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 JavaScript modules and instantiate JavaScript objects to abstract and 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 serves 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.

**Note** The game engine we will develop in this book is based on the latest version of WebGL specification: version 2.0. For brevity, the term WebGL will be used to refer to this API.

# 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.html5\_canvas folder.

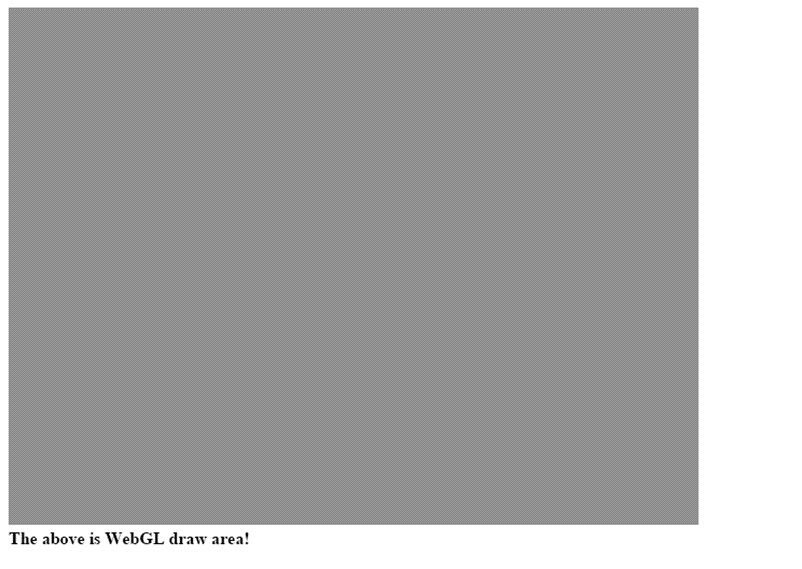


Figure 2-1. Running the HTML5 Canvas project

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 project by creating a new folder named html5\_canvas in your chosen directory and copying and pasting the index.html file you created in the previous project in chapter 1 into it.

**Note** Henceforth, throughout this book, when asked to create a new project you should follow the process describe previously. That is, create a new folder with the projects name and copy the previous projects files into it. In this way your new projects can expand upon your old ones while retaining their intended functionality within the original project during your development.

1. Open the index.html file in the editor by opening the html5\_canvas folder, expanding it if needed and double-clicking the index.html file, as illustrated in Figure 2-2.

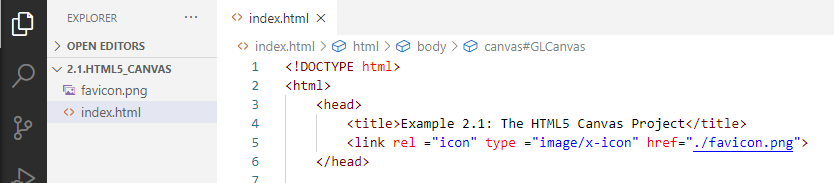


Figure 2-2. Editing the index.html file in your project

1. Create the HTML canvas for drawing by adding the following lines 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 WebGL.

**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.

1. 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 project. You will now write JavaScript for the remainder of the example.

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

"use strict";

let canvas = document.getElementById("GLCanvas");

**Note** The “let” JavaScript keyword defines variables.

The first line, "use strict", is a JavaScript directive indicating that the code should be executed in "strict mode", where the use of undeclared variables is a runtime error. The second line creates a new variable named canvas and references the variable to the GLCanvas drawing area.

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

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

let gl = canvas.getContext("webgl2") || canvas.getContext("experimental-webgl2");

As the code indicates, the retrieved reference to the WebGL version 2 context is stored in the local variable named gl. From this variable, you have access to all the functionality of WebGL 2.0. Once again, in the rest of this book, the term WebGL will be used to refer to the WebGL version 2.0 API.

1. 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);

gl.clear(gl.COLOR\_BUFFER\_BIT);

}

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 later chapters. For now, always assign 1.0 to the alpha channel. The specified color, (0.0, 0.8, 0.0, 1.0), has zero values for the red and blue channels and a 0.8, or 80%, intensity on the green channel. For this reason, the canvas area is cleared to a light green color.

1. Add a simple write command to the document to identify the canvas by inserting the following line:

document.write("<br><b>The above is WebGL draw area!</b>");

You can refer to the final source code in the index.html file in the chapter2/2.1.html5\_canvas 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 changing the 640×480 values 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 classes in 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. You can accomplish this by creating a separate JavaScript source code file named core.js which implements 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\_source\_file folder.



