CHAPTER 5

Working with Textures, Sprites, and Fonts

After completing this chapter, you will be able to:

- Use any image or photograph as a texture representing characters or objects in your game
- Understand and use texture coordinates to identify a location on an image
- Optimize texture memory utilization by combining multiple characters and objects into one image
- Produce and control animations using sprite sheets
- Display texts of different fonts and sizes anywhere in your game

Introduction

Custom-composed images are used to represent almost all objects including characters, backgrounds, and even animations in most 2D games. For this reason, the proper support of image operations is core to 2D game engines. A game typically works with an image in three distinct stages: loading, rendering, and unloading.

Loading is the reading of the image from the hard drive of the web server into the client's system main memory, where it is processed and stored in the graphics subsystem. *Rendering* occurs during gameplay when the loaded image is drawn continuously to represent the respective game objects. *Unloading* happens when an image is no longer required by the game and the associated resources are reclaimed for future uses. Because of the slower response time of the hard drive and the potentially large amount of data that must be transferred and processed, loading images can be slower than real time. This, together with the fact that, just like the objects that images represent, the usefulness of an image is usually associated with individual game level, image loading and unloading operations typically occur during game-level transitions. To optimize the number of loading and unloading operations, it is a common practice to combine multiple lower-resolution images and form a single larger image. This larger image is referred to as a *sprite sheet*.

To represent objects, images with meaningful drawings are pasted, or *mapped*, on simple geometries. For example, a horse in a game can be represented by a square that is mapped with an image of a horse. In this way, a game developer can manipulate the transformation of the square to control the horse. This mapping of images on geometries is referred to as *texture mapping* in computer graphics.

The illusion of movement, or animation, can be created by cycling through strategically mapping selected images on the same geometry. For example, during subsequent game loop updates, different images of the same horse with strategically drawn leg positions can be mapped on the same square to create the illusion that the horse is galloping. Usually, these images of different animated positions are stored in one sprite sheet, or an animated sprite sheet, and the process of sequencing through these images to create animation is referred to as *sprite animation* or *sprite sheet animation*.

This chapter first introduces you to the concept of texture coordinates such that you can understand and program with the WebGL texture mapping interface. You will then build a core texture component and the associated supporting classes to support mapping with simple textures, working with sprite sheets that contain multiple objects, creating and controlling motions with animated sprite sheets, and extracting characters from a sprite sheet to display text messages.

■ **Note** A texture is an image that is loaded into the graphics system and ready to be mapped onto a geometry. When discussing the process of texture mapping, you'll hear "an image" and "a texture" often used interchangeably.

Texture Mapping and Texture Coordinates

As discussed, texture mapping is the process of pasting an image on a geometry, just like putting a sticker on an object. In the case of your game engine, instead of drawing a constant color for each pixel occupied by the unit square, you will create GLSL shaders to strategically select texels from the texture and display the corresponding texel colors at the screen pixel locations covered by the unit square. The process of selecting a texel, or converting a group of texels into a single color, to be displayed to a screen pixel location is referred to as *texture sampling*. To render a texture-mapped pixel, the texture must be sampled to extract a corresponding texel color.

■ **Note** Just as a pixel is a color location in an image, a texel is a color location in a texture.

The process of mapping a texture of any resolution to a fixed-size geometry can be daunting. The Texture Coordinate System that specifies the Texture Space is designed to hide the resolution of textures to facilitate this mapping process. As depicted in Figure 5-1, the Texture Coordinate System is a normalized system defined over the entire texture with the origin located at the lower-left corner and (1,1) located at the top-right corner. This simple fact, that the normalized 0 to 1 range is always defined over the entire texture regardless of the resolution, is the elegance of the Texture Coordinate System. Given a texture of any resolution, (0.5, 0.5) is always the center, (0, 1) is always the top-left corner, and so on. Notice that in Figure 5-1 the horizontal axis is labeled as the u-axis, and the vertical axis is labeled as the v-axis. Oftentimes a texture coordinate, or the uv values associated with a texture coordinate, is used interchangeably to refer to a location in the Texture Coordinate System.

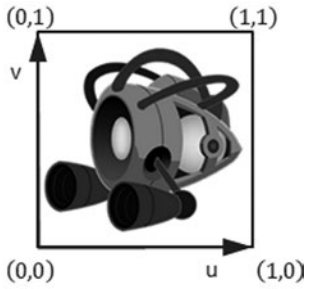

Figure 5-1. *The Texture Coordinate System and the corresponding uv values defined for all images*

■ **Note** There are conventions that define the v-axis increasing either upward or downward. In all examples of this book, you will program WebGL to follow the convention in Figure 5-1, with the v-axis increasing upward.

To map a texture onto a unit square, you must define a corresponding uv value for each of the vertex positions. As illustrated in Figure 5-2, in addition to defining the value of the xy position for each of the four corners of the square, to map an image onto this square, a corresponding uv coordinate must also be defined. In this case, the top-left corner has xy=(-0.5, 0.5) and uv=(0,1), the top-right corner has xy=(0.5, 0.5) and uv=(1, 1), and so on. Given this definition, it is possible to compute a unique uv value for any position inside the square by linearly interpolating the uv values defined at the vertices. For example, given the settings shown in Figure 5-2, you know that the midpoint along the top edge of the square maps to a uv of (0.5, 1.0) in Texture Space, the midpoint along the left edge maps to a uv of (0, 0.5), and so on.

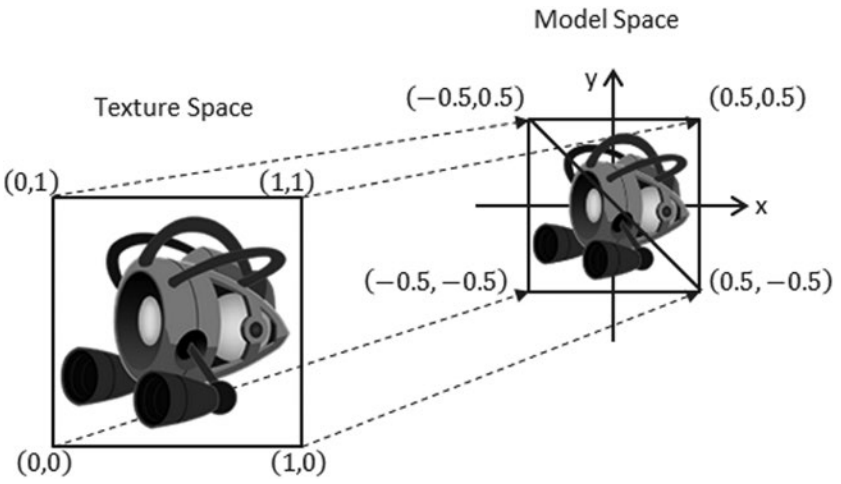

Figure 5-2. *Defining texture space uv values to map the entire image onto the geometry in Model Space*

The Texture Shaders Project

This project demonstrates the loading, rendering, and unloading of textures with WebGL. You can see an example of this project running in Figure 5-3 with the left and right screenshots from the two scenes implemented. Notice the naturally appearing objects without white borders in the left screenshot and the images with white backgrounds in the right screenshot. This project will also highlight the differences between images with and without the alpha channel, or *transparency*. The source code to this project is defined in the `Chapter5/5.1.TextureShaders` folder.

Figure 5-3. *Running the Texture Shaders project with both scenes*

The controls of the project are as follows, for both scenes:

- *Right arrow key*: Moves the middle rectangle toward the right. If this rectangle passes the right window boundary, it will be wrapped to the left side of the window.
- *Left arrow key*: Moves the middle rectangle toward the left. If this rectangle crosses the left window boundary, the game will transition to the next scene.

The goals of the project are as follows:

- To demonstrate how to define uv coordinates for geometries with WebGL
- To create a texture coordinate buffer in the graphics system with WebGL
- To build GLSL shaders to render the textured geometry
- To define the texture core engine component to load and process an image into a texture and to unload a texture
- To implement simple texture tinting, a modification of all texels with a programmer-specified color

You can find the following external resource files in the `assets` folder: a scene-level file (`BlueLevel.xml`) and four images (`minion_collector.jpg`, `minion_collector.png`, `minion_portal.jpg`, and `minion_portal.png`).

Overview

Creating and integrating textures involves relatively significant changes and new classes to be added to the game engine. The following overview contextualizes and describes the reasons for the changes:

- `TextureVS.glsl` and `TextureFS.glsl`: These are new files created to define GLSL shaders for supporting drawing with uv coordinates. Recall that the GLSL shaders must be loaded into WebGL and compiled during the initialization of the game engine.
- `Engine_VertexBuffer.js`: This file is modified to create a corresponding uv coordinate buffer to define the texture coordinate for the vertices of the unit square.
- `TextureShader.js`: This is a new file that defines `TextureShader` as a subclass of `SimpleShader` to interface the game engine to the corresponding GLSL shaders (`TextureVS` and `TextureFS`).
- `Engine_DefaultResources.js`: This is a new file that defines a core engine component to facilitate the sharing of systemwide resources. In this case, it's to facilitate the sharing of both `SimpleShader` and `TextureShader` by the corresponding `Renderable` objects.
- `Renderable.js`: This file is modified to facilitate `Renderable` serving as the base class to all future types of `Renderable` objects and to share the `Shader` resource provided by `gEngine_DefaultResources`.
- `TextureRenderable.js`: This is a new file that defines `RenderableTexture` as a subclass of `Renderable` to facilitate the creation, manipulation, and drawing of multiple instances of textured objects.
- `Engine_Core.js`: This file is modified to configure WebGL to support drawing with a texture map.
- `Engine_Textures.js`: This is a new file that defines the core engine component that is capable of loading, activating (for rendering), and unloading texture images.
- `MyGame.js` and `BlueLevel.js`: These game engine client files are modified to test the new texture mapping functionality.

Extension of Shader/Renderable Architecture

Recall that the `Shader/Renderable` object pair is designed to load relevant game engine data to the `SimpleVS/FS` GLSL shaders and to support instantiating multiple copies of renderable geometries by the game engine clients. As illustrated in Figure 5-4, the horizontal dotted line separates the game engine from WebGL. Notice that the GLSL shaders, `SimpleVS` and `SimpleFS`, are modules in WebGL and outside the game engine. The `Shader` object maintains references to all attributes and uniform variables in the GLSL shaders and acts as the conduit for sending all transformation and vertex information to the `SimpleVS/FS` shaders. Although not depicted explicitly in Figure 5-4, there is only one instance of the `Shader` object created in the game engine, in `Engine.DefaultResoruces`, and this instance is shared by all `Renderable` objects.

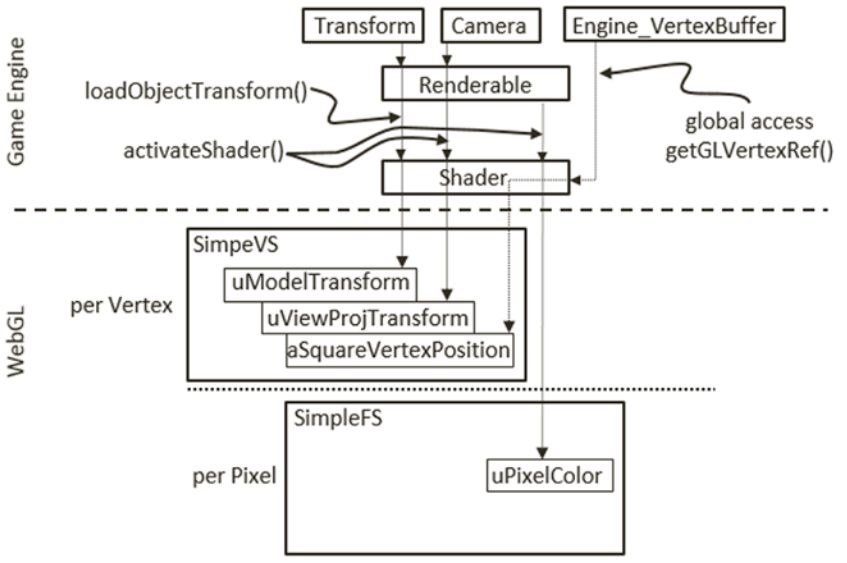

Figure 5-4. *The* `Shader` *and* `Renderable` *architecture*

The proper support of texture mapping demands new GLSL vertex and fragment shaders and thus requires that a corresponding shader and renderable object pair be defined in the game engine. As illustrated in Figure 5-5, both the GLSL `TextureVS/FS` shaders and `TextureShader/TextureRenderable` object pair are extensions (or subclasses) to the corresponding existing objects. The `TextureShader/TextureRenderable` object pair extends from the corresponding `Shader/Renderable` objects to forward texture coordinates to the GLSL shaders. The `TextureVS/FS` shaders are extensions to the corresponding `SimpleVS/FS` shaders to read texels from the provided texture map when computing pixel colors. Note that since GLSL does not support subclassing, the `TextureVS/FS` source code is copied from the `SimpleVS/FS` files.

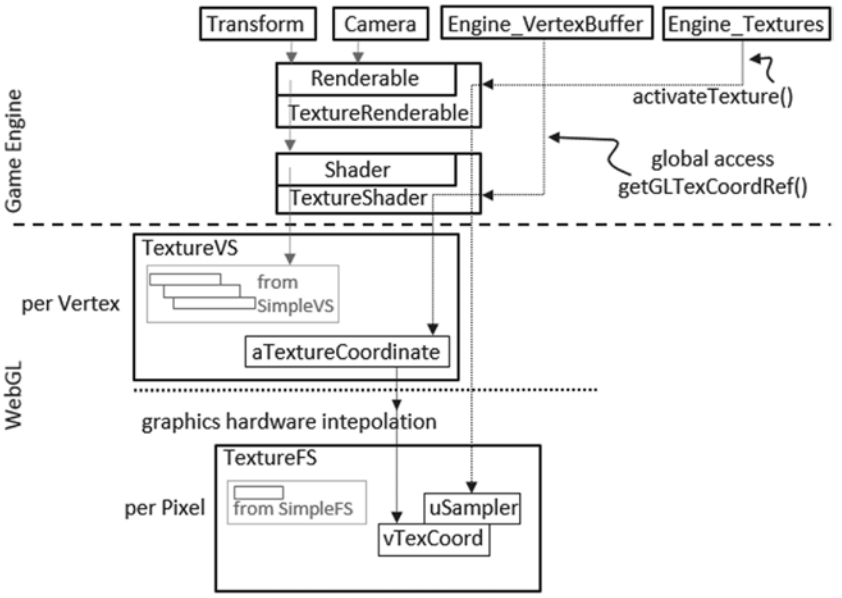

Figure 5-5. *The TextureVS/FS GLSL shaders and the corresponding TextureShader/TextureRenderable object pair*

GLSL Texture Shader

To support drawing with textures, you must create a shader that accepts both geometric (xy) and texture (uv) coordinates at each of the vertices. You will create new GLSL texture vertex and fragment shaders by copying and modifying the corresponding `SimpleVS` and `SimpleFS` programs. Now, create the texture vertex shader.

