Implementing Common Components of Video Games

After completing this chapter, you will be able to:

* Control the Renderable object’s position, size, and rotation to construct complex movements and animations
* Receive keyboard input from the player and animate Renderable objects
* Work with asynchronous loading and unloading of external assets
* Define, load, and execute a simple game level from a scene file
* Change game levels by loading a new scene
* Work with sound clips for background music and audio cues

# Introduction

In the previous chapters, a skeletal game engine was constructed to support basic drawing operations. Drawing is the first step to constructing your game engine because it allows you to observe the output while continuing to expand the game engine functionality. In this chapter, the two important mechanisms, interactivity and resource support, will be examined and added to the game engine. Interactivity allows the engine to receive and interpret player input, while resource support refers to the functionality of working with external files like the GLSL shader source code files, audio clips, and images.

This chapter begins by introducing you to the game loop, a critical component that creates the sensation of real-time interaction and immediacy in nearly all video games. Based on the game loop foundation, player keyboard input will be supported via integrating the corresponding HTML5 input functionality into the game engine. A resource management infrastructure will be constructed from the ground up to support the efficient loading, storing, retrieving, and utilization of external files. Functionality for working with external text files (for example, the GLSL shader source code files) and audio clips will be integrated with corresponding example projects. Additionally, game scene architecture will be derived to support the ability to work with multiple scenes and scene transitions, including scenes that are defined in external scene files. By the end of this chapter, your game engine will support player interaction via the keyboard, have the ability to provide audio feedback, and be able to transition between distinct game levels including loading a level from an external file.

# The Game Loop

One of the most basic operations of any video game is the support of seemingly instantaneous interactions between the players’ input and the graphical gaming elements. In reality, these interactions are implemented as a continuous running loop that receives and processes player input, updates the game state, and renders the game. This constantly running loop is referred to as the game loop.

To convey the proper sense of instantaneity, each cycle of the game loop must be completed within a normal human’s reaction time. This is often referred to as real time, which is the amount of time that is too short for humans to detect visually. Typically, real-time can be achieved when the game loop is running at a rate of higher than 40 to 60 cycles in a second. Since there is usually one drawing operation in each game loop cycle, the game loop cycle’s rate can also be expressed as frames per second (FPS), or the frame rate. An FPS of 60 is a good target for performance. This is to say, your game engine must receive player input, update the game world, and then draw the game world all within 1/60th of a second!

The game loop itself, including the implementation details, is the most fundamental control structure for a game. With the main goal of maintaining real-time performance, the details of a game loop’s operation are of no concern to the rest of the game engine. For this reason, the implementation of a game loop should be tightly encapsulated in the core of the game engine with its detailed operations hidden from other gaming elements.

## Typical Game Loop Implementations

A game loop is the mechanism through which logic and drawing are continuously executed. A simple game loop consists of processing the input, updating the state of objects, and drawing those objects, as illustrated in the following pseudocode:

initialize();

while(game running) {

input();

update();

draw();

}

As discussed, an FPS of 60 is required to maintain the sense of real-time interactivity. When the game complexity increases, one problem that may arise is when sometimes a single loop can take longer than 1/60th of a second to complete, causing the game to run at a reduced frame rate. When this happens, the entire game will appear to slow down. A common solution is to prioritize which operations to emphasis and which to skip. Since correct input and updates are required for a game to function as designed, it is often the draw operation that is skipped when necessary. This is referred to as frame skipping, and the following pseudocode illustrates one such implementation:

elapsedTime = now;

previousLoop = now;

while(game running) {

elapsedTime += now - previousLoop;

previousLoop = now;

input();

while( elapsedTime >= UPDATE\_TIME\_RATE ) {

update();

elapsedTime -= UPDATE\_TIME\_RATE;

}

draw();

}

In the previous pseudocode listing, UPDATE\_TIME\_RATE is the required real-time update rate. When the elapsed time between the game loop cycle is greater than the UPDATE\_TIME\_RATE, update() will be called until it is caught up. This means that the draw() operation is essentially skipped when the game loop is running too slowly. When this happens, the entire game will appear to run slowly, with lagging gameplay input responses and skipped frames. However, the game logic will continue to be function correctly.

Notice that the while loop that encompasses the update() function call simulates a fixed update time step of UPDATE\_TIME\_RATE. This fixed time step update allows for a straightforward implementation in maintaining a deterministic game state. This is an important component to make sure your game engine functions as expected whether running optimally or slowly.