Figure 2-3. Running the JavaScript Source File project

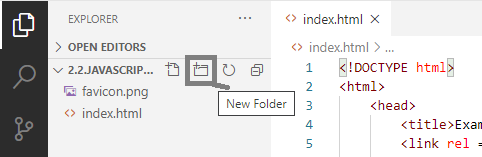
The goals of the project are as follows:

* 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 javascript\_source\_file. Recall that a new project is created by creating a folder with the appropriate name, copying files from the previous project, and editing the <Title> element of the index.html to reflect the new project.
2. Create a new folder named src inside the project folder by clicking the new folder icon while hovering over the project folder, as illustrated in Figure 2-4. This folder will contain all of your source code.



1. Figure 2-4. Creating a new source code folderCreate a new source code file within the src folder by right-clicking the src folder, as illustrated in Figure 2-5. Name the new source file core.js.

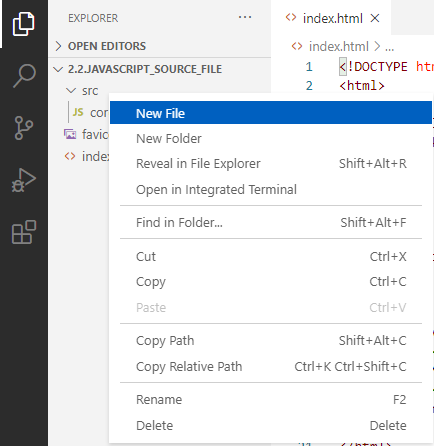


Figure 2-5. Adding a new JavaScript source code file**Note** In Visual Studio Code you can create/copy/rename folders and files by using the right-click menus in the Explorer window.

1. Open the new core.js source file for editing.
2. Define a variable for referencing the WebGL context and add a function which allows you to access the variable.

"use strict";

let mGL = null;

function getGL() { return mGL; }

**Note** Variables that are accessible throughout a file, or a module, have names that begin with lowercase “m”, as in mGL.

1. Define the initWebGL() function to retrieve GLCanvas by passing in the proper canvas id as a parameter, bind the drawing area to the WebGL context, store the results in the defined mGL variable, and clear the drawing area.

function initWebGL(htmlCanvasID) {

let canvas = document.getElementById(htmlCanvasID);

mGL = canvas.getContext("webgl2") ||

canvas.getContext("experimental-webgl2");

if (mGL === null) {

document.write("<br><b>WebGL 2 is not supported!</b>");

return;

}

mGL.clearColor(0.9, 0.9, 0.9, 1.0);

}

Notice that this function is similar to the JavaScript source code you typed in the previous project. This is because all you are doing differently in this case is separating JavaScript source code from HTML code.

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

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

function clearCanvas() {

mGL.clear(mGL.COLOR\_BUFFER\_BIT);

}

1. Define a function to carry out the initialization and clearing of the canvas area after the web browser has completed the loading of the index.html file.

window.onload = function() {

initWebGL("GLCanvas");

clearCanvas();

}

### Load and Run JavaScript Source Code from index.html

With all the JavaScript functionality defined in the core.js file, you now need to load this file to operate on your web page through 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 core.js source code by including the following code within the head element:

<script type="module" src="./src/core.js"></script>

With this code, the core.js file will be loaded as part of the index.html defined web page. Recall that you have defined a function for window.onload and that function will be invoked when the loading of index.html is completed.

You can refer to the final source code in the core.js and index.html files in the chapter2/2.2.javascript\_source\_file project folder. 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 extracted to core.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.

# 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 the CPU 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.draw\_one\_square folder.

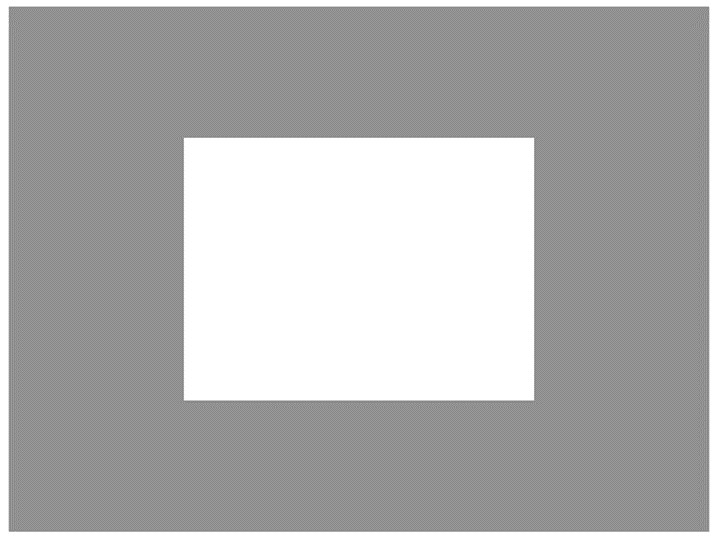


Figure 2-6. Running the Draw One Square project

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
* ms based on simple source code files

