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.

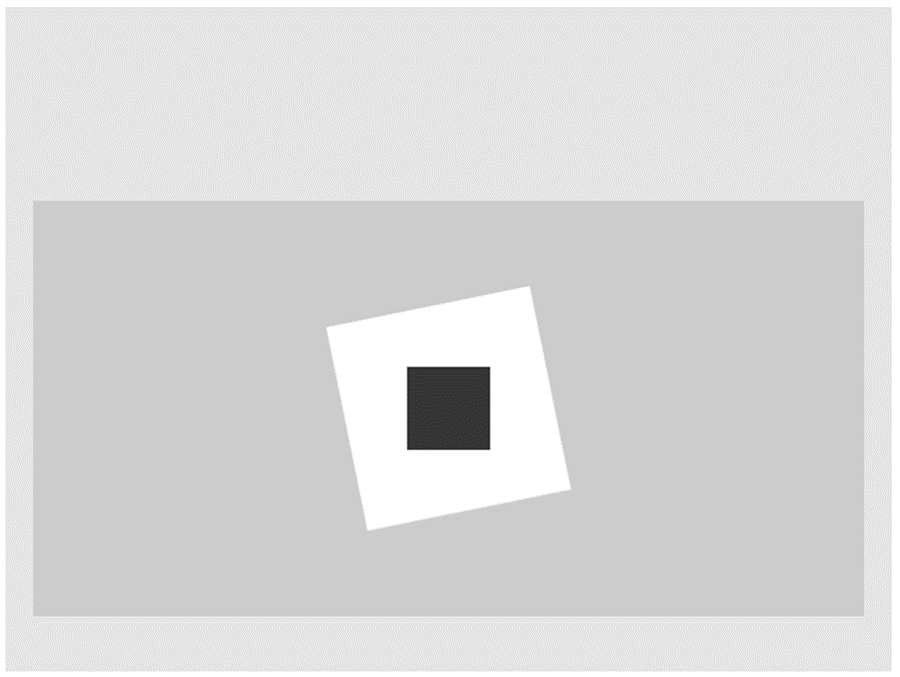


Figure 4-1. Running the Game Loop project

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 Engine\_Core or Engine\_VertexBuffer, as a property of the gEngine object in a file defined in the src/engine/core folder. The actual implementation is similar to the pseudocode listing discussed; for clarity it’s shown without the input support for now.

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 FPS = 60; // Frames per second

const MPF = 1000 / FPS; // Milliseconds per frame.

// 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 FPS is the frames per second discussed and that MPF is milliseconds per frame. It is important to try and maintain the game at an update interval of FPS.

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

// This function assumes it is sub-classed from MyGame

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 = Date.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 = Date.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. 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:

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);

}

The initialization is rather similar to previous examples, where a camera is defined and two squares are set up. The interesting change in this case is the last step, step F, where the game loop is started with MyGame as the parameter. Recall that the game loop \_runLoop() function is set up such that the update() and draw() functions of MyGame will be called.

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

// The update function, updates the application state. Make sure to \_NOT\_ draw

// anything from this function!

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 FPS, 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:
* A white square rotating while moving toward the right and reappearing when it reaches the right boundary
* A red square increasing in size and reducing to a size of 2 when the size reaches 5, thus appearing to be pulsing

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

// This is the draw function, make sure to setup proper drawing environment, and more

// importantly, make sure to \_NOT\_ change any state.

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. Export

export default MyGame;

1. onload

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 parameters to 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 \_runLoop() function to be called multiple times within a single kMPF interval. With the given \_runLoop() 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