To ensure the focus is solely on the understanding of the core game loop’s update and draw operations, input will be ignored until the next project.

## The Game Loop Project

This project demonstrates how to incorporate a game loop into your game engine and to support real-time animation by updating and drawing the squares accordingly. You can see an example of this project running in Figure 4-1. The source code to this project is defined in the chapter4/4.1.game\_loop folder.

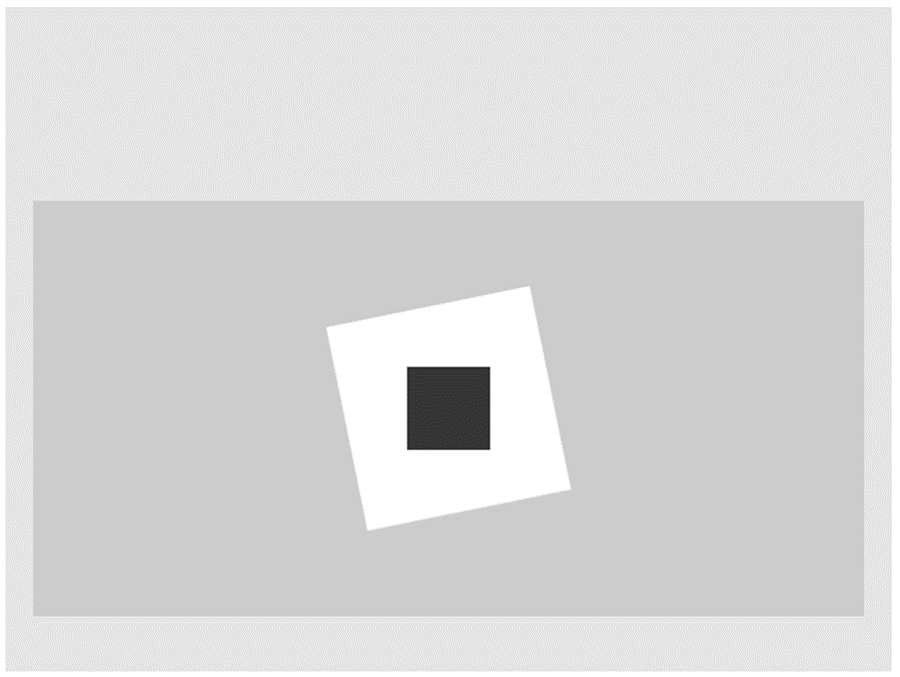


Figure 4-1. Running the Game Loop project

The goals of the project are as follows:

* To understand the internal operations of a game loop
* To implement and encapsulate the operations of a game loop
* To gain experience with continuous update and draw to create animation

### Implement the Game Loop Component

The game loop component is core to the game engine’s functionality and thus should be implemented similarly to vertex\_buffer, as a file defined in the src/engine/core folder.

1. Create a new file for the loop module in the src/engine/core folder and name the file loop.js.
2. Add the following instance variables to keep track of frame rate, processing time or milliseconds per frame, the gameloop’s current run state, and a reference to the current scene as follows:

"use strict"

const UPS = 60; // Updates per second

const MPF = 1000 / UPS; // Milliseconds per update.

// Variables for timing gameloop.

let mPrevTime;

let mLagTime;

// The current loop state (running or should stop)

let mLoopRunning = false;

let mCurrentScene = null;

let mFrameID = -1;

Notice that UPS is the updates per second similar to the FPS discussed and that MPF is milliseconds per frame. It is important to try and maintain the game at an update interval of 60 updates per second as stated in addition to 60 FPS.

**Note** When the game is running optimally, that is there is no lag UPS and FPS can be thought of interchangeably. That is, they both target 60 iterations per second. UPS through the engine controlled loop and FPS through requestAnimationFrame(). However, when lag occurs the loop prioritizes updates over frames.

1. Add a function to run the core loop as follows:

function loopOnce() {

if (mLoopRunning) {

// Step A: set up for next call to LoopOnce and update input!

mFrameID = requestAnimationFrame(loopOnce);

// Step B: now let's draw

// draw() MUST be called before update()

// as update() may stop the loop!

mCurrentScene.draw();

// Step C: compute how much time has elapsed since last loopOnce was executed

let currentTime = performance.now();

let elapsedTime = currentTime - mPrevTime;

mPrevTime = currentTime;

mLagTime += elapsedTime;

// Step D: Make sure we update the game the appropriate number of times.

