0, 1.0)     blue
(1.0, 1.0, 0.0, 1.0)     yellow
(1.0, 0.0, 1.0, 1.0)     purple
(0.0, 1.0, 1.0, 1.0)     light blue
(1.0, 1.0, 1.0, 1.0)     white
```

[2] In this book, the item "errors" is shown for all specifications of WebGL-related methods. This indicates errors that cannot be represented by the return value of the method when the method will result in as error. By default, the errors are not displayed, but they can be displayed in a JavaScript console by specifying `true` as the second argument of `getWebGLContext()`.

You might have noticed that in our 2D programming example in this chapter, `DrawRectangle`, each value for color is specified from 0 to 255. However, because WebGL is based on OpenGL, it uses the traditional OpenGL values from 0.0 to 1.0. The higher the value is, the more intense the color becomes. Similarly, for the alpha parameter (fourth parameter), the higher the value, the less transparent the color.

Once you specify the clear color, the color is retained in the **WebGL system** and not changed until another color is specified by a call to `gl.clearColor()`. This means you don't need to specify the clear color again if at some point in the future you want to clear the area again using the same color.

Clear <canvas>

Finally, you can use `gl.clear()` to clear the drawing area with the specified clear color:

```
17    gl.clear(gl.COLOR_BUFFER_BIT);
```

Note that the argument of this method is `gl.COLOR_BUFFER_BIT`, not, as you might expect, the `<canvas>` element that defines the drawing area to be cleared. This is because the WebGL method `gl.clear()` is actually relying on OpenGL, which uses a more sophisticated model than simple canvases, instead using multiple underlying buffers. One such buffer, the color buffer, is used in this example. By using `gl.COLOR_BUFFER_BIT`, you are telling WebGL to use the color buffer when clearing the canvas. WebGL uses a number of buffers in addition to the color buffer, including a depth buffer and a stencil buffer. The color buffer will be covered in detail later in this chapter, and you'll see the depth buffer in action in Chapter 7, "Toward the 3D World." The stencil buffer will not be covered in this book because it is seldom used.

Clearing the color buffer will actually cause WebGL to clear the `<canvas>` area on the web page.

`gl.clear(buffer)`

Clear the specified buffer to preset values. In the case of a color buffer, the value (color) specified by `gl.clearColor()` is used.

Parameters buffer — Specifies the buffer to be cleared. Bitwise `OR` (`|`) operators are used to specify multiple buffers.

`gl.COLOR_BUFFER_BIT`	Specifies the color buffer.
`gl.DEPTH_BUFFER_BIT`	Specifies the depth buffer.
`gl.STENCIL_BUFFER_BIT`	Specifies the stencil buffer.

Return value None

Errors INVALID_VALUE — *buffer* is none of the preceding three values.

If no color has been specified (that is, you haven't made a call to `gl.clearColor()`), then the following default value is used (see Table 2.1).

Table 2.1 Default Values to Clear Each Buffer and Associated Methods

Buffer Name	Default Value	Setting Method
Color buffer	(0.0, 0.0, 0.0, 0.0)	`gl.clearColor(red, green, blue, alpha)`
Depth buffer	1.0	`gl.clearDepth(depth)`
Stencil buffer	0	`gl.clearStencil(s)`

Now that you've read through and understand this simple WebGL example, you should load `HelloCanvas` into your browser to check that the drawing area is cleared to black. Remember, you can run all the examples in the book directly from the companion website. However, if you want to experiment with any, you need to download the examples from the book's website to a location on your local disk. If you've done that, to load the example, navigate to that location on your disk and load `HelloCanvas.html` into your browser.

Experimenting with the Sample Program

Let's experiment a little with the sample program to become familiar with the way you specify colors in WebGL by trying some other colors for the clear operation. Using your favorite editor, rewrite Line 14 of `HelloCanvas.js` as follows and save your modification back to the original file:

```
14   gl.clearColor(0.0, 0.0, 1.0, 1.0);
```

After reloading `HelloCanvas.html` into your browser, `HelloCanvas.js` is also reloaded, and then `main()` is executed to clear the drawing area to blue. Try to use other colors and check the result. For example, `gl.clearColor(0.5, 0.5, 0.5, 1.0)` clears the area to gray.

Draw a Point (Version 1)

In the previous section, you saw how to initialize WebGL and use some simple WebGL-related methods. In this section, you are going to go one step further and construct a sample program to draw the simplest shape of all: a point. The program will draw a red point using 10 pixels at (0.0, 0.0, 0.0). Because WebGL deals with three-dimensional graphics, three coordinates are necessary to specify the position of the point. You'll be introduced to coordinates later, but for now simply accept that a point drawn at (0.0, 0.0, 0.0) is displayed at the center of the `<canvas>` area.

The sample program name is `HelloPoint1` and, as shown in Figure 2.9, it draws a red point (rectangle) at the center of the `<canvas>`, which has been cleared to black.[3] You will actually be using a filled rectangle as a point instead of a filled circle because a rectangle can be drawn faster than a circle. (We will deal with how to draw a rounded point in Chapter 9, "Hierarchical Objects.")

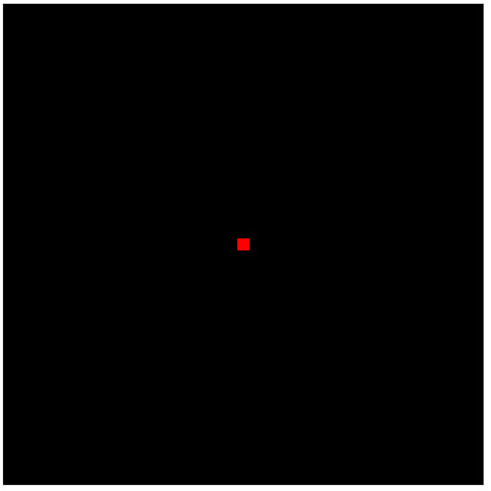

Figure 2.9 HelloPoint1

Just like clearing the color in the previous section, the color of a point must be specified in RGBA. For red, the value of R is 1.0, G is 0.0, B is 0.0, and A is 1.0. You will remember that `DrawRectangle.js` earlier in the chapter specifies the drawing color and then draws a rectangle as follows:

```
ctx.fillStyle='rgba(0, 0, 255, 1.0)';
ctx.fillRect(120, 10, 150, 150);
```

So you are probably thinking that WebGL would do something similar, perhaps something like this:

```
gl.drawColor(1.0, 0.0, 0.0, 1.0);
gl.drawPoint(0, 0, 0, 10); // The position of center and the size of point
```

[3] The sample programs in Chapter 2 are written in the simplest way possible so the reader can focus on understanding the functionality of shaders. In particular, they don't use "buffer objects" (see Chapter 3), which are generally used in WebGL. Although this helps by simplifying the explanation, some browsers (especially Firefox) expect buffer objects and may fail to display correctly any examples without them. In later chapters, and in actual application development, this will not cause problems because you will be using "buffer objects." However, if you are having problems, try another browser. You can switch back in the next chapter.

Unfortunately, this is not possible. WebGL relies on a new type of drawing mechanism called a **shader**, which offers a flexible and powerful mechanism for drawing 2D and 3D objects and must be used by all WebGL applications. Shaders, although powerful, are more complex, and you can't just specify a simple draw command.

Because the shader is a critical core mechanism in WebGL programming that you will use throughout this book, let's examine it one step at a time so that you can understand it easily.

HelloPoint1.html

Listing 2.4 shows `HelloPoint1.html`, which is functionally equivalent to `HelloCanvas.html` (refer to Figure 2.7). The title of the web page and the JavaScript filename were changed (lines 5 and 16), but everything else remains the same. From now on, unless the HTML file is different from this example, we'll skip showing the HTML files for the examples.

Listing 2.4 HelloPoint1.html

```
 1  <!DOCTYPE html>
 2  <html lang="en">
 3    <head>
 4      <meta charset="utf-8" />
 5       <title>Draw a point (1)</title>
 6    </head>
 7
 8    <body onload="main()">
 9      <canvas id="webgl" width="400" height="400">
10      Please use the browser supporting "canvas".
11      </canvas>
12
13      <script src="../libs/webgl-utils.js"></script>
14      <script src="../libs/webgl-debug.js"></script>
15      <script src="../libs/cuon-utils.js"></script>
16      <script src="HelloPoint1.js"></script>
17    </body>
18  </html>
```

HelloPoint1.js

Listing 2.5 shows `HelloPoint1.js`. As you can see from the comments in the listing, two "shader programs" are prepended to the JavaScript (lines 2 to 13). Glance through the shader programs, and then go to the next section, where you'll see more detailed explanations.

Listing 2.5 HelloPoint1.js

```
// HelloPoint1.js
// Vertex shader program
var VSHADER_SOURCE =
  'void main() {\n' +
  '  gl_Position = vec4(0.0, 0.0, 0.0, 1.0);\n' + // Coordinates
  '  gl_PointSize = 10.0;\n' +             // Set the point size
  '}\n';

// Fragment shader program
var FSHADER_SOURCE =
  'void main() {\n' +
  '  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n' + // Set the color
  '}\n';

function main() {
  // Retrieve <canvas> element
  var canvas = document.getElementById('webgl');

  // Get the rendering context for WebGL
  var gl = getWebGLContext(canvas);
  if (!gl) {
    console.log('Failed to get the rendering context for WebGL');
    return;
  }

  // Initialize shaders
  if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
    console.log('Failed to initialize shaders.');
    return;
  }

  // Set the color for clearing <canvas>
  gl.clearColor(0.0, 0.0, 0.0, 1.0);

  // Clear <canvas>
  gl.clear(gl.COLOR_BUFFER_BIT);

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

What Is a Shader?

`HelloPoint1.js` is our first WebGL program that uses shaders. As mentioned earlier, shader programs are necessary when you want to draw something on the screen in WebGL. Essentially, shader programs are "embedded" in the JavaScript file and, in this case, set up at the start. This seems at first sight to be complicated, but let's take it one step at a time.

WebGL needs the following two types of shaders, which you saw at line 2 and line 9:

- **Vertex shader:** Vertex shaders are programs that describe the traits (position, colors, and so on) of a vertex. The **vertex** is a point in 2D/3D space, such as the corner or intersection of a 2D/3D shape.
- **Fragment shader:** A program that deals with per-fragment processing such as lighting (see Chapter 8, "Lighting Objects"). The **fragment** is a WebGL term that you can consider as a kind of pixel (picture element).

You'll explore shaders in more detail later, but simply put, in a 3D scene, it's not enough just to draw graphics. You have to also account for how they are viewed as light sources hit them or the viewer's perspective changes. Shading does this with a high degree of flexibility and is part of the reason that today's 3D graphics are so realistic, allowing them to use new rendering effects to achieve stunning results.

The shaders are read from the JavaScript and stored in the WebGL system ready to be used for drawing. Figure 2.10 shows the basic processing flow from a JavaScript program into the WebGL system, which applies the shader programs to draw shapes that the browser finally displays.