1. Create a new file in the `src/GLSLShaders/` folder and name it `TextureVS.glsl`.
2. Add the following code to the `TextureVS.glsl` file:

```
attribute vec3 aSquareVertexPosition; // Expects one vertex position
attribute vec2 aTextureCoordinate;

// texture coordinate that will map the entire image to the entire square
varying vec2 vTexCoord;

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

void main(void) {
    gl_Position = uViewProjTransform * uModelTransform *
        vec4(aSquareVertexPosition, 1.0);

    // pass the texture coordinate to the fragment shader
    vTexCoord = aTextureCoordinate;
}
```

You may notice that the TextureVS shader is similar to the SimpleVS shader, with only three additional lines of code.

a. Add the aTextureCoordinate attribute. This defines a vertex to include a vec3 (aSquareVertexPosition, the xyz position of the vertex) and a vec2 (aTextureCooridnate, the uv coordinate of the vertex).

b. Declare the varying vTexCoord variable. The varying keyword in GLSL signifies that the associated variable will be linearly interpolated and passed to the fragment shader. As explained earlier and illustrated in Figure 5-2, uv values are defined only at vertex positions. In this case, the varying vTexCoord variable instructs the graphics hardware to linearly interpolate the uv values to compute the texture coordinate for each invocation of the fragment shader.

c. Assign the vertex uv coordinate values to the varying variable for interpolation and forwarding to the fragment shader.

With the vertex shader defined, you can now create the associated fragment shader.

1. Create a new file in the src/GLSLShaders/ folder and name it TextureFS.glsl.

2. Add the following code to the TextureFS.glsl file to declare the variables:

```
precision mediump float; // precision for floating point computation

// The object that fetches data from texture.
// Must be set outside the shader.
uniform sampler2D uSampler;

// Color of pixel
uniform vec4 uPixelColor;

// The "varying" keyword is for signifing that the texture coordinate will be
// interpolated and thus varies.
varying vec2 vTexCoord;
```

The sampler2D data type is a GLSL utility that is capable of reading texel values from a 2D texture. In this case, the uSampler object will be bounded to a GLSL texture such that texel values can be sampled for every pixel rendered. The uPixelColor is the same as the one from SimpleFS. The vTexCoord is the interpolated uv coordinate value for each pixel.

3. Add the following code to compute the color for each pixel:

```
void main(void)  {
    // texel color look up based on interpolated UV value in vTexCoord
    vec4 c = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));

    // tint the textured area. Leave transparent area as defined by the texture
    vec3 r = vec3(c) * (1.0-uPixelColor.a) + vec3(uPixelColor) * uPixelColor.a;
    vec4 result = vec4(r, c.a);

    gl_FragColor = result;
}
```

4. The `texture2D()` function samples and reads the texel value from the texture that is associated with `uSampler` using the interpolated uv values from `vTexCoord`. In this example, the texel color is modified, or tinted, by a weighted sum of the color value defined in `uPixelColor` according to the *transparency*, or the value of the corresponding alpha channel. In general, there is no agreed-upon definition for tinting texture colors. You are free to experiment with different ways to combine `uPixelColor` and the sampled texel color; for example, you can try multiplying the two. In the provided source code file, a few alternatives are suggested. Please do experiment with them.

Define and Set Up Texture Coordinates

Recall that all shaders share the same xy coordinate buffer of a unit square that is defined in the `Engine_VertexBuffer.js` file. In a similar fashion, a corresponding buffer must be defined to supply texture coordinates to the GLSL shaders.

1. Modify `Engine_VertexBuffer.js` to define both xy and uv coordinates for the unit square.

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

// reference to the texture positions for the square vertices in the gl context
var mTextureCoordBuffer = null;

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

// Second: define the corresponding texture cooridnates
var textureCoordinates = [
    1.0, 1.0,
    0.0, 1.0,
    1.0, 0.0,
    0.0, 0.0
];
```

As illustrated in Figure 5-2, the `textureCoordinates` variable defines the uv values for the corresponding four xy values of the unit square defined in `verticesOfSquare`, sequentially. For example, (1, 1) are the uv values associated with the (0.5, 0.5, 0) xy position, (0, 1) for (-0.5, 0.5, 0), and so on. The new `mTextureCoordBuffer` instance variable will be initialized to refer to the WebGL buffer that stores the values of `textureCoordinates`.

2. Modify the `initialize()` function to the following:

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

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

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

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

    // Step  B: Allocate and store texture coordinates
    // Create a buffer on the gGL context for our vertex positions
    mTextureCoordBuffer = gl.createBuffer();

    // Activate vertexBuffer
    gl.bindBuffer(gl.ARRAY_BUFFER, mTextureCoordBuffer);

    // Loads verticesOfSquare into the vertexBuffer
    gl.bufferData(gl.ARRAY_BUFFER,
            new Float32Array(textureCoordinates), gl.STATIC_DRAW);
};
```

Step B of the `initialize()` function handles the initialization of the texture coordinates as a WebGL buffer and is identical to how the vertex xy coordinates are handled with the `mTextureCoordBuffer` variable.

3. Add a function to retrieve the texture coordinates.

```
var getGLTexCoordRef = function() { return mTextureCoordBuffer; };
```

4. Finally, remember to add the changes to the public interface.

```
var mPublic = {
    initialize: initialize,
    getGLVertexRef: getGLVertexRef,
    getGLTexCoordRef: getGLTexCoordRef
};
```

Interface GLSL Shader to the Engine

Just as the `SimpleShader` object was defined to interface to the `SimpleVS` and `SimpleFS` shaders, a corresponding shader object needs to be created in the game engine to interface to the `TextureVS` and `TextureFS` GLSL shaders. In addition, you will create a new folder to organize the growing number of different shaders.

1. Create a new folder called `Shaders` in `src/Engine`. Move the `SimpleShader.js` file into this folder, and do not forget to update the reference path in `index.html`.

2. Create a new file in the `src/Engine/Shaders/` folder and name it `TextureShader.js`. Add the following code to construct the object:

```
// constructor
function TextureShader(vertexShaderPath, fragmentShaderPath) {
    // Call super class constructor
    SimpleShader.call(this, vertexShaderPath, fragmentShaderPath);

    // reference to aTextureCoordinate within the shader
    this.mShaderTextureCoordAttribute = null;

    // get the reference of aTextureCoordinate from the shader
    var gl = gEngine.Core.getGL();
    this.mShaderTextureCoordAttribute =
        gl.getAttribLocation(this.mCompiledShader, "aTextureCoordinate");
}
// get all the prototype functions from SimpleShader
gEngine.Core.inheritPrototype(TextureShader, SimpleShader);
```

The `SimpleShader.call()` syntax invokes the constructor of `SimpleShader` with the current `TextureShader` object as the caller. This line of code is simply invoking the superclass constructor. Recall that the `SimpleShader` constructor will load and compile the GLSL shaders defined by the `vertexShaderPath` and `fragmentShaderPath` and will locate a reference to the `mShaderVertexPositionAttribute` attribute defined in the shader. In the rest of the `TextureShader` constructor, the `mShaderTextureCoordAttribute` keeps a reference to the `aTextureCoordinate` attribute defined in the `TextureVS`. In this way, both of the vertex attributes defined in `TextureVS.glsl` are referenced by the JavaScript `TextureShader` object.

3. Override the `activateShader()` function to enable the texture coordinate data.

```
// Overriding the Activation of the shader for rendering
TextureShader.prototype.activateShader = function(pixelColor, vpMatrix) {
    // first call the super class's activate
    SimpleShader.prototype.activateShader.call(this, pixelColor, vpMatrix);

    // now our own functionality: enable texture coordinate array
    var gl = gEngine.Core.getGL();
    gl.bindBuffer(gl.ARRAY_BUFFER, gEngine.VertexBuffer.getGLTexCoordRef());
    gl.enableVertexAttribArray(this.mShaderTextureCoordAttribute);
    gl.vertexAttribPointer(this.mShaderTextureCoordAttribute, 2, gl.FLOAT,
        false, 0,0);
};
```

The superclass `activateShader.call()` sets up the xy vertex position and passes the `pixelColor` to the shader. The rest of the code binds `mShaderTextureCoordAttribute` to the texture coordinate buffer defined in the `gEngine.VertexBuffer` component, as discussed in the Define and Set up Up Texture Coordinates section previously.

In this way, after the `activateShader()` function call, both the geometry's xy coordinate (`mShaderVertexPositionAttribute`) and the texture's uv coordinate (`mShaderTextureCoordAttribute`) of each vertex are connected to the corresponding buffers in the GLSL shaders.

Facilitate Sharing with Engine_DefaultResources

In the same manner as `SimpleShader` is a reusable resource, only one instance of the `TextureShader` needs to be created, and this instance can be shared. The `DefaultResources` component should be modified to reflect this.

1. In `Engine_DefaultResources.js`, add the variables to hold a texture shader.

```
// Texture Shader
var kTextureVS = "src/GLSLShaders/TextureVS.glsl";  // Path to VertexShader
var kTextureFS = "src/GLSLShaders/TextureFS.glsl";  // Path to FragmentShader
var mTextureShader = null;
```

2. Define a function to retrieve the texture shader.

```
var getTextureShader = function() { return mTextureShader; };
```

3. Add the creation of the texture shader to the `_createShaders()` function.

```
var _createShaders = function(callBackFunction) {
    mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);
    mTextureShader = new TextureShader(kTextureVS, kTextureFS);
    callBackFunction();
};
```

4. Modify the `initialize()` function to load the source files for the texture shader.

```
var initialize = function(callBackFunction) {
    // constant color shader: SimpleVS, and SimpleFS
    gEngine.TextFileLoader.loadTextFile(kSimpleVS,
        gEngine.TextFileLoader.eTextFileType.eTextFile);
    gEngine.TextFileLoader.loadTextFile(kSimpleFS,
        gEngine.TextFileLoader.eTextFileType.eTextFile);

    // texture shader:
    gEngine.TextFileLoader.loadTextFile(kTextureVS,
        gEngine.TextFileLoader.eTextFileType.eTextFile);
    gEngine.TextFileLoader.loadTextFile(kTextureFS,
        gEngine.TextFileLoader.eTextFileType.eTextFile);

    gEngine.ResourceMap.setLoadCompleteCallback(
        function(){ _createShaders(callBackFunction); }
    );
};
```

5. Lastly, remember to add the changes to the public interface.

```
var mPublic =
{
    initialize: initialize,
    getConstColorShader: getConstColorShader,
    getTextureShader: getTextureShader
};
```

Renderable Texture Object

Just as the `Renderable` class encapsulates and facilitates the definition and drawing of multiple instances of `SimpleShader` objects, a corresponding `TextureRenderable` class needs to be defined to support the drawing of multiple instances of `TextureShader` objects.

Changes to the Renderable Object

For the same reason as creating and organizing shader classes in the `Shaders` folder, a `Renderables` folder should be created to organize the growing number of different kinds of `Renderable` objects. In addition, the `Renderable` class must be modified to support it being the base class of all `Renderable` objects.

1. Create the `src/Engine/Renderables` folder and move `Renderable.js` into this folder. Remember to update `index.html` to reflect the change.

2. Modify the `Renderable` constructor to refer to `SimpleShader`, the constant color shader, by default.

```
function Renderable() {
    this.mShader = gEngine.DefaultResources.getConstColorShader();
    this.mXform = new Transform(); // transform that moves this object around
    this.mColor = [1, 1, 1, 1];
}
```

3. Define a function that sets the shader for the `Renderable`.

```
Renderable.prototype._setShader = function(s) { this.mShader = s; };
```

This is a protected function designed for subclasses to modify the `mShader` variable to refer to the appropriate shaders for each corresponding subclass.

The TextureRenderable Object

You are now ready to create the `TextureRenderable` object. As noted, `TextureRenderable` is derived from and extends the `Renderable` object functionality to render the object with a texture mapped to it.

1. Create a new file in the src/Engine/Renderables/ folder and name it TextureRenderable.js. Add the constructor.

```
// Constructor and object definition
function TextureRenderable(myTexture) {
    Renderable.call(this);
    Renderable.prototype.setColor.call(this, [1, 1, 1, 0]);
        // Alpha 0: switch off tinting
    Renderable.prototype._setShader.call(this, gEngine.DefaultResources.
        getTextureShader());
    this.mTexture = myTexture; // the object's texture, cannot be null.
}
gEngine.Core.inheritPrototype(TextureRenderable, Renderable);
```

Renderable.call(this) is a call to the superclass (Renderable) constructor. Likewise, the setColor() and _setShader() functions are invoked with the TextureRenderable context to set the corresponding variables in the superclass. As will be discussed, the myTexture parameter is the path to the file that contains the texture image.

2. Define a draw() function to overwrite the function defined in the Renderable object to support textures.

```
TextureRenderable.prototype.draw = function(vpMatrix) {
    // activate the texture
    gEngine.Textures.activateTexture(this.mTexture);
    Renderable.prototype.draw.call(this, vpMatrix);
};
```

The activateTexture() function activates and allows drawing with the specific texture. The details of this function will be discussed in the following section.

3. Finally, define a getter and setter for the texture reference.

```
TextureRenderable.prototype.getTexture = function() { return this.mTexture; };
TextureRenderable.prototype.setTexture = function(t) { this.mTexture = t; };
```

Texture Support in the Engine

To support drawing with textures, the rest of the game engine requires two main modifications: WebGL context configuration and a dedicated engine component to support operations associated with textures.

Configure WebGL to Support Textures

The configuration of WebGL context must be updated to support textures. In Engine_Core.js, update _initializeWebGL() according to the following:

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

```
    // Get standard webgl, or experimental
    // binds webgl to the Canvas area on the web-page to the variable mGL
    mGL = canvas.getContext("webgl", {alpha: false}) ||
        canvas.getContext("experimental-webgl", {alpha: false});

    // Allows transperency with textures.
    mGL.blendFunc(mGL.SRC_ALPHA, mGL.ONE_MINUS_SRC_ALPHA);
    mGL.enable( mGL.BLEND ) ;

    // Set images to flip the y axis to match the texture coordinate space.
    mGL.pixelStorei(mGL.UNPACK_FLIP_Y_WEBGL, true);

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

The parameter passed to `canvas.getContext()` informs the browser that the canvas should be opaque. This can speed up the drawing of transparent content and images. The `blendFunc()` function enables transparencies when drawing images with the alpha channel. The `pixelStorei()` function defines the origin of the uv coordinate to be at the lower-left corner.

Create the Texture Management Engine Component

Like audio and text files, a new engine component must be defined to support the corresponding operations, including reading from the server file system, loading to the WebGL context, activating the WebGL texture buffer for drawing, and unloading from WebGL.

1. Create a new file in the `src/Engine/Core/` folder and name it `Engine_Textures.js`. This file will implement the `Textures` engine component.
2. Add the following object definition to represent a texture in the game engine:

```
function TextureInfo(name, w, h, id) {
    this.mName = name;
    this.mWidth = w;
    this.mHeight = h;
    this.mGLTexID = id;
};
```

`mWidth` and `mHeight` are the pixel resolution of the texture image, `mName` stores the path to the image file, and `mGLTexID` is a reference to the WebGL texture storage.

■ **Note** For an efficient implementation, many graphics hardware only supports texture with image resolutions in powers of 2, such as 2x4 ($2^1 x 2^2$), or 4x16 ($2^2 x 2^4$), or 64x256 ($2^6 x 2^8$), and so on. This is also the case for your configuration of WebGL. All examples in this book work only with textures with resolutions that are powers of 2.

---

3. Now, define a `Textures` component similar to the other engine components.

```
var gEngine = gEngine || { };

gEngine.Textures = (function(){
    var mPublic = { };
    return mPublic;
}());
```

4. Define a function to load an image asynchronously.

