Working with HTML5 and WebGL

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 itself, such as mechanics, logic, and aesthetics.

WebGL is a modern JavaScript graphical application programming interface (API) designed for web browser–based applications that offers quality and efficiency via direct access to the graphics 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 programming, and builds the foundational infrastructure to organize a complex source code system to support future expansion.

At the end of this lesson, students should be able to do the following…

- Create a new JavaScript source code file for your simple game engine
- Draw a simple constant color square with WebGL
- Define JavaScript modules and classes to encapsulate and implement core game engine functionality

Lab 2.1
The HTML5 Canvas Project

Creating and Clearing the HTML Canvas

To draw, you must first define and dedicate an area within the web page. You can achieve this easily by using the HTML `canvas` element to define an area for WebGL drawing. The `canvas` element is a container for drawing that you can access and manipulate with JavaScript.

1. Create a new project by creating a new folder named `CH2.1` in your chosen directory and copying and pasting the `index.html` file you created in the previous project in lesson 1.

Note: From this point on, when asked to create a new project, you should follow the process described previously. That is, create a new folder with the project's name and copy/paste the previous project's files. In this way, your new projects can expand upon your old ones while retaining the original functionality.

2. Open the `index.html` file in the editor by opening the `CH2.1` folder, expanding it if needed and clicking the `index.html` file.

3. Create the HTML canvas for drawing by adding the following lines in the `index.html` file within the body element (between the `<body>` and `</body>` tags:

```
<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 these lessons, "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 (be sure to put this after the creation of the canvas element as the JavaScript will reference the previously created canvas):

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

This takes care of the HTML portion of this project. You will now write JavaScript code for the remainder of the example:

Note The portion in **bold** will be replaced by JavaScript code.

5. Retrieve a reference to the `GLCanvas` in JavaScript code 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`.

The use of `document` as a predefined JavaScript object comes from DOM (Document Object Model). We are going to be using DOM quite a bit so it is important to understand this JavaScript concept. Information about DOM can be found at
https://www.w3schools.com/js/js_htmldom.asp

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

```
let gl = canvas.getContext("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 these lessons, the term WebGL will be used to refer to the WebGL version 2.0 API.

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

This code checks to ensure that 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 which is used to identify the level of opacity. 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 percent) intensity on the green channel. For this reason, the canvas area is cleared to a light green color.

WebGL uses buffers to facilitate drawing on the screen. We indicate which buffer(s) we want to clear with a bitwise OR of one or more constants provided by the API (in this case COLOR_BUFFER_BIT to clear the buffers for color writing).

Note: A buffer is a chunk of memory where information is stored. OpenGL uses many buffers serving different purposes to facilitate drawing.

8. 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>");
```

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.

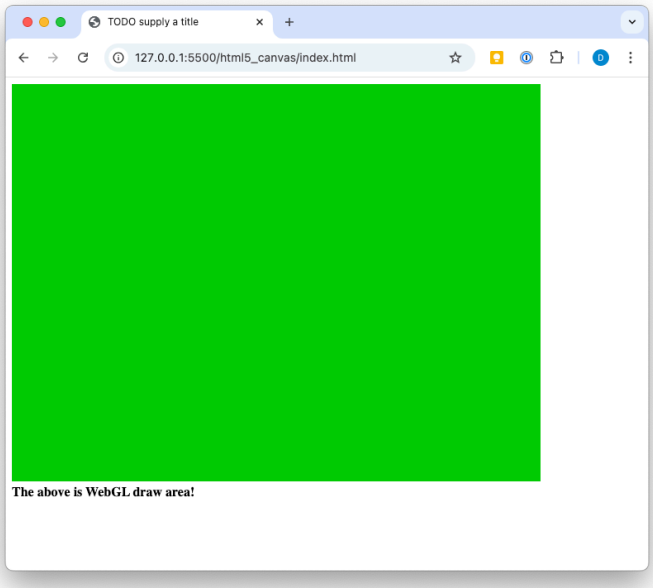

Figure 1. The 640x480 WebGL Canvas

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.

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 more complex systems. From this point on, all JavaScript source code will be added to separate source code files.

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, 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 an object-oriented programming style. To begin this process, the HTML and JavaScript source code from the previous project will be separated into different source code files.

Lab 2.2
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 these lessons. Our goal in this section is to separate our JavaScript code from our HTML code. We will organize our JavaScript code into multiple files contained within a folder called `src`. Then we will learn how to have an HTML file reference the code written in other files.

1. Create a new HTML5 project titled `CH2.2`.

Note 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. This folder will contain all of your source code.

3. Create a new source code file within the `src` folder by right-clicking the `src` folder and selecting **New File…** (this can also be done from the **File** menu). Name the new source file `core.js`.

Note: In VS Code, you can create/copy/rename folders and files by using the right-click menus in the Explorer window.

4. Open the new `core.js` source file for editing. 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`.

5. 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");
   if (mGL === null) {
      document.write("<br><b>WebGL 2 is not supported!</b>");
      return;
   }
   mGL.clearColor(0.0, 0.8, 0.0, 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 function names begin with a lowercase letter, as in `initWebGL()`.

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

