Working with Textures, Sprites, and Fonts

Drawing with Sprite Sheets

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



Figure 5-7. Example sprite sheet: minion\_sprite.png composed of lower-resolution images of different objects

Sprite sheets are defined to optimize both memory and processing requirements. For example, recall that WebGL only supports textures that are defined by images with resolutions. This requirement means that the Dye character at a resolution of 120x180, must be stored in a 128x256 () 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 5-8 focuses on the uv values of the collector minion in Figure 5-7, the rightmost element on the third row. The top, center, and bottom rows of Figure 5-8 show coordinate values of the portal element.

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



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

## The Sprite Shaders Project

This project demonstrates how to draw objects with sprite sheet elements by defining appropriate abstractions and classes. You can see an example of this project running in Figure 5-9. The source code to this project is defined in the chapter5/5.2.sprite\_shaders folder.

Graphical user interface, application

Description automatically generated

Figure 5-9. Running the Sprite Shaders project

The controls of the project are as follows:

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

The goals of the project are as follows:

* To gain a deeper understanding for texture 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

You can find the following external resource files in the assets folder: consolas-72.png and minion\_sprite.png. Notice that minion\_sprite.png is the image shown in Figure 5-7.

As depicted in Figure 5-5, one of the main advantages and shortcomings of the texture support defined in the previous section is that the texture coordinate accessed via the 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 per-object texture coordinates in the SpriteShader and SpriteRenderable objects. Notice that there are no new GLSL shaders defined since their functionality remains the same.