```
// Loads an texture so that it can be drawn.
// If already in the map, will do nothing.
var loadTexture = function(textureName) {
    if (!(gEngine.ResourceMap.isAssetLoaded(textureName)))  {
        // Create new Texture object.
        var img = new Image();

        // Update resources in loading counter.
        gEngine.ResourceMap.asyncLoadRequested(textureName);

        // When the texture loads, convert it to the WebGL format then put
        // it back into the mTextureMap.
        img.onload = function () {
            _processLoadedImage(textureName, img);
        };
        img.src = textureName;
    } else {
        gEngine.ResourceMap.incAssetRefCount(textureName);
    }
};
```

Note the similarity between the loading of textures and the loading of audio or text files. In this case, once an image is loaded, it is passed to the `_processLoadedImage()` function with its file path as the name.

5. Add an `unloadTexture()` function to clean up the engine and release WebGL resources.

```
// Remove the reference to allow associated memory
// be available for subsequent garbage collection
var unloadTexture = function(textureName) {
    var gl = gEngine.Core.getGL();
    var texInfo = gEngine.ResourceMap.retrieveAsset(textureName);
    gl.deleteTexture(texInfo.mGLTexID);
    gEngine.ResourceMap.unloadAsset(textureName);
};
```

6. Now define a function to convert the format of an image and store it to the WebGL context.

```
var _processLoadedImage = function(textureName, image) {
    var gl = gEngine.Core.getGL();

    // Creates a WebGL texture object
    var textureID = gl.createTexture();

    // bind texture with the current texture functionality in webGL
    gl.bindTexture(gl.TEXTURE_2D, textureID);

    // Load the texture into the texture data structure with descriptive info.
    // Parameters:
    //  1: Which "binding point" or target the texture is being loaded to.
    //  2: Level of detail. Used for mipmapping. 0 is base texture level.
    //  3: Internal format. The composition of each element, i.e. pixels.
    //  4: Format of texel data. Must match internal format.
    //  5: The data type of the texel data.
    //  6: Texture Data.
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);

    // Creates a mipmap for this texture.
    gl.generateMipmap(gl.TEXTURE_2D);

    // Tells WebGL we are done manipulating data at the mGL.TEXTURE_2D target.
    gl.bindTexture(gl.TEXTURE_2D, null);

    var texInfo = new TextureInfo(textureName,
        image.naturalWidth, image.naturalHeight, textureID);
    gEngine.ResourceMap.ayncLoadCompleted(textureName, texInfo);
};
```

The `createTexture()` function creates a WebGL texture buffer and returns a unique ID. The `texImage2D()` function stores the image into the WebGL texture buffer, and `generateMipmap()` computes a mipmap for the texture. Lastly, a `TextureInfo` object is instantiated to refer to the WebGL texture and stored into `ResourceMap` according to the file path to the texture image file.

■ **Note** A *mipmap* is a representation of the texture image that facilitates high-quality rendering. Please consult a computer graphics reference book to learn more about mipmap representation and the associated texture mapping algorithms.

---

7. Define a function to activate a WebGL texture for drawing.

```
var activateTexture = function (textureName) {
        var gl = gEngine.Core.getGL();
        var texInfo = gEngine.ResourceMap.retrieveAsset(textureName);

        // Binds our texture reference to the current webGL texture functionality
        gl.bindTexture(gl.TEXTURE_2D, texInfo.mGLTexID);

        // To prevent texture wrappings
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);

        // Handles how magnification and minimization filters will work.
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
            gl.LINEAR_MIPMAP_LINEAR);

        // For pixel-graphics where you want the texture to look "sharp"
        //     do the following:
        // gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
        // gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
    };
```

a. The `retrieveAsset()` function locates the `TextureInfo` object from the `ResourceMap` based on the `textureName`. The located `mGLTexID` is used in the `bindTexture()` function to activate the corresponding WebGL texture buffer for rendering.

b. The `texParameteri()` function defines the rendering behavior for the texture. The `TEXTURE_WRAP_S/T` parameters ensure that the texel values will not wrap around at the texture boundaries. The `TEXTURE_MAG_FILTER` parameter defines how to magnify a texture, in other words, when a low-resolution texture is rendered to many pixels in the game window. The `TEXTURE_MIN_FILTER` parameter defines how to minimize a texture, in other words, when a high-resolution texture is rendered to a small number of pixels. The `LINEAR` and `LINEAR_MIPMAP_LINEAR` configurations generate smooth textures by blurring the details of the original images, while the commented-out `NEAREST` option will result in sharp textures but color boundaries of the texture image may appear jagged.

■ **Note** In general, it is best to use texture images with similar resolution as the number of pixels occupied by the objects in the game.

---

8. Define a function to deactivate a texture as follows:

```
var deactivateTexture = function() {
   var gl = gEngine.Core.getGL();
   gl.bindTexture(gl.TEXTURE_2D, null);
};
```

This function sets the WebGL context to a state of not working with a texture.

9. Add a getter function to retrieve the texture information by its name (a file path).

```
var getTextureInfo = function(textureName) {
    return gEngine.ResourceMap.retrieveAsset(textureName);
}
```

10. Finally, remember to add the public functions to the public interface.

```
// Public interface for this object. Anything not in here will
// not be accessable.
var mPublic = {
    loadTexture: loadTexture,
    unloadTexture: unloadTexture,
    activateTexture: activateTexture,
    deactivateTexture: deactivateTexture,
    getTextureInfo: getTextureInfo
};
return mPublic;
```

Testing of Texture Mapping Functionality

With the previous modifications, the game engine can now render constant color objects as well as objects with interesting and different types of textures. The following testing code is similar to that from the previous example where two scenes, `MyGame` and `BlueLevel`, are used to demonstrate the newly added texture mapping functionality. The main modifications include the loading and unloading of texture images and the creation and drawing of `TextureRenderable` objects. In addition, the `MyGame` scene highlights transparent texture maps with alpha channel using PNG images, and the `BlueScene` scene shows corresponding textures with images in JPEG format.

As in all cases of building a game, it is essential to ensure that all external resources are located at proper locations. Recall that the `assets` folder is created specifically for this purpose. Take note of the four new texture files located in the `assets` folder: `minion_collector.jpg`, `minion_collector.png`, `minion_portal.jpg`, and `minion_portal.png`.

Modify the BlueLevel Scene File to Support Textures

The `BlueLevel.xml` scene file is modified from the previous example to support texture mapping.

```
<MyGameLevel>
    <!-- cameras -->
                <!-- Viewport: x, y, w, h -->
    <Camera CenterX="20" CenterY="60" Width="20"
            Viewport="20 40 600 300"
            BgColor="0 0 1 1.0"/>

    <!-- The red rectangle -->
    <Square PosX="20" PosY="60" Width="2" Height="3" Rotation="0"  Color="1 0 0 1" />
```

```
    <!-- Textures Square -->
    <TextureSquare PosX="15" PosY="60" Width="3" Height="3" Rotation="-5"
        Color="1 0 0 0.3"
        Texture="assets/minion_portal.jpg" />

    <TextureSquare PosX="25" PosY="60" Width="3" Height="3" Rotation="5"
        Color="0 0 0 0"
        Texture="assets/minion_collector.jpg"/>
        <!-- without tinting, alpha should be 0 -->
</MyGameLevel>
```

The TextureSquare element is similar to Square with the addition of a Texture attribute that specifies which image file should be used as a texture map for the square. Note that as implemented in TextureFS.glsl, the alpha value of the Color element is used for tinting the texture map. The previous code shows slight tinting of the minion_portal.jpg texture and no tinting of the minion_collector.jpg texture. This texture tinting effect can be observed in the right image of Figure 5-3. In addition, notice that both images specified are in the JPEG format. Since the JPEG format does not support the storing of alpha channel, there are the white areas outside the portal and collector minions in the right image of Figure 5-3.

Modify SceneFileParser

The scene file parser, SceneFileParser.js, is modified to support the parsing of the updated BlueScene.xml, in particular, to parse Square elements into Renderable objects and TextureSquare elements into TextureRenderable objects. For details of the changes, please refer to the source code file in the src/MyGame/Util folder.

Test BlueLevel with JPEGs

The modifications to BlueLevel.js are in the loadScene(), unloadScene(), and initialize() functions where the texture images are loaded and unloaded and new TextureRenderable objects are parsed.

1. Modify the constructor to define constants to represent the texture images that will be used.

```
function BlueLevel() {
    // scene file name
    this.kSceneFile = "assets/BlueLevel.xml";

    // textures: ( Note: jpg does not support transparency )
    this.kPortal = "assets/minion_portal.jpg";
    this.kCollector = "assets/minion_collector.jpg";

    // all square
    this.mSqSet = [];        // these are the renderable objects

    // The camera to view the rectangles
    this.mCamera = null;
};
gEngine.Core.inheritPrototype(BlueLevel, Scene);
```

2. Initiate loading of the textures in the `loadScene()` function.

```
BlueLevel.prototype.loadScene = function() {
    // load the scene file
    gEngine.TextFileLoader.loadTextFile(this.kSceneFile,
        gEngine.TextFileLoader.eTextFileType.eXMLFile);

    // load the textures
    gEngine.Textures.loadTexture(this.kPortal);
    gEngine.Textures.loadTexture(this.kCollector);
};
```

3. Likewise, add code to clean up by unloading the textures in the `unloadScene()` function.

```
BlueLevel.prototype.unloadScene = function() {
    // unload the scene flie and loaded resources
    gEngine.TextFileLoader.unloadTextFile(this.kSceneFile);
    gEngine.Textures.unloadTexture(this.kPortal);
    gEngine.Textures.unloadTexture(this.kCollector);

    var nextLevel = new MyGame();  // load the next level
    gEngine.Core.startScene(nextLevel);
};
```

4. Parse the textured squares in the `initialize()` function.

```
BlueLevel.prototype.initialize = function() {
    var sceneParser = new SceneFileParser(this.kSceneFile);

    // Step A: Read in the camera
    this.mCamera = sceneParser.parseCamera();

    // Step B: Read all the squares and textureSquares
    sceneParser.parseSquares(this.mSqSet);
    sceneParser.parseTextureSquares(this.mSqSet);
};
```

5. Include extra code in the `update()` function to continuously change the tinting of the portal `TextureRenderable`, as follows:

```
BlueLevel.prototype.update = function() {
    // ... Identical to previous code ...

    // continously change texture tinting
    var c = this.mSqSet[1].getColor();
    var ca = c[3] + deltaX;
    if (ca > 1) {
      ca = 0;
    }
    c[3] = ca;
};
```

a. Index 1 of `mSqSet` is the portal `TextureRenderable` object, and index 3 of the color array is the alpha channel.

b. The previous code continuously increases and wraps the alpha value of the `mColor` variable in the `TextureRenderable` object. Recall this variable is passed to `TextureShader` and then loaded into the `uPixelColor` of `TextureFS` for tinting the texture map results.

c. As defined in the first `TextureSquare` element in the `BlueScene.xml` file, the color defined for the portal object is red. For this reason, when running the example for this project, in the blue level the portal object appears to be blinking in red.

Test MyGame with PNGs

Similar to the `BlueLevel` scene, `MyGame` is a straightforward modification of the previous example with changes to load and unload texture images and to create `TextureRenderable` objects.

1. Modify the `MyGame` constructor to define texture image files that will be used and the variables for referencing the `TextureRenderable` objects that will be instantiated.

```
function MyGame() {
    // textures: ( Note: supports png with transparency )
    this.kPortal = "assets/minion_portal.png";
    this.kCollector = "assets/minion_collector.png";

    // The camera to view the rectangles
    this.mCamera = null;

    // the hero and the support objects
    this.mHero = null;
    this.mPortal = null;
    this.mCollector = null;
};
gEngine.Core.inheritPrototype(MyGame, Scene);
```

2. Load the textures in `loadScene()`.

```
MyGame.prototype.loadScene = function() {
   // loads the textures
   gEngine.Textures.loadTexture(this.kPortal);
   gEngine.Textures.loadTexture(this.kCollector);
};
```

3. Make sure you remember to unload the textures in `unloadScene()`.

```
MyGame.prototype.unloadScene = function() {
    // Game loop not running, unload all assets
    gEngine.Textures.unloadTexture(this.kPortal);
    gEngine.Textures.unloadTexture(this.kCollector);
```

```
    // starts the next level
    var nextLevel = new BlueLevel();  // next level to be loaded
    gEngine.Core.startScene(nextLevel);
};
```

4. Create and initialize the `TextureRenderables` objects in the `initialize()` function.

```
MyGame.prototype.initialize = function() {
    // Step A: set up the cameras
    this.mCamera = new Camera(
            vec2.fromValues(20, 60),   // position of the camera
            20,                        // width of camera
            [20, 40, 600, 300]         // viewport (orgX, orgY, width, height)
            );
    this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);
            // sets the background to gray

    // Step B: Create the game objects
    this.mPortal = new TextureRenderable(this.kPortal);
    this.mPortal.setColor([1, 0, 0, 0.2]);  // tints red
    this.mPortal.getXform().setPosition(25, 60);
    this.mPortal.getXform().setSize(3, 3);

    this.mCollector = new TextureRenderable(this.kCollector);
    this.mCollector.setColor([0, 0, 0, 0]);  // No tinting
    this.mCollector.getXform().setPosition(15, 60);
    this.mCollector.getXform().setSize(3, 3);

