Working with Textures, Sprites, and Fonts

After completing this chapter, you will be able to:

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

# Introduction

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

*Loading* is the reading of the image from the hard drive of the web server into the client’s system main memory, where it is processed and stored in the graphics subsystem. *Rendering* occurs during gameplay when the loaded image is drawn continuously to represent the respective game objects. *Unloading* happens when an image is no longer required by the game and the associated resources are reclaimed for future uses. Because of the slower response time of the hard drive and the potentially large amount of data that must be transferred and processed, loading images can take a noticeable amount of 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 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 alphabet characters from a sprite sheet to display text messages.

**Note** A texture is an image that is loaded into the graphics system and ready to be mapped onto a geometry. When discussing the process of texture mapping, you’ll hear “an image” and “a texture” often used interchangeably. 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 and the corresponding uv values defined for all images

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

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



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

## The Texture Shaders Project

This project demonstrates the loading, rendering, and unloading of textures with WebGL. You can see an example of this project running in Figure 5-3 with the left and right screenshots from the two scenes implemented. Notice the natural-looking objects without white borders in the left screenshot and the images with white backgrounds in the right screenshot. This project will also highlight the differences between images with and without the alpha channel, or *transparency*. The source code to this project is defined in the chapter5/5.1.texture\_shaders folder.

Graphical user interface, application

Description automatically generated

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

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

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

The goals of the project are as follows:

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

You can find the following external resource files in the assets folder: a scene-level file (blue\_level.xml) and four images (minion\_collector.jpg, minion\_collector.png, minion\_portal.jpg, and minion\_portal.png).

### Overview

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

* 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 to 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 a texture map.
* 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. This is in anticipating the many new shader and renderer types required to support the corresponding texture related functionality. Once again, continuous source code re-organization 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 Shader/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 Shader object maintains references to all attributes and uniform variables in the GLSL shaders and acts as the conduit for sending all transformation and vertex information to the SimpleVS/FS shaders. Although not depicted explicitly in Figure 5-4, there is only one instance of the SimpleShader object created in the game engine, in shader\_resoruces, and this instance is shared by all Renderable objects.

