Working with Textures, Sprites, and Fonts

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, "an image" and "a texture" are often used interchangeably. A *pixel* is a color location in an image and a *texel* is a color location in a texture.

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.

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.

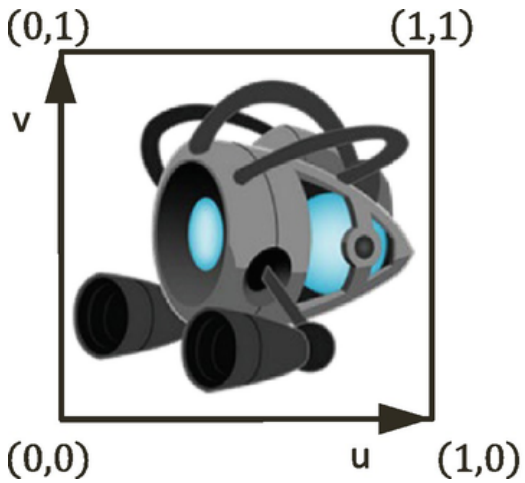

Figure 5-1. The Texture Coordinate System

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.

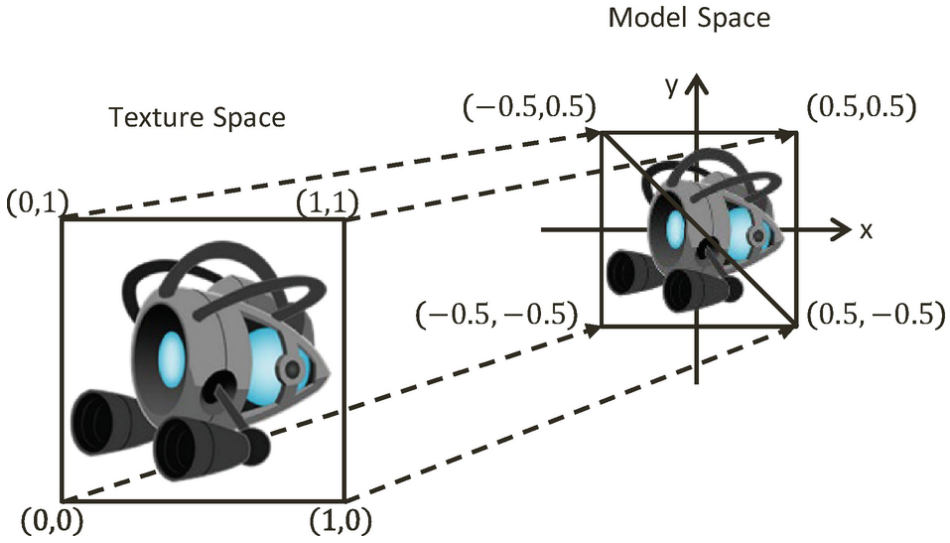

Figure 5-2. Mapping Texture Space to Model Space

Graphic Assets

A number graphics resources will be needed for these projects. Ask your instructor for assets as they are needed. We will commonly use four images as we start building parts of the game engine that are more "game-like". These images will commonly be referred to as "hero", "portal", "collector" and "minion" which you can see in figure 5-3. Variable names will use these designators.

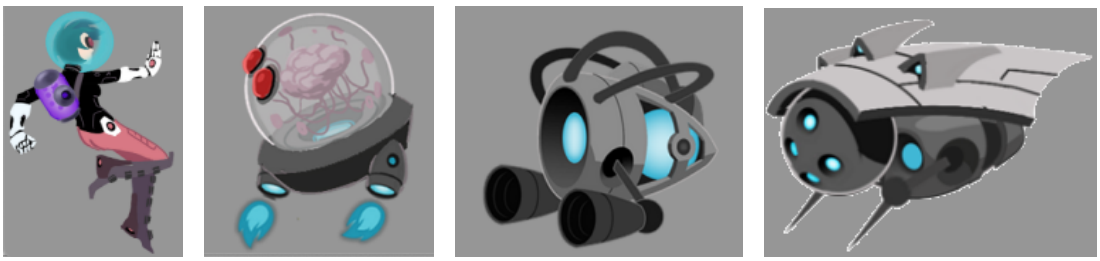

Figure 5-3. Hero, Portal, Collector, Minion

Lab 5.1
The Texture Shaders Project

This project demonstrates the loading, rendering, and unloading of textures with WebGL.

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

Several graphics resources will be needed for this project. These will be provided by the instructor: a scene-level file (`blue_level.xml`) and four images (`minion_collector.jpg`, `minion_collector.png`, `minion_portal.jpg`, and `minion_portal.png`).

TASK This project will not be using audio. If you copied from a previous project that uses audio, you can remove all audio assets from the `assets` folder and make sure to remove all audio-related code from `my_game.js` and `blue_level.js`. You can do this as you encounter them during the development of this project.

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:

- `texture_vs.glsl` and `texture_fs.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.
- `vertex_buffer.js`: This file is modified to create a corresponding uv coordinate buffer to define the texture coordinate for the vertices of the unit square.
- `texture_shader.js`: This is a new file that defines `TextureShader` as a subclass of `SimpleShader` to interface the game engine with the corresponding GLSL shaders (`TextureVS` and `TextureFS`).
- `texture_renderable.js`: This is a new file that defines `TextureRenderable` as a subclass of `Renderable` to facilitate the creation, manipulation, and drawing of multiple instances of textured objects.

- `shader_resources.js`: Recall that this file defines a single instance of `SimpleShader` to wrap over the corresponding GLSL shaders to be shared system wide by all instances of `Renderable` objects. In a similar manner, this file is modified to define an instance of `TextureShader` to be shared by all instances of `TextureRenderable` objects.
- `gl.js`: This file is modified to configure WebGL to support drawing with texture maps.
- `texture.js`: This is a new file that defines the core engine component that is capable of loading, activating (for rendering), and unloading texture images.
- `my_game.js` and `blue_level.js`: These game engine client files are modified to test the new texture mapping functionality.

Two new source code folders, `src/engine/shaders` and `src/engine/renderables`, are created for organizing the engine source code. These folders are created in anticipation of the many new shader and renderer types required to support the corresponding texture-related functionality. Once again, continuous source code reorganization is important in supporting the corresponding increase in complexity. A systematic and logical source code structure is critical in maintaining and expanding the functionality of large software systems.

Extension of Shader/Renderable Architecture

Recall that the `SimpleShader`/`Renderable` object pair is designed to support the loading of 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 `SimpleShader` 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 `SimpleShader` object created in the game engine, in `shader_resources`, and this instance is shared by all `Renderable` objects.

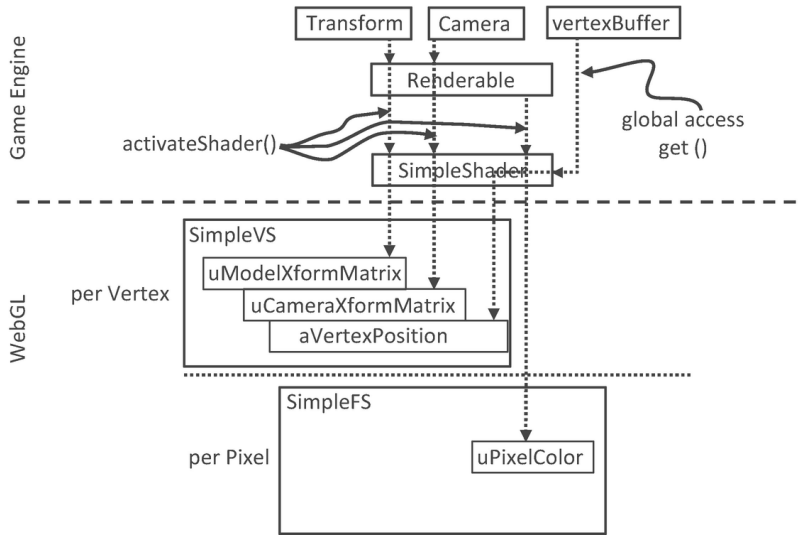

Figure 5-4. The Shader and Renderable architecture

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 `SimpleShader/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.

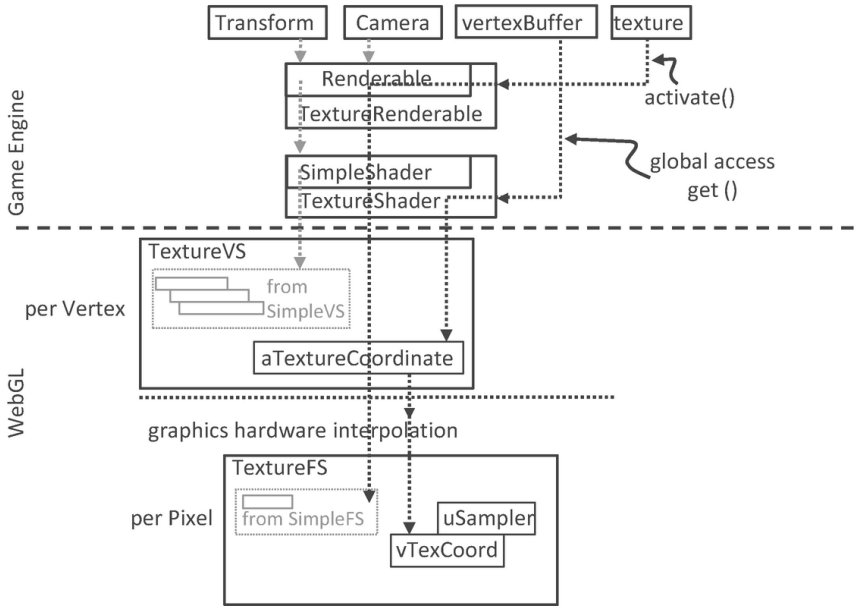

Figure 5-5. The TextureVS/FS GLSL shaders and the corresponding TextureShader/TextureRenderable object pair

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. You can now begin to create the texture vertex shader.

1. Create a new file in the `src/glsl_shaders` folder and name it `texture_vs.glsl`.

2. Add the following code to the `texture_vs.glsl` file:

```
attribute vec3 aVertexPosition; // expects one vertex position
attribute vec2 aTextureCoordinate; // texture coordinate attribute

// texture coordinate that maps image to the square
varying vec2 vTexCoord;

// to transform the vertex position
uniform mat4 uModelXformMatrix;
uniform mat4 uCameraXformMatrix;

void main(void) {
  // Convert the vec3 into vec4 for scan conversion and
  // transform by uModelXformMatrix and uCameraXformMatrix before
  // assign to gl_Position to pass the vertex to the fragment shader
  gl_Position = uCameraXformMatrix *
                uModelXformMatrix *
                vec4(aVertexPosition, 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. The first additional line adds the `aTextureCoordinate` attribute. This defines a vertex to include a `vec3` (`aVertexPosition`, the xyz position of the vertex) and a `vec2` (`aTextureCoordinate`, the uv coordinate of the vertex).
b. The second declares 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. The third and final line assigns 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/glsl_shaders` folder and name it `texture_fs.glsl`.

2. Add the following code to the `texture_fs.glsl` file to declare the variables. 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 bound 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.

```
// The object that fetches data from texture.

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

// Must be set outside the shader.
uniform sampler2D uSampler;

// Color of pixel
uniform vec4 uPixelColor;

// "varying" keyword signifies that the texture coordinate will be
// interpolated and thus varies.
varying vec2 vTexCoord;
```

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. transparent area 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;
}
```

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 `vertex_buffer.js` file. In a similar fashion, a corresponding buffer must be defined to supply texture coordinates to the GLSL shaders.