    // Step C: Create the hero object in blue
    this.mHero = new Renderable();
    this.mHero.setColor([0, 0, 1, 1]);
    this.mHero.getXform().setPosition(20, 60);
    this.mHero.getXform().setSize(2, 3);
};
```

Remember that the texture file path is used as the unique identifier in the `ResourceMap`. For this reason, it is essential for file texture loading and unloading and for the creation of `TextureRenderable` objects to refer to the same file path. In the previous code, all three functions refer to the same constants defined in the constructor.

5. The modification to the `draw()` function draws the two new `TextureRenderable` objects by calling their corresponding `draw()` functions, while the modification to the `update()` function is similar to that of the `BlueLevel` discussed earlier. Please refer to the `MyGame.js` source code file in the `src/MyGame` folder for details.

When running the example for this project in the `Chapter5/5.1.TextureShaders` folder, once again take note of the results of continuously changing the texture tinting—the blinking of the portal minion in red. In addition, notice the differences between the PNG-based textures in the `MyGame` level and the corresponding JPEG ones with white borders in the `BlueLevel`. It is visually more pleasing and accurate to represent objects using textures with the alpha (or transparency) channel. PNG is one of the most popular image formats supporting the alpha channel.

Observations

This project has been the longest and most complicated one that you have worked with. This is because working with texture mapping requires you to understand texture coordinates, the implementation cuts across many of the files in the engine, and the fact that actual images must be loaded, converted into textures, and stored/accessed from WebGL. To help summarize the changes, Figure 5-6 shows the game engine states in relation to the states of an image used for texture mapping and some of the main game engine operations.

The left column of Figure 5-6 identifies the main game engine states, from WebGL initialization to the initialization of a scene, to the game loop, and to the eventual unloading of the scene. The middle column shows the corresponding states of an image that will be used as a texture. Initially, this image is stored on the server file system. During scene initialization, the `loadScene()` function will invoke the `gEngine_Texture.loadTexture()` function to load the image and process it with the `gEngine_Texture._processLoadedImage()` function into a corresponding WebGL texture and store it in a WebGL context. During the game loop cycles, the `TextureRenderable.draw()` function activates the appropriate WebGL texture via the `gEngine_Texture.activateTexture()` function. This enables the corresponding GLSL fragment shader sample from the correct texture during rendering. Finally, when a texture is no longer needed by the game engine, the `gEngineTexture_unloadTexture()` call will remove its existence from the system.

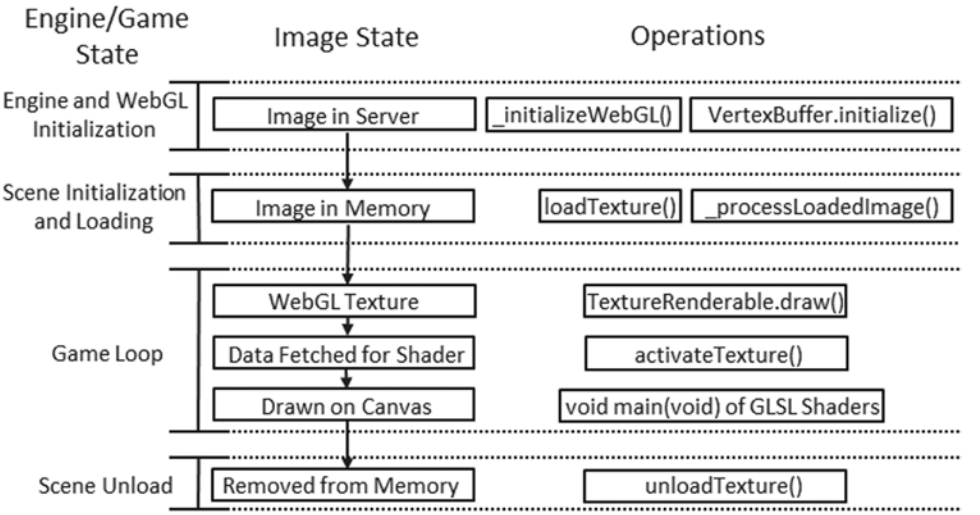

Figure 5-6. *Overview of the states of an image file and the corresponding WebGL texture*

Drawing with Sprite Sheets

As described earlier, a sprite sheet is an image that is composed of multiple lower-resolution images that individually represent different objects. Each of these individual images is referred to as a *sprite sheet element*. For example, Figure 5-7 shows a sprite sheet with 13 elements showing four different objects. Each of the top two rows contains five elements of the same object in different animated positions, and in the last row there are three elements of different objects: the character Dye, the portal minion, and the collector minion. The artist or software program that created the sprite sheet must communicate the pixel locations of each sprite element to the game developer, in much the same way as illustrated in Figure 5-7.

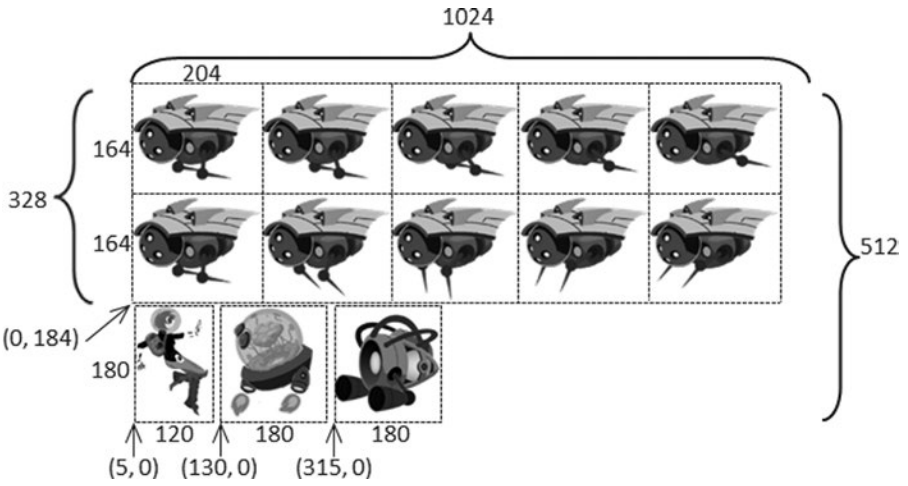

Figure 5-7. *Example sprite sheet: minion_sprite.png composed of lower-resolution images of different objects*

Sprite sheets are defined to optimize both memory and processing requirements. For example, recall that WebGL supports only textures that are defined by images with $2^x \times 2^y$ resolutions; this means a 256x128 ($2^8 \times 2^7$) image would be required in order to create a WebGL texture for the Dye character. In addition, if the 13 elements of Figure 5-7 were stored as separate images, 13 slow file system accesses would be required to load all the images, instead of one single system access to load the sprite sheet.

The key to working with a sprite sheet and the associated elements is to remember that the texture coordinate uv values are defined over the 0 to 1 normalized range regardless of the actual image resolution. For example, Figure 5-8 focuses on the uv values of the collector minion in Figure 5-7, the third row's rightmost element. The top, center, and bottom rows of Figure 5-8 show coordinate values of the portal element.

- *Pixel positions*: The lower-left corner is (315, 0), and the upper-right corner is (495, 180).
- *UV values*: The lower-left corner is (0.308, 0.0), and the upper-right corner is (0.483, 0.352).
- *Use in Model Space*: Texture mapping of the element is accomplished by associating the corresponding uv values with the xy values at each vertex position.

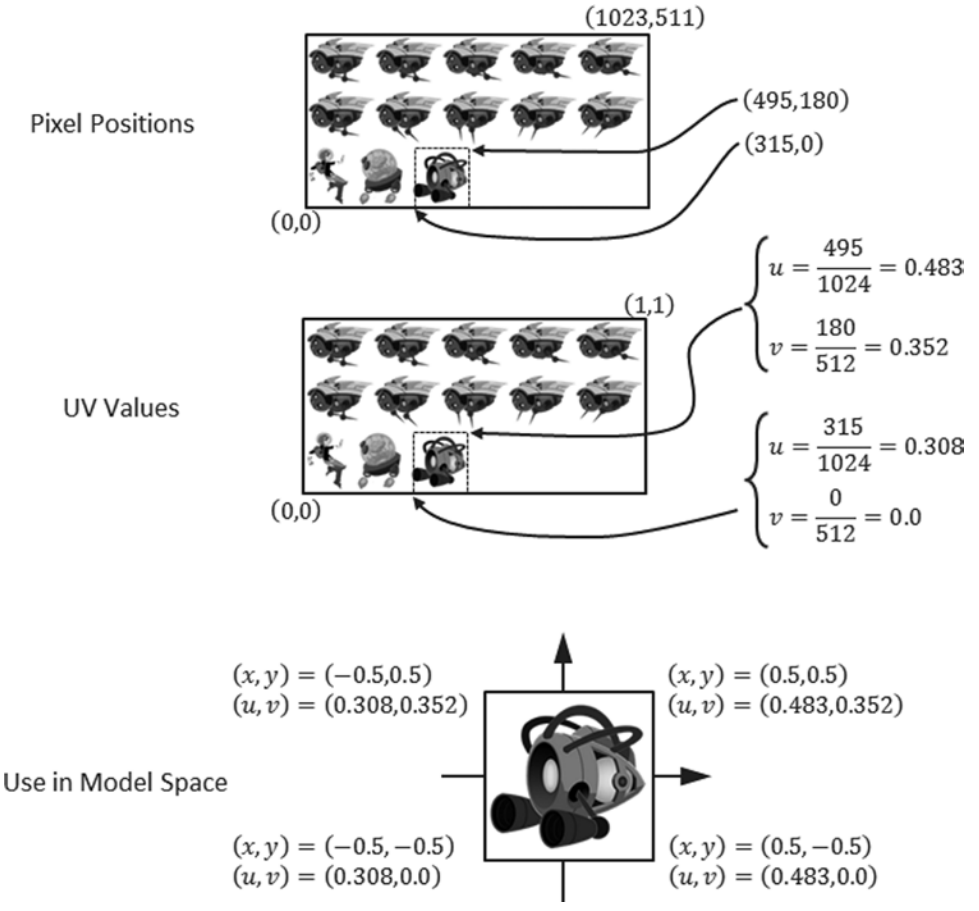

Figure 5-8. *A conversion of coordinates from pixel position to uv values and used for mapping on geometry*

The Sprite Shaders Project

This project demonstrates how to draw objects with sprite sheet elements by defining appropriate abstractions and classes. You can see an example of this project running in Figure 5-9. The source code to this project is defined in the `Chapter5/5.2.SpriteShaders` folder.

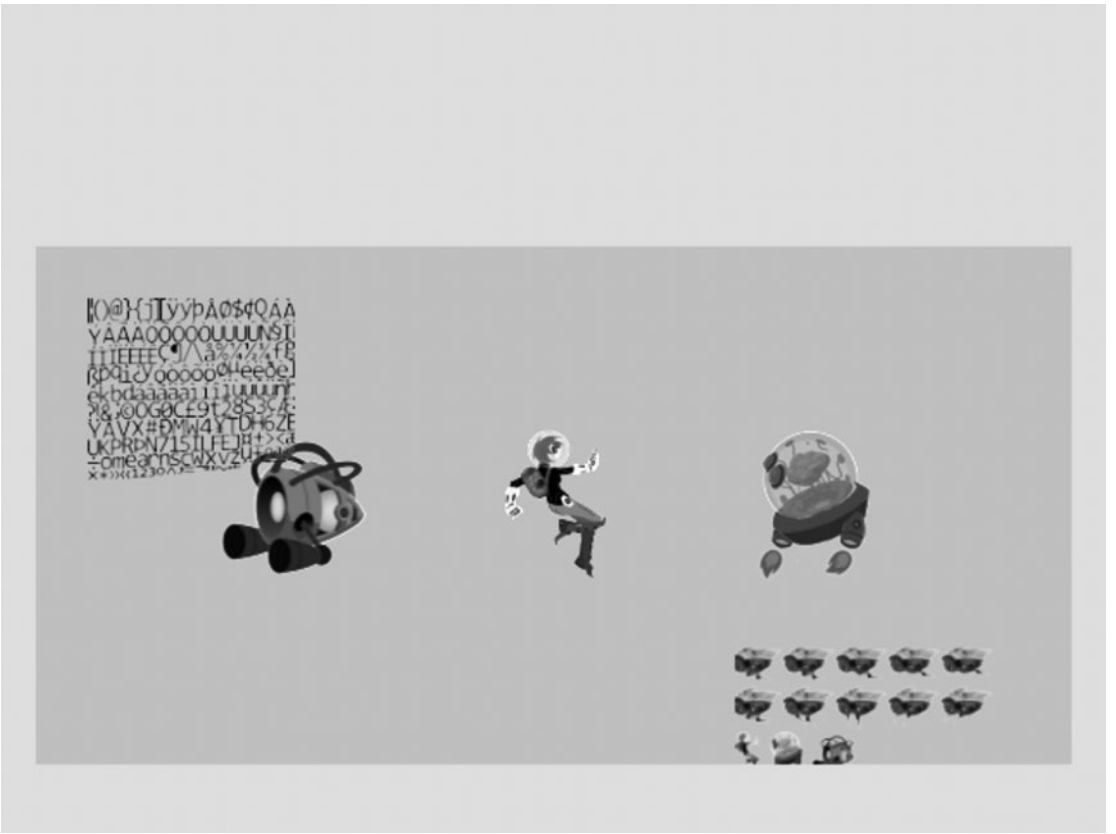

Figure 5-9. *Running the Sprite Shaders project*

The controls of the project are as follows:

- *Right arrow key*: Moves the Dye character (the hero) right and loops to the left boundary when the right boundary is reached
- *Left arrow key*: Moves the hero left and resets the position to the middle of the window when the left boundary is reached

The goals of the project are as follows:

- To gain a deeper understanding for texture coordinate
- To experience defining subregions within an image for texture mapping
- To draw squares by mapping from sprite sheet elements
- To prepare for working with sprite animation and bitmap fonts

You can find the following external resource files in the `assets` folder: `Consolas-72.png` and `minion_sprite.png`. Notice that `minion_sprite.png` is the image shown in Figure 5-7.

As depicted in Figure 5-5, one of the main advantages and shortcomings of the texture support defined in the previous section is that the texture coordinate accessed via the `getGLTexCoordRef()` function is statically defined in the `Engine_VertexBuffer.js` file. This is an advantage because in those cases where

an entire image is mapped onto a square, all instances of `TextureShader` objects can share the same default uv values. This is also a shortcoming because the static texture coordinate buffer does not allow support for working with different subregions of an image. In other words, the static texture coordinate values do not allow the support for working with sprite sheet elements. As illustrated in Figure 5-10, the example from this section overcomes this shortcoming by defining per-object texture coordinates in the `SpriteShader` and `SpriteRenderable` objects. Notice that there are no new GLSL shaders defined since their functionality remains the same.

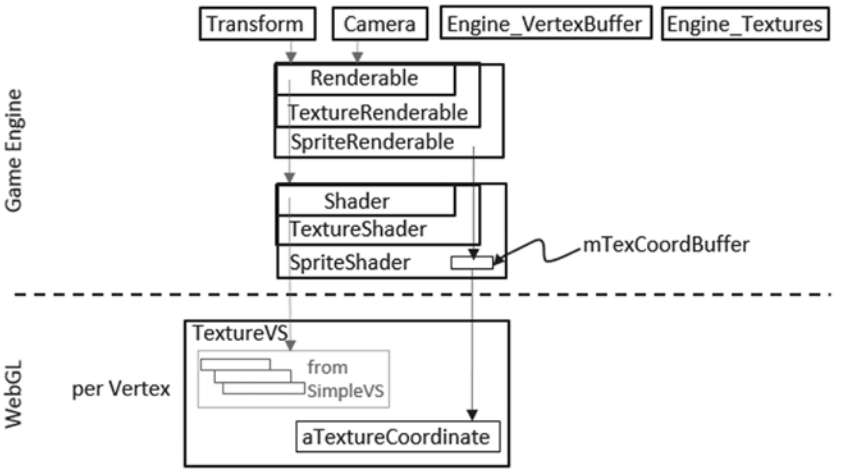

Figure 5-10. *Defining a texture coordinate buffer in the `SpriteShader`*

Interface GLSL Texture Shaders to the Engine with SpriteShader

Shaders supporting texture mapping with sprite sheet elements must be able to identify different unique subregions of an image. To support this functionality, you will implement the `SpriteShader` to define its own texture coordinates. Since this new shader extends the functionality of `TextureShader`, it is convenient to implement it as a subclass.

1. Create a new file in the `src/Engine/Shaders` folder and name it `SpriteShader.js`.
2. Define the `SpriteShader` constructor to derive it from `TextureShader`.

```
function SpriteShader(vertexShaderPath, fragmentShaderPath) {
    // Call super class constructor
    TextureShader.call(this, vertexShaderPath, fragmentShaderPath);

    this.mTexCoordBuffer = null; // gl buffer containing texture coordinate
    var initTexCoord = [
      1.0, 1.0,
      0.0, 1.0,
      1.0, 0.0,
      0.0, 0.0
    ];
```

```
        var gl = gEngine.Core.getGL();
        this.mTexCoordBuffer = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, this.mTexCoordBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(initTexCoord),
            gl.DYNAMIC_DRAW);
    }
    // get all the prototype functions from SimpleShader
    gEngine.Core.inheritPrototype(SpriteShader, TextureShader);
```

SpriteShader defines its own texture coordinate buffer in WebGL, and the reference to this buffer is kept by mTexCoordBuffer. Notice that in the previous code when creating this buffer in the gl.bufferData() function, the DYNAMIC_DRAW option is specified. This is compared with the STATIC_DRAW option used in Engine_VertexBuffer.js when defining the system default texture coordinate buffer. In this case, the dynamic option informs the WebGL graphics system that the content to this buffer will be subject to changes.

3. Define a function to set the WebGL texture coordinate buffer.