```
function clearCanvas() {
   mGL.clear(mGL.COLOR_BUFFER_BIT);
}
```

8. 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 and remove all JavaScript functionality.

Note If you copied the `index.html` file from a previous project, you need to **remove the JavaScript functionality** previously defined to see the proper results.

2. We need to keep the `GLCanvas` as was created 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.

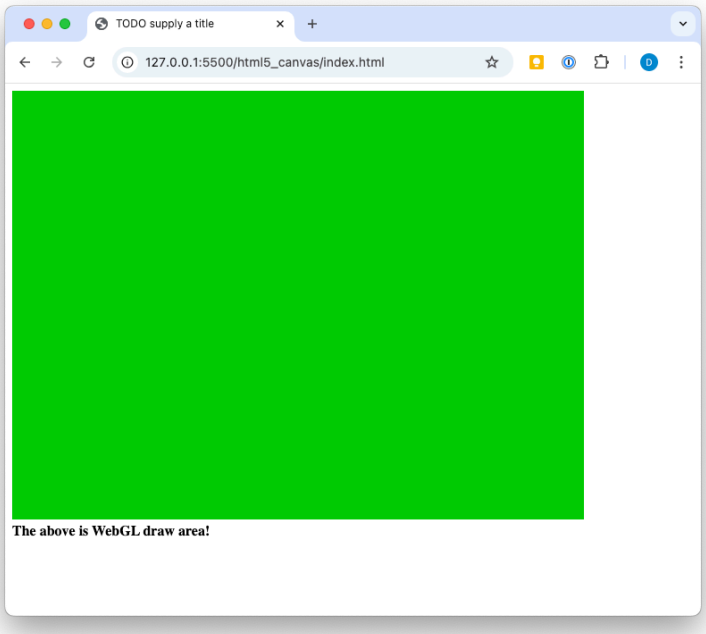

Figure 1. The 640x480 WebGL Canvas

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.

Note: Recall that to run a project, you click the "Go Live" button on the lower right of the VS Code window, while the associated `index.html` file is opened in the Editor window. In this case, the project will not run if you click the "Go Live" button while the `core.js` file is opened in the Editor window.

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 more complex systems. From this point on, all JavaScript source code will be added to separate source code files.

Lab 2.3
The Draw One Square Project

In this lab we are going to draw in the canvas. To start, we are going to draw a simple square. Drawing in WebGL involves a number of components. We are going to focus on two components to draw a simple square in the canvas to start building our engine ...

- Geometric data
- Instructions for processing the data

Geometric Data
The geometric data is an array of the vertex coordinates in 3D space that encompass the object we want to draw. As we are making a 2D game engine, we will always have one coordinate component anchored at zero. The coordinates of each object will be stored in an array and that array will be passed to OpenGL to create a **vertex buffer**.

Instructions for processing the data
In the case of WebGL, the instructions for processing the data are specified in the OpenGL Shading Language (GLSL) and are referred to as **shaders**. Shaders are blocks of code that manipulate data (typically in buffers).

In order to draw with WebGL, programmers must define the geometric data and GLSL shaders in the CPU and load both to the drawing hardware, or the graphics 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 involved when building your game engine.

In the following project, you will learn about drawing with WebGL by focusing on the most elementary operations. This includes the loading of the simple geometry of a square from the CPU to the GPU, the creation of a constant color shader, and the basic instructions for drawing a simple square with two triangles. Our goals…

- 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 required to draw with WebGL
- To demonstrate the implementation of a singleton-like JavaScript module based on simple source code files

Set up and load 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 1x1 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` 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 project, `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.

3. 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; }
```

4. 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 is 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.

