CHAPTER 3

Drawing Objects in the World

After completing this chapter, you will be able to:

- Create and draw multiple rectangular objects
- Control the position, size, rotation, and color of the created rectangular objects
- Define a coordinate system to draw from
- Define a target subarea on the canvas to draw to
- Work with abstract representations of renderable objects, transformation operators, and cameras

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.

While this book is about building abstractions for a game engine, this chapter focuses on creating the fundamental abstractions to support drawing. Based on the soccer game example, drawing support for 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 upon which they play. Additionally, to support proper presentation, the game engine must allow drawing to specific subregions on the canvas such 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, builds JavaScript objects 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 functionality of a game engine, the details of how drawings are accomplished can actually be distractions to the gameplay programming. For example, it is important to create, control the locations of, and draw soccer players in a soccer game. However, leaving the details of how each player is actually defined (by a collection of vertices that form triangles) exposed can quickly overwhelm and complicate the process of the game's development. 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.

The Renderable Objects Project

This project introduces the `Renderable` object to encapsulate the drawing operation. Over the next few projects you will learn more supporting concepts to refine the implementation of the `Renderable` object such that multiple instances of this object can be created and manipulated. Figure 3-1 shows the output of running the Renderable Objects project. The source code to this project is defined in the `Chapter3/3.1.RenderableObjects` folder.

Figure 3-1. *Running the Renderable Objects project*

The goals of the project are as follows:

- To begin the process of building an object to encapsulate the drawing operations by first abstracting the drawing functionality
- To demonstrate how to create different instances of `SimpleShader`
- To demonstrate the ability to create multiple `Renderable` objects

The Renderable Object

This project introduces the `Renderable` object to encapsulate the drawing process and reorganizes the source code folders to maintain the structure as the number of source code files increases. You will continue to see this pattern of learning new concepts followed by defining abstractions to hide the details of the concepts to support programmability and extensibility.

1. Define the `Renderable` object in the game engine by creating a new source code file in the `src/Engine` folder and name the file `Renderable.js`. Remember to load this new source file in `index.html`.

2. Open `Renderable.js` and create a constructor that receives a `SimpleShader` as a parameter with a color instance variable.

```
function Renderable(shader) {
    this.mShader = shader; // the shader for shading this object
    this.mColor = [1, 1, 1, 1];  // Color for fragment shader
}
```

3. Define a `draw` function for `Renderable`.

```
Renderable.prototype.draw = function() {
    var gl = gEngine.Core.getGL();
    this.mShader.activateShader(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 `activateShader()` function before sending the vertices with the `gl.drawArrays()` function.

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

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

5. Finally, as the number of source files in the `src/Engine` continues to increase, it is important to constantly keep this folder organized. In this case, create a folder under `src/Engine` and name it `Core`. Move `Engine_Core.js` and `Engine_VertexBuffer.js` into this new folder. The `src/Engine/Core` folder will store all components that belong to the core of the game engine.

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`, a white instance and a red instance of the object are created and drawn as follows:

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

    // Step B: Create the shader
    this.mConstColorShader = new SimpleShader(
            "src/GLSLShaders/SimpleVS.glsl",   // Path to the VertexShader
            "src/GLSLShaders/SimpleFS.glsl");  // Path to the FragmentShader

    // Step C: Create the Renderable objects:
    this.mWhiteSq = new Renderable(this.mConstColorShader);
    this.mWhiteSq.setColor([1, 1, 1, 1]);
    this.mRedSq = new Renderable(this.mConstColorShader);
    this.mRedSq.setColor([1, 0, 0, 1]);

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

    // Step D1: Draw Renderable objects with the white shader
    this.mWhiteSq.draw();

    // Step D2: Draw Renderable objects with the red shader
    this.mRedSq.draw();
};
```

In the code from `src/MyGame.js`, the `MyGame` constructor is modified to include the following steps:

1. Step A initializes the `gEngine.Core`.
2. Step B creates the `mConstColorShader` based on the `SimpleVS.glsl` and `SimpleFS.glsl`.
3. Step C creates two instances of `Renderable` using the shader and sets the colors of the new `Renderable` objects accordingly.
4. Step D clears the canvas; steps D1 and D2 simply call the respective `draw` functions of the white and red squares. Although both of the squares are drawn, for now you will only be able to see the last drawn square in the canvas. Please refer to the following discussion for the details.

Observations

Run the project and you will notice 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.

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

Transforming a Renderable Object

A mechanism is required to manipulate the position, size, and orientation of a `Renderable` object. Over the next few projects you will learn about how matrix transformations can be used to translate or move an object's position, scale the size of an object, and change the orientation or rotate an object on the canvas. These operations are the most intuitive ones for object manipulations. However, before the implementation of transformation matrices, a quick review of the operations and capabilities of matrices is required.

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 study of these subjects. Instead, the focus is on the application of a small collection of relevant concepts and operators from the perspective of what the game engine requires (or, rather, how to utilize the operators and not study the theories behind the mathematics). 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 in depth by delving into relevant books on linear algebra and computer graphics.

A matrix itself is an *m*-rows by *n*-columns 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.

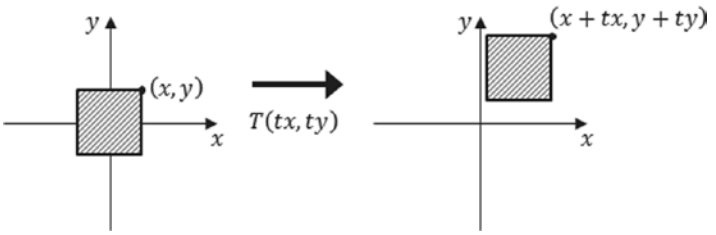

Figure 3-2. *Translating a square by T(tx,ty)*

- 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.

