## Implementing Common Components of Video Games

After completing this chapter, you will be able to:

- Control `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;
   draw();
   input();
   while( elapsedTime >= UPDATE_TIME_RATE ) {
    update();
    elapsedTime -= UPDATE_TIME_RATE;
   }
}
```

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 play input response and frames skipped. However, the game logic will continue to be correct.

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.
To ensure focusing on the core game loop operation, input will be ignored until the next project.

## Lab 4.1
## 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.

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

### *Loop Component*

The game loop component is a core game engine functionality and thus should be implemented similar to `vertex_buffer.js`, as 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 in the `src/engine/core` folder and name the file `loop.js`.

2. Define the following instance variables to keep track of frame rate, processing time in milliseconds per frame, the game loop's current run state, and a reference to the current scene as follows:

```
"use strict"
const kUPS = 60; // Updates per second
const kMPF = 1000 / kUPS; // 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 `kUPS` is the updates per second similar to the `FPS` discussed, and it is set to 60 or 60 updates per second. The time available for each update is simply 1/60 of a second. Since there are 1000 milliseconds in a second, the available time for each update in milliseconds is 1000 * (1/60), or `kMPF`.

**Note** When the game is running optimally, frame drawing and updates are both maintained at the same rate; `FPS` and `kUPS` can be thought of interchangeably. However, when lag occurs, the `loop` skips frame drawing and prioritizes updates. In this case, `FPS` will decrease, while `kUPS` will be maintained.

3. Add a function to run the loop as follows:

```
function loopOnce() {
  if (mLoopRunning) {
    // Step A: set up for next call to LoopOnce
    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 time elapsed since last loopOnce was executed
    let currentTime = performance.now();
    let elapsedTime = currentTime - mPrevTime;
    mPrevTime = currentTime;
    mLagTime += elapsedTime;
    // Step D: update the game the appropriate number of times.
    //      Update only every kMPF (1/60 of a second)
    //      If lag larger then update frames, update until caught up.
    while ((mLagTime >= kMPF) && mLoopRunning) {
          mCurrentScene.update();
          mLagTime -= kMPF;
    }
  }
}
```

**Note** The `performance.now()` is a JavaScript function that returns a timestamp in milliseconds.

Notice the similarity between the pseudocode examined previously and the steps B, C, and D of the `loopOnce()` function, 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 outermost while loop is implemented based on the HTML5 `requestAnimationFrame()` function call at step A. The `requestAnimationFrame()`

function will, at an approximated rate of 60 times per second, invoke the function pointer that is passed in as its parameter. In this case, the `loopOnce()` function will be called continuously at approximately 60 times per second. 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, if the system is lagging, multiple updates can occur during this single frame.

Note The `requestAnimationFrame()` function is an HTML5 utility provided by the browser that hosts your game. The precise behavior of this function is browser implementation dependent. 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 (e.g., for level transitions or the end of the game).

4. Declare a function to `start` the game loop. This function initializes the game or scene, the frame time variables, and the loop running flag before calling the first `requestAnimationFrame()` with the `loopOnce` function as its parameter to begin the game loop.

```
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);
}
```

5. Declare a function to `stop` the game loop. This function simply stops the loop by setting `mLoopRunning` to false and cancels the last requested animation frame.

```
function stop() {
  mLoopRunning = false;
  // make sure no more animation frames
  cancelAnimationFrame(mFrameID);
}
```

6. Lastly, remember to `export` the desired functionality to the rest of the game engine, in this case just the `start` and `stop` functions:

```
export {start, stop}
```

*Working with the Game Loop*

To test the game loop implementation, your game class must now implement `draw()`, `update()`, and `init()` functions. This is because to coordinate the beginning and the continual operation of your game, these functions are being called from the core of the game loop — the `init()` function is called from `loop.start()`, while the `draw()` and `update()` function s are called from `loop.loopOnce()`.

1. Edit your `my_game.js` file to provide access to the loop by importing from the module. Allowing game developer access to the game loop module is a temporary measure and will be corrected in later sections.

```
// Accessing engine internal is not ideal,
//      this must be resolved! (later)
import * as loop from "../engine/core/loop.js";
```

2. 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;
}
```

3. Add an initialization function to set up a camera and two `Renderable` objects:

```
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: Init the white Renderable: 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);
}
```

4. 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);
}
```

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

```
update() {
    // Simple game: move the white square and pulse the red
    let whiteXform = this.mWhiteSq.getXform();
    let deltaX = 0.05;
    // Step A: Rotate the white square
    if (whiteXform.getXPos() > 30) // 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:

a. A white square rotating while moving toward the right and upon reaching the right boundary wrapping around to the left boundary
b. A red square increasing in size and reducing to a size of 2 when the size reaches 5, thus appearing to be pulsing

6. Start the game loop from 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, and 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 in the core of the loop module, it is entirely possible for the `requestAnimationFrame()` function to invoke the `loopOnce()` function multiple times within a single `kMPF` interval. When this happens, the `draw()` function will be called multiples times without any `update()` function calls. In 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.

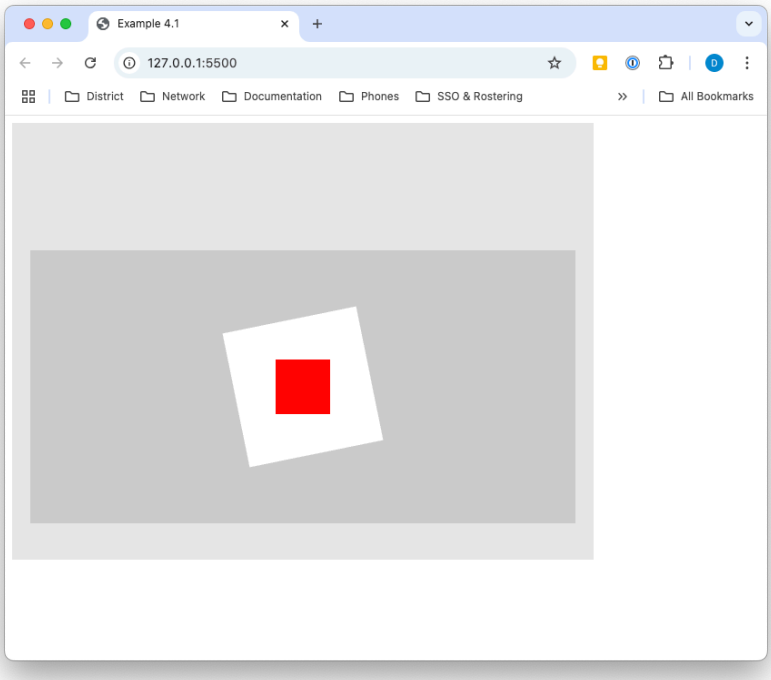


**Figure 4-1. Project Output**

## Lab 4.2
## 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.

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*

Recall that the loop component is part of the core of the game engine and should not be accessed by the client game developer. In contrast, a well-defined input module should support

the client game developer to query keyboard states without being distracted by any details. For this reason, the input module will be defined in the `src/engine` folder.

1. Create a new file in the `src/engine` folder and name it `input.js`.

2. Define a JavaScript dictionary to capture the key code mapping.