// Update only every Milliseconds per frame.

// If lag larger then update frames, update until caught up.

while ((mLagTime >= MPF) && mLoopRunning) {

mCurrentScene.update();

mLagTime -= MPF;

}

}

}

Notice the similarity between the pseudocode examined previously and the steps B, C, and D of the loopOnce() function shown previously. That is, the drawing of the scene or game in step B, the calculation of the elapsed time since last update in step C, and the prioritization of update if the engine is lagging behind. The main difference is that the functionality of the outermost while loop is implemented with the requestAnimationFrame() function call at step A, where the loopOnce() function is set up to be called continuously. More specifically, the requestAnimationFrame() function registers the loopOnce() function with the browser which the browser will call on the next available frame.

Notice that each call to the requestAnimationFrame() function will result in exactly one execution of the corresponding loopOnce() function and thus draw only once. However, multiple updates can occur during this single frame if the drawing is lagging behind.

**Note** The mLoopRunning condition of the while loop in step D is a redundant check for now. This condition will become important in later sections when update() can call stop() to stop the loop (for example, for level transitions or the end of the game).

1. Declare a function to start the game loop as follows. This function initializes or starts the core game loop, initializing the game or scene, setting up the frame time, and setting the loop running flag to true before calling the first requestAnimationFrame(loopOnce).

function start(scene) {

if (mLoopRunning) {

throw new Error("loop already running")

}

mCurrentScene = scene;

mCurrentScene.init();

mPrevTime = performance.now();

mLagTime = 0.0;

mLoopRunning = true;

mFrameID = requestAnimationFrame(loopOnce);

}

1. Declare a function to stop the game loop as follows. This function simply stops the loop by setting mLoopRunning to false and cancels the last frame.

function stop() {

mLoopRunning = false;

// make sure no more animation frames

cancelAnimationFrame(mFrameID);

}

1. Lastly, remember to export the needed functionality. In this case both start and stop.

export {start, stop}

### Using the Game Loop

To test the game loop implementation, your game class should implement the update() and draw() functions. In this case, the MyGame object will also define an initialize() function.

1. Provide temporary access to the loop by importing the loop module as follows.

// Accessing engine internal is not ideal,

// this must be resolved! (later)

import \* as loop from "../engine/core/loop.js";

1. In my\_game.js, replace the MyGame constructor with the following:

constructor() {

// variables for the squares

this.mWhiteSq = null; // these are the Renderable objects

this.mRedSq = null;

// The camera to view the scene

this.mCamera = null;

}

1. Add an initialization function to the class as follows. The initialization is rather similar to previous examples, where a camera is defined and two squares are set up.

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 Renderable objects:

this.mWhiteSq = new engine.Renderable();

this.mWhiteSq.setColor([1, 1, 1, 1]);

this.mRedSq = new engine.Renderable();

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

// Step C: Initialize the white Renderable object: centered, 5x5, rotated

this.mWhiteSq.getXform().setPosition(20, 60);

this.mWhiteSq.getXform().setRotationInRad(0.2); // In Radians

this.mWhiteSq.getXform().setSize(5, 5);

// Step D: Initialize the red Renderable object: centered 2x2

this.mRedSq.getXform().setPosition(20, 60);

this.mRedSq.getXform().setSize(2, 2);

}

1. Draw the scene as before by clearing the canvas, setting up the camera, and drawing each square.

draw() {

// Step A: clear the canvas

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

// Step B: Activate the drawing Camera

this.mCamera.setViewAndCameraMatrix();

// Step C: Activate the white shader to draw

this.mWhiteSq.draw(this.mCamera);

// Step D: Activate the red shader to draw

this.mRedSq.draw(this.mCamera);

}

1. Add an update() function to animate a moving white square and a pulsing red square.

update() {

// For this very simple game, let's move the white square and pulse the red

let whiteXform = this.mWhiteSq.getXform();

let deltaX = 0.05;

// Step A: Rorate the white square

if (whiteXform.getXPos() > 30) // this is the right-bound of the window

whiteXform.setPosition(10, 60);

whiteXform.incXPosBy(deltaX);

whiteXform.incRotationByDegree(1);