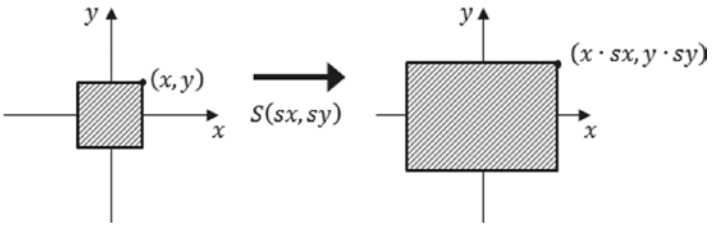

Figure 3-3. *Scaling a square by S(sx,sy)*

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

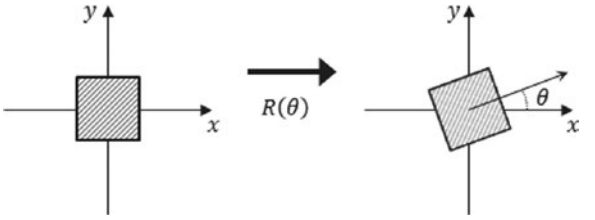

Figure 3-4. *Rotating a square by R(θ)*

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 re-apply the results of multiple operators.

Finally, notice that when working with transformation operators, the order of precedence 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`, as shown in Figure 3-5, and download, unzip, and store the resulting `glMatrix.js` source file into the new `lib` folder.

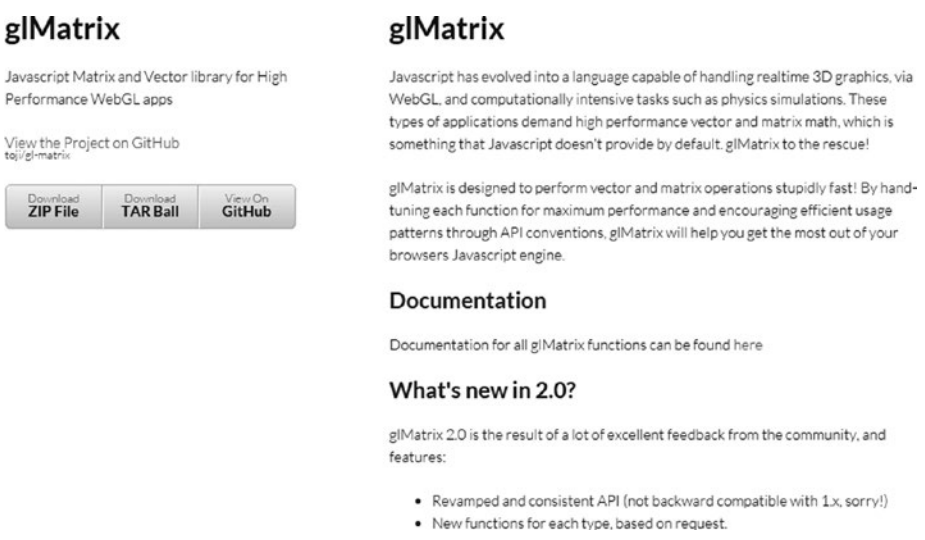

Figure 3-5. *Downloading the glMatrix library*

All projects in this book are based on `glMatrix` version 2.2.2.

3. Just like any of your own JavaScript source files, remember to load this new source file in `index.html` by adding the following:

```
<script type="text/javascript" src="src/lib/gl-matrix.js"></script>
```

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. Figure 3-6 shows the output of running the Matrix Transform project. The source code to this project is defined in the `Chapter3/3.2.MatrixTransform` folder.

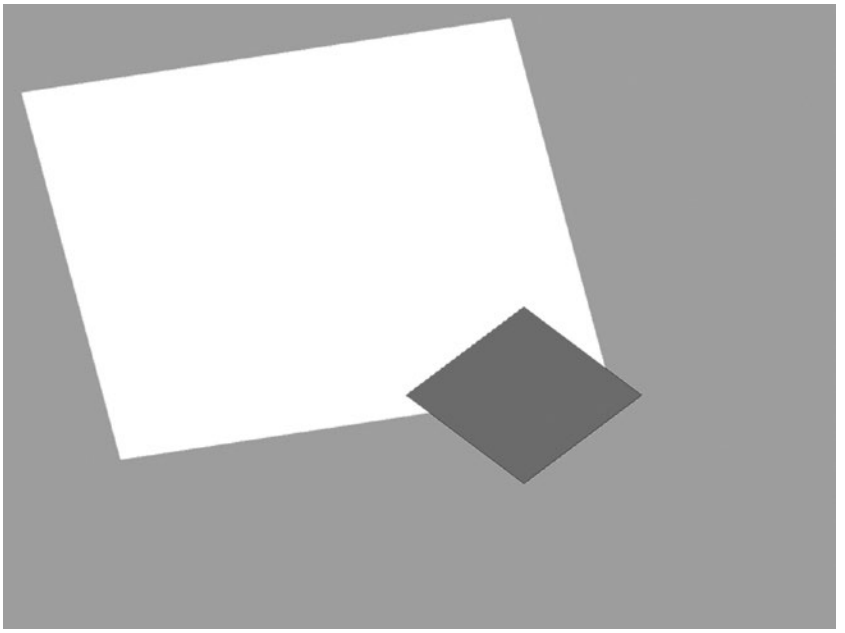

Figure 3-6. *Running the Matrix Transform project*

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 `SimpleVS.glsl` to declare a uniform 4×4 matrix.

```
// to transform the vertex position
uniform mat4 uModelTransform;
```

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 `uModelTransform` variable is the transform operator, and it maintains the operator values for all vertices of the square.

■ **Note** GLSL uniform variable names always begin with a *u*.