```
"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 unique numbers representing each keyboard character. Note that there are up to 222 unique keys. In the listing, only a small subset of the keys, those that are relevant to this project, are defined in the dictionary.

**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.

3. Create array instance variables for tracking the states of every key.

```
// Previous key state
var mKeyPreviousState = [];
// The pressed keys.
var mIsKeyPressed = [];
// Click events: once an event is set, it will remain there until polled
var mIsKeyClicked = [];
```

All three arrays define the state of every key as a boolean. The `mKeyPreviousState` records the key states from the previous update cycle, and the `mIsKeyPressed` records the current state of the keys. The key code entries of these two arrays are true when the corresponding keyboard keys are pressed, and 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 released to being pressed in two consecutive update cycles.

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 releasing it, then the duration of that entire second `KeyPress` for *A* is true, while `KeyClick` for *A* is true only once—the update cycle right after the key is pressed.

4. Define 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 these functions are called, the key code from the parameter is used to record the corresponding keyboard state changes. It is expected that the caller of these functions will pass the appropriate key code in the argument.

5. Add a function to initialize all the key states, and register the key event handlers with the browser. The `window.addEventListener()` function registers the `onKeyUp/Down()` event handlers with the browser such that the corresponding functions will be called when the player presses or releases keys on the keyboard.

```
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);
}
```

6. 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];
  }
}
```

7. Add public functions for inquires to current keyboard states to support the client game developer:

```
// Function for GameEngine programmer to test if a key is pressed down
function isKeyPressed(keyCode) {
  return mIsKeyPressed[keyCode];
}
function isKeyClicked(keyCode) {
  return mIsKeyClicked[keyCode];
}
```

8. Finally, export the public functions and key constants:

```
export {keys, init,
  update,
  isKeyClicked,
  isKeyPressed
}
```

*Modify the Engine to Support Keyboard Input*

To properly support input, before the game loop begins, the engine must first initialize the `mIsKeyPressed`, `mIsKeyClicked`, and `mKeyPreviousState` arrays. To properly capture the player actions, during gameplay from within the core of the game loop, these arrays must be updated accordingly.

1. Input state initialization: Modify `index.js` by importing the `input.js` module, adding the initialization of the input to the engine `init()` function, and adding the `input` module to the exported list to allow access from the client game developer.

```
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
}
```

2. To accurately capture keyboard state changes, the input component must be integrated with the core of the game loop. 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";
... identical to previous code ...
function loopOnce() {
  if (mLoopRunning) {
    ... identical to previous code ...
    // Step D: update the game the appropriate number of times.
    //      Update only every kMPF (1/60 of a second)
    //      If lag larger then update frames, update until caught up.
    while ((mLagTime >= kMPF) && mLoopRunning) {
          input.update();
          mCurrentScene.update();
          mLagTime -= kMPF;
  }
  }
}
```

*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() {
   // Simple game: 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) { // 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 continuously key press events and thus will not cause the white square to continuously rotate. Step C tests for the pressing and holding of the down-arrow key to pulse the red square.

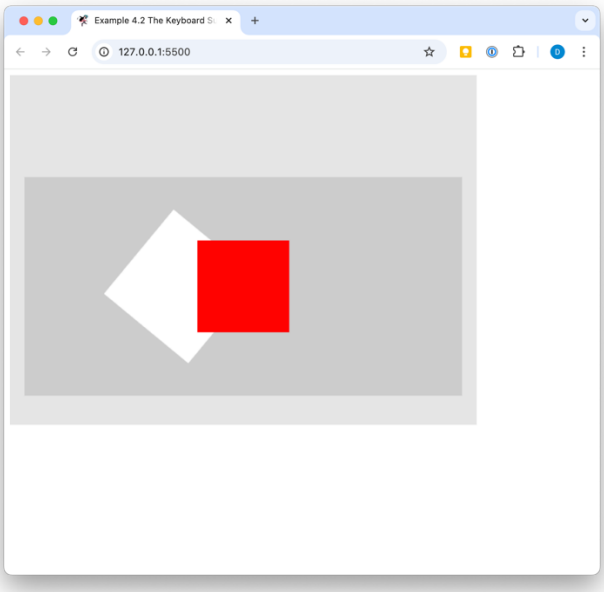


**Figure 4-2. Canvas with keyboard controlled Renderables**

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 upward, A leftward, S downward, and D rightward.

## Resource Management and Asynchronous Loading

Video games typically utilize a multitude of artistic assets, or resources, including audio clips and images. The required resources to support a game can be large. Additionally, it is important to maintain the independence between the resources and the actual game such that they can be updated independently, for example, changing the background audio without changing the game itself. For these reasons, game resources are typically stored externally on a system hard drive or a server across the network. Being stored external to the game, the resources are sometimes referred to as external resources or assets.

After a game begins, external resources must be explicitly loaded. For efficient memory utilization, 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, at any instance in a game, only a portion of resources are kept in memory, where the loading operations are strategically executed to avoid interrupting the game. In most cases, resources required in each level are kept in memory during the gameplay of that level. With this approach, external resource loading can occur during level transitions where players are expecting a new game environment and are more likely to tolerate slight delays for loading.

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 management, which is 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 resource 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 a 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.