// Step B: pulse the red square

let redXform = this.mRedSq.getXform();

if (redXform.getWidth() > 5)

redXform.setSize(2, 2);

redXform.incSizeBy(0.05);

}

Recall that the update() function is called at about 60 times per second, and each time the following happens:

* Step A for the white square: Increase the rotation by 1 degree; increase the x-position by 0.05 and reset to 10 if the resulting x-position is greater than 30.
* Step B for the red square: Increase the size by 0.05 and reset it to 2 if the resulting size is greater than 5.
* Since the previous operations are performed continuously at about 60 times a second, you can expect to see the following:

1. A white square rotating while moving toward the right and reappearing when it reaches the right boundary
2. A red square increasing in size and reducing to a size of 2 when the size reaches 5, thus appearing to be pulsing
3. Export the MyGame client so that it can be used via the engine.

export default MyGame;

1. Start the game loop upon the loading completion of the window.onload function. Notice that a reference to an instance of MyGame is passed to the loop.

window.onload = function () {

engine.init("GLCanvas");

let myGame = new MyGame();

// new begins the game

loop.start(myGame);

}

You can now run the project to observe the rightward-moving, rotating white square and the pulsing red square. You can control the rate of the movement, rotation, and pulsing by changing the corresponding values of the incXPosBy(), incRotationByDegree(), and incSizeBy() functions. In these cases, the positional, rotational, and size values are changed by a constant amount in a fixed time interval. In effect, the parameters to these functions are the rate of change; or, the speed, incXPosBy(0.05), is the rightward speed of 0.05 units per 1/60th of a second, or 3 units per second. In this project, the width of the world is 20 units with the white square traveling at 3 units per second. You can verify that it takes slightly more than 6 seconds for the white square to travel from the left to the right boundary.

Notice that when the loop is running quickly, it is entirely possible for the loopOnce() function to be called multiple times within a single MPF interval. With the given loopOnce() implementation, the draw() function will be called multiples times without any update() function calls. This way, the game loop can end up drawing the same game state multiple times. Please refer to the following references for discussions of supporting extrapolations in the draw() function to take advantage of efficient game loops:

* http://gameprogrammingpatterns.com/game-loop.html#play-catch-up
* http://gafferongames.com/game-physics/fix-your-timestep/

To clearly describe each component of the game engine and illustrate how these components interact, this book does not support extrapolation of the draw() function.

# Keyboard Input

It is obvious that proper support to receive player input is important to interactive video games. For a PC, the two common input devices are the keyboard and the mouse. While keyboard input is received in the form of a stream of characters, mouse input is packaged with positional information and is related to camera views. For this reason, keyboard input is simpler to support at this point in the engine’s development. This section will introduce and integrate keyboard support into your game engine. Mouse input will be examined in the “Mouse Input project” of Chapter 7, after the coverage of supporting multiple cameras in the same game.

## The Keyboard Support Project

This project examines keyboard input support and incorporates the functionality into the game engine. The position, rotation, and size of the game objects in this project are under your input control. You can see an example of this project running in Figure 4-2. The source code to this project is defined in the chapter4/4.2.keyboard\_support folder.

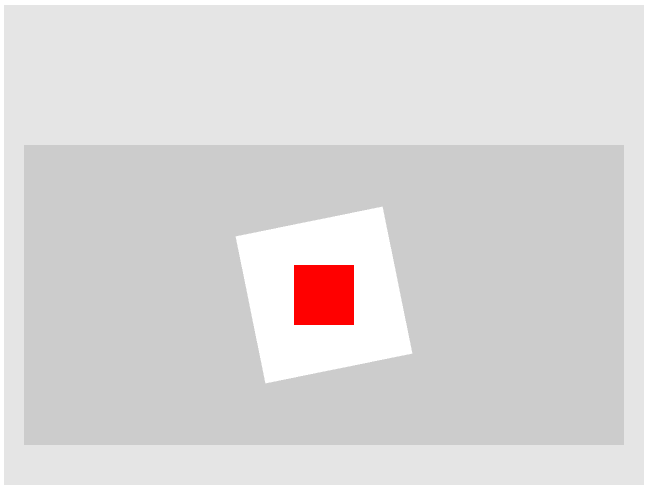


Figure 4-2. Running the Keyboard Support project

The controls of the project are as follows:

* Right arrow key: Moves the white square right and wraps it to the left of the game window
* Up arrow key: Rotates the white square
* Down arrow key: Increases the size of the red square and then resets the size at a threshold

The goals of the project are as follows:

* To implement an engine component to receive keyboard input
* To understand the difference between key state (if a key is released or pressed) and key event (when the key state changes)
* To understand how to integrate the input component in the game loop

### Add an Input Component to the Engine

The input component of the game follows the same pattern as the other core engine modules such as the vertex buffer or the game loop. A well-defined input module should allow the rest of the game engine to query keyboard state changes without being distracted by any details. To accurately capture keyboard state changes, the input component will be integrated with the core of game loop.