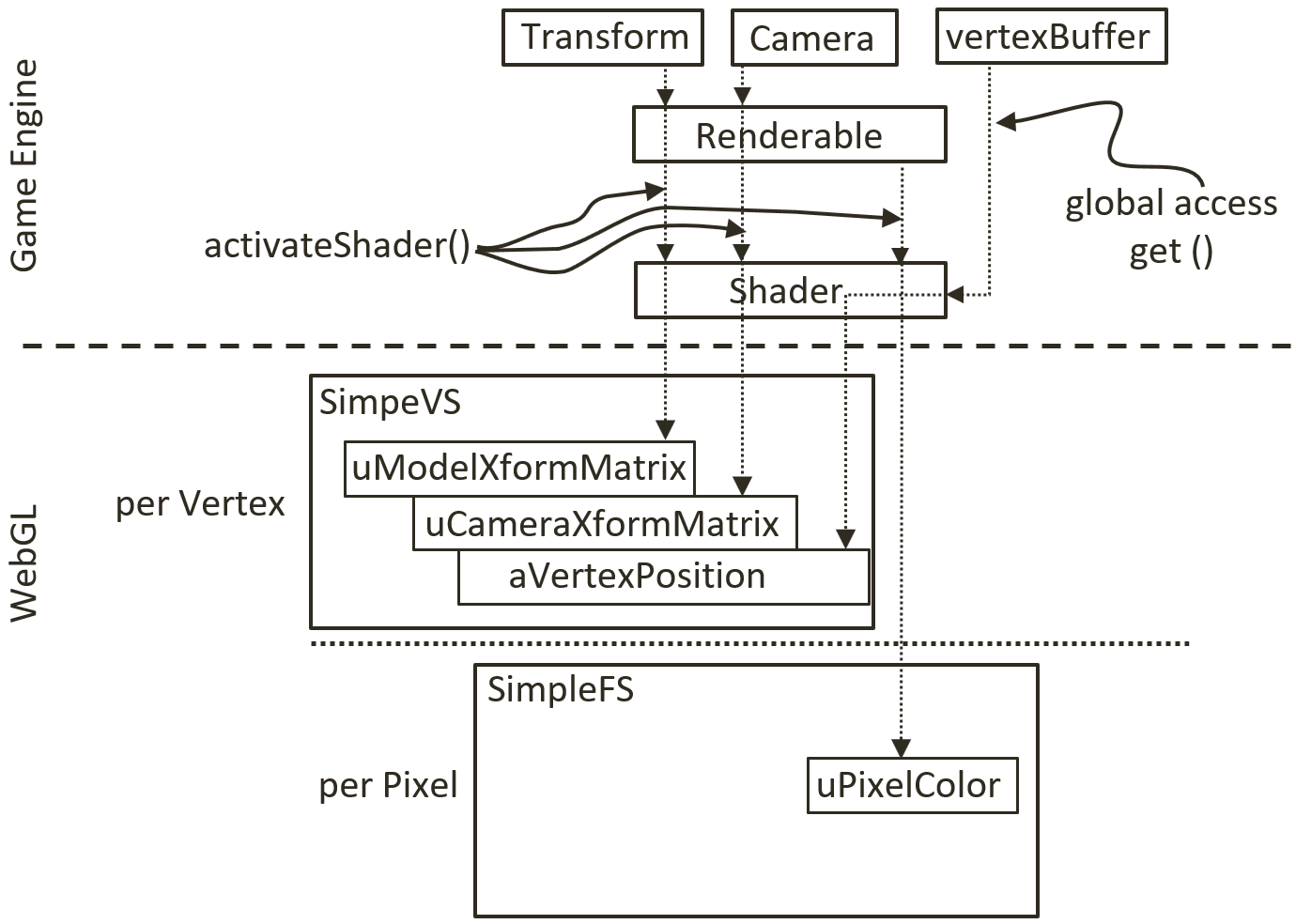


Figure 5-4. The Shader and Renderable architecture

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 sub-classing, the TextureVS/FS source code is copied from the SimpleVS/FS files.

