CHAPTER 2

Working with HTML5 and WebGL

After completing this chapter, you will be able to:

- Draw a simple constant color square with WebGL
- Create a new JavaScript source code file for your simple game engine
- Define new `Singleton`-like JavaScript objects to implement core game engine functionality
- Appreciate the importance of abstraction and organize your source code structure to support growth in complexity

Introduction

Drawing is one of the most essential functionalities common to all video games. A game engine should offer a flexible and programmer-friendly interface to its drawing system. In this way, when building a game, the designers and developers can focus on the important aspects of the game, such as mechanics, logic, and aesthetics.

WebGL is a modern graphical application programming interface (API) that offers quality and efficiency via direct access to the graphical hardware. For these reasons, WebGL can serve as an excellent base to support drawing in a game engine, especially for video games that are designed to be played across the Internet.

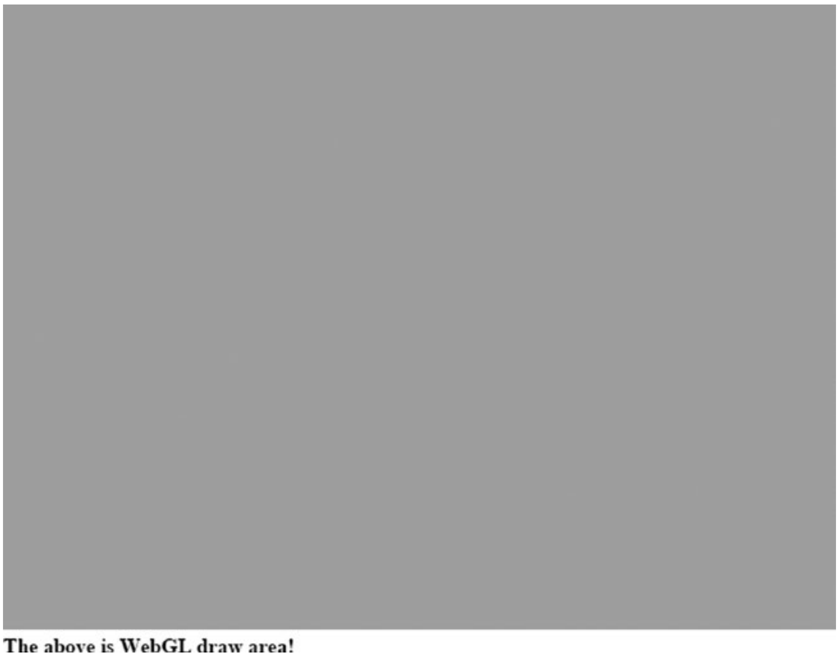
This chapter examines the fundamentals of drawing with WebGL, designs abstractions to encapsulate irrelevant details to facilitate easy programming, and builds the foundational infrastructure to organize a complex source code system to support future expansion.

Canvas for Drawing

To draw, you must first define and dedicate an area within the web page. We will begin with using the HTML `canvas` element to define an area for WebGL drawing.

The HTML5 Canvas Project

This project demonstrates how to draw and clear a `canvas` element on a web page. Figure 2-1 shows an example of running this project, which is defined in the `Chapter2/2.1.HTML5Canvas` folder.



The above is WebGL draw area!

Figure 2-1. *Running the HTML5 Canvas project*

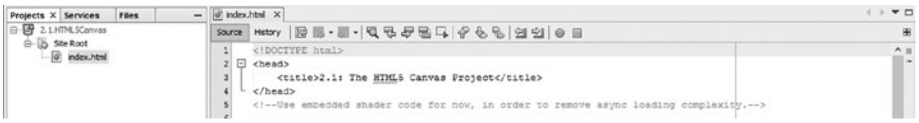
The goals of the project are as follows:

- To learn how to set up the HTML `canvas` element
- To learn how to retrieve the `canvas` element from an HTML document for use in JavaScript
- To learn how to create a reference context to WebGL from the retrieved `canvas` element and manipulate the canvas from the WebGL context

Creating and Clearing the HTML Canvas

In this first project, you will create an empty HTML5 canvas and clear the canvas to a specific color with WebGL.

1. Create a new HTML5 project titled HTML5 Canvas.
2. Open the `index.html` file in the editor by double-clicking the project name (HTML5Canvas) in the Project view, then double-clicking `Site Root`, and lastly double-clicking the `index.html` file, as illustrated in Figure 2-2.



```
<!DOCTYPE html>
<head>
    <title>2.1: The HTML5 Canvas Project</title>
</head>
<!--Use embedded shader code for now, in order to remove async loading complexity.-->
```

Figure 2-2. *Editing the index.html file in your project*

3. Create the HTML canvas for drawing by adding the following line in the `index.html` file within the body element:

```
<canvas id="GLCanvas" width="640" height="480">
        Your browser does not support the HTML5 canvas.
</canvas>
```

The code defines a `canvas` element named `GLCanvas` with the specified `width` and `height` attributes. As you will experience later, you will retrieve the reference to the `GLCanvas` to draw into this area. The text inside the element will be displayed if your browser does not support drawing with `canvas`.

■ **Note** The lines between the `<body>` and `</body>` tags are referred to as "within the `body` element." For the rest of this book, "within the `AnyTag` element" will be used to refer to any line between the beginning (`<AnyTag>`) and end (`</AnyTag>`) of the element.

---

4. Create a `script` element for the inclusion of JavaScript programming code, once again within the body element.

```
<script type="text/javascript">
    // JavaScript code goes here.
</script>
```

This takes care of the HTML portion of this example. Now you will write JavaScript for the remainder of the example.

5. Retrieve a reference to the `GLCanvas` in your JavaScript by adding the following line within the `script` element:

```
var canvas = document.getElementById("GLCanvas");
```

The code creates a new variable named `canvas` and stores a reference to the `GLCanvas` drawing area in this variable.

■ **Note** All local variable names begin with a lowercase letter, as in `canvas`.

---

6. Retrieve and bind a reference to the WebGL context to the drawing area by adding the following code:

```
var gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
```

As the code indicates, the retrieved reference to the WebGL context is stored in the local variable named `gl`. From this variable, you have access to all WebGL functionality.

7. Clear the canvas drawing area to your favorite color through WebGL by adding the following:

```
if (gl !== null) {
    gl.clearColor(0.0, 0.8, 0.0, 1.0);  // set the color to be cleared
    gl.clear(gl.COLOR_BUFFER_BIT);   // clear the colors
}
```

This code checks to ensure the WebGL context is properly retrieved, sets the clear color, and clears the drawing area. Note that the clearing color is given in RGBA format, with floating-point values ranging from 0.0 to 1.0. The fourth number in the RGBA format is the alpha channel. You will learn more about the alpha channel in the later chapters. For now, always assign 1.0 to the alpha channel.

You can refer to the final source code in the `index.html` file in the `Chapter2/2.1.HTML5Canvas` project. Run the project, and you should see a light green area on your browser window. This is the 640×480 canvas drawing area you defined.

You can try changing the cleared color to white by setting the RGBA of `gl.clearColor()` to 1 or to black by setting the color to 0 and leaving the alpha value 1. Notice that if you set the alpha channel to 0, the canvas color will disappear. This is because a 0 value in the alpha channel represents complete transparency, and thus you will "see through" the canvas and observe the background color of the web page. You can also try altering the resolution of the canvas by change the 640×480 value to any number you fancy. Notice that these two numbers refer to the pixel counts and thus must always be integers.