1. Create a new file in the src/engine/core folder and name it input.js.
2. Define a set of keyboard keys to map key codes.

"use strict"

// Key code constants

const keys = {

// arrows

Left: 37,

Up: 38,

Right: 39,

Down: 40,

// space bar

Space: 32,

// numbers

Zero: 48,

One: 49,

Two: 50,

Three: 51,

Four: 52,

Five : 53,

Six : 54,

Seven : 55,

Eight : 56,

Nine : 57,

// Alphabets

A : 65,

D : 68,

E : 69,

F : 70,

G : 71,

I : 73,

J : 74,

K : 75,

L : 76,

Q : 81,

R : 82,

S : 83,

W : 87,

LastKeyCode: 222

}

Key codes are the codes used by the keyboard handler where each keyboard character has its own unique number, the corresponding key code. Note that there are up to 222 keys tracked. In the previous listing, only the constants are shown in the public interface.

**Note** Key codes for the alphabets are continuous, starting from 65 for A and ending with 90 for Z. You should feel free to add any characters for your own game engine. For a complete list of key codes, see **www.cambiaresearch.com/articles/15/javascript-char-codes-key-codes**.

1. Create array instance variables to track the various key states.

// Previous key state

let mKeyPreviousState = []; // a new array

// The pressed keys.

let mIsKeyPressed = [];

// Click events: once an event is set, it will remain there until polled

let mIsKeyClicked = [];

Each of the three arrays contain all the key states as booleans. The mKeyPreviousState records the key states of the previous update cycle, and the mIsKeyPressed object records the current state of the keys. The key code entries of these two objects are true when the corresponding keyboard keys are pressed, and they are false otherwise. The mIsKeyClicked array captures key click events. The key code entries of this array are true only when the corresponding keyboard key goes from being pressed to being released.

It is important to note that KeyPress is the state of a key, while KeyClicked is an event. For example, if a player presses the A key for one second before she releases it, for the duration of that entire second, KeyPress for A is true, while KeyClick for A is true only once when the player releases the key.

1. Add functions to capture the actual keyboard state changes.

// Event handler functions

function onKeyDown(event) {

mIsKeyPressed[event.keyCode] = true;

}

function onKeyUp(event) {

mIsKeyPressed[event.keyCode] = false;

}

When the previous functions are called, they use their corresponding key code to record keyboard state changes.

1. Add a function to initialize all the key states and register the key event handlers to the browser. Notice that the window.addEventListener() function registers the onKeyUp/Down() event handler functions with the browser such that these functions will be called to register the keyboard state changes.

function init() {

let i;

for (i = 0; i < keys.LastKeyCode; i++) {

mIsKeyPressed[i] = false;

mKeyPreviousState[i] = false;

mIsKeyClicked[i] = false;

}

// register handlers

window.addEventListener('keyup', onKeyUp);

window.addEventListener('keydown', onKeyDown);

}

1. Add an update() function to derive the key click events. The update() function uses mIsKeyPressed and mKeyPreviousState to determine whether a key clicked event has occurred.

function update() {

let i;

for (i = 0; i < keys.LastKeyCode; i++) {

mIsKeyClicked[i] = (!mKeyPreviousState[i]) && mIsKeyPressed[i];

mKeyPreviousState[i] = mIsKeyPressed[i];

}

}

1. Add public functions for clean inquires to current keyboard states for use with the game client.

// Function for GameEngine programmer to test if a key is pressed down

function isKeyPressed(keyCode) {

return mIsKeyPressed[keyCode];

}

function isKeyClicked(keyCode) {

return mIsKeyClicked[keyCode];

}

1. Finally, export the public functions and key constants needed for the engine and client.

export {keys, init,

update,

isKeyClicked,

isKeyPressed

}

### Modify the Engine to Support Keyboard Input

To properly support input, the engine must first initialize the arrays that represent the keyboard state, in other words, mIsKeyPressed, mIsKeyClicked, and mKeyPreviousState, and be followed by a continuous update of these arrays in the core of the game loop.