**Lab 4.3**
**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.

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

**Right-arrow key**: Moves the white square toward the 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

Note For more information about asynchronous JavaScript operations, you can refer to many excellent resources online, for example, https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous.

## *Add Resource Map Component to the Engine*

The `resource_map` engine component manages resource loading, storage, and retrieval after the resources are loaded. These operations are internal to the game engine and should not be accessed by the game engine client. As in the case of all core engine components, for example, the game loop, the source code file is created in the `src/engine/core` folder. The details are as follows.

1. Create a new file in the `src/engine/core` folder and name it `resource_map.js`.

2. Define the `MapEntry` class to support reference counting of loaded resources. Reference counting is essential to avoid multiple loading or premature unloading of a resource.

```
class MapEntry {
  constructor(data) {
    this.mData = data;
    this.mRefCount = 1;
  }
  decRef() { this.mRefCount--; }
  incRef() { this. mRefCount++; }
  set(data) { this.mData = data;}
  data() { return this.mData; }
  canRemove() { return (this.mRefCount == 0); }
}
```

3. Define a key-value pair map, `mMap`, for storing and retrieving of resources and an array, `mOutstandingPromises`, to capture all outstanding asynchronous loading operations:

```
let mMap = new Map();
let mOutstandingPromises = [];
```

**Note** A JavaScript `Map` object holds a collection of key-value pairs.

4. Define functions for querying the existence of, retrieving, and setting a resource. Notice that as suggested by the variable name of the parameter, `path`, it is expected that the full path to the external resource file will be used as the key for accessing the corresponding resource, for example, using the path to the `src/glsl_shaders/simple_vs.glsl` file as the key for accessing the content of the file.

```
function has(path) { return mMap.has(path) }
function get(path) {
  if (!has(path)) {
   throw new Error("Error [" + path + "]: not loaded");
  }
  return mMap.get(path).data();
}
function set(key, value) { mMap.get(key).set(value); }
```

5. Define functions to indicate that loading has been requested, increase the reference count of a loaded resource, and to properly unload a resource. Due to the asynchronous nature of the loading operation, a load request will result in an empty `MapEntry` which will be updated when the load operation is completed sometime in the future. Note that each unload request will decrease the reference count and may or may not result in the resource being unloaded.

```
function loadRequested(path) {
  mMap.set(path, new MapEntry(null));
}
function incRef(path) {
  mMap.get(path).incRef();
}
function unload(path) {
  let entry = mMap.get(path);
  entry.decRef();
  if (entry.canRemove())
   mMap.delete(path)
  return entry.canRemove();
}
```

6. Define a function to append an ongoing asynchronous loading operation to the `mOutstandingPromises` array

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

7. Define a loading function, `loadDecodeParse()`. If the resource is already loaded, the corresponding reference count is incremented. Otherwise, the function first issues a `loadRequest()` to create an empty `MapEntry` in `mMap`. The function then creates an HTML5 `fetch` promise, using the path to the resource as key, to asynchronously fetch the external resource, decode the network packaging, parse the results into a proper format, and update the results into the created `MapEntry`. This created promise is then pushed into the `mOutstandingPromises` array.

```
// generic loading function,
//   Step 1: fetch from server
//   Step 2: decodeResource on the loaded package
//   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 fetchPromise = null;
  if (!has(path)) {
   loadRequested(path);
   fetchPromise =  fetch(path)
         .then(res => decodeResource(res) )
         .then(data => parseResource(data) )
         .then(data => { return set(path, data) } )
         .catch(err => { throw err });
   pushPromise(fetchPromise);
  } else {
   incRef(path);  // increase reference count
  }
  return fetchPromise;
}
```

Notice that the decoding and parsing functions are passed in as parameters and thus are dependent upon the actual resource type that is being fetched. For example, the decoding and parsing of simple text, XML (Extensible Markup Language)-formatted text, audio clips, and images all have distinct requirements. It is the responsibility of the actual resource loader to define these functions.

The HTML5 `fetch()` function returns a JavaScript `promise` object. A typical JavaScript `promise` object contains operations that will be completed in the future. A `promise` is `fulfilled` when the operations are completed. In this case, the `fetchPromise` is

`fulfilled` when the `path` is properly fetched, decoded, parsed, and updated into the corresponding `MapEntry`. This `promise` is being kept in the `mOutstandingPromises` array. Note that by the end of the `loadDecodeParse()` function, the asynchronous `fetch()` loading operation is issued and ongoing but not guaranteed to be completed. In this way, the `mOutstandingPromises` is an array of ongoing and unfulfilled, or outstanding, promises.

8. Define a JavaScript `async` function to block the execution and wait for all outstanding promises to be fulfilled, or wait for all ongoing asynchronous loading operations to be completed:

```
// will block, wait for all outstanding promises complete
// before continue
async function waitOnPromises() {
  await Promise.all(mOutstandingPromises);
  mOutstandingPromises = []; // remove all
}
```

Note The JavaScript `async/await` keywords are paired where only `async` functions can `await` for a `promise`. The `await` statement blocks and returns the execution back to the caller of the `async` function. When the `promise` being waited on is fulfilled, execution will continue to the end of the `async` function.

9. Finally, export functionality to the rest of the game engine:

```
export {has, get, set,
  loadRequested, incRef, loadDecodeParse,
  unload,
  pushPromise, waitOnPromises}
```

Notice that although the storage-specific functionalities—query, get, and set—are well defined, `resource_map` is actually not capable of loading any specific resources. This module is designed to be utilized by resource type–specific modules where the decoding and parsing functions can be properly defined. In the next subsection, a text resource loader is defined to demonstrate this idea.

### *Define a Text Resource Module*

This section will define a text module that utilizes 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. Replacing synchronous with asynchronous loading support is a significant upgrade to the 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/resources` folder and name it `text.js`.