Separating HTML and JavaScript

In the previous project you created an HTML `canvas` element and cleared the area defined by the canvas using WebGL. Notice that all the functionality is clustered in the `index.html` file. As the project complexity increases, this clustering of functionality can quickly become unmanageable and negatively impact the programmability of your system. For this reason, throughout the development process in this book, after a concept is introduced, efforts will be spent on separating the associated source code into either well-defined source code files or object-oriented programming. To begin this process, the HTML and JavaScript source code from the previous project will be separated into different source code files.

The JavaScript Source File Project

This project demonstrates how to logically separate the source code into appropriate files. This is accomplished by creating a separate JavaScript source code file named `WebGL.js` to implement the corresponding functionality in the `index.html` file. The web page will load the JavaScript source code as instructed by the code in the `index.html` file. As illustrated in Figure 2-3, this project looks identical as the previous project when running. The source code of this project is located in the `Chapter2/2.2.JavaScript SourceFile` folder.

The above is WebGL draw area!

Figure 2-3. *Running the JavaScript Source File project*

The goals of the project are as follows:

- To learn how to separate source code into different files
- To organize your code in a logical structure

Separate JavaScript Source Code File

This section details how to create and edit a new JavaScript source code file. You should familiarize yourself with this process because you'll create numerous source code files throughout this book.

1. Create a new HTML5 project titled JavaScriptSourceFile.
2. Create a new folder named `src` inside the `Site Root` folder by right-clicking and creating a new folder, as illustrated in Figure 2-4.

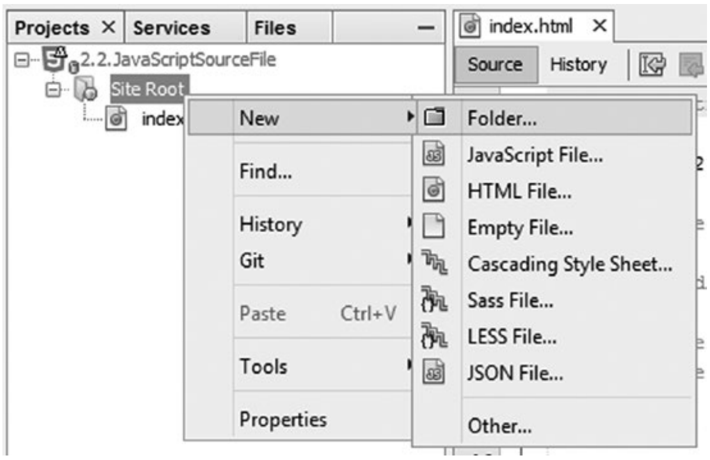


Figure 2-4. *Creating a new source code folder*

This folder will contain all of your source code.

■ **Note** In NetBeans you can create new folders, create new files, copy/paste projects, and rename projects by using the right-click menus in the Projects window.

---

3. Create a new source code file within the `src` folder by right-clicking the `src` folder, as illustrated in Figure 2-5.

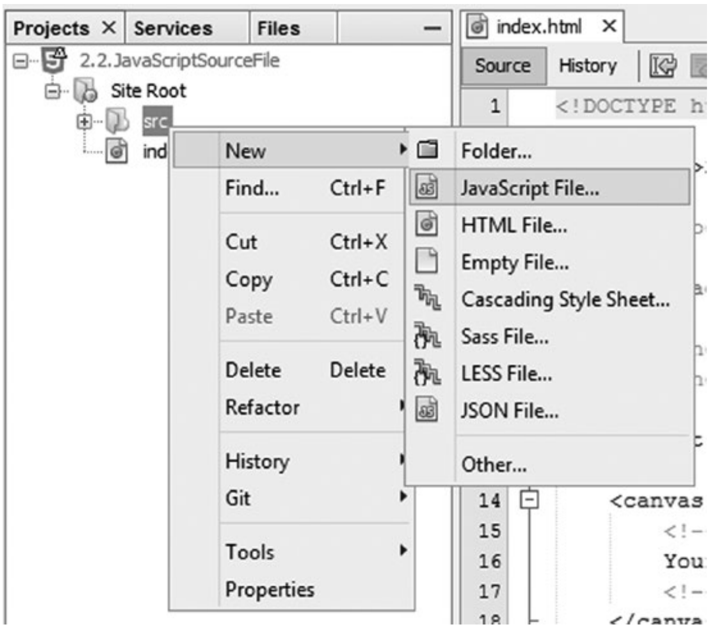


Figure 2-5. *Adding a new JavaScript source code file*

Name the new source file `WebGL.js`.

4. Open the new `WebGL.js` source file for editing.
5. Create a global variable referencing the WebGL context.

```
"use strict";
var gGL = null;
```

■ **Note** All global variable names begin with a lowercase *g*, as in `gGL`.

6. Define the `initializeGL()` function to retrieve `GLCanvas`, bind the drawing area with the WebGL context, and store the results in the global `gGL` variable.

```
function initializeGL() {
    var canvas = document.getElementById("GLCanvas");

    gGL = canvas.getContext("webgl") ||
          canvas.getContext("experimental-webgl");
```

```
        if (gGL !== null) {
            gGL.clearColor(0.0, 0.8, 0.0, 1.0);  // set the color to be cleared
        } else {
            document.write("<br><b>WebGL is not supported!</b>");
        }
    }
```

■ **Note** All public function names begin with a lowercase letter, as in `initializeGL()`.

Notice this function is similar to the `JavaScript` source code you typed in the previous project.

7. Define the `clearCanvas()` function to invoke the WebGL context to clear the canvas drawing area.

```
function clearCanvas() {
    gGL.clear(gGL.COLOR_BUFFER_BIT);     // clear to the color previously set
}
```

8. Define the `doGLDraw()` function to carry out the initialization and clearing of the canvas area.

```
function doGLDraw() {
    initializeGL();
    clearCanvas();
}
```

Load and Run JavaScript Source Code from index.html

With all the JavaScript functionality defined in the `WebGL.js` file, you now need to load this file and invoke the `doGLDraw()` function from your web page, the `index.html` file.

1. Open the `index.html` file for editing.
2. Create the HTML canvas, `GLCanvas`, as in the previous project.
3. Load the `WebGL.js` source code by including the following code within the `body` element:

```
<script type="text/javascript" src="src/WebGL.js"></script>
```

You can include this line either before or after the definition of canvas, as long as it is within the `body` element.

4. Execute the `doGLDraw()` function after `WebGL.js` is loaded.

```
<body onload="doGLDraw();">
```

The modification to the body tag says once all loading operations are done, the `doGLDraw()` function should be executed.

You can refer to the final source code in the WebGL.js and index.html files in the Chapter2/2.2.JavaScriptSourceFile project. Although the output from this project is identical to that from the previous project, the organization of your code will allow you to expand, debug, and understand the game engine as you continue to add new functionality.

Observations

Examine your index.html file closely and compare its content to the same file from the previous project. You will notice that the index.html file from the previous project contains two types of information (HTML and JavaScript code) and that the same file from this project contains only the former, with all JavaScript code being moved to WebGL.js. This clean separation of information allows for easy understanding of the source code and improves support for complex systems. From this point on, all JavaScript source code will be added to separate source code files. In all cases, in the same manner as you have included the loading of WebGL.js, you will have to remember to load the new source code, such as NewSourceFile.js, by including the following line within the body element of the index.html file:

```
<script type="text/javascript" src="src/NewSourceFile.js"></script>
```

Elementary Drawing with WebGL

Drawing with WebGL is a multiple-step process that involves transferring geometric data and OpenGL Shading Language (GLSL) instructions (the shaders) from memory to the drawing hardware, or the graphical processing unit (GPU). This process involves a significant number of WebGL function calls. This section presents the WebGL drawing steps in detail. It is important to focus on learning these basic steps and avoid being distracted by the less important WebGL configuration nuances such that you can continue to learn the overall concepts and build the game engine.