1. Modify index.js by importing the input.js module, adding the initialization of the input to the init() function, and adding the input module to the exported files to allowing access to the engine component as follows:

import \* as input from "./input.js";

function init(htmlCanvasID) {

glSys.init(htmlCanvasID);

vertexBuffer.init();

shaderResources.init();

input.init();

}

export default {

// input support

input,

// Util classes

Camera, Transform, Renderable,

// functions

init, clearCanvas

}

1. Include the input’s update() function in the core game loop by adding the following lines to loop.js. Notice the rest of the code is identical.

import \* as input from "../input.js";

function loopOnce() {

if (mLoopRunning) {

// … rest of the code is identical …

// Step D: Make sure we update the game the appropriate number of times.

// Update only every Milliseconds per frame.

// If lag larger then update frames, update until caught up.

while ((mLagTime >= MPF) && mLoopRunning) {

input.update();

mCurrentScene.update();

mLagTime -= MPF;

}

}

}

### Test Keyboard Input

You can test the input functionality by modifying the renderable objects in your MyGame class.

Replace the code in the MyGame update() function with the following:

update() {

// For this very simple game, let's move the white square and pulse the red

let whiteXform = this.mWhiteSq.getXform();

let deltaX = 0.05;

// Step A: test for white square movement

if (engine.input.isKeyPressed(engine.input.keys.Right)) {

if (whiteXform.getXPos() > 30) { // this is the right-bound of the window

whiteXform.setPosition(10, 60);

}

whiteXform.incXPosBy(deltaX);

}

// Step B: test for white square rotation

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

whiteXform.incRotationByDegree(1);

}

let redXform = this.mRedSq.getXform();

// Step C: test for pulsing the red square

if (engine.input.isKeyPressed(engine.input.keys.Down)) {

if (redXform.getWidth() > 5) {

redXform.setSize(2, 2);

}

redXform.incSizeBy(0.05);

}

}

In the previous code, step A ensures that pressing and holding the right arrow key will move the white square toward the right. Step B checks for the pressing and then the releasing of the up arrow key event. The white square is rotated when such an event is detected. Notice that pressing and holding the up arrow key will not generate a key press event and thus will not cause the white square to rotate. Step C tests for the pressing and holding of the down arrow key to pulse the red square.

You can run the project and include additional controls for manipulating the squares. For example, include support for the **WASD** keys to control the location of the red square. Notice once again that by increasing/decreasing the position change amount, you are effectively controlling the speed of the object’s movement.

**Note** The term "**WASD** keys" is used to refer to the key binding of the popular game controls: key W to move upwards, A leftwards, S downwards, and D rightwards.

# Resource Management and Asynchronous Loading

Video games typically utilize a multitude of artistic assets, or resources, including audio clips and images. When a game first begins to execute, these resources are typically stored externally on a system hard drive or a server across the network. For this reason, these resources are sometimes referred to as external resources or assets. External resources must be explicitly loaded into a game.

Since there can be a large number of required resources to support an entire game, storing them with the running game can potentially be memory intensive. A game should load and unload resources dynamically based on necessity. However, loading external resources may involve input/output device operations or network packet latencies and thus can be time intensive and potentially affect real-time interactivity. For these reasons, only a portion of resources are kept in memory, with loading operations strategically executed to avoid interrupting the game. In most cases, resources required in each level are kept in memory to support real-time interaction during the game play of that level. With this approach, external resource loading can be implemented during level transitions where players are expecting a new game environment and slight delays for loadings can be tolerated.

Once loaded, a resource must be readily accessible to support interactivity. The efficient and effective management of resources is essential to any game engine. Take note of the clear differentiation between resource managements, the responsibility of a game engine, and the actual ownerships of the resources. For example, a game engine must support the efficient loading and playing of the background music for a game, and it is the game (or client of the game engine) that actually owns and supplies the audio file for the background music. When implementing support for external resources management, it is important to remember that the actual resources are not part of the game engine.