### 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 variable. Learning from the previous project, the corresponding JavaScript code will be stored in a new source code file, vertex\_buffer.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 vertex\_buffer.js.
2. Import all the exported functionality from the core.js file as core module with the Javascript import statement.

"use strict";

import \* as core from "./core.js";

**Note** With the JavaScript import and, soon to be encountered, export statements, features and functionalities defined in a file can be conveniently encapsulated and accessed. In this case, the functionality exported from core.js is imported in vertex\_buffer.js and accessible via the module identifier, core. For example, as you will see, in this example core.js defines and exports a getGL() function. With the given import statement, this function can be accessed as core.getGL() in the vertex\_buffer.js file.

1. Declare the variable mGLVertexBuffer to store the reference to the WebGL buffer location. Remember to define a function for accessing this variable.

let mGLVertexBuffer = null;

function get() { return mGLVertexBuffer; }

1. Define the variable mVerticesOfSquare and initialize it with vertices of a unit square.

let mVerticesOfSquare = [

0.5, 0.5, 0.0,

-0.5, 0.5, 0.0,

0.5, -0.5, 0.0,

-0.5, -0.5, 0.0

];

In the code shown, each row of three numbers are the x, y, and z coordinate position of a vertex. Notice that the z-dimension is set to 0.0 because you are building a 2D game engine. Also notice that 0.5 is being used so that we define a square in 2D space which has sides equal to 1 and centered at the origin, or a unit square.

1. Define the init() function to define a buffer on and load the vertices to the GPU.

function init() {

let gl = core.getGL();

// Step A: Create a buffer on the gl context for our vertex positions

mGLVertexBuffer = gl.createBuffer();

// Step B: Activate vertexBuffer

gl.bindBuffer(gl.ARRAY\_BUFFER, mGLVertexBuffer);

// Step C: Loads mVerticesOfSquare into the vertexBuffer

gl.bufferData(gl.ARRAY\_BUFFER,

new Float32Array(mVerticesOfSquare), gl.STATIC\_DRAW);

}

This code first gets access to the WebGL drawing context through the core.getGL() function. Afterwhich, 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 variable mGLVertexBuffer. Step B activates the newly created buffer, and step C loads the vertex position of the square into the activated buffer on the GPU. The keyword STATIC\_DRAW informs the drawing hardware that the content of this buffer will not be changed.

**Tip** Remember that the mGL variable accessed through the getGL() function is defined in the core.js file and initialized by the initWebGL() function. You will define an export statement in the core.js file to provide access to this function in the coming steps.

1. Provide access to the init() and get() functions to the rest of your engine by exporting them with the following code.

export {init, get}

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 CPU 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 shader\_support.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 head element, add the following code:

<script type="x-shader/x-vertex" id="VertexShader">

// this is the vertex shader

attribute vec3 aVertexPosition; // Vertex shader expects one vertex position

// naming convention, attributes always begin with "a"

void main(void) {

// Convert the vec3 into vec4 for scan conversion and

// assign to gl\_Position to pass the vertex to the fragment shader

gl\_Position = vec4(aVertexPosition, 1.0);

}

// End of vertex shader

</script>

**Note** Shader attribute variables have names that begin with a lowercase a, as in aVertexPosition.

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 aVertexPosition attribute is of data type vec3 or an array of three floating-point numbers. As you will see in later steps, aVertexPosition will be set to reference to the vertex positions for the unit 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 aVertexPosition into a vec4 and passing the information to WebGL.

1. Define the fragment shader in index.html by adding the following code within the head element:

<script type="x-shader/x-fragment" id="FragmentShader">

// this is the fragment (or pixel) shader

void main(void) {

// for every pixel called (within the square) sets

// constant color white with alpha-channel value of 1.0

gl\_FragColor = vec4(1.0, 1.0, 1.0, 1.0);

}

// End of fragment/pixel shader

</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, shader\_support.js.

1. Create a new JavaScript file, shader\_support.js.
2. Import functionality from the core.js and vertex\_buffer.js files.