In the following project, you will learn about drawing with WebGL by focusing on the most elementary operations, including the loading to the GPU for the simple geometry of a square, a constant color shader, and basic instructions of drawing a square as two triangles.

The Draw One Square Project

This project leads you through the steps required to draw a single square on the canvas. Figure 2-6 shows an example of running this project, which is defined in the Chapter2/2.3.DrawOneSquare folder.

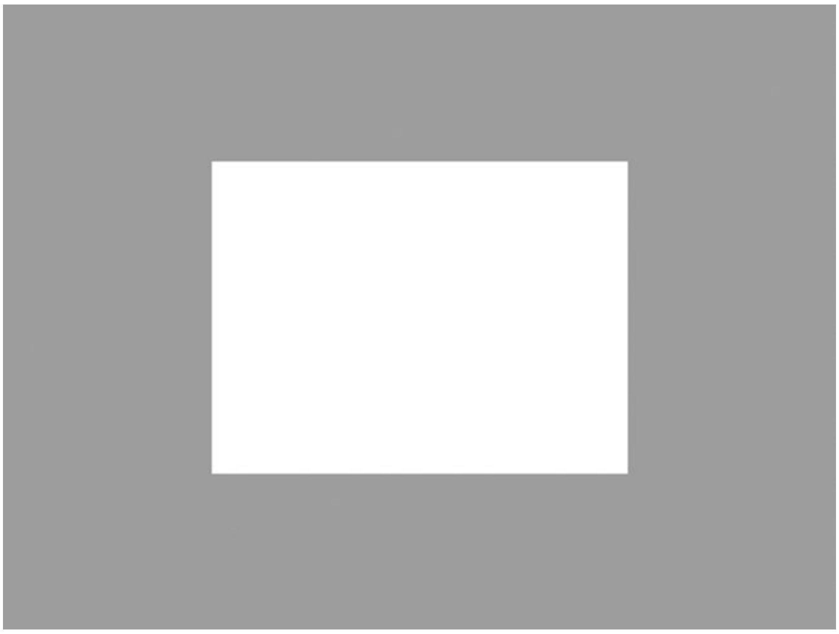


Figure 2-6. *Running the Draw One Square project*

The goals of the project are as follows:

- To understand how to load geometric data to the GPU
- To learn about simple GLSL shaders for drawing with WebGL
- To learn how to compile and load shaders to the GPU
- To understand the steps to draw with WebGL

Set Up and Load the Primitive Geometry Data

To draw efficiently with WebGL, the data associated with the geometry to be drawn, such as the vertex positions of a square, should be stored in the GPU hardware. In the following steps, you will create a contiguous buffer in the GPU, load the vertex positions of a unit square into the buffer, and store the reference to the GPU buffer in a global variable. Learning from the previous project, the corresponding JavaScript code will be stored in a new source code file, `VertexBuffer.js`.

■ **Note** A unit square is a 1×1 square centered at the origin.

---

1. Create a new JavaScript source file in the `src` folder and name it `VertexBuffer.js`.

2. Declare a global variable gSquareVertexBuffer to store the reference to the WebGL buffer location.

```
"use strict";
var gSquareVertexBuffer = null;
```

3. Define the initSquareBuffer() function to create and load vertices onto the GPU.

```
function initSquareBuffer() {

    // First: define the vertices for a square
    var verticesOfSquare = [
        0.5, 0.5, 0.0,
        -0.5, 0.5, 0.0,
        0.5, -0.5, 0.0,
        -0.5, -0.5, 0.0
    ];

    // Step A: Create a buffer on the gGL context for our vertex positions
    gSquareVertexBuffer = gGL.createBuffer();

    // Step B: Activate vertexBuffer
    gGL.bindBuffer(gGL.ARRAY_BUFFER, gSquareVertexBuffer);

    // Step C: Loads verticesOfSquare into the vertexBuffer
    gGL.bufferData(gGL.ARRAY_BUFFER, new Float32Array(verticesOfSquare),
    gGL.STATIC_DRAW);
}
```

In the code shown, the vertices of a unit square are defined first. Notice the z-dimension is set to 0.0 because you are building a 2D game engine. Step A creates a buffer on the GPU for storing the vertex positions of the square and stores the reference to the GPU buffer in the global variable gSquareVertexBuffer. Step B activates the newly created buffer, and step C loads the vertex position of the square into the activated buffer. The keyword STATIC_DRAW informs the drawing hardware that this buffer will not be changed.

■ **Tip** Remember that the gGL global variable is defined in the WebGL.js file and initialized by the initializedGL() function.

---

4. Lastly, remember to load the VertexBuffer.js source code in your web page by adding the following code within the body element of the index.html file:

```
<script type="text/javascript" src="src/VertexBuffer.js"></script>
```

■ **Tip** From now on, when you are reminded to “load the new source file in index.html,” you should add the script line (remember to change VertexBuffer.js to your new source code file name) into index.html.

With the functionality of loading vertex positions defined, you are now ready to define and load the GLSL shaders.

Set Up the GLSL Shaders

The term *shader* refers to programs that run on the GPU. In the context of the game engine, shaders must always be defined in pairs consisting of a vertex shader and a corresponding fragment shader. The GPU will execute the vertex shader once per primitive vertex and the fragment shader once per pixel covered by the primitive. For example, you can define a square with four vertices and display this square to cover a 100×100 pixel area. To draw this square, WebGL will invoke the vertex shader 4 times (once for each vertex) and execute the fragment shader 10,000 times (once for each of the 100×100 pixels)!

In the case of WebGL, both the vertex and fragment shaders are implemented in the OpenGL Shading Language (GLSL). GLSL is a language with syntax that is similar to the C programming language and designed specifically for processing and displaying graphical primitives. You will learn sufficient GLSL to support the drawing for the game engine when required.

In the following steps, you will load into memory the source code for both vertex and fragment shaders, compile and link them into a single shader program, and load the compiled program into the GPU. In this project, the shader source code is defined in the `index.html` file, while the loading, compiling, and linking of the shaders are defined in the `ShaderSupport.js` source file.

■ **Note** The WebGL context can be considered as an abstraction of the GPU hardware. To facilitate readability, the two terms *WebGL* and *GPU* are sometimes used interchangeably.

Define the Vertex and Fragment Shaders

GLSL shaders are simply programs consisting of GLSL instructions.

1. Define the vertex shader by opening the `index.html` file, and within the body element, add the following code:

```
<script type="x-shader/x-vertex" id="VertexShader">
    attribute vec3 aSquareVertexPosition;
    void main(void) {
        gl_Position = vec4(aSquareVertexPosition, 1.0);
    }
</script>
```

The `script` element type is set to `x-shader/x-vertex` because that is a common convention for shaders. As you will see, the `id` field with the value `VertexShader` allows you to identify and load this vertex shader into memory.

The GLSL `attribute` keyword identifies per-vertex data that will be passed to the vertex shader in the GPU. In this case, the `aSquareVertexPosition` attribute is of data type `vec3` or an array of three floating-point numbers. As you will see in later steps, `aSquareVertexPosition` will contain vertex positions for a square.

The `gl_Position` is a GLSL built-in variable, specifically, an array of four floating-point numbers that must contain the vertex position. In this case, the fourth position of the array will always be 1.0. The code shows the shader converting the `aSquareVertexPosition` into a `vec4` and passing the information to WebGL.