3. Import the core resource management and reuse the relevant functionality from `resource_map`:

```
"use strict"
import * as map from "../core/resource_map.js";
// functions from resource_map
let unload = map.unload;
let has = map.has;
let get = map.get;
```

4. Define the text decoding and parsing functions for `loadDecodeParse()`. Notice that there are no requirements for parsing the loaded text, and thus, the text parsing function does not perform any useful operation.

```
function decodeText(data) {
  return data.text();
}
function parseText(text) {
  return text;
}
```

5. Define the `load()` function to call the `resource_map` `loadDecodeParse()` function to trigger the asynchronous `fetch()` operation:

```
function load(path) {
  return map.loadDecodeParse(path, decodeText, parseText);
}
```

6. Export the functionality to provide access to the rest of the game engine:

```
export {has, get, load, unload}
```

7. Lastly, remember to update the defined functionality for the client in the `index.js`:

```
import * as text from "./resources/text.js";
... identical to previous code ...
export default {
  // resource support
  text,
  ... identical to previous code ...
}
```

**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. Recall that a single instance of `SimpleShader` object is created in the `shader_resources` module and shared among all `Renderables`. You can now asynchronously load the GLSL shader files before the creation of the `SimpleShader` object.

1. Edit `shader_resources.js` and import functionality from the text and `resource_map` modules:

```
import * as text from "../resources/text.js";
import * as map from "./resource_map.js";
```

2. Replace the content of the `init()` function. Define a JavaScript `promise`, `loadPromise`, to load the two GLSL shader files asynchronously, and when the loading is completed, trigger the calling of the `createShaders()` function. Store the `loadPromise` in the `mOutstandingPromises` array of the `resource_map` by calling the `map.pushPromise()` function:

```
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);
  }
```

Notice that after the `shader_resources init()` function, the loading of the two GLSL shader files would have begun. At that point, it is not guaranteed that the loading operations are completed and the `SimpleShader` object may not have been created. However, the promise that is based on the completion of these operations is stored in the `resource_map mOutstandingPromises` array. For this reason, it is guaranteed that these operations must have completed by the end of the `resource_map waitOnPromises()` function.

*Modify SimpleShader to Retrieve Shader Files*

With the understanding that the GLSL shader files are already loaded, the changes to the `SimpleShader` class 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. Edit the `simple_shader.js` file and add an import from the `text` module for retrieving the content of the GLSL shaders:

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

2. Since no loading operations are required, you should change the `loadAndCompileShader()` function name to simply `compileShader()` and replace the file-loading commands by `text` resource retrievals. Notice that the synchronous loading operations are replaced by a single call to `text.get()` to retrieve the file content based on the `filePath` or the unique resource name for the shader file.

```
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;
    }
    ... identical to previous code ...
}
```

3. 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*

With outstanding loading operations and incomplete shader creation, a client's game cannot be initialized because without `SimpleShader`, `Renderable` objects cannot be properly created. For this reason, the game engine must wait for all outstanding promises to be fulfilled before proceeding to initialize the client's game. Recall that client's game initialization is performed in the game loop `start()` function, right before the beginning of the first loop iteration.

1. Edit the `loop.js` file and import from the `resource_map` module:

```
import * as map from "./resource_map.js";
```

2. Modify the `start()` function to be an `async` function such that it is now possible to issue `await` and hold the execution by calling `map.waitOnPromises()` to wait for the fulfilment of all outstanding promises:

```
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.

The rest of this chapter further develops and formalizes the interface between the client, `MyGame`, and the rest of the game engine. The goal is to define the interface to the client such that multiple game-level 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.

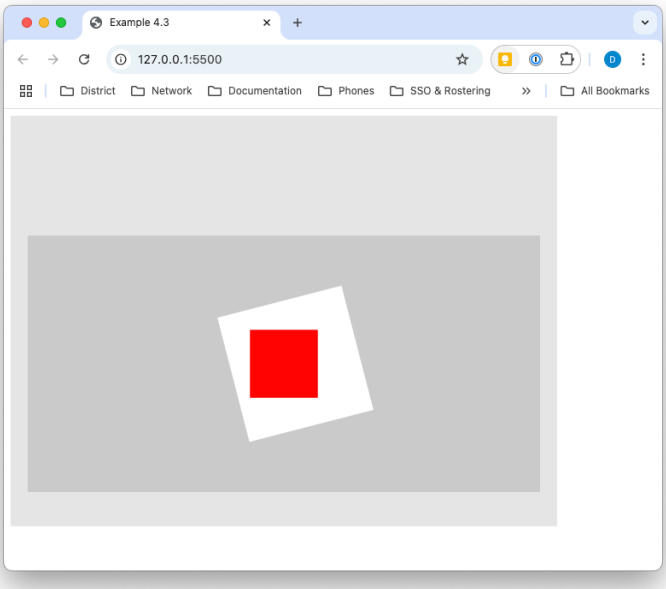


**Figure 4-3. Canvas with keyboard controlled Renderables**

**Game Level from a Scene File**

The operations involved in initiating a game level from a scene file can assist in the derivation and refinement of the formal interface between the game engine and its client. 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.

**Lab 4.4**
**The Scene File Project**

This project uses the loading of a scene file as the vehicle to examine the necessary public methods for a typical game level.

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 introduce the protocol for supporting asynchronous loading of the resources of a game
To develop the proper game engine support for the protocol
To identify and define the public interface methods for a general game level

While the parsing and loading process of a scene file is interesting to a game engine designer, the client should never need to concern themselves with these details. This project aims at developing a well-defined interface between the engine and the client. This interface will hide the complexity of the engine internal core from the client and thus avoid situations such as requiring access to the `loop` module from `MyGame` in the first project of this chapter.

**The Scene File**

Instead of hard-coding the creation of all objects to a game in the `init()` function, the information can be encoded in a file, and the file can be loaded and parsed during runtime. The advantage of such encoding in an external file is the flexibility to modify a scene without the need to change the game source code, while the disadvantages are the complexity and time required for loading and parsing. In general, the importance of flexibility dictates that most game engines support the loading of game scenes from a file.

Objects in a game scene can be defined in many ways. The key decision factors are that the format can properly describe the game objects and be easily parsed. Extensible Markup Language (XML) is well suited to serve as the encoding scheme for scene files.

*Define an XML Resource Module*

In order to support an XML-encoded scene file, you first need to expand the engine to support the asynchronous loading of an XML file resource. Similar to the `text` resource module, an XML resource module should also be based on the `resource_map`: store the loaded XML content in `mMap` of the `resource_map`, and define the specifics for decoding and parsing for the calling of the `loadDecodeParse()` function of the `resource_map`.

1. Define a new file in the `src/engine/resources` folder and name it `xml.js`. Edit this file and import the core resource management functionality from the `resource_map`.