At this point, the game engine you have been building handles only one type of resource—the GLSL shader files. Recall that the SimpleShader object loads and compiles the simple\_vs.glsl and simple\_fs.glsl files in its constructor. So far, the shader file loading has been accomplished via synchronous XMLHttpRequest.open(). This synchronous loading is an example of inefficient resource management because no operations can occur while the browser attempts to open and load the shader file. An efficient alternative would be to issue an asynchronous load command and allow additional operations to continue while the file is being opened and loaded.

This section builds an infrastructure to support asynchronous loading and efficient accessing of the loaded resources. Based on this infrastructure, over the next few projects, the game engine will be expanded to support batch resource loading during scene transitions.

## The Resource Map and Shader Loader Project

This project guides you to develop the resource\_map component, an infrastructural module for resource management, and demonstrates how to work with this module to load shader files asynchronously. You can see an example of this project running in Figure 4-3. This project appears to be identical to the previous project, with the only difference being how the GLSL shaders are loaded. The source code to this project is defined in the chapter4/4.3.resource\_map\_and\_shader\_loader folder.



Figure 4-3. Running the Resource Map and Shader Loader project

The controls of the project are identical to the previous project as follows:

* Right arrow key: Moves the white square right and wraps it to the left of the game window
* Up arrow key: Rotates the white square
* Down arrow key: Increases the size of the red square and then resets the size at a threshold

The goals of the project are as follows:

* To understand the handling of asynchronous loading
* To build an infrastructure that supports future resource loading and accessing
* To experience asynchronous resource loading via loading of the GLSL shader files

### Add a Resource Map Component to the Engine

The resource\_map engine component manages resource loading, storage, and retrieval after the resources are loaded. As in the case of all core engine components (for example, input or game loop), the implementation is as follows:

1. Create a new file in the src/engine/core folder and name it resource\_map.js.
2. TEMP TEXT

"use strict"

let mMap = new Map();

let mOutstandingPromises = [];

1. TEMP TEXT

function has(path) { return mMap.has(path) }

1. TEMP TEXT

function get(path) {

if (!has(path)) {

throw new Error("Error [" + path + "]: not loaded");

}

return mMap.get(path);

}

1. TEMP TEXT

// generic loading function,

// Step 1: fech from server

// Step 2: decodeResource on the loaded

// Step 3: parseResource on the decodedResource

// Step 4: store result into the map

// Push the promised operation into an array

function loadDecodeParse(path, decodeResource, parseResource) {

let r = null;

if (!has(path)) {

r = fetch(path)

.then(res => decodeResource(res) )

.then(data => parseResource(data) )

.then(data => { return mMap.set(path, data) } )

.catch(err => { throw err });

pushPromise(r);

}

return r;

}

1. TEMP TEXT

function unload(path) { mMap.delete(path) }

1. TEMP TEXT

function set(key, value) { mMap.set(key, value); }

1. TEMP TEXT

function pushPromise(p) { mOutstandingPromises.push(p); }

1. TEMP TEXT

// will block, wait for all oustanding promises complete

// before continue

async function waitOnPromises() {

await Promise.all(mOutstandingPromises);

mOutstandingPromises = []; // remove all

}

1. TEMP TEXT

export {has, get, set, loadDecodeParse, unload, pushPromise, waitOnPromises}

### Define a Text File as an Engine Resource

This section will define a text module that works with the resource\_map module to load your text files asynchronously. This module serves as an excellent example of how to take advantage of the resource\_map facility and allows you to replace the synchronous loading of GLSL shader files in order to verify its asynchronous functionality while also upgrading your game engine.

1. Create a new folder in src/engine/ and name it resources. This new folder is created in anticipation of the necessary support for many resource types and to maintain a clean source code organization.
2. Create a new file in the src/engine/core/resources folder and name it text.js.
3. TEMP TEXT

"use strict"

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

1. TEMP TEXT

// functions from resource\_map

let unload = map.unload;

let has = map.has;

let get = map.get;

1. TEMP TEXT

function decodeText(data) {

return data.text();

}

1. TEMP TEXT

function parseText(text) {

return text;

}

1. TEMP TEXT

function load(path) {

return map.loadDecodeParse(path, decodeText, parseText);

}

1. TEMP TEXT

export {has, get, load, unload}

### Load Shaders Asynchronously

The text resource module can now be used to assist the loading of the shader files asynchronously as plain-text files. Since it is impossible to predict when an asynchronous loading operation will be completed, it is important to issue the load commands before the resources are needed and to ensure that the loading operations are completed before proceeding to retrieve the resources.