2. Define the fragment shader in index.html by adding the following code within the body element:

```
<script type="x-shader/x-fragment" id="FragmentShader">
    void main(void) {
        gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
    }
</script>
```

Note the different type and id fields. Recall that the fragment shader is invoked once per pixel. The variable gl_FragColor is the built-in variable that determines the color of the pixel. In this case, a color of (1,1,1,1), or white, is returned. This means all pixels covered will be shaded to a constant white color.

With both the vertex and fragment shaders defined in the index.html file, you are now ready to implement the functionality to compile, link, and load the resulting shader program to the GPU.

Compile, Link, and Load the Vertex and Fragment Shaders

To maintain source code in logically separated source files, you will create shader support functionality in a new source code file, ShaderSupport.js.

1. Create a new JavaScript file, ShaderSupport.js.
2. Create two global variables, gSimpleShader and gShaderVertexPositionAttribute, to store the reference to the shader program and the vertex position attribute in the GPU.

```
var gSimpleShader = null;
var gShaderVertexPositionAttribute = null;
```

3. Create a function to load and compile a shader from index.html.

```
function loadAndCompileShader(id, shaderType) {
    var shaderText, shaderSource, compiledShader;

    // Step A: Get the shader source from index.html
    shaderText = document.getElementById(id);
    shaderSource = shaderText.firstChild.textContent;

    // Step B: Create the shader based on the source type: vertex or fragment
    compiledShader = gGL.createShader(shaderType);

    // Step C: Compile the created shader
    gGL.shaderSource(compiledShader, shaderSource);
    gGL.compileShader(compiledShader);

    // Step D: check for error and return result
    if (!gGL.getShaderParameter(compiledShader, gGL.COMPILE_STATUS)) {
        alert("A shader compiling error occurred: " +
            gGL.getShaderInfoLog(compiledShader));
    }
    return compiledShader;
}
```

Step A of the code finds shader source code from the `index.html` file using the `id` field you specified when defining the shaders, either `VertexShader` or `FragmentShader`. Step B creates a specified shader (either vertex or fragment) in the GPU. Step C specifies the shader source code and compiles the shader. Finally, step D checks and returns the reference to the compiled shader where an error will result in a `null` value.

4. You are now ready to create and compile a shader program by defining the `initSimpleShader` function.

```
function initSimpleShader(vertexShaderID, fragmentShaderID) {
    // Step A: load and compile the vertex and fragment shaders
    var vertexShader = loadAndCompileShader(vertexShaderID, gGL.VERTEX_SHADER);
    var fragmentShader = loadAndCompileShader(fragmentShaderID,
        gGL.FRAGMENT_SHADER);

    // Step B: Create and link the shaders into a program.
    gSimpleShader = gGL.createProgram();
    gGL.attachShader(gSimpleShader, vertexShader);
    gGL.attachShader(gSimpleShader, fragmentShader);
    gGL.linkProgram(gSimpleShader);

    // Step C: check for error
    if (!gGL.getProgramParameter(gSimpleShader, gGL.LINK_STATUS))
        alert("Error linking shader");

    // Step D: Gets a reference to the aSquareVertexPosition attribute
    gShaderVertexPositionAttribute = gGL.getAttribLocation(gSimpleShader,
        "aSquareVertexPosition");

    // Step E: Activates the vertex buffer loaded in VertexBuffer.js
    gGL.bindBuffer(gGL.ARRAY_BUFFER, gSquareVertexBuffer);
    // Step F: Describe the characteristic of the vertex position attribute
    gGL.vertexAttribPointer(gShaderVertexPositionAttribute,
        3,          // each vertex element is a 3-float (x,y,z)
        gGL.FLOAT,  // data type is FLOAT
        false,      // if the content is normalized vectors
        0,          // number of bytes to skip in between elements
        0);         // offsets to the first element
}
```

Step A of the code compiles the shader code you defined in `index.html` by calling the `loadAndCompileShader()` function with the corresponding parameters. Step B loads the compiled shader onto the GPU and links the two shaders into a program. The reference to this program is stored in the global variable `gSimpleShader`. After error checking in step C, step D locates and stores the reference to the `aSquareVetexPosition` attribute defined in your vertex shader. Step E activates the vertex buffer you loaded in `VertexBuffer.js`, and step F connects the activated buffer to the `aSquareVertexPosition` attribute by describing the data format of the vertex buffer, where each vertex position is a three-float (x, y, z) position.

5. Finally, as with any new source code file, remember to load the `ShaderSupport.js` file the `index.html` file.

The shader loading and compiling functionality is now defined. You can now activate these functions to draw with WebGL.

Set Up Drawing with WebGL

With the vertex data and shaders functionality defined, you can now execute the following steps to draw with WebGL. Recall from the previous project that the initialization and drawing code is stored in the `WebGL.js` file. Now open this file for editing.

1. Modify the `initializeGL()` function to include the initialization of the vertex buffer and the shader program.

```
function initializeGL() {
    var canvas = document.getElementById("GLCanvas");
    gGL = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");

    if (gGL !== null) {
        gGL.clearColor(0.0, 0.8, 0.0, 1.0);  // set the color to be cleared

        // A. initialize the vertex buffer
        initSquareBuffer();  // This function is defined VertexBuffer.js

        // B. now load and compile the vertex and fragment shaders
        initSimpleShader("VertexShader", "FragmentShader");
                // the two shaders are defined in the index.html file
                // initSimpleShader() function is defined in ShaderSupport.js
    } else {
        document.write("<br><b>WebGL is not supported!</b>");
    }
}
```

The code in bold shows you should modify the `initializeGL()` function to call the `initSquareBuffer()` and `initSimpleShader()` functions after successfully obtaining the WebGL context.

2. Replace the `clearCanvas()` function with the `drawSquare()` function for drawing the defined square.

```
function drawSquare() {
    gGL.clear(gGL.COLOR_BUFFER_BIT);

    // Step A: Activate the shader to use
    gGL.useProgram(gSimpleShader);

    // Step B: Enable the vertex position attribute
    gGL.enableVertexAttribArray(gShaderVertexPositionAttribute);

    // Step C: Draw with the above settings
    gGL.drawArrays(gGL.TRIANGLE_STRIP, 0, 4);
}
```

This code shows the steps to draw with WebGL. Step A activates the shader program to use. Step B enables the vertex attribute for the vertex shader. Finally, step C issues the draw command. In this case, you are issuing a command to draw the four vertices as two connected triangles that form a square.

3. Lastly, modify `doGLDraw()` to call the `drawSquare()` function.

```
function doGLDraw() {
    initializeGL();        // Binds gGL context to WebGL functionality
    drawSquare();          // Clears the GL area and draws one square
}
```

Recall that `doGLDraw()` is the function called by `index.html` after all source code files are completely loaded. For this reason, WebGL will be initialized and the white square drawn. For reference, you can refer to the source code in the `Chapter2/2.3.DrawOneSquare` project.

Observations

Run the project and you will see a white rectangle on a green canvas. What happened to the square? Remember that the vertex position of your 1×1 square was defined at locations (±0.5, ±0.5). Now observe the project output: the white rectangle is located in the middle of the green canvas covering exactly half of the canvas's width and height. As it turns out, WebGL draws vertices within the ±1.0 range onto the entire defined drawing area. In this case, the ±1.0 in the x-dimension is mapped to 640 pixels, while the ±1.0 in the y-dimension is mapped to 480 pixels (the created canvas dimension is 640×480); the 1x1 square is drawn onto a 640x480 area, or an area with an aspect ratio of 4:3. Since the 1:1 aspect ratio of the square does not match the 4:3 aspect ratio of the display area, the square shows up as a 4:3 rectangle. This problem will be resolved later in this chapter.