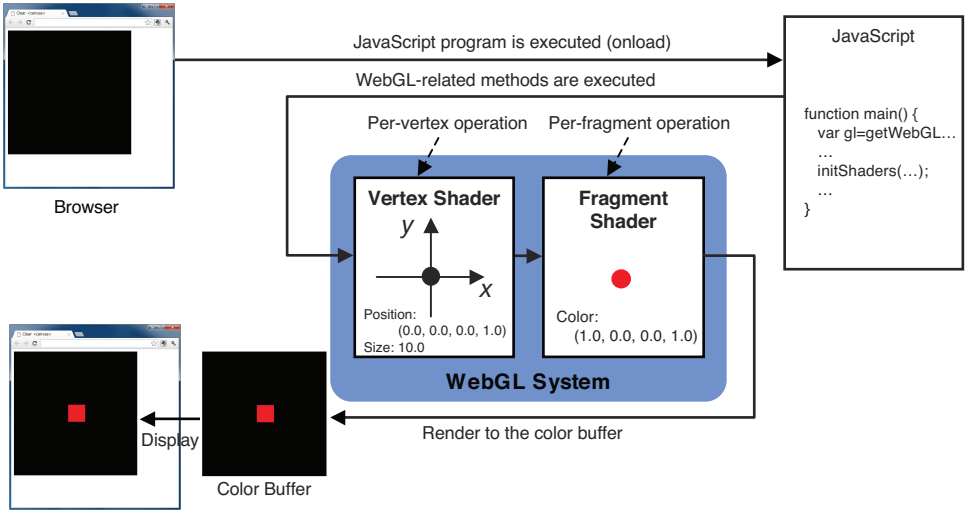

Figure 2.10 The processing flow from executing a JavaScript program to displaying the result in a browser

You can see two browser windows on the left side of the figure. These are the same; the upper one shows the browser before executing the JavaScript program, and the lower one shows the browser after execution. Once the WebGL-related methods are called from the JavaScript program, the vertex shader in the WebGL system is executed, and the fragment shader is executed to draw the result into the color buffer. This is the clear part—that is, step 4 in Figure 2.8, described in the original `HelloCanvas` example. Then the content in the color buffer is automatically displayed on the drawing area specified by the `<canvas>` in the browser.

You'll be seeing this figure frequently in the rest of this book. So we'll use a simplified version to save space (see Figure 2.11). Note that the flow is left to right and the right-most component is a color buffer, not a browser, because the color buffer is automatically displayed in the browser.

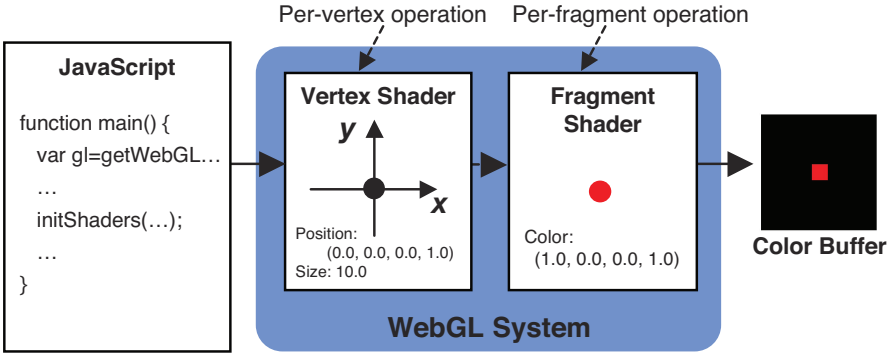

Figure 2.11 The simplified version of Figure 2.9

Getting back to our example, the goal is to draw a 10-pixel point on the screen. The two shaders are used as follows:

- The vertex shader specifies the position of a point and its size. In this sample program, the position is (0.0, 0.0, 0.0), and the size is 10.0.
- The fragment shader specifies the color of fragments displaying the point. In this sample program, the color is red (1.0, 0.0, 0.0, 1.0).

The Structure of a WebGL Program that Uses Shaders

Based on what you've learned so far, let's look at `HelloPoint1.js` again (refer to Listing 2.5). This program has 40 lines and is a little more complex than `HelloCanvas.js` (18 lines). It consists of three parts, as shown in Figure 2.12. The `main()` function in JavaScript starts from line 15, and shader programs are located from lines 2 to 13.

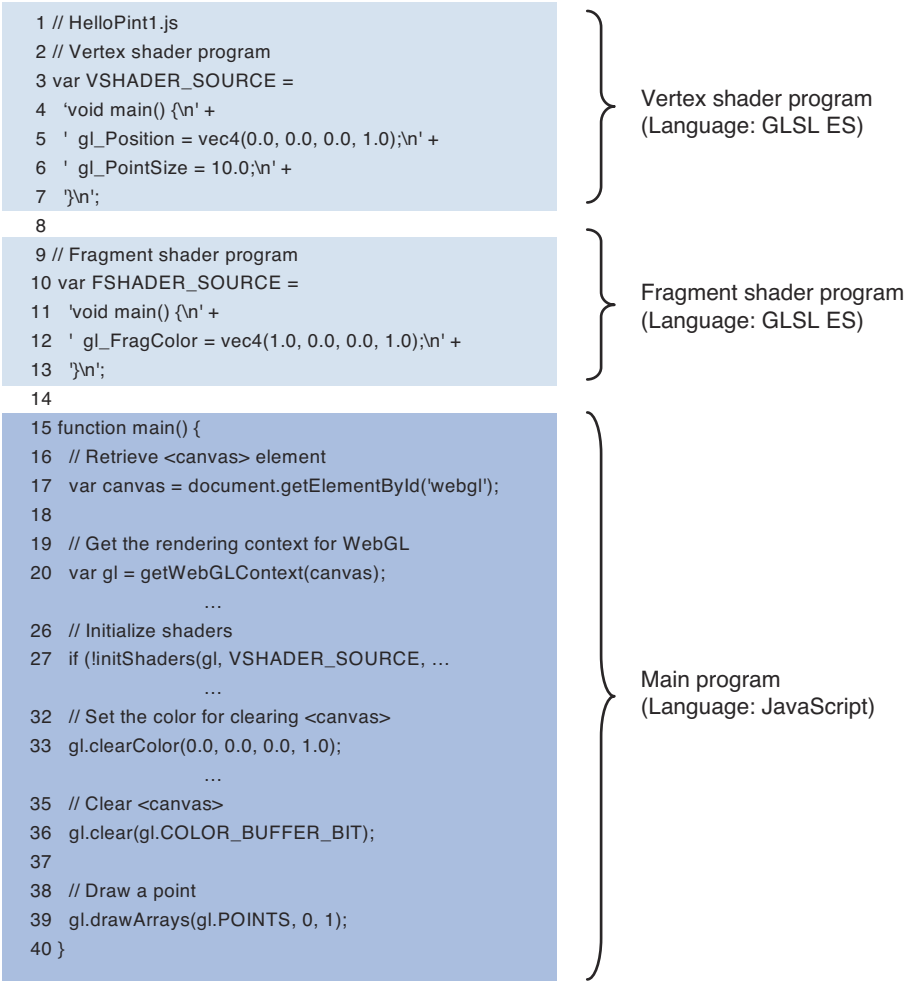

Figure 2.12 The basic structure of a WebGL program with embedded shader programs

The vertex shader program is located in lines 4 to 7, and the fragment shader is located in lines 11 to 13. These programs are actually the following shader language programs but written as a JavaScript string to make it possible to pass the shaders to the WebGL system:

```
// Vertex shader program
void main() {
  gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
  gl_PointSize = 10.0;
}
// Fragment shader program
void main() {
  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

As you learned in Chapter 1, shader programs must be written in the **OpenGL ES shading language (GLSL ES)**, which is similar to the C language. Finally, GLSL ES comes onto the stage! You will get to see the details of GLSL ES in Chapter 6, "The OpenGL ES Shading Language (GLSL ES)," but in these early examples, the code is simple and should be understandable by anybody with a basic understanding of C or JavaScript.

Because these programs must be treated as a single string, each line of the shader is concatenated using the `+` operator into a single string. Each line has `\n` at the end because the line number is displayed when an error occurs in the shader. The line number is helpful to check the source of the problem in the codes. However, the `\n` is not mandatory, and you could write the shader without it.

At lines 3 and 10, each shader program is stored in the variables `VSHADER_SOURCE` and `FSHADER_SOURCE` as a string:

```
 2 // Vertex shader program
 3 var VSHADER_SOURCE =
 4   'void main() {\n' +
 5   '  gl_Position = vec4(0.0, 0.0, 0.0, 1.0);\n' +
 6   '  gl_PointSize = 10.0;\n' +
 7   '}\n';
 8
 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';
```

If you are interested in loading the shader programs from files, refer to Appendix F, "Loading Shader Programs from Files."

Initializing Shaders

Before looking at the details of each shader, let's examine the processing flow of `main()` that is defined from line 15 in the JavaScript (see Figure 2.13). This flow, shown in Figure 2.13, is the basic processing flow of most WebGL applications. You will see the same flow throughout this book.

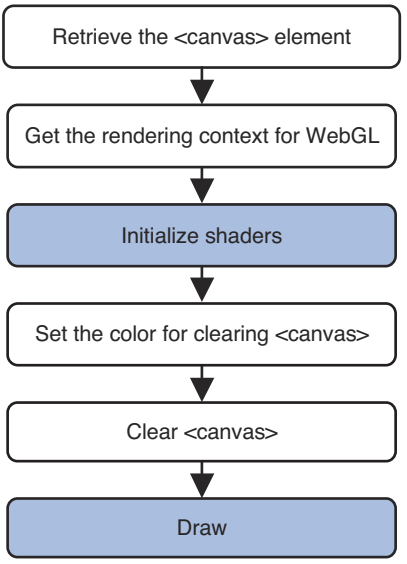

Figure 2.13 The processing flow of a WebGL program

This flow is similar to that shown in Figure 2.8 except that a third step ("Initialize Shaders") and a sixth step ("Draw") are added.

The third step "Initialize Shaders" initializes and sets up the shaders that are defined at line 3 and line 10 within the WebGL system. This step is done using the convenience function `initShaders()` that is defined in `cuon-util.js`. Again, this is one of those special functions we have provided for this book.

`initShaders(gl, vshader, fshader)`

Initialize shaders and set them up in the WebGL system ready for use:

Parameters	gl	Specifies a rendering context.
	vshader	Specifies a vertex shader program (string).
	fshader	Specifies a fragment shader program (string).
Return value	true	Shaders successfully initialized.
	false	Failed to initialize shaders.

Figure 2.14 shows how the convenience function `initShaders()` processes the shaders. You will examine in detail what this function is doing in Chapter 8. For now, you just need to understand that it sets up the shaders in the WebGL system and makes them ready for use.

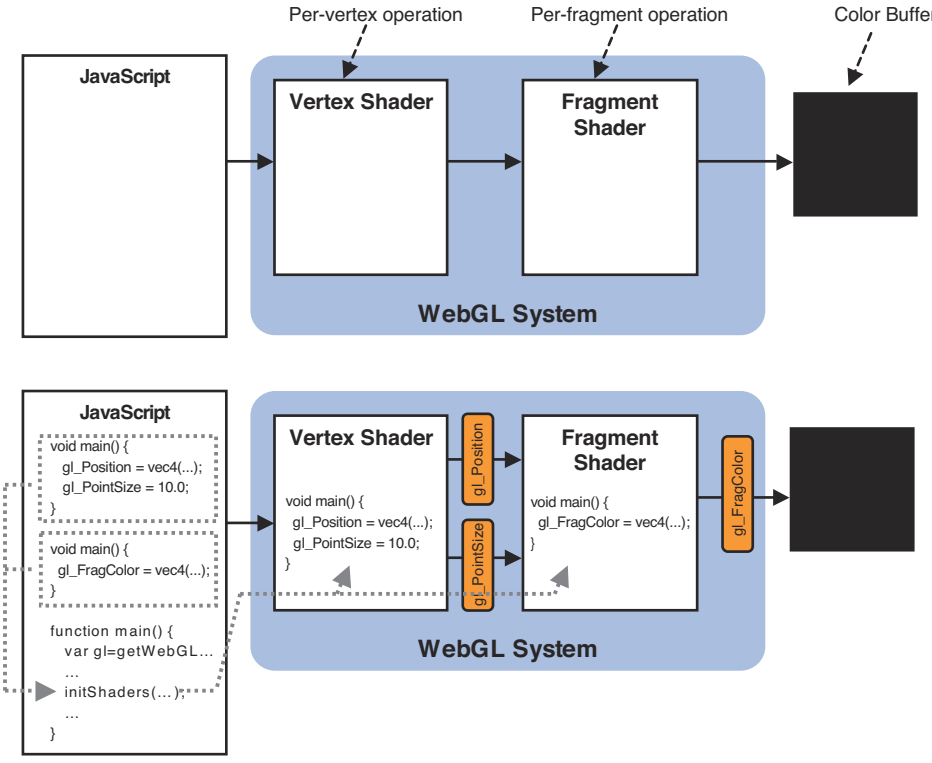

Figure 2.14 Behavior of initShaders()

As you can see in the upper figure in Figure 2.14, the WebGL system has two containers: one for a vertex shader and one for a fragment shader. This is actually a simplification, but helpful at this stage. We return to the details in Chapter 10. By default, the contents of these containers are empty. To make the shader programs, written as JavaScript strings and ready for use in the WebGL system, we need something to pass these strings to the system and then set them up in the appropriate containers; `initShaders()` performs this operation. Note that the shader programs are executed within the WebGL system, not the JavaScript program.

The lower portion in Figure 2.14 shows that after executing `initShaders()`, the shader programs that are passed as a string to the parameters of `initShaders()` are set up in the containers in the WebGL system and then made ready for use. The lower figure schematically illustrates that a vertex shader is passing `gl_Position` and `gl_PointSize` to a fragment shader and that just after assigning values to these variables in the vertex shader, the fragment shader is executed. In actuality, the fragments that are generated after processing these values are passed to the fragment shader. Chapter 5 explains this mechanism in detail, but for now you can consider the attributes to be passed.

The important point here is that *WebGL applications consist of a JavaScript program executed by the browser and shader programs that are executed within the WebGL system.*

Now, having completed the explanation of the second step "Initialize Shaders" in Figure 2.13, you are ready to see how the shaders are actually used to draw a simple point. As mentioned, we need three items of information for the point: its position, size, and color, which are used as follows:

- The vertex shader specifies the position of a point and its size. In this sample program, the position is (0.0, 0.0, 0.0), and the size is 10.0.
- The fragment shader specifies the color of the fragments displaying the point. In this sample program, they are red (1.0, 0.0, 0.0, 1.0).

Vertex Shader

Now, let us start by examining the vertex shader program listed in `HelloPoint1.js` (refer to Listing 2.5), which sets the position and size of the point:

```
2  // Vertex shader program
3  var VSHADER_SOURCE =
4    'void main() {\n' +
5    '  gl_Position = vec4(0.0, 0.0, 0.0, 1.0);\n' +
6    '  gl_PointSize = 10.0;\n' +
7    '}\n';
```

The vertex shader program itself starts from line 4 and must contain a single `main()` function in a similar fashion to languages such as C. The keyword `void` in front of `main()` indicates that this function does not return a value. You cannot specify other arguments to `main()`.

Just like JavaScript, we can use the `=` operator to assign a value to a variable in a shader. Line 5 assigns the position of the point to the variable `gl_Position`, and line 6 assigns its size to the variable `gl_PointSize`. These two variables are built-in variables available only in a vertex shader and have a special meaning: `gl_Position` specifies a position of a vertex (in this case, the position of the point), and `gl_PointSize` specifies the size of the point (see Table 2.2).

Table 2.2 Built-In Variables Available in a Vertex Shader

Type and Variable Name	Description
vec4 `gl_Position`	Specifies the position of a vertex
float `gl_PointSize`	Specifies the size of a point (in pixels)

Note that `gl_Position` should always be written. If you don't specify it, the shader's behavior is implementation dependent and may not work as expected. In contrast, `gl_PointSize` is only required when drawing points and defaults to a point size of 1.0 if you don't specify anything.

For those of you mostly familiar with JavaScript, you may be a little surprised when you see "type" specified in Table 2.2. Unlike JavaScript, GLSL ES is a "typed" programming language; that is, it requires the programmer to specify what type of data a variable holds. C and Java are examples of typed languages. By specifying "type" for a variable, the system can easily understand what type of data the variable holds, and then it can optimize its processing based on that information. Table 2.3 summarizes the "type" in GLSL ES used in the shaders in this section.

Table 2.3 Data Types in GLSL ES

Type	Description
float	Indicates a floating point number
vec4	Indicates a vector of four floating point numbers `float` `float` `Float` `float`

Note that an error will occur when the type of data that is assigned to the variable is different from the type of the variable. For example, the type of `gl_PointSize` is float, and you must assign a floating point number to it. So, if you change line 6 from

```
gl_PointSize = 10.0;
```

to

```
gl_PointSize = 10;
```

it will generate an error simply because `10` is interpreted as an integer number, whereas `10.0` is a floating point number in GLSL ES.

The type of the variable `gl_Position`, the built-in variable for specifying the position of a point, is `vec4`; `vec4` is a vector made up of three floats. However, you only have three floats (0.0, 0.0, 0.0) representing X, Y, and Z. So you need to convert these to a `vec4` somehow. Fortunately, there is a built-in function, `vec4()`, that will do this for you and return a value of type `vec4` –, which is just what you need!

vec4 vec4(v0, v1, v2, v3)

Construct a `vec4` object from *v0*, *v1*, *v2*, and *v3*.

Parameters	v0, v1, v2, v3	Specifies floating point numbers.
Return value	A `vec4` object made from *v0*, *v1*, *v2*, and *v3*.	

In this sample program, `vec4()` is used at line 5 as follows:

```
gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
```

Note that the value that is assigned to `gl_Position` has 1.0 added as a fourth component. This four-component coordinate is called a **homogeneous coordinate** (see the boxed article below) and is often used in 3D graphics for processing three-dimensional information efficiently. Although the homogeneous coordinate is a four-dimensional coordinate, if the last component of the homogeneous coordinate is 1.0, the coordinate indicates the same position as a three-dimensional one. So, you can supply 1.0 as the last component if you need to specify four components as a vertex coordinate.

Homogeneous Coordinates

The homogeneous coordinates use the following coordinate notation: (x, y, z, w). The homogeneous coordinate (x, y, z, w) is equivalent to the three-dimensional coordinate (x/w, y/w, z/w). So, if you set w to 1.0, you can utilize the homogeneous coordinate as a three-dimensional coordinate. The value of w must be greater than or equal to 0. If w approaches zero, the coordinates approach infinity. So we can represent the concept of infinity in the homogeneous coordinate system. Homogeneous coordinates make it possible to represent vertex transformations described in the next chapter as a multiplication of a matrix and the coordinates. These coordinates are often used as an internal representation of a vertex in 3D graphics systems.

Fragment Shader

After specifying the position and size of a point, you need to specify its color using a fragment shader. As explained earlier, a **fragment** is a pixel displayed on the screen, although technically the fragment is a pixel along with its position, color, and other information.

The fragment shader is a program that processes this information in preparation for displaying the fragment on the screen. Looking again at the fragment shader listed in `HelloPoint1.js` (refer to Listing 2.5), you can see that just like a vertex shader, a fragment shader is executed from `main()`:

```
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';
```

The job of the shader is to set the color of the point as its per-fragment operation, which is carried out at line 12. `gl_FragColor` is a built-in variable only available in a fragment shader; it controls the color of a fragment, as shown in Table 2.4.

Table 2.4 The Built-In Value Available in a Fragment Shader

Type and Variable Name	Description
vec4 `gl_FragColor`	Specify the color of a fragment (in RGBA)

When we assign a color value to the built-in variable, the fragment is displayed using that color. Just like the position in the vertex shader, the color value is a `vec4` data type consisting of four floating point numbers representing the RGBA values. In this sample program, a red point will be displayed because you assign (1.0, 0.0, 0.0, 1.0) to the variable.

The Draw Operation

Once you set up the shaders, the remaining task is to draw the shape, or in our case, a point. As before, you need to clear the drawing area in a similar way to that described in `HelloCanvas.js`. Once the drawing area is cleared, you can draw the point using `gl.drawArrays()`, as in line 39:

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

`gl.drawArrays()` is a powerful function that is capable of drawing a variety of basic shapes, as detailed in the following box.

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 which vertex to start drawing from (integer).
	count	Specifies the number of vertices to be used (integer).
Return value	None	
Errors	INVALID_ENUM *mode* is none of the preceding values.	
	INVALID_VALUE *first* is negative or *count* is negative.	

In this sample program, because you are drawing a point, you specify `gl.POINTS` as the *mode* in the first parameter. The second parameter is set to 0 because you are starting from the first vertex. The third parameter, *count*, is 1 because you are only drawing 1 point in this sample program.

Now, when the program makes a call to `gl.drawArrays()`, the vertex shader is executed *count* times, each time working with the next vertex. In this sample program, the shader is executed once (*count* is set to 1) because we only have one vertex: our point. When the shader is executed, the function `main()` in the shader is called, and then each line in the function is executed sequentially, resulting in (0.0, 0.0, 0.0, 1.0) being assigned to `gl_Position` (line 5) and then 10.0 assigned to `gl_PointSize` (line 6).

Once the vertex shader executes, the fragment shader is executed by calling its `main()` function which, in this example, assigns the color value (red) to `gl_FragColor` (line 12). As a result, a red point of 10 pixels is drawn at (0.0, 0.0, 0.0, 1.0), or the center of the drawing area (see Figure 2.15).

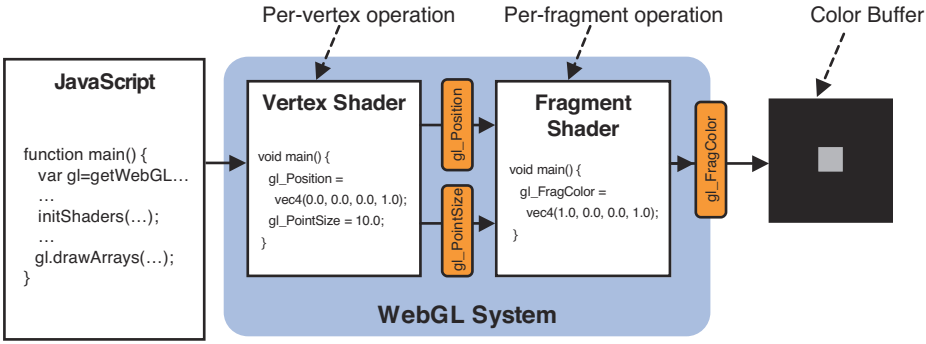

Figure 2.15 The behavior of shaders

At this stage, you should have a rough understanding of the role of a vertex shader and a fragment shader and how they work. In the rest of this chapter, you'll build on this basic understanding through a series of examples, allowing you to become more accustomed to WebGL and shaders. However, before that, let's quickly look at how WebGL describes the position of shapes using its coordinate system.

The WebGL Coordinate System

Because WebGL deals with 3D graphics, it uses a three-dimensional coordinate system along the x-, y-, and z-axis. This coordinate system is easy to understand because our world has the same three dimensions: width, height, and depth. In any coordinate system, the direction of each axis is important. Generally, in WebGL, when you face the computer screen, the horizontal direction is the x-axis (right direction is positive), the vertical direction is the y-axis (up direction is positive), and the direction from the screen to the viewer is the z-axis (the left side of Figure 2.16). The viewer's eye is located at the origin (0.0, 0.0, 0.0), and the line of sight travels along the negative direction of the z-axis, or from you into the screen (see the right side of Figure 2.16). This coordinate system is also called the **right-handed coordinate system** because it can be expressed using the right hand (see Figure 2.17) and is the one normally associated with WebGL. Throughout this book, we'll use the right-handed coordinate system as the default for WebGL. However, you should note that it's actually more complex than this. In fact, WebGL is neither left handed nor right handed. This is explained in detail in Appendix D, "WebGL/OpenGL: Left or Right Handed?," but it's safe to treat WebGL as right handed for now.

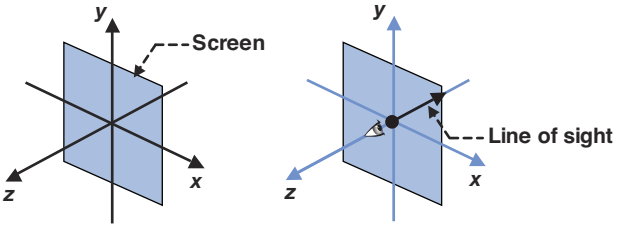

Figure 2.16 WebGL coordinate system

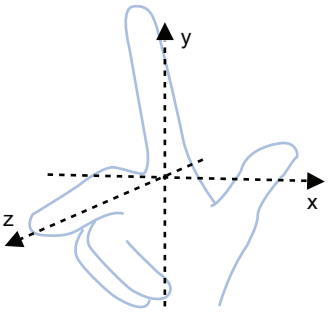

Figure 2.17 The right-handed coordinate system

As you have already seen, the drawing area specified for a `<canvas>` element in JavaScript is different from WebGL's coordinate system, so a mapping is needed between the two. By default, as you see in Figure 2.18, WebGL maps the coordinate system to the area as follows:

- The center position of a `<canvas>`: (0.0, 0.0, 0.0)
- The two edges of the x-axis of the `<canvas>`: (–1.0, 0.0, 0.0) and (1.0, 0.0, 0.0)
- The two edges of the y-axis of the `<canvas>`: (0.0, –1.0, 0.0) and (0.0, 1.0, 0.0)

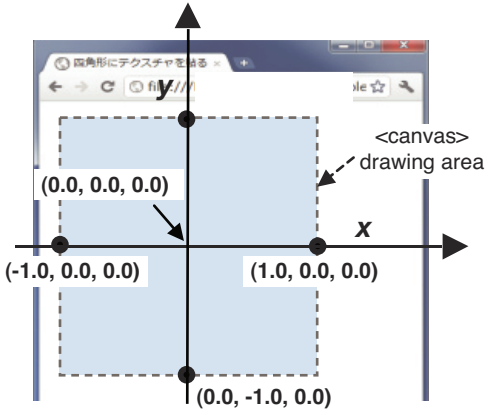

Figure 2.18 The <canvas> drawing area and WebGL coordinate system

As previously discussed, this is the default. It's possible to use another coordinate system, which we'll discuss later, but for now this default coordinate system will be used. Additionally, to help you stay focused on the core functionality of WebGL, the example programs will mainly use the x and y coordinates and not use the z or depth coordinate. Therefore, until Chapter 7, the z-axis value will generally be specified as 0.0.

Experimenting with the Sample Program

First, you can modify line 5 to change the position of the point and gain a better understanding of the WebGL coordinate system. For example, let's change the x coordinate from 0.0 to 0.5 as follows:

```
5    '  gl_Position = vec4(0.5, 0.0, 0.0, 1.0);\n' +
```

Save the modified `HelloPoint1.js` and click the Reload button on your browser to reload it. You will see that the point has moved and is now displayed on the right side of the `<canvas>` area (see Figure 2.19, left side).

Now change the y coordinate to move the point toward the top of the `<canvas>` as follows:

```
5    '  gl_Position = vec4(0.0, 0.5, 0.0, 1.0);\n' +
```

Again, save the modified `HelloPoint1.js` and reload it. This time, you can see the point has moved and is displayed in the upper part of the canvas (see Figure 2.19, right side).

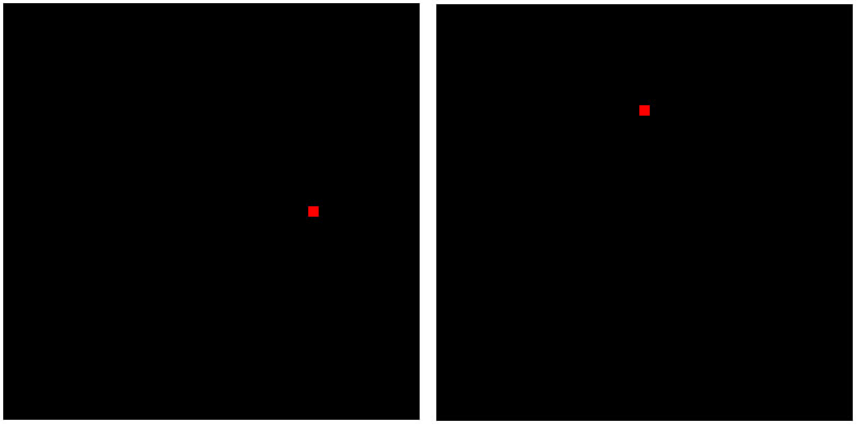

Figure 2.19 Modifying the position of the point

As another experiment, let's try changing the color of the point from red to green by modifying line 12, as follows:

```
12    '  gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);\n' +
```

Let's conclude this section with a quick recap. You've been introduced to the two basic shaders we use in WebGL—the vertex shader and the fragment shader—and seen how, although they use their own language, they can be executed from within JavaScript. You've also seen that the basic processing flow of a WebGL application using shaders is the same as in other types of WebGL applications. A key lesson from this section is that a WebGL program consists of a JavaScript program executing in conjunction with shader programs.

For those of you with experience in using OpenGL, you may feel that something is missing; there is no code to swap color buffers. One of the significant features of WebGL is that it does not need to do that. For more information, see Appendix A, "No Need to Swap Buffers in WebGL."

Draw a Point (Version 2)

In the previous section, you explored drawing a point and the related core functions of shaders. Now that you understand the fundamental behavior of a WebGL program, let's examine how to pass data between JavaScript and the shaders. `HelloPoint1` always draws a point at the same position because its position is directly written ("hard-coded") in the vertex shader. This makes the example easy to understand, but it lacks flexibility. In this section, you'll see how a WebGL program can pass a vertex position from JavaScript to the vertex shader and then draw a point at that position. The name of the program is `HelloPoint2`, and although the result of the program is the same as `HelloPoint1`, it's a flexible technique you will use in future examples.

Using Attribute Variables

Our goal is to pass a position from the JavaScript program to the vertex shader. There are two ways to pass data to a vertex shader: attribute variable and uniform variable (see Figure 2.20). The one you use depends on the nature of the data. The **attribute variable** passes data that differs for each vertex, whereas the **uniform variable** passes data that is the same (or uniform) in each vertex. In this program, you will use the attribute variable because each vertex generally has different coordinates.

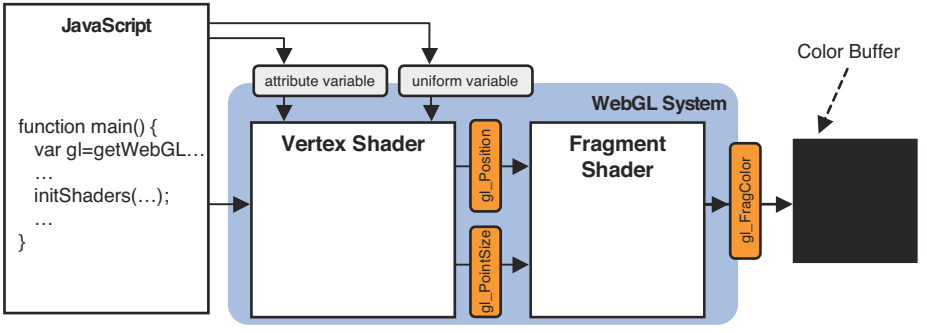

Figure 2.20 Two ways to pass data to a vertex shader

The attribute variable is a GLSL ES variable which is used to pass data from the world outside a vertex shader into the shader and is only available to vertex shaders.

To use the attribute variable, the sample program involves the following three steps:

1. Prepare the attribute variable for the vertex position in the vertex shader.
2. Assign the attribute variable to the `gl_Position` variable.
3. Pass the data to the attribute variable.

Let's look at the sample program in more detail to see how to carry out these steps.

Sample Program (HelloPoint2.js)

In `HelloPoint2` (see Listing 2.6), you draw a point at a position the JavaScript program specifies.

Listing 2.6 HelloPoint2.js

```
 1 // HelloPoint2.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
    ... snipped because it is the same as HelloPoint1.js
15
16 function main() {
17   // Retrieve <canvas> element
18   var canvas = document.getElementById('webgl');
19
20   // Get the rendering context for WebGL
21   var gl = getWebGLContext(canvas);
       ...
26
27   // Initialize shaders
28   if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
         ...
31   }
32
33   // Get the storage location of attribute variable
34   var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
35   if (a_Position < 0) {
36     console.log('Failed to get the storage location of a_Position');
37     return;
```

```
38   }
39
40   // Pass vertex position to attribute variable
41   gl.vertexAttrib3f(a_Position, 0.0, 0.0, 0.0);
42
43   // Set the color for clearing <canvas>
44   gl.clearColor(0.0, 0.0, 0.0, 1.0);
45
46   // Clear <canvas>
47   gl.clear(gl.COLOR_BUFFER_BIT);
48
49   // Draw a point
50   gl.drawArrays(gl.POINTS, 0, 1);
51  }
```

As you can see, the attribute variable is prepared within the shader on line 4:

```
4    'attribute vec4 a_Position;\n' +
```

In this line, the keyword `attribute` is called a **storage qualifier**, and it indicates that the following variable (in this case, `a_Position`) is an attribute variable. This variable must be declared as a global variable because data is passed to it from outside the shader. The variable must be declared following a standard pattern `<Storage Qualifier> <Type> <Variable Name>`, as shown in Figure 2.21.

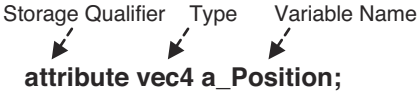

Figure 2.21 The declaration of the attribute variable

In line 4, you declare `a_Position` as an attribute variable with data type `vec4` because, as you saw in Table 2.2, it will be assigned to `gl_Position`, which always requires a `vec4` type.

Note that throughout this book, we have adopted a programming convention in which all attribute variables have the prefix `a_`, and all uniform variables have the prefix `u_` to easily determine the type of variables from their names. Obviously, you can use your own convention when writing your own programs, but we find this one simple and clear.

Once `a_Position` is declared, it is assigned to `gl_Position` at line 6:

```
6    '  gl_Position = a_Position;\n' +
```

At this point, you have completed the preparation in the shader for receiving data from the outside. The next step is to pass the data to the attribute variable from the JavaScript program.

Getting the Storage Location of an Attribute Variable

As you saw previously, the vertex shader program is set up in the WebGL system using the convenience function `initShaders()`. When the vertex shader is passed to the WebGL system, the system parses the shader, recognizes it has an attribute variable, and then prepares the location of its attribute variable so that it can store data values when required. When you want to pass data to `a_Position` in the vertex shader, you need to ask the WebGL system to give you the location it has prepared, which can be done using `gl.getAttribLocation()`, as shown in line 34:

```
33   // Get the location of attribute variable
34   var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
35   if (a_Position < 0) {
36     console.log('Fail to get the storage location of a_Position');
37     return;
38   }
```

The first argument of this method specifies a **program object** that holds the vertex shader and the fragment shader. You will examine the program object in Chapter 8, but for now, you can just specify `gl.program` as the argument here. Note that you should use `gl.program` only after `initShaders()` has been called because `initShaders()` assigns the program object to the variable. The second parameter specifies the attribute variable name (in this case `a_Position`) whose location you want to know.

The return value of this method is the storage location of the specified attribute variable. This location is then stored in the JavaScript variable, `a_Position`, at line 34 for later use. Again, for ease of understanding, this book uses JavaScript variable names for attribute variables, which are the same as the GLSL ES attribute variable name. You can, of course, use any variable name.

The specification of `gl.getAttribLocation()` is as follows:

`gl.getAttribLocation(program, name)`

Retrieve the storage location of the attribute variable specified by the *name* parameter.

Parameters	program	Specifies the program object that holds a vertex shader and a fragment shader.
	name	Specifies the name of the attribute variable whose location is to be retrieved.
Return value	greater than or equal to 0	The location of the specified attribute variable.
	-1	The specified attribute variable does not exist or its name starts with the reserved prefix `gl_` or `webgl_`.

Errors	INVALID_OPERATION	*program* has not been successfully linked (See Chapter 9.)
	INVALID_VALUE	The length of *name* is more than the maximum length (256 by default) of an attribute variable name.

Assigning a Value to an Attribute Variable

Once you have the attribute variable location, you need to set the value using the `a_Position` variable. This is performed at line 41 using the `gl.vertexAttrib3f()` method.

```
40    // Set vertex position to attribute variable
41    gl.vertexAttrib3f(a_Position, 0.0, 0.0, 0.0);
```

The following is the specification of `gl.vertexAtrrib3f()`.

`gl.vertexAttrib3f(location, v0, v1, v2)`

Assign the data (*v0*, *v1*, and *v2*) to the attribute variable specified by *location*.

Parameters	location	Specifies the storage location of an attribute variable to be modified.
	v0	Specifies the value to be used as the first element for the attribute variable.
	v1	Specifies the value to be used as the second element for the attribute variable.
	v2	Specifies the value to be used as the third element for the attribute variable.
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).

The first argument of the method call specifies the location returned by `gl.getAttrib-Location()` at line 34. The second, third, and fourth arguments specify the floating point number to be passed to `a_Position` representing the x, y, and z coordinates of the point. After calling the method, these three values are passed as a group to `a_Position`, which was prepared at line 4 in the vertex shader. Figure 2.22 shows the processing flow of getting the location of the attribute variable and then writing a value to it.

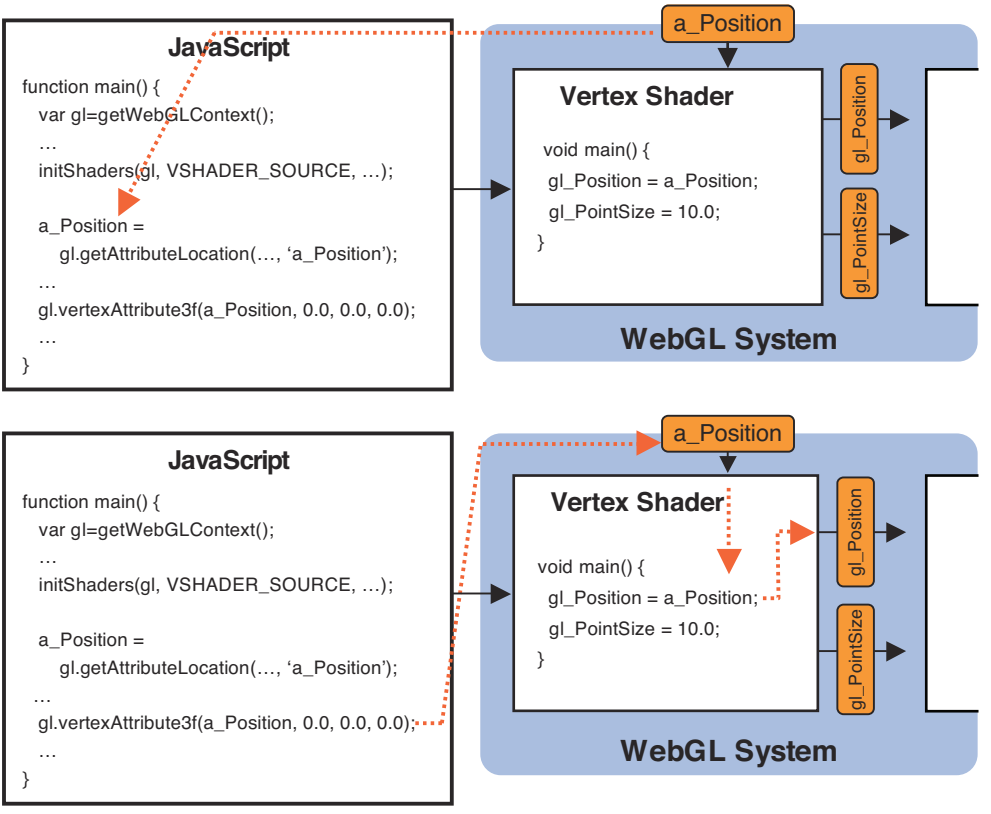

Figure 2.22 Getting the storage location of an attribute variable and then writing a value to the variable

`a_Position` is then assigned to `gl_Position` at line 6 in the vertex shader, in effect passing the x, y, and z coordinates from your JavaScript, via the attribute variable into the shader, where it's written to `gl_Position`. So the program has the same effect as `HelloPoint1`, where `gl_Position` is used as the position of a point. However, `gl_Position` has now been set dynamically from JavaScript rather than statically in the vertex shader:

```
4    'attribute vec4 a_Position;\n' +
5    'void main() {\n' +
6    '  gl_Position = a_Position;\n' +
7    '  gl_PointSize = 10.0;\n' +
8    '}\n';
```

Finally, you clear the `<canvas>` using `gl.clear()` (line 47) and draw the point using `gl.drawArrays()` (line 50) in the same way as in `HelloPoint1.js`.

As a final note, you can see `a_Position` is prepared as `vec4` at line 4 in the vertex shader. However, `gl.vertexAttrib3f()` at line 41 specifies only three values (x, y, and z), not four. Although you may think that one value is missing, this method automatically supplies the value 1.0 as the fourth value (see Figure 2.23). As you saw earlier, a default fourth value of 1.0 for a color ensures it is fully opaque, and a default fourth value of 1.0 for a homogeneous coordinate maps a 3D coordinate into a homogenous coordinate, so essentially the method is supplying a "safe" fourth value.

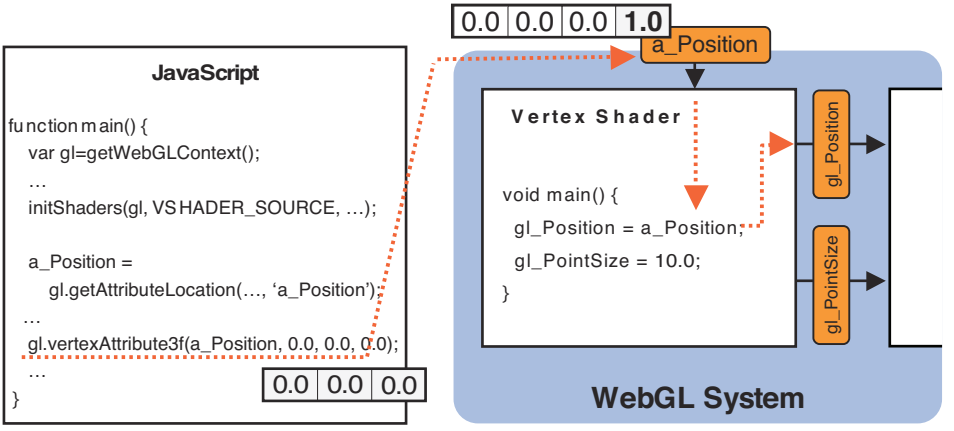

Figure 2.23 The missing data is automatically supplied

Family Methods of gl.vertexAttrib3f()

`gl.vertexAttrib3f()` is part of a family of methods that allow you to set some or all of the components of the attribute variable. `gl.vertexAttrib1f()` is used to assign a single value (v0), `glvertexAttrib2f()` assigns two values (v0 and v1), and `gl.vertexAttrib4f()` assigns four values (v0, v1, v2, and v3).

```
gl.vertexAttrib1f(location, v0)
gl.vertexAttrib2f(location, v0, v1)
gl.vertexAttrib3f(location, v0, v1, v2)
gl.vertexAttrib4f(location, v0, v1, v2, v3)
```

Assign data to the attribute variable specified by *location*. `gl.vertexAttrib1f()` indicates that only one value is passed, and it will be used to modify the first component of the attribute variable. The second and third components will be set to 0.0, and the fourth component will be set to 1.0. Similarly, `gl.vertexAttrib2f()` indicates that values are provided for the first two components, the third component will be set to 0.0, and the fourth component will be set to 1.0. `gl.vertexAttrib3f()` indicates that values are provided for the first three components, and the fourth component will be set to 1.0, whereas `gl.vertexAttrib4f()` indicates that values are provided for all four components.

Parameters	location	Specifies the storage location of the attribute variable.
	v0, v1, v2, v3	Specifies the values to be assigned to the first, second, third, and fourth components of the attribute variable.
Return value	None	
Errors	INVALID_VALUE	*location* is greater than or equal to the maximum number of attribute variables (8 by default).

The vector versions of these methods are also available. Their name contains "v" (vector), and they take a typed array (see Chapter 4) as a parameter. The number in the method name indicates the number of elements in the array. For example,

```
var position = new Float32Array([1.0, 2.0, 3.0, 1.0]);
gl.vertexAttrib4fv(a_Position, position);
```

In this case, `4` in the method name indicates that the length of the array is 4.

The Naming Rules for WebGL-Related Methods

You may be wondering what `3f` in `gl.vertexAttrib3f()` actually means. WebGL bases its method names on the function names in OpenGL ES 2.0, which as you now know, is the base specification of WebGL. The function names in OpenGL comprise the three components: `<base function name> <number of parameters> < parameter type>`. The name of each WebGL method also uses the same components: `<base method name> <number of parameters> <parameter type>` as shown in Figure 2.24.

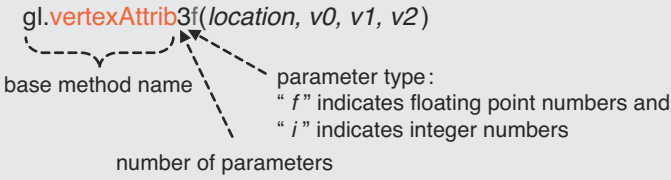

Figure 2.24 The naming rules of WebGL-related methods

As you can see in the example, in the case of `gl.vertexAttrib3f()`, the base method name is `vertexAttrib`, the number of parameters is 3, and the parameter type is f (that is, float or floating point number). This method is a WebGL version of the function `glVertexAttrib3f()` in OpenGL. Another character for the parameter type is i, which indicates integer. You can use the following notation to represent all methods from `gl.vertexAttrib1f()` to `gl.vertexAttrib4f()`: `gl.vertexAttrib[1234]f()`.

Where `[]` indicates that one of the numbers in it can be selected.

When v is appended to the name, the methods take an array as a parameter. In this case, the number in the method name indicates the number of elements in the array.