"use strict"; // Operate in Strict mode such that variables must be declared before used!

import \* as core from "./core.js"; // access as core module

import \* as vertexBuffer from "./vertex\_buffer.js"; // access as vertexBuffer module

1. Define two variables, mCompiledShader and mVertexPositionRef, for referencing to the shader program and the vertex position attribute in the GPU.

let mCompiledShader = null;

let mVertexPositionRef = null;

1. Create a function to load and compile the shader you defined in the index.html.

function loadAndCompileShader(id, shaderType) {

let shaderSource = null, compiledShader = null;

// Step A: Get the shader source from index.html

let shaderText = document.getElementById(id);

shaderSource = shaderText.firstChild.textContent;

let gl = core.getGL();

// 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)) {

throw new Error("A shader compiling error occurred: " +

gl.getShaderInfoLog(compiledShader));

}

return compiledShader;

}

Step A of the code finds shader source code in 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 source code and compiles the shader. Finally, step D checks and returns the reference to the compiled shader while throwing an error if the shader compilation is unsuccessful.

1. You are now ready to create and compile a shader program by defining the init() function.

function init(vertexShaderID, fragmentShaderID) {

let gl = core.getGL();

// Step A: load and compile vertex and fragment shaders

let vertexShader = loadAndCompileShader(vertexShaderID, gl.VERTEX\_SHADER);

let fragmentShader = loadAndCompileShader(fragmentShaderID, gl.FRAGMENT\_SHADER);

// Step B: Create and link the shaders into a program.

mCompiledShader = gl.createProgram();

gl.attachShader(mCompiledShader, vertexShader);

gl.attachShader(mCompiledShader, fragmentShader);

gl.linkProgram(mCompiledShader);

// Step C: check for error

if (!gl.getProgramParameter(mCompiledShader, gl.LINK\_STATUS)) {

throw new Error("Error linking shader");

return null;

}

// Step D: Gets a reference to the aVertexPosition attribute within the shaders.

mVertexPositionRef = gl.getAttribLocation(mCompiledShader, "aVertexPosition");

}

Step A of the code loads and 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 variable mCompiledShader. After error checking in step C, step D locates and stores the reference to the aVertexPosition attribute defined in your vertex shader.

1. Define a function to allow the activation of the shader so that it can be used for drawing the square.

function activate() {

// Step A: access to the webgl context

let gl = core.getGL();

// Step B: identify the compiled shader to use

gl.useProgram(mCompiledShader);

// Step C: bind the vertex buffer to the attribute defined in the vertex shader

gl.bindBuffer(gl.ARRAY\_BUFFER, vertexBuffer.get());

gl.vertexAttribPointer(this.mVertexPositionRef,

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

gl.enableVertexAttribArray(this.mVertexPosition);

}

In the code shown, step A sets the gl variable as access to the WebGL context. Step B enables the compiled shader program, while step C binds the vertex buffer created in vertex\_buffer.js to the aVertexPosition attribute defined in the vertex shader. The gl.vertexAttribPoint() function captures the fact that the vertex buffer was loaded with vertices of a unit square consisting of three floating point values for each vertex position.

1. Lastly, provide access to the init() and activate() functions to the rest of the game engine by exporting them with the export statement.

export { init, activate }

**Note** Notice that the loadAndCompileShader() function is excluded from the export statement. This function is not needed elsewhere and thusly should remain private to this file, following good development practice of hiding local implementation details.

The shader loading and compiling functionality is now defined. You can now utilize and 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 defined in the core.js file. Now open this file for editing.

1. Import the defined functionality from vertex\_buffer.js and shader\_support.js files.

import \* as vertexBuffer from "./vertex\_buffer.js";

import \* as simpleShader from "./shader\_support.js";

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

function initWebGL(htmlCanvasID) {

let 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("webgl2") || canvas.getContext("experimental-webgl2");

if (mGL === null) {

document.write("<br><b>WebGL 2 is not supported!</b>");

return;

}

mGL.clearColor(0.0, 0.8, 0.0, 1.0); // set the color to be cleared

// 1. initialize the buffer with the vertex positions for the unit square

vertexBuffer.init(); // This function is defined in the vertex\_buffer.js file

// 2. now load and compile the vertex and fragment shaders

simpleShader.init("VertexShader", "FragmentShader");

// the two shaders are defined in the index.html file

// init() function is defined in shader\_support.js file

}

As shown in the code, after successfully obtaining the reference to the WebGL context and setting the clear color, you should first call the init() function defined in vertex\_buffer.js to initialize the GPU vertex buffer with the unit square vertices and secondly call the init() function defined in shader\_support.js to load and compile the vertex and fragment shaders.