You can try editing the fragment shader in `index.html` by changing the color set in the `gl_FragColor` function to change the color of the white square. Notice that a value of less than 1 in the alpha channel will result in the white square becoming transparent and showing through some of the greenish canvas color.

Finally, note that this project defines many global variables with little attempt at hiding information. This organization does not lend itself to supporting changes in functionality or growth in complexity. In the next sections, you will encapsulate and abstract portions of this example to form the basis of the game engine framework.

Abstraction with JavaScript Objects

The previous project decomposed the drawing of a square into logical modules and implemented the modules as files containing global functions and variables. In software engineering, this solution process is referred to as *functional decomposition*, and the implementation is referred to as *procedural programming*. Procedural programming produces solutions that are well-structured, easy to understand, and often fast to create. This is why it is often used to prototype a concept or to learn new techniques.

This project enhances the Draw One Square solution with object-oriented analysis and programming to introduce data abstraction. As additional concepts are introduced and as the game engine complexity grows, proper data abstraction supports straightforward design and code reuse through inheritance.

The JavaScript Objects Project

This project demonstrates how to abstract the global functions and variables from the Draw One Square project into JavaScript objects. This objected-oriented abstraction will result in a framework that offers manageability and expandability for subsequent projects. As illustrated in Figure 2-7, when running, this project displays a white rectangle in a greenish canvas, identical to that from the Draw One Square project. The source code to this project is defined in the `Chapter2/2.4.JavaScriptObjects` folder.

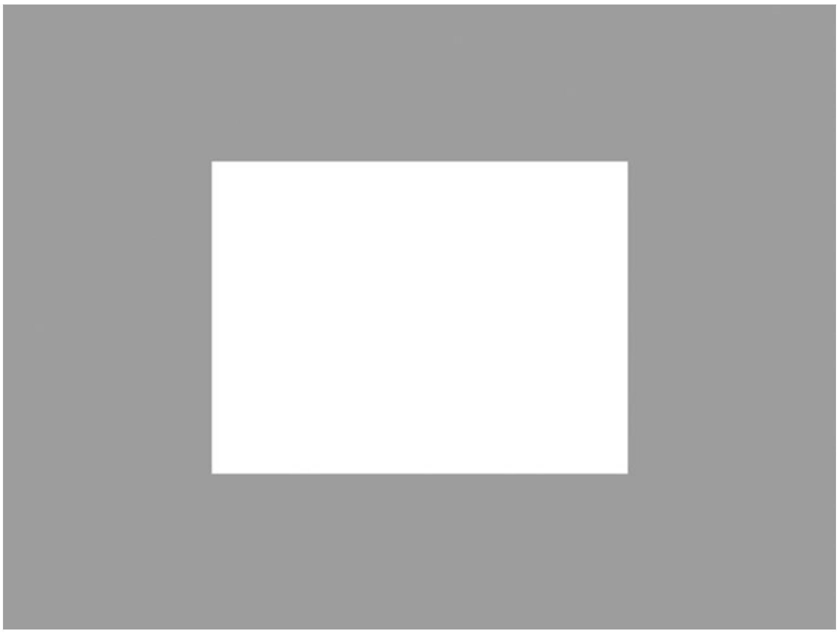


Figure 2-7. *Running the JavaScript Objects project*

The goals of the project are as follows:

- To separate the game engine from the game logic code
- To demonstrate the implementation of a `Singleton`-like object based on the JavaScript Module pattern
- To understand how to build abstractions with JavaScript objects

The steps for creating the project are as follows:

1. Create separate folders to organize the source code for the game engine and the logic of the game.
2. Define JavaScript objects to abstract the game engine functionality: `Core`, `VertexBuffer`, and `SimpleShader`. These objects will be defined in corresponding JavaScript source code files.
3. Define a JavaScript object to implement the drawing of one square, which is the logic of your simple game for now.

Source Code Organization

Create a new HTML5 project with the NetBeans IDE with a source code folder named src. Within src, create Engine and MyGame as subfolders, as illustrated in Figure 2-8.

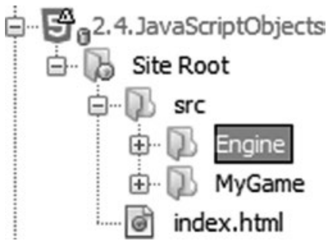


Figure 2-8. *Creating Engine under the src folder*

The src/Engine folder will contain all the source code to the game engine, and the `src/MyGame` folder will contain the source for the logic of your game. It is important to organize source code diligently because the complexity of the system and the number of files will increase rapidly as more concepts are introduced. A well-organized source code structure facilitates understanding and expansion.

■ **Tip** The source code in the `MyGame` folder implements the game by relying on the functionality provided by the game engine defined in the `Engine` folder. For this reason, in this book, the source code in the `MyGame` folder is often referred to as the *client* of the game engine.

Abstracting the Game Engine

A completed game engine would include many self-contained subsystems to fulfill different responsibilities. For example, you may be familiar with or have heard of the geometry subsystem for managing the geometries to be drawn, the resource management subsystem for managing images and audio clips, the physics subsystem for managing object interactions, and so on. In most cases, the game engine would include one unique instance of each of these subsystems, that is, one instance of the geometry subsystem, of the resource management subsystem, of the physics subsystem, and so on.

These subsystems will be covered in later chapters of this book. This section focuses on establishing the mechanism and organization for implementing these single-instance or `Singleton`-like objects based on the JavaScript Module pattern.

The Core of the Game Engine: gEngine.Core

The core of the game engine contains the common functionality shared by the entire system. This can include one-time initialization of the WebGL (or GPU), shared resources, utility functions, and so on.

1. Create a new source file in the `src/Engine` folder and name the file `Engine_Core.js`. Remember to load this new source file in `index.html`.

2. At the beginning of the file, create `gEngine`, a new global variable representing the core of the game engine as follows:

```
"use strict";  // Operate in Strict mode

var gEngine = gEngine || { };
    // initialize the variable while ensuring it is not redefined
```

This line instructs the system to retain a current gEngine variable if it is defined; otherwise, it creates a new instance with an initial empty object. Different subsystems of the game engine will be implemented as distinct properties of the gEngine object in separate files. This line ensures that properties defined in separate source files will be retained independent from the order upon which they are loaded.

3. Define the gEngine.Core property as follows:

```
gEngine.Core = (function() {
    // instance variable: the graphical context for drawing
    var mGL = null;

    // Accessor of the webgl context
    var getGL = function() { return mGL; };

    // Contains the functions and variables that will be accessible.
    var mPublic = {
        getGL: getGL
    };

    return mPublic;
}());
```

■ **Note** All instance variable names begin with an *m* and are followed by a capital letter, as in `mVariable`. Though not enforced by JavaScript, you should never access an instance variable from outside the object. For example, you should never access `gEngine.Core.mGL` directly; instead, call the `gEngine.Core.getGL()` function to access the variable.

The code defines the gEngine.Core property as a global object. This object contains one private instance variable and a public accessor. It is important to note the following about the syntax:

a. The mPublic object defines public methods for gEngine.Core. In this case, getGL() is the only public method of the gEngine.Core object.

b. The () at the end of the function block signals to the system to execute the function immediately, and thus only a single instance of gEngine.Core will be created.

c. This is the JavaScript Module pattern, which will be used to implement all the subsystems in the game engine.