#### Modify Shader Resources for Asynchronous Support

To avoid loading the GLSL shader files synchronously, the files must be loaded before the creation of a SimpleShader object. Based on this observation, your shader\_resources engine component module should be used to load the GLSL shader files during engine initialization as it is used to create sharable instances of the SimpleShader object. In general, the shader\_resources component can serve as the infrastructure that supports all future loading and sharing of game engine shader resources.

1. TEMP TEXT

import \* as text from "../resources/text.js";

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

1. TEMP TEXT

function init() {

let loadPromise = new Promise(

async function(resolve) {

await Promise.all([

text.load(kSimpleFS),

text.load(kSimpleVS)

]);

resolve();

}).then(

function resolve() { createShaders(); }

);

map.pushPromise(loadPromise);

}

#### Modify SimpleShader to Retrieve Shader Files

With the understanding that the GLSL shader files are already loaded, the changes to the SimpleShader object are straightforward. Instead of synchronously loading the shader files in the loadAndCompileShader() function, the contents to these files can simply be retrieved via the text resource.

1. TEMP TEXT

import \* as text from "./resources/text.js";

1. Since no loading operations are required, you should change the loadAndCompileShader() function name to simply compileShader() and change the file-loading commands to text resource retrievals. Notice to function remains largely the same except the loading has already been handled and thus the shader text files can be retrieved easily. Also

function compileShader(filePath, shaderType) {

let shaderSource = null, compiledShader = null;

let gl = glSys.get();

// Step A: Access the shader textfile

shaderSource = text.get(filePath);

if (shaderSource === null) {

throw new Error("WARNING:" + filePath + " not loaded!");

return null;

}

// Step B: Create the shader based on the shader type: vertex or fragment

compiledShader = gl.createShader(shaderType);

// Step C: Compile the created shader

gl.shaderSource(compiledShader, shaderSource);

gl.compileShader(compiledShader);

// Step D: check for errors and return results (null if error)

// The log info is how shader compilation errors are typically displayed.

// This is useful for debugging the shaders.

if (!gl.getShaderParameter(compiledShader, gl.COMPILE\_STATUS)) {

throw new Error("Shader ["+ filePath +"] compiling error: " + gl.getShaderInfoLog(compiledShader));

}

return compiledShader;

}

1. Remember that in the SimpleShader constructor, the calls to loadAndCompileShader() functions should be replaced by the newly modified compileShader() functions, as follows:

constructor(vertexShaderPath, fragmentShaderPath) {

// … identical to previous code …

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

this.mVertexShader = compileShader(vertexShaderPath, gl.VERTEX\_SHADER);

this.mFragmentShader = compileShader(fragmentShaderPath, gl.FRAGMENT\_SHADER);

// … identical to previous code …

}

#### Wait for Asynchronous Loading to Complete

Before using any resources, such as a shader text resource file it is important to wait for the asynchronous loading to fully complete. Because of this upon starting your game you will need to block your code from continuing to execute while you await for your resource\_map promises you defined previously to be kept. This is often thought of as asset loading and frequently occurs when a game starts or changes stages/scenes.

1. TEMP

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

1. TEMP

async function start(scene) {

if (mLoopRunning) {

throw new Error("loop already running")

}

// Wait for any async requests before game-load

await map.waitOnPromises();

mCurrentScene = scene;

mCurrentScene.init();

mPrevTime = performance.now();

mLagTime = 0.0;

mLoopRunning = true;

mFrameID = requestAnimationFrame(loopOnce);

}

### Test the Asynchronous Shader Loading

You can now run the project with shaders being loaded asynchronously. Though the output and interaction experience are identical to the previous project, you now have a game engine that is much better equipped to manage the loading and accessing of external resources efficiently.

The rest of this chapter further develops and formalizes the interface between the client, in other words, MyGame, and the rest of the game engine. The goal is to define the interface to the client such that multiple instances can be created and interchanged during runtime. With this new interface, you will be able to define what a game level is and allow the game engine to load any level in any order.

# Game Level from a Scene File

The scene file is a formal interface between the game engine and its client because it triggers a sequence of function calls to create a playable game level. With a game level defined in a scene file, the game engine must first initiate asynchronous loading, wait for the load completion, and then initialize the client for the game loop. These steps present a complete functional interface between the game engine and the client. By examining and deriving the proper support for these steps, the interface between the game engine and its client can be refined.