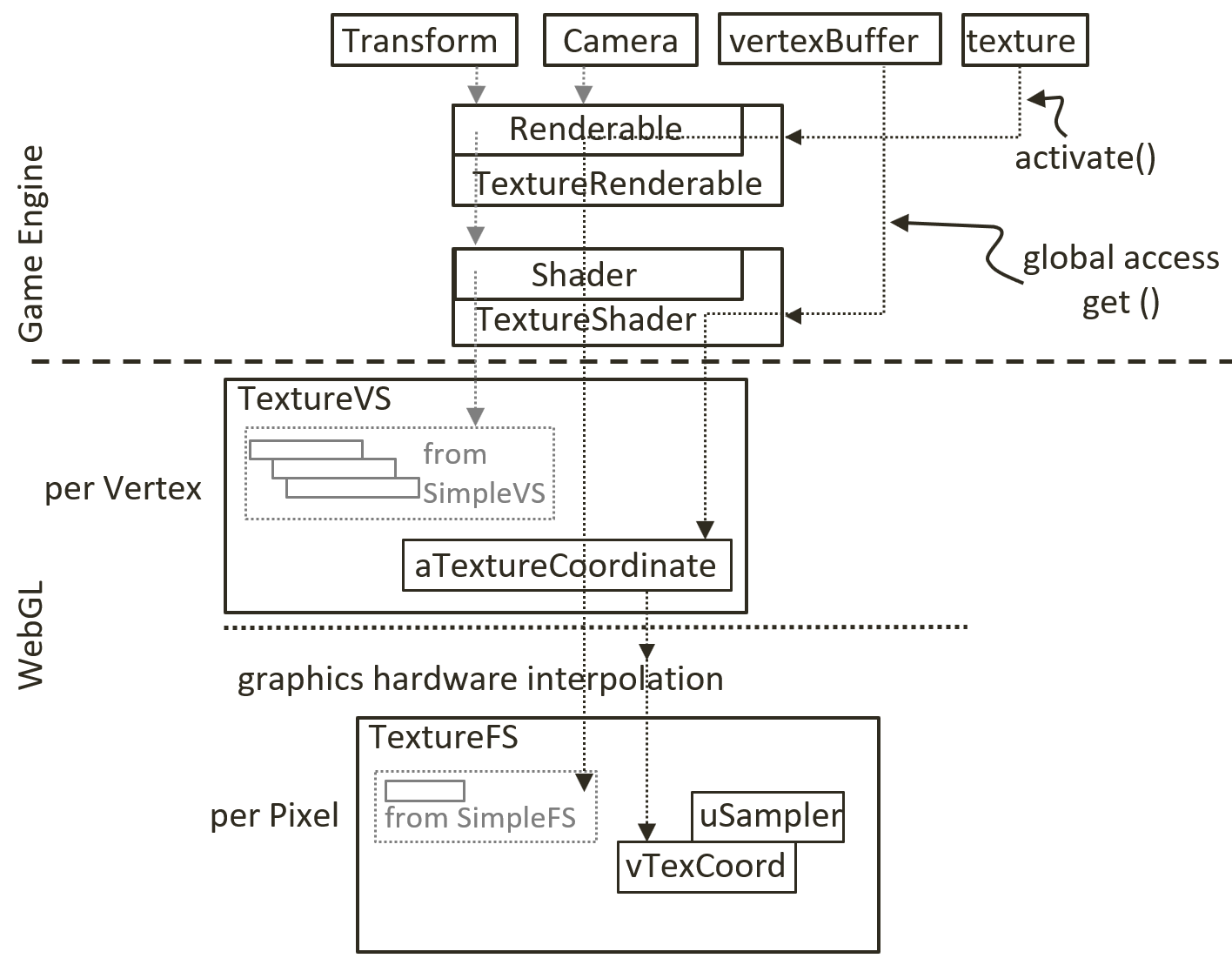


Figure 5-5. The TextureVS/FS GLSL shaders and the corresponding TextureShader/TextureRenderable object pair

### GLSL Texture Shader

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

1. Create a new file in the src/glsl\_shaders folder and name it texture\_vs.glsl.
2. Add the following code to the texture\_vs.glsl file:

attribute vec3 aVertexPosition; // Vertex shader expects one vertex position

attribute vec2 aTextureCoordinate; // This is the 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.

1. Add the aTextureCoordinate attribute. This defines a vertex to include a vec3 (aSquareVertexPosition, the xyz position of the vertex) and a vec2 (aTextureCooridnate, the uv coordinate of the vertex).
2. Declare the varying vTexCoord variable. The varying keyword in GLSL signifies that the associated variable will be linearly interpolated and passed to the fragment shader. As explained earlier and illustrated in Figure 5-2, uv values are defined only at vertex positions. In this case, the varying vTexCoord variable instructs the graphics hardware to linearly interpolate the uv values to compute the texture coordinate for each invocation of the fragment shader.
3. Assign the vertex uv coordinate values to the varying variable for interpolation and forwarding to the fragment shader.

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

1. Create a new file in the src/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 bounded to a GLSL texture such that texel values can be sampled for every pixel rendered. The uPixelColor is the same as the one from SimpleFS. The vTexCoord is the interpolated uv coordinate value for each pixel.

// The object that fetches data from texture.

// Must be set outside the shader.

uniform sampler2D uSampler;

// Color of pixel

uniform vec4 uPixelColor;

// The "varying" keyword is for signifying that the texture coordinate will be

// interpolated and thus varies.

varying vec2 vTexCoord;

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

void main(void) {

// texel color look up based on interpolated UV value in vTexCoord

vec4 c = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));

// tint the textured area. Leave transparent area as defined by the texture

vec3 r = vec3(c) \* (1.0-uPixelColor.a) + vec3(uPixelColor) \* uPixelColor.a;

vec4 result = vec4(r, c.a);

gl\_FragColor = result;

}

1. 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. The new mGLTextureCoordBuffer instance variable will be initialized to refer to the WebGL buffer that stores the values of mTextureCoordinates.