```
SpriteShader.prototype.setTextureCoordinate = function(texCoord) {
    var gl = gEngine.Core.getGL();
    gl.bindBuffer(gl.ARRAY_BUFFER, this.mTexCoordBuffer);
    gl.bufferSubData(gl.ARRAY_BUFFER, 0, new Float32Array(texCoord));
};
```

Note that texCoord is an array of eight floating-point numbers that specifies texture coordinate locations to the WebGL context. The format and content of this array are defined by the WebGL interface. texCoord must be a float array with eight floating-point numbers that identify four corners of a subregion in a Texture Space: top-right, top-left, bottom-right, and bottom-left corners. In your case, these should be the four corners of a sprite sheet element.

4. Override the shader activation function to enable the custom texture coordinate buffer for rendering.

```
SpriteShader.prototype.activateShader = function(pixelColor, vpMatrix) {
    // first call the super class's activate
    SimpleShader.prototype.activateShader.call(this, pixelColor, vpMatrix);

    // now binds the proper texture coordinate buffer
    var gl = gEngine.Core.getGL();
    gl.bindBuffer(gl.ARRAY_BUFFER, this.mTexCoordBuffer);
    gl.vertexAttribPointer(this.mShaderTextureCoordAttribute, 2, gl.FLOAT,
        false, 0, 0);
    gl.enableVertexAttribArray(this.mShaderTextureCoordAttribute);
};
```

Notice that the superclass activateShader() function is called to SimpleShader instead of TextureShader. This is to avoid TextureShader activating the system default texture coordinate buffer for rendering.

SpriteRenderable Object

Similar to `Renderable` objects (which are shaded with `SimpleShader`) and `TextureRenderable` objects (which are shaded with `TextureShader`), a corresponding `SpriteRenderable` object should be defined to represent objects that will be shaded with `SpriteShader`.

1. Create a new file in the `src/Engine/Renderables` folder and name it `SpriteRenderable.js`.

2. Define the `SpriteRenderable` constructor to derive it from `TextureRenderable`.

```
function SpriteRenderable(myTexture) {
    TextureRenderable.call(this, myTexture);
    Renderable.prototype._setShader.call(this, gEngine.DefaultResources.
        getSpriteShader());

    this.mTexLeft = 0.0;   // bounds of texture coord (0 is left, 1 is right)
    this.mTexRight = 1.0;  //
    this.mTexTop = 1.0;    // 1 is top and 0 is bottom of image
    this.mTexBottom = 0.0; //
}
gEngine.Core.inheritPrototype(SpriteRenderable, TextureRenderable);
```

A `SpriteRenderable` object extends the `TextureRenderable` object by defining the set of variables that identifies the texture coordinate bounds of a subregion within the Texture Space, the bounds of a sprite sheet element.

3. Define an enumerated data type with values that identify corresponding offset positions of a WebGL texture coordinate specification array.

```
// the expected texture cooridnate array is an array of 8 floats where:
//  [0] [1]: is u/v cooridnate of Top-Right
//  [2] [3]: is u/v coordinate of Top-Left
//  [4] [5]: is u/v coordinate of Bottom-Right
//  [6] [7]: is u/v coordinate of Bottom-Left
SpriteRenderable.eTexCoordArray = Object.freeze({
            eLeft: 2,
            eRight: 0,
            eTop: 1,
            eBottom: 5
});
```

■ **Note** e*Name* is an enumerated data type.

---

4. Define functions to allow the specification of a sprite sheet element's uv values in both texture coordinate space (normalized between 0 to 1) and with pixel positions (which will be converted to uv values).

```
SpriteRenderable.prototype.setElementUVCoordinate = function(left, right,
bottom, top) {
    this.mTexLeft = left;
    this.mTexRight = right;
    this.mTexBottom = bottom;
    this.mTexTop = top;
};

SpriteRenderable.prototype.setElementPixelPositions = function(left, right,
bottom, top) {
    var texInfo = gEngine.ResourceMap.retrieveAsset(this.mTexture);

    // entire image width, height
    var imageW = texInfo.mWidth;
    var imageH = texInfo.mHeight;

    this.mTexLeft = left / imageW;
    this.mTexRight = right / imageW;
    this.mTexBottom = bottom / imageH;
    this.mTexTop = top / imageH;
};
```

Note that the `setElementPixelPositions()` function converts from pixel to texture coordinates before storing the results with the corresponding instance variables.

5. Add a function to construct the texture coordinate specification array that is appropriate for passing the corresponding values to the WebGL context.

```
SpriteRenderable.prototype.getElementUVCoordinateArray = function() {
    return [
      this.mTexRight,  this.mTexTop,          // x,y of top-right
      this.mTexLeft,   this.mTexTop,
      this.mTexRight,  this.mTexBottom,
      this.mTexLeft,   this.mTexBottom
    ];
};
```

6. Override the `draw()` function to load the specific texture coordinates values into WebGL context before the actual drawing.

```
SpriteRenderable.prototype.draw = function(pixelColor, vpMatrix) {
    // set the current texture coordinate
    this.mShader.setTextureCoordinate(this.getElementUVCoordinateArray());
    TextureRenderable.prototype.draw.call(this, pixelColor, vpMatrix);
};
```

SpriteShader as a Default Resource

Similar to SimpleShader and TextureShader, the SpriteShader is a resource that can be shared. Thus, it should be added to the engine's DefaultResources.

1. In the Engine_DefaultResources.js file, add a variable for storing a SpriteShader.

```
var mSpriteShader = null;
```

2. Define a getter function for the SpriteShader.

```
var getSpriteShader = function() { return mSpriteShader; };
```

3. Modify the _createShaders function to also create the SpriteShader.

```
var _createShaders = function(callBackFunction) {
    gEngine.ResourceMap.setLoadCompleteCallback(null);
    mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);
    mTextureShader = new TextureShader(kTextureVS, kTextureFS);
    mSpriteShader =  new SpriteShader(kTextureVS, kTextureFS);
    callBackFunction();
};
```

Notice that the SpriteShader actually wraps over the existing GLSL shaders defined in the TextureVS.glsl and TextureFS.glsl files. From the perspective of WebGL, the functionality of drawing with texture remains the same; the only difference with SpriteShader is that the texture's coordinate values are now programmable.

4. Remember to update the public interface.

```
var mPublic = {
    initialize: initialize,
    getConstColorShader: getConstColorShader,
    getTextureShader: getTextureShader,
    getSpriteShader: getSpriteShader
};
```

Testing the SpriteRenderable

There are two important functionalities of working with sprite elements and texture coordinates that should be tested: the proper extraction, drawing, and controlling of a sprite sheet element as an object; and the changing and controlling of uv coordinate on an object. For proper testing of the added functionality, you must modify the MyGame.js file.

1. The constructing, loading, unloading, and drawing of MyGame are similar to previous examples, so the details will not be repeated here. Please refer to the source code in the src/MyGame folder for details.

2. In the `initialize()` function, do the following:

```
MyGame.prototype.initialize = function() {
    // Step A: set up the cameras
    this.mCamera = new Camera(
            vec2.fromValues(20, 60),   // position of the camera
            20,                        // width of camera
            [20, 40, 600, 300]         // viewport (orgX, orgY, width, height)
            );
    this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);
            // sets the background to gray

    // Step B: Create the support objects
    this.mPortal = new SpriteRenderable(this.kMinionSprite);
    this.mPortal.setColor([1, 0, 0, 0.2]);  // tints red
    this.mPortal.getXform().setPosition(25, 60);
    this.mPortal.getXform().setSize(3, 3);
    this.mPortal.setElementPixelPositions(130, 310, 0, 180);

    this.mCollector = new SpriteRenderable(this.kMinionSprite);
    this.mCollector.setColor([0, 0, 0, 0]);  // No tinting
    this.mCollector.getXform().setPosition(15, 60);
    this.mCollector.getXform().setSize(3, 3);
    this.mCollector.setElementPixelPositions(315, 495, 0, 180);

    // Step C: Create the font and minion images using sprite
    this.mFontImage = new SpriteRenderable(this.kFontImage);
    this.mFontImage.setColor([1, 1, 1, 0]);
    this.mFontImage.getXform().setPosition(13, 62);
    this.mFontImage.getXform().setSize(4, 4);

    this.mMinion= new SpriteRenderable(this.kMinionSprite);
    this.mMinion.setColor([1, 1, 1, 0]);
    this.mMinion.getXform().setPosition(26, 56);
    this.mMinion.getXform().setSize(5, 2.5);

    // Step D: Create the hero object with texture from lower-left corner
    this.mHero = new SpriteRenderable(this.kMinionSprite);
    this.mHero.setColor([1, 1, 1, 0]);
    this.mHero.getXform().setPosition(20, 60);
    this.mHero.getXform().setSize(2, 3);
    this.mHero.setElementPixelPositions(0, 120, 0, 180);
};
```

   a. After the camera is set up, in step B, notice that both `mPortal` and `mCollector` are created based on the same image, `kMinionSprite`, with the respective `setElementPixelPositions()` calls to specify the actual sprite element to use for rendering.

b. Step C creates two additional `SpriteRenderable` objects: `mFontImage` and `mMinion`. The sprite element uv coordinate settings are left to the defaults where the texture image will cover the entire geometry.

c. Similar to step B, step C creates the hero character as a `SpriteRenderable` object based on the same `kMinionSprite` image. The sprite sheet element that corresponds to the hero is identified with the `setElementPixelPositions()` call.

Notice that in this example, four of the five `SpriteRenderable` objects created are based on the same `kMinionSprite` image.

3. The `update()` function is modified to support the controlling of the hero object and changes to the uv values.

```
MyGame.prototype.update = function() {
    // let's only allow the movement of hero,
    // and if hero moves too far off, this level ends, we will
    // load the next level
    var deltaX = 0.05;
    var xform = this.mHero.getXform();

    // Support hero movements
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Right)) {
        xform.incXPosBy(deltaX);
        if (xform.getXPos() > 30)  // this is the right-bound of the window
            xform.setPosition(12, 60);
    }

    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Left)) {
        xform.incXPosBy(-deltaX);
        if (xform.getXPos() < 11) {  // this is the left-bound of the window
            xform.setXPos(20);
        }
    }

    // continously change texture tinting
    var c = this.mPortal.getColor();
    var ca = c[3] + deltaX;
    if (ca > 1) ca = 0;
    c[3] = ca;

    // New update code for changing the sub-texture regions being shown
    var deltaT = 0.001;

    // The font image:
    // zoom into the texture by updating texture coordinate
    // For font: zoom to the upper left corner by changing bottom right
    var texCoord = this.mFontImage.getElementUVCoordinateArray();
            // The 8 elements:
            //      mTexRight,  mTexTop,          // x,y of top-right
            //      mTexLeft,   mTexTop,
```

```
            //      mTexRight,  mTexBottom,
            //      mTexLeft,   mTexBottom
        var b = texCoord[SpriteRenderable.eTexCoordArray.eBottom] + deltaT;
        var r = texCoord[SpriteRenderable.eTexCoordArray.eRight] - deltaT;

        if (b > 1.0)  b = 0;
        if (r < 0)   r = 1.0;
        this.mFontImage.setElementUVCoordinate(
               texCoord[SpriteRenderable.eTexCoordArray.eLeft], r,
               b, texCoord[SpriteRenderable.eTexCoordArray.eTop]);

        // The minion image:
        // For minion: zoom to the bottom right corner by changing top left
        var texCoord = this.mMinion.getElementUVCoordinateArray();
            // The 8 elements:
            //      mTexRight,  mTexTop,          // x,y of top-right
            //      mTexLeft,   mTexTop,
            //      mTexRight,  mTexBottom,
            //      mTexLeft,   mTexBottom
        var t = texCoord[SpriteRenderable.eTexCoordArray.eTop] - deltaT;
        var l = texCoord[SpriteRenderable.eTexCoordArray.eLeft] + deltaT;

        if (l > 0.5) l = 0;
        if (t < 0.5) t = 1.0;

        this.mMinion.setElementUVCoordinate(
               l, texCoord[SpriteRenderable.eTexCoordArray.eRight],
               texCoord[SpriteRenderable.eTexCoordArray.eBottom], t);
};
```

a. Observe that the keyboard control and the drawing of the hero object are identical to previous projects.

b. Notice the calls to `setElementUVCoordinate()` for `mFontImage` and `mMinion`. These calls continuously decrease and reset the V values that correspond to the bottom, the U values that correspond to the right for `mFontImage`, the V values that correspond to the top, and the U values that correspond to the left for `mMinion`. The end results are the continuous changing of texture and the appearance of a zooming animation on these two objects

Sprite Animations

In games, you often want to create animations that reflect the movements or actions of your characters. In the previous chapter, you learned about moving the geometries of these objects with transformation operators. However, as you have observed when controlling the hero character in the previous example, if the textures on these objects do not change in ways that correspond to the control, the interaction conveys the sensation of moving a static image rather than setting a character in motion. What is needed is the ability to create the illusion of animations on geometries when desired.

In the previous example, you observed from the `mFontImage` and `mMinion` objects that the appearance of an animation can be created by constantly changing the uv values on a texture-mapped geometry. As discussed at the beginning of this chapter, one way to control this type of animation is by working with an animated sprite sheet.

Overview of Animated Sprite Sheets

Recall that an animated sprite sheet is a sprite sheet that contains the sequence of images of an object in an animation, typically in one or more rows and columns. For example, in Figure 5-11 you can see a 2x5 animated sprite sheet that contains two separate animations organized in two rows. The animations depict an object retracting its spikes toward the right in the top row and extending them toward the left in the bottom row. In this example, the animations are separated into separate rows; however, this is not always the case. The organization of a sprite sheet and the details of element pixel locations are generally handled by its creator and must be explicitly communicated to the game developer for use in games.

Figure 5-11. *An animated sprite sheet organized into two rows representing two animated sequences of the same object*

Figure 5-12 shows that to achieve the animated effect of an object retracting its spikes toward the right, as depicted by the top row of Figure 5-11, you map the elements from the left to the right in the sequence 1, 2, 3, 4, 5. When these images are mapped onto the same geometry, sequenced, and looped in an appropriate rate, it conveys the sense that the object is indeed repeating the action of retracting its spikes. Alternatively, if the sequence is reversed where the elements are mapped in the right-to-left sequence, it would create the animation that corresponds to the object extending the spikes toward the left. Last, it is also possible to map the sequence in a swing loop from left to right and then back from right to left. In this case, the animation would correspond to the object going through the motion of retracting and extending its spikes continuously.

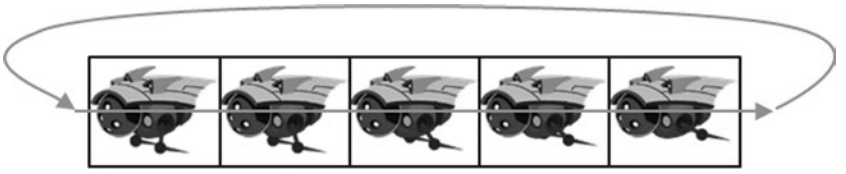

Figure 5-12. *A sprite animation sequence that loops*

With a firm background in sprite sheet animation and understanding the different ways of generating sprite animation, you can now tackle the implementation.

The Sprite Animation Project

This project demonstrates how to work with animated sprite sheet and generate continuous sprite animations. You can see an example of this project running in Figure 5-13. The project scene contains the objects from the previous scene plus two animated objects. The source code to this project is defined in the `Chapter5/5.3.SpriteAnimation` folder.

Figure 5-13. *Running the Sprite Animation project*

The controls of the project are as follows:

- *Right arrow key*: Moves the hero right; when crossing the right boundary, the hero is wrapped back to the left boundary
- *Left arrow key*: Opposite of the right arrow key
- *Number 1 key*: Animates by showing sprite elements continuously from right to left
- *Number 2 key*: Animates by showing sprite elements moving back and forth continuously from left to right and right to left
- *Number 3 key*: Animates by showing sprite elements continuously from left to right
- *Number 4 key*: Increases the animation speed
- *Number 5 key*: Decreases the animation speed

The goals of the project are as follows:

- To gain a deeper understanding of animated sprite sheets
- To experience the creation of sprite animations
- To define abstractions for implementing sprite animations

You can find the same files as in the previous project in the `assets` folder.

SpriteAnimateRenderable Object

Sprite animation can be implemented by strategically controlling the uv values of a `SpriteRenderable` to display the appropriate sprite element at desired time periods. For this reason, only a single class, `SpriteAnimateRenderable`, needs to be defined to support sprite animations.

For simplicity and ease of understanding, the following implementation assumes that all sprite elements associated with an animation are always organized along the same row. Animated sprite elements organized along a column are not supported. For example, in Figure 5-11, the rightward retraction and leftward extension movements of the spikes are each organized along a row; neither spans more than one single row, and neither is organized along a column.

1. Create a new file in the `src/Engine/Renderables` folder and name it `SpriteAnimateRenderable.js`.

2. Define an enumerated data type that describes the different ways to animate with a sprite sheet.

