Drawing Objects in the World

Introduction

Ideally, a video game engine should provide proper abstractions to support designing and building games in meaningful contexts. For example, when designing a soccer game, instead of a single square with a fixed ±1.0 drawing range, a game engine should provide proper utilities to support designs in the context of players running on a soccer field. This high-level abstraction requires the encapsulation of basic operations with data hiding and meaningful functions for setting and receiving the desired results.

This chapter focuses on creating the fundamental abstractions to support drawing. Based on the soccer game example, the support for drawing in an effective game engine would likely include the ability to easily create the soccer players, control their size and orientations, and allow them to be moved and drawn on the soccer field. Additionally, to support proper presentation, the game engine must allow drawing to specific subregions on the canvas so that a distinct game status can be displayed at different subregions, such as the soccer field in one subregion and player statistics and scores in another subregion.

This chapter identifies proper abstraction entities for the basic drawing operations, introduces operators that are based on foundational mathematics to control the drawing, overviews the WebGL tools for configuring the canvas to support subregion drawing, defines JavaScript classes to implement these concepts, and integrates these implementations into the game engine while maintaining the organized structure of the source code.

Encapsulating Drawing

Although the ability to draw is one of the most fundamental functionalities of a game engine, the details of how drawings are implemented are generally a distraction to gameplay programming. For example, it is important to create, control the locations of, and draw soccer players in a soccer game. However, exposing the details of how each player is actually defined (by a collection of vertices that form triangles) can quickly overwhelm and complicate the game development process. Thus, it is important for a game engine to provide a well-defined abstraction interface for drawing operations.

With a well-organized source code structure, it is possible to gradually and systematically increase the complexity of the game engine by implementing new concepts with localized changes to the corresponding folders. The first task is to expand the engine to support the encapsulation of drawing such that it becomes possible to manipulate drawing operations as a logical entity or as an object that can be rendered.

> **Note** In the context of computer graphics and video games, the word render refers to the process of changing the color of pixels corresponding to an abstract representation. For example, in the previous chapter, you learned how to render a square.

Lab 3.1
The Renderable Objects Project

This project introduces the `Renderable` class to encapsulate the drawing operation. Over the next few projects, you will learn more supporting concepts to refine the implementation of the `Renderable` class such that multiple instances can be created and manipulated.

The goals of the project are as follows:

- To reorganize the source code structure in anticipation for functionality increases
- To support game engine internal resource sharing
- To introduce a systematic interface for the game developer via the `index.js` file
- To begin the process of building a class to encapsulate drawing operations by first abstracting the related drawing functionality
- To demonstrate the ability to create multiple `Renderable` objects

Source Code Structure Reorganization
Before introducing additional functionality to the game engine, it is important to recognize some shortfalls of the engine source code organization from the previous project. In particular, take note of the following:

1. The `core.js` source code file contains the WebGL interface, engine initialization, and drawing functionalities. These should be modularized to support the anticipated increase in system complexity.
2. A system should be defined to support the sharing of game engine internal resources. For example, `SimpleShader` is responsible for interfacing from the game engine to the GLSL shader compiled from the `simple_vs.glsl` and `simple_fs.glsl` source code files. Since there is only one copy of the compiled shader, there only needs to be a single instance of the `SimpleShader` object. The game engine should facilitate this by allowing the convenient creation and sharing of the object.
3. As you have experienced, the JavaScript `export` statement can be an excellent tool for hiding detailed implementations. However, it is also true that determining which classes or modules to import from a number of files can be confusing and overwhelming in a large and complex system, such as the game engine you are about to develop. An easy to work with and systematic interface should be provided such that the game developer, users of the game engine, can be insulated from these details.

In the following section, the game engine source code will be reorganized to address these issues.

Define a WebGL-Specific Module
The first step in source code reorganization is to recognize and isolate functionality that is internal and should not be accessible by the clients of the game engine:

1. In your project, under the `src/engine` folder, create a new folder and name it `core`. From this point forward, this folder will contain all functionality that is internal to the game engine and will not be exported to the game developers.
2. Create a new source code file in the `src/engine/core` folder, name it `gl.js`, and define WebGL's initialization and access methods:

```
"use strict"
let mCanvas = null;
let mGL = null;

function get() { return mGL; }

function init(htmlCanvasID) {
  mCanvas = document.getElementById(htmlCanvasID);
  if (mCanvas == null)
   throw new Error("Engine init [" +
                    htmlCanvasID + "] HTML element id not found");
  // Get webgl and bind to the Canvas area ands
  // store the results to the instance variable mGL
  mGL = mCanvas.getContext("webgl2");
  if (mGL === null) {
   document.write("<br><b>WebGL 2 is not supported!</b>");
   return;
  }
}

export {init, get}
```

3. You can move the `vertex_buffer.js` source code file from the previous project into the `src/engine/core` folder. The details of the primitive vertices are internal to the game engine and should not be visible or accessible by the clients of the game engine.

4. Modify the `import` of `vertex_buffer.js` to reference the new `glSys` instead of `core` (the file `core.js` can be deleted).

```
import * as glSys from "./gl.js";
```

5. Modify the `import` of `simple_shader.js` to reference the new `glSys` instead of `core` and to reference the new location of `vertex_buffer.js`.

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

6. Modify the `let` statement in the `init()` function of `vertex_buffer.js`. Also modify ALL OCCURENCES of this let statement in `simple_shader.js` within 1) the `constructor`, 2) the `loadAndCompileShader()` function and 3) the `activate()` functions to reference the new `get()` function provided by `glSys`.

```
let gl = core.getGL();        ➔     let gl = glSys.get();
```

Notice that the `init()` function is identical to the `initWebGL()` function in `core.js` from the previous project. Unlike the previous `core.js` source code file, the `gl.js` file contains only WebGL-specific functionality.

Define a System for Internal Shader Resource Sharing

Since only a single copy of the GLSL shader is created and compiled from the `simple_vs.glsl` and `simple_fs.glsl` source code files, only a single copy of `SimpleShader` object is required within the game engine to interface with the compiled shader. You will now create a simple resource sharing system to support future additions of different types of shaders.

Create a new source code file in the `src/engine/core` folder, name it `shader_resources.js`, and define the creation and accessing methods for `SimpleShader`.

> **Note** Recall from the previous chapter that the `SimpleShader` class is defined in the `simple_shader.js` file which is located in the `src/engine` folder. Remember to copy all relevant source code files from the previous project.

```
"use strict";
import SimpleShader from "../simple_shader.js";
// Simple Shader
let kSimpleVS = "src/glsl_shaders/simple_vs.glsl"; // to VertexShader
let kSimpleFS = "src/glsl_shaders/simple_fs.glsl"; // to FragmentShader
let mConstColorShader = null;

function createShaders() {
  mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);
}

function init() {
  createShaders();
}

function getConstColorShader() { return mConstColorShader; }

export {init, getConstColorShader}
```

Note Variables referencing constant values have names that begin with lowercase "k", as in `kSimpleVS`.

Since the `shader_resources` module is located in the `src/engine/core` folder, the defined shaders are shared within and cannot be accessed from the clients of the game engine.

Define an Access File for the Game Developer

You will define an engine access file, `index.js`, to implement the fundamental functions of the game engine and to serve a similar purpose as a C++ header file, the `import` statement in Java, or the `using` statement in C#, where functionality can be readily accessed without in-depth knowledge of the engine source code structure. That is, by importing `index.js`, the client can access all the components and functionality from the engine to build their game.

1. Create `index.js` file in the `src/engine` folder; `import` from `gl.js`, `vertex_buffer.js`, and `shader_resources.js`; and define the `init()` function to initialize the game engine by calling the corresponding `init()` functions of the three imported modules:

```
// local to this file only
import * as glSys from "./core/gl.js";
import * as vertexBuffer from "./core/vertex_buffer.js";
import * as shaderResources from "./core/shader_resources.js";

// general engine utilities
function init(htmlCanvasID) {
   glSys.init(htmlCanvasID);
   vertexBuffer.init();
   shaderResources.init();
}
```

2. Define the `clearCanvas()` function to clear the drawing canvas:

```
function clearCanvas(color) {
   let gl = glSys.get();
   gl.clearColor(color[0], color[1], color[2], color[3]);
   gl.clear(gl.COLOR_BUFFER_BIT); // clear to the color set
}
```

3. Now, to properly expose the `Renderable` symbol to the clients of the game engine, we import such that the class can be properly exported (the `Renderable` class will be introduced in details in the next section). Add the following among the imports:

```
// general utilities
import Renderable from "./renderable.js";
```

4. Finally, remember to export the proper symbols and functionality for the clients of the game engine:

```
export  default {
  // Util classes
  Renderable,
  // functions
  init, clearCanvas
}
```

With proper maintenance and update of this `index.js` file, the clients of your game engine, the game developers, can simply import from the `index.js` file to gain access to the entire game engine functionality without any knowledge of the source code structure. Lastly, notice that the `glSys`, `vertexBuffer`, and `shaderResources` internal functionality defined in the `engine/src/core` folder are not exported by `index.js` and thus are not accessible to the game developers.