// 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

];

1. Define the variable to keep a reference to the WebGL buffer storage for the texture coordinates and the corresponding getter function.

let mGLTextureCoordBuffer = null;

function getTexCoord() { return mGLTextureCoordBuffer; }

1. Modify the init() function to include a step D to handle the initialization of the texture coordinates as a WebGL buffer. Notice the initialization process is identical to how the vertex xy coordinates are handled except 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);;

}

1. Remember to release the allocated buffer during final cleanup.

function cleanUp() {

// … identical to previous code …

if (mGLTextureCoordBuffer !== null) {

gl.deleteBuffer(mGLTextureCoordBuffer);

mGLTextureCoordBuffer = null;

}

}

1. 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 chapter overview, you will also create a new folder to organize the growing number of different shaders.

1. Create a new folder called shaders in src/engine. Move the simple\_shader.js file into this folder, and do not forget to update the reference path in index.js.
2. Create a new file in the src/engine/shaders/ folder and name it texture\_shader.js. Notice that the defined TextureShader class is an extension, or subclass, to the SimpleShader Class. 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. In the rest of the TextureShader constructor, the mTextureCoordinateRef keeps a reference to the aTextureCoordinate attribute defined in the texture\_vs.glsl. In this way, both the vertex position (aVertexPosition) and texture coordinate (aTextureCoordinate) attributes are referenced by a JavaScript TextureShader object.

class TextureShader extends SimpleShader {

constructor(vertexShaderPath, fragmentShaderPath) {

// Call super class constructor

super(vertexShaderPath, fragmentShaderPath); // call SimpleShader constructor

// 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");

}

…

1. 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 to the aTextureCoordinate, texture coordinate buffer defined in the vertex\_buffer component 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

}

In this way, 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 as SimpleShader is a reusable resource, only one instance of the TextureShader needs to be created, and this instance can be shared. The shader\_resources component should be modified to reflect this.

1. In shader\_resources.js, add the variables to hold a texture shader.

// Texture Shader

let kTextureVS = "src/glsl\_shaders/texture\_vs.glsl"; // Path to the VertexShader

let kTextureFS = "src/glsl\_shaders/texture\_fs.glsl"; // Path to the texture FragmentShader

let mTextureShader = null;

1. Define a function to retrieve the texture shader.

function getTextureShader() { return mTextureShader; }

1. Add the creation of the texture shader to the createShaders() function.

function createShaders() {

mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);

mTextureShader = new TextureShader(kTextureVS, kTextureFS);

}

1. 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);

}

1. 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);

}

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

1. Create the src/engine/renderables folder and move renderable.js into this folder. Remember to update index.js to reflect the file location change.
2. 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; }

#### 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 discussed, the myTexture parameter is the path to the file that contains the texture image.

class TextureRenderable extends Renderable {

constructor(myTexture) {

super();

super.setColor([1, 1, 1, 0]); // Alpha of 0: switch off tinting of texture

super.\_setShader(shaderResources.getTextureShader());

this.mTexture = myTexture; // texture for this object, cannot be a "null"

}

…

1. 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);

}

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

getTexture() { return this.mTexture; }

setTexture(newTexture) { this.mTexture = newTexture; }

### 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");

// Get 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}) ||

mCanvas.getContext("experimental-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

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

1. Create a new file in the src/engine/resources folder and name it texture.js. This file will implement the Textures engine component.
2. 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 an efficient implementation, many graphics hardware only supports texture with image resolutions in powers of 2, such as 2x4 , or 4x16 , or 64x256 , and so on. This is also the case for your configuration of WebGL. All examples in this book work only with textures with resolutions that are powers of 2.

1. Import the core resource management functionality from the resource\_map.

import \* as map from "../core/resource\_map.js";

// functions from resource\_map

let has = map.has;

let get = map.get;

1. 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. Note that distinct from the text and audio resources, JavaScript Image API supports straightforward image file loading and the map.loadDecideParse() 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.