Note: The coordinates for the square are ordered top-right, top-left, bottom-right, bottom-left essentially forming a zig-zag pattern. This identifies the vertices of the triangle mesh needed to draw a square.

5. Define the `init()` function to allocate a buffer in the GPU via the `gl` context, and load the vertices to the allocated buffer in 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. After which,
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 binding it to the OpenGL pipeline.
Step C loads the vertex position of the square into the activated buffer on the GPU. The keyword `ARRAY_BUFFER` identifies buffer as holding an array. The `Float32Array()` function creates a new array of 32-bit floats and populates it using the JavaScript array we created. 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.

6. 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, or a collection of instructions, 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 GPU memory the source code for both vertex and fragment shaders, compile and link them into a single shader program, and load the linked program into the GPU memory for drawing. 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

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;  // Expects one vertex position

   void main(void) {
      // Convert the vec3 into vec4 for scan conversion and
      // assign to gl_Position to pass 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 the vertex positions we stored in the vertex buffer 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.

Note: The vertices are stored as `vec4` with 4 components with the last component always being 1.0. The reason for this is to faciliate matrix multiplication to implement vertex and vector translations.

2. 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` within the `src` folder.
2. Import functionality from the `core.js` and `vertex_buffer.js` files:

```
"use strict";               // Variables must be declared before used!
import * as core from "./core.js";      // access as core module
import * as vertexBuffer from "./vertex_buffer.js"; //vertexBuffer module
```

3. 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;
```

4. 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 shader based on 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 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.

5. You are now ready to create, compile, and link 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 reference to aVertexPosition attribute in the shader
  mVertexPositionRef = gl.getAttribLocation(mCompiledShader,
                          "aVertexPosition");
}
```

Step A loads and compiles the shader code you defined in `index.html` by calling the `loadAndCompileShader()` function with the corresponding parameters.
Step B attaches the compiled shaders 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.

6. 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 vertex buffer to attribute defined in 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.mVertexPositionRef);
}
```

In the code shown,
Step A sets the `gl` variable to the WebGL context through the core module.
Step B loads the compiled shader program to the GPU memory
Step C binds the vertex buffer created in `vertex_buffer.js` to the `aVertexPosition` attribute defined in the vertex shader. The `gl.vertexAttribPointer()` 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.

7. 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 thus, following the good development practice of hiding local implementation details, should remain private to this file.

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 shader 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 into `core.js`:

```
import * as vertexBuffer from "./vertex_buffer.js";
import * as simpleShader from "./shader_support.js";
```

2. 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 standard or experimental webgl and bind to the Canvas area
  // store the results to the instance variable mGL
  mGL = canvas.getContext("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 buffer with vertex positions for the unit square
  vertexBuffer.init(); // function defined in the vertex_buffer.js
  // 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 then call the `init()` function defined in `shader_support.js` to load and compile the vertex and fragment shaders.

3. 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. The `TRIANGE_STRIP` keyword is used to issue a command to draw the four vertices as two connected triangles that form a square. The 0 indicates the first index in the vertex array buffer and 4 indicates the total number of vertices we wish to draw (in this case, all of them).

4. 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
}
```

5. 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.

When you run the project, and you will see a white rectangle on a green canvas.