1. Add a drawSquare() function for drawing the defined square.

function drawSquare() {

// Step A: Activate the shader

simpleShader.activate();

// Step B. draw with the above settings

mGL.drawArrays(mGL.TRIANGLE\_STRIP, 0, 4);

}

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

1. Now you just need to modify the window.onload function to call the newly defined drawSquare() function.

window.onload = function() {

initWebGL("GLCanvas"); // Binds mGL context to WebGL functionality

clearCanvas(); // Clears the GL area

drawSquare(); // Draws one square

}

1. Finally, provide access to the WebGL context to the rest of the engine by exporting the getGL() function. Remember that this function is imported and has been called to access the WebGL context in both vertex\_buffer.js and simple\_shader.js.

export {getGL}

Recall that the function that is bounded to window.onload will be invoked after indexl.html has been loaded by the web browser. For this reason, WebGL will be initialized, the canvas cleared to light green, and a white square will be drawn. You can refer to the source code in the chapter2/2.3.draw\_one\_square project for the entire system described.

## 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 the next chapter.

You can try editing the fragment shader in index.html by changing the color set in the gl\_FragColor function to alter 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 three separate files and hides information with the import/export statements of JavaScript. The functionality defined in these files with the corresponding import and export statements are referred to as JavaScript modules. A module can be considered as a global singleton object and, as you have observed in the modules, is excellent for hiding detailed implementations. However, modules are not-suited for supporting abstraction and specialization. In the next sections, you will begin to work with JavaScript classes to further encapsulate 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 function. In software engineering, this solution process is referred to as functional decomposition, and the implementation is referred to as procedural programming. Procedural programming results in solutions that are well-structured, easy to understand, and often fast to create. This is why functional decomposition and procedural programming are often used to prototype concepts 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, behavior specialization, and code reuse through inheritance.

## The JavaScript Objects Project

This project demonstrates how to abstract the global functions from the Draw One Square project into JavaScript classes and 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 can be found in the chapter2/2.4.javaScript\_objects folder.



Figure 2-7. Running the JavaScript Objects project

The goals of the project are as follows:

* To separate the code for the game engine from that for the game logic
* To understand how to build abstractions with JavaScript classes and objects

The steps for creating this 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 a JavaScript class to abstract the simple\_shader and work with an instance of this class.
3. Define a JavaScript class 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 Visual Studio Code by creating a new folder and adding a sub source code folder named src. Within src, create engine and my\_game as subfolders, as illustrated in Figure 2-8.

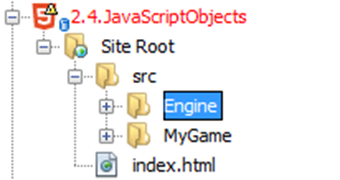


Figure 2-8. Creating engine under the src folder

The src/engine folder will contain all the source code to the game engine, and the src/my\_game 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 my\_game 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 my\_game 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 functionality based on the JavaScript module you have worked with in the previous project.

**Note** All module and 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 a module or instance variable from outside the module/class. For example, you should never access core.mGL directly; instead, call the core.getGL() function to access the variable.

#### The Shader Class

Although the code in the shader\_support.js file from the previous project properly implements the required functionality, the variables and functions do not lend themselves well to behavior specialization and code reuse. For example, in the cases when different types of shaders are required, it can be challenging to modify the implementation while achieving behavior and code reuse. This section follows the object-oriented design principles and defines a SimpleShader class to abstract the behaviors and hide internal representations of shaders. Besides the ability to create multiple instances of the SimpleShader object the basic functionality remains largely unchanged.

1. Create a new source file in the src/engine folder and name the file simple\_shader.js to implement the SimpleShader object.
2. Import both the core and vertex\_buffer modules.

import \* as core from "./core.js";

import \* as vertexBuffer from "./vertex\_buffer.js";

1. Declare the SimpleShader as a JavaScript class.

class SimpleShader { … }

**Note** The “…” represents details of the implementation to be discussed subsequently.

1. Define the constructor within the SimpleShader class to load, compile, and link the vertex and fragment shaders into a program and to create a reference to the aVertexPosition attribute in the vertex shader for loading the square vertex positions from the WebGL vertex buffer for drawing.

constructor(vertexShaderID, fragmentShaderID) {

// instance variables

// Convention: all instance variables: mVariables

this.mCompiledShader = null; // reference to the compiled shader in webgl context

this.mVertexPositionRef = null; // reference to VertexPosition within the shader

let gl = core.getGL();