// Loads a texture so that it can be drawn.

function load(textureName) {

let image = new Image();

let texturePromise = new Promise(

function(resolve) {

image.onload = resolve;

image.src = textureName;

}).then(

function resolve() {

processLoadedImage(textureName, image); }

);

map.pushPromise(texturePromise);

return texturePromise;

}

1. 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 gl = glSys.get();

let texInfo = get(textureName);

gl.deleteTexture(texInfo.mGLTexID);

map.unload(textureName);

}

1. Now define a function to convert the format of an image and store it to the WebGL context. theresource\_map

function processLoadedImage(path, image) {

let gl = glSys.get();

// Generate a texture reference to the webGL context

let textureID = gl.createTexture();

// bind the texture reference with the current texture functionality in the webGL

gl.bindTexture(gl.TEXTURE\_2D, textureID);

// Load the texture into the texture data structure with descriptive info.

// Parameters:

// 1: Which "binding point" or target the texture is being loaded to.

// 2: Level of detail. Used for mipmapping. 0 is base texture level.

// 3: Internal format. The composition of each element. i.e. pixels.

// 4: Format of texel data. Must match internal format.

// 5: The data type of the texel data.

// 6: Texture Data.

gl.texImage2D(gl.TEXTURE\_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED\_BYTE, image);

// Creates a mipmap for this texture.

gl.generateMipmap(gl.TEXTURE\_2D);

// Tells WebGL that we are 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.

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

function activate(textureName) {

let gl = glSys.get();

let texInfo = get(textureName);

// Binds our 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 pixel-graphics where you want the texture to look "sharp" do the following:

// gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_MAG\_FILTER, gl.NEAREST);

// gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_MIN\_FILTER, gl.NEAREST);

}

1. 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.
2. The texParameteri() functions 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.
3. 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 pixel texture.

1. 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);

}

1. 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 TextureRenderable.

// … 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 located at proper locations. Recall that the assets folder is created specifically for this purpose. Take note of the four new texture files located in the assets folder: minion\_collector.jpg, minion\_collector.png, minion\_portal.jpg, and minion\_portal.png.

#### Modify the BlueLevel Scene File to Support Textures

The 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. This texture tinting effect can be observed in the right image of Figure 5-3. In addition, notice that both images specified are in the JPEG format. Since the JPEG format does not support the storing of alpha channel, the unused regions of the two images show up as white areas outside the portal and collector minions in the right image of Figure 5-3.

#### 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. For details of the changes, please refer to the source code file in the src/my\_game/util folder.

#### 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.

1. Edit blue\_level.js and modify the constructor to define constants to represent 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;

}

…

1. 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);

}

1. 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);

}

1. Support loading of the next scene with the next() function.

next() {

super.next();

let nextLevel = new MyGame(); // load the next level

nextLevel.start();

}

1. Parse the textured squares in the init() function.

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);

}

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

update() {

// … Identical to previous code …

// continously change texture tinting

let c = this.mSqSet[1].getColor();

let ca = c[3] + deltaX;

if (ca > 1) {

ca = 0;

}

c[3] = ca;

}

1. Index 1 of mSqSet is the portal TextureRenderable object, and index 3 of the color array is the alpha channel.
2. 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 is passed to TextureShader and then loaded to the uPixelColor of TextureFS for tinting the texture map results.
3. 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"; // supports 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;

}

1. Load the textures in the load() function.

load() {

// loads the textures

engine.texture.load(this.kPortal);

engine.texture.load(this.kCollector);

}

1. 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);

}

1. 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();

}

1. 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 (orgX, orgY, width, height)

);

this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);

// sets the background to gray

// Step B: Create the game objects

this.mPortal = new 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.

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

When running the example for this project in the chapter5/5.1.texture\_shaders folder, once again take note of the results of continuously changing the texture tinting—the blinking of the portal minion in red. In addition, notice the differences between the PNG-based textures in the MyGame level and the corresponding JPEG ones with white borders in the BlueLevel. It is visually more pleasing and accurate to represent objects using textures with the alpha (or transparency) channel. PNG is one of the most popular image formats that supports the alpha channel.