2. In the `main()` function, apply the `uModelTransform` to each vertex position in the shader.

```
gl_Position = uModelTransform * vec4(aSquareVertexPosition, 1.0);
```

Notice that the operation follows directly from the discussion on matrix transformation operators. The reason for converting `aSquareVertexPosition` 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 `uModelTransform` 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 `uModelTransform`.

Modify SimpleShader to Load the Transform Operator

Follow these steps:

1. Edit `SimpleShader.js` and add an instance variable to hold the reference to the `uModelTransform` matrix in the vertex shader.

```
this.mModelTransform = null;
```

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

```
// Step G: Gets a reference to the uniform variables:
//     uPixelColor and uModelTransform
this.mPixelColor = gl.getUniformLocation(this.mCompiledShader, "uPixelColor");
this.mModelTransform = gl.getUniformLocation(this.mCompiledShader,
    "uModelTransform");
```

3. Add a new function to `SimpleShader` to load the transform operator to the vertex shader.

```
// Loads per-object model transform to the vertex shader
SimpleShader.prototype.loadObjectTransform = function(modelTransform) {
    var gl = gEngine.Core.getGL();
    gl.uniformMatrix4fv(this.mModelTransform, false, modelTransform);
};
```

The `gl.uniformMatrix4fv()` function copies `modelTransform` to the vertex shader location identified by `mModelTransform` or the `uModelTransform` operator in the vertex shader.

Modify Renderable Object to Set the Transform Operator

Edit `Renderable.js` to modify the `draw()` function to receive a transform operator as a parameter, and after activating the proper GLSL shader with the `mShader.activateShader()` function, send this operator into the shader before the actual drawing operation.

```
Renderable.prototype.draw = function(modelTransform) {
    var gl = gEngine.Core.getGL();     // always activate the shader first!
    this.mShader.activateShader(this.mColor);
    this.mShader.loadObjectTransform(modelTransform);
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
};
```

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

Testing the Transforms

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

1. Edit `MyGame.js`; after step D, instead of activating and drawing the two squares, replace steps D1 and D2 to create a new identity transform operator.

   ```
   // create a new identify transform operator
   var xform = mat4.create();
   ```

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

   ```
   // Step E: compute the white square transform
   mat4.translate(xform, xform, vec3.fromValues(-0.25, 0.25, 0.0));
   mat4.rotateZ(xform, xform, 0.2);      // rotation is in radian
   mat4.scale(xform, xform, vec3.fromValues(1.2, 1.2, 1.0));

   // Step F: draw the white square with the computed transform
   this.mWhiteSq.draw(xform);
   ```

Step E 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 `xform=TRS`. In step F after the shader is activated, the `Renderable` object is drawn with the `xform` operator or a 1.2×1.2 white rectangle slightly rotated and located somewhat to the upper left from the center.

3. Finally, steps G and H are used to define and draw the red square.

   ```
   // Step G: compute the red square transform
   mat4.identity(xform); // restart
   mat4.translate(xform, xform, vec3.fromValues(0.25, -0.25, 0.0));
   mat4.rotateZ(xform, xform, -0.785);   // rotation of about -45-degrees
   mat4.scale(xform, xform, vec3.fromValues(0.4, 0.4, 1.0));

   // Step H: draw the red square with the computed transform
   this.mRedSq.draw(xform);
   ```

Step G defines the `xform` operator that draws 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.

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.

Now that you understand how to utilize 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 an object. In this way, you can maintain the modularity and accessibility of the game engine by supporting further expansion while maintaining programmability.

The Transform Objects Project

This project defines the `Transform` object to provide a logical interface for manipulating transformation operators and to hide the details of computing matrix transformation operators. Figure 3-7 shows the output of running the Matrix Transform project; notice that the output of this project is identical to that from the previous project. The source code to this project is defined in the `Chapter3/3.3.TransformObjects` folder.

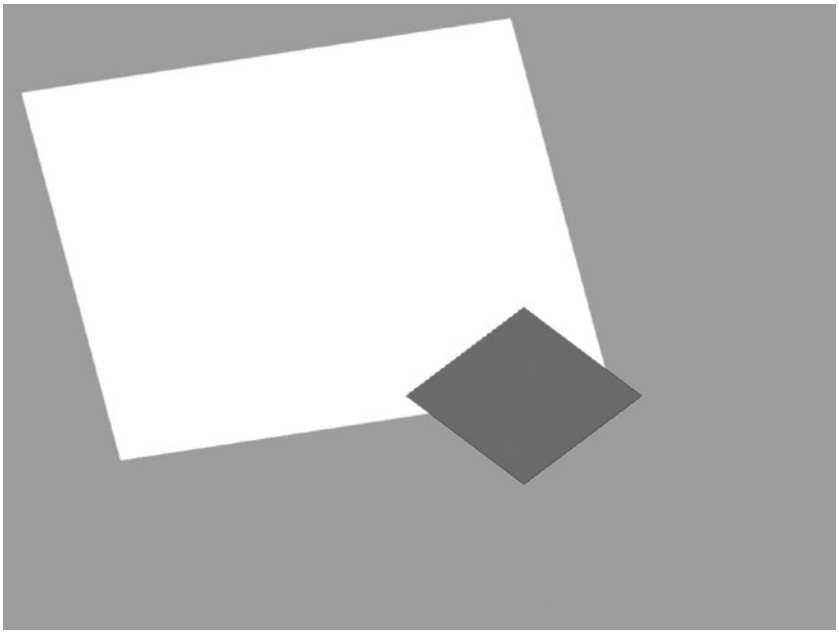

Figure 3-7. *Running the Transform Objects project*

The goals of the project are as follows:

- To create the `Transform` object so it can encapsulate the matrix transformation functionality
- To integrate the `Transform` object into the game engine
- To demonstrate how to work with the `Transform` object

The Transform Object

Continue working with the previous project.