```
// Assumption: first sprite in an animation is always the left-most element.
SpriteAnimateRenderable.eAnimationType = Object.freeze({
    eAnimateRight: 0,  // Animate from left to right, then restart to left
    eAnimateLeft: 1,   // Animate from right to left, then restart to right
    eAnimateSwing: 2   // Animate first left to right, then animates backwards
});
```

   `eAnimationType` defines three modes for animation.

   a. `eAnimateRight` starts at the leftmost element and animates by iterating toward the right along the same row. When the last element is reached, the animation continues by starting from the leftmost element again.

   b. `eAnimateLeft` is the reverse of `eAnimateRight`; it starts from the right, animates toward the left, and continues by starting from the rightmost element after reaching the leftmost element.

   c. `eAnimateSwing` is a continuous loop from left to right and then from right to left.

3. Define the `SpriteAnimateRenderable` constructor to derive it from `SpriteRenderable`.

```
function SpriteAnimateRenderable(myTexture) {
    SpriteRenderable.call(this, myTexture);
    Renderable.prototype._setShader.call(this, gEngine.DefaultResources.
        getSpriteShader());
```

```
        // All coordinates are in texture coordinate (UV between 0 to 1)

        // Information on the sprite element
        this.mFirstElmLeft = 0.0; // 0.0 is left corner of image
        this.mElmTop = 1.0;  // 1.0 is top corner of image
        this.mElmWidth = 1.0;  // default sprite element size is the entire image
        this.mElmHeight = 1.0;
        this.mWidthPadding = 0.0;
        this.mNumElems = 1;   // number of elements in an animation

        // per animation settings
        this.mAnimationType = SpriteAnimateRenderable.eAnimationType.eAnimateRight;
        this.mUpdateInterval = 1;   // how often to advance

        // current animation state
        this.mCurrentAnimAdvance = -1;
        this.mCurrentElm = 0;

        this._initAnimation();
    }
    gEngine.Core.inheritPrototype(SpriteAnimateRenderable, SpriteRenderable);
```

The `SpriteAnimateRenderable` constructor defines three sets of variables:

a. The first set, including `mFirstElmLeft`, `mElmTop`, and so on, defines the location and dimensions of each sprite element and the number of elements in the animation. This information can be used to accurately compute the texture coordinates for each sprite element when the elements are ordered by rows and columns. Note that all coordinates are in Texture Space (0 to 1).

b. The second set stores information on how to animate: the `mAnimationType` of left, right, or swing; and how many `mUpdateInterval` time to wait before advancing to the next sprite element to control the speed of the animation. This information can be changed during runtime to reverse a character's movement, loop the character's movement, or speed up or slow down the movement.

c. The third set, `mCurrentAnimAdvance` and `mCurrentElm`, describes the current animation state, which frame, and the direction of the animation. Both of these variables are in units of element counts, are not accessible by the game programmer, and are used internally to compute the next sprite element for display.

The `_initAnimation()` function computes the values of `mCurrentAnimAdvance` and `mCurrentElm` to initialize an animation sequence.

4. Define a function to set the animation type.

```
SpriteAnimateRenderable.prototype.setAnimationType = function(animationType) {
    this.mAnimationType = animationType;
    this.mCurrentAnimAdvance = -1;
    this.mCurrentElm = 0;
    this._initAnimation();
};
```

Note that the animation is always reset to start from the beginning when the animation type (left, right, or swing) is changed.

5. Define the `_initAnimation()` function to compute the proper vales for `mCurrentAnimAdance` and `mCurrentElm` according to the current animation type.

```
SpriteAnimateRenderable.prototype._initAnimation = function() {
    // Currently running animation
    this.mCurrentTick = 0;
    switch (this.mAnimationType) {

        case SpriteAnimateRenderable.eAnimationType.eAnimateRight:
            this.mCurrentElm = 0;
            this.mCurrentAnimAdvance = 1; // either 1 or -1
            break;

        case SpriteAnimateRenderable.eAnimationType.eAnimateSwing:
            this.mCurrentAnimAdvance = -1 * this.mCurrentAnimAdvance;
            this.mCurrentElm += 2*this.mCurrentAnimAdvance;
            break;

        case SpriteAnimateRenderable.eAnimationType.eAnimateLeft:
            this.mCurrentElm = this.mNumElems - 1;
            this.mCurrentAnimAdvance = -1; // either 1 or -1
            break;
    }
    this._setSpriteElement();
};
```

The previous code shows that `mCurrentElm` is the number of elements offset from the leftmost element, and `mCurrentAnimAdvance` records whether the `mCurrentElm` offset should be incremented (for rightward animation) or decremented (for leftward animation) during each update.

The `_setSpriteElement()` is called to set the uv values that correspond to the currently identified sprite element for displaying on the geometry (by the superclass, `SpriteRenderable`).

6. Define the `_setSpriteElement()` function to compute and load the uv values of the currently identified sprite element for rendering.

```
SpriteAnimateRenderable.prototype._setSpriteElement = function() {
    var left = this.mFirstElmLeft + (this.mCurrentElm * (this.mElmWidth +
    this.mWidthPadding));

    SpriteRenderable.prototype.setElementUVCoordinate.call(this, left,
                                                left+this.mElmWidth,
                                                this.mElmTop-this.mElmHeight,
                                                this.mElmTop);
};
```

The variable `left` is the left u value of `mCurrentElm`. Based on this value, the right, bottom, and top uv values can be derived and set to `SpriteRenderable`.

7. Define a function to allow game programmers to specify a sprite animation.

```
// Always set the right-most element to be the first
SpriteAnimateRenderable.prototype.setSpriteSequence = function(
        topPixel,          // offset from top-left
        rightPixel,        // offset from top-left
        elmWidthInPixel,
        elmHeightInPixel,
        numElements,       // number of elements in sequence
        wPaddingInPixel    // left/right padding
        )
{
    var texInfo = gEngine.ResourceMap.retrieveAsset(this.mTexture);
    // entire image width, height
    var imageW = texInfo.mWidth;
    var imageH = texInfo.mHeight;

    this.mNumElems = numElements;   // number of elements in animation
    this.mFirstElmLeft = rightPixel / imageW;
    this.mElmTop = topPixel / imageH;
    this.mElmWidth = elmWidthInPixel / imageW;
    this.mElmHeight = elmHeightInPixel / imageH;
    this.mWidthPadding = wPaddingInPixel / imageW;
    this._initAnimation();
};
```

The inputs of the `setSpriteSequence()` function are in pixels and are converted to texture coordinates by dividing by the width and height of the image.

8. Implement functions to change animation speed, either directly or by an offset.

```
SpriteAnimateRenderable.prototype.setAnimationSpeed = function(tickInterval) {
    // number of update calls before advancing animation
    this.mUpdateInterval = tickInterval;   // how often to advance
};

SpriteAnimateRenderable.prototype.incAnimationSpeed = function(deltaInterval) {
    // number of update calls before advancing animation
    this.mUpdateInterval += deltaInterval;   // how often to advance
};
```

9. Finally, define a function to advance the animation for each game loop update.

```
SpriteAnimateRenderable.prototype.updateAnimation = function() {
    this.mCurrentTick++;
    if (this.mCurrentTick >= this.mUpdateInterval) {
        this.mCurrentTick = 0;
        this.mCurrentElm += this.mCurrentAnimAdvance;
        if ((this.mCurrentElm >= 0) && (this.mCurrentElm < this.mNumElems))
            this._setSpriteElement();
```

```
            else
                this._initAnimation();
        }
    };
```

Each time the `updateAnimation()` function is called, the `mCurrentTick` counter is incremented, and when the number of ticks reaches the `mUpdateInterval` value, the animation is re-initialized by the `_initAnimation()` function. It is important to note that the time unit for controlling the animation is the number of times the `updateAnimation()` function is called and not the real-world elapsed time. Recall that the engine `GameLoop _runLoop()` function ensures systemwide updates to occur at `kMPF` intervals even when frame rate lags. The game engine architecture ensures the `updateAnimation()` function calls are `kMPF` millisecond apart.

Testing Sprite Animation

The test cases for the `SpriteAnimateRenderable` object must demonstrate how the forward, reverse, and swing modes of animation and the animation speed are under the programmer's control. The `MyGame` object is modified to accomplish these purposes.

1. The constructing, loading, unloading, and drawing of `MyGame` are similar to the previous example, so the details will not be repeated here. Please refer to the source code in the `src/MyGame` folder for details.

2. In the `initialize()` function, add code to create and initialize the `SpriteAnimateRenderable` objects.

```
MyGame.prototype.initialize = function() {
    // ... Identical to previous code ...

    // The right minion
    this.mRightMinion= new SpriteAnimateRenderable(this.kMinionSprite);
    this.mRightMinion.setColor([1, 1, 1, 0]);
    this.mRightMinion.getXform().setPosition(26, 56.5);
    this.mRightMinion.getXform().setSize(4, 3.2);
    this.mRightMinion.setSpriteSequence(
        512, 0,     // first element position: top-right, 512 is top, 0 is right
        204,164,    // width x height in pixels
        5,          // number of elements in this sequence
        0);         // horizontal padding in between
    this.mRightMinion.setAnimationType(SpriteAnimateRenderable.eAnimationType.
        eAnimateRight);
    this.mRightMinion.setAnimationSpeed(50);

    // the left minion
    this.mLeftMinion= new SpriteAnimateRenderable(this.kMinionSprite);
    this.mLeftMinion.setColor([1, 1, 1, 0]);
    this.mLeftMinion.getXform().setPosition(15, 56.5);
    this.mLeftMinion.getXform().setSize(4, 3.2);
```

```
        this.mLeftMinion.setSpriteSequence(
            348, 0,     // first element: top-right, 164 from 512 is top, 0 is right
            204,164,    // widthxheight in pixels
            5,          // number of elements in this sequence
            0);         // horizontal padding in between
        this.mLeftMinion.setAnimationType(SpriteAnimateRenderable.eAnimationType.
            eAnimateRight);
        this.mLeftMinion.setAnimationSpeed(50);

        // ... Identical to previous code ...
    }
```

The SpriteAnimateRenderable objects are created in similar ways as SpriteRenderable objects with a sprite sheet as the texture parameter. Additionally, in this case, it is essential to call the setSpriteSequence() function to identify the elements involved in the animation including the location, dimension, and total number of elements.

3. The update() function must invoke the SpriteAnimateRenderable object's updateAnimation() function to advance the sprite animation.

```
MyGame.prototype.update = function() {
    // ... Identical to previous code ...

    // remember to update the minion's animation
    this.mRightMinion.updateAnimation();
    this.mLeftMinion.updateAnimation();

    // Animate left on the sprite sheet
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.One)) {
        this.mRightMinion.setAnimationType(
            SpriteAnimateRenderable.eAnimationType.eAnimateLeft);
        this.mLeftMinion.setAnimationType(
            SpriteAnimateRenderable.eAnimationType.eAnimateLeft);
    }

    // swing animation
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Two)) {
        this.mRightMinion.setAnimationType(
            SpriteAnimateRenderable.eAnimationType.eAnimateSwing);
        this.mLeftMinion.setAnimationType(
            SpriteAnimateRenderable.eAnimationType.eAnimateSwing);
    }

    // Animate right on the sprite sheet
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Three)) {
        this.mRightMinion.setAnimationType(
            SpriteAnimateRenderable.eAnimationType.eAnimateRight);
        this.mLeftMinion.setAnimationType(
            SpriteAnimateRenderable.eAnimationType.eAnimateRight);
    }
```

```
            // decrease the duration of showing sprite elements,
            // speeding up the animation
            if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Four)) {
                this.mRightMinion.incAnimationSpeed(-2);
                this.mLeftMinion.incAnimationSpeed(-2);
            }

            // increase the duration of showing sprite elements,
            // slowing down the animation
            if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Five)) {
                this.mRightMinion.incAnimationSpeed(2);
                this.mLeftMinion.incAnimationSpeed(2);
            }
        };
```

The keys 1, 2, and 3 change the animation type, and keys 4 and 5 change the animation speed. Note that the limit of the animation speed is the update rate of the game loop.

Fonts and Drawing of Text

A valuable tool that many games use for a variety of tasks is text output. Drawing of text messages is an efficient way to communicate to the user as well as you, the developer. For example, text messages can be used to communicate the game's story, the player's score, or debugging information during development. Unfortunately, WebGL does not support the drawing of text. This section briefly introduces bitmap fonts and introduces `FontRenderable` objects to support the drawing of texts.

Bitmap Fonts

A font must be defined such that individual characters can be extracted for the drawing of text messages. A bitmap font, as the name implies, is a simple map describing which bit (or pixel) must be switched on to represent characters in the font. Combining all characters of a bitmap font into a single image and defining an accompanied decoding description document provide a straightforward solution for drawing text output. For example, Figure 5-14 shows a bitmap font sprite where all the defined characters are tightly organized into the same image. Figure 5-15 is a snippet of the accompanying decoding description in XML format.

Figure 5-14. *An example bitmap font sprite image*