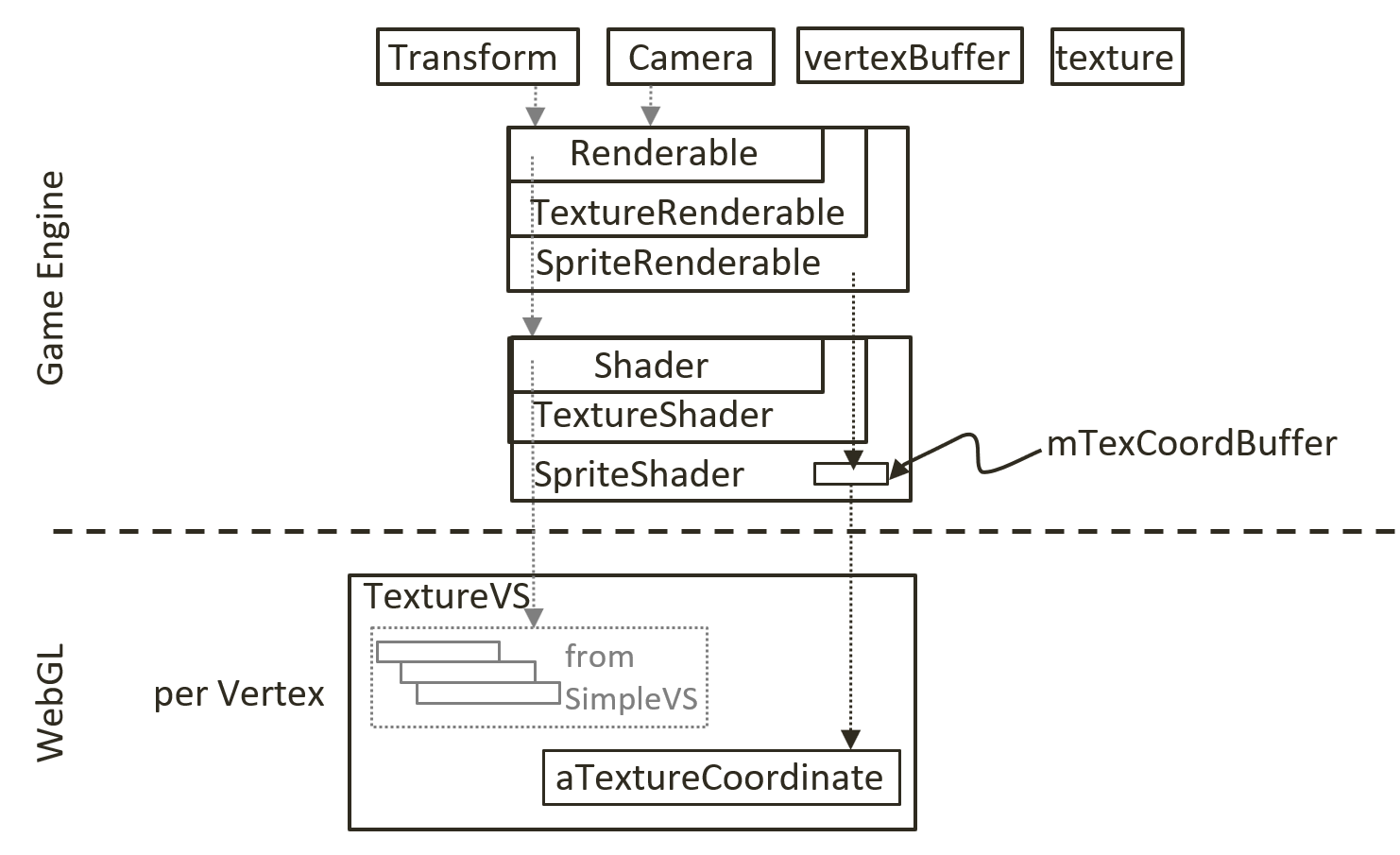


Figure 5-10. Defining a texture coordinate buffer in the SpriteShader

### Interface GLSL Texture Shaders to the Engine with SpriteShader

Shaders supporting texture mapping with sprite sheet elements must be able to identify distinct subregions of an image. To support this functionality, you will implement the SpriteShader to define its own texture coordinates. Since this new shader extends the functionality of TextureShader, it is 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); // call TextureShader constructor

this.mTexCoordBuffer = null; // reference to gl buffer containing the actual 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!

}

…

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.

1. 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.

1. Override the texture coordinate accessing function, \_getTexCoordBuffer(), such that the local allocated dynamic, instead of the global static, buffer is returned when the shader is activated (the activate() function is implemented by **TextureShader**)

\_getTexCoordBuffer() {

return this.mTexCoordBuffer;

}

### SpriteRenderable Object

Similar to 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 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; // bounds of texture coordinate (0 is left, 1 is right)

this.mElmRight = 1.0; //

this.mElmTop = 1.0; // 1 is top and 0 is bottom of image

this.mElmBottom = 0.0; //

}

…

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

// the expected 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.

1. Define functions to allow the specification of uv values for a sprite sheet element in both texture coordinate space (normalized between 0 to 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;

}

// specify element region by pixel positions (between 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 coordinates before storing the results with the corresponding instance variables.

1. 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

];

}

1. Override the draw() function to load the specific texture coordinates 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);

}

### 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; }

1. Modify the createShaders() function to also 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.

1. Update the cleanUp() function for proper release of resources.

function cleanUp() {

mConstColorShader.cleanUp();

mTextureShader.cleanUp();

mSpriteShader.cleanUp();

// … identical to previous code

}

1. 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 datatype for accessing the WebGL texture coordinates 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 working with sprite elements and texture coordinates that should be tested: the proper extraction, drawing, and controlling of a sprite sheet element as an object; and the changing and controlling of uv coordinate on an object. For proper testing of the added functionality, you must modify the my\_game.js file.

1. The constructing, loading, unloading, and drawing of MyGame are similar to previous examples, so the details will not be repeated here. Please refer to the source code in the src/my\_game folder for details.
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.setElementPixelPositions(315, 495, 0, 180);

// 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 the hero object with texture from the 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);

}

1. After the camera is set up, in step B, notice that both mPortal and mCollector are created based on the same image, kMinionSprite, with the respective setElementPixelPositions() calls to specify the actual sprite element to use for rendering.
2. Step C creates two additional SpriteRenderable objects: mFontImage and mMinion. The sprite element uv coordinate settings are left to the defaults where the texture image will cover the entire geometry.
3. Similar to step B, step C creates the hero character as a SpriteRenderable object based on the same kMinionSprite image. The sprite sheet element that corresponds to the hero is identified with the setElementPixelPositions() call.

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

1. 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) { // 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

xform.setXPos(20);

}

}

// continously 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

);

}

1. Observe that the keyboard control and the drawing of the hero object are identical to previous projects.
2. Notice the calls to setElementUVCoordinate() for mFontImage and mMinion. These calls continuously decrease and reset the V values that correspond to the bottom, the U values that correspond to the right for mFontImage, the V values that correspond to the top, and the U values that correspond to the left for mMinion. The end results are the continuous changing of texture and the appearance of a zooming animation on these two objects

# Sprite Animations

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

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

## Overview of Animated Sprite Sheets

Recall that an animated sprite sheet is a sprite sheet that contains the sequence of images of an object in an animation, typically in one or more rows or columns. For example, in Figure 5-11 you can see a 2x5 animated sprite sheet that contains two separate animations organized in two rows. The animations depict an object retracting its spikes toward the right in the top row and extending them toward the left in the bottom row. In this example, the animations are separated into rows. It is also possible for an animated sprite sheet to define animations that are organized in 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 5-11. An animated sprite sheet organized into two rows representing two animated sequences of the same object

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



Figure 5-12. A sprite animation sequence that loops

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

## The Sprite Animation Project

This project demonstrates how to work with an animated sprite sheet and generate continuous sprite animations. You can see an example of this project running in Figure 5-13. The project scene contains the objects from the previous scene plus two animated objects. The source code to this project is defined in the chapter5/5.3.sprite\_animate\_shaders folder.