4. Inside the gEngine.Core object, add a function to initialize the WebGL context from the HTML canvas ID.

```
// initialize the WebGL, the vertex buffer and compile the shaders
var initializeWebGL = function(htmlCanvasID) {
    var canvas = document.getElementById(htmlCanvasID);
```

```
        // Get the standard or experimental webgl and binds to the Canvas area
        // store the results to the instance variable mGL
        mGL = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");

        if (mGL === null) {
            document.write("<br><b>WebGL is not supported!</b>");
            return;
        }

        // now initialize the VertexBuffer
        gEngine.VertexBuffer.initialize();
    };
```

Apart from the syntax involved with the JavaScript Module pattern, the previous function should appear very similar to the `initializeGL()` function from the previous project and actually accomplishes the same task as it. A difference to note is the call to `gEngine.VertexBuffer.initialize()`. This function call sets up the WebGL vertex buffer and will be described in detail.

5. Add an additional function to clear the canvas to the desired color.

```
// Clears the draw area and draws one square
var clearCanvas = function(color) {
   mGL.clearColor(color[0], color[1], color[2], color[3]);  // set the color to
   be cleared
   mGL.clear(mGL.COLOR_BUFFER_BIT);      // clear to the color previously set
};
```

6. Finally, export the public functions through the returned `mPublic` object.

```
var mPublic = {
    getGL: getGL,
    initializeWebGL: initializeWebGL,
    clearCanvas: clearCanvas
};
```

The Shared Vertex Buffer

In the game engine that you are building, all graphical objects will be drawn based on the unit square. For this reason, the geometry subsystem is rather simplistic. The `gEngine.VertexBuffer` object implements the geometry subsystem.

1. Create a new source file in the `src/Engine` folder and name the file `Engine_VertexBuffer.js`. Remember to load this new source file in your `index.html` file.
2. Follow the JavaScript Module pattern and define `VertexBuffer` as a property of gEngine, as follows:

```
"use strict";  // Operate in Strict mode

var gEngine = gEngine || { };

// The VertexBuffer object
gEngine.VertexBuffer = (function() {
        // First: define the vertices for a square
        var verticesOfSquare = [
            0.5, 0.5, 0.0,
            -0.5, 0.5, 0.0,
            0.5, -0.5, 0.0,
            -0.5, -0.5, 0.0
        ];

        // reference to the vertex positions for the square in the gl context
        var mSquareVertexBuffer = null;

        var getGLVertexRef = function() { return mSquareVertexBuffer; };

        var initialize = function() {
            var gl = gEngine.Core.getGL();

            // Step A: Create a buffer on the gGL context for our vertex positions
            mSquareVertexBuffer = gl.createBuffer();

            // Step B: Activate vertexBuffer
            gl.bindBuffer(gl.ARRAY_BUFFER, mSquareVertexBuffer);

            // Step C: Loads verticesOfSquare into the vertexBuffer
            gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(verticesOfSquare),
                gl.STATIC_DRAW);
        };

        var mPublic = {
            initialize: initialize,
            getGLVertexRef: getGLVertexRef
        };
        return mPublic;
    }());
```

The `gEngine.VertexBuffer` object implements the same functionality as the source code in `VertexBuffer.js` from the previous project: it creates and loads the unit square geometry to the WebGL vertex buffer. Notice that instead of being global, `mSquareVertexBuffer`, the reference to the WebGL vertex buffer, is now a private variable with a proper accessor. The `initialize()` function contains the same three setup steps as the `initSquareBuffer()` function from the previous project.

Recall that the `gEngine.VertexBuffer.initialize()` function is called from the end of the `gEngine.Core.initializeWebGL()` function. In this way, after the `InitialzeWebGL()` function, the WebGL will be initialized and loaded with the vertex positions of the unit square, and the vertex buffer reference will be ready for drawing operations.

The Shader Object

Although the code in the `ShaderSupport.js` file from the previous project properly implements the required functionality, the global variables and functions do not lend themselves well to modification and code reuse. This section follows the object-oriented design principles and creates a `SimpleShader` object to abstract the behaviors and hide internal representations of shaders.

1. Create a new source file in the `src/Engine` folder and name the file `SimpleShader.js` to implement the `SimpleShader` object. Remember to load this new source file in `index.html`.

2. Define the constructor for `SimpleShader` to load, compile, and link the shaders into a program and to create a reference for loading from the WebGL vertex buffer for drawing.

```
function SimpleShader(vertexShaderID, fragmentShaderID) {
    // instance variables (Convention: all instance variables: mVariables)
    this.mCompiledShader = null;
        // reference to the compiled shader in webgl context
    this.mShaderVertexPositionAttribute = null;
        // reference to SquareVertexPosition in shader

    var gl = gEngine.Core.getGL();

    // start of constructor code
    //
    // Step A: load and compile vertex and fragment shaders
    var vertexShader = this._loadAndCompileShader(vertexShaderID, gl.VERTEX_SHADER);
    var fragmentShader = this._loadAndCompileShader(fragmentShaderID,
        gl.FRAGMENT_SHADER);

    // Step B: Create and link the shaders into a program.
    this.mCompiledShader = gl.createProgram();
    gl.attachShader(this.mCompiledShader, vertexShader);
    gl.attachShader(this.mCompiledShader, fragmentShader);
    gl.linkProgram(this.mCompiledShader);

    // Step C: check for error
    if (!gl.getProgramParameter(this.mCompiledShader, gl.LINK_STATUS)) {
        alert("Error linking shader");
        return null;
    }

    // Step D: Gets a reference to the aSquareVertexPosition attribute
    this.mShaderVertexPositionAttribute = gl.getAttribLocation(this.mCompiledShader,
                                  "aSquareVertexPosition");

    // Step E: Activates the vertex buffer loaded in Engine.Core_VertexBuffer
    gl.bindBuffer(gl.ARRAY_BUFFER, gEngine.VertexBuffer.getGLVertexRef());

    /// Step F: Describe the characteristic of the vertex position attribute
    gl.vertexAttribPointer(this.mShaderVertexPositionAttribute,
        3,              // each element is a 3-float (x,y.z)
        gl.FLOAT,       // data type is FLOAT
        false,          // if the content is normalized vectors
        0,              // number of bytes to skip in between elements
        0);             // offsets to the first element
}
```

Notice that this constructor is similar to the `initSimpleShader()` function from the previous project, with the following exceptions:

a. The global `gShaderVertexPositionAttribute` and `gSimpleShader` variables are now the private instance variables `mShaderVertexPositionAttribute` and `mCompiledShader`.

b. The reference to WebGL context, the `gl` variable, is now accessed through `gEngine.Core`.

3. Add a `_loadAndCompileShader()` method to the `SimpleShader` prototype to perform the actual loading and compiling functionality.

```
// Returns a complied shader from a shader in the dom.
// The id is the id of the script in the html tag.
SimpleShader.prototype._loadAndCompileShader = function(id, shaderType) {
    var shaderText, shaderSource, compiledShader;
    var gl = gEngine.Core.getGL();

    // Step A: Get the shader source from index.html
    shaderText = document.getElementById(id);
    shaderSource = shaderText.firstChild.textContent;

    // Step B: Create the shader based on the shader type: vertex or fragment
    compiledShader = gl.createShader(shaderType);

    // Step C: Compile the created shader
    gl.shaderSource(compiledShader, shaderSource);
    gl.compileShader(compiledShader);

    // Step D: check for errors and return results (null if error)
    // The log info is how shader compilation errors are typically displayed.
    // This is useful for debugging the shaders.
    if (!gl.getShaderParameter(compiledShader, gl.COMPILE_STATUS)) {
        alert("A shader compiling error occurred: " +
            gl.getShaderInfoLog(compiledShader));
    }

    return compiledShader;
};
```

Notice that this method is almost identical to the `LoadAndCompile()` function from the previous project. The only modification is that the WebGL context is now accessed through `gEngine.Core`.