```
<?xml version="1.0"?>
<font>
  <info face="Consolas" size="24" bold="0" italic="0" charset="" unicode="1" stretchH="100" smooth="1" aa="1"
padding="0,0,0,0" spacing="1,1" outline="0"/>
  <common lineHeight="24" base="19" scaleW="256" scaleH="128" pages="1" packed="0" alphaChnl="0" redChnl="3"
greenChnl="3" blueChnl="3"/>
  <pages>
    <page id="0" file="Consolas-24-NoKerning_0.png" />
  </pages>
  <chars count="193">
    <char id="0" x="252" y="35" width="3" height="1" xoffset="-1" yoffset="23" xadvance="11" page="0" chnl="15" />
    <char id="13" x="254" y="0" width="0" height="1" xoffset="0" yoffset="23" xadvance="0" page="0" chnl="15" />
    <char id="32" x="17" y="38" width="3" height="1" xoffset="-1" yoffset="23" xadvance="11" page="0" chnl="15" />
```

Figure 5-15. *A snippet of the XML file with the decoding information for the bitmap font image shown in Figure 5-14*

Notice that the decoding information as shown in Figure 5-15 uniquely defines the uv coordinate positions for each character in the image, as shown in Figure 5-14. In this way, the texture mapping of individual characters from a bitmap font sprite image can be performed in a straightforward manner by the `SpriteRenderable` objects.

■ **Note** There are many bitmap font file formats. The format used in this book is the AngleCode BMFont-compatible font in XML form. BMFont is an open source software that converts vector fonts, such as TrueType and OpenType, into bitmap fonts. See `http://www.angelcode.com/products/bmfont/` for more information.

The Font Support Project

This project demonstrates how to draw text from a bitmap font using the `SpriteRenderable` object. You can see an example of this project running in Figure 5-16. This project consists of two scenes. The first is a simple extension from the previous project with sample text output from different bitmap fonts. You can control the hero's position with the arrow keys. If the hero exits the window from the left boundary, the second scene will be invoked. The second scene simply shows the "Game Over!" message and stops the game loop. The source code to this project is defined in the `Chapter5/5.4.FontSupport` folder.

Figure 5-16. *Running the Font Support project*

The controls of the project are as follows:

- *Number keys 0, 1, 2, and 3*: Selects the Consolas, 16, 24, 32, or 72 fonts, respectively, for size modification
- *Up/down key while holding down X/Y key*: Increases or decreases (arrow keys) the width (X key) or the height (Y key) of the selected font
- *Left arrow key*: Moves the hero left; if the hero exits the screen from the left boundary, the `GameOver` level is invoked and the game ends
- *Right arrow key*: Moves the hero right; when crossing the right boundary, the hero is wrapped back to the left boundary

The goals of the project are as follows:

- To gain a basic understanding of drawing text strings in a game
- To understand what bitmap fonts are
- To implement text drawing support in your game engine

You can find the following external resource files in the `assets` folder: `Consolas-72.png` and `minion_sprite.png`. In the `assets/fonts` folder are the bitmap font sprite image files and the associated XML files that contain the decoding information: `Consolas-16.fnt`, `Consolas-16.png`, `Consolas-24.fnt`, `Consolas-24.png`, `Consolas-32.fnt`, `Consolas-32.png`, `Consolas-72.fnt`, `Consolas-72.png`, `Segment7-96.fnt`, `Segment7-96.png`, `system-default-font.fnt`, and `system-default-font.png`.

Notice that the `.fnt` and `.png` files are paired. The former contains decoding information for the latter. These file pairs must be included in the same folder for the engine to load the font properly. `system-default-font` is the default font for the game engine, and it is assumed that this font is always present in the `asset/fonts` folder.

■ **Note** The actions of parsing, decoding, and extracting of character information from the `.fnt` files are independent from the foundational operations of a game engine. For this reason, the details of these operations are not presented. If you are interested, you should consult the source code.

Loading and Storing Fonts in the Engine

Loading font files is special because fonts are defined in pairs: the `.fnt` file that contains decoding information and the corresponding `.png` sprite image file. However, since the `.fnt` file is an XML file and the `.png` file is a simple texture image, the actual loading of these two files is already supported by the existing engine functionality. The details of loading and storing fonts in the engine will be hidden by a new engine component.

1. Create a new file in the `src/Engine/Resources` folder and name it `Engine_Fonts.js`.
2. Before implementing the `gEngine_Fonts` component, first define an object for storing pixel location and display information associated with the characters.

```
function CharacterInfo() {
  // in texture coordinate (0 to 1) maps to the entier image
  this.mTexCoordLeft = 0;
  this.mTexCoordRight = 1;
  this.mTexCoordBottom = 0;
  this.mTexCoordTop = 0;
```

```
        // nominal character size, 1 is "standard width/height" of a char
        this.mCharWidth = 1;
        this.mCharHeight = 1;
        this.mCharWidthOffset = 0;
        this.mCharHeightOffset = 0;

        // reference of char width/height ration
        this.mCharAspectRatio = 1;
    }
```

The uv coordinate and character appearance information can be computed based on the contents from the `.fnt` file.

3. Following the pattern of previous engine components, implement the `Fonts` engine component.

```
var gEngine = gEngine || { };

gEngine.Fonts = (function() {
    var mPublic = { };
    return mPublic;
}());
```

4. Define a function to load font files from a given path.

```
var loadFont = function(fontName) {
    if (!(gEngine.ResourceMap.isAssetLoaded(fontName))) {
        var fontInfoSourceString = fontName + ".fnt";
        var textureSourceString = fontName + ".png";

        // register an entry in the map
        gEngine.ResourceMap.asyncLoadRequested(fontName);

        gEngine.Textures.loadTexture(textureSourceString);
        gEngine.TextFileLoader.loadTextFile(fontInfoSourceString,
                        gEngine.TextFileLoader.eTextFileType.eXMLFile,
                        _storeLoadedFont);
    } else {
        gEngine.ResourceMap.incAssetRefCount(fontName);
    }
};
```

a. `fontName` is a path to the font files but without any file extensions. For example, `assets/fonts/system-default-font` is the string that identifies the two associated `.fnt` and `.png` files.

b. Two file load operations are actually invoked: one to load the `.fnt` as a text file and the second to load the `.png` as a texture image file.

c. The last parameter of the `loadTextFile()` function specifies that the `_storeLoadedFont()` function should be invoked when the load operation is completed.

5. Define the `_storeLoadedFont()` function to store the loaded results in the `ResourceMap`.

```
var _storeLoadedFont = function(fontInfoSourceString) {
    var fontName = fontInfoSourceString.slice(0, -4);  // trims .fnt extension
    var fontInfo = gEngine.ResourceMap.retrieveAsset(fontInfoSourceString);
    fontInfo.FontImage = fontName + ".png";
    gEngine.ResourceMap.ayncLoadCompleted(fontName, fontInfo);
};
```

Recall that the `ResourceMap.retrieveAsset()` function returns an `MapEntry` object that contains the reference to the loaded asset, the XML file content in this case, and a reference count. The previous code creates an additional property, `FontImage`, on the returned `MapEntry` to record the associated sprite image for the font.

6. Define a function to unload a font and release the associated memory.

```
var unloadFont = function(fontName) {
    gEngine.ResourceMap.unloadAsset(fontName);
    if (!(gEngine.ResourceMap.isAssetLoaded(fontName))) {
        var fontInfoSourceString = fontName + ".fnt";
        var textureSourceString = fontName + ".png";

        gEngine.Textures.unloadTexture(textureSourceString);
        gEngine.TextFileLoader.unloadTextFile(fontInfoSourceString);
    };
};
```

It is important to remember to unload both the text `.fnt` file and the `.png` texture image file.

7. Define a function to compute `CharacterInfo` based on the information presented in the `.fnt` file.

```
var getCharInfo = function(fontName, aChar) {
    // ... details omitted for lack of relevancy

    returnInfo = new CharacterInfo();

    // computes and fills in the contents of CharacterInfo
    // ... details omitted for lack of relevancy

    return returnInfo;
};
```

Details of decoding and extracting information for the given character are omitted because they are unrelated to the rest of the game engine implementation.

■ **Note** For details of the `.fnt` format information, please refer to `http://www.angelcode.com/products/bmfont/doc/file_format.html`.

---

8. Finally, remember to add the public functions to the public interface.

```
var mPublic = {
    loadFont: loadFont,
    unloadFont: unloadFont,
    getCharInfo: getCharInfo
};
return mPublic;
```

Defining a FontRenderable Object to Draw Texts

The defined `gEngine_Fonts` component is capable of loading font files and extracting per-character uv coordinate and appearance information. With this functionality, the drawing of a text string can be accomplished by identifying each character in the string, extracting the corresponding texture mapping information from the `gEngine_Fonts` component, and rendering the character using the `SpriteRenderable` object. The `FontRenderable` object will be defined to accomplish this.

1. Create a new file in the `src/Engine/Renderables` folder and name it `FontRenderable.js`.

2. Implement a constructor for `FontRenderable` that accepts a string as its parameter.

```
function FontRenderable(aString) {
    this.mFont = gEngine.DefaultResources.getDefaultFont();
    this.mOneChar = new SpriteRenderable(this.mFont + ".png");
    this.mXform = new Transform(); // transform that moves this object around
    this.mText = aString;
}
```

   a. The `aString` variable is the message to be drawn.

   b. Notice that `FontRenderable` objects do not customize the behaviors of `SpriteRenderable` objects. Rather, it relies on a `SpriteRenderable` object to draw each character in the string. For this reason, `FontRenderable` is not a subclass of but instead contains an instance of the `SpriteRenderable` object, the `mOneChar` variable.

3. Define the `draw()` function to parse and draw each character in the string using the `mOneChar` variable.

```
FontRenderable.prototype.draw = function(vpMatrix) {
    var widthOfOneChar = this.mXform.getWidth() / this.mText.length;
    var heightOfOneChar = this.mXform.getHeight();
    var yPos = this.mXform.getYPos();

    // center position of the first char
    var xPos = this.mXform.getXPos() - (widthOfOneChar / 2) +
    (widthOfOneChar * 0.5);
    var charIndex, aChar, charInfo, xSize, ySize, xOffset, yOffset;
    for (charIndex = 0; charIndex < this.mText.length; charIndex++) {
        aChar = this.mText.charCodeAt(charIndex);
        charInfo = gEngine.Fonts.getCharInfo(this.mFont, aChar);
```

```
            // set the texture coordinate
            this.mOneChar.setElementUVCoordinate(charInfo.mTexCoordLeft,
            charInfo.mTexCoordRight,
                charInfo.mTexCoordBottom, charInfo.mTexCoordTop);

            // now the size of the char
            xSize = widthOfOneChar * charInfo.mCharWidth;
            ySize = heightOfOneChar * charInfo.mCharHeight;
            this.mOneChar.getXform().setSize(xSize, ySize);

            // how much to offset from the center
            xOffset = widthOfOneChar * charInfo.mCharWidthOffset * 0.5;
            yOffset = heightOfOneChar * charInfo.mCharHeightOffset * 0.5;

            this.mOneChar.getXform().setPosition(xPos - xOffset, yPos - yOffset);

            this.mOneChar.draw(vpMatrix);

            xPos += widthOfOneChar;
        }
    };
```

The dimension of each character is defined by `widthOfOneChar` and `heightOfOneChar` where the width is simply dividing the total `FontRenderable` width by the number of characters in the string. The `for` loop then does the following:

a. Extracts each character in the string

b. Calls the `getCharInfo()` function to receive the character's uv values and appearance information in `charInfo`

c. Uses the uv values from `charInfo` to identify the sprite element location for `mOneChar` (by calling and passing the information to the `mOneChar.setElementUVCoordinate()` function)

d. Uses the appearance information from `charInfo` to compute the actual size (`xSize` and `ySize`) and location offset for the character (`xOffset` and `yOffset`) and draws the character `mOneChar` with the appropriate settings

4. Implement the getters and setters for the transform, the text message to be drawn, the font to use for drawing, and the color.

```
FontRenderable.prototype.getXform = function() { return this.mXform; };
FontRenderable.prototype.getText = function() { return this.mText; };
FontRenderable.prototype.setText = function(t) {
    this.mText = t;
    this.setTextHeight(this.getXform().getHeight());
};

FontRenderable.prototype.getFont = function() { return this.mFont; };
FontRenderable.prototype.setFont = function(f) {
    this.mFont = f;
    this.mOneChar.setTexture(this.mFont + ".png");
};
```

```
FontRenderable.prototype.setColor = function(c){ this.mOneChar.setColor(c); };
FontRenderable.prototype.getColor = function() {
    return this.mOneChar.getColor();
};
```

5. Define the `setTextHeight()` function to define the height of the message to be output.

```
FontRenderable.prototype.setTextHeight = function(h) {
    // this is for "A"
    var charInfo = gEngine.Fonts.getCharInfo(this.mFont, "A".charCodeAt(0));
    var w = h * charInfo.mCharAspectRatio;
    this.getXform().setSize(w * this.mText.length, h);
};
```

Notice that the width of the entire message to be drawn is automatically computed based on the message string length and maintaining the character width to height aspect ratio.

■ **Note** `FontRenderable` does not support the rotation of the entire message. Text messages are always drawn horizontally from left to right.

Adding a Default Font to the Engine

A default system font should be provided by the game engine for the convenience of the game programmer. This can be accomplished with simple modifications to the `gEngine_DefaultResources` component.

1. Edit the `Engine_DefaultResources.js` file to define the default system font name and to define a getter function to retrieve it.

```
// Default font
var kDefaultFont = "assets/fonts/system-default-font";
var getDefaultFont = function() { return kDefaultFont; };
```

2. Modify the `initialize()` function to also load the default font.

```
var initialize = function(callBackFunction) {
    // ... Identical to previous code ...

    // load default font
    gEngine.Fonts.loadFont(kDefaultFont);

    // ... Identical to previous code ...
};
```

After Game Cleanup

In all the examples, the WebGL resource allocation and cleanup have been carefully handled during scene transitions where loaded external resources are always unloaded. However, when the entire game is ready to terminate, there is no current support to release the allocated shared resources. In the game engine, there are two components that allocated WebGL resources for sharing: `gEngine_VertexBuffer` for sharing unit square vertex positions and static texture coordinate buffers, and `gEngine_DefaultResources` for the allocated shared resources.

The following files must be modified to support proper cleanup at the end of a game:

1. In the `Engine_Core.js` file, add the following function to instruct `VertexBuffer` and `DefaultResources` to release their allocated resources, and remember to add the `cleanup()` function to the public interface:

```
var cleanUp = function() {
    gEngine.VertexBuffer.cleanUp();
    gEngine.DefaultResources.cleanUp();
};

var mPublic = {
    // ... Identical to previous code ...
    cleanUp: cleanUp
};
```

2. In the `Engine_DefaultResources.js` file, add the following function to release all default resources, and remember to add the `cleanup()` function to the public interface so that this function is accessible:

```
var cleanUp = function() {
    mConstColorShader.cleanUp();
    mTextureShader.cleanUp();
    mSpriteShader.cleanUp();

    gEngine.TextFileLoader.unloadTextFile(kSimpleVS);
    gEngine.TextFileLoader.unloadTextFile(kSimpleFS);

    // textuire shader:
    gEngine.TextFileLoader.unloadTextFile(kTextureVS);
    gEngine.TextFileLoader.unloadTextFile(kTextureFS);

    // default font
    gEngine.Fonts.unloadFont(kDefaultFont);
};
```

3. In the `Engine_VertexBuffer.js` file, add the following function to release the allocated WebGL buffers, and remember to add the `cleanup()` function to the public interface so that this function is accessible:

```
var cleanUp = function() {
    var gl = gEngine.Core.getGL();
    gl.deleteBuffer(mSquareVertexBuffer);
    gl.deleteBuffer(mTextureCoordBuffer);
};
```

4. In the SpriteShader.js file, add the following function to release the allocated texture coordinate buffer:

```
SpriteShader.prototype.cleanUp = function() {
    var gl = gEngine.Core.getGL();
    gl.deleteBuffer(this.mTexCoordBuffer);

    // now call super class's clean up ...
    SimpleShader.prototype.cleanUp.call(this);
};
```

5. In the SimpleShader.js file, the WebGL memory allocated for vertex shader, fragment shader, and compiled GLSL shader must all be released. To accomplish this, instance variables must be defined to refer to the corresponding shaders.