1. Modify `vertex_buffer.js` to define both xy and uv coordinates for the unit square. As illustrated in Figure 5-2, the `mTextureCoordinates` variable defines the uv values for the corresponding four xy values of the unit square defined sequentially in `mVerticesOfSquare`. 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.

```
// First: define the vertices for a square
let mVerticesOfSquare = [
  0.5, 0.5, 0.0,
  -0.5, 0.5, 0.0,
  0.5, -0.5, 0.0,
  -0.5, -0.5, 0.0
];
// Second: define the corresponding texture coordinates
let mTextureCoordinates = [
  1.0, 1.0,
  0.0, 1.0,
  1.0, 0.0,
  0.0, 0.0
];
```

2. Define the variable, `mGLTextureCoordBuffer`, to keep a reference to the WebGL buffer storage for the texture coordinate values of `mTextureCoordinates` and the corresponding getter function:

```
let mGLTextureCoordBuffer = null;
function getTexCoord() { return mGLTextureCoordBuffer; }
```

3. Modify the `init()` function to include a step D to initialize the texture coordinates as a WebGL buffer. Notice the initialization process is identical to that of the vertex xy coordinates except that the reference to the new buffer is stored in `mGLTextureCoordBuffer` and the transferred data are the uv coordinate values.

```
function init() {
  let gl = glSys.get();
  ... identical to previous code ...
```

```
    // Step  D: Allocate and store texture coordinates
    // Create a buffer on the gl context for texture coordinates
    mGLTextureCoordBuffer = gl.createBuffer();
    // Activate texture coordinate buffer
    gl.bindBuffer(gl.ARRAY_BUFFER, mGLTextureCoordBuffer);
    // Loads textureCoordinates into the mGLTextureCoordBuffer
    gl.bufferData(gl.ARRAY_BUFFER,
                  new Float32Array(mTextureCoordinates), gl.STATIC_DRAW);
  }
```

4. Remember to release the allocated buffer during final cleanup:

```
  function cleanUp() {
    ... identical to previous code ...
    if (mGLTextureCoordBuffer !== null) {
        gl.deleteBuffer(mGLTextureCoordBuffer);
        mGLTextureCoordBuffer = null;
    }
  }
```

5. Finally, remember to export the changes:

```
export {init, cleanUp, get, getTexCoord}
```

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. As mentioned in the overview of this project, you will also create a new folder to organize the growing number of different shaders.

1. Create a new folder called `shaders` in `the src/engine` folder. Move the `simple_shader.js` file into this folder.

TASK 5.1A You must also update the reference path in `shader_resources.js`.

2. Create a new file in the `src/engine/shaders` folder and name it `texture_shader.js`.

```
"use strict";

class TextureShader extends SimpleShader {
  constructor(vertexShaderPath, fragmentShaderPath) {
```

```
        // Call super class constructor
        super(vertexShaderPath, fragmentShaderPath);
        // reference to aTextureCoordinate within the shader
        this.mTextureCoordinateRef = null;
        // get the reference of aTextureCoordinate within the shader
        let gl = glSys.get();
        this.mTextureCoordinateRef = gl.getAttribLocation(
                                        this.mCompiledShader,
                                        "aTextureCoordinate");
        this.mSamplerRef = gl.getUniformLocation(this.mCompiledShader,
                                                 "uSampler");
    }
    ... implementation to follow ...
}
```

a. The defined `TextureShader` class is an extension, or subclass, to the `SimpleShader` class.
b. The constructor implementation first calls `super()`, the constructor of `SimpleShader`. Recall that the `SimpleShader` constructor will load and compile the GLSL shaders defined by the `vertexShaderPath` and `fragmentShaderPath` parameters and set `mVertexPositionRef` to reference the `aVertexPosition` attribute defined in the shader.
c. In the rest of the constructor, the `mTextureCoordinateRef` keeps a reference to the `aTextureCoordinate` attribute defined in the `texture_vs.glsl`.
d. In this way, both the vertex position (`aVertexPosition`) and texture coordinate (`aTextureCoordinate`) attributes are referenced by a JavaScript `TextureShader` object.

TASK 5.1B - Two steps are intentionally omitted.
First, there are no `import` statements indicated in the code. Determine which imports are needed and add them.

Hint1: 3 import statements are needed

Hint2: The following lines of code from above should help you figure out which imports are needed

```
class TextureShader extends SimpleShader
let gl = glSys.get();
return vertexBuffer.getTexCoord();
```

You will also need to provide an `export`.

Be sure to check for other missing elements (such as closing curly braces). Visual Studio Code will highlight things that may be errors and variables that you have declared but aren't using.

3. Override the `activate()` function to enable the texture coordinate data. The superclass `super.activate()` function sets up the xy vertex position and passes the values of `pixelColor`, `trsMatrix`, and `cameraMatrix` to the shader. The rest of the code binds `mTextureCoordinateRef`, the texture coordinate buffer defined in

the `vertex_buffer` module, to the `aTextureCoordinate` attribute in the GLSL shader and `mSampler` to texture unit 0 (to be detailed later).

```
// Overriding the Activation of the shader for rendering
activate(pixelColor, trsMatrix, cameraMatrix) {
  // first call the super class's activate
  super.activate(pixelColor, trsMatrix, cameraMatrix);
  // now our own functionality: enable texture coordinate array
  let gl = glSys.get();
  gl.bindBuffer(gl.ARRAY_BUFFER, this._getTexCoordBuffer());
  gl.vertexAttribPointer(this.mTextureCoordinateRef, 2,
                         gl.FLOAT, false, 0, 0);
  gl.enableVertexAttribArray(this.mTextureCoordinateRef);
  // bind uSampler to texture 0
  gl.uniform1i(this.mSamplerRef, 0);
      // texture.activateTexture() binds to Texture0
}

_getTexCoordBuffer() {
   return vertexBuffer.getTexCoord();
}
```

With the combined functionality of `SimpleShader` and `TextureShader`, after the `activate()` function call, both of the attribute variables (`aVertexPosition` and `aTextureCoordinate`) in the GLSL `texture_vs` shader are connected to the corresponding buffers in the WebGL memory.

Facilitate Sharing with shader_resources

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

1. Import `TextureShader`.

```
import TextureShader from "../shaders/texture_shader.js";
```

2. In `shader_resources.js`, add the variables to hold a texture shader:

```
// Texture Shader
let kTextureVS = "src/glsl_shaders/texture_vs.glsl"; // VertexShader
let kTextureFS = "src/glsl_shaders/texture_fs.glsl"; // FragmentShader
let mTextureShader = null;
```

3. Define a function to retrieve the texture shader:

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

4. Create the instance of texture shader in the `createShaders()` function:

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

5. Modify the `init()` function to append the `loadPromise` to include the loading of the texture shader source files:

```
function init() {
  let loadPromise = new Promise(
      async function(resolve) {
          await Promise.all([
              text.load(kSimpleFS),
              text.load(kSimpleVS),
              text.load(kTextureFS),
              text.load(kTextureVS)
          ]);
          resolve();
      }).then(
          function resolve() { createShaders(); }
      );
  map.pushPromise(loadPromise);
}
```

6. Remember to release newly allocated resources during cleanup:

```
function cleanUp() {
  mConstColorShader.cleanUp();
  mTextureShader.cleanUp();
  text.unload(kSimpleVS);
  text.unload(kSimpleFS);
  text.unload(kTextureVS);
  text.unload(kTextureFS);
}
```

7. Lastly, remember to export the newly defined functionality:

```
export {init, cleanUp, getConstColorShader, getTextureShader}
```

TextureRenderable Class

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 Class

As mentioned in the project overview, 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 a folder called `renderables` inside the `src/engine/` folder and move `renderable.js` into this folder.

TASK 5.1C update the `import` statement in `index.js` to point to the new location for `renderable.js`.

2. In `renderable.js`, define the `_setShader()` function to set the shader for the `Renderable`. This is a protected function which allows subclasses to modify the `mShader` variable to refer to the appropriate shaders for each corresponding subclass.

```
// this is private/protected
_setShader(s) { this.mShader = s; }
```

Note Functions with names that begin with "_" are either private or protected and should not be called from outside of the class. This is a convention followed in this book and not enforced by JavaScript.

Define the TextureRenderable Class

You are now ready to define the `TextureRenderable` class. As noted, `TextureRenderable` is derived from and extends the `Renderable` class functionality to render texture mapped objects.

1. Create a new file in the `src/engine/renderables` folder and name it `texture_renderable.js`. Add the constructor. Recall that `super()` is a call to the superclass (`Renderable`) constructor; similarly, the `super.setColor()` and `super._setShader()` are calls to the superclass functions. As will be detailed when discussing the engine texture resource module, the `myTexture` parameter is the path to the file that contains the texture image.

```
"use strict"
```

```
import * as shaderResources from "../core/shader_resources.js";
import Renderable from "./renderable.js";
import * as texture from "../resources/texture.js";
class TextureRenderable extends Renderable {
  constructor(myTexture) {
      super();
      super.setColor([1, 1, 1, 0]); // Alpha 0: no texture tinting
      super._setShader(shaderResources.getTextureShader());
      this.mTexture = myTexture;  // cannot be a "null"
  }
... implementation to follow ...
}
```

2. Define a `draw()` function to append the function defined in the `Renderable` class to support textures. The `texture.activate()` function activates and allows drawing with the specific texture. The details of this function will be discussed in the following section.

```
draw(camera) {
  // activate the texture
  texture.activate(this.mTexture);
  super.draw(camera);
}
```

3. Define a getter and setter for the texture reference:

```
getTexture() { return this.mTexture; }
setTexture(newTexture) { this.mTexture = newTexture; }
```

4. Finally, remember to export the class:

```
export default TextureRenderable;
```

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 `gl.js`, update the `init()` function according to the following:

```
function init(htmlCanvasID) {
  mCanvas = document.getElementById(htmlCanvasID);
  if (mCanvas == null)
      throw new Error("Engine init [" +
                      htmlCanvasID + "] HTML element id not found");
  // the standard or experimental webgl and binds to the Canvas area
  // store the results to the instance variable mGL
   mGL = mCanvas.getContext("webgl2", {alpha: false});

  if (mGL === null) {
      document.write("<br><b>WebGL 2 is not supported!</b>");
      return;
  }
  // Allows transparency with textures.
  mGL.blendFunc(mGL.SRC_ALPHA, mGL.ONE_MINUS_SRC_ALPHA);
  mGL.enable(mGL.BLEND);
  // Set images to flip y axis to match the texture coordinate space.
  mGL.pixelStorei(mGL.UNPACK_FLIP_Y_WEBGL, true);
}
```

The parameter passed to `mCanvas.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 Resource Module

Similar to text and audio files, a new engine component must be defined to support the corresponding texture operations including loading from the server file system, storing via the WebGL context to the GPU memory, activating the texture buffer for drawing, and removing from the GPU:

1. Create a new file in the `src/engine/resources` folder and name it `texture.js`. This file will implement the `Texture` engine component.