The Renderable Class

At last, you are ready to define the `Renderable` class to encapsulate the drawing process:

1. Define the `Renderable` class in the game engine by creating a new source code file in the `src/engine` folder, and name the file `renderable.js`.

2. Open `renderable.js`, import from `gl.js` and `shader_resources.js`, and define the `Renderable` class with a constructor to initialize a reference to a shader and a color instance variable. Notice that the shader is a reference to the shared `SimpleShader` instance defined in `shader_resources`.

```
import * as glSys from "./core/gl.js";
import * as shaderResources from "./core/shader_resources.js";

class Renderable {
  constructor() {
    this.mShader = shaderResources.getConstColorShader();
    this.mColor = [1, 1, 1, 1]; // color of pixel
  }
... implementation to follow ...
}
```

3. As part of the implementation, define a `draw()` function for `Renderable`:

```
draw() {
  let gl = glSys.get();
  this.mShader.activate(this.mColor);
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
}
```

Notice that it is important to activate the proper GLSL shader in the GPU by calling the `activate()` function before sending the vertices with the `gl.drawArrays()` function.

4. Define the getter and setter functions for the color instance variable:

```
setColor(color) {this.mColor = color; }
getColor() { return this.mColor; }
```

5. Export the `Renderable` symbol as default to ensure this identifier cannot be renamed:

```
export default Renderable;
```

Though this example is simple, it is now possible to create and draw multiple instances of the `Renderable` objects with different colors.

Testing the Renderable Object

To test `Renderable` objects in `MyGame`, we replace the existing constructor and with one that creates white and red instances are created and drawn as follows:

```
// import from engine/index.js for all engine symbols
import engine from "../engine/index.js";

class MyGame {
  constructor(htmlCanvasID) {
   // Step A: Initialize the webGL Context
   engine.init(htmlCanvasID);
   // Step B: Create the Renderable objects:
   this.mWhiteSq = new engine.Renderable();
   this.mWhiteSq.setColor([1, 1, 1, 1]);
   this.mRedSq = new engine.Renderable();
   this.mRedSq.setColor([1, 0, 0, 1]);
   // Step C: Draw!
   engine.clearCanvas([0, 0.8, 0, 1]);  // Clear the canvas
   // Step C1: Draw Renderable objects with the white shader
   this.mWhiteSq.draw();
   // Step C2: Draw Renderable objects with the red shader
```

```
      this.mRedSq.draw();
    }
  }

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

Notice that the `import` statement is modified to import from the engine access file, `index.js`. Additionally, the `MyGame` constructor is modified to include the following steps:

1. Step A initializes the engine.
2. Step B creates two instances of `Renderable` and sets the colors of the objects accordingly.
3. Step C clears the canvas; steps C1 and C2 simply call the respective `draw()` functions of the white and red squares. Although both of the squares are drawn, for now, you are only able to see the last of the drawn squares in the canvas. Please refer to the following discussion for the details.

Observations

Run the project and you will notice that only the red square is visible! What happens is that both of the squares are drawn to the same location. Being the same size, the two squares simply overlap perfectly. Since the red square is drawn last, it overwrites all the pixels of the white square. You can verify this by commenting out the drawing of the red square (comment out the line `mRedSq.draw()`) and rerunning the project. An interesting observation to make is that objects that appear in the front are drawn last (the red square). You will take advantage of this observation much later when working with transparency.

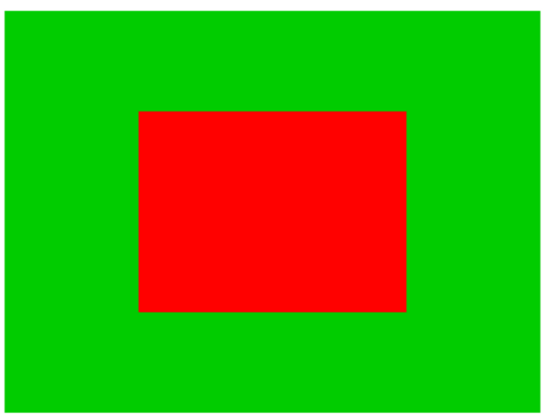

Figure 3-1. Canvas with Square

This simple observation leads to your next task—to allow multiple instances of `Renderable` to be visible at the same time. Each instance of `Renderable` object needs to support the ability to be drawn at different locations, with different sizes and orientations so that they do not overlap one another.

Matrices as Transform Operators

Before we begin, it is important to recognize that matrices and transformations are general topic areas in mathematics. The following discussion does not attempt to include a comprehensive coverage of these subjects. Instead, the focus is on a small collection of relevant concepts and operators from the perspective of what the game engine requires. In this way, the coverage is on how to utilize the operators and not the theories. If you are interested in the specifics of matrices and how they relate to computer graphics, please refer to the discussion in Chapter 1 where you can learn more about these topics by delving into relevant books on linear algebra and computer graphics.

A matrix is an *m* rows by *n* columns 2D array of numbers. For the purposes of this game engine, you will be working exclusively with 4×4 matrices. While a 2D game engine could get by with 3×3 matrices, a 4×4 matrix is used to support features that will be introduced in the later chapters. Among the many powerful applications, 4×4 matrices can be constructed as transform operators for vertex positions. The most important and intuitive of these operators are the translation, scaling, rotation, and identity operators.

- The translation operator `T(tx,ty)`, as illustrated in Figure 3-2, translates or moves a given vertex position from (x, y) to (x+tx, y+ty). Notice that `T(0,0)` does not change the value of a given vertex position and is a convenient initial value for accumulating translation operations.

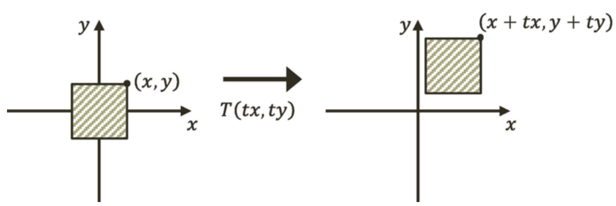

Figure 3-2. Translating a Square by *`T(tx,ty)`*

- The scaling operator `S(sx, sy)`, as illustrated by Figure 3-3, scales or resizes a given vertex position from (x, y) to (x×sx, y×sy). Notice that `S(1, 1)` does not change the value of a given vertex position and is a convenient initial value for accumulating scaling operations.

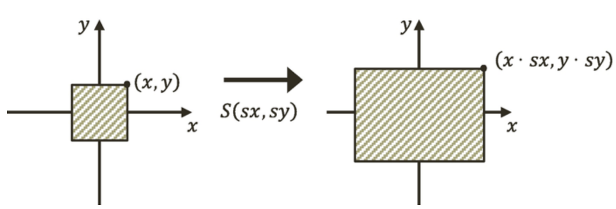

Figure 3-3. Scaling a Square by *`S(sx,sy)`*

- The rotation operator R(θ), as illustrated in Figure 3-4, rotates a given vertex position with respect to the origin.

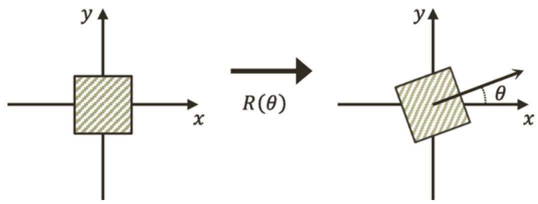

Figure 3-4. Scaling a Square by $R\ (\theta)$

In the case of rotation, R(0) does not change the value of a given vertex and is the convenient initial value for accumulating rotation operations. The values for θ are typically expressed in radians (and not degrees).

- The identity operator I does not affect a given vertex position. This operator is mostly used for initialization.

As an example, a 4×4 identity matrix looks like the following:

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Mathematically, a matrix transform operator operates on a vertex through a matrix-vector multiplication. To support this operation, a vertex position p = (x, y, z) must be represented as a 4x1 vector as follows:

$$p = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Note The z component is the third dimension, or the depth information, of a vertex position. In most cases, you should leave the z component to be 0.

For example, if position p' is the result of a translation operator T operating on the vertex position p, mathematically, p' would be computed by the following:

$$p' = T \times p = Tp$$

Concatenation of Matrix Operators

Multiple matrix operators can be concatenated, or combined, into a single operator while retaining the same transformation characteristics as the original operators. For example, you may want to apply the scaling operator S, followed by the rotation operator R, and finally the translation operator T, on a given vertex position, or to compute p' with the following:

$$p' = TRSp$$

Alternatively, you can compute a new operator `M` by concatenating all the transform operators, as follows:

$$M = TRS$$

And then operate `M` on vertex position `p`, as follows, to produce identical results:

$$p' = Mp$$

The `M` operator is a convenient and efficient way to record and reapply the results of multiple operators.

Finally, notice that when working with transformation operators, the order of operation is important. For example, a scaling operation followed by a translation operation is in general different from a translation followed by a scaling or, in general:

$$ST \neq TS$$

The glMatrix Library

The details of matrix operators and operations are nontrivial to say the least. Developing a complete matrix library is time-consuming and not the focus of this book. Fortunately, there are many well-developed and well-documented matrix libraries available in the public domain. The glMatrix library is one such example. To integrate this library into your source code structure, follow these steps:

1. Create a new folder under the `src` folder, and name the new folder `lib`.

2. Go to http://glMatrix.net , and download, unzip, and store the resulting `gl-Matrix.js` source file (found in the `dist` folder) into the new `lib` folder.

3. As a library that must be accessible by both the game engine and the client game developer, you will load the source file in the main `index.html` by adding the following before the loading of `my_game.js`:

```
<!-- external library -->
<script type="text/javascript" src="src/lib/gl-matrix.js"></script>
<!-- our game -->
<script type="module" src="./src/my_game/my_game.js"></script>
```

Lab 3.2
The Matrix Transform Project

This project introduces and demonstrates how to use transformation matrices as operators to manipulate the position, size, and orientation of `Renderable` objects drawn on the canvas. In this way, a `Renderable` can now be drawn to any location, with any size and any orientation.

The goals of the project are as follows:

- To introduce transformation matrices as operators for drawing a `Renderable`
- To understand how to work with the transform operators to manipulate a `Renderable`

Modify the Vertex Shader to Support Transforms

As discussed, matrix transform operators operate on vertices of geometries. The vertex shader is where all vertices are passed in from the WebGL context and is the most convenient location to apply the transform operations.

You will continue working with the previous project to support the transformation operator in the vertex shader:

1. Edit `simple_vs.glsl` to declare a uniform 4×4 matrix:

Note Recall from the discussion in Chapter 2 that glsl files contain OpenGL Shading Language (GLSL) instructions that will be loaded to and executed by the GPU. You can find out more about GLSL by referring to the WebGL and OpenGL references provided at the end of Chapter 1.

```
// to transform the vertex position
// Must be set outside the shader.
uniform mat4 uModelXformMatrix;
```

Recall that the `uniform` keyword in a GLSL shader declares a variable with values that do not change for all the vertices within that shader. In this case, the `uModelXformMatrix` variable is the transform operator for all the vertices.

Note GLSL uniform variable names always begin with lowercase "u", as in `uModelXformMatrix`.

2. In the `main()` function, apply the `uModelXformMatrix` to the currently referenced vertex position:

```
gl_Position = uModelXformMatrix * vec4(aVertexPosition, 1.0);
```

Notice that the operation follows directly from the discussion on matrix transformation operators. The reason for converting `aVertexPosition` to a `vec4` is to support the matrix-vector multiplication.

With this simple modification, the vertex positions of the unit square will be operated on by the `uModelXformMatrix` operator, and thus the square can be drawn to different locations. The task now is to set up `SimpleShader` to load the appropriate transformation operator into `uModelXformMatrix`.

Modify SimpleShader to Load the Transform Operator

Follow these steps:

1. Edit `simple_shader.js` and add an instance variable to hold the reference to the `uModelXformMatrix` matrix in the vertex shader (done in the constructor):

```
this.mModelMatrixRef = null;
```

2. At the end of the `SimpleShader` constructor under step E, after setting the reference to `uPixelColor`, add the following code to initialize this reference:

```
// Step E: Gets a reference to uniform variables in fragment shader
this.mPixelColorRef = gl.getUniformLocation(
                         this.mCompiledShader, "uPixelColor");
this.mModelMatrixRef = gl.getUniformLocation(
                         this.mCompiledShader, "uModelXformMatrix");
```

3. Modify the `activate()` function to receive a second parameter, and load the value to `uModelXformMatrix` via `mModelMatrixRef`:

```
activate(pixelColor, trsMatrix) {
   let gl = glSys.get();
   gl.useProgram(this.mCompiledShader);
       ... identical to previous code ...
   // load uniforms
   gl.uniform4fv(this.mPixelColorRef, pixelColor);
   gl.uniformMatrix4fv(this.mModelMatrixRef, false, trsMatrix);
}
```

The `gl.uniformMatrix4fv()` function copies the values from `trsMatrix` to the vertex shader location identified by `this.mModelMatrixRef` or the `uModelXfromMatrix` operator in the vertex shader. The name of the variable, `trsMatrix`, signifies that it should be a matrix operator containing the concatenated result of translation (`T`), rotation (`R`), and scaling (`S`) or `TRS`.

Modify Renderable Class to Set the Transform Operator

Edit `renderable.js` to modify the `draw()` function to receive and to forward a transform operator to the `mShader.activate()` function to be loaded to the GLSL shader:

```
draw(trsMatrix) {
   let gl = glSys.get();
   this.mShader.activate(this.mColor, trsMatrix);
   gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
}
```

In this way, when the vertices of the unit square are processed by the vertex shader, the `uModelXformMatrix` will contain the proper operator for transforming the vertices and thus drawing the square at the desired location, size, and rotation.

Testing the Transforms

Now that the game engine supports transformation, you need to modify the client code to draw with it:

1. Edit `my_game.js`; after step C, instead of activating and drawing the two squares, replace steps C1 and C2 to create a new identity transform operator, `trsMatrix`:

```
// create a new identify transform operator
let trsMatrix = mat4.create();
```

2. Compute the concatenation of matrices to a single transform operator that implements translation (`T`), rotation (`R`), and scaling (`S`) or `TRS`:

```
// Step D: compute the white square transform
mat4.translate(trsMatrix, trsMatrix, vec3.fromValues(-0.25, 0.25, 0.0));
mat4.rotateZ(trsMatrix, trsMatrix, 0.2);      // rotation is in radian
mat4.scale(trsMatrix, trsMatrix, vec3.fromValues(1.2, 1.2, 1.0));
// Step E: draw the white square with the computed transform
this.mWhiteSq.draw(trsMatrix);
```

Step D concatenates `T(-0.25, 0.25)`, moving to the left and up; with `R(0.2)`, rotating clockwise by 0.2 radians; and `S(1.2, 1.2)`, increasing size by 1.2 times. The concatenation order applies the scaling operator first, followed by rotation, with translation being the last operation, or `trsMatrix=TRS`. In step E, the `Renderable` object is drawn with the `trsMatrix` operator or a 1.2×1.2 white rectangle slightly rotated and located somewhat to the upper left from the center.

3. Finally, step F defines the `trsMatrix` operator to draw a 0.4×0.4 square that is rotated by 45 degrees and located slightly toward the lower right from the center of the canvas, and step G draws the red square:

```
// Step F: compute the red square transform
mat4.identity(trsMatrix); // restart
mat4.translate(trsMatrix, trsMatrix, vec3.fromValues(0.25, -0.25, 0.0));
mat4.rotateZ(trsMatrix, trsMatrix, -0.785);   // about -45-degrees
mat4.scale(trsMatrix, trsMatrix, vec3.fromValues(0.4, 0.4, 1.0));
// Step G: draw the red square with the computed transform
this.mRedSq.draw(trsMatrix);
```

Observations

Run the project, and you should see the corresponding white and red rectangles drawn on the canvas. You can gain some intuition of the operators by changing the values; for example, move and scale the squares to different locations with different sizes. You can try changing the order of concatenation by moving the corresponding line of code; for example, move `mat4.scale()` to before `mat4.translate()`. You will notice that, in general, the transformed results do not correspond to your intuition. In this book, you will always apply the transformation operators in the fixed `TRS` order. This ordering of transformation operators corresponds to typical human intuition. The `TRS` operation order is followed by most, if not all, graphical APIs and applications that support transformation operations.

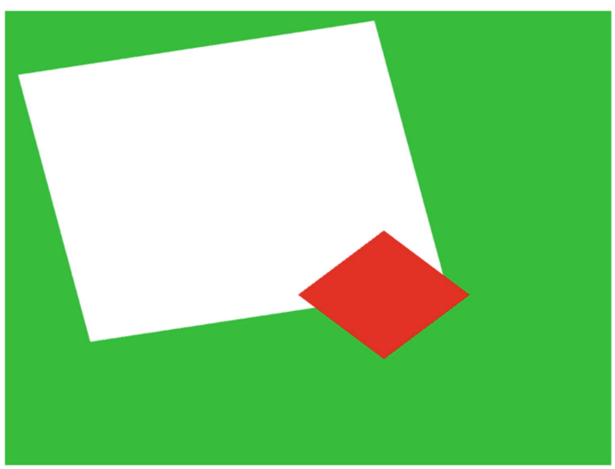

Figure 3-5. Canvas with Renderables

Now that you understand how to work with the matrix transformation operators, it is time to abstract them and hide their details.

Encapsulating the Transform Operator

In the previous project, the transformation operators were computed directly based on the matrices. While the results were important, the computation involves distracting details and repetitive code. This project guides you to follow good coding practices to encapsulate the transformation operators by hiding the detailed computations with a class. In this way, you can maintain the modularity and accessibility of the game engine by supporting further expansion while maintaining programmability.

Lab 3.3
The Transform Objects Project

This project defines the `Transform` class to provide a logical interface for manipulating and hiding the details of working with the matrix transformation operators.

The goals of the project are as follows:

- To create the `Transform` class to encapsulate the matrix transformation functionality
- To integrate the `Transform` class into the game engine
- To demonstrate how to work with `Transform` objects

The Transform Class

Continue working with the previous project:

1. Define the `Transform` class in the game engine by creating a new source code file in the `src/engine` folder and name the file `transform.js`.

2. Define the constructor to initialize instance variables that correspond to the operators: `mPosition` for translation, `mScale` for scaling, and `mRotationInRad` for rotation.