1. Define the `Transform` object in the game engine by creating a new source code file in the `src/Engine` folder, and name the file `Transform.js`. Remember to load the new source file in `index.html`.
2. Add the constructor for `Transform`.

```
function Transform() {
    this.mPosition = vec2.fromValues(0, 0); // translation operator
    this.mScale = vec2.fromValues(1,1);     // Scaling operator
    this.mRotationInRad = 0.0;              // Rotation in radians!
};
```

These instance variables store the values for the corresponding operators: `mPosition` for translation, `mScale` for scaling, and `mRotationInRad` for rotation.

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

```
// Position getters and setters
Transform.prototype.setPosition = function(xPos,yPos) { this.setXPos(xPos);
    this.setYPos(yPos); };
Transform.prototype.getPosition = function() { return this.mPosition;};
// ... additional get and set functions for position not shown
// Size setters and getters
Transform.prototype.setSize = function(width, height) { this.setWidth(width);
    this.setHeight(height); };
Transform.prototype.getSize = function(){ return this.mScale; };
// ... additional get and set functions for size not shown
// Rotation getters and setters
Transform.prototype.setRotationInRad = function(rotationInRadians) {
    this.mRotationInRad = rotationInRadians;
    while (this.mRotationInRad > (2*Math.PI))
        this.mRotationInRad -= (2*Math.PI);
};
Transform.prototype.setRotationInDegree = function (rotationInDegree)
    { this.setRotationInRad(rotationInDegree * Math.PI/180.0); };
// ... additional get and set functions for rotation not shown
```

4. Add a function to compute and return the concatenated transform operator, TRS.

```
Transform.prototype.getXform = function() {
    // Creates a blank identity matrix
    var matrix = mat4.create();

    // Step 1: compute translation, for now z is always at 0.0
    mat4.translate(matrix, matrix, vec3.fromValues(this.getXPos(),
    this.getYPos(), 0.0));
    // Step 2: concatenate with rotation.
    mat4.rotateZ(matrix, matrix, this.getRotationInRad());
    // Step 3: concatenate with scaling
    mat4.scale(matrix, matrix,  vec3.fromValues(this.getWidth(),
    this.getHeight(), 1.0));
    return matrix;
};
```

This code is similar to steps E and G in MyGame.js from the previous project. The concatenated operator TRS performs scaling first, followed by rotation, and last by translation.

Transformable Renderable Objects

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

1. Edit Renderable.js and add a Transform instance variable.

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

2. Define an accessor for the transform operator.

```
Renderable.prototype.getXform = function() { return this.mXform; }
```

3. Modify the draw() function to load the mXform operator to the vertex shader before sending the vertex positions of the unit square.

```
Renderable.prototype.draw = function() {
    var gl = gEngine.Core.getGL();
    this.mShader.activateShader(this.mColor);
        // always activate the shader first!
    this.mShader.loadObjectTransform(this.mXform.getXform());
    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 Drawing to Support Transform Object

To test the Transform object and the modified Renderable object, the MyGame constructor can be modified to set the transform operators in each of the Renderable objects accordingly.

```
// Step E: 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 F: draws the white square (transform behavior in the object)
this.mWhiteSq.draw();

// Step G: sets the red square transform
this.mRedSq.getXform().setXPos(0.25);    // to show alternative to setPosition
this.mRedSq.getXform().setYPos(-0.25);   // it is possible to setX/Y separately
this.mRedSq.getXform().setRotationInDegree(45);  // this is in Degree
this.mRedSq.getXform().setWidth(0.4);     // to show alternative to setSize
this.mRedSq.getXform().setHeight(0.4);    // that it is possible to width/height separately
// Step H: draw the red square (transform in the object)
this.mRedSq.draw();
```

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

View, Projection, and Viewports

When designing and building a video game, the game programmers must be able to focus on the intrinsic logic and presentation of the games themselves. To facilitate this, it is important that the programmer 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 itself. 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. 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 unit-less 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 the corresponding View and Projection matrices.

In all cases, to support 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 1:1 drawing range of the WebGL in order to draw geometries proportionally.

Coordinate Systems and Transformations

A 2D coordinate system uniquely identifies every position on a 2D plane. For example, as illustrated in Figure 3-8, 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*. The perpendicular directions for measuring the distances are known as the *major axes*. In 2D space, these are the familiar x- and y-axes.

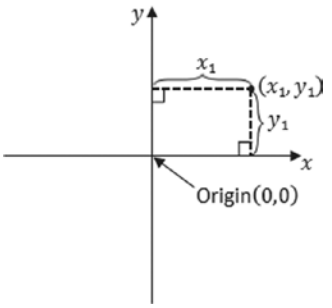

Figure 3-8. *Working with a 2D Cartesian coordinate system*

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. Second, the coordinate space that the WebGL draws to, the one where the x/y-axes ranges are bounded to ±1.0, is defined by the Normalized Device Coordinates (NDC) System. As you have experienced, WebGL always draws to NDC space and shows the results 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 `uModelTransform` variable in `SimpleVS.glsl` is the Modeling transform. As illustrated in Figure 3-9, in that case, the Modeling transform transformed the unit square into the WebGL's NDC space.

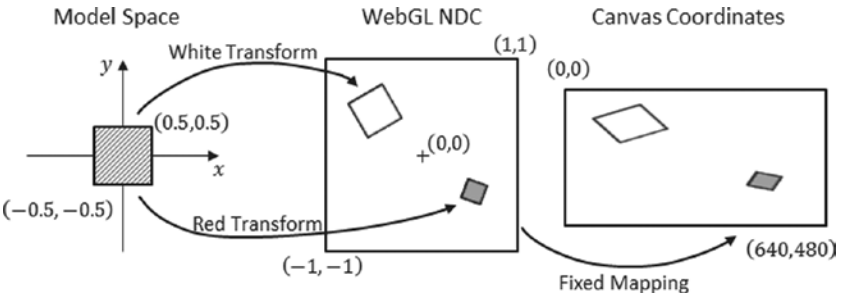

Figure 3-9. *Transforming the square from Model to NDC space*

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 describing a convenient World Space was introduced to remedy 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 in Figure 3-10, 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 dimensions or width and height. The reference position can be either the lower-left corner or the center of the WC.

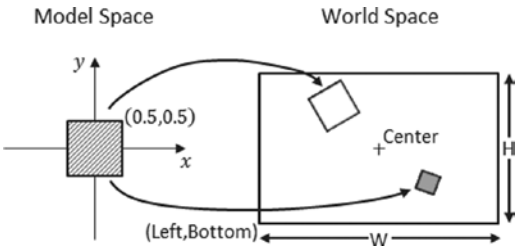

Figure 3-10. *Working with a World Coordinate (WC) System*

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 WC to NDC. This transform is referred to as the View-Projection transform. To accomplish this transform, you will take advantage of two important functions provided by the `glMatrix` library.