```
var positions = new Float32Array([1.0, 2.0, 3.0, 1.0]);
gl.vertexAttrib4fv(a_Position, positions);
```

Experimenting with the Sample Program

Now that you have the program to pass the position of a point from a JavaScript program to a vertex shader, let's change that position. For example, if you wanted to display a point at (0.5, 0.0, 0.0), you would modify the program as follows:

```
33    gl.vertexAttrib3f(a_Position, 0.5, 0.0, 0.0);
```

You could use other family methods of `gl.vertexAttrib3f()` to perform the same task in the following way:

```
gl.vertexAttrib1f(a_Position, 0.5);
gl.vertexAttrib2f(a_Position, 0.5, 0.0);
gl.vertexAttrib4f(a_Position, 0.5, 0.0, 0.0, 1.0);
```

Now that you are comfortable using attribute variables, let's use the same approach to change the size of the point from within your JavaScript program. In this case, you will need a new attribute variable for passing the size to the vertex shader. Following the naming rule used in this book, let's use `a_PointSize`. As you saw in Table 2.2, the type of `gl_PointSize` is `float`, so you need to prepare the variable using the same type as follows:

```
attribute float a_PointSize;
```

So, the vertex shader becomes:

```
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';
```

Then, after you get the storage location of `a_PointSize`, to pass the size of the point to the variable, you can use `gl.vertexAttrib1f()`. Because the type of `a_PointSize` is `float`, you can use `gl.vertexAttrib1f()` as follows:

```
33  // Get the storage location of attribute variable
34  var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
    ...
```

```
39  var a_PointSize = gl.getAttribLocation(gl.program, 'a_PointSize');
40  // Set vertex position to attribute variable
41  gl.vertexAttrib3f(a_Position, 0.0, 0.0, 0.0);
42  gl.vertexAttrib1f(a_PointSize, 5.0);
```

At this stage, you might want to experiment a little with the program and make sure you understand how to use these attribute variables and how they work.

Draw a Point with a Mouse Click

The previous program `HelloPoint2` can pass the position of a point to a vertex shader from a JavaScript program. However, the position is still hard-coded, so it is not so different from `HelloPoint1`, in which the position was also directly written in the shader.

In this section, you add a little more flexibility, exploiting the ability to pass the position from a JavaScript program to a vertex shader, to draw a point at the position where the mouse is clicked. Figure 2.25 shows the screen shot of `ClickedPoint`.[4]

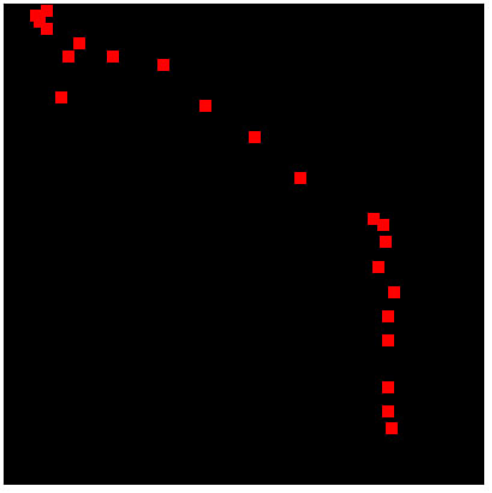

Figure 2.25 ClickedPoint

This program uses an event handler to handle mouse-related events, which will be familiar to those of you who have written JavaScript programs.

Sample Program (ClickedPoints.js)

Listing 2.7 shows `ClickedPoints.js`. For brevity, we have removed code sections that are the same as the previous example and replaced these with ....

[4] © 2012 Marisuke Kunnya

Listing 2.7 ClickedPoints.js

```
// ClickedPoints.js
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'void main() {\n' +
  '  gl_Position = a_Position;\n' +
  '  gl_PointSize = 10.0;\n' +
  '}\n';

// Fragment shader program
 ...
function main() {
  // Retrieve <canvas> element
  var canvas = document.getElementById('webgl');

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

  // Get the storage location of a_Position variable
  var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
 ...
  // Register function (event handler) to be called on a mouse press
  canvas.onmousedown = function(ev) { click(ev, gl, canvas, a_Position); };
 ...
  gl.clear(gl.COLOR_BUFFER_BIT);
}

var g_points = []; // The array for a mouse press
function click(ev, gl, canvas, a_Position) {
  var x = ev.clientX; // x coordinate of a mouse pointer
  var y = ev.clientY; // y coordinate of a mouse pointer
  var rect = ev.target.getBoundingClientRect();

  x = ((x - rect.left) - canvas.height/2)/(canvas.height/2);
  y = (canvas.width/2 - (y - rect.top))/(canvas.width/2);
  // Store the coordinates to g_points array
  g_points.push(x); g_points.push(y);

```

```
61    // Clear <canvas>
62    gl.clear(gl.COLOR_BUFFER_BIT);
63
64    var len = g_points.length;
65    for(var i = 0; i < len; i+=2) {
66      // Pass the position of a point to a_Position variable
67      gl.vertexAttrib3f(a_Position, g_points[i], g_points[i+1], 0.0);
68
69      // Draw a point
70      gl.drawArrays(gl.POINTS, 0, 1);
71    }
72  }
```

Register Event Handlers

The processing flow from lines 17 to 39 is the same as `HelloPoint2.js`. These lines get the WebGL context, initialize the shaders, and then retrieve the location of the attribute variable. The main differences from `HelloPoint2.js` are the registration of an event handler (line 41) and the definition of the function `click()` as the handler (from line 51).

An event handler is an asynchronous callback function that handles input on a web page from a user, such as mouse clicks or key presses. This allows you to create a dynamic web page that can change the content that's displayed according to the user's input. To use an event handler, you need to register the event handler (that is, tell the system what code to run when the event occurs). The `<canvas>` supports special properties for registering event handlers for a specific user input, which you will use here.

For example, when you want to handle mouse clicks, you can use the `onmousedown` property of the `<canvas>` to specify the event handler for a mouse click as follows:

```
40    // Register function (event handler) to be called on a mouse press
41    canvas.onmousedown = function(ev) { click(ev, gl, canvas, a_Position); };
```

Line 41 uses the statement `function(){ ... }` to register the handler:

```
function(ev){ click(ev, gl, canvas, a_Position); }
```

This mechanism is called an **anonymous function**, which, as its name suggests, is a convenient mechanism when you define a function that does not need to have a name.

For those of you unfamiliar with this type of function, let's explain the mechanism using an example. Next, we define the variable `thanks` as follows using the form:

```
var thanks = function () { alert(' Thanks a million!'); }
```

We can execute the variable as a function as follows:

```
thanks(); // 'Thanks a million!' is displayed
```

You can see that the variable `thanks` can be operated as a function. These lines can be rewritten as follows:

```
function thanks() { alert('Thanks a million!'); }
thanks(); // 'Thanks a million!' is displayed
```

So why do you need to use an anonymous function? Well, when you draw a point, you need the following three variables: `gl`, `canvas`, and `a_Position`. These variables are local variables that are prepared in the function `main()` in the JavaScript program. However, when a mouse click occurs, the browser will automatically call the function that is registered to the `<canvas>`'s `onmousedown` property with a **predefined single parameter** (that is, an event object that contains information about the mouse press. Therefore, usually, you will register the event handler and define the function for it as follows:

```
canvas.onmousedown = mousedown; // Register "mousedown" as event handler
...
function mousedown(ev) { // Event handler: It takes one parameter "ev"
  ...
}
```

However, if you define the function in this way, it cannot access the local variables `gl`, `canvas`, and `a_Position` to draw a point. The anonymous function at line 41 provides the solution to make it possible to access them:

```
41    canvas.onmousedown = function(ev) { click(ev, gl, canvas, a_Position); };
```

In this code, when a mouse click occurs, the anonymous function `function(ev)` is called first. Then it calls `click(ev, gl, canvas, a_Position)` with the local variables defined in `main()`. Although this may seem a little complicated, it is actually quite flexible and avoids the use of global variables, which is always good. Take a moment to make sure you understand this approach, because you will often see this way of registering event handlers in this book.

Handling Mouse Click Events

Let's look at what the function `click()` is doing. The processing flow follows:

1. Retrieve the position of the mouse click and then store it in an array.
2. Clear `<canvas>`.
3. For each position stored in the array, draw a point.

```
50  var g_points = [];  // The array for mouse click positions
51  function click(ev, gl, canvas, a_Position) {
52    var x = ev.clientX; // x coordinate of a mouse pointer
53    var y = ev.clientY; // y coordinate of a mouse pointer
54    var rect = ev.target.getBoundingClientRect();
```

```
55
56    x = ((x - rect.left) - canvas.height/2)/(canvas.height/2);
57    y = (canvas.width/2 - (y - rect.top))/(canvas.width/2);
58    // Store the coordinates to a_points array                    <- (1)
59    g_points.push(x); g_points.push(y);
60
61    // Clear <canvas>
62    gl.clear(gl.COLOR_BUFFER_BIT);                                <- (2)
63
64    var len = g_points.length;
65    for(var i = 0; i < len; i+=2) {
66      // Pass the position of a point to a_Position variable      <- (3)
67      gl.vertexAttrib3f(a_Position, g_points[i], g_points[i+1], 0.0);
68
69      // Draw a point
70      gl.drawArrays(gl.POINTS, 0, 1);
71    }
72  }
```

The information about the position of a mouse click is stored as an event object and passed by the browser using the argument `ev` to the function `click()`. `ev` holds the position information, and you can get the coordinates by using `ev.clientX` and `ev.clientY`, as shown in lines 52 and 53. However, you cannot use the coordinates directly in this sample program for two reasons:

1. The coordinate is the position in the "client area" in the browser, not in the `<canvas>` (see Figure 2.26).

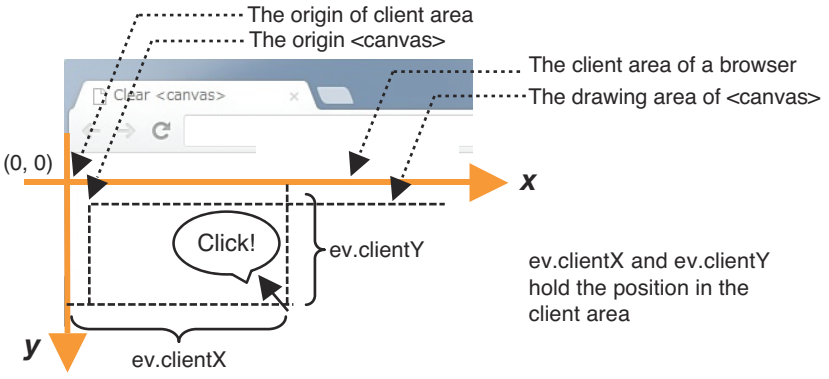

Figure 2.26 The coordinate system of a browser's client area and the position of the <canvas>

2. The coordinate system of the `<canvas>` is different from that of WebGL (see Figure 2.27) in terms of their origin and the direction of the y-axis.

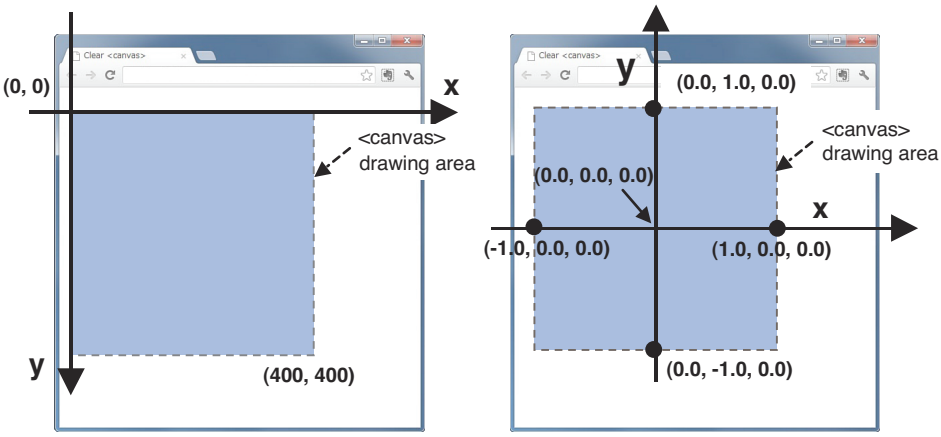

Figure 2.27 The coordinate system of <canvas> (left) and that of WebGL on <canvas> (right)

First you need to transform the coordinates from the browser area to the canvas, and then you need to transform them into coordinates that WebGL understands. Let's look at how to do that.

These transformations are performed at lines 56 and 57 in the sample program:

```
52    var x = ev.clientX;
53    var y = ev.clientY;
54    var rect = ev.target.getBoundingClientRect();
55    ...
56    x = ((x - rect.left) - canvas.width/2)/(canvas.width/2);
57    y = (canvas.height/2 - (y - rect.top))/(canvas.height/2);
```

Line 54 gets the position of the `<canvas>` in the client area. The `rect.left` and `rec.top` indicate the position of the origin of the `<canvas>` in the browser's client area (refer to Figure 2.26). So, `(x - rect.left)` at line 56 and `(y - rect.top)` at line 57 slide the position (x, y) in the client area to the correct position on the `<canvas>` element.

Next, you need to transform the `<canvas>` position into the WebGL coordinate system shown in Figure 2.27. To perform the transformation, you need to know the center position of the `<canvas>`. You can get the size of the `<canvas>` by `canvas.height` (in this case, 400) and `canvas.width` (in this case, 400). So, the center of the element will be `(canvas.height/2, canvas.width/2)`.

Next, you can implement this transformation by sliding the origin of the `<canvas>` into the origin of the WebGL coordinate system located at the center of `<canvas>`. The `((x - rect.left) - canvas.width/2)` and `(canvas.height/2 - (y -rect.top))` perform the transformation.

Finally, as shown in Figure 2.27, the range of the x-axis in the `<canvas>` goes from 0 to `canvas.width` (400), and that of the y-axis goes from 0 to `canvas.height` (400). Because

the axes in WebGL range from –1.0 to 1.0, the last step is to map the range of the `<canvas>` coordinate system to that of the WebGL coordinate system by dividing the x coordinate by `canvas.width/2` and the y coordinate by `canvas.height/2`. You can see this division in lines 56 and 57.

Line 59 stores the resulting position of the mouse click in the array `g_points` using the `push()` method, which appends the data to the end of the array:

```
59    g_points.push(x); g_points.push(y);
```

So every time a mouse click occurs, the position of the click is appended to the array, as shown in Figure 2.28. (The length of the array is automatically stretched.) Note that the index of an array starts from 0, so the first position is `g_points[0]`.

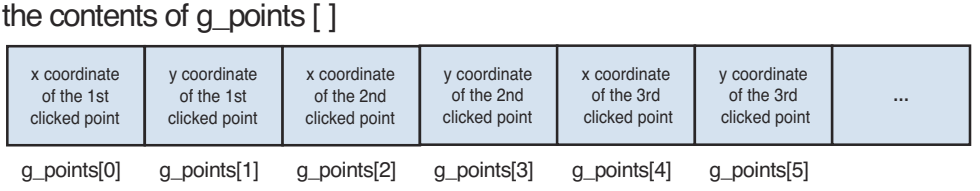

Figure 2.28 The content of g_points

You may be wondering why you need to store all the positions rather than just the most recent mouse click. This is because of the way WebGL uses the color buffer. You will remember from Figure 2.10 that in the WebGL system, first the drawing operation is performed to the color buffer, and then the system displays its content to the screen. After that, the color buffer is reinitialized and its content is lost. (This is the default behavior, which you'll investigate in the next section.) Therefore, it is necessary to save all the positions of the clicked points in the array, so that on each mouse click, the program can draw all the previous points as well as the latest. For example, the first point is drawn at the first mouse click. The first and second points are drawn at the second mouse click. The first, second, and third points are drawn on the third click, and so on.

Returning to the program, the `<canvas>` is cleared at line 62. After that, the `for` statement at line 65 passes each position of the point stored in `g_points` to `a_Position` in the vertex shader. Then `gl.drawArrays()` draws the point at that position:

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

Just like in `HelloPoint2.js`, you will use `gl.vertexAttrib3f()` to pass the point position to the attribute variable `a_Position`, which was passed as the fourth parameter of `click()` at line 51.

The array `g_points` holds the x coordinate and y coordinate of the clicked points, as shown in Figure 2.28. Therefore, `g_points[i]` holds the x coordinate, and `g_points[i+1]` holds the y coordinate, so the loop index `i` in the `for` statement at line 65 is incremented by 2 using the convenient `+` operator.

Now that you have completed the preparations for drawing the point, the remaining task is just to draw it using `gl.drawArrays()`:

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

Although it's a little complicated, you can see that the use of event handlers combined with attribute variable provides a flexible and generic means for user input to change what WebGL draws.

Experimenting with the Sample Program

Now let's experiment a little with this sample program `ClickedPoints`. After loading `ClickedPoints.html` in your browser, every time you click, it draws a point at the clicked position, as shown in Figure 2.26.

Let's examine what will happen when you stop clearing the `<canvas>` at line 62. Comment out the line as follows, and then reload the modified file into your browser.

```
61    // Clear <canvas>
62    // gl.clear(gl.COLOR_BUFFER_BIT);
63
64    var len = g_points.length;
65    for(var i = 0; i < len; i+=2) {
66      // Pass the position of a point to a_Position variable
67      gl.vertexAttrib3f(a_Position, g_points[i], g_points[i+1], 0.0);
68
69      // Draw a point
70      gl.drawArrays(gl.POINTS, 0, 1);
71    }
72  }
```

After running the program, you initially see a black background, but at the first click, you see a red point on a white background. This is because WebGL reinitializes the color buffer to the default value (0.0, 0.0, 0.0, 0.0) after drawing the point (refer to Table 2.1). The alpha component of the default value is 0.0, which means the color is transparent. Therefore, the color of the `<canvas>` becomes transparent, so you see the background color of the web page (white, in this case) through the `<canvas>` element. If you don't want this behavior, you should use `gl.clearColor()` to specify the clear color and then always call `gl.clear()` before drawing something.

Another interesting experiment is to look at how to simplify the code. In `ClickedPoints.js`, the x and y coordinates are stored separately in the `g_points` array. However, you can store the x and y coordinates as a group into the array as follows:

```
58    // Store the coordinates to a_points array          <- (1)
59    g_points.push([x, y]);
```

In this case, the new two-element array `[x, y]` is stored as an element in the array `g_points`. Conveniently, JavaScript can store an array in an array.

You can retrieve the x and y coordinates from the array as follows. First, you retrieve an element from the array by specifying its index (line 66). Because the element itself is an array containing (x, y), you can retrieve the first element and the second element as the x and y coordinate of each point from the array (line 67):

```
65  for(var i = 0; i < len; i++) {
66    var xy = g_points[i];
67    gl.vertexAttrib3f(a_Position, xy[0], xy[1], 0.0);
      ...
71  }
```

In this way, you can deal with the x and y coordinates as a group, simplifying your program and increasing the readability.

Change the Point Color

By now you should have a good feel for how the shaders work and how to pass data to them from your JavaScript program. Let's build on your understanding by constructing a more complex program that draws points whose colors vary depending on their position on the `<canvas>`.

You already studied how to change the color of a point when you looked at `HelloPoint1`. In that example, you modified the fragment shader program directly to embed the color value into the shader. In this section, let's construct the program so that you can specify the color of a point from JavaScript. This is similar to `HelloPoint2` earlier, where you passed the position of a point from a JavaScript program to the vertex shader. However, in this sample program, you need to pass the data to a "fragment shader," not to a vertex shader.

The name of the sample program is `ColoredPoints`. If you load the example into your browser, the result is the same as `ClickedPoints` except that each point's color varies depending on its position (see Figure 2.29).

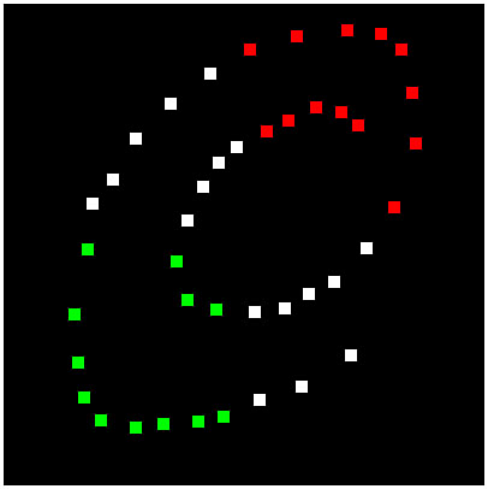

Figure 2.29 ColoredPoints

To pass data to a fragment shader, you can use a uniform variable and follow the same steps that you used when working with attribute variables. However, this time the target is a fragment shader, not a vertex shader:

1. Prepare the uniform variable for the color in the fragment shader.
2. Assign the uniform variable to the `gl_FragColor` variable.
3. Pass the color data to the uniform variable from the JavaScript program.

Let's look at the sample program and see how these steps are programmed.

Sample Program (ColoredPoints.js)

The vertex shader of this sample program is the same as in `ClickedPoints.js`. However, this time, the fragment shader plays a more important role because the program changes the color of the point dynamically and, as you will remember, fragment shaders handle colors. Listing 2.8 shows `ColoredPoints.js`.

Listing 2.8 ColoredPoints.js

```
1 // ColoredPoints.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
11 var FSHADER_SOURCE =
12   'precision mediump float;\n' +
13   'uniform vec4 u_FragColor;\n' + // uniform variable                    <- (1)
14   'void main() {\n' +
15   '  gl_FragColor = u_FragColor;\n' +                                    <- (2)
16   '}\n';
17
18 function main() {
     ...
29   // Initialize shaders
30   if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
     ...
33   }
34
35   // Get the storage location of a_Position variable
36   var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
     ...
42   // Get the storage location of u_FragColor variable
43   var u_FragColor = gl.getUniformLocation(gl.program, 'u_FragColor');
     ...
49   // Register function (event handler) to be called on a mouse press
50   canvas.onmousedown = function(ev){ click(ev, gl, canvas, a_Position,
                                                          ➥u_FragColor) };
     ...
56   gl.clear(gl.COLOR_BUFFER_BIT);
57 }
58
59 var g_points = [];  // The array for a mouse press
60 var g_colors = [];  // The array to store the color of a point
61 function click(ev, gl, canvas, a_Position, u_FragColor) {
62   var x = ev.clientX; // x coordinate of a mouse pointer
63   var y = ev.clientY; // y coordinate of a mouse pointer
64   var rect = ev.target.getBoundingClientRect();
65
66   x = ((x - rect.left) - canvas.width/2)/(canvas.width/2);
67   y = (canvas.height/2 - (y - rect.top))/(canvas.height/2);
68
69   // Store the coordinates to g_points array
70   g_points.push([x, y]);
71   // Store the color to g_colors array
72   if(x >= 0.0 && y >= 0.0) {           // First quadrant
73     g_colors.push([1.0, 0.0, 0.0, 1.0]);  // Red
```