```
class Transform {
  constructor() {
    this.mPosition = vec2.fromValues(0, 0);  // translation
    this.mScale = vec2.fromValues(1, 1);     // width (x), height (y)
    this.mRotationInRad = 0.0;               // in radians!
  }
  ... implementation to follow ...
}
```

3. Add getters and setters for the values of each operator:

```
// Position getters and setters
setPosition(xPos, yPos) { this.setXPos(xPos); this.setYPos(yPos); }
getPosition() { return this.mPosition; }
setXPos(xPos) { this.mPosition[0] = xPos; }
setYPos(yPos) { this.mPosition[1] = yPos; }
```

```
getXPos() { return this.mPosition[0]; }
getYPos() { return this.mPosition[1]; }

// Size setters and getters
setSize(width, height) {
  this.setWidth(width);
  this.setHeight(height);
}

getSize() { return this.mScale; }
setWidth(width) { this.mScale[0] = width; }
setHeight(height) { this.mScale[1] = height; }
getWidth() { return this.mScale[0]; }
getHeight() { return this.mScale[1]; }

// Rotation getters and setters
setRotationInRad(rotationInRadians) {
  this.mRotationInRad = rotationInRadians;
  while (this.mRotationInRad > (2 * Math.PI)) {
   this.mRotationInRad -= (2 * Math.PI);
  }
}
setRotationInDegree(rotationInDegree) {
  this.setRotationInRad(rotationInDegree * Math.PI / 180.0);
}
getRotationInRad() {  return this.mRotationInRad; }
getRotationInDegree() { return this.mRotationInRad * 180.0 / Math.PI; }
```

4. Add incrementors for the values of each operator:

```
incXPosBy(delta) { this.mPosition[0] += delta; }
incYPosBy(delta) { this.mPosition[1] += delta; }
incSizeBy(delta) {
     this.incWidthBy(delta);
     this.incHeightBy(delta);
 }
incWidthBy(delta) { this.mScale[0] += delta; }
incHeightBy(delta) { this.mScale[1] += delta; }
incRotationByDegree(deltaDegree) {
   this.incRotationByRad(deltaDegree * Math.PI / 180.0);
}
incRotationByRad(deltaRad) {
   this.setRotationInRad(this.mRotationInRad + deltaRad);
}
```

5. Define the `getTRSMatrix()` function to compute and return the concatenated transform operator, `TRS`:

```
getTRSMatrix() {
  // Creates a blank identity matrix
  let matrix = mat4.create();
```

```
    // Step A: compute translation, for now z is always at 0.0
    mat4.translate(matrix, matrix,
                   vec3.fromValues(this.getXPos(), this.getYPos(), 0.0));
    // Step B: concatenate with rotation.
    mat4.rotateZ(matrix, matrix, this.getRotationInRad());
    // Step C: concatenate with scaling
    mat4.scale(matrix, matrix,
               vec3.fromValues(this.getWidth(), this.getHeight(), 1.0));
    return matrix;
}
```

This code is similar to steps D and F in `my_game.js` from the previous project. The concatenated operator `TRS` performs scaling first, followed by rotation, and lastly by translation.

6. Finally, remember to export the newly defined `Transform` class:

```
export default Transform;
```

The Transformable Renderable Class

By integrating the `Transform` class, a `Renderable` object can now have a position, size (scale), and orientation (rotation). This integration can be easily accomplished through the following steps:

1. Edit `renderable.js`; import from the newly define `transform.js` file:

```
import Transform from "./transform.js";
```

2. Edit `renderable.js` and add a new instance variable to reference a `Transform` object in the constructor:

```
this.mXform = new Transform();     // transform operator for the object
```

3. Define an accessor for the transform operator:

```
getXform() { return this.mXform; }
```

4. Modify the `draw()` function to pass the `trsMatrix` operator of the `mXform` object to activate the shader before drawing the unit square:

```
draw() {
  let gl = glSys.get();
  this.mShader.activate(this.mColor, this.mXform.getTRSMatrix());
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
}
```

With this simple modification, `Renderable` objects will be drawn with characteristics defined by the values of its own transformation operators.

Modify the Engine Access File to Export Transform

It is important to maintain the engine access file, index.js, up to date such that the newly defined Transform class can be accessed by the game developer:

1. Edit `index.js`; import from the newly define `transform.js` file:

```
// general utilities
import Transform from "./transform.js";
import Renderable from "./renderable.js";
```

2. Export Transform for client's access:

```
export default {
  // Util classes
  Transform, Renderable,
  // functions
  init, clearCanvas
}
```

Modify Drawing to Support Transform Object

To test the Transform and the improved `Renderable` classes, the `MyGame` constructor can be modified to set the transform operators in each of the `Renderable` objects accordingly:

```
// Step D: sets the white Renderable object's transform
this.mWhiteSq.getXform().setPosition(-0.25, 0.25);
this.mWhiteSq.getXform().setRotationInRad(0.2); // In Radians
this.mWhiteSq.getXform().setSize(1.2, 1.2);
// Step E: draws the white square (transform behavior in the object)
this.mWhiteSq.draw();
// Step F: sets the red square transform
this.mRedSq.getXform().setXPos(0.25); // alternative to setPosition
this.mRedSq.getXform().setYPos(-0.25);// setX/Y separately
this.mRedSq.getXform().setRotationInDegree(45);  // this is in Degree
this.mRedSq.getXform().setWidth(0.4); // alternative to setSize
this.mRedSq.getXform().setHeight(0.4);// set width/height separately
```