Graphical user interface, application, Teams

Description automatically generated

Figure 5-13. Running the Sprite Animate Shaders project

The controls of the project are as follows:

* Right-arrow key: Moves the hero right; when crossing the right boundary, the hero is wrapped back to the left boundary
* **Left-arrow key:** Opposite 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 Object

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

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

1. Create a new file in the src/engine/renderables folder and name it sprite\_animate\_renderable.js.
2. Define an enumerated datatype for the three different sequences to animate with a sprite sheet.

// Assumption is that the first sprite in an animation is always the left-most element.

const eAnimationType = Object.freeze({

eRight: 0, // Animate from first (left) towards right, when hit the end, start from the left again

eLeft: 1, // Compute find the last element (in the right), start from the right animate left-wards,

eSwing: 2 // Animate from first (left) towards the right, when hit the end, animates backwards

});

eAnimationType defines three modes for animation.

1. 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.
2. 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.
3. eSwing is a continuous loop from left to right and then from right to left.
4. 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; // 1.0 is top corner of image (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();

}

The SpriteAnimateRenderable constructor defines three sets of variables:

1. The first set, including mFirstElmLeft, mElmTop, and so on, defines the location and dimensions of each sprite element and the number of elements in the animation. This information can be used to accurately compute the texture coordinates for each sprite element when the elements are ordered by rows and columns. Note that all coordinates are in Texture Coordinates Space (0 to 1).
2. 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.
3. 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, 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.

1. 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; // swings ...

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.

1. 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 value. With the assumption that all animation sequence is along the same row of sprite elements, the top and bottom v values are constant and do not change over a given animation sequence. These uv values are set to SpriteRenderable for drawing.

1. 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();

}

1. 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 left-most 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();

}

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

setAnimationSpeed(tickInterval) { this.mUpdateInterval = tickInterval; }

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

1. Finally, 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 systemwide updates to occur at kMPF intervals even when frame rate lags. The game engine architecture ensures the updateAnimation() function calls are kMPF milliseconds apart.

### 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";

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 programmer’s control over the modes (left, right, swing), and speed of animation. The MyGame object is modified to accomplish these purposes.

1. The constructing, loading, unloading, and drawing of MyGame are similar to the previous example and the details are not repeated.
2. In the init() function, add code to create and initialize the 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);

// … Identical to previous code …

}

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.

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

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 the duration of showing each sprite element, thereby speeding up the animation

if (engine.input.isKeyClicked(engine.input.keys.Four)) {

this.mRightMinion.incAnimationSpeed(-2);

this.mLeftMinion.incAnimationSpeed(-2);

}

// increase the duration of showing each sprite element, thereby slowing down the animation

if (engine.input.isKeyClicked(engine.input.keys.Five)) {

this.mRightMinion.incAnimationSpeed(2);

this.mLeftMinion.incAnimationSpeed(2);

}

// … Identical to previous code …

}

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.

Shape

Description automatically generated with medium confidence

Figure 5-14. An example bitmap font sprite image

A picture containing text

Description automatically generated

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

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

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

## The Font Support Project

This project demonstrates how to draw text from a bitmap font using the SpriteRenderable object. You can see an example of this project running in Figure 5-16. The source code to this project is defined in the chapter5/5.4.font\_support folder.



Figure 5-16. Running the Font Support project

The controls of the project are as follows:

* Number keys 0, 1, 2, and 3: Selects the Consolas, 16, 24, 32, or 72 fonts, respectively, for size modification
* Up/down key while holding down X/Y key: Increases or decreases (arrow keys) the width (X key) or the height (Y key) of the selected font
* Left- and Right-arrow keys: Move the hero left or right. The hero wraps if it exits the bounds.

The goals of the project are as follows:

* To understand bitmap fonts
* To implement text drawing support in your game engine

You can find the following external resource files in the assets folder: consolas-72.png and minion\_sprite.png. In the assets/fonts folder are the bitmap font sprite image files and the associated XML files that contain the decoding information: consolas-16.fnt, consolas-16.png, consolas-24.fnt, consolas-24.png, consolas-32.fnt, consolas-32.png, consolas-72.fnt, consolas-72.png, segment7-96.fnt, segment7-96.png, system-default-font.fnt, and system-default-font.png.