```
74    } else if(x < 0.0 && y < 0.0) {        // Third quadrant
75      g_colors.push([0.0, 1.0, 0.0, 1.0]);  // Green
76    } else {                              // Others
77      g_colors.push([1.0, 1.0, 1.0, 1.0]);  // White
78    }
79
80    // Clear <canvas>
81    gl.clear(gl.COLOR_BUFFER_BIT);
82
83    var len = g_points.length;
84    for(var i = 0; i < len; i++) {
85      var xy = g_points[i];
86      var rgba = g_colors[i];
87
88      // Pass the position of a point to a_Position variable
89      gl.vertexAttrib3f(a_Position, xy[0], xy[1], 0.0);
90      // Pass the color of a point to u_FragColor variable
91      gl.uniform4f(u_FragColor, rgba[0],rgba[1],rgba[2],rgba[3]);          <-(3)
92      // Draw a point
93      gl.drawArrays(gl.POINTS, 0, 1);
94    }
95  }
```

Uniform Variables

You likely remember how to use an attribute variable to pass data from a JavaScript program to a vertex shader. Unfortunately, the attribute variable is only available in a vertex shader, and when using a fragment shader, you need to use a uniform variable. There is an alternative mechanism, a varying variable (bottom of Figure 2.30); however, it is more complex, so you won't use it until Chapter 5.

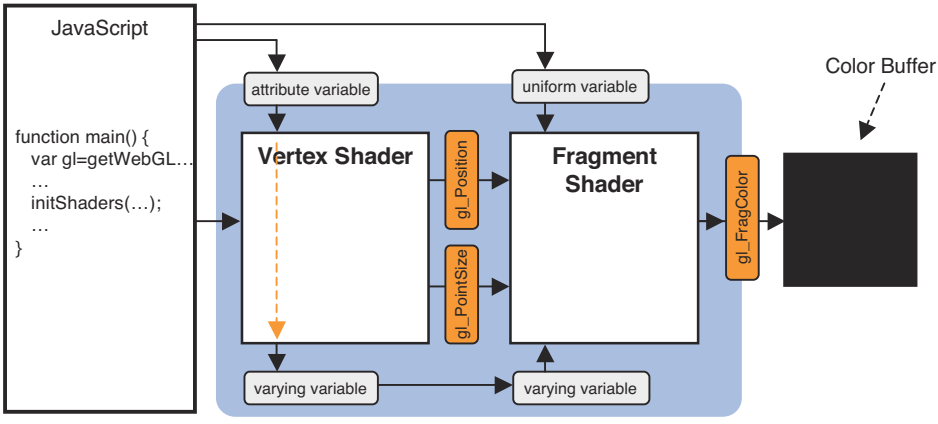

Figure 2.30 Two ways of passing a data to a fragment shader

When you were introduced to attribute variables, you saw that a uniform variable is a variable for passing "uniform" (nonvariable) data from a JavaScript program to all vertices or fragments in the shader. Let's now use that property.

Before using a uniform variable, you need to declare the variable using the form `<Storage Qualifier> <Type> <Variable name>` in the same way (see Figure 2.31) as declaring an attribute variable. (See the section "Sample Program (HelloPoint2.js).")[5]

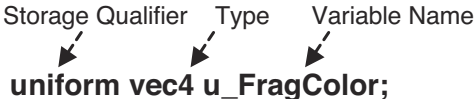

Figure 2.31 The declaration of the uniform variable

In this sample program, the uniform variable `u_FragColor` is named after the variable `gl_FragColor` because you will assign the uniform variable to the `gl_FragColor` later. The `u_` in `u_FragColor` is part of our programming convention and indicates that it is a uniform variable. You need to specify the same data type as `gl_FragColor` for `u_FragColor` because you can only assign the same type of variable to `gl_FragColor`. Therefore, you prepare `u_FragColor` at line 13, as follows:

```
10  // Fragment shader program
11  var FSHADER_SOURCE =
12    'precision mediump float;\n' +
13    'uniform vec4 u_FragColor;\n' + // uniform variable
14    'void main() {\n' +
15    '  gl_FragColor = u_FragColor;\n' +
16    '}\n';
```

Note that line 12 specifies the range and precision of variables by using a **precision qualifier**, in this case medium precision, which is covered in detail in Chapter 5.

Line 15 assigns the color in the uniform variable `u_FragColor` to `gl_FragColor`, which causes the point to be drawn in whatever color is passed. Passing the color through the uniform variable is similar to using an attribute variable; you need to retrieve the storage location of the variable and then write the data to the location within the JavaScript program.

Retrieving the Storage Location of a Uniform Variable

You can use the following method to retrieve the storage location of a uniform variable.

[5] In GLSL ES, you can only specify `float` data types for an attribute variable; however, you can specify any type for a uniform variable. (See Chapter 6 for more details.)

gl.getUniformLocation(program, name)

Retrieve the storage location of the uniform variable specified by the *name* parameter.

Parameters	program	Specifies the program object that holds a vertex shader and a fragment shader.
	name	Specifies the name of the uniform variable whose location is to be retrieved.
Return value	non-null	The location of the specified uniform variable.
	null	The specified uniform variable does not exist or its name starts with the reserved prefix `gl_` or `webgl_`.
Errors	INVALID_OPERATION	*program* has not been successfully linked (See Chapter 9.)
	INVALID_VALUE	The length of *name* is more than the maximum length (256 by default) of a uniform variable.

The functionality and parameters of this method are the same as in `gl.getAttribLocation()`. However, the return value of this method is `null`, not –1, if the uniform variable does not exist or its name starts with a reserved prefix. For this reason, unlike attribute variables, you need to check whether the return value is `null`. You can see this error checking line 44. In JavaScript, `null` can be treated as `false` when checking the condition of an `if` statement, so you can use the `!` operator to check the result:

```
42  // Get the storage location of uniform variable
43  var u_FragColor = gl.getUniformLocation(gl.program, 'u_FragColor');
44  if (!u_FragColor) {
45    console.log('Failed to get u_FragColor variable');
46    return;
47  }
```

Assigning a Value to a Uniform Variable

Once you have the location of the uniform variable and the WebGL method, `gl.uniform4f()` is used to write data to it. It has the same functionality and parameters as those of `gl.vertexAttrib[1234]f()`.

gl.uniform4f(location, v0, v1, v2, v3)

Assign the data specified by *v0*, *v1*, *v2*, and *v3* to the uniform variable specified by *location*.

Parameters	location	Specifies the storage location of a uniform variable to be modified.
	v0	Specifies the value to be used as the first element of the uniform variable.
	v1	Specifies the value to be used as the second element of the uniform variable.
	v2	Specifies the value to be used as the third element of the uniform variable.
	v3	Specifies the value to be used as the fourth element of the uniform variable.
Return value	None	
Errors	INVALID_OPERATION	There is no current program object.
		location is an invalid uniform variable location.

Let's look at the portion of the sample program where the `gl.uniform4f()` method is used to assign the data (line 91). As you can see, several processing steps are needed in advance:

```
71    // Store the color to g_colors array
72    if(x >= 0.0 && y >= 0.0) {              // First quadrant
73      g_colors.push([1.0, 0.0, 0.0, 1.0]);  // Red
74    } else if(x < 0.0 && y < 0.0) {         // Third quadrant
75      g_colors.push([0.0, 1.0, 0.0, 1.0]);  // Green
76    } else {                                // Other quadrants
77      g_colors.push([1.0, 1.0, 1.0, 1.0]);  // White
78    }
       ...
83    var len = g_points.length;
84    for(var i = 0; i < len; i++) {
85      var xy = g_points[i];
86      var rgba = g_colors[i];
          ...
91      gl.uniform4f(u_FragColor, rgba[0], rgba[1], rgba[2], rgba[3]);
```

To understand this program logic, let's return to the goal of this program, which is to vary the color of a point according to where on the `<canvas>` the mouse is clicked. In the first

quadrant, the color is set to red; in the third quadrant, the color is green; in the other quadrants, the color is white (see Figure 2.32).

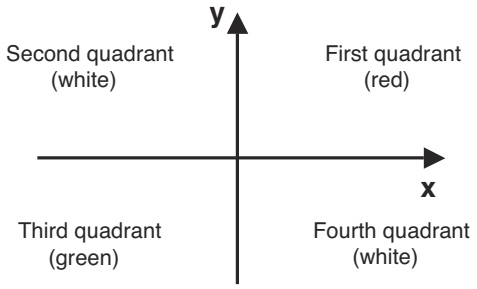

Figure 2.32 The name of each quadrant in a coordinate system and its drawing colors

The code from lines 72 to 78 simply determines which quadrant the mouse click was in and then, based on that, writes the appropriate color into the array `g_colors`. Finally, at line 84, the program loops through the points, passing the correct color to the uniform variable `u_FragColor` at line 91. This causes WebGL to write all the points stored so far to the color buffer, which then results in the browser display being updated.

Before finishing this chapter, let's take a quick look at the rest of the family methods for `gl.uniform[1234]f()`.

Family Methods of gl.uniform4f()

`gl.uniform4f()` also has a family of methods. `gl.uniform1f()` is a method to assign a single value (v0), `gl.uniform2f()` assigns two values (v0 and v1), and `gl.uniform3f()` assigns three values (v0, v1, and v2).

```
gl.uniform1f(location, v0)
gl.uniform2f(location, v0, v1)
gl.uniform3f(location, v0, v1, v2)
gl.uniform4f(location, v0, v1, v2, v3)
```

Assign data to the uniform variable specified by *location*. `gl.uniform1f()` indicates that only one value is passed, and it will be used to modify the first component of the uniform variable. The second and third components will be set to 0.0, and the fourth component will be set to 1.0. Similarly, `gl.uniform2f()` indicates that values are provided for the first two components, the third component will be set to 0.0, and the fourth component will be set to 1.0. `gl.uniform3f()` indicates that values are provided for the first three components and the fourth component will be set to 1.0, whereas `gl.unifrom4f()` indicates that values are provided for all four components.

Parameters	location	Specifies the storage location of a uniform variable.
	v0, v1, v2, v3	Specifies the values to be assigned to the first, second, third, and fourth component of the uniform variable.
Return value	None	
Errors	INVALID_OPERATION	There is no current program object.
		location is an invalid uniform variable location.

Summary

In this chapter, you saw the core functions of WebGL and how to use them. In particular, you learned about shaders, which are the main mechanism used in WebGL to draw graphics. Based on this, you constructed several sample programs starting with a simple program to draw a red point, and then you added complexity by changing its position based on a mouse click and changed its color. In both cases, these examples helped you understand how to pass data from a JavaScript program to the shaders, which will be critical for future examples.

The shapes found in this chapter were still just two-dimensional points. However, you can apply the same knowledge of the use of the core WebGL functions and shaders to more complex shapes and to three-dimensional objects.

A key learning point in this chapter is this: A vertex shader performs per-vertex operations, and a fragment shader performs per-fragment operations. The following chapters explain some of the other functions of WebGL, while slowly increasing the complexity of the shapes that you manipulate and display on the screen.