// Step A: load and compile vertex and fragment shaders

this.mVertexShader = loadAndCompileShader(vertexShaderID,

gl.VERTEX\_SHADER);

this.mFragmentShader = loadAndCompileShader(fragmentShaderID,

gl.FRAGMENT\_SHADER);

// Step B: Create and link the shaders into a program.

this.mCompiledShader = gl.createProgram();

gl.attachShader(this.mCompiledShader, this.mVertexShader);

gl.attachShader(this.mCompiledShader, this.mFragmentShader);

gl.linkProgram(this.mCompiledShader);

// Step C: check for error

if (!gl.getProgramParameter(this.mCompiledShader, gl.LINK\_STATUS)) {

throw new Error("Error linking shader");

return null;

}

// Step D: Gets a reference to the aVertexPosition attribute within the shaders.

this.mVertexPositionRef = gl.getAttribLocation(this.mCompiledShader, "aVertexPosition");

}

Notice that this constructor is essentually the same as the init() function in the shader\_support.js module from the previous project.

1. Add a method to the SimpleShader class to activate the shader for drawing. Once again, similar to your activate() function in shader\_support.js from previous project.

activate() {

let gl = core.getGL();

gl.useProgram(this.mCompiledShader);

// bind vertex buffer

gl.bindBuffer(gl.ARRAY\_BUFFER, vertexBuffer.get());

gl.vertexAttribPointer(this.mVertexPositionRef,

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

gl.enableVertexAttribArray(this.mVertexPositionRef);

}

1. Add a private method, which cannot be access from outside the simple\_shader.js file, by creating a function outside the SimpleShader class to perform the actual loading and compiling functionality.

function loadAndCompileShader(id, shaderType) {

let shaderSource = null, compiledShader = null;

let gl = core.getGL();

// Step A: Get the shader source from index.html

let 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)) {

throw new Error("A shader compiling error occurred: " +

gl.getShaderInfoLog(compiledShader));

}

return compiledShader;

}

Notice that this function is identical to the one you created in shader\_support.js.

1. Finally, add an export for the SimpleShader class such that it can be accessed and instantiated outside of this file.

export default SimpleShader;

**Note** The default keyword signifies that the name SimpleShader cannot changed by import statements.

#### The Core of the Game Engine: core.js

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 copy of your core.js under the new folder src/engine.
2. Add a function to create a new instance of the SimpleShader object.

// The shader

let mShader = null;

function createShader() {

mShader = new SimpleShader(

"VertexShader", // IDs of the script tag in the index.html

"FragmentShader"); //

}

1. Modify the initWebGL() function to focus on only initializing the WebGL as follows.

// initialize the WebGL

function initWebGL(htmlCanvasID) {

let 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("webgl2") || canvas.getContext("experimental-webgl2");

if (mGL === null) {

document.write("<br><b>WebGL 2 is not supported!</b>");

return;

}

}

1. Create an init() function to perform engine-wide system initialization, which includes initializing WebGL and the vertex buffer, and creating an instance of the simple shader.

function init(htmlCanvasID) {

initWebGL(htmlCanvasID); // setup mGL

vertexBuffer.init(); // setup mGLVertexBuffer

createShader(); // create the shader

}

1. Modify the clear canvas function to parameterize the color to be cleared to.

function clearCanvas(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 set

}

1. Export the relevant functions for access by the rest of the game engine.
2. Finally, remove the window.onload function as the behavior the actual game should be defined by the client of the game engine, or, the MyGame class in this case.

The src/engine folder now contains the basic source code for the entire game engine. Due to these structural changes to your source code, the game engine can now function as a simple library that provides functionality for creating games, or a simple Application Programming Interface (API). For now, your game engine consists of three files that support the initialization of WebGL and the drawing of a unit square, the core module, the vertex\_buffer module and the SimpleShader class. New source files and functionality will continue to be added to this folder throughout the remaining projects. Eventually this folder will contain a complete and sophisticated game engine API. However, the core library-like framework defined here will persist.

### The Client Source Code

The src/my\_game folder will contain the actual source code for the 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 my\_game folder will focus on drawing a simple square by utilizing the functionality of the simple game engine you defined.