2. Import the core resource management functionality from the `resource_map`:

```
"use strict";
import * as glSys from "../core/gl.js";
import * as map from "../core/resource_map.js";
// functions from resource_map
let has = map.has;
let get = map.get;
```

3. Define the `TextureInfo` class to represent a texture in the game engine. The `mWidth` and `mHeight` are the pixel resolution of the texture image, and `mGLTexID` is a reference to the WebGL texture storage.

```
class TextureInfo {
  constructor(w, h, id) {
      this.mWidth = w;
      this.mHeight = h;
      this.mGLTexID = id;
  }
}
```

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

4. Define a function to load an image asynchronously as a promise and push the promise to be part of the pending promises in the `map`. Distinct from the text and audio resources, JavaScript `Image` API supports straightforward image file loading, and the `map.loadDecodeParse()` is not required in this case. Once an image is loaded, it is passed to the `processLoadedImage()` function with its file path as the name.

```
function load(textureName) {
   let texturePromise = null;
   if (map.has(textureName)) {
    map.incRef(textureName);
   } else {
    map.loadRequested(textureName);
    let image = new Image();
    texturePromise = new Promise(
          function(resolve) {
                image.onload = resolve;
                image.src = textureName;
          }).then(
          function resolve() {
                processLoadedImage(textureName, image);
          }
    );
    map.pushPromise(texturePromise);
   }
   return texturePromise;
}
```

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

```
// Remove the reference to allow associated memory
// to be available for subsequent garbage collection
```

```
function unload(textureName) {
   let texInfo = get(textureName);
   if (map.unload(textureName)) {
      let gl = glSys.get();
      gl.deleteTexture(texInfo.mGLTexID);
   }
}
```

6. Now define the `processLoadedImage()` function to convert the format of an image and store it to the WebGL context. The `gl.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 the `resource_map` according to the file path to the texture image file.

```
function processLoadedImage(path, image) {
  let gl = glSys.get();
  // Generate a texture reference to the webGL context
  let textureID = gl.createTexture();
  // binds texture reference with current texture in the webGL
  gl.bindTexture(gl.TEXTURE_2D, textureID);
  // Loads texture to texture data structure with descriptive info.
  // Parameters:
  //  1: "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 done manipulating data at the mGL.TEXTURE_2D target.
  gl.bindTexture(gl.TEXTURE_2D, null);
  let texInfo = new TextureInfo(image.naturalWidth,
                                image.naturalHeight, textureID);
  map.set(path, texInfo);
}
```

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:

```
function activate(textureName) {
  let gl = glSys.get();
  let texInfo = get(textureName);

  // Binds texture reference to the current webGL texture functionality
  gl.activeTexture(gl.TEXTURE0);
  gl.bindTexture(gl.TEXTURE_2D, texInfo.mGLTexID);

  // To prevent texture wrapping
  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 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 `get()` function locates the `TextureInfo` object from the `resource_map` 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.
c. 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 unprocessed textures best suitable for pixelated effects. Notice that in this case, 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. For example, a square that occupies a 64x64 pixel space should ideally use a 64x64 texel texture.

8. Define a function to deactivate a texture as follows. This function sets the WebGL context to a state of not working with any texture.

```
function deactivate() {
  let gl = glSys.get();
  gl.bindTexture(gl.TEXTURE_2D, null);
}
```

9. Finally, remember to export the functionality:

```
export {has, get, load, unload,
  TextureInfo,
  activate, deactivate}
```

Export New Functionality to the Client

The last step in integrating texture functionality into the engine involves modifying the engine access file, `index.js`. Edit `index.js` and add in the following import and export statements to grant the client access to the `texture` resource module and the `TextureRenderable` class:

```
... identical to previous code ...
import * as texture from "./resources/texture.js";
// renderables
import Renderable from "./renderables/renderable.js";
import TextureRenderable from "./renderables/texture_renderable.js";
... identical to previous code ...
export default {
   // resource support
   audio, text, xml, texture,
   // input support
   input,
   // Util classes
   Camera, Scene, Transform,
   // Renderables
   Renderable, TextureRenderable,
   // functions
   init, cleanUp, clearCanvas
}
```

Testing of Texture Mapping Functionality

With the described 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 the JPEG format.

As in all cases of building a game, it is essential to ensure that all external resources are properly organized. Recall that the assets folder is created specifically for the organization of external resources.

Modify the BlueLevel Scene File to Support Textures

The `blue_level.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 `texture_fs.glsl`, the alpha value of the `Color` element is used for tinting the texture map. The XML scene description is meant to support slight tinting of the `minion_portal.jpg` texture and no tinting of the `minion_collector.jpg` texture.

Modify SceneFileParser

The scene file parser, `scene_file_parser.js`, is modified to support the parsing of the updated `blue_scene.xml`, in particular, to parse `Square` elements into `Renderable` objects and `TextureSquare` elements into `TextureRenderable` objects.

1. Update the constructor to take in a file path for a parameter. We call the `get()` function of `resource_map` to retrieve the file.

```
constructor(sceneFilePath) {
  this.mSceneXml = engine.text.get(sceneFilePath);
}
```

2. Convert the `getElm()` function to a private method in the class called `_getElm()`. We modify the method to use `this.mSceneXml`.

```
_getElm(tagElm) {
   let theElm = this.mSceneXml.getElementsByTagName(tagElm);
   if (theElm.length === 0) {
    console.error("Warning: Level element:[" + tagElm + "]: not found!");
   }
   return theElm;
}
```

3. Modify `parseCamera()` to use the new function

```
parseCamera() {
   let camElm = this._getElm("Camera");
   ... identical to previous code ...
}
```

4. Modify `parseSquares()` to use the new function

```
parseSquares(sqSet) {
      let elm = this._getElm("Square");
   ... identical to previous code ...
}
```

5. Create `parseTextureSquares()` to process the texture information.

```
parseTextureSquares(sqSet) {
   let elm = this._getElm("TextureSquare");
   let i, j, x, y, w, h, r, c, t, sq;
   for (i = 0; i < elm.length; i++) {
    x = Number(elm.item(i).attributes.getNamedItem("PosX").value);
    y = Number(elm.item(i).attributes.getNamedItem("PosY").value);
    w = Number(elm.item(i).attributes.getNamedItem("Width").value);
    h = Number(elm.item(i).attributes.getNamedItem("Height").value);
    r = Number(elm.item(i).attributes.getNamedItem("Rotation").value);
    c = elm.item(i).attributes.getNamedItem("Color").value.split(" ");
    t = elm.item(i).attributes.getNamedItem("Texture").value;
    sq = new engine.TextureRenderable(t);
    // make sure color array contains numbers
    for (j = 0; j < 4; j++) {
          c[j] = Number(c[j]);
          sq.setColor(c);
          sq.getXform().setPosition(x, y);
          sq.getXform().setRotationInDegree(r); // In Degree
          sq.getXform().setSize(w, h);
          sqSet.push(sq);
    }
   }
}
```

Test BlueLevel with JPEGs

The modifications to `blue_level.js` are in the constructor, `load()`, `unload()`, `next()`, and `init()` functions where the texture images are loaded and unloaded and new `TextureRenderable` objects are parsed. We also rename the variables storing the path to our scene as that is a constant:

1. Edit `blue_level.js` and modify the constructor to define constants to represent the paths to the texture images:

```
class BlueLevel extends engine.Scene {
  constructor() {
      super();
      // scene file name
      this.kSceneFile = "assets/blue_level.xml";
      // textures: (Note: jpg does not support transparency)
      this.kPortal = "assets/minion_portal.jpg";
      this.kCollector = "assets/minion_collector.jpg";
      // all squares
      this.mSqSet = [];        // these are the Renderable objects
      // The camera to view the scene
      this.mCamera = null;
  }
```

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

```
load() {
   // load the scene file
   engine.xml.load(this.kSceneFile);
   // load the textures
   engine.texture.load(this.kPortal);
   engine.texture.load(this.kCollector);
}
```

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

```
unload() {
   // unload the scene file and loaded resources
   engine.xml.unload(this.kSceneFile);
   engine.texture.unload(this.kPortal);
   engine.texture.unload(this.kCollector);
}
```

4. Support loading of the next scene with the `next()` function:

```
next() {
  super.next();
  let nextLevel = new MyGame();  // load the next level
  nextLevel.start();
}
```

5. Modify step B in the `init()` function to parse the textured squares:

```
init() {
   let 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);
}
```

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

```
update() {
   // For this very simple game, let's move the first square
   let xform = this.mSqSet[0].getXform();
   let deltaX = 0.05;

   // Move right and swap over
   if (engine.input.isKeyPressed(engine.input.keys.Right)) {
    xform.incXPosBy(deltaX);
    if (xform.getXPos() > 30) {  // this is the right-bound of the window
          xform.setPosition(12, 60);
    }
   }
   // Step A: test for white square movement
   if (engine.input.isKeyPressed(engine.input.keys.Left)) {
    xform.incXPosBy(-deltaX);
    if (xform.getXPos() < 11) { // this is the left-boundary
          this.next();
    }
   }
   // continuously change texture tinting
   let c = this.mSqSet[1].getColor();
   let 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 listed code continuously increases and wraps the alpha value of the `mColor` variable in the `TextureRenderable` object. Recall that the values of this variable are passed to `TextureShader` and then loaded to the `uPixelColor` of `TextureFS` for tinting the texture map results.
c. As defined in the first `TextureSquare` element in the `blue_scene.xml` file, the color defined for the portal object is red. For this reason, when running 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. Edit `my_game.js`; modify the `MyGame` constructor to define texture image files and the variables for referencing the `TextureRenderable` objects:

```
class MyGame extends engine.Scene {
  constructor() {
      super();
      // textures:
      this.kPortal = "assets/minion_portal.png"; // with transparency
      this.kCollector = "assets/minion_collector.png";
      // The camera to view the scene
      this.mCamera = null;
      // the hero and the support objects
      this.mHero = null;
      this.mPortal = null;
      this.mCollector = null;
  }
```

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

```
load() {
   // loads the textures
   engine.texture.load(this.kPortal);
   engine.texture.load(this.kCollector);
}
```

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

```
unload() {
   // Game loop not running, unload all assets
   engine.texture.unload(this.kPortal);
   engine.texture.unload(this.kCollector);
}
```

4. Define the `next()` function to start the blue level:

```
next() {
  super.next();
  // starts the next level
  let nextLevel = new BlueLevel();  // next level to be loaded
  nextLevel.start();
}
```

5. Create and initialize the `TextureRenderables` objects in the `init()` function:

```
init() {
  // Step A: set up the cameras
  this.mCamera = new engine.Camera(
              vec2.fromValues(20, 60),   // position of the camera
              20,                // width of camera
              [20, 40, 600, 300] // viewport (X, Y, 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 engine.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 engine.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 engine.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 `resource_map`. 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 given code, all three functions refer to the same constants defined in the constructor.

6. 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.

```
draw() {
  // Step A: clear the canvas
  engine.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray

  // Step  B: Activate the drawing Camera
  this.mCamera.setViewAndCameraMatrix();

  // Step  C: Draw everything
  this.mPortal.draw(this.mCamera);
  this.mHero.draw(this.mCamera);
  this.mCollector.draw(this.mCamera);
}

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

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

  if (engine.input.isKeyPressed(engine.input.keys.Left)) {
   xform.incXPosBy(-deltaX);
   if (xform.getXPos() < 11) {  // this is the left-bound of the window
         this.next();
   }
  }

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

When running the example for this project, 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 that supports the alpha channel.

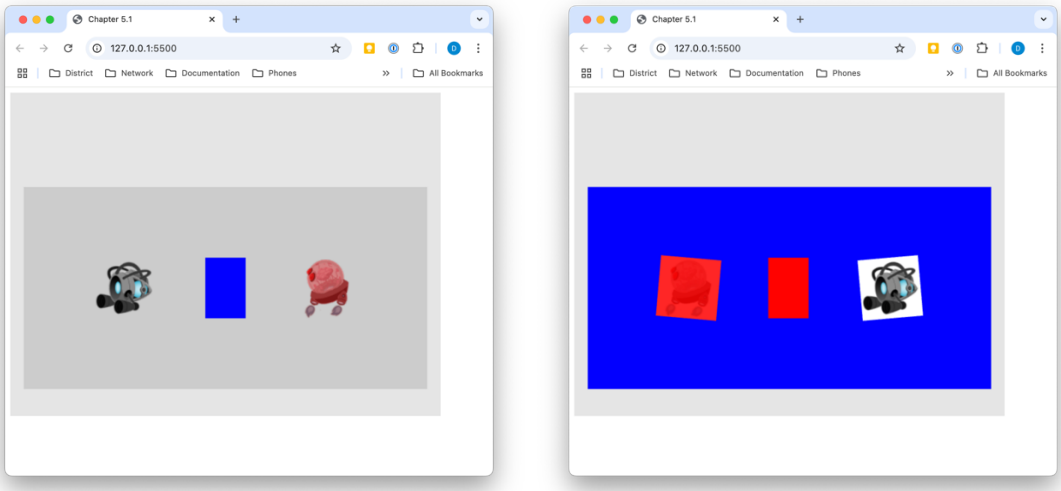

Figure 5-6. Output Textures

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 via WebGL. To help summarize the changes, Figure 5-7 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.