```
mat4.lookAt(viewMatrix,
          [cx, cy, 10],   // (cx,cy) is center of the WC
          [cx, cy, 0],
          [0, 1, 0]);     // orientation

mat4.ortho(projMatrix,
        -distToLeft,      // distant from (cx,cy) to left of WC
         distToRight,     // distant from (cx,cy) to right of WC
        -distToBottom,    // distant from (cx,cy) to bottom of WC
         distToTop,       // distant from (cx,cy) to top of WC
         0,               // the z-distant to near plane
         1000             // the z-distant to far plane
    );
```

As shown, the `mat4.lookAt()` function defines the center, and the `mat4.ortho()` function defines the dimension of the WC. In both cases, results are returned as matrix operators: the View matrix (`viewMatrix`) and the Projection matrix (`projMatrix`). Since this book focuses on the 2D space, for now you do not need to be concerned with the z-component of the parameters. Notice that the distances in the `mat4.ortho()` function are signed quantities; in other words, the distance to the right/up is positive, while the distance to the left/bottom is negative. For example, 5 units to the left is -5.

The View-Projection transform operator, `vpMatrix`, is simply the concatenation of the View and Projection matrices, the results from the `mat4.lookAt()` and `mat4.ortho()` functions.

$$\text{vpMatrix} = \text{projMatrix} \times \text{viewMatrix}$$

■ **Note** Interested readers can consult a 3D computer graphics reference book to learn more about the View and Projection transforms.

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. Fortunately, 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. Figure 3-11 illustrates the View-Projection transform and drawing with a viewport.

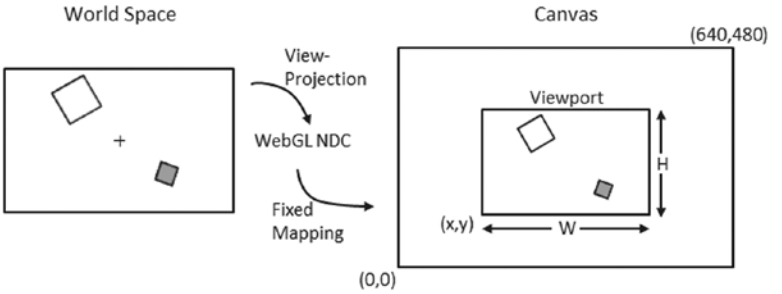

Figure 3-11. *Working with the WebGL viewport*

The View Projection and Viewport Project

This project demonstrates how to use a View-Projection transform to draw from any desired coordinate location to any subregion of the canvas, or a viewport. Figure 3-12 shows the output of running the View Projection and Viewport project. The source code to this project is defined in the `Chapter3/3.4.ViewProjectionAndViewport` folder.

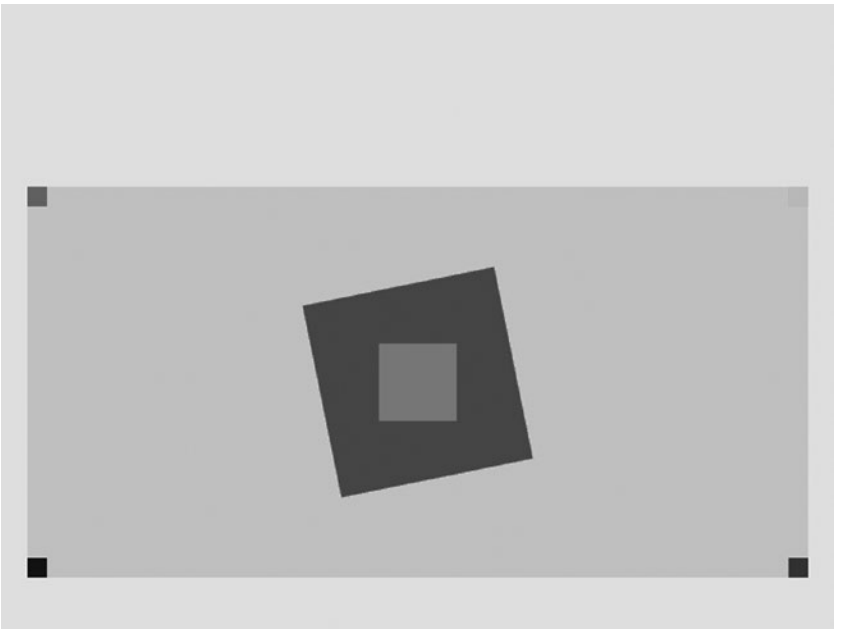

Figure 3-12. *Running the View Projection and Viewport project*

The goals of the project are as follows:

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

You are now ready to modify the game engine to support the View-Projection 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.

Modifying the Vertex Shader to Support the View-Projection Transform

Relatively minor changes are required to add the support for the View-Projection transform.

1. Edit `SimpleVS.glsl` to add a new `uniform` matrix operator to represent the View-Projection transform.

   ```
   uniform mat4 uViewProjTransform;
   ```

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

   ```
   gl_Position = uViewProjTransform * uModelTransform *
       vec4(aSquareVertexPosition, 1.0);
   ```

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

Modifying SimpleVertex to Support the View-Projection Transform

The SimpleShader object must be modified to pass the View-Projection matrix to the vertex shader.

1. Edit SimpleShader.js and in the constructor add an instance variable for storing the reference to the View-Projection transform operator in SimpleVS.glsl.