### Observations

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

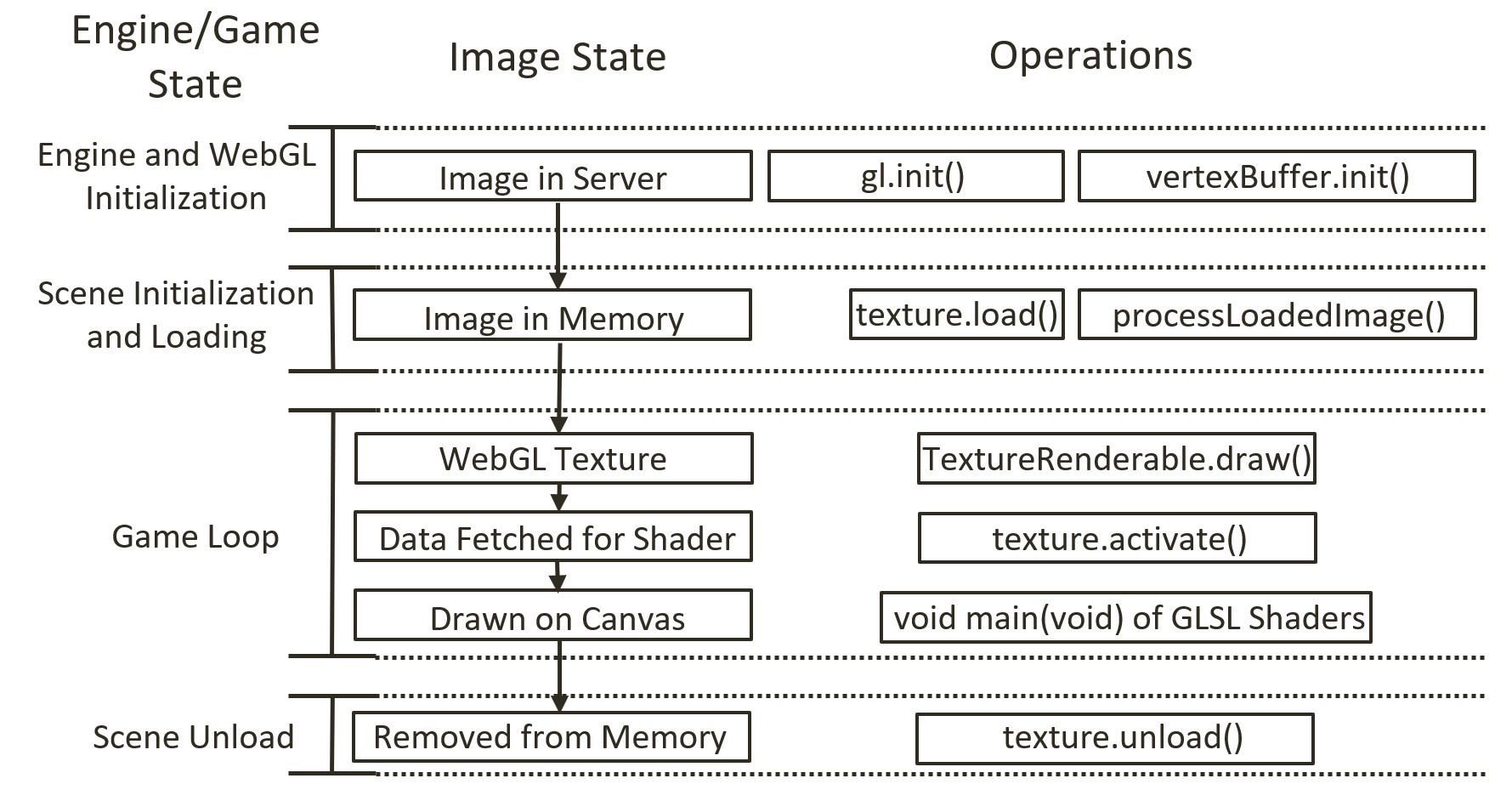


Figure 5-6. Overview of the states of an image file and the corresponding WebGL texture

The left column of Figure 5-6 identifies the main game engine states, from WebGL initialization to the initialization of a scene, to the game loop, and to the eventual unloading of the scene. The middle column shows the corresponding states of an image that will be used as a texture. Initially, this image is stored on the server file system. During 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 a 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 sample from the correct texture during rendering. Finally, when a texture is no longer needed by the game engine, the Scene.unload() function will call engine/resources/texture.unload() to remove the loaded image from the system.