```
function SimpleShader(vertexShaderPath, fragmentShaderPath) {

    // ... Identical to previous code ...

    // Step A: load and compile vertex and fragment shaders
    this.mVertexShader = this._CompileShader(vertexShaderPath,
                                             gl.VERTEX_SHADER);
    this.mFragmentShader = this._CompileShader(fragmentShaderPath,
                                             gl.FRAGMENT_SHADER);

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

    // Step C, D, E, F, G:
    // ... Identical to previous code ...
};

SimpleShader.prototype.cleanUp = function() {
    var gl = gEngine.Core.getGL();
    gl.detachShader(this.mCompiledShader, this.mVertexShader);
    gl.detachShader(this.mCompiledShader, this.mFragmentShader);
    gl.deleteShader(this.mVertexShader);
    gl.deleteShader(this.mFragmentShader);
};
```

   a. The constructor is modified to reference the vertex and fragment shaders with the mVertexShader and mFragmentShader variables.

   b. The new cleanUp() function releases the memory associated with the shaders.

Testing Fonts

You are now ready to test the system font support for drawing text messages and to actually end the game by stopping the game loop. The first step is to define the GameOver scene.

The GameOver Scene

The goal of this scene is to output a "Game Over" message for the player and end the game gracefully. The scene will set up a proper WC system, draw the "Game Over" text, and stop the game loop.

1. Create a new file in the src/Engine/MyGame folder and name it GameOver.js.

2. Implement the constructor, with the mCamera to set up a convenient World Coordinate (WC) system and the mMsg to contain the "Game Over" message.

```
function GameOver() {
    this.mCamera = null;
    this.mMsg = null;
};
gEngine.Core.inheritPrototype(GameOver, Scene);
```

3. In the initialize() function, define a convenient WC and initialize the mMsg object.

```
GameOver.prototype.initialize = function() {
    // Step A: set up the cameras
    this.mCamera = new Camera(
            vec2.fromValues(50, 33),  // position of the camera
            100,                      // width of camera
            [0, 0, 600, 400]          // viewport (orgX, orgY, width, height)
            );
    this.mCamera.setBackgroundColor([0.9, 0.9, 0.9, 1]);

    this.mMsg = new FontRenderable("Game Over!");
    this.mMsg.setColor([0, 0, 0, 1]);
    this.mMsg.getXform().setPosition(22, 32);
    this.mMsg.setTextHeight(10);
};
```

The mMsg is instantiated as a FontRenderable object with "Game Over" as the default message. Notice that only the height of the text is set, setTextHeight(); the width of the entire message is computed based on preserving the aspect ratio of the font.

4. Draw the text in the same manner as all Renderable objects.

```
GameOver.prototype.draw = function() {
    // Step A: clear the canvas
    gEngine.Core.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray

    // Step  B: Activate the drawing Camera
    this.mCamera.setupViewProjection();
    this.mMsg.draw(this.mCamera.getVPMatrix());
};
```

5. In the `update()` function, stop the `GameLoop` to trigger the call to the `unloadScene()` function.

```
GameOver.prototype.update = function() {
    gEngine.GameLoop.stop();
};
```

6. In the `unloadScene()` function, call `gEngine_Core` to clean up all system default resources.

```
GameOver.prototype.unloadScene = function() {
    gEngine.Core.cleanUp(); // release gl resources
};
```

The `GameOver` scene is designed to print the "Game Over" message and terminate the entire game.

The MyGame Scene

You can now finally modify the `MyGame` scene to print messages with the various fonts found in the `assets` folder.

1. In the `MyGame.js` file, modify the constructor to contain corresponding variables for printing the messages, and modify the `draw()` function to draw all objects accordingly. Please refer to the `src/MyGame/MyGame.js` file for the details of the code.

2. Modify the `loadScene()` function to load the textures and fonts.

```
MyGame.prototype.loadScene = function() {
    // Step A: loads the textures
    gEngine.Textures.loadTexture(this.kFontImage);
    gEngine.Textures.loadTexture(this.kMinionSprite);

    // Step B: loads all the fonts
    gEngine.Fonts.loadFont(this.kFontCon16);
    gEngine.Fonts.loadFont(this.kFontCon24);
    gEngine.Fonts.loadFont(this.kFontCon32);
    gEngine.Fonts.loadFont(this.kFontCon72);
    gEngine.Fonts.loadFont(this.kFontSeg96);
};
```

3. Modify the `unloadScene()` function to unload the textures and fonts and to start the `GameOver` scene.

```
MyGame.prototype.unloadScene = function() {
    gEngine.Textures.unloadTexture(this.kFontImage);
    gEngine.Textures.unloadTexture(this.kMinionSprite);

    // unload the fonts
    gEngine.Fonts.unloadFont(this.kFontCon16);
    gEngine.Fonts.unloadFont(this.kFontCon24);
```

```
    gEngine.Fonts.unloadFont(this.kFontCon32);
    gEngine.Fonts.unloadFont(this.kFontCon72);
    gEngine.Fonts.unloadFont(this.kFontSeg96);

    // Step B: starts the next level
    var nextLevel = new GameOver();  // next level to be loaded
    gEngine.Core.startScene(nextLevel);
};
```

4. Define a private `_initText()` function to set the color, location, and height of a `FontRenderable` object. Modify the `initialize()` function to set up the proper WC system and initialize the fonts.

```
MyGame.prototype._InitText = function(font, posX, posY, color, textH) {
    font.setColor(color);
    font.getXform().setPosition(posX, posY);
    font.setTextHeight(textH);
};

MyGame.prototype.initialize = function() {
    // Step A: set up the cameras
    // ... Identical to previous code ...

    // Step B: Create the font and minion images using sprite
    // ... Identical to previous code ...

    // Create the fonts!
    this.mTextSysFont = new FontRenderable("System Font: in Red");
    this._InitText(this.mTextSysFont, 50, 60, [1, 0, 0, 1], 3);

    this.mTextCon16 = new FontRenderable("Consolas 16: in black");
    this.mTextCon16.setFont(this.kFontCon16);
    this._InitText(this.mTextCon16, 50, 55, [0, 0, 0, 1], 2);

    this.mTextCon24 = new FontRenderable("Consolas 24: in black");
    this.mTextCon24.setFont(this.kFontCon24);
    this._InitText(this.mTextCon24, 50, 50, [0, 0, 0, 1], 3);

    this.mTextCon32 = new FontRenderable("Consolas 32: in white");
    this.mTextCon32.setFont(this.kFontCon32);
    this._InitText(this.mTextCon32, 40, 40, [1, 1, 1, 1], 4);

    this.mTextCon72 = new FontRenderable("Consolas 72: in blue");
    this.mTextCon72.setFont(this.kFontCon72);
    this._InitText(this.mTextCon72, 30, 30, [0, 0, 1, 1], 6);

    this.mTextSeg96  = new FontRenderable("Segment7-92");
    this.mTextSeg96.setFont(this.kFontSeg96);
    this._InitText(this.mTextSeg96 , 30, 15, [1, 1, 0, 1], 7);

    this.mTextToWork = this.mTextCon16;
};
```

Notice the calls to `setFont()` function to change the font type for each message.

5. Modify the `update()` function with the following:

```
MyGame.prototype.update = function() {
    // Controlling of hero and zooming of font image
    // ... Identical to previous code ...

    // choose which text to work on
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Zero))
        this.mTextToWork = this.mTextCon16;
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.One))
        this.mTextToWork = this.mTextCon24;
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Three))
        this.mTextToWork = this.mTextCon32;
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Four))
        this.mTextToWork = this.mTextCon72;

    var deltaF = 0.005;
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Up)) {
         if (gEngine.Input.isKeyPressed(gEngine.Input.keys.X)) {
             this.mTextToWork.getXform().incWidthBy(deltaF);
         }
         if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Y)) {
            this.mTextToWork.getXform().incHeightBy(deltaF);
        }
        this.mTextSysFont.setText(
            this.mTextToWork.getXform().getWidth().toFixed(2) +
            "x" + this.mTextToWork.getXform().getHeight().toFixed(2));
    }

    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Down)) {
         if (gEngine.Input.isKeyPressed(gEngine.Input.keys.X)) {
             this.mTextToWork.getXform().incWidthBy(-deltaF);
         }
         if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Y)) {
            this.mTextToWork.getXform().incHeightBy(-deltaF);
        }
        this.mTextSysFont.setText(
            this.mTextToWork.getXform().getWidth().toFixed(2) + "x"
            + this.mTextToWork.getXform().getHeight().toFixed(2));
    }
};
```

The previous code shows that you can do the following:

a. Select which `FontRenderable` object to work with based on keyboard 0 to 4 input.

b. Control the width and height of the selected `FontRenderable` object when both the left/right arrow and x/y keys are pressed.

You can now interact with the FontSupport project to modify each of the displayed font message sizes and to move the hero to trigger the end of the game.

Summary

In this chapter, you learned how to paste, or texture map, images on unit squares to better represent objects in your games. You also learned how to identify and texture map a selected subregion of an image to the unit square based on the normalize-ranged texture coordinate system. The chapter then explained how sprite sheets can conserve the time required for loading texture images and facilitate the creation of animations. This knowledge was then generalized and applied to the drawing of bitmap fonts.

The implementation of texture mapping and sprite sheet rendering take advantage of an important aspect of game engine architecture: the `Shader/Renderable` object pair where JavaScript `Shader` objects are defined to interface with corresponding GLSL shaders and `Renderable` objects to facilitate the creation and interaction with multiple object instances. For example, you created `TextureShader` to interface with `TextureVS` and `TextureFS` GLSL shaders and created `TextureRenderable` for the game programmers to work with. This same pattern is repeated for `SpriteShader` and `SpriteRenderable`. The experience from `SpriteShader` objects pairing with `SpriteAnimateRenderable` shows that, when appropriate, the same `Shader` object can support multiple renderable object types in the game engine. This `Shader/Renderable` pair implementation pattern will appear again in Chapter 8, when you learn to create 3D illumination effects. Lastly, you have learned about the importance of cleaning up system resources when the game terminates.

At the beginning of this chapter, your game engine supports the player manipulating objects with the keyboard and the drawing of these objects in various sizes and orientations. With the functionality from this chapter, you can now represent these objects with interesting images and create animations of these objects when desired. In the next chapter, you will learn about supporting intrinsic properties for these objects including pseudo autonomous behaviors such as chasing and collision detections.

Game Design Considerations

In Chapter 4 you learned how responsive game feedback is essential to making players feel connected to a game world and that this sense of connection is known as *presence* in game design. As you move through future chapters in this book, you'll notice that most game design is ultimately focused on enhancing the sense of presence in one way or another, and you'll discover that visual design is one of the most important contributors to presence. Imagine, for example, a game where an object controlled by the player (referred to as the *hero* moving forward) must maneuver through a 2D platformer-style game world; the player's goal might be to use the mouse and keyboard to jump the hero between individual surfaces rendered in the game without falling through gaps that exist between those surfaces. The visual representation of the hero and other objects in the environment determine how the player identifies with the game setting, which in turn determines how effectively the game creates presence. Is the hero represented as a living creature or just an abstract shape like a square or circle? Are the surfaces represented as building rooftops, as floating rocks on an alien planet, or simply as abstract rectangles? There is no right or wrong answer when it comes to selecting a visual representation or game setting, but it is important to design a visual style for all game elements that feels unified and integrated into whatever game setting you choose (for example, abstract rectangle platforms may negatively impact presence if your game setting is a tropical rainforest).

The Texture Shaders project demonstrated how `.png` images with transparency more effectively integrate game elements into the game environment than formats like JPEG that don't support transparency. If you move the hero (represented here as simply a rectangle) to the right, nothing on the screen changes, but if you move the hero to the left, you'll eventually trigger a state change that alters the displayed visual elements as you did in the Scene Objects project from Chapter 4. Notice how much more effectively the robot sprites are integrated into the game scene when they're `.png` files with transparency on the gray background compared to when they're `.jpg` images without transparency on the blue background. The Sprite Shaders project introduces

a hero that more closely matches other elements in the game setting. You've replaced the hero rectangle from the Texture Shaders project with a humanoid figure stylistically matched to the flying robots on the screen, and the area of the rectangular hero image not occupied by the humanoid figure is transparent. If you were to combine the hero from the Sprite Shaders project with the screen-altering action in the Texture Shaders project, imagine that as the hero moves toward the robot on the right side of the screen, the robot might turn red when the hero gets too close. The coded events are still simple at this point, but you can see how the visual design and a few simple triggered actions can already begin to convey a game setting and enhance presence.

Note that as game designers, we often become enamored with highly detailed and elaborate visual designs, and we begin to believe that higher fidelity and more elaborate visual elements are required to make the best games. This drive for ever-more powerful graphics is the familiar race that many AAA games engage in with their competition. While it's true that game experiences and the sense of presence can be considerably enhanced when paired with excellent art direction, excellence does not always require elaborate and complex. Good art direction relies on developing a unified visual language where all elements harmonize with each other and contribute to driving the game forward, and that harmony can be achieved with anything from simple shapes and colors in a 2D plane to hyper-real 3D environments, and every combination in between.

Adding animated motion to the game's visual elements can further enhance game presence because animation brings a sense of cinematic dynamism to gameplay that further connects players to the game world. We typically experience motion in our world as interconnected systems. When you walk across the room, for example, you don't just glide without moving your body; you move different parts of your body together in different ways. By adding targeted animations to objects onscreen that cause those objects to behave in ways you might expect complex systems to move or act, you connect players in a more immersive and engaging way to what's going on in the game world. The Sprite Animation project demonstrates how animation increases presence by allowing you to articulate the flying robot's spikes, controlling direction and speed. And again, imagine combining the Sprite Animation project with the earlier projects in this chapter. As the hero moves closer to the robot, it might turn first turn red; closer still might trigger the robot's animations and move it either toward or away from the player. Animations often come fairly late in the game design process because it's helpful to have the game mechanic and other systems well defined so that animations, which tend to be time-consuming to change once complete, can be produced after level designs have been finalized and tested with placeholder assets.

As was the case with visual design, the animation approach need not be especially complex to be effective. Animation needs to be intentional and unified, and it should feel smooth and stutter-free unless it's intentionally designed to be otherwise; a wide degree of artistic license can be employed in how movement is represented onscreen.

The Font Support project introduced you to game fonts. While fonts rarely have a direct impact on gameplay, they can have a dramatic impact on presence. Fonts are a form of visual communication, and the style of the font is often as important as the words it conveys in setting tone and mood and can either support or detract from the game setting and visual style. Pay particular attention to the fonts displayed in this project. Note how the yellow font conveys a digital feeling that's matched to the science fiction–inspired visual style of the hero and robots, while the Consolas font family with its round letterforms feels a bit out of place with this game setting (sparse though the game setting may still be). As a more extreme example, imagine how disconnected a flowing calligraphic script font (the type typically used in high-fantasy games) would appear in a futuristic game that takes place on a spaceship.

There are as many visual style possibilities for games as there are people and ideas, and great games can feature extremely simple graphics. Remember that excellent game design is a combination of the nine contributing elements (return to the introduction if you need to refresh your memory), and the most important thing to keep in mind as a game designer is maintaining focus on how each of those elements harmonizes with and elevates the others to create something greater than the sum of its parts.