```
"use strict"
import * as map from "../core/resource_map.js";
// functions from resource_map
let unload = map.unload;
let has = map.has;
let get = map.get;
```

2. Instantiate an XML `DOMParser`, define the decoding and parsing functions, and call the `loadDecodeParse()` function of the `resource_map` with the corresponding parameters to initiate the loading of the XML file:

```
let mParser = new DOMParser();
function decodeXML(data) {
  return data.text();
}
function parseXML(text) {
  return mParser.parseFromString(text, "text/xml");
}
function load(path) {
  return map.loadDecodeParse(path, decodeXML, parseXML);
}
```

3. Remember to export the defined functionality:

```
export {has, get, load, unload}
```

4. Lastly, remember to export the defined functionality for the client in the `index.js`:

```
import * as xml from "./resources/xml.js";
... identical to previous code ...
export default {
  // resource support
  text, xml,
  ... identical to previous code ...
}
```

The newly defined `xml` module can be conveniently accessed by the client and used in a similar fashion as the `text` module in loading external XML-encoded text files.

**Note** The JavaScript `DOMParser` provides the ability to parse XML or HTML text strings.

**Modify the Engine to Integrate Client Resource Loading**

The scene file is an external resource that is being loaded by the client. With asynchronous operations, the game engine must stop and wait for the completion of the load process before it can initialize the game. This is because the game initialization will likely require the loaded resources.

*Coordinate Client Load and Engine Wait in the Loop Module*

Since all resource loading and storage are based on the same `resource_map`, the client issuing of the load requests and the engine waiting for the load completions can be coordinated in the `loop.start()` function in the file `loop.js` as follows:

```
async function start(scene) {
  if (mLoopRunning) {
      throw new Error("loop already running")
  }
  mCurrentScene = scene;
  mCurrentScene.load();
  // Wait for any async requests before game-load
  await map.waitOnPromises();
  mCurrentScene.init();
  mPrevTime = performance.now();
  mLagTime = 0.0;
  mLoopRunning = true;
  mFrameID = requestAnimationFrame(loopOnce);
}
```

Note that this function is exactly two lines different from the previous project—`mCurrentScene` is assigned a reference to the parameter, and the client's `load()` function is called **before** the engine waits for the completion of all asynchronous loading operations.

**Derive a Public Interface for the Client**

Though slightly involved, the details of XML-parsing specifics are less important than the fact that XML files can now be loaded. It is now possible to use the asynchronous loading of an external resource to examine the required public methods for interfacing a game level to the game engine.

**Public Methods of MyGame**

While the game engine is designed to facilitate the building of games, the actual state of a game is specific to each individual client. In general, there is no way for the engine to anticipate the required operations to initialize, update, or draw any particular game. For this reason, such operations are defined to be part of the public interface between the game engine and the client. At this point, it is established that `MyGame` should define the following:

- `constructor()`: For declaring variables and defining constants.
- `init()`: For instantiating the variables and setting up the game scene. This is called from the `loop.start()` function before the first iteration of the game loop.
- `draw()`/`update()`: For interfacing to the game loop with these two functions being called continuously from within the core of the game loop, in the `loop.loopOnce()` function.

With the requirement of loading a scene file, or any external resources, two additional public methods should be defined:

- `load()`: For initiating the asynchronous loading of external resources, in this case, the scene file. This is called from the `loop.start()` function before the engine waits for the completion of all asynchronous loading operations.
- `unload()`: For unloading of external resources when the game has ended. Currently, the engine does not attempt to free up resources. This will be rectified in the next project.

**Implement the Client**

You are now ready to create an XML-encoded scene file to test external resource loading by the client and to interface to the client with game engine based on the described public methods.

*Define a Scene File*

Define a simple scene file to capture the game state from the previous project:

1. Create a new folder at the same level as the `src` folder and name it `assets`. This is the folder where all external resources, or assets, of a game will be stored including the scene files, audio clips, texture images, and fonts.

**Tip** It is important to differentiate between the `src/engine/resources` folder that is created for organizing game engine source code files and the `assets` folder that you just created for storing client resources. Although GLSL shaders are also loaded at runtime, they are considered as source code and will continue to be stored in the `src/glsl_shaders` folder.

2. Create a new file in the `assets` folder and name it `scene.xml`. This file will store the client's game scene. Add the following content. The listed XML content describes the same scene as defined in the `init()` functions from the previous `MyGame` class.

```
<MyGameLevel>
<!--  *** be careful!! comma (,) is not a supported syntax!!  -->
<!--  make sure there are no comma in between attributes -->
<!--  e.g., do NOT do:  PosX="20", PosY="30" -->
<!--  notice the "comma" between PosX and PosY: Syntax error! -->
  <!-- cameras -->
  <!-- Viewport: x, y, w, h -->
  <Camera CenterX="20" CenterY="60" Width="20"
          Viewport="20 40 600 300"
          BgColor="0.8 0.8 0.8 1.0"
  />
  <!-- Squares Rotation is in degree -->
  <Square PosX="20" PosY="60" Width="5" Height="5"
          Rotation="30" Color="1 1 1 1" />
  <Square PosX="20" PosY="60" Width="2" Height="2"
          Rotation="0"  Color="1 0 0 1" />
</MyGameLevel>
```

**Tip** The JavaScript XML parser does not support delimiting attributes with commas.

*Parse the Scene File*

A specific parser for the listed XML scene file must be defined to extract the scene information. Since the scene file is specific to a game, the parser should also be specific to the game and be created within the `my_game` folder.

1. Create a new folder in the `src/my_game` folder and name it `util`. Add a new file in the `util` folder and name it `scene_file_parser.js`. This file will contain the specific parsing logic to decode the listed scene file.

2. Define a new class, name it `SceneFileParser`, and add a constructor with code as follows:

```
"use strict";
import engine from "../../engine/index.js";
class SceneFileParser {
  constructor (xml) {
    this.xml = xml
  }
  ... implementation to follow ...
}
```

**Note** that the xml parameter is the actual content of the loaded XML file.

**Note** The following XML parsing is based on JavaScript XML API. Please refer to https://www.w3schools.com/xml for more details.

3. Add a function to the `SceneFileParser` to parse the details of the `Camera` from the `xml` file you created:

```
parseCamera() {
  let camElm = getElm(this.xml, "Camera");
  let cx = Number(camElm[0].getAttribute("CenterX"));
  let cy = Number(camElm[0].getAttribute("CenterY"));
  let w = Number(camElm[0].getAttribute("Width"));
  let viewport = camElm[0].getAttribute("Viewport").split(" ");
  let bgColor = camElm[0].getAttribute("BgColor").split(" ");
  // make sure viewport and color are number
  let j;
  for (j = 0; j < 4; j++) {
      bgColor[j] = Number(bgColor[j]);
      viewport[j] = Number(viewport[j]);
  }
  let cam = new engine.Camera(
      vec2.fromValues(cx, cy),  // position of the camera
      w,                        // width of camera
      viewport                  // viewport (orgX, orgY, width, height)
      );
  cam.setBackgroundColor(bgColor);
  return cam;
}
```

The camera parser finds a camera element and constructs a `Camera` object with the retrieved information. Notice that the viewport and background color are arrays of four numbers. These are input as strings of four numbers delimited by spaces. Strings can be split into arrays, which is the case here with the space delimiter. The JavaScript `Number()` function ensures that all strings are converted into numbers.

4. Add a function to the `SceneFileParser` to parse the details of the squares from the `xml` file you created:

```
parseSquares(sqSet) {
  let elm = getElm(this.xml, "Square");
  let i, j, x, y, w, h, r, c, sq;
  for (i = 0; i < elm.length; i++) {
     x = Number(elm.item(i).attributes.getNamedItem("PosX").value);
     y = Number(elm.item(i).attributes.getNamedItem("PosY").value);
     w = Number(elm.item(i).attributes.getNamedItem("Width").value);
     h = Number(elm.item(i).attributes.getNamedItem("Height").value);
     r = Number(elm.item(i).attributes.getNamedItem("Rotation").value);
     c = elm.item(i).attributes.getNamedItem("Color").value.split(" ");
     sq = new engine.Renderable();
     // make sure color array contains numbers
     for (j = 0; j < 4; j++) {
         c[j] = Number(c[j]);
```

```
        }
        sq.setColor(c);
        sq.getXform().setPosition(x, y);
        sq.getXform().setRotationInDegree(r); // In Degree
        sq.getXform().setSize(w, h);
        sqSet.push(sq);
    }
}
```

This function parses the XML file to create `Renderable` objects to be placed in the array that is passed in as a parameter.

5. Add a function outside the `SceneFileParser` to parse for contents of an XML element:

```
function getElm(xmlContent, tagElm) {
  let theElm = xmlContent.getElementsByTagName(tagElm);
  if (theElm.length === 0) {
      console.error("Warning: Level element:[" +
                    tagElm + "]: is not found!");
  }
  return theElm;
}
```

6. Finally, export the `SceneFileParser`:

```
export default SceneFileParser;
```

*Implement MyGame*

The implementations of the described public functions for this project are as follows:

1. Edit `my_game.js` file and import the `SceneFileParser`:

```
import SceneFileParser from "./util/scene_file_parser.js";
```

2. Modify the `MyGame` constructor to define the scene file path, the array `mSqSet` for storing the `Renderable` objects, and the `camera`:

```
constructor() {
  // scene file name
  this.mSceneFile = "assets/scene.xml";
```

```
  // all squares
  this.mSqSet = [];        // these are the Renderable objects
  // The camera to view the scene
  this.mCamera = null;
}
```

3. Change the `init()` function to create objects based on the scene parser. Note the retrieval of the XML file content via the `engine.xml.get()` function where the file path to the scene file is used as the key.

```
init() {
  let sceneParser = new SceneFileParser(
                        engine.xml.get(this.mSceneFile));
  // Step A: Read in the camera
  this.mCamera = sceneParser.parseCamera();
  // Step B: Read all the squares
  sceneParser.parseSquares(this.mSqSet);
}
```

4. The `draw` and `update` functions are similar to the previous examples with the exception of referencing the corresponding array elements.

```
draw() {
   // Step A: clear the canvas
   engine.clearCanvas([0.9, 0.9, 0.9, 1.0]);
   // Step B: activate the drawing camera
   this.mCamera.setViewAndCameraMatrix();
   // Step C: draw all the squares
   let i;
   for (i = 0; i < this.mSqSet.length; i++)
    this.mSqSet[i].draw(this.mCamera);
}
update() {
   // simple game: move the white square and pulse the red
   let xform = this.mSqSet[0].getXform();
   let deltaX = 0.05;
   // Step A: test for white square movement
   if (engine.input.isKeyPressed(engine.input.keys.Right)) {
    if (xform.getXPos() > 30) { // this is the right-bound of the window
          xform.setPosition(10, 60);
    }
    xform.incXPosBy(deltaX);
   }
   // Step B: test for white square rotation
   if (engine.input.isKeyClicked(engine.input.keys.Up)) {
    xform.incRotationByDegree(1);
   }
   xform = this.mSqSet[1].getXform();
   // Step C: test for pulsing the red square
   if (engine.input.isKeyPressed(engine.input.keys.Down)) {
```

```
    if (xform.getWidth() > 5) {
          xform.setSize(2, 2);
    }
    xform.incSizeBy(0.05);
   }
}
```

5. Lastly, define the functions to load and unload the scene file.