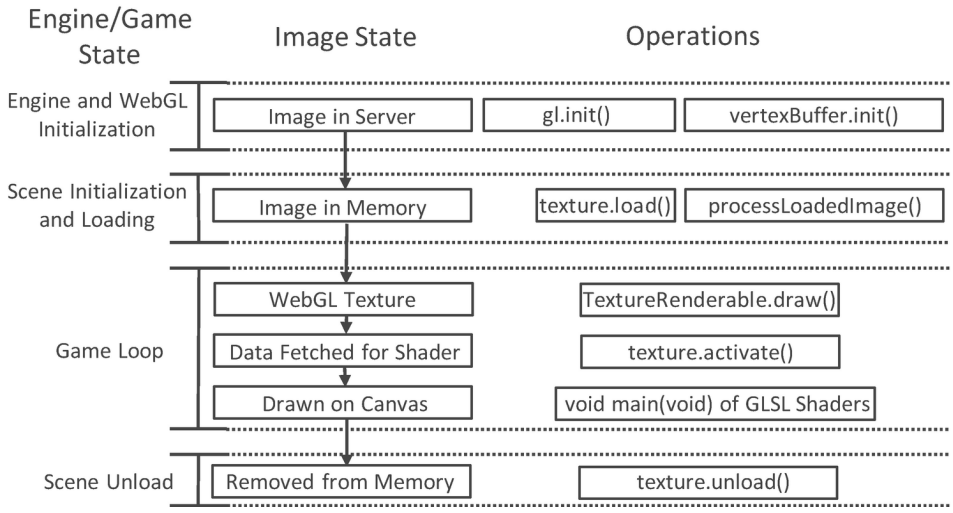

Figure 5-7. States of Image Files

The left column of Figure 5-7 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 the scene initialization, the `Scene.load()` function will invoke the `engine/resources/texture.load()` function to load the image and cause the loaded image to be processed by the `engine/resources/texture.processLoadedImage()` function into a corresponding WebGL texture to be stored in the GPU texture buffer. During the game loop cycle, the `TextureRenderable.draw()` function activates the appropriate WebGL texture via the `engine/resources/texture.activate()` function. This enables the corresponding GLSL fragment shader to sample from the correct texture during rendering. Finally, when a texture is no longer needed by the game, the `Scene.unload()` function will call `engine/resources/texture.unload()` to remove the loaded image from the system.

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 5-8 is a sprite sheet with 13 elements from 4 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.

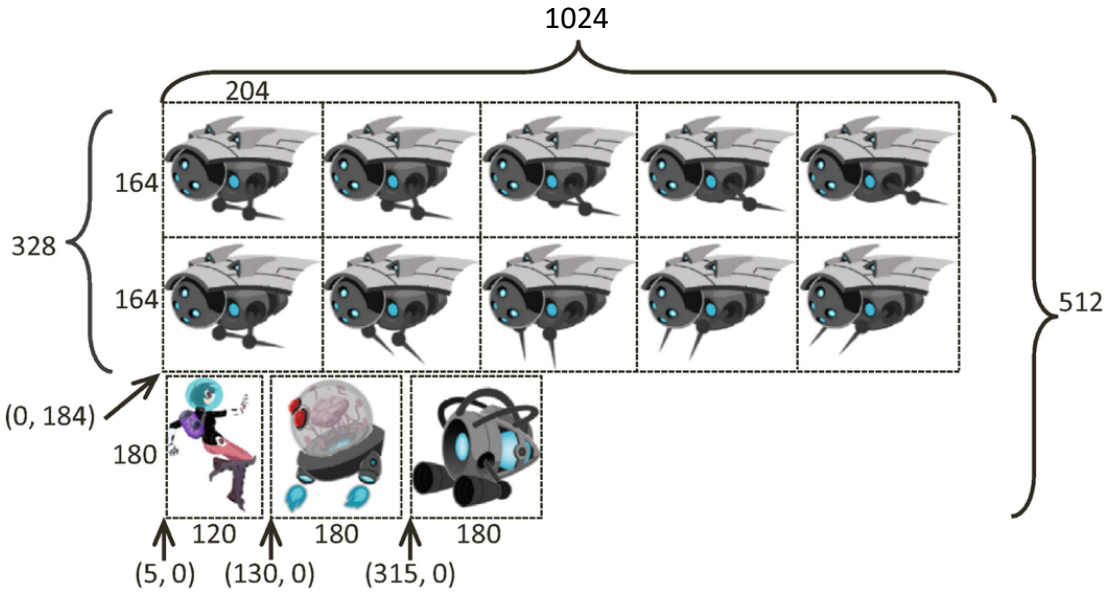

Figure 5-8. Sprite sheet composed of images of different objects

Sprite sheets are defined to optimize both memory and processing requirements. For example, recall that WebGL only supports textures that are defined by images with 2x × 2y resolutions. This requirement means that the Dye character at a resolution of 120x180 must be stored in a 128x256 (27 × 28) image in order for it to be created as a WebGL texture. Additionally, if the 13 elements of Figure 5-7 were stored as separate image files, then it would mean 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 5-9 focuses on the uv values of the collector minion in Figure 5-8, the rightmost sprite element on the third row. The top, center, and bottom rows of Figure 5-9 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.

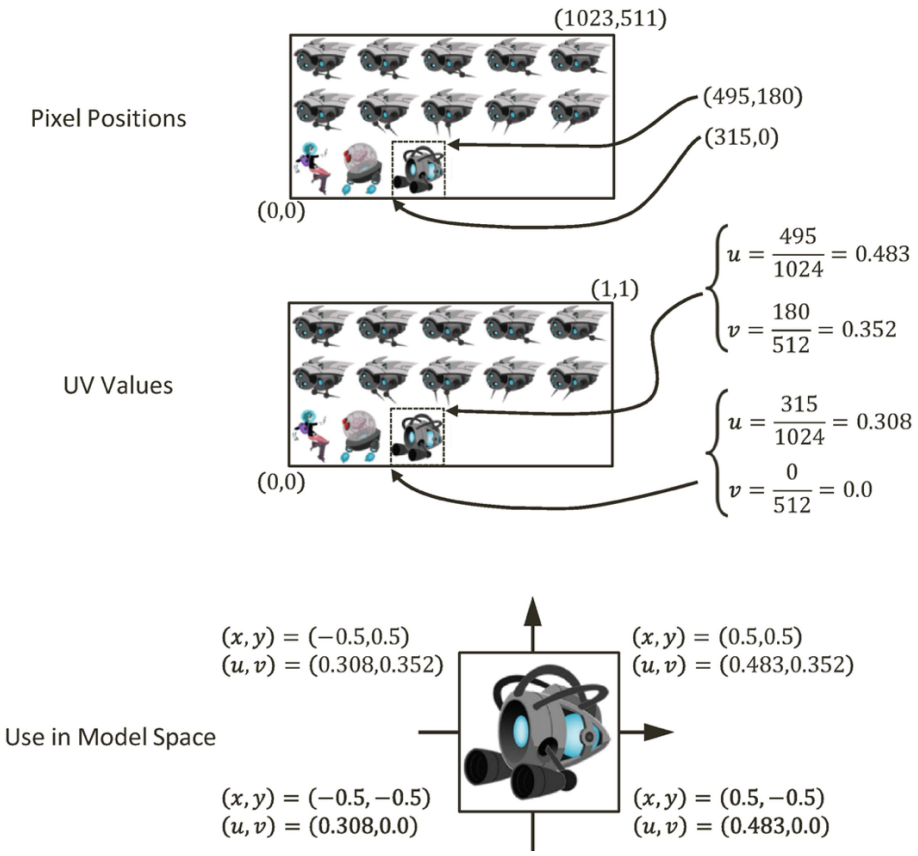

Figure 5-9. From pixel position to uv values used for mapping on geometry

Lab 5.2
The Sprite Shaders Project

This project demonstrates how to draw objects with sprite sheet elements by defining appropriate abstractions and classes.

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 of texture coordinates
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

See your instructor for obtaining the resource files needed (two images: `consolas-72.png` and `minion_sprite.png`) for this project and add them to the `assets` folder. No other assets are needed.

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 `getTexCoord()` function is statically defined in the `vertex_buffer.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 working with different subregions of an image and thus does not support working with sprite sheet elements. As illustrated in Figure 5-10, the example from this section overcomes this shortcoming by defining a per-object texture coordinate in the `SpriteShader` and `SpriteRenderable` objects. Notice that there are no new GLSL shaders defined since their functionality remains the same as `TextureVS/FS`.

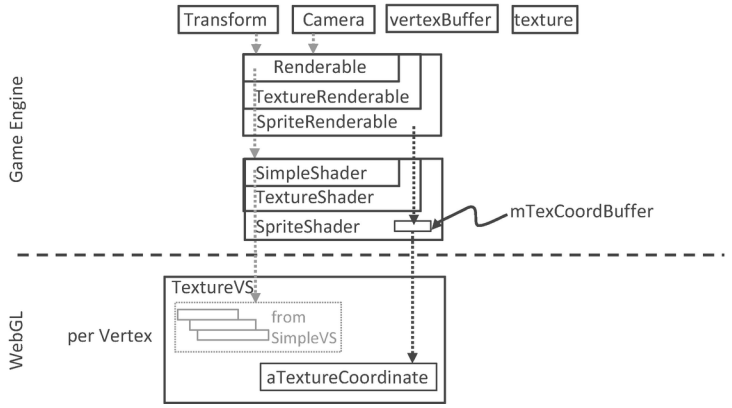

Figure 5-10. Defining a Texture Coordinate Buffer in `SpriteShader`

Interface GLSL Texture Shaders to the Engine with SpriteShader

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

1. Create a new file in the `src/engine/shaders` folder and name it `sprite_shader.js`.

2. Define the `SpriteShader` class and its constructor to extend the `TextureShader` class:

```
class SpriteShader extends TextureShader {
  constructor(vertexShaderPath, fragmentShaderPath) {
      // Call super class constructor
      super(vertexShaderPath, fragmentShaderPath);
      this.mTexCoordBuffer = null; // gl buffer with texture coordinate
      let initTexCoord = [
          1.0, 1.0,
          0.0, 1.0,
          1.0, 0.0,
          0.0, 0.0
      ];
      let gl = glSys.get();
      this.mTexCoordBuffer = gl.createBuffer();
      gl.bindBuffer(gl.ARRAY_BUFFER, this.mTexCoordBuffer);
      gl.bufferData(gl.ARRAY_BUFFER,
                    new Float32Array(initTexCoord), gl.DYNAMIC_DRAW);
                   // DYNAMIC_DRAW: says buffer content may change!
  }
... implementation to follow ...
}
```

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

TASK 5.2A Notice that there are no imports listed and `"use strict";` has not be included. Students should be able to add the necessary lines to make the program work. This **TASK** may not be indicated in every instance. **Hint** Look at the code and see what it calls. For example, the line `class SpriteShader extends TextureShader` tells us that we need to inport `TextureShader`.