1. Create a new source file in the src/my\_game folder, or the client folder, and name the file my\_game.js.
2. Import the core module as follows:

import \* as engine from "../engine/core.js";

1. Define MyGame as a JavaScript class and add a constructor to initialize the game engine, clears the canvas, and draws a square.

class MyGame {

constructor(htmlCanvasID) {

// Step A: Initialize the game engine

engine.init(htmlCanvasID);

// Step B: Clear the canvas

engine.clearCanvas([0, 0.8, 0, 1]);

// Step C: Draw the square

engine.drawSquare();

}

}

1. Bind the creation of a new instance of the MyGame object to the window.onload function.

window.onload = function() {

new MyGame('GLCanvas');

}

1. Finally, modify the index.html to load the game client rather than the engine core.js within the head element.

<script type="module" src="./src/my\_game/my\_game.js"></script>

## Observations

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

# Separating GLSL from HTML

Thus far in your projects 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 a cluttered and unmanageable 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.shader\_source\_files 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 the previous project, open the SimpleShader.js file, edit the loadAndCompileShader() function, to receive a file path instead of an HTML ID.

function loadAndCompileShader(filePath, shaderType)

1. Within the loadAndCompileShader() function replace the HTML element retrieval code in step A with the following XMLHttpRequest to load a file.

let **xmlReq**, shaderSource = null, compiledShader = null;

let gl = core.getGL();

// Step A: Request the text from the given file location.

xmlReq = new XMLHttpRequest();

xmlReq.open('GET', filePath, false);

try {

xmlReq.send();

} catch (error) {

throw new Error("Failed to load shader: "

+ filePath

+ " [Hint: you cannot double click index.html to run this project. "

+ "The index.html file must be loaded by a web-server.]");

return null;

}

shaderSource = xmlReq.responseText;

if (shaderSource === null) {

throw new Error("WARNING: Loading of:" + filePath + " Failed!");

return null;

}

Notice that the file loading will occur synchronously where the web browser will actually stop and wait for the completion of the xmlReq.open() function to return with the content of the opened file. 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 Visual Studio Code with the installed Go Live extension. 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 of the vertex and fragment shaders from the index.html file and create separate files for storing them.

1. Create a new folder that will contain all of the GLSL shader source code files in the src folder and name it glsl\_shaders, as illustrated in Figure 2-10.

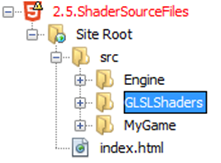


Figure 2-10. Creating the glsl\_shaders folder

1. Create two new text files within the glsl\_shaders folder and name them simple\_vs.glsl and white\_fs.glsl for simple vertex shader and white fragment shader.

**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.

1. Create the GLSL vertex shader source code by editing simple\_vs.glsl and pasting the vertex shader code in the index.html file from the previous project.

attribute vec3 aVertexPosition; // Vertex shader expects one vertex position

void main(void) {

// Convert the vec3 into vec4 for scan conversion and

// assign to gl\_Position to pass the vertex to the fragment shader

gl\_Position = vec4(aVertexPosition, 1.0);

}

1. Create the GLSL fragment shader source code by editing white\_fs.glsl and pasting the fragment shader code in the index.html file from the previous project.

precision mediump float; // sets the precision for floating point computation

void main(void) {

// for every pixel called (within the square) sets

// constant color white with alpha-channel value of 1.0

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>

<html>

<head>

<title>Example 2.5: The Shader Source File Project</title>

<link rel ="icon" type ="image/x-icon" href="./favicon.png">

<!-- the following says there are javascript source code contained in

the external source files

-->

<!-- Client game code -->

<script type="module" src="./src/my\_game/my\_game.js"></script>

</head>

<body>

<canvas id="GLCanvas" width="640" height="480">

<!-- GLCanvas is the area we will draw in: a 640x480 area. -->

Your browser does not support the HTML5 canvas.

<!-- this message will show only if WebGL clearing failed -->

</canvas>

</body>

</html>

Notice that index.html no longer contains any GLSL shader code and only a single reference to JavaScript code. With this organization, the index.html file can properly be considered as representing the web page where you do not need to edit this file to modify the shaders from now on.

1. Modify the createShader() function in core.js to load the shader files instead of HTML element IDs.

function createShader() {

mShader = new SimpleShader(

"src/glsl\_shaders/simple\_vs.glsl", // Path to the VertexShader

"src/glsl\_shaders/white\_fs.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/glsl\_shaders: This is the folder that contains all the GLSL shader source code files that draws the elements of your game.
* src/engine: This is the folder that contains all the source code for your game engine.
* src/my\_game: 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, white\_fs.glsl, with a shader that can be parameterized to draw with any color.