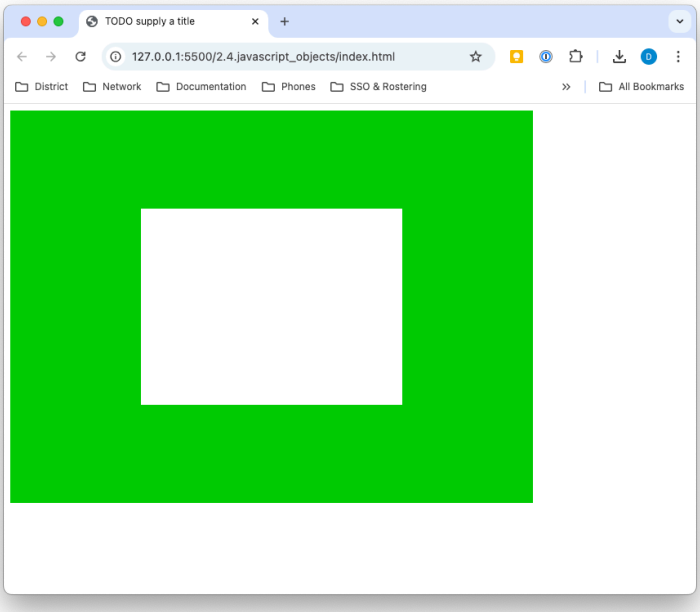

Figure 2. Canvas with Square

What happened to the square? Remember that the vertex positions 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' 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 (which follows the created canvas dimension of 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.

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 does not result in the white square becoming transparent. Transparency of drawn primitives will be discussed in later chapters.

Finally, note that this project defines three separate files and hides information with the JavaScript import/export statements. The functionality defined in these files with the corresponding import and export statements is referred to as JavaScript modules. A module can be considered as a global singleton object and is excellent for hiding implementation details. The `loadAndCompileShader()` function in the `shader_support` module serves as a great example of this concept. However, modules are not well 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.

Lab 2.4
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 object-oriented abstraction will result in a framework that offers manageability and expandability for subsequent projects. Our goals will be:

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

Source Code Organization

Create a new HTML5 project with VS Code by creating a new folder and adding a source code folder named `src`. Within `src`, create `engine` and `my_game` as subfolders.

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.

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 the internal representations of shaders. Besides the ability to create multiple instances of the `SimpleShader` object, the basic functionality remains largely unchanged.

Note Module identifiers begin with lower case, for example, `core` or `vertexBuffer`. Class names begin with upper case, for example, `SimpleShader` or `MyGame`.

1. Create a new source file in the `src/engine` folder and name the file `simple_shader.js` to implement the `SimpleShader` class.

2. Import both the `core` and `vertex_buffer` modules:

```
import * as core from "./core.js";
import * as vertexBuffer from "./vertex_buffer.js";
```

3. Declare the `SimpleShader` as a JavaScript class:

```
class SimpleShader {
... implementation to follow ...
}
```

Note We are going to modify existing files to create a better organizational structure and add functionality. When there are changes to existing code, these will be in **bold** formatting.

4. 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. Remember, like Java, these functions need to be implemented with the class context. This code will be placed where the "… implementation to follow …" section is mentioned above.

```
constructor(vertexShaderID, fragmentShaderID) {
   // instance variables
   // Convention: all instance variables: mVariables
   this.mCompiledShader = null;  // ref to compiled shader in webgl
   this.mVertexPositionRef = null; // ref to VertexPosition in 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: reference to aVertexPosition attribute in the shaders
   this.mVertexPositionRef = gl.getAttribLocation(
            this.mCompiledShader, "aVertexPosition");
}
```

Notice that this constructor is essentially the same as the `init()` function in the `shader_support.js` module from the previous project.

Note The JavaScript `constructor` keyword defines the constructor of a class.