4. Add a function to activate the shader for drawing.

```
SimpleShader.prototype.activateShader = function() {
    var gl = gEngine.Core.getGL();
    gl.useProgram(this.mCompiledShader);
    gl.enableVertexAttribArray(this.mShaderVertexPositionAttribute);
};
```

■ **Note** Functions that are meant to be private will have names that begin with an underscore (_), as in `_loadAndCompileShader()`. Though not enforced by JavaScript, you should never call a function with a name that begins with an underscore from outside the object. For example, the `_loadAndCompileShader()` function is not designed to be called from outside the `SimpleShader` object.

5. Finally, add an accessor for the actual WebGL shader program.

```
SimpleShader.prototype.getShader = function() { return this.mCompiledShader; };
```

The `src/Engine` folder contains the source code to the game engine. In this way, the game engine is simply a library that provides functionality for creating games. For now, your game engine consists of three objects that support the initialization of WebGL and the drawing of a unit square. This is the folder that you will continue to add source files and functionality to, which eventually will become a complete and sophisticated game engine.

The Client Source Code

The `src/MyGame` folder will contain the actual source code to a game. As mentioned, the code in this folder will be referred to as the *client* of the game engine. For now, the source code in the `MyGame` folder will focus on testing the functionality of the simple game engine.

1. Create a new source file in the `src/MyGame` folder, or the client folder, and name the file `MyGame.js`. Remember to load this new source file in `index.html`.

2. Create a constructor that receives an HTML canvas ID as a parameter and contains a shader as an instance variable.

```
function MyGame(htmlCanvasID) {
    // The shader for drawing
    this.mShader = null;
    ...
}
```

3. Within the constructor, initialize the WebGL context and the `VertexBuffer`.

```
gEngine.Core.initializeWebGL(htmlCanvasID);
```

4. Then, create, load, and compile the shaders by doing the following:

```
this.mShader = new SimpleShader("VertexShader", "FragmentShader");
```

5. Finally, clear the canvas, activate the shader, and draw.

```
// Step C1: Clear the canvas
gEngine.Core.clearCanvas([0, 0.8, 0, 1]);

// Step C2: Activate the proper shader
this.mShader.activateShader();
```

```
        // Step C3: Draw with the currently activated geometry and the activated shader
        var gl = gEngine.Core.getGL();
        gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
```

Remember to create the MyGame object from within the index.html body element.

```
<body onload="new MyGame('GLCanvas');">
```

Observations

Although you're accomplishing the same tasks as with the previous project, with this project you have created infrastructure that supports subsequent modifications and expansions of your game engine. You have organized your source code into separate and logical folders, created Singleton-like objects to implement core functionality of the engine, and gained experience with abstracting the SimpleShader object that will support future design and code reuse. With the engine now comprised of well-defined objects with clean interface methods, you can learn new concepts and build new abstractions, which you can continually add to your engine.

Separating GLSL from HTML

Recall that in your projects thus far the GLSL shader code is embedded in the HTML source code of index.html. This organization means that new shaders must be added through the editing of the index.html file. Logically, GLSL shaders should be organized separately from HTML source files; logistically, continuously adding to index.html will result in an unmanageable, cluttered file that would become difficult to work with. For these reasons, the GLSL shaders should be stored in separate source files.

The Shader Source Files Project

This project demonstrates how to separate the GLSL shaders into separate files. As illustrated in Figure 2-9, when running this project, a white rectangle is displayed on a greenish canvas, identical to the previous projects. The source code to this project is defined in the Chapter2/2.5.ShaderSourceFiles folder.

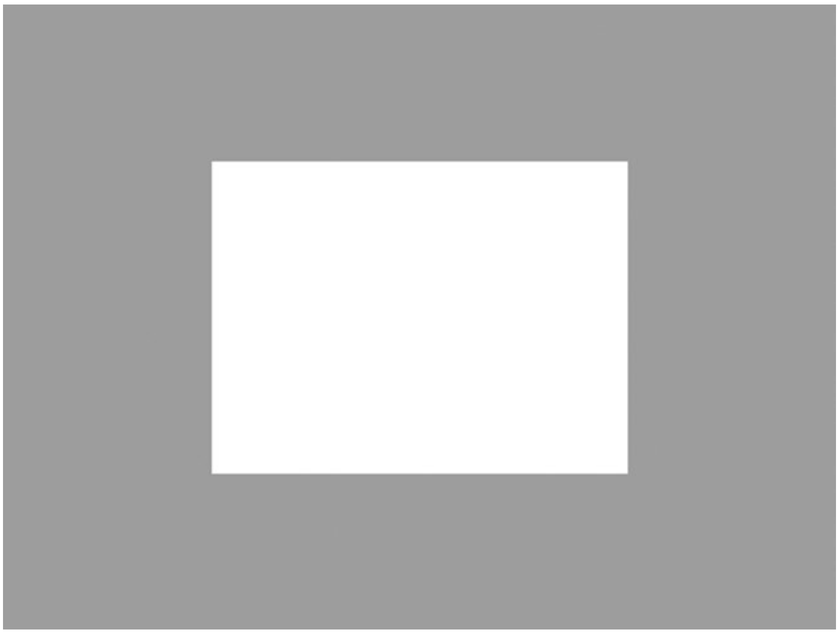


Figure 2-9. *Running the Shader Source Files project*

The goals of the project are as follows:

- To separate the GLSL shaders from the HTML source code
- To demonstrate how to load the shader source files during runtime

Loading Shaders in SimpleShader

Instead of loading the GLSL shaders as part of the HTML document, the `_loadAndCompileShader()` in `SimpleShader` can be modified to load the GLSL shaders as separate files.

1. Continue from previous project and open the `SimpleShader.js` file to edit the `_loadAndCompileShader()` function to receive a file path instead of an HTML ID:

   ```
   SimpleShader.prototype._loadAndCompileShader = function(filePath, shaderType)
   ```

2. Replace the HTML element retrieval code with the following `XMLHttpRequest` to load a file:

   ```
   xmlReq = new XMLHttpRequest();
   xmlReq.open('GET', filePath, false);
   try {
       xmlReq.send();
   } catch (error) {
       alert("Failed to load shader: " + filePath);
       return null;
   }
   shaderSource = xmlReq.responseText;
   ```

```
        if (shaderSource === null) {
            alert("WARNING: Loading of:" + filePath + " Failed!");
            return null;
        }
```

Notice that the file loading will occur synchronously where the web page will actually stop and wait for the completion of the `xmlReq.open()` function to return with the opened file content. If the file should be missing, the opening operation will fail, and the response text will be null.

The synchronized "stop and wait" for the completion of `xmlReq.open()` function is inefficient and may result in slow loading of the web page. This shortcoming will be addressed in Chapter 4 when you learn about the asynchronous loading of game resources.

■ **Note** The `XMLHttpRequest()` object requires a running web server to fulfill the HTTP get request. This means you will be able to test this project from within the NetBeans IDE. However, unless there is a web server running on your machine, you will not be able to run this project by double-clicking the `index.html` file directly. This is because there is no server to fulfill the HTTP get requests and the GLSL shader loading will fail.

With this modification, the `SimpleShader` constructor can now be modified to receive and forward file paths to the `_loadAndCompileShader()` function instead of the HTML element IDs.

Extracting Shaders into Their Own Files

The following steps retrieve the source code to the vertex and fragment shaders from the `index.html` file and create separate files for storing them.