```
this.mViewProjTransform = null;
```

2. At the end of the SimpleShader constructor, retrieve the reference to the View-Projection transform operator after retrieving those for the uModelTransform and uPixelColor.

```
// Step G: references: uniforms: uModelTransform, uPixelColor, and
//     uViewProjTransform
this.mModelTransform = gl.getUniformLocation(this.mCompiledShader,
    "uModelTransform");
this.mPixelColor = gl.getUniformLocation(this.mCompiledShader, "uPixelColor
this.mViewProjTransform = gl.getUniformLocation(
                              this.mCompiledShader, "uViewProjTransform");
```

3. Modify the activateShader function to receive a View-Projection matrix and pass it to the shader, as shown here:

```
SimpleShader.prototype.activateShader = function(vpMatrix) {
    var gl = gEngine.Core.getGL();
    gl.useProgram(this.mCompiledShader);
    gl.uniformMatrix4fv(this.mViewProjTransform, false, vpMatrix);
    gl.enableVertexAttribArray(this.mShaderVertexPositionAttribute);
    gl.uniform4fv(this.mPixelColor, pixelColor);
};
```

As you have seen previously, the gl.uniformMatrix4fv() function copies the content of vpMatrix to the uViewProjTransform operator.

Modifying RenderObject to Support the View-Projection Transform

Recall that shaders are activated in the Renderable object's draw() function; as such, Renderable must also be modified to receive and pass vpMatrix to activate the shaders.

```
Renderable.prototype.draw = function(pixelColor, vpMatrix) {
    var gl = gEngine.Core.getGL();
    this.mShader.activateShader(this.mColor, vpMatrix);  // activate first!
    this.mShader.loadObjectTransform(this.mXform.getXform());
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
};
```

Testing the View-Projection Transform and the Viewport

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

Designing the Scene

As illustrated in Figure 3-13, 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 corners of the WC.

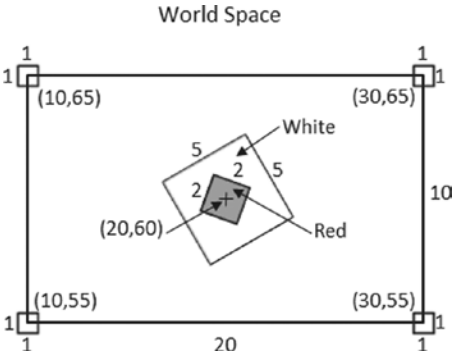

Figure 3-13. *Designing a WC to support drawing*

As illustrated in Figure 3-14, 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 matches that of the 600:300.

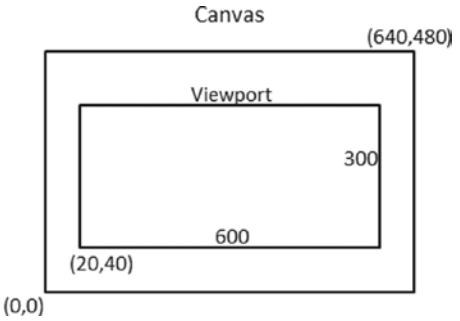

Figure 3-14. *Drawing the WC to the viewport*

Implementing the Design

The MyGame object will be modified to implement the design.

1. Edit MyGame.js to initialize the WebGL, create a constant color shader, and create six Renderable objects (two to be drawn at the center, with four at each corner of the WC) with corresponding colors.

```
// Step A: Initialize the webGL Context
gEngine.Core.initializeWebGL(htmlCanvasID);
var gl = gEngine.Core.getGL();

// Step B: Create the shader
this.mConstColorShader = new SimpleShader(
        "src/GLSLShaders/SimpleVS.glsl",     // Path to the VertexShader
        "src/GLSLShaders/SimpleFS.glsl");    // Path to the FragmentShader

// Step C: Create the Renderable objects:
this.mBlueSq = new Renderable(this.mConstColorShader);
this.mBlueSq.setColor([0.25, 0.25, 0.95, 1]);
this.mRedSq = new Renderable(this.mConstColorShader);
this.mRedSq.setColor([1, 0.25, 0.25, 1]);
this.mTLSq = new Renderable(this.mConstColorShader);
this.mTLSq.setColor([0.9, 0.1, 0.1, 1]);    // Top-Left shows red
this.mTRSq = new Renderable(this.mConstColorShader);
this.mTRSq.setColor([0.1, 0.9, 0.1, 1]);    // Top-Right shows green
this.mBRSq = new Renderable(this.mConstColorShader);
this.mBRSq.setColor([0.1, 0.1, 0.9, 1]);    // Bottom-Right shows blue
this.mBLSq = new Renderable(this.mConstColorShader);
this.mBLSq.setColor([0.1, 0.1, 0.1, 1]);    // Bottom-Left shows dark gray
```

2. Clear the entire canvas, set up the viewport, and clear the viewport to a different color.

```
// Step D: Clear the entire canvas first
gEngine.Core.clearCanvas([0.9, 0.9, 0.9, 1]);    // Clear the canvas

// Step E: Setting up Viewport
// Step E1: 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 E2: 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 E3: enable the scissor area, clear, and then disable the scissor area
gl.enable(gl.SCISSOR_TEST);
    gEngine.Core.clearCanvas([0.8, 0.8, 0.8, 1.0]);  // clear the scissor area
gl.disable(gl.SCISSOR_TEST);
```

Step E1 defines the viewport, and step E2 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.

3. Define the WC by setting up the View-Projection transform operator.