5. Add a method to the `SimpleShader` class to activate the shader for drawing. Once again, similar to the `activate()` function in `shader_support.js` from the 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);
}
```

6. Add a private method, which cannot be accessed 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 shader based on 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 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`.

Note The JavaScript **#** prefix that defines private members is not used because the lack of visibility from subclasses complicates specialization of behaviors in inheritance.

7. 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 be changed by `import` statements.

The Core of the Game Engine: core.js

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

1. Relocate the `core.js` and `vertex_buffer.js` files to the new folder `src/engine` (`shader_support.js` is not needed as all of its functionality has been replicated in `simple_shader.js`).

2. At the top of the `core.js` file, remove the import for `shader_support.js` and add an import for the `SimpleShader` class defined earlier:

```
import * as vertexBuffer from "./vertex_buffer.js";
import SimpleShader from "./simple_shader.js";
```

3. Define 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 tags in index.html
    "FragmentShader");     // for the shaders
}
```

4. Modify the `initWebGL()` to no longer include setting the clear color, vertex buffer initialization or shader functionality and to focus on only initializing the WebGL. The removed functionality will be handled by new functions we will soon create. The complete function is as follows:

```
// initialize the WebGL
function initWebGL(htmlCanvasID) {
  let canvas = document.getElementById(htmlCanvasID);
  // Get standard or experimental webgl and binds to the Canvas area
  // store the results to the instance variable mGL
  mGL = canvas.getContext("webgl2");
  if (mGL === null) {
    document.write("<br><b>WebGL 2 is not supported!</b>");
    return;
  }
}
```

5. Create an `init()` function to perform engine-wide system initialization, which includes initializing of 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
}
```

6. 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]);
  mGL.clear(mGL.COLOR_BUFFER_BIT);     // clear to the color set
}
```

The `color` parameter is an array and the caller will need to make sure the array contains 4 elements and the `clearColor()` method requires each element to be of the OpenGL type `GLclampf`. This is essentially a floating-point number between 0.0 and 1.0 inclusive.

7. Modify the `drawSquare()` function to use the `SimpleShader` instance created earlier:

```
function drawSquare() {
  // Step A: Activate the shader
  mShader.activate();

  // Step B. draw with the above settings
  mGL.drawArrays(mGL.TRIANGLE_STRIP, 0, 4);
}
```

8. Export the relevant functions for access by the rest of the game engine:

```
export { getGL, init, clearCanvas, drawSquare }
```

9. Finally, remove the `window.onload` function as the behavior of the actual game should be defined by the client of the game engine or, in this case, the `MyGame` class.

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. 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";
```

3. Define `MyGame` as a JavaScript class and add a `constructor` to initialize the game `engine`, clear the `canvas`, and draw the 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();
   }
}
```

4. Bind the creation of a new instance of the `MyGame` object to the `window.onload` function:

```
window.onload = function() {
   new MyGame('GLCanvas');
}
```

5. 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>
```

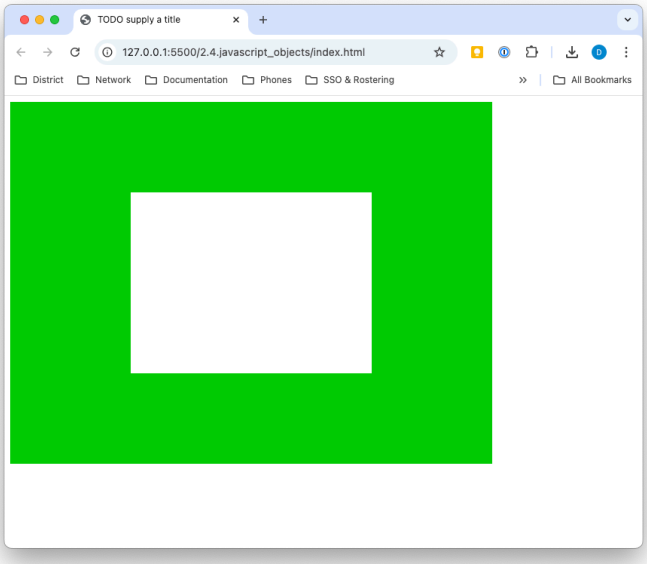

Figure 3. Canvas with Square

Lab 2.5
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 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 code files during runtime

Note As we separate the project into different files, we will be changing the code on existing files. These changes will be notated in **bold**. Make sure to follow those changes VERY CAREFULLY.

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. Create `CH2.5` from the previous project, open the `simple_shader.js` file, and edit the `loadAndCompileShader()` function, to receive a file path instead of an HTML ID:

```
function loadAndCompileShader(filePath, shaderType)
```

2. 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 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 later 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 VS 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`.

2. 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.

3. Create the GLSL vertex shader source code by editing `simple_vs.glsl` and pasting the vertex shader code from the `index.html` file from the previous project:

```
attribute vec3 aVertexPosition;  // Vertex shader expects one 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);
}
```

4. Create the GLSL fragment shader source code by editing `white_fs.glsl` and pasting the fragment shader code from the `index.html` file from the previous project:

```
precision mediump float; // precision for float 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">
      <!-- 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.