```
// Step G: draw the red square (transform in the object)
this.mRedSq.draw();
```

Run the project to observe identical output as from the previous project. You can now create and draw a `Renderable` at any location in the canvas, and the transform operator has now been properly encapsulated.

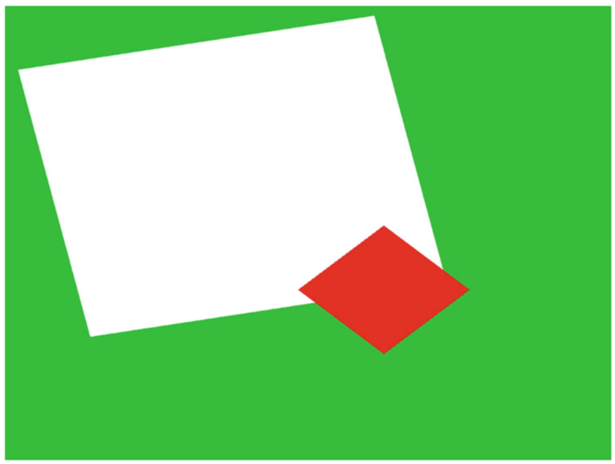

Figure 3-6. Canvas with Renderables

The Camera Transform and Viewports

When designing and building a video game, the game designers and programmers must be able to focus on the intrinsic logic and presentation. To facilitate these aspects, it is important that the designers and programmers can formulate solutions in a convenient dimension and space.

For example, continuing with the soccer game idea, consider the task of creating a soccer field. How big is the field? What is the unit of measurement? In general, when building a game world, it is often easier to design a solution by referring to the real world. In the real world, soccer fields are around 100 meters long. However, in the game or graphics world, units are arbitrary. So, a simple solution may be to create a field that is 100 units in meters and a coordinate space where the origin is located at the center of the soccer field. In this way, opposing sides of the fields can simply be determined by the sign of the x value, and drawing a player at location (0, 1) would mean drawing the player 1 meter to the right from the center of the soccer field.

A contrasting example would be when building a chess-like board game. It may be more convenient to design the solution based on a unitless n×n grid with the origin located at the lower-left corner of the board. In this scenario, drawing a piece at location (0, 1) would mean drawing the piece at the location one cell or unit toward the right from the lower-left corner of the board. As will be discussed, the ability to define specific coordinate systems is often accomplished by computing and working with a matrix representing the view from a camera.

In all cases, to support a proper presentation of the game, it is important to allow the programmer to control the drawing of the contents to any location on the canvas. For example, you may want to draw the soccer field and players to one subregion and draw a mini-map into another subregion. These axis-aligned rectangular drawing areas or subregions of the canvas are referred to as viewports.

In this section, you will learn about coordinate systems and how to use the matrix transformation as a tool to define a drawing area that conforms to the fixed ±1 drawing range of the WebGL.

Coordinate Systems and Transformations

A 2D coordinate system uniquely identifies every position on a 2D plane. All projects in this book follow the Cartesian coordinate system where positions are defined according to perpendicular distances from a reference point known as the *origin*, as illustrated below. The perpendicular directions for measuring the distances are known as the *major axes*. In 2D space, these are the familiar x and y axes.

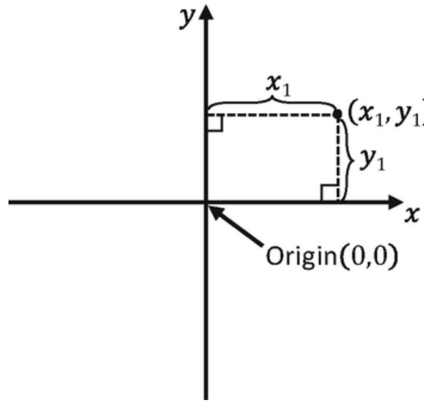

Figure 3-7. 2D Cartesian Coordinate System

Modeling and Normalized Device Coordinate Systems

So far in this book, you have experience with two distinct coordinate systems. The first is the coordinate system that defines the vertices for the 1×1 square in the vertex buffer. This is referred to as the Modeling Coordinate System, which defines the **Model Space**. The Model Space is unique for each geometric object, as in the case of the unit square. The Model Space is defined to describe the geometry of a single model. The second coordinate system that you have worked with is the one that WebGL draws to, where the x-/y-axis ranges are bounded to ±1.0. This is known as the Normalized Device Coordinate (**NDC**) System. As you have experienced, WebGL always draws to the NDC space and that the contents in the ±1.0 range cover all the pixels in the canvas.

The Modeling transform, typically defined by a matrix transformation operator, is the operation that transforms geometries from its Model Space into another coordinate space that is convenient for drawing. In the previous project, the `uModelXformMatrix` variable in `simple_vs.glsl` is the Modeling transform. As illustrated below, in that case, the Modeling transform transformed the unit square into the WebGL's NDC space. The rightmost arrow annotated with the Fixed Mapping label in the figure that points from WebGL NDC to **Canvas Coordinates** signifies that WebGL always displays the entire content of the NDC space in the canvas.

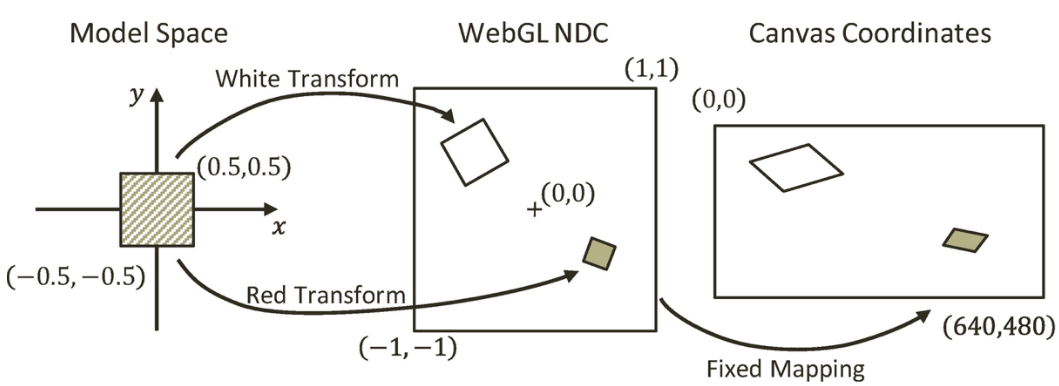

Figure 3-8. Transforming from Model to NDC Space

The World Coordinate System

Although it is possible to draw to any location with the Modeling transform, the disproportional scaling that draws squares as rectangles is still a problem. In addition, the fixed -1.0 and 1.0 NDC space is not a convenient coordinate space for designing games. The World Coordinate (WC) System describes a convenient World Space that resolves these issues. For convenience and readability, in the rest of this book, **WC** will also be used to refer to the World Space that is defined by a specific World Coordinate System.

As illustrated below, with a WC instead of the fixed NDC space, Modeling transforms can transform models into a convenient coordinate system that lends itself to game designs. For the soccer game example, the World Space dimension can be the size of the soccer field. As in any Cartesian coordinate system, the WC system is defined by a reference position and its width and height. The reference position can either be the lower-left corner or the center of the WC.

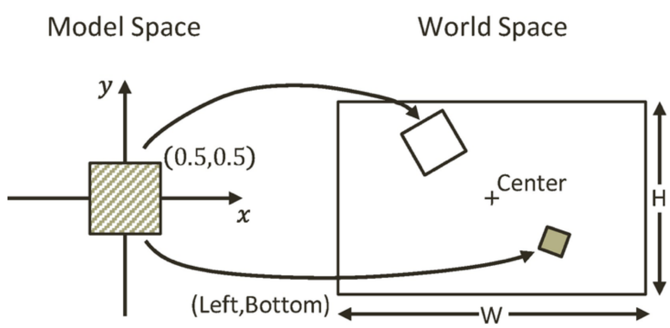

Figure 3-9. Working with World Coordinate (WC) System

The WC is a convenient coordinate system for designing games. However, it is not the space that WebGL draws to. For this reason, it is important to transform from WC to NDC. In this book, this transform is referred to as the Camera transform. To accomplish this transform, you will have to construct an operator to align WC center with that of the NDC (the origin) and then to scale the WC WxH dimension to match the NDC width and height. Note that the NDC space has a constant range of -1 to +1 and thus a fixed dimension of 2x2. In this way, the Camera transform is simply a translation followed by a scaling operation:

$$M = S\left(\frac{2}{W}, \frac{2}{H}\right) T(-center.x, -center.y)$$

In this case, `(center.x, center.y)` and `WxH` are the center and the dimension of the WC system.

The Viewport

A viewport is an area to be drawn to. As you have experienced, by default, WebGL defines the entire canvas to be the viewport for drawing. Conveniently, WebGL provides a function to override this default behavior:

```
gl.viewport(
   x,     // x position of bottom-left corner of the area to be drawn
   y,     // y position of bottom-left corner of the area to be drawn
   width, // width of the area to be drawn
   height // height of the area to be drawn
);
```

The `gl.viewport()` function defines a viewport for all subsequent drawings. The figure below illustrates the Camera transform and drawing with a viewport.

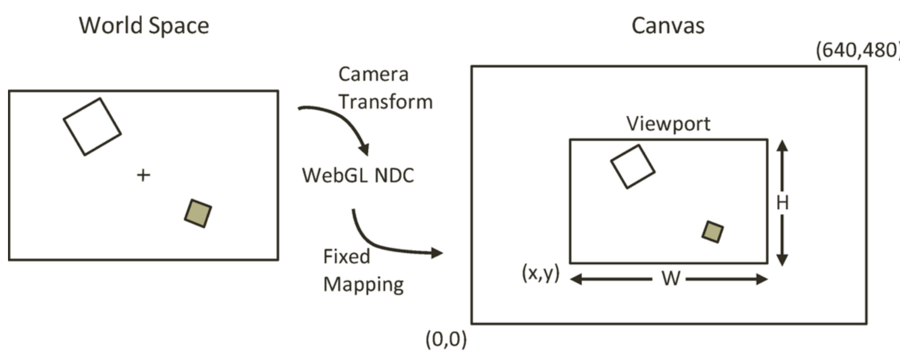

Figure 3-10. Working with WebGL Viewport

Lab 3.4
The Camera Transform and Viewport Project

This project demonstrates how to use the Camera transform to draw from any desired coordinate location to any subregion of the canvas or a viewport.

The goals of the project are as follows:

- To understand the different coordinate systems
- To experience working with a WebGL viewport to define and draw to different subregions within the canvas
- To understand the Camera transform
- To begin drawing to the user-defined World Coordinate System

You are now ready to modify the game engine to support the Camera transform to define your own WC and the corresponding viewport for drawing. The first step is to modify the shaders to support a new transform operator.

Modify the Vertex Shader to Support the Camera Transform

Relatively minor changes are required to add the support for the Camera transform:

1. Edit `simple_vs.glsl` to add a new uniform matrix operator to represent the Camera transform:

```
uniform mat4 uCameraXformMatrix;
```

2. Make sure to apply the operator on the vertex positions in the vertex shader program:

```
gl_Position = uCameraXformMatrix *
              uModelXformMatrix *
              vec4(aVertexPosition, 1.0);
```

Recall that the order of matrix operations is important. In this case, the `uModelXformMatrix` first transforms the vertex positions from Model Space to WC, and then the `uCameraXformMatrix` transforms from WC to NDC. The order of `uModelxformMatrix` and `uCameraXformMatrix` cannot be switched.

Modify SimpleShader to Support the Camera Transform

The `SimpleShader` object must be modified to access and pass the Camera transform matrix to the vertex shader:

1. Edit `simple_shader.js` and, in the constructor, add an instance variable for storing the reference to the Camera transform operator in `simple_vs.glsl`:

```
this.mCameraMatrixRef = null;
```

2. At the end of the `SimpleShader` constructor, retrieve the reference to the Camera transform operator, `uCameraXformMatrix`, after retrieving those for the `uModelXformMatrix` and `uPixelColor`:

```
// Step E: Gets reference to uniform variables in fragment shader
this.mPixelColorRef = gl.getUniformLocation(
                        this.mCompiledShader, "uPixelColor");
this.mModelMatrixRef = gl.getUniformLocation(
                        this.mCompiledShader, "uModelXformMatrix");
this.mCameraMatrixRef = gl.getUniformLocation(
                        this.mCompiledShader, "uCameraXformMatrix");
```

3. Modify the `activate` function to receive a Camera transform matrix and pass it to the shader:

```
activate(pixelColor, trsMatrix, cameraMatrix) {
   let gl = glSys.get();
   gl.useProgram(this.mCompiledShader);
   ... identical to previous code ...
   // load uniforms
   gl.uniform4fv(this.mPixelColorRef, pixelColor);
   gl.uniformMatrix4fv(this.mModelMatrixRef, false, trsMatrix);
   gl.uniformMatrix4fv(this.mCameraMatrixRef, false, cameraMatrix);
}
```

As you have seen previously, the `gl.uniformMatrix4fv()` function copies the content of `cameraMatrix` to the `uCameraXformMatrix` operator.

Modify Renderable to Support the Camera Transform

Shaders are activated in the `draw()` function of the `Renderable` class; `Renderable` must also be modified to receive and pass `cameraMatrix` to activate the shader:

```
draw(cameraMatrix) {
   let gl = glSys.get();
   this.mShader.activate(this.mColor,
                        this.mXform.getTRSMatrix(), cameraMatrix);
   gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
}
```

It is now possible to set up a WC for drawing and define a subarea in the canvas to draw to.

Design the Scene

As illustrated in figure 3-11, for testing purposes, a World Space (WC) will be defined to be centered at (20, 60) with a dimension of 20×10. Two rotated squares, a 5x5 blue square and a 2×2 red square, will be drawn at the center of the WC. To verify the coordinate bounds, a 1×1 square with a distinct color will be drawn at each of the WC corners.

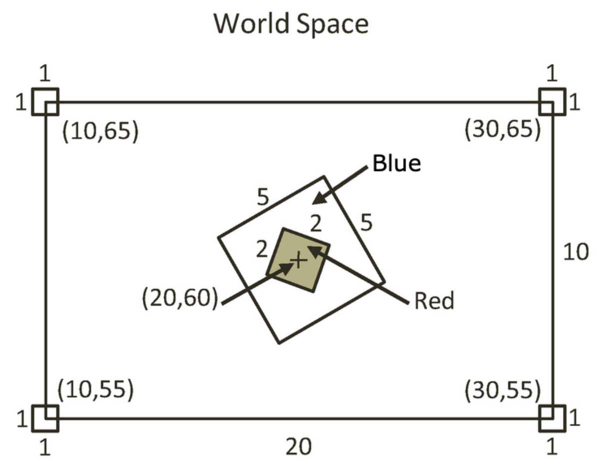

Figure 3-11. Designing a WC to support drawing

As illustrated in figure 3-12, the WC will be drawn into a viewport with the lower-left corner located at (20, 40) and a dimension of 600×300 pixels. It is important to note that in order for squares to show up proportionally, the width-to-height aspect ratio of the WC must match that of the viewport. In this case, the WC has a 20:10 aspect ratio, and this 2:1 ratio matches that of the 600:300 of the viewport.

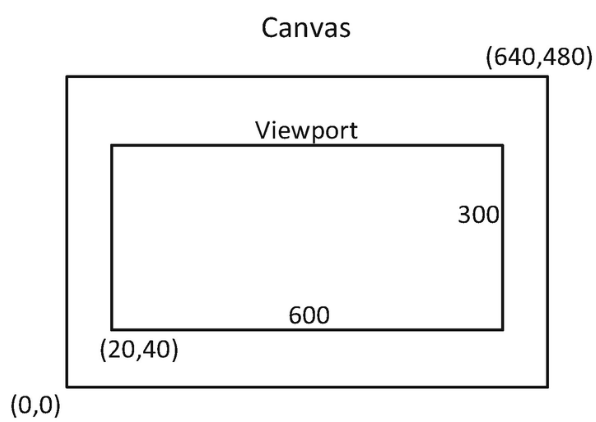

Figure 3-12. Drawing the WC to the viewport

Implement the Design

The `MyGame` class will be modified to implement the design:

1. Edit `my_game.js`; import from the `gl.js` file so that we can setup the viewport:

```
import * as glSys from "../engine/core/gl.js";
import engine from "../engine/index.js";
```

2. Edit `my_game.js`. In the constructor, we will replace the existing steps with new functionality. Add step A to initialize the game engine and step B to create six `Renderable` objects (two to be drawn at the center, with four at each corner of the WC) with corresponding colors.

```
constructor(htmlCanvasID) {
   // Step A: Initialize the game engine
   engine.init(htmlCanvasID);
   // Step B: Create the Renderable objects:
   this.mBlueSq = new engine.Renderable();
   this.mBlueSq.setColor([0.25, 0.25, 0.95, 1]);
   this.mRedSq = new engine.Renderable();
   this.mRedSq.setColor([1, 0.25, 0.25, 1]);
   this.mTLSq = new engine.Renderable();
   this.mTLSq.setColor([0.9, 0.1, 0.1, 1]);
   this.mTRSq = new engine.Renderable();
   this.mTRSq.setColor([0.1, 0.9, 0.1, 1]);
   this.mBRSq = new engine.Renderable();
   this.mBRSq.setColor([0.1, 0.1, 0.9, 1]);
   this.mBLSq = new engine.Renderable();
   this.mBLSq.setColor([0.1, 0.1, 0.1, 1]);
}
```

3. Steps C and D clear the entire canvas, set up the viewport, and clear the viewport to a different color:

```
   // Step C: Clear the entire canvas first
   engine.clearCanvas([0.9, 0.9, 0.9, 1]);
   // get access to the gl connection to the GPU
   let gl = glSys.get();
   // Step D: Setting up Viewport
   // Step D1: Set up the viewport: area on canvas to be drawn
   gl.viewport(
    20,     // x position of bottom-left corner of the area to be drawn
    40,     // y position of bottom-left corner of the area to be drawn
    600,    // width of the area to be drawn
    300);   // height of the area to be drawn
   // Step D2: set up the corresponding scissor area to limit clear area
   gl.scissor(
```

```
    20,     // x position of bottom-left corner of the area to be drawn
    40,     // y position of bottom-left corner of the area to be drawn
    600,    // width of the area to be drawn
    300);   // height of the area to be drawn
   // Step D3: enable scissor area, clear and then disable the scissor area
   gl.enable(gl.SCISSOR_TEST);
   engine.clearCanvas([0.8, 0.8, 0.8, 1.0]);  // clear the scissor area
   gl.disable(gl.SCISSOR_TEST);
```

Step D1 defines the viewport, and step D2 defines a corresponding scissor area. The scissor area tests and limits the area to be cleared. Since the testing involved in gl.scissor() is computationally expensive, it is disabled immediately after use.

4. Step E defines the WC with the Camera transform by concatenating the proper scaling and translation operators:

```
// Step E: Set up camera transform matrix
// assume camera position and dimension
let cameraCenter = vec2.fromValues(20, 60);
let wcSize = vec2.fromValues(20, 10);
let cameraMatrix = mat4.create();
// Step E1: after translation, scale to: -1 to 1: a 2x2 square at origin
mat4.scale(cameraMatrix, mat4.create(),
        vec3.fromValues(2.0/wcSize[0], 2.0/wcSize[1], 1.0));
// Step E2: first to perform is to translate camera center to origin
mat4.translate(cameraMatrix, cameraMatrix,
        vec3.fromValues(-cameraCenter[0], -cameraCenter[1], 0));
```

Step E1 defines the scaling operator, `S(2/W, 2/H)`, to scale the WC WxH to the NDC 2x2 dimension, and step E2 defines the translation operator, `T(-center.x, -center.y)`, to align the WC with the NDC center. Note that the concatenation order implements the translation first followed by the scaling operator. This is precisely the Camera transform described earlier that defines the WC as follows:

a. Center: (20,60)
b. Top-left corner: (10, 65)
c. Top-right corner: (30, 65)
d. Bottom-right corner: (30, 55)
e. Bottom-left corner: (10, 55)

Recall that the order of multiplication is important and that the order of scaling and translation operators cannot be swapped.

5. Set up the slightly rotated 5x5 blue square at the center of WC, and draw with the Camera transform operator, `cameraMatrix`:

```
// Step F: Draw the blue square
// Center Blue, slightly rotated square
this.mBlueSq.getXform().setPosition(20, 60);
this.mBlueSq.getXform().setRotationInRad(0.2); // In Radians
this.mBlueSq.getXform().setSize(5, 5);
this.mBlueSq.draw(cameraMatrix);
```

6. Now draw the other five squares, first the 2x2 in the center and one each at a corner of the WC:

```
// Step G: Draw the center and the corner squares
// center red square
this.mRedSq.getXform().setPosition(20, 60);
this.mRedSq.getXform().setSize(2, 2);
this.mRedSq.draw(cameraMatrix);
// top left
this.mTLSq.getXform().setPosition(10, 65);
this.mTLSq.draw(cameraMatrix);
// top right
this.mTRSq.getXform().setPosition(30, 65);
this.mTRSq.draw(cameraMatrix);
// bottom right
this.mBRSq.getXform().setPosition(30, 55);
this.mBRSq.draw(cameraMatrix);
// bottom left
this.mBLSq.getXform().setPosition(10, 55);
this.mBLSq.draw(cameraMatrix);
```

Run this project and observe the distinct colors at the four corners: the top left (`mTLSq`) in red, the top right (`mTRSq`) in green, the bottom right (`mBRSq`) in blue, and the bottom left (`mBLSq`) in dark gray. Change the locations of the corner squares to verify that the center positions of these squares are located in the bounds of the WC, and thus, only one quarter of the squares are actually visible. For example, set `mBlSq` to (12, 57) to observe the dark-gray square is actually four times the size. This observation verifies that the areas of the squares outside of the viewport/scissor area are clipped by WebGL.

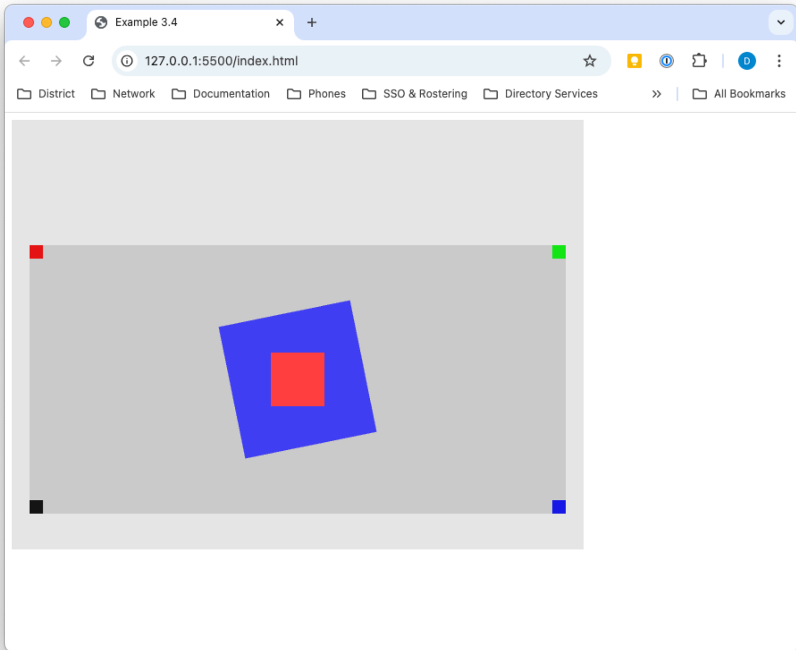

Figure 3-13. Camera Transform and Viewport

Although lacking proper abstraction, it is now possible to define any convenient WC system and any rectangular subregions of the canvas for drawing. With the Modeling and Camera transformations, a game programmer can now design a game solution based on the semantic needs of the game and ignore the irrelevant WebGL NDC drawing range. However, the code in the `MyGame` class is complicated and can be distracting. As you have seen so far, the important next step is to define an abstraction to hide the details of Camera transform matrix computation.

Lab 3.5
The Camera Objects Project

This project demonstrates how to abstract the Camera transform and the viewport to hide the details of matrix computation and WebGL configurations.

The goals of the project are as follows:

- To define the Camera class to encapsulate the definition of WC and the viewport functionality
- To integrate the Camera class into the game engine
- To demonstrate how to work with a Camera object

The Camera Class

The `Camera` class must encapsulate the functionality defined by the scaling and translation operators in the `MyGame` constructor from the previous example. A clean and reusable class design should be completed with appropriate getter and setter functions.

1. Define the `Camera` class in the game engine by creating a new source file in the `src/engine` folder, and name the file `camera.js`.

2. Add import for `gl.js`:

```
import * as glSys from "./core/gl.js";
```

3. Add the constructor for Camera:

```
class Camera {
   constructor(wcCenter, wcWidth, viewportArray) {
    // WC and viewport position and size
    this.mWCCenter = wcCenter;
    this.mWCWidth = wcWidth;
    this.mViewport = viewportArray;  // [x, y, width, height]
    // Camera transform operator
    this.mCameraMatrix = mat4.create();
    // background color
    this.mBGColor = [0.8, 0.8, 0.8, 1]; // RGB and Alpha
   }
  ... implementation to follow ...
}
```

The `Camera` defines the WC center and width, the viewport, the Camera transform operator, and a background color. Take note of the following:

a. The `mWCCenter` is a `vec2` (`vec2` is defined in the `glMatrix` library). It is a float array of two elements. The first element, index position 0, of `vec2` is the x, and the second element, index position 1, is the y position.
b. The four elements of the `viewportArray` are the x and y positions of the lower-left corner and the width and height of the viewport, in that order. This compact representation of the viewport keeps the number of instance variables to a minimum and helps keep the `Camera` class manageable.
c. The `mWCWidth` is the width of the WC. To guarantee a matching aspect ratio between WC and the viewport, the height of the WC is always computed from the aspect ratio of the viewport and `mWCWidth`.
d. `mBgColor` is an array of four floats representing the red, green, blue, and alpha components of a color.

4. Outside of the `Camera` class definition, define enumerated indices for accessing the `viewportArray`:

```
const eViewport = Object.freeze({
  eOrgX: 0,
  eOrgY: 1,
  eWidth: 2,
  eHeight: 3
});
```

Note Enumerated elements have names that begin with lowercase "e", as in `eViewport` and `eOrgX`.

5. Define the function to compute the WC height based on the aspect ratio of the viewport:

```
getWCHeight() {
  // viewportH/viewportW
  let ratio = this.mViewport[eViewport.eHeight] /
              this.mViewport[eViewport.eWidth];
  return this.getWCWidth() * ratio;
}
```

6. Add getters and setters for the instance variables:

```
setWCCenter(xPos, yPos) {
  this.mWCCenter[0] = xPos;
  this.mWCCenter[1] = yPos;
}
getWCCenter() { return this.mWCCenter; }
setWCWidth(width) { this.mWCWidth = width; }
getWCWidth() { return this.mWCWidth; }
setViewport(viewportArray) { this.mViewport = viewportArray; }
getViewport() { return this.mViewport; }
setBackgroundColor(newColor) { this.mBGColor = newColor; }
getBackgroundColor() { return this.mBGColor; }
```

7. Create a function to set the viewport and compute the Camera transform operator for this Camera:

```
// Initializes the camera to begin drawing
setViewAndCameraMatrix() {
  let gl = glSys.get();
  // Step A: Configure the viewport
  ... implementation to follow ...
  // Step B: compute the Camera Matrix
  ... implementation to follow ...
}
```

Note that this function is called `setViewAndCameraMatrix()` because it configures WebGL to draw to the desire viewport and sets up the Camera transform operator. The following explains the details of steps A and B.

8. The code to configure the viewport under step A is as follows:

```
// Step A1: Set up the viewport: area on canvas to be drawn
gl.viewport(this.mViewport[0],  // x of bottom-left of area to be drawn
  this.mViewport[1],  // y of bottom-left of area to be drawn
  this.mViewport[2],  // width of the area to be drawn
  this.mViewport[3]); // height of the area to be drawn
// Step A2: set up the corresponding scissor area to limit the clear area
gl.scissor(this.mViewport[0], // x of bottom-left of area to be drawn
  this.mViewport[1], // y of bottom-left of area to be drawn
  this.mViewport[2], // width of the area to be drawn
  this.mViewport[3]);// height of the area to be drawn
// Step A3: set the color to be clear
gl.clearColor(this.mBGColor[0], this.mBGColor[1],
            this.mBGColor[2], this.mBGColor[3]);
// set the color to be cleared
// Step A4: enable scissor area, clear and then disable the scissor area
gl.enable(gl.SCISSOR_TEST);
gl.clear(gl.COLOR_BUFFER_BIT);
gl.disable(gl.SCISSOR_TEST);
```

Notice the similarity of these steps to the viewport setup code in `MyGame` of the previous example. The only difference is the proper references to the instance variables via `this`.

9. The code to set up the `Camera` transform operator under step B is as follows:

```
// Step B: Compute the Camera Matrix
let center = this.getWCCenter();
// Step B1: after translation, scale to -1 to 1: 2x2 square at origin
mat4.scale(this.mCameraMatrix, mat4.create(),
         vec3.fromValues(2.0 / this.getWCWidth(),
                         2.0 / this.getWCHeight(), 1.0));
// Step B2: first translate camera center to the origin
mat4.translate(this.mCameraMatrix, this.mCameraMatrix,
             vec3.fromValues(-center[0], -center[1], 0));
```

Once again, this code is similar to the `MyGame` constructor from the previous example.

10. Define a function to access the computed camera matrix:

```
getCameraMatrix() { return this.mCameraMatrix; }
```

11. Finally, remember to export the newly defined `Camera` class:

```
export default Camera;
```

Modify Renderable to Support the Camera Class

The `draw()` function of the `Renderable` class must be modified to receive the newly defined `Camera` in order to access the computed camera matrix:

```
draw(camera) {
   let gl = glSys.get();
   this.mShader.activate(this.mColor, this.mXform.getTRSMatrix(),
                         camera.getCameraMatrix());
   gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
}
```

Modify the Engine Access File to Export Camera

It is important to maintain the engine access file, `index.js`, up to date such that the newly defined `Camera` class can be accessed by the game developer:

1. Edit `index.js`; import from the newly defined `camera.js` file:

```
// general utilities
import Camera from "./camera.js";
import Transform from "./transform.js";
import Renderable from "./renderable.js";
```

2. Export `Camera` for client's access:

```
export default {
   // Util classes
   Camera, Transform, Renderable,
   // functions
   init, clearCanvas
}
```

Test the Camera

With the Camera class properly defined, testing it from `my_game.js` is straightforward:

1. Edit `my_game.js`; you can remove the import for `gl.js` as it is no longer needed since `index.js` has imports for all that will be accessed.

2. Edit `my_game.js`; after the initialization of the game engine in step A, create an instance of the `Camera` object with settings that define the WC and viewport from the previous project in step B:

```
class MyGame {
  constructor(htmlCanvasID) {
     // Step A: Initialize the game engine
     engine.init(htmlCanvasID);
     // Step B: Setup the camera
     this.mCamera = new engine.Camera(
         vec2.fromValues(20, 60),    // center of the WC
         20,                         // width of WC
         [20, 40, 600, 300]          // viewport:orgX, orgY, W, H
         );
}
```

3. Continue with the creation of the six `Renderable` objects and the clearing of the canvas in steps C and D:

```
// Step C: Create the Renderable objects:
this.mBlueSq = new engine.Renderable();
this.mBlueSq.setColor([0.25, 0.25, 0.95, 1]);
this.mRedSq = new engine.Renderable();
this.mRedSq.setColor([1, 0.25, 0.25, 1]);
this.mTLSq = new engine.Renderable();
this.mTLSq.setColor([0.9, 0.1, 0.1, 1]);
this.mTRSq = new engine.Renderable();
this.mTRSq.setColor([0.1, 0.9, 0.1, 1]);
this.mBRSq = new engine.Renderable();
this.mBRSq.setColor([0.1, 0.1, 0.9, 1]);
this.mBLSq = new engine.Renderable();
this.mBLSq.setColor([0.1, 0.1, 0.1, 1]);
// Step D: Clear the canvas
engine.clearCanvas([0.9, 0.9, 0.9, 1]);        // Clear the canvas
```

4. Now, call the `setViewAndCameraMatrix()` function of the `Camera` object in to configure the WebGL viewport and compute the camera matrix in step E, and draw all the `Renderable`s using the `Camera` object in steps F and G.

```
// Step E: Starts the drawing by activating the camera
this.mCamera.setViewAndCameraMatrix();
// Step F: Draw the blue square
// Center Blue, slightly rotated square
this.mBlueSq.getXform().setPosition(20, 60);
this.mBlueSq.getXform().setRotationInRad(0.2); // In Radians
this.mBlueSq.getXform().setSize(5, 5);
this.mBlueSq.draw(this.mCamera);
// Step G: Draw the center and the corner squares
```

```
// center red square
this.mRedSq.getXform().setPosition(20, 60);
this.mRedSq.getXform().setSize(2, 2);
this.mRedSq.draw(this.mCamera);

// top left
this.mTLSq.getXform().setPosition(10, 65);
this.mTLSq.draw(this.mCamera);

// top right
this.mTRSq.getXform().setPosition(30, 65);
this.mTRSq.draw(this.mCamera);

// bottom right
this.mBRSq.getXform().setPosition(30, 55);
this.mBRSq.draw(this.mCamera);

// bottom left
this.mBLSq.getXform().setPosition(10, 55);
this.mBLSq.draw(this.mCamera);
```

The `mCamera` object is passed to the `draw()` function of the `Renderable` objects such that the Camera transform matrix operator can be retrieved and used to activate the shader.

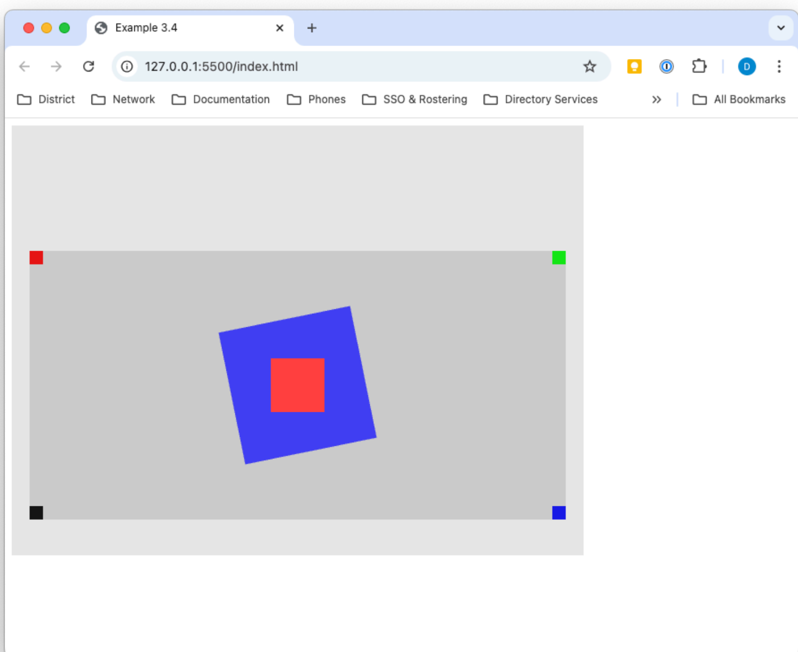

Figure 3-14. Camera Transform and Viewport

Summary

In this lesson, you learned how to create a system that can support the drawing of many objects. The system is composed of three parts: the objects, the details of each object, and the display of the objects on the browser's canvas. The objects are encapsulated by the `Renderable`, which uses a `Transform` to capture its details—the position, size, and rotation. The particulars of displaying the objects are defined by the `Camera`, where objects at specific locations can be displayed at desirable subregions on the canvas.

You also learned that objects are all drawn relative to a World Space or WC, a convenient coordinate system. A WC is defined for scene compositions based on coordinate transformations. Lastly, the Camera transform is used to select which portion of the WC to actually display on the canvas within a browser. This can be achieved by defining an area that is viewable by the `Camera` and using the viewport functionality provided by WebGL.

As you built the drawing system, the game engine source code structure has been consistently refactored into abstracted and encapsulated components. In this way, the source code structure continues to support further expansion including additional functionality which will be discussed in the next chapter.