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

```
setTextureCoordinate(texCoord) {
  let gl = glSys.get();
  gl.bindBuffer(gl.ARRAY_BUFFER, this.mTexCoordBuffer);
  gl.bufferSubData(gl.ARRAY_BUFFER, 0, new Float32Array(texCoord));
}
```

Note that `texCoord` parameter 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 to be 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 texture coordinate accessing function, `_getTexCoordBuffer()`, such that when the shader is activated, the locally allocated dynamic buffer is returned and not the global static buffer. Note that the `activate()` function is inherited from `TextureShader`.

```
_getTexCoordBuffer() {
  return this.mTexCoordBuffer;
}
```

5. Remember to export the class:

```
export default SpriteShader;
```

SpriteRenderable Class

Similar to the `Renderable` class (which are shaded with `SimpleShader`) and `TextureRenderable` class (which are shaded with `TextureShader`), a corresponding `SpriteRenderable` class 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 `sprite_renderable.js`.

2. Define an enumerated data type (similar to the Java `enum`) with values that identify corresponding offset positions of a WebGL texture coordinate specification array:

```
// texture coordinate array is an array of 8 floats where elements:
//  [0] [1]: is u/v coordinate 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
const eTexCoordArrayIndex = Object.freeze({
  eLeft: 2,
  eRight: 0,
  eTop: 1,
  eBottom: 5
});
```

Note An enumerated data type has a name that begins with an "e", as in `eTexCoordArrayIndex`.

3. Define the `SpriteRenderable` class and constructor to extend from the `TextureRenderable` class. Notice that the four instance variables, `mElmLeft`, `mElmRight`, `mElmTop`, and `mElmBottom`, together identify a subregion within the Texture Space. These are the bounds of a sprite sheet element.

```
class SpriteRenderable extends TextureRenderable {
  constructor(myTexture) {
      super(myTexture);
      super._setShader(shaderResources.getSpriteShader());
      // sprite coordinate
      this.mElmLeft = 0.0;   // texture coordinate bound
      this.mElmRight = 1.0;  //   0-left, 1-right
      this.mElmTop = 1.0;    //   1-top   0-bottom
      this.mElmBottom = 0.0; // of image
  }
... implementation to follow ...
}
```

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

```
// specify element region by texture coordinate (between 0 to 1)
setElementUVCoordinate(left, right, bottom, top) {
  this.mElmLeft = left;
  this.mElmRight = right;
  this.mElmBottom = bottom;
  this.mElmTop = top;
}
// element region defined pixel positions (0 to image resolutions)
setElementPixelPositions(left, right, bottom, top) {
  let texInfo = texture.get(this.mTexture);
  // entire image width, height
  let imageW = texInfo.mWidth;
  let imageH = texInfo.mHeight;
  this.mElmLeft = left / imageW;
```

```
    this.mElmRight = right / imageW;
    this.mElmBottom = bottom / imageH;
    this.mElmTop = top / imageH;
  }
```

Note that the `setElementPixelPositions()` function converts from pixel to texture coordinate 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 to the WebGL context:

```
getElementUVCoordinateArray() {
  return [
      this.mElmRight,  this.mElmTop,          // x,y of top-right
      this.mElmLeft,   this.mElmTop,
      this.mElmRight,  this.mElmBottom,
      this.mElmLeft,   this.mElmBottom
  ];
}
```

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

```
draw(camera) {
   // set the current texture coordinate
   // activate the texture
   this.mShader.setTextureCoordinate(this.getElementUVCoordinateArray());
   super.draw(camera);
}
```

7. Finally, remember to export the class and the defined enumerated type:

```
export default SpriteRenderable;
export {eTexCoordArrayIndex}
```

Facilitate Sharing with shader_resources

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

1. In the `engine/core/shader_resources.js` file, import `SpriteShader`, add a variable for storing, and define the corresponding getter function to access the shared `SpriteShader` instance:

```
import SpriteShader from "../shaders/sprite_shader.js";
let mSpriteShader = null;
function getSpriteShader() { return mSpriteShader; }
```

2. Modify the `createShaders()` function to create the `SpriteShader`:

```
function createShaders() {
  mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);
  mTextureShader = new TextureShader(kTextureVS, kTextureFS);
  mSpriteShader = new SpriteShader(kTextureVS, kTextureFS);
}
```

Notice that the `SpriteShader` actually wraps over the existing GLSL shaders defined in the `texture_vs.glsl` and `texture_fs.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.

3. Update the `cleanUp()` function for proper release of resources:

```
function cleanUp() {
  mConstColorShader.cleanUp();
  mTextureShader.cleanUp();
  mSpriteShader.cleanUp();
  ... identical to previous code ...
}
```

4. Make sure to export the new functionality:

```
export {init, cleanUp,
  getConstColorShader, getTextureShader, getSpriteShader}
```

Export New Functionality to the Client

The last step in integrating sprite element functionality into the engine involves modifying the engine access file, `index.js`. Edit `index.js` and add in the following import and export statements to grant client access to `SpriteRenderable` and `eTexCoordArrayIndex`, the enumerated data type for accessing the WebGL texture coordinate array.

```
// renderables
import Renderable from "./renderables/renderable.js";
import TextureRenderable from "./renderables/texture_renderable.js";
import SpriteRenderable from "./renderables/sprite_renderable.js";
```

```
import { eTexCoordArrayIndex } from "./renderables/sprite_renderable.js";
... identical to previous code ...
export default {
  ... identical to previous code ...
  // Renderables
  Renderable, TextureRenderable, SpriteRenderable,
  // constants
  eTexCoordArrayIndex,
  // functions
  init, cleanUp, clearCanvas
}
```

Testing the SpriteRenderable

There are two important functionalities of sprite elements and texture coordinate 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 `my_game.js` file.

1. The constructing, loading, unloading, and drawing of `MyGame` are similar to previous examples.

```
constructor() {
     super();
     // textures:
     this.kFontImage = "assets/consolas-72.png";
     this.kMinionSprite = "assets/minion_sprite.png";

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

     // the hero and the support objects
     this.mHero = null;
     this.mPortal = null;
     this.mCollector = null;
     this.mFontImage = null;
     this.mMinion = null;
 }

 load() {
     // loads the textures
     engine.texture.load(this.kFontImage);
     engine.texture.load(this.kMinionSprite);
 }

 unload() {
     engine.texture.unload(this.kFontImage);
     engine.texture.unload(this.kMinionSprite);
 }
```

2. Modify the `init()` function as follows.

```
init() {
  // Step A: set up the cameras
  this.mCamera = new engine.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 engine.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 engine.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.setElementUVCoordinate(0.308, 0.483, 0, 0.352);

  // Step C: Create the font and minion images using sprite
  this.mFontImage = new engine.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 engine.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 hero object with texture from lower-left corner
  this.mHero = new engine.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 A, notice that in step B both `mPortal` and `mCollector` are created based on the same image, `kMinionSprite`, with the respective `setElementPixelPositions()` and `setElementUVCoordinate()` 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 the defaults where the texture image will cover the entire geometry.
c. Similar to step B, step D 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 `draw()` function will draw the five objects created in the `init()` function.

```
draw() {
... identical to previous code ...

    // Step  C: Draw everything
    this.mPortal.draw(this.mCamera);
    this.mCollector.draw(this.mCamera);
    this.mHero.draw(this.mCamera);
    this.mFontImage.draw(this.mCamera);
    this.mMinion.draw(this.mCamera);
}
```

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

```
update() {
  // let's only allow the movement of hero,
  let deltaX = 0.05;
  let xform = this.mHero.getXform();
  // Support hero movements
  if (engine.input.isKeyPressed(engine.input.keys.Right)) {
      xform.incXPosBy(deltaX);
      if (xform.getXPos() > 30) { // right-bound of the window
          xform.setPosition(12, 60);
      }
  }
  if (engine.input.isKeyPressed(engine.input.keys.Left)) {
      xform.incXPosBy(-deltaX);
      if (xform.getXPos() < 11) {  // left-bound of the window
          xform.setXPos(20);
      }
  }
  // continuously change texture tinting
  let c = this.mPortal.getColor();
  let ca = c[3] + deltaX;
  if (ca > 1) {
      ca = 0;
  }
  c[3] = ca;

  // New update code for changing the sub-texture regions being shown"
  let 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
    let texCoord = this.mFontImage.getElementUVCoordinateArray();
    // The 8 elements:
    //      mTexRight,  mTexTop,          // x,y of top-right
    //      mTexLeft,   mTexTop,
    //      mTexRight,  mTexBottom,
    //      mTexLeft,   mTexBottom
    let b = texCoord[engine.eTexCoordArrayIndex.eBottom] + deltaT;
    let r = texCoord[engine.eTexCoordArrayIndex.eRight] - deltaT;
    if (b > 1.0) {
        b = 0;
    }
    if (r < 0) {
        r = 1.0;
    }
    this.mFontImage.setElementUVCoordinate(
        texCoord[engine.eTexCoordArrayIndex.eLeft],
        r,
        b,
        texCoord[engine.eTexCoordArrayIndex.eTop]
    );
    //
    // The minion image:
    // For minion: zoom to the bottom right corner by changing top left
    texCoord = this.mMinion.getElementUVCoordinateArray();
    // The 8 elements:
    //      mTexRight,  mTexTop,          // x,y of top-right
    //      mTexLeft,   mTexTop,
    //      mTexRight,  mTexBottom,
    //      mTexLeft,   mTexBottom
    let t = texCoord[engine.eTexCoordArrayIndex.eTop] - deltaT;
    let l = texCoord[engine.eTexCoordArrayIndex.eLeft] + deltaT;
    if (l > 0.5) {
        l = 0;
    }
    if (t < 0.5) {
        t = 1.0;
    }
    this.mMinion.setElementUVCoordinate(
        l,
        texCoord[engine.eTexCoordArrayIndex.eRight],
        texCoord[engine.eTexCoordArrayIndex.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

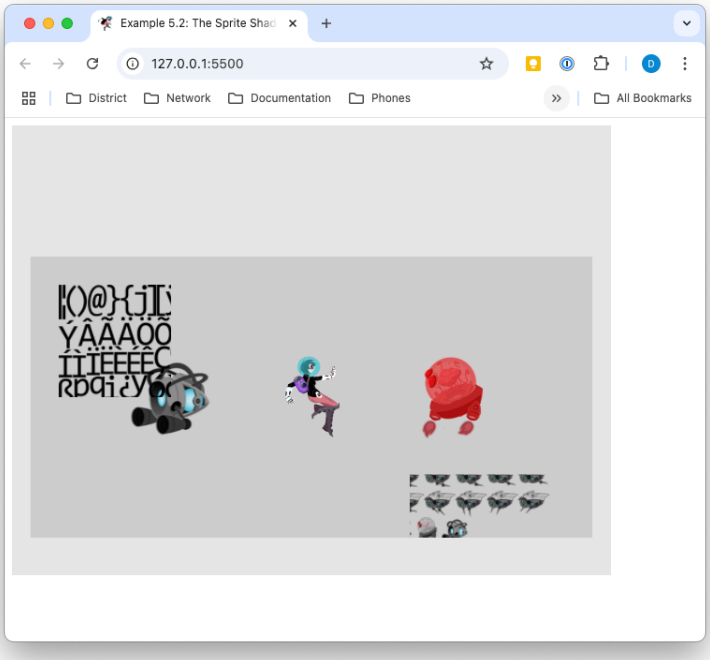

Figure 5-11. SpriteRenderables

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 or columns. For example, in Figure 5-12 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 rows. It is also possible for an animated sprite sheet to define animations that are along columns. 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 5-12. An animated sprite sheet organized into two rows representing two animated sequences of the same object

Figure 5-13 shows that to achieve the animated effect of an object retracting its spikes toward the right, as depicted by the top row of Figure 5-12, 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. 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.

