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 supporting classes to support mapping with simple textures, working with sprite sheets that contain multiple objects, creating and controlling motions with animated sprite sheets, and extracting characters from a sprite sheet to display text messages.

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

# Texture Mapping and Texture Coordinates

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

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

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



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

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

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



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

## The Texture Shaders Project

This project demonstrates the loading, rendering, and unloading of textures with WebGL. You can see an example of this project running in Figure 5-3 with the left and right screenshots from the two scenes implemented. Notice the naturally appearing objects without white borders in the left screenshot and the images with white backgrounds in the right screenshot. This project will also highlight the differences between images with and without the alpha channel, or *transparency*. The source code to this project is defined in the chapter5/5.1.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.
* 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).
* Engine\_DefaultResources.js: This is a new file that defines a core engine component to facilitate the sharing of systemwide resources. In this case, it’s to facilitate the sharing of both SimpleShader and TextureShader by the corresponding Renderable objects.
* renderable.js: This file is modified to facilitate Renderable serving as the base class to all future types of Renderable objects and to share the Shader resource provided by gEngine\_DefaultResources.
* 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.
* Engine\_Core.js: This file is modified to configure WebGL to support drawing with a texture map.
* Engine\_Textures.js: This is a new file that defines the core engine component that is capable of loading, activating (for rendering), and unloading texture images.
* my\_game.js and blue\_level.js: These game engine client files are modified to test the new texture mapping functionality.

### Extension of Shader/Renderable Architecture

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

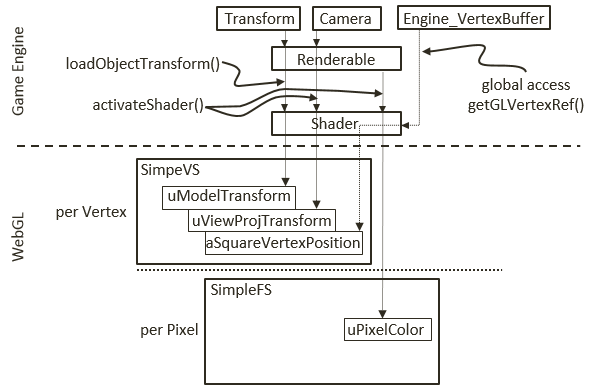


Figure 5-4. The Shader and Renderable architecture

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

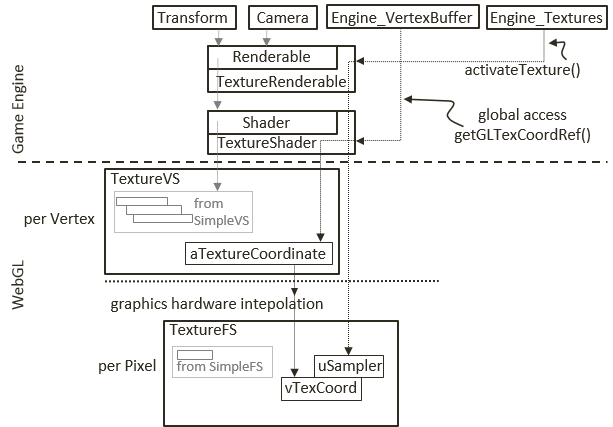


Figure 5-5. The TextureVS/FS GLSL shaders and the corresponding TextureShader/TextureRenderable object pair

### GLSL Texture Shader

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

1. Create a new file in the src/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:

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

// The object that fetches data from texture.

// Must be set outside the shader.

uniform sampler2D uSampler;

// Color of pixel

uniform vec4 uPixelColor;

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

// interpolated and thus varies.

varying vec2 vTexCoord;

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

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.

// reference to the vertex positions for the square in the gl context

let mGLVertexBuffer = null;

function get() { return mGLVertexBuffer; }

// 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

];

// reference to the texture coordinates for the square vertices in the gl context

let mGLTextureCoordBuffer = null;

// Second: define the corresponding texture coordinates

let mTextureCoordinates = [

1.0, 1.0,

0.0, 1.0,

1.0, 0.0,

0.0, 0.0

];

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.

1. Modify the init() function to the following:

function init() {

let gl = glSys.get();

// Step A: Create a buffer on the gl context for our vertex positions

mGLVertexBuffer = gl.createBuffer();

// Step B: Activate vertexBuffer

gl.bindBuffer(gl.ARRAY\_BUFFER, mGLVertexBuffer);

// Step C: Loads mVerticesOfSquare into the vertexBuffer

gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(mVerticesOfSquare), gl.STATIC\_DRAW);

// Step B: Allocate and store texture coordinates"

// Create a buffer on the gl context for texture coordinates

mGLTextureCoordBuffer = gl.createBuffer();

// Activate vertexBuffer

gl.bindBuffer(gl.ARRAY\_BUFFER, mGLTextureCoordBuffer);

// Loads textureCoordinates into the vertexBuffer

gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(mTextureCoordinates), gl.STATIC\_DRAW);

}

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

1. Add a function to retrieve the texture coordinates.

function getTexCoord() { return mGLTextureCoordBuffer; }

1. Finally, remember to export the changes to the public interface.

export {init,

cleanUp,

get,

getTexCoord}