```
// Step F: Set up View and Projection matrices
var viewMatrix = mat4.create();
var projMatrix = mat4.create();

// Step F1: define the view matrix
mat4.lookAt(viewMatrix,
    [20, 60, 10],   // camera position
    [20, 60, 0],    // look at position
    [0, 1, 0]);     // orientation

// Step F2: define the projection matrix
mat4.ortho(projMatrix,
    -10,   // distant to left of WC
     10,   // distant to right of WC
    -5,    // distant to bottom of WC
     5,    // distant to top of WC
     0,    // z-distant to near plane
     1000  // z-distant to far plane
);

// Step F3: concatenate to form the View-Projection operator
var vpMatrix = mat4.create();
mat4.multiply(vpMatrix, projMatrix, viewMatrix);
```

In step F1, the `mat4.lookAt()` function defines the center of WC to be located at (20,60). Step F2 defines the distances from the center position to the left and right boundaries to be 10 units and to the top and bottom boundaries to be 5 units away. Together, these define 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 `projMatrix` and `viewMatrix` should not be swapped.

4. Set up the slightly rotated 5x5 blue square at the center of WC and draw with the `vpMatrix` operator.

```
// Step G: Draw the blue square
// Centre 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(vpMatrix);
```

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

```
// Step H: Draw with the red shader
// centre red square
this.mRedSq.getXform().setPosition(20, 60);
this.mRedSq.getXform().setSize(2, 2);
this.mRedSq.draw(vpMatrix);

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

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

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

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

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.

It is now possible to define any convenient WC system and any rectangular subregions of the canvas for drawing. With the Modeling and View-Projection 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 create abstraction to hide the details of View-Projection matrix computation.

The Camera

The View-Projection transform allows the definition of a WC to draw from. In the physical world, this is analogous to taking a photograph with the camera. The center of the viewfinder of your camera is the center of the WC, and the width and height of the world visible through the viewfinder are the dimensions of WC. With this analogy, the act of taking the photograph is equivalent to computing the image drawing of each object in the WC. Lastly, the viewport describes the location to display the computed image.

The Camera Objects Project

This project demonstrates how to abstract the View-Projection transform and the viewport to hide the details of matrix operator computation and WebGL configurations. Figure 3-15 shows the output of running the Camera Objects project; notice the output of this project is identical to that from the previous project. The source code to this project is defined in the `Chapter3/3.5.CameraObjects` folder.

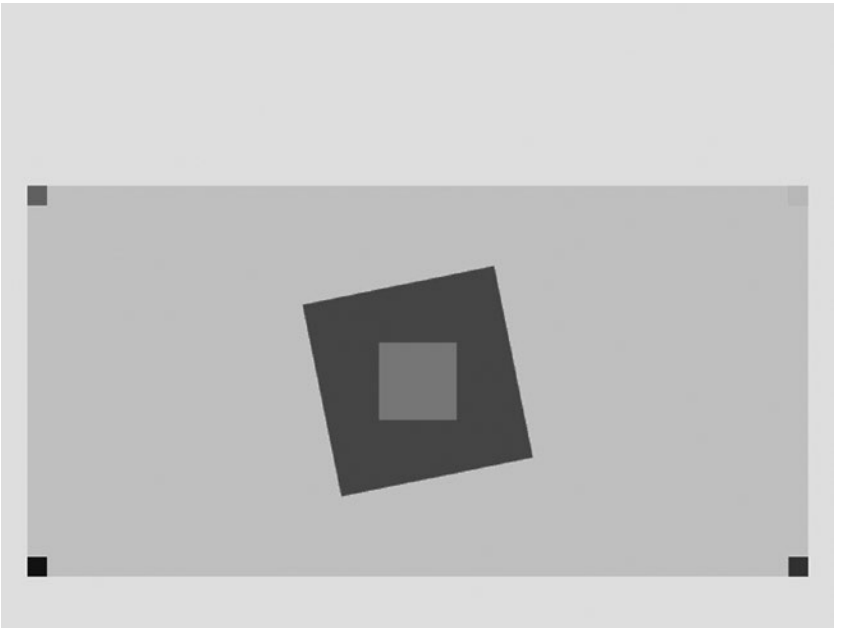

Figure 3-15. *Running the Camera Objects project*

The goals of the project are as follows:

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

The Camera Object

The Camera object basically encapsulates the functionality defined by the View-Projection and viewport setup code in the MyGame constructor from the previous example. A clean and reusable object design can be completed with appropriate getter and setter functions.

1. Define the Camera object in the game engine by creating a new source file in the src/Engine folder, and name the file Camera.js. Remember to load the new source file in index.html.

2. Add the constructor for Camera.

```
function Camera(wcCenter, wcWidth, viewportArray) {
    // WC and viewport position and size
    this.mWCCenter = wcCenter;
    this.mWCWidth = wcWidth;
    this.mViewport = viewportArray;  // [x, y, width, height]
    this.mNearPlane = 0;
    this.mFarPlane = 1000;

    // transformation matrices
    this.mViewMatrix = mat4.create();
    this.mProjMatrix = mat4.create();
    this.mVPMatrix = mat4.create();

    // background color
    this.mBgColor = [0.8, 0.8, 0.8, 1]; // RGB and Alpha
}
```

The Camera object defines the WC center and width, the viewport, and the View-Projection transform operator. Take note of the following:

   a. The mWCPosition 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 object manageable.

   c. mBgColor is an array of four floats representing the red, green, blue, and alpha components of a color.

   d. 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.