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

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

### Loading and Storing Fonts in the Engine

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

1. Define a class for storing uv coordinate location 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;

// reference to nominal character size, 1 is "standard width/height" of a char

this.mCharWidth = 1;

this.mCharHeight = 1;

this.mCharWidthOffset = 0;

this.mCharHeightOffset = 0;

// reference of char width/height ratio

this.mCharAspectRatio = 1;

}

}

1. 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;}

1. 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) {

texture.load(imageName(fontName));

return null;

}

function unload(fontName) {

texture.unload(imageName(fontName));

}

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

function has(fontName) {

return texture.has(imageName(fontName)) && xml.has(descName(fontName));

}

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

function getCharInfo(fontName, aChar) {

// … details omitted for lack of relevancy

returnInfo = new CharacterInfo();

// computes and fills in the contents of CharacterInfo

// … details omitted for lack of relevancy

return returnInfo;

};

Details of decoding and extracting information for 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.

1. Finally, remember to export the functions from this module.

export {has, load, unload,

imageName, descName,

CharacterInfo,

getCharInfo

}

To accomplish this, an engine utility should be defined to allocate 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. A similar pattern can be duplicated for sharing of default resources with the client—a default\_resoruces 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";

1. Definean 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 module, in the loop.start() function, 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 would 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);

### Defining a FontRenderable Object to Draw Texts

The defined font component 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, extracting the corresponding texture mapping information from the font component, 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 accepts a string as its parameter.

class FontRenderable {

constructor(aString) {

this.mFontName = defaultResources.getDefaultFontName();

this.mOneChar = new SpriteRenderable(font.imageName(this.mFontName));

this.mXform = new Transform(); // transform that moves this object around

this.mText = aString;

}

…

1. The aString variable is the message to be drawn.
2. 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 = function (camera) {

// we will draw the text string by calling to mOneChar for each of the

// chars in the mText string.

let widthOfOneChar = this.mXform.getWidth() / this.mText.length;

let heightOfOneChar = this.mXform.getHeight();

// this.mOneChar.getXform().SetRotationInRad(this.mXform.getRotationInRad());

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 does the following:

1. Extracts each character in the string
2. Calls the getCharInfo() function to receive the character’s uv values and appearance information in charInfo
3. 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)
4. Uses the appearance information from charInfo to compute the actual size (xSize and ySize) and location offset for the character (xOffset and yOffset) and draws the character mOneChar with the appropriate settings
5. 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(); }

1. 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)); // this is for "A"

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.

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

### After Game Cleanup

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

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

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

var cleanUp = function() {

gEngine.VertexBuffer.cleanUp();

gEngine.DefaultResources.cleanUp();

};

var mPublic = {

// … Identical to previous code …

cleanUp: cleanUp

};

1. In the Engine\_DefaultResources.js file, add the following function to release all default resources, and remember to add the cleanup() function to the public interface so that this function is accessible:

var cleanUp = function() {

mConstColorShader.cleanUp();

mTextureShader.cleanUp();

mSpriteShader.cleanUp();

gEngine.TextFileLoader.unloadTextFile(kSimpleVS);

gEngine.TextFileLoader.unloadTextFile(kSimpleFS);

// textuire shader:

gEngine.TextFileLoader.unloadTextFile(kTextureVS);

gEngine.TextFileLoader.unloadTextFile(kTextureFS);

// default font

gEngine.Fonts.unloadFont(kDefaultFont);

};

1. In the Engine\_VertexBuffer.js file, add the following function to release the allocated WebGL buffers, and remember to add the cleanup() function to the public interface so that this function is accessible:

var cleanUp = function() {

var gl = gEngine.Core.getGL();

gl.deleteBuffer(mSquareVertexBuffer);

gl.deleteBuffer(mTextureCoordBuffer);

};

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

SpriteShader.prototype.cleanUp = function() {

var gl = gEngine.Core.getGL();

gl.deleteBuffer(this.mTexCoordBuffer);

// now call super class's clean up ...