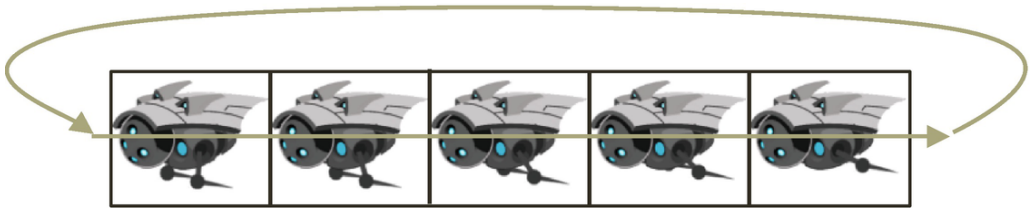

Figure 5-13. A sprite animation sequence that loops

Lab 5.3
The Sprite Animation Project

This project demonstrates how to work with an animated sprite sheet and generate continuous sprite animations.

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 movements 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 understand 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 Class

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. 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. Animated sprite elements organized along a column are not supported.

1. Create a new file in the `src/engine/renderables` folder and name it `sprite_animate_renderable.js`.

2. Define an enumerated data type for the three different sequences to animate:

```
// Assumption: first sprite is always the leftmost element.
const eAnimationType = Object.freeze({
  eRight: 0, // from left to right, when ended, start from left again
  eLeft: 1,  // from right animate left-wards,
  eSwing: 2  // left to right, then, right to left
});
```

The `eAnimationType` enum defines three modes for animation:

a. `eRight` 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. `eLeft` is the reverse of `eRight`; it starts from the right, animates toward the left, and continues by starting from the rightmost element after reaching the leftmost element.
c. `eSwing` is a continuous loop from left to right and then from right to left.

3. Define the `SpriteAnimateRenderable` class to extend from `SpriteRenderable` and define the constructor:

```
class SpriteAnimateRenderable extends SpriteRenderable {
  constructor(myTexture) {
      super(myTexture);
      super._setShader(shaderResources.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;  // image top corner (from SpriteRenderable)
      this.mElmWidth = 1.0;
      this.mElmHeight = 1.0;
      this.mWidthPadding = 0.0;
      this.mNumElems = 1;   // number of elements in an animation
      // per animation settings
      this.mUpdateInterval = 1;   // how often to advance
      this.mAnimationType = eAnimationType.eRight;
      this.mCurrentAnimAdvance = -1;
      this.mCurrentElm = 0;
      this._initAnimation();
  }
  ... implementation to follow ...
}
```

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 coordinate for each sprite element when the elements are ordered by rows and columns. Note that all coordinates are in texture coordinate space (0 to 1).
b. The second set stores information on how to animate, the `mAnimationType` of left, right, or swing, and how much time, `mUpdateInterval`, to wait before advancing to the next sprite element. This information can be changed during runtime to reverse, loop, or control the speed of a character's movement.
c. The third set, `mCurrentAnimAdvance` and `mCurrentElm`, describes offset for advancing and the current frame number. Both of these variables are in units of element

counts and are not designed to be accessed by the game programmer because they 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 the `_initAnimation()` function to compute the proper vales for `mCurrentAnimAdance` and `mCurrentElm` according to the current animation type:

```
_initAnimation() {
      // Currently running animation
      this.mCurrentTick = 0;
      switch (this.mAnimationType) {
      case eAnimationType.eRight:
          this.mCurrentElm = 0;
          this.mCurrentAnimAdvance = 1; // either 1 or -1
          break;
      case eAnimationType.eSwing:
          this.mCurrentAnimAdvance = -1 * this.mCurrentAnimAdvance;
          this.mCurrentElm += 2 * this.mCurrentAnimAdvance;
          break;
      case eAnimationType.eLeft:
          this.mCurrentElm = this.mNumElems - 1;
          this.mCurrentAnimAdvance = -1; // either 1 or -1
          break;
      }
      this._setSpriteElement();
  }
```

The `mCurrentElm` is the number of elements to offset from the leftmost, and `mCurrentAnimAdvance` records whether the `mCurrentElm` offset should be incremented (for rightward animation) or decremented (for leftward animation) during each update. The `_setSpriteElement()` function is called to set the uv values that correspond to the currently identified sprite element for displaying.

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

```
_setSpriteElement() {
  let left = this.mFirstElmLeft +
             (this.mCurrentElm * (this.mElmWidth+this.mWidthPadding));
  super.setElementUVCoordinate(left, left + this.mElmWidth,
                          this.mElmTop - this.mElmHeight, this.mElmTop);
}
```

The variable left is the left u value of `mCurrentElm` and is used to compute the right u value, with the assumption that all animation sequences are along the same row of sprite elements and that the top and bottom v values are constant where they do not change over a given animation sequence. These uv values are set to the super class `SpriteRenderable` for drawing.

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

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

7. Define a function for specifying a sprite animation sequence. The inputs to the function are in pixels and are converted to texture coordinates by dividing by the width and height of the image.

```
// Always set the leftmost element to be the first
setSpriteSequence(
  topPixel,   // offset from top-left
  leftPixel, // offset from top-left
  elmWidthInPixel,
  elmHeightInPixel,
  numElements,      // number of elements in sequence
  wPaddingInPixel  // left/right padding
) {
  let texInfo = texture.get(this.mTexture);
  // entire image width, height
  let imageW = texInfo.mWidth;
  let imageH = texInfo.mHeight;
  this.mNumElems = numElements;   // number of elements in animation
  this.mFirstElmLeft = leftPixel / imageW;
  this.mElmTop = topPixel / imageH;
  this.mElmWidth = elmWidthInPixel / imageW;
  this.mElmHeight = elmHeightInPixel / imageH;
  this.mWidthPadding = wPaddingInPixel / imageW;
  this._initAnimation();
}
```

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

```
setAnimationSpeed(tickInterval) {
```

```
    this.mUpdateInterval = tickInterval; }
incAnimationSpeed(deltaInterval) {
    this.mUpdateInterval += deltaInterval; }
```

9. Define a function to advance the animation for each game loop update:

```
updateAnimation() {
  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 `loop.loopOnce()` function ensures system-wide updates to occur at `kMPF` intervals even when frame rate lags. The game engine architecture ensures the `updateAnimation()` function calls are `kMPF` milliseconds apart.

10. Finally, remember to export the defined class and enumerated animation type:

```
export default SpriteAnimateRenderable;
export {eAnimationType}
```

TASK 5.3A Create the needed import statements at the top of the file.

Export New Functionality to the Client

The last step in integrating animated sprite element functionality into the engine involves modifying the engine access file, `index.js`. Edit `index.js` and add in the following import and export statements to grant client access to `SpriteAnimateRenderable` and `eAnimationType`:

```
// renderables
import Renderable from "./renderables/renderable.js";
  ... identical to previous code ...
import SpriteRenderable from "./renderables/sprite_renderable.js";
import SpriteAnimateRenderable from
     "./renderables/sprite_animate_renderable.js";
import { eTexCoordArrayIndex } from "./renderables/sprite_renderable.js";
import { eAnimationType } from "./renderables/sprite_animate_renderable.js";

  ... identical to previous code ...

export default {
   ... identical to previous code ...
   // Renderables
   Renderable, TextureRenderable,
   SpriteRenderable, SpriteAnimateRenderable,
   // constants
   eTexCoordArrayIndex, eAnimationType,
   // functions
   init, cleanUp, clearCanvas
}
```

Testing Sprite Animation

The test cases for the `SpriteAnimateRenderable` object must demonstrate the game programmer's control over the modes (left, right, swing) and speed of animation. The `MyGame` object is modified to accomplish these purposes. In the previous project, we created `mMinion` that displayed the entire minion sprite sheet in the bottom right corner. We are going to remove this `SpriteRenderable` and add two `SpriteAnimateRenderable` objects to animate.

1. The constructor of `MyGame` is similar to the previous example.

TASK 5.3B Modify the constructor to remove all references to `mMinion` and create variables for `mLeftMinion` and `mRightMinion`.

2. In the `draw()` function,

TASK 5.3C Modify the `draw()` function to remove all references to `mMinion` and add code for drawing `mLeftMinion` and `mRightMinion`.

3. Loading and unloading of `MyGame` are similar to the previous example and the details are not repeated.

4. In the `init()` function, replace the existing minion code to create and initialize the new `SpriteAnimateRenderable` objects between steps C and D:

```
init() {
  ... identical to previous code ...
  // The right minion
  this.mRightMinion = new engine.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 pixel positions: top: 512 left: 0
      204, 164, // widthxheight in pixels
      5,       // number of elements in this sequence
      0);      // horizontal padding in between
  this.mRightMinion.setAnimationType(engine.eAnimationType.eRight);
  this.mRightMinion.setAnimationSpeed(50);

  // the left minion
  this.mLeftMinion = new engine.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 pixel positions: top: 164 left: 0
      204, 164, // widthxheight in pixels
      5,       // number of elements in this sequence
      0);      // horizontal padding in between
  this.mLeftMinion.setAnimationType(engine.eAnimationType.eRight);
  this.mLeftMinion.setAnimationSpeed(50);
}
```

The `SpriteAnimateRenderable` objects are created in similar ways as `SpriteRenderable` objects with a sprite sheet as the texture parameter. 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.

5. The `update()` function must invoke the `SpriteAnimateRenderable` object's `updateAnimation()` function to advance the sprite animation. The following code will replace the `mMinion` code from the previous project:

```
update() {
  ... identical to previous code ...
  // remember to update the minion's animation
  this.mRightMinion.updateAnimation();
  this.mLeftMinion.updateAnimation();
  // Animate left on the sprite sheet
  if (engine.input.isKeyClicked(engine.input.keys.One)) {
```

```
            this.mRightMinion.setAnimationType(engine.eAnimationType.eLeft);
            this.mLeftMinion.setAnimationType(engine.eAnimationType.eLeft);
        }
        // swing animation
        if (engine.input.isKeyClicked(engine.input.keys.Two)) {
            this.mRightMinion.setAnimationType(engine.eAnimationType.eSwing);
            this.mLeftMinion.setAnimationType(engine.eAnimationType.eSwing);
        }
        // Animate right on the sprite sheet
        if (engine.input.isKeyClicked(engine.input.keys.Three)) {
            this.mRightMinion.setAnimationType(engine.eAnimationType.eRight);
            this.mLeftMinion.setAnimationType(engine.eAnimationType.eRight);
        }
        // decrease duration of each sprite element to speed up animation
        if (engine.input.isKeyClicked(engine.input.keys.Four)) {
            this.mRightMinion.incAnimationSpeed(-2);
            this.mLeftMinion.incAnimationSpeed(-2);
        }
        // increase duration of each sprite element to slow down animation
        if (engine.input.isKeyClicked(engine.input.keys.Five)) {
            this.mRightMinion.incAnimationSpeed(2);
            this.mLeftMinion.incAnimationSpeed(2);
        }
    }
```

TASK 5.3D There are a number of references to the blue level from the previous project. All of the related code for that level can be removed for this project.

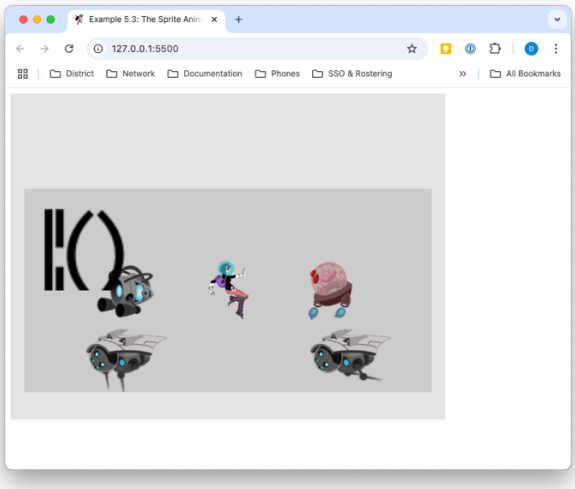

Figure 5-14. SpriteAnimateRenderables

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,

displaying 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 www.angelcode.com/products/bmfont/ for more information.

Lab 5.4
The Font Support Project

This project demonstrates how to draw text from a bitmap font using the `SpriteRenderable` object

The controls of the project are as follows:

Number keys 0, 1, 2, and 3: Select 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- and right-arrow keys: Move the hero left or right. The hero wraps if it exits the bounds.

TASK 5.4A This project will use the X and Y keys. Your `input.js` file does not currently support the entire alphabet. Modify the file to support all letters A thru Z.

The goals of the project are as follows:

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

Assets

You can obtain the following external resource files from your instructor. Create a folder called `fonts` in your `assets` folder.

1. In the `assets/fonts` folder, add 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.

2. Add the following to your `assets` folder: `consolas-72.png` and `minion_sprite.png`. These will be used in a similar way as in the previous project.

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 loading of these two types of files is already supported by the engine. The details of loading and storing fonts in the engine are encapsulated by a new engine component, `font`.

1. Create a new file in the `src/engine/resources` folder and name it `font.js`.

2. Import the resource management functionality from the xml module for loading the `.fnt` file and the texture module for the `.png` sprite image file, and define local constants for these file extensions:

```
import * as xml from "./xml.js";
import * as texture from "./texture.js";
let kDescExt = ".fnt";   // extension for the bitmap font description
let kImageExt = ".png";  // extension for the bitmap font image
```

3. Define a class for storing uv coordinate locations and the size associated with a character. This information can be computed based on the contents from the `.fnt` file.

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

      // nominal char 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 ratio
      this.mCharAspectRatio = 1;
  }
}
```

4. Define two functions to return proper extensions based on a path with no file extension. Note that `fontName` is a path to the font files but without any file extensions. For example, `assets/fonts/system-default-font` is the string and the two functions identify the two associated `.fnt` and `.png` files.

```
function descName(fontName) { return fontName+kDescExt;}
function imageName(fontName) { return fontName+kImageExt;}
```

5. Define the `load()` and `unload()` functions. Notice that two file operations are actually invoked in each: one for the `.fnt` and the second for the `.png` files.

```
function load(fontName) {
  xml.load(descName(fontName));
  texture.load(imageName(fontName));
}
function unload(fontName) {
  xml.unload(descName(fontName));
  texture.unload(imageName(fontName));
}
```

6. Define a function to inquire the loading status of a given font:

```
function has(fontName) {
  return texture.has(imageName(fontName)) &&
         xml.has(descName(fontName));
}
```

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

```
function getCharInfo(fontName, aChar) {
  let returnInfo = null;
  let fontInfo = xml.get(descName(fontName));
  let commonPath = "font/common";
  let commonInfo = fontInfo.evaluate(commonPath, fontInfo, null,
          XPathResult.ANY_TYPE, null);
  commonInfo = commonInfo.iterateNext();
  if (commonInfo === null) {
      return returnInfo;
  }
  let charHeight = commonInfo.getAttribute("base");

  let charPath = "font/chars/char[@id=" + aChar + "]";
  let charInfo = fontInfo.evaluate(charPath, fontInfo, null,
          XPathResult.ANY_TYPE, null);
  charInfo = charInfo.iterateNext();

  if (charInfo === null) {
      return returnInfo;
  }
```

```
    returnInfo = new CharacterInfo();
    let texInfo = texture.get(imageName(fontName));
    let leftPixel = Number(charInfo.getAttribute("x"));
    let rightPixel = leftPixel + Number(charInfo.getAttribute("width")) - 1;
    let topPixel = (texInfo.mHeight - 1) -
            Number(charInfo.getAttribute("y"));
    let bottomPixel = topPixel - Number(charInfo.getAttribute("height")) + 1;

    // texture coordinate information
    returnInfo.mTexCoordLeft = leftPixel / (texInfo.mWidth - 1);
    returnInfo.mTexCoordTop = topPixel / (texInfo.mHeight - 1);
    returnInfo.mTexCoordRight = rightPixel / (texInfo.mWidth - 1);
    returnInfo.mTexCoordBottom = bottomPixel / (texInfo.mHeight - 1);

    // relative character size
    let charWidth = charInfo.getAttribute("xadvance");
    returnInfo.mCharWidth = charInfo.getAttribute("width") / charWidth;
    returnInfo.mCharHeight = charInfo.getAttribute("height") / charHeight;
    returnInfo.mCharWidthOffset = charInfo.getAttribute("xoffset") /
            charWidth;
    returnInfo.mCharHeightOffset = charInfo.getAttribute("yoffset") /
            charHeight;
    returnInfo.mCharAspectRatio = charWidth / charHeight;

    return returnInfo;
}
```

Details of decoding and extracting information for a 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 www.angelcode.com/products/bmfont/doc/file_format.html.

8. Finally, remember to export the functions from this module:

```
  export {has, load, unload,
      imageName, descName,
      CharacterInfo,
      getCharInfo
  }
```

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. To accomplish this, an engine utility should be defined to load and initialize default resources to be shared with the game developer. Recall that the `shader_resources` module in the `src/engine/core` folder is defined to support engine-wide sharing of shaders. This pattern can be duplicated for sharing of default resources with the client. A

default_resources module can be defined in the src/engine/resources folder to accomplish this sharing.

1. Create a file in the src/engine/resources folder and name it default_resources.js, import functionality from the font and resource_map modules, and define a constant string and its getter function for the path to the default system font:

```
import * as font from "./font.js";
import * as map from "../core/resource_map.js";
// Default font
let kDefaultFont = "assets/fonts/system_default_font";
function getDefaultFontName() { return kDefaultFont; }
```

2. Define an init() function to issue the default system font loading request in a JavaScript Promise and append the Promise to the array of outstanding load requests in the resource_map. Recall that the loop.start() function in the loop module waits for the fulfillment of all resource_map loading promises before starting the game loop. For this reason, as in the case of all other asynchronously loaded resources, by the time the game loop begins, the default system font will have been properly loaded.

```
function init() {
  let loadPromise = new Promise(
    async function (resolve) {
         await Promise.all([
             font.load(kDefaultFont)
         ]);
         resolve();
    }).then(
         function resolve() { /* nothing to do for font */ }
    );
  map.pushPromise(loadPromise);
}
```

3. Define the cleanUp() function to release all allocated resources, in this case, unload the font:

```
// unload all resources

function cleanUp() {
  font.unload(kDefaultFont);
}
```

4. Lastly, remember to export all defined functionality:

```
export {
  init, cleanUp,
  // default system font name: this is guaranteed to be loaded
  getDefaultFontName
}
```

Defining a FontRenderable Object to Draw Texts

The defined font module is capable of loading font files and extracting per-character uv coordinate and size information. With this functionality, the drawing of a text string can be accomplished by identifying each character in the string, retrieving the corresponding texture mapping information, and rendering the character using a `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 `font_renderable.js`.

2. Define the `FontRenderable` class and its constructor to accept a string as its parameter:

```
"use strict";

import Transform from "../transform.js";
import SpriteRenderable from "./sprite_renderable.js";
import * as defaultResources from "../resources/default_resources.js";
import * as font from "../resources/font.js";

class FontRenderable {
  constructor(aString) {
      this.mFontName = defaultResources.getDefaultFontName();
      this.mOneChar = new SpriteRenderable(
                          font.imageName(this.mFontName));
      this.mXform = new Transform(); // to move this object around
      this.mText = aString;
  }
  ... implementation to follow ...
}
```

   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:

```
draw(camera) {
  // we will draw the text string by calling mOneChar for each of the
  // chars in the mText string.
  let widthOfOneChar = this.mXform.getWidth() / this.mText.length;
  let heightOfOneChar = this.mXform.getHeight();
  let yPos = this.mXform.getYPos();
  // center position of the first char
  let xPos = this.mXform.getXPos() -
             (widthOfOneChar / 2) + (widthOfOneChar * 0.5);
  let charIndex, aChar, charInfo, xSize, ySize, xOffset, yOffset;
  for (charIndex = 0; charIndex < this.mText.length; charIndex++) {
      aChar = this.mText.charCodeAt(charIndex);
      charInfo = font.getCharInfo(this.mFontName, 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(camera);
      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 performs the following operations:

a. Extracts each character in the string
b. Calls the `getCharInfo()` function to receive the character's uv values and size 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 size 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:

```
getXform() { return this.mXform; }
getText() { return this.mText; }
setText(t) {
  this.mText = t;
  this.setTextHeight(this.getXform().getHeight());
}
getFontName() { return this.mFontName; }
setFontName(f) {
  this.mFontName = f;
  this.mOneChar.setTexture(font.imageName(this.mFontName));
}
setColor(c) { this.mOneChar.setColor(c); }
getColor() { return this.mOneChar.getColor(); }
```

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

```
setTextHeight(h) {
  let charInfo = font.getCharInfo(this.mFontName, "A".charCodeAt(0));
  let 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.

6. Finally, remember to export the defined class:

```
export default FontRenderable;
```

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

Initialize, Cleaning, and Export Font Functionality

As in all engine functionality, it is important to update the engine access file, `index.js`, to grant access to the game developer. In this case, it is also essential to initialize and clean up resources associated with the default system font.

1. Edit `index.js` to import functionality from the font and `default_resources` modules and the `FontRenderable` class:

```
// resources
import * as audio from "./resources/audio.js";
import * as text from "./resources/text.js";
import * as xml from "./resources/xml.js";
import * as texture from "./resources/texture.js";
import * as font from "./resources/font.js";
import * as defaultResources from "./resources/default_resources.js";
... identical to previous code ...
// renderables
import Renderable from "./renderables/renderable.js";
import SpriteRenderable from "./renderables/sprite_renderable.js";
import SpriteAnimateRenderable from
                  "./renderables/sprite_animate_renderable.js";
import FontRenderable from "./renderables/font_renderable.js";
... identical to previous code ...
```

2. Add default resources initialization and cleanup in the engine `init()` and `cleanUp()` functions:

```
function init(htmlCanvasID) {
  glSys.init(htmlCanvasID);
  vertexBuffer.init();
  input.init();
  audio.init();
  shaderResources.init();
  defaultResources.init();
}
function cleanUp() {
  loop.cleanUp();
  shaderResources.cleanUp();
  defaultResources.cleanUp();
  audio.cleanUp();
  input.cleanUp();
  vertexBuffer.cleanUp();
  glSys.cleanUp();
}
```

3. Remember to export the newly defined functionality:

```
export default {
  // resource support
  audio, text, xml, texture, font, defaultResources,
  ... identical to previous code ...
  // Renderables
  Renderable, TextureRenderable,
  SpriteRenderable, SpriteAnimateRenderable, FontRenderable,
  ... identical to previous code ...
}
```

Testing Fonts

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

1. In the `my_game.js` file, modify the constructor to define corresponding variables for printing the messages, and modify the `draw()` function to draw all objects accordingly.