3. Add getters and setters for the instance variables.

```
// Setter/getter of WC and viewport
Camera.prototype.setWCCenter = function(xPos, yPos) {
    this.mWCCenter[0] = xPos;
    this.mWCCenter[1] = yPos;
};
```

```
Camera.prototype.getWCCenter = function() { return this.mWCCenter; };
Camera.prototype.setWCWidth = function(width) { this.mWCWidth = width; };

Camera.prototype.setViewport = function(viewportArray) {
    this.mViewport = viewportArray; };
Camera.prototype.getViewport = function() { return this.mViewport; };

Camera.prototype.setBackgroundColor = function(newColor) {
    this.mBgColor = newColor; };
Camera.prototype.getBackgroundColor = function() { return this.mBgColor; };

// Getter for the View-Projection transform operator
Camera.prototype.getVPMatrix = function() { return this.mVPMatrix; };
```

4. Create a function to compute the View-Projection operator for this Camera:

```
// Initializes the camera to begin drawing
Camera.prototype.setupViewProjection = function() {
    var gl = gEngine.Core.getGL();
    // Step A: Configure the viewport
    // ... details to follow

    // Step B: define the View-Projection matrix
    // ... details to follow
};
```

Note that this function is called `setupViewProjection()` because it configures WebGL to draw to the desire viewport and sets up the View-Projection transform operator. The following steps explain the details of steps A and B.

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

```
// Step A: Set up and clear the Viewport
// Step A1: Set up the viewport: area on canvas to be drawn
gl.viewport(this.mViewport[0], // x position of bottom-left corner
        this.mViewport[1],     // y position of bottom-left corner
        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 clear area
gl.scissor( this.mViewport[0], // x position of bottom-left corner
        this.mViewport[1],     // y position of bottom-left corner
        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 to black
gl.clearColor(this.mBgColor[0], this.mBgColor[1],
     this.mBgColor[2], this.mBgColor[3]);  // set the color to be cleared
```

```
// Step A4: enable and clear 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 use of the corresponding `Camera` instance variables.

6. The code to set up the View-Projection transform operator under step B is as follows:

```
// Step B: Set up the View-Projection transform operator
// Step B1: define the view matrix
mat4.lookAt(this.mViewMatrix,
    [this.mWCCenter[0], this.mWCCenter[1], 10],// WC center
    [this.mWCCenter[0], this.mWCCenter[1], 0], //
    [0, 1, 0]);     // orientation

// Step B2: define the projection matrix
var halfWCWidth = 0.5 * this.mWCWidth;
var halfWCHeight = halfWCWidth *  this.mViewport[3] / this.mViewport[2];
                  // WCHeight = WCWidth * viewportHeight / viewportWidth
mat4.ortho(this.mProjMatrix,
        -halfWCWidth,      // distant to left of WC
         halfWCWidth,      // distant to right of WC
        -halfWCHeight,     // distant to bottom of WC
         halfWCHeight,     // distant to top of WC
         this.mNearPlane,  // z-distant to near plane
         this.mFarPlane    // z-distant to far plane);

// Step B3: concatnate view and project matrices
mat4.multiply(this.mVPMatrix, this.mProjMatrix, this.mViewMatrix);
```

Once again, this code is similar to that from the previous example. In addition, take note that to guarantee a matching aspect ratio between WC and viewport, in step B2, the WC height, `halfWCHeight`, is computed based on the WC width, `mWCWidth`, and the aspect ratio of the viewport, which is height divided by width (`mViewport[3]/mViewport[2]`).

Testing the Camera

With the `Camera` object properly defined, testing it from `MyGame.js` is straightforward.

1. Edit `MyGame.js`; after the initialization of `WebGL`, create an instance of the `Camera` object with settings that define the WC and viewport from the previous project.

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

```
    // Step B: Setup the camera
    this.mCamera = new Camera(
            vec2.fromValues(20, 60),   // center of the WC
            20,                        // width of WC
            [20, 40, 600, 300]         // viewport (orgX, orgY, width, height)
            );
    ...
```

2. Continue with the creation of the `SimpleShader`, the six `Renderable` objects, and the clearing of the canvas.

```
// Step C: Create the shader
this.mConstColorShader = new SimpleShader(
        "src/GLSLShaders/SimpleVS.glsl",     // Path to the VertexShader
        "src/GLSLShaders/SimpleFS.glsl");    // Path to the simple FragmentShader

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

// Step E: Clear the canvas
gEngine.Core.clearCanvas([0.9, 0.9, 0.9, 1]);        // Clear the canvas
```

3. Now, draw with the `Camera` object.

```
// Step F: Starts the drawing by activating the camera
this.mCamera.setupViewProjection();
var vpMatrix = this.mCamera.getVPMatrix();

// Step G: Draw the blue square
// Centre 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(vpMatrix);

// Step H: Draw the center and the corner squares
// centre red square
this.mRedSq.getXform().setPosition(20, 60);
this.mRedSq.getXform().setSize(2, 2);
this.mRedSq.draw(vpMatrix);

... rest of the code is identical to the previous project ...
```

Step F calls `mCamera.setupViewProjection()` to compute the View-Projection transform operator. This operator is used to activate the shaders in steps G and H. The rest of the squares' model transform and drawing code are identical to the previous project.

Summary

In this chapter, you learned how to create a system that can support the drawing of many objects. The system is composed of three parts: the objects drawn, the scene drawn to, and how the scene is displayed on the browser's canvas. The objects being drawn are encapsulated by a `Renderable` object, which uses a `Transform` object to define its position, size, and rotation. Position, though, is relative to the scene you are drawing to.

You also learned that objects are all drawn relative to the scene and that a World Space, a convenient coordinate system, can be defined in order to compose the entire scene by utilizing coordinate transformations. Lastly, the View-Projection transformation is used to select which portion of the world to actually display on the canvas within a browser. This was achieved by defining an area that is viewable by the camera and using the viewport functionality provided through 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 drawing functionality to be discussed in the next chapter.