2. 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 VertexShader
     "src/glsl_shaders/white_fs.glsl"); // Path to 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 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 files for your game engine.
- `src/my_game`: This is the client folder that contains the source code for the actual game.

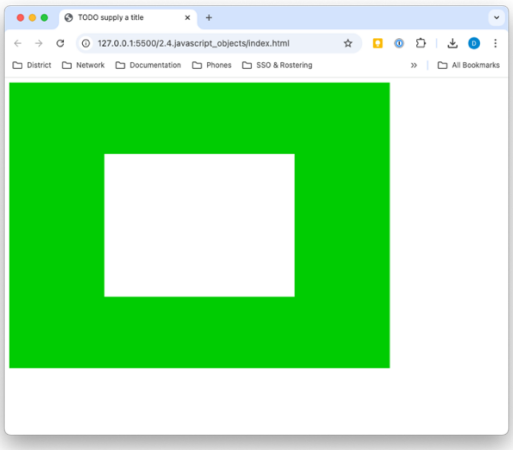

Figure 4. Canvas with White Square

Lab 2.6
The Parameterized Fragment Shader Project

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. We will replace the `white_fs.glsl` file with `simple_fs.glsl` that supports drawing with any color.

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

Create a new HTML5 project called `CH2.6` by copying from the previous project. You can delete the `white_fs.glsl` from the `glsl_shaders` subfolder.

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; // precision for float 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; // pixelColor uniform in fragment shader
```

2. Add code to the end of the constructor to create the reference to the `uPixelColor`:

```
// Step E: Gets uniform variable uPixelColor in fragment shader
this.mPixelColorRef = gl.getUniformLocation(
            this.mCompiledShader, "uPixelColor");
```

3. 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 `mPixelColorRef` 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 create a `SimpleShader` with the new `simple_fs` and use the parameterized color when drawing with the new shader:

```
function createShader() {
  mShader = new SimpleShader(
       "src/glsl_shaders/simple_vs.glsl", // Path to the VertexShader
       "src/glsl_shaders/simple_fs.glsl"); // Path to the FragmentShader
}

function drawSquare(color) {
  // Step A: Activate the shader
  mShader.activate(color);
  // Step B: Draw with currently activated geometry and shader
  mGL.drawArrays(mGL.TRIANGLE_STRIP, 0, 4);
}
```

Lastly, edit the `constructor` of the `MyGame` class in `my_game.js` 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 or modified. In this case, only changes to the `simple_shader.js` file and minor modifications to `core.js` and the `my_game.js` were required. This demonstrates the benefit of proper encapsulation and source code organization.

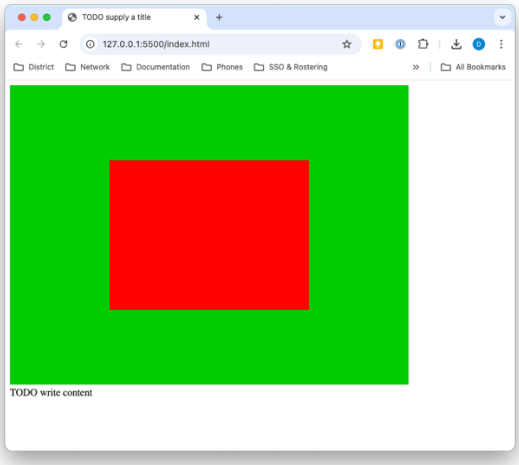

Figure 5. Canvas with Red Square

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 required to build an excellent foundation for the game engine. You have also structured the source code 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.