```
load() {
  engine.xml.load(this.mSceneFile);
}
unload() {
  // unload the scene file and loaded resources
  engine.xml.unload(this.mSceneFile);
}
```

You can now run the project and see that it behaves the same as the previous two projects. While this may not seem interesting, through this project, a simple and well-defined interface between the engine and the client has been derived where the complexities and details of each are hidden. Based on this interface, additional engine functionality can be introduced without the requirements of modifying any existing clients, and at the same time, complex games can be created and maintained independently from engine internals. The details of this interface will be introduced in the next project.

Before continuing, you may notice that the `MyGame.unload()` function is never called. This is because in this example the game loop never stopped cycling and `MyGame` is never unloaded. This issue will be addressed in the next project.

**Scene Object: Client Interface to the Game Engine**

At this point, in your game engine, the following is happening:

- The `window.onload` function initializes the game engine and calls the `loop.start()` function, passing in `MyGame` as a parameter.
- The `loop.start()` function, through the `resource_map`, waits for the completion of all asynchronous loading operations before it calls to initialize `MyGame` and starts the actual game loop cycle.

From this discussion, it is interesting to recognize that any object with the appropriately defined public methods can replace the `MyGame` object. Effectively, at any point, it is possible to call the `loop.start()` function to initiate the loading of a new scene. This section expands on this idea by introducing the `Scene` object for interfacing the game engine with its clients.

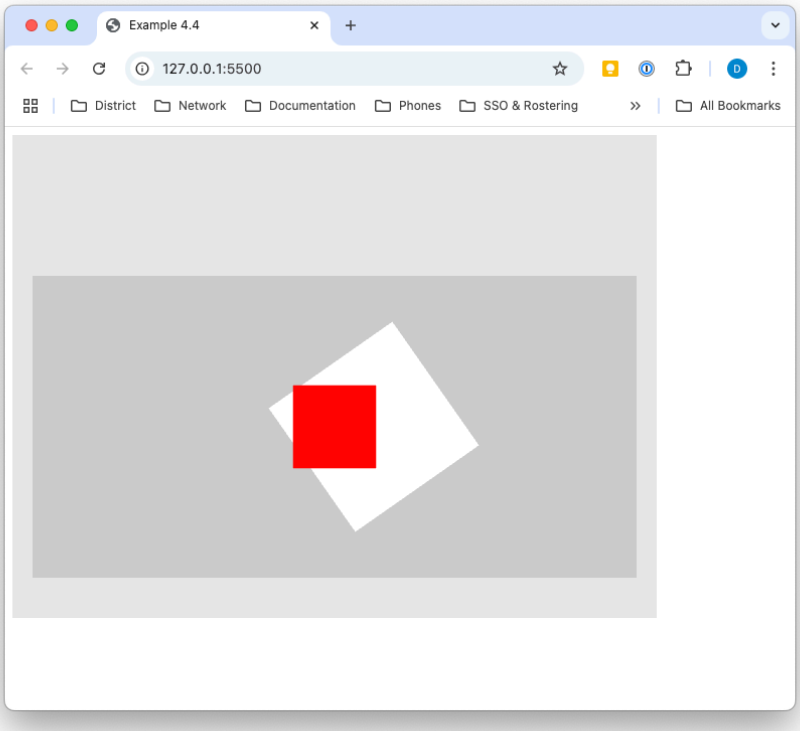


**Figure 4-4. Canvas with keyboard controlled Renderables**

**Lab 4.5**
**The Scene Objects Project**

This project defines the `Scene` as an abstract superclass for interfacing with your game engine. From this project on, all client code must be encapsulated in subclasses of the abstract `Scene` class, and the game engine will be able to interact with these classes in a coherent and well-defined manner.

There are two distinct levels in this project: the `MyGame` level with a blue rectangle drawn above a red square over a gray background and the `BlueLevel` level with a red rectangle drawn above a rotated white square over a dark blue background. For simplicity, the controls for both levels are the same.

> **Left-/right-arrow key**: Move the front rectangle left and right
> **Q key**: Quits the game

Notice that on each level, moving the front rectangle toward the left to touch the left boundary will cause the loading of the other level. The `MyGame` level will cause `BlueLevel` to be loaded, and `BlueLevel` will cause the `MyGame` level to be loaded.

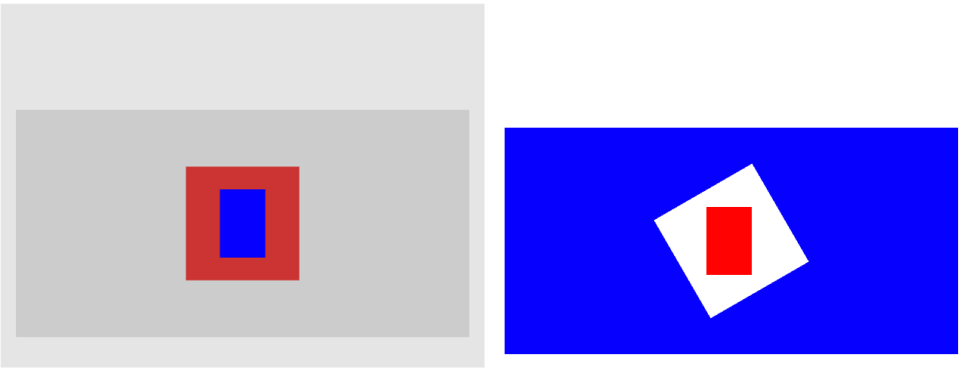

**Figure 4-5. Two Scenes**

The goals of the project are as follows:

> To define the abstract Scene class to interface to the game engine
> To experience game engine support for scene transitions
> To create scene-specific loading and unloading support

## *The Abstract Scene Class*

Based on the experience from the previous project, an abstract Scene class for encapsulating the interface to the game engine must at the very least define these functions: `init()`, `draw()`, `update()`, `load()`, and `unload()`. Missing from this list is the support for level transitions to `start`, advance to the `next` level, and, if desired, to `stop` the game.

1. Create a new JavaScript file in the `src/engine` folder and name it `scene.js`, and import from the `loop` module and the engine access file `index.js`. These two modules are required because the `Scene` object must start and end the game loop when the game level begins and ends, and the engine must be cleaned up if a level should decide to terminate the game.

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

**Note** The game loop must not be running before a `Scene` has begun. This is because the required resources must be properly loaded before the `update()` function of the `Scene` can be called from the running game loop. Similarly, unloading of a level can only be performed after a game loop has stopped running.

2. Define JavaScript Error objects for warning the client in case of misuse:

```
const kAbstractClassError = new Error("Abstract Class")
const kAbstractMethodError = new Error("Abstract Method")
```

3. Create a new class named Scene and export it:

```
class Scene { ... implementation to follow ... }
export default Scene;
```

4. Implement the constructor to ensure only subclasses of the Scene class are instantiated:

```
constructor() {
  if (this.constructor === Scene) {
      throw kAbstractClassError
  }
}
```

5. Define scene transition functions: `start()`, `next()`, and `stop()`. The `start()` function is an async function because it is responsible for starting the game loop, which in turn is waiting for all the asynchronous loading to complete. Both the `next()` and the `stop()` functions stop the game loop and call the `unload()` function to unload the loaded resources. The difference is that the `next()` function is expected to be overwridden and called from a subclass where after unloading the current scene, the subclass can proceed to advance to the next level. After unloading, the `stop()` function assumes the game has terminated and proceeds to clean up the game engine.