SimpleShader.prototype.cleanUp.call(this);

};

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

function SimpleShader(vertexShaderPath, fragmentShaderPath) {

// … Identical to previous code …

// Step A: load and compile vertex and fragment shaders

this.mVertexShader = this.\_CompileShader(vertexShaderPath,

gl.VERTEX\_SHADER);

this.mFragmentShader = this.\_CompileShader(fragmentShaderPath,

gl.FRAGMENT\_SHADER);

// Step B: Create and link the shaders into a program.

this.mCompiledShader = gl.createProgram();

gl.attachShader(this.mCompiledShader, this.mVertexShader);

gl.attachShader(this.mCompiledShader, this.mFragmentShader);

gl.linkProgram(this.mCompiledShader);

// Step C, D, E, F, G:

// … Identical to previous code …

};

SimpleShader.prototype.cleanUp = function() {

var gl = gEngine.Core.getGL();

gl.detachShader(this.mCompiledShader, this.mVertexShader);

gl.detachShader(this.mCompiledShader, this.mFragmentShader);

gl.deleteShader(this.mVertexShader);

gl.deleteShader(this.mFragmentShader);

};

1. The constructor is modified to reference the vertex and fragment shaders with the mVertexShader and mFragmentShader variables.
2. The new cleanUp() function releases the memory associated with the shaders.

### Testing Fonts

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

#### The GameOver Scene

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

1. Create a new file in the src/my\_game folder and name it GameOver.js.
2. Implement the constructor, with the mCamera to set up a convenient World Coordinate (WC) system and the mMsg to contain the “Game Over” message.

function GameOver() {

this.mCamera = null;

this.mMsg = null;

};

gEngine.Core.inheritPrototype(GameOver, Scene);

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

GameOver.prototype.initialize = function() {

// Step A: set up the cameras

this.mCamera = new Camera(

vec2.fromValues(50, 33), // position of the camera

100, // width of camera

[0, 0, 600, 400] // viewport (orgX, orgY, width, height)

);

this.mCamera.setBackgroundColor([0.9, 0.9, 0.9, 1]);

this.mMsg = new FontRenderable("Game Over!");

this.mMsg.setColor([0, 0, 0, 1]);

this.mMsg.getXform().setPosition(22, 32);

this.mMsg.setTextHeight(10);

};

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

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

GameOver.prototype.draw = function() {

// Step A: clear the canvas

gEngine.Core.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray

// Step B: Activate the drawing Camera

this.mCamera.setupViewProjection();

this.mMsg.draw(this.mCamera.getVPMatrix());

};

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

GameOver.prototype.update = function() {

gEngine.GameLoop.stop();

};

1. In the unloadScene() function, call gEngine\_Core to clean up all system default resources.

GameOver.prototype.unloadScene = function() {

gEngine.Core.cleanUp(); // release gl resources

};

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

#### The MyGame Scene

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

1. In the my\_game.js file, modify the constructor to contain corresponding variables for printing the messages, and modify the draw() function to draw all objects accordingly. Please refer to the src/my\_game/my\_game.js file for the details of the code.
2. Modify the load() function to load the textures and 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);

}

1. Modify the unload() function to unload the textures and fonts and to start the GameOver scene.

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);

}

1. 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.

\_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 right 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 pixel position: top and 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 the hero object with texture from the 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;

}

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

1. Modify the update() function with the following:

update() {

// … Identical to previous code …

// 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.Three)) {

this.mTextToWork = this.mTextCon32;

}

if (engine.input.isKeyClicked(engine.input.keys.Four)) {

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 previous code shows that you can do the following:

1. Select which FontRenderable object to work with based on keyboard 0 to 4 input.
2. Control the width and height of the selected FontRenderable object when both the left/right arrow and x/y keys are pressed.

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

# Summary

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

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

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

## Game Design Considerations

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

The Texture Shaders project demonstrated how .png images with transparency more effectively integrate game elements into the game environment than formats like JPEG that don’t support transparency. If you move the hero (represented here as simply a rectangle) to the right, nothing on the screen changes, but if you move the hero to the left, you’ll eventually trigger a state change that alters the displayed visual elements as you did in the Scene Objects project from Chapter 4. Notice how much more effectively the robot sprites are integrated into the game scene when they’re .png files with transparency on the gray background compared to when they’re .jpg images without transparency on the blue background. The Sprite Shaders project introduces a hero that more closely matches other elements in the game setting. You’ve replaced the hero rectangle from the Texture Shaders project with a humanoid figure stylistically matched to the flying robots on the screen, and the area of the rectangular hero image not occupied by the humanoid figure is transparent. If you were to combine the hero from the Sprite Shaders project with the screen-altering action in the Texture Shaders project, imagine that as the hero moves toward the robot on the right side of the screen, the robot might turn red when the hero gets too close. The coded events are still simple at this point, but you can see how the visual design and a few simple triggered actions can already begin to convey a game setting and enhance presence.

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

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

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

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

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