1. Create a new folder in the `src` folder and name it `GLSLShaders`, as illustrated in Figure 2-10.

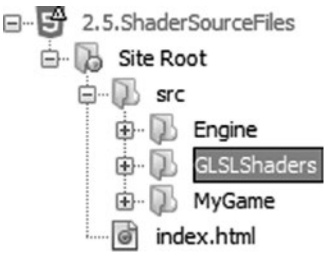


Figure 2-10. *Creating the GLSLShaders folder*

This will be the folder that contains all of the GLSL shader source code files.

2. Create two new text files within the `GLSLShaders` folder and name them `SimpleVS.glsl` and `WhiteFS.glsl`.

■ **Note** All GLSL shader source code files will end with the `.glsl` extension. The `VS` in the shader file names signifies that the file contains a vertex shader, while `FS` signifies a fragment shader.

3. To create the GLSL vertex shader, edit `SimpleVS.glsl` to add the existing vertex shader code in the `index.html` file from the previous project.

```
attribute vec3 aSquareVertexPosition;  // Expects one vertex position
void main(void) {
    gl_Position = vec4(aSquareVertexPosition, 1.0);
}
```

4. To create the GLSL fragment shader, edit `WhiteFS.glsl` to add the fragment shader code in the `index.html` file from the previous project.

```
void main(void) {
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

Cleaning Up HTML Code

With vertex and fragment shaders being stored in separate files, it is now possible to clean up the `index.html` file such that it contains only HTML code.

1. Remove all the GLSL shader code from `index.html`, such that this file becomes as follows:

```
<!DOCTYPE html>
<head>
    <title>2.5: The Shader Source Files Project</title>
</head>
<body onload="new MyGame('GLCanvas');">
    <!-- Engine code -->
        <script type="text/javascript" src="src/Engine/Engine_Core.js"></script>
    <script type="text/javascript" src="src/Engine/Engine_VertexBuffer.js">
    </script>
        <script type="text/javascript" src="src/Engine/SimpleShader.js"></script>
    <!-- Client game code -->
        <script type="text/javascript" src="src/MyGame/MyGame.js"></script>
    ...
</body>
```

Notice that `index.html` no longer contains any GLSL shader and only a single line of JavaScript code (to create the `MyGame` object). With this organization, the `index.html` file can properly be considered as representing the web page where you will not need to edit it to modify a shader.

2. Modify the game to load the shader files instead of HTML element IDs. Edit `MyGame.js` to modify step B to refer to the shader files created.

```
this.mShader = new SimpleShader(
    "src/GLSLShaders/SimpleVS.glsl",    // Path to the VertexShader
    "src/GLSLShaders/WhiteFS.glsl");    // Path to the FragmentShader
```

Source Code Organization

The separation of logical components in the engine source code has progressed to the following state:

- `index.html`: This is the file that contains the HTML code that defines the canvas on the web page for the game and loads all of the source code for your game.
- `src/GLSLShaders`: This is the folder that contains all the GLSL source code files that shade the elements of your game.
- `src/Engine`: This is the folder that contains all the source code for your game engine.
- `src/MyGame`: This is the client folder that contains the source code for the actual game.

Changing the Shader and Controlling the Color

With GLSL shaders being stored in separate source code files, it is now possible to edit or replace the shaders with relatively minor changes to the rest of the source code. The next project demonstrates this convenience by replacing the restrictive constant white color fragment shader, `WhiteFS.glsl`, with a shader that can be parameterized to draw with any color.

The Parameterized Fragment Shader Project

This project replaces `WhiteFS.glsl` with a `SimpleFS.glsl` that supports the drawing with any color. Figure 2-11 shows the output of running the Parameterized Fragment Shader project; notice that a blue square replaces the white square from previous projects. The source code for this project is defined in the `Chapter2/2.6.ParameterizedFragmentShader` folder.

Figure 2-11. *Running the Parameterized Fragment Shader project*

The goals of the project are as follows:

- To gain experience with creating a GLSL shader in the source code structure
- To learn about the `uniform` variable and define a fragment shader with the color parameter

Defining the SimpleFS.glsl Fragment Shader

A new fragment shader needs to be created to support changing the pixel color for each draw operation. This can be accomplished by creating a new GLSL fragment shader in the `src/GLSLShaders` folder and name it `SimpleFS.glsl`. Edit this file to add the following:

```
precision mediump float;   // sets the precision for floating point computation
uniform vec4 uPixelColor;  // to transform the vertex position
void main(void) {
    gl_FragColor = uPixelColor;
}
```

Recall that the GLSL `attribute` keyword identifies data that changes for every vertex position. In this case, the `uniform` keyword denotes that a variable is constant for all the vertices. The `uPixelColor` variable can be set from JavaScript to control the eventual pixel color. The `precision mediump` keywords define the floating precisions for computations.

■ **Note** Floating-point precision trades the accuracy of computation for performance. Please follow the references in Chapter 1 for more information on WebGL.

Modify the SimpleShader to Support the Color Parameter

The `SimpleShader` can now be modified to gain access to the new `uPixelColor` variable.

1. Edit `SimpleShader.js` and add a new instance variable for referencing the `uPixelColor`.

```
this.mPixelColor = null;
// reference to the pixelColor uniform in the fragment shader
```

2. Add code to the end of the constructor to create the reference.

```
// Step G: Gets a reference to the uniform variable uPixelColor in the
//     fragment shader
this.mPixelColor = gl.getUniformLocation(this.mCompiledShader, "uPixelColor");
```

3. Modify the shader activation to allow the setting of the pixel color.

```
SimpleShader.prototype.activateShader = function (pixelColor) {
    var gl = gEngine.Core.getGL();
    gl.useProgram(this.mCompiledShader);
    gl.enableVertexAttribArray(this.mShaderVertexPositionAttribute);
    gl.uniform4fv(this.mPixelColor, pixelColor);
};
```

The `gl.uniform4fv()` function copies four floating-point values from `pixelColor` to the `mPixelColor`, or the `uPixelColor`, in the `SimpleFS.glsl` fragment shader.

Drawing with the New Shader

To test `SimpleFV.glsl`, modify the `MyGame` constructor to create and draw with the new shader.

```
function MyGame(htmlCanvasID) {
    // Step A: Initialize the webGL Context and the VertexBuffer
    gEngine.Core.initializeWebGL(htmlCanvasID);

    // Step B: Create, load and compile the shaders
    this.mShader = new SimpleShader(
            "src/GLSLShaders/SimpleVS.glsl",      // Path to the VertexShader
            "src/GLSLShaders/SimpleFS.glsl");     // Path to the FragmentShader

    // Step C: Draw!
    // Step C1: Clear the canvas
    gEngine.Core.clearCanvas([0, 0.8, 0, 1]);

    // Step C2: Activate the proper shader
    this.mShader.activateShader([0, 0, 1, 1]);

    // Step C3: Draw with the currently activated geometry and the activated shader
    var gl = gEngine.Core.getGL();
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
}
```

Notice that the `mConstColorShader` is created with the new `SimpleFS.glsl` (instead of `WhiteFS`) and that it is now important to set the drawing color when activating the shader. With the new `SimpleFS`, you can now experiment with drawing the squares with any desired color.

As you have experienced in this project, the source code structure supports simple and localized changes when the game engine is expanded, in this case only changes to `SimpleShader.js` file. This demonstrates the benefit of proper encapsulation and source code organization.

Summary

By this point the game engine is simple and supports only the initialization of WebGL and the drawing of one colored square. However, through the projects in this chapter, you have gained experience with the techniques needed in order to build an excellent foundation for the game engine. You have also structured the source code in a way that allows you to support further complexity with limited modification to the existing code base, and you are now ready to further encapsulate the functionality of the game engine to facilitate additional features. The next chapter will focus on building a proper framework in the game engine to support more flexible and configurable drawings.