Be Aware: `mLeftMinion` and `mRightMinion` are now condensed back into a single sprite called `mMinion`. You will need to clean up all references for the other methods in `my_game.js`.

```
constructor() {
    super();
    // textures:
    this.kFontImage = "assets/consolas-72.png";
    this.kMinionSprite = "assets/minion_sprite.png";

    // the fonts
    this.kFontCon16 = "assets/fonts/consolas-16";
    this.kFontCon24 = "assets/fonts/consolas-24";
    this.kFontCon32 = "assets/fonts/consolas-32";
    this.kFontCon72 = "assets/fonts/consolas-72";
    this.kFontSeg96 = "assets/fonts/segment7-96";

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

    // the hero and the support objects
    this.mHero = null;
    this.mFontImage = null;
    this.mMinion = null;

    this.mTextSysFont = null;
    this.mTextCon16 = null;
    this.mTextCon24 = null;
    this.mTextCon32 = null;
    this.mTextCon72 = null;
    this.mTextSeg96 = null;

    this.mTextToWork = null;
}

draw() {
    // Step A: clear the canvas
    engine.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray
    // Step  B: Activate the drawing Camera
    this.mCamera.setViewAndCameraMatrix();
    // Step  C: Draw everything
    this.mHero.draw(this.mCamera);
    this.mFontImage.draw(this.mCamera);
    this.mMinion.draw(this.mCamera);
    // drawing the text output
```

```
        this.mTextSysFont.draw(this.mCamera);
        this.mTextCon16.draw(this.mCamera);
        this.mTextCon24.draw(this.mCamera);
        this.mTextCon32.draw(this.mCamera);
        this.mTextCon72.draw(this.mCamera);
        this.mTextSeg96.draw(this.mCamera);
    }
```

2. Modify the `load()` function to load the textures and fonts. Once again, notice that the font paths, for example, assets/fonts/consolas-16, do not include file name extensions. Recall that this path will be appended with .fnt and .png, where two separate files will be loaded to support the drawing of fonts.

```
load() {
    // Step A: loads the textures
    engine.texture.load(this.kFontImage);
    engine.texture.load(this.kMinionSprite);
    // Step B: loads all the fonts
    engine.font.load(this.kFontCon16);
    engine.font.load(this.kFontCon24);
    engine.font.load(this.kFontCon32);
    engine.font.load(this.kFontCon72);
    engine.font.load(this.kFontSeg96);
}
```

3. Modify the `unload()` function to unload the textures and fonts:

```
unload() {
    engine.texture.unload(this.kFontImage);
    engine.texture.unload(this.kMinionSprite);
    // unload the fonts
    engine.font.unload(this.kFontCon16);
    engine.font.unload(this.kFontCon24);
    engine.font.unload(this.kFontCon32);
    engine.font.unload(this.kFontCon72);
    engine.font.unload(this.kFontSeg96);
}
```

4. Define a private `_initText()` function to set the color, location, and height of a `FontRenderable` object. Modify the `init()` function to set up the proper WC system and initialize the fonts. Notice the calls to `setFont()` function to change the font type for each message.

```
_initText(font, posX, posY, color, textH) {
    font.setColor(color);
    font.getXform().setPosition(posX, posY);
```

```
  font.setTextHeight(textH);
}
init() {
  // Step A: set up the cameras
  this.mCamera = new engine.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([0.8, 0.8, 0.8, 1]);
  // sets the background to gray
  // Step B: Create the font and minion images using sprite
  this.mFontImage = new engine.SpriteRenderable(this.kFontImage);
  this.mFontImage.setColor([1, 1, 1, 0]);
  this.mFontImage.getXform().setPosition(15, 50);
  this.mFontImage.getXform().setSize(20, 20);
  // The minion
  this.mMinion = new engine.SpriteAnimateRenderable(
                           this.kMinionSprite);
  this.mMinion.setColor([1, 1, 1, 0]);
  this.mMinion.getXform().setPosition(15, 25);
  this.mMinion.getXform().setSize(24, 19.2);
  this.mMinion.setSpriteSequence(512, 0,  // first element: top, left
      204, 164,    // widthxheight in pixels
      5,           // number of elements in this sequence
      0);          // horizontal padding in between
  this.mMinion.setAnimationType(engine.eAnimationType.eSwing);
  this.mMinion.setAnimationSpeed(15);
  // show each element for mAnimSpeed updates
  // Step D: Create hero object with texture from lower-left corner
  this.mHero = new engine.SpriteRenderable(this.kMinionSprite);
  this.mHero.setColor([1, 1, 1, 0]);
  this.mHero.getXform().setPosition(35, 50);
  this.mHero.getXform().setSize(12, 18);
  this.mHero.setElementPixelPositions(0, 120, 0, 180);
  // Create the fonts
  this.mTextSysFont = new engine.FontRenderable("System Font: in Red");
  this._initText(this.mTextSysFont, 50, 60, [1, 0, 0, 1], 3);
  this.mTextCon16 = new engine.FontRenderable("Consolas 16: in black");
  this.mTextCon16.setFontName(this.kFontCon16);
  this._initText(this.mTextCon16, 50, 55, [0, 0, 0, 1], 2);
  this.mTextCon24 = new engine.FontRenderable("Consolas 24: in black");
  this.mTextCon24.setFontName(this.kFontCon24);
  this._initText(this.mTextCon24, 50, 50, [0, 0, 0, 1], 3);
  this.mTextCon32 = new engine.FontRenderable("Consolas 32: in white");
  this.mTextCon32.setFontName(this.kFontCon32);
  this._initText(this.mTextCon32, 40, 40, [1, 1, 1, 1], 4);
  this.mTextCon72 = new engine.FontRenderable("Consolas 72: in blue");
  this.mTextCon72.setFontName(this.kFontCon72);
  this._initText(this.mTextCon72, 30, 30, [0, 0, 1, 1], 6);
  this.mTextSeg96 = new engine.FontRenderable("Segment7-92");
  this.mTextSeg96.setFontName(this.kFontSeg96);
  this._initText(this.mTextSeg96, 30, 15, [1, 1, 0, 1], 7);
  this.mTextToWork = this.mTextCon16;
}
```

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

```
update() {
   // let's only allow the movement of hero,
   let deltaX = 0.5;
   let xform = this.mHero.getXform();

   // Support hero movements
   if (engine.input.isKeyPressed(engine.input.keys.Right)) {
    xform.incXPosBy(deltaX);
    if (xform.getXPos() > 100) { // the right-bound of the window
          xform.setPosition(0, 50);
    }
   }

   if (engine.input.isKeyPressed(engine.input.keys.Left)) {
    xform.incXPosBy(-deltaX);
    if (xform.getXPos() < 0) {  // the left-bound of the window
          xform.setPosition(100, 50);
    }
   }

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

   // <editor-fold desc="The font image:">
   // zoom into the texture by updating texture coordinate
   // For font: zoom to the upper left corner by changing bottom right
   let texCoord = this.mFontImage.getElementUVCoordinateArray();
   // The 8 elements:
   //      mTexRight,  mTexTop,          // x,y of top-right
   //      mTexLeft,   mTexTop,
   //      mTexRight,  mTexBottom,
   //      mTexLeft,   mTexBottom
   let b = texCoord[engine.eTexCoordArrayIndex.eBottom] + deltaT;
   let r = texCoord[engine.eTexCoordArrayIndex.eRight] - deltaT;
   if (b > 1.0) {
    b = 0;
   }
   if (r < 0) {
    r = 1.0;
   }
   this.mFontImage.setElementUVCoordinate(
          texCoord[engine.eTexCoordArrayIndex.eLeft],
          r,
          b,
          texCoord[engine.eTexCoordArrayIndex.eTop]
   );
   //

   // remember to update this.mMinion's animation
   this.mMinion.updateAnimation();

   // interactive control of the display size
```

```
    // choose which text to work on
    if (engine.input.isKeyClicked(engine.input.keys.Zero)) {
     this.mTextToWork = this.mTextCon16;
    }
    if (engine.input.isKeyClicked(engine.input.keys.One)) {
     this.mTextToWork = this.mTextCon24;
    }
    if (engine.input.isKeyClicked(engine.input.keys.Two)) {
     this.mTextToWork = this.mTextCon32;
    }
    if (engine.input.isKeyClicked(engine.input.keys.Three)) {
     this.mTextToWork = this.mTextCon72;
    }

    let deltaF = 0.005;
    if (engine.input.isKeyPressed(engine.input.keys.Up)) {
     if (engine.input.isKeyPressed(engine.input.keys.X)) {
           this.mTextToWork.getXform().incWidthBy(deltaF);
     }
     if (engine.input.isKeyPressed(engine.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 (engine.input.isKeyPressed(engine.input.keys.Down)) {
     if (engine.input.isKeyPressed(engine.input.keys.X)) {
           this.mTextToWork.getXform().incWidthBy(-deltaF);
     }
     if (engine.input.isKeyPressed(engine.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 listed code shows that you can perform the following operations during runtime:

a. Select which `FontRenderable` object to work with based on keyboard 0 to 3 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 Font Support project to modify the size of each of the displayed font message and to move the hero toward the left and right.

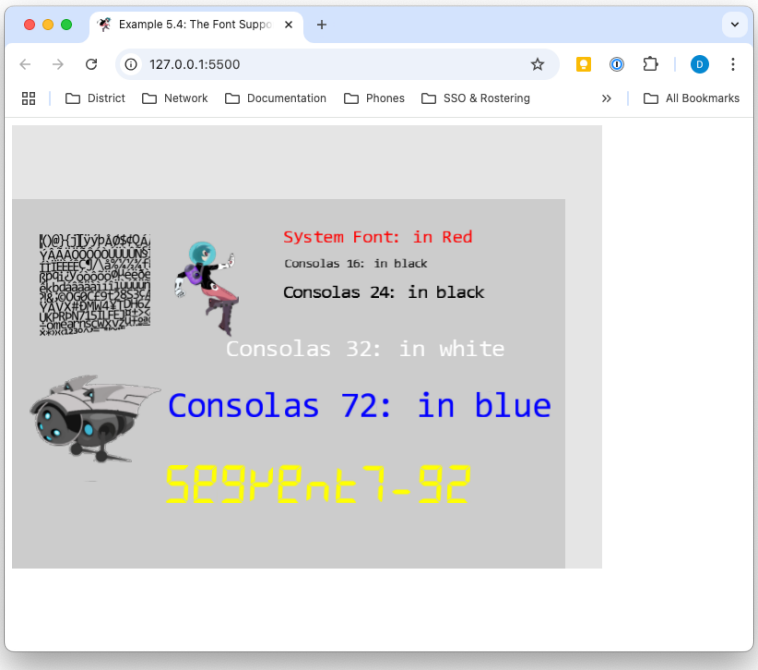

Figure 5-16Running the Font Support project

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 a selected subregion of an image and texture map to the unit square based on the normalize-ranged Texture Coordinate System. The chapter then explained how sprite sheets can reduce the time required for loading texture images while facilitating 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 takes advantage of an important aspect of game engine architecture: the `SimpleShader`/`Renderable` object pair where JavaScript `SimpleShader` 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 paired with `SpriteAnimateRenderable` shows that, when appropriate, the same shader object can support multiple `renderable` object types in the game engine. This `SimpleShader`/`Renderable` pair implementation pattern will appear again in Chapter 8, when you learn to create 3D illumination effects.

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 defining and supporting behaviors 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 for 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 (also known as the hero object) 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 determines 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 (e.g., 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 `.jpg` 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 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. Great 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 hyperreal 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 but 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 convincing 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. Imagine again combining the *Sprite Animation* project with the earlier projects in this chapter; as the hero moves closer to the robot, it might first turn red, eventually triggering the robot's animations and moving it either toward or away from the player. Animations often come fairly late in the game design process because it's usually necessary to first have the game mechanic and other systems well defined to avoid time-consuming changes that may be required as environments and level designs are updated. Designers typically use simple placeholder assets in the early stages of development, adding polished and animated final assets only when all of the other elements of gameplay have been finalized to minimize the need for rework.

As was the case with visual design, the animation approach need not be complex to be effective. While animation needs to be intentional and unified and 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, and 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.