## The Parameterized Fragment Shader Project

This project replaces white\_fs.glsl with a simple\_fs.glsl that supports the drawing with any color. Figure 2-11 shows the output of running the Parameterized Fragment Shader project; notice that a red square replaces the white square from previous projects. The source code for this project is defined in the chapter2/2.6.parameterized\_fragment\_shader folder.

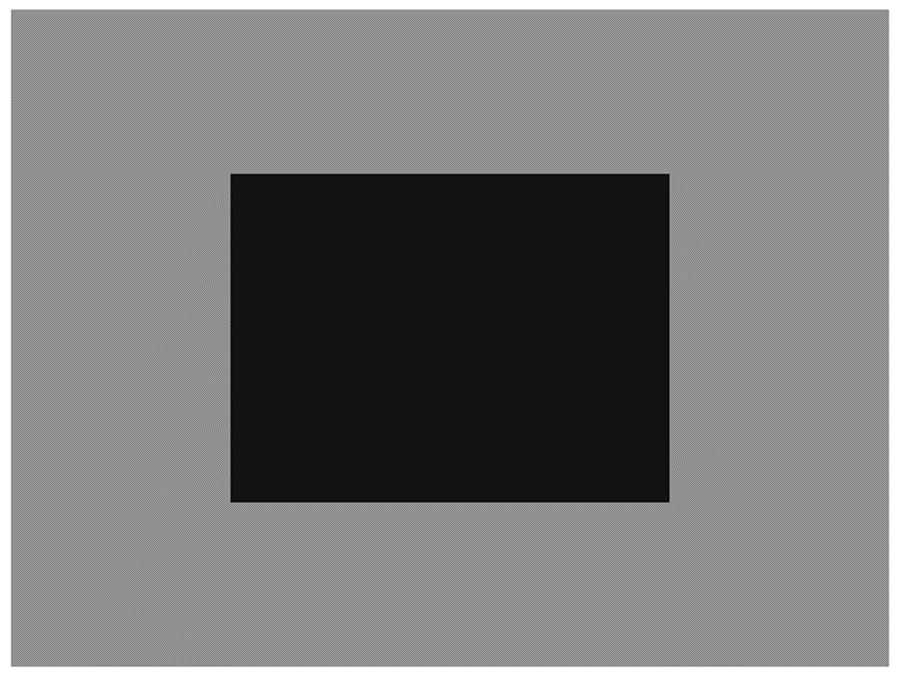


Figure 2-11. Running the Parameterized Fragment Shader project

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 simple\_fs.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/glsl\_shaders folder and name it simple\_fs.glsl. Edit this file to add the following:

precision mediump float; // sets the precision for floating point computation

// Color of pixel

uniform vec4 uPixelColor;

void main(void) {

// for every pixel called sets to the user specified color

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 class can now be modified to gain access to the new uPixelColor variable.

1. Edit simple\_shader.js and add a new instance variable for referencing the uPixelColor in the constructor.

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

1. Add code to the end of the constructor to create the reference to the uPixelColor.

// Step E: Gets a reference to the uniform variable uPixelColor in the fragment shader

this.mPixelColorRef = gl.getUniformLocation(this.mCompiledShader, "uPixelColor");

1. Modify the shader activation to allow the setting of the pixel color via the uniform4fv() function.

activate(pixelColor) {

let gl = core.getGL();

gl.useProgram(this.mCompiledShader);

// bind vertex buffer

gl.bindBuffer(gl.ARRAY\_BUFFER, vertexBuffer.get());

gl.vertexAttribPointer(this.mVertexPositionRef,

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

gl.enableVertexAttribArray(this.mVertexPositionRef);

// load uniforms

gl.uniform4fv(this.mPixelColorRef, pixelColor);

}

The gl.uniform4fv() function copies four floating-point values from the pixelColor float array to the WebGL location referenced by mPixelColor, or the uPixelColor in the simple\_fs.glsl fragment shader.

### Drawing with the New Shader

To test simple\_fs.glsl, modify the core.js module to use the parametrized color when drawing with the new shader.

function drawSquare(color) {

// Step A: Activate the shader

mShader.activate(color);

// Step B: Draw with the currently activated geometry and the activated shader

mGL.drawArrays(mGL.TRIANGLE\_STRIP, 0, 4);

}

Lastly, edit the constructor of the MyGame class to include a color when drawing the square. In this case red.

// Step C: Draw the square in red

engine.drawSquare([1, 0, 0, 1]);

Notice that a color value, an array of four floats, is now required with the new simple\_fs.glsl (instead of white\_fs) shader and that it is important to pass in the drawing color when activating the shader. With the new simple\_fs, 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 the simple\_shader.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.