```
async start() {
  await loop.start(this);
}
next() {
  loop.stop();
  this.unload();
}
stop() {
  loop.stop();
  this.unload();
  engine.cleanUp();
}
```

6. Define the rest of the derived interface functions. Notice that the `Scene` class is an abstract class because all of the interface functions are empty. While a subclass can choose to only implement a selective subset of the interface functions, the `draw()` and `update()` functions are not optional because together they form the central core of a level.

```
init() { /* to initialize the level (called from loop.start()) */ }
load() { /* to load necessary resources */ }
unload() { /* unload all resources */ }
// draw/update must be over-written by subclass
draw() { throw kAbstractMethodError; }
update() { throw kAbstractMethodError; }
```

Together these functions present a protocol to interface with the game engine. It is expected that subclasses will override these functions to implement the actual game behaviors.

**Note** JavaScript does not support abstract classes. The language does not prevent a game programmer from instantiating a `Scene` object; however, the created instance will be completely useless, and the error message will provide them with a proper warning.

**Modify Game Engine to Support the Scene Class**

The game engine must be modified in two important ways. First, the game engine access file, `index.js`, must be modified to export the newly introduced symbols to the client as is done with all new functionality. Second, the `Scene.stop()` function introduces the possibility of stopping the game and handles the cleanup and resource deallocation required.

*Export the Scene Class to the Client*

Edit the `index.js` file to import from `scene.js` and export `Scene` for the client:

```
... identical to previous code ...
import Scene from "./scene.js";
```

```
... identical to previous code ...
export default {
  ... identical to previous code ...
  Camera, Scene, Transform, Renderable,
  ... identical to previous code ...
}
```

*Implement Engine Cleanup Support*

It is important to release the allocated resources when the game engine shuts down. The cleanup process is rather involved and occurs in the reverse order of system component initialization.

1. Edit `index.js` once again, this time to implement support for game engine cleanup. Import from the `loop` module, and then define and export the `cleanup()` function.

```
... identical to previous code ...
import * as loop from "./core/loop.js";
... identical to previous code ...
function cleanUp() {
  loop.cleanUp();
  input.cleanUp();
  shaderResources.cleanUp();
  vertexBuffer.cleanUp();
  glSys.cleanUp();
}
... identical to previous code ...
export default {
  ... identical to previous code ...
  init, cleanUp, clearCanvas
  ... identical to previous code ...
}
```

**Note** Similar to other core engine internal components, such as `gl` or `vertex_buffer`, `loop` should not be accessed by the client. For this reason, `loop` module is imported but not exported by `index.js`, imported such that game loop cleanup can be invoked, not exported, such that the client can be shielded from irrelevant complexity within the engine.

Notice that none of the components have defined their corresponding cleanup functions. You will now remedy this. In each of the following cases, make sure to remember to export the newly defined `cleanup()` function when appropriate.

2. Edit `loop.js` to define and export a `cleanUp()` function to stop the game loop and unload the currently active scene:

```
... identical to previous code ...
```

```
function cleanUp() {
  if (mLoopRunning) {
      stop();
      // unload all resources
      mCurrentScene.unload();
      mCurrentScene = null;
  }
}
export {start, stop, cleanUp}
```

3. Edit `input.js` to define and export a `cleanUp()` function. For now, no specific resources need to be released.

```
... identical to previous code ...
function cleanUp() {}  // nothing to do for now
export {keys, init, cleanUp,
... identical to previous code ...
```

4. Edit `shader_resources.js` to define and export a `cleanUp()` function to clean up the created shader and unload its source code:

```
... identical to previous code ...
function cleanUp() {
  mConstColorShader.cleanUp();
  text.unload(kSimpleVS);
  text.unload(kSimpleFS);
}
export {init, cleanUp, getConstColorShader}
```

5. Edit `simple_shader.js` to define the `cleanUp()` function for the `SimpleShader` class to release the allocated WebGL resources:

```
cleanUp() {
  let gl = glSys.get();
  gl.detachShader(this.mCompiledShader, this.mVertexShader);
  gl.detachShader(this.mCompiledShader, this.mFragmentShader);
  gl.deleteShader(this.mVertexShader);
  gl.deleteShader(this.mFragmentShader);
  gl.deleteProgram(this.mCompiledShader);
}
```

6. Edit `vertex_buffer.js` to define and export a `cleanUp()` function to delete the allocated buffer memory:

```
... identical to previous code ...
function cleanUp() {
  if (mGLVertexBuffer !== null) {
      glSys.get().deleteBuffer(mGLVertexBuffer);
      mGLVertexBuffer = null;
  }
}
export {init, get, cleanUp}
```

7. Lastly, edit `gl.js` to define and export a `cleanUp()` function to inform the player that the engine is now shut down:

```
... identical to previous code ...
function cleanUp() {
  if ((mGL == null) || (mCanvas == null))
      throw new Error("Engine cleanup: system is not initialized.");
  mGL = null;
  // let the user know
  mCanvas.style.position = "fixed";
  mCanvas.style.backgroundColor = "rgba(200, 200, 200, 0.5)";
  mCanvas = null;
  document.body.innerHTML +=
          "<br><br><h1>End of Game</h1><h1>GL System Shut Down</h1>";
}
export {init, get, cleanUp}
```

*Test the Scene Class Interface to the Game Engine*

With the abstract Scene class definition and the resource management modifications to the game engine core components, it is now possible to stop an existing scene and load a new scene at will. This section cycles between two subclasses of the Scene class, `MyGame` and `BlueLevel`, to illustrate the loading and unloading of scenes.

For simplicity, the two test scenes are almost identical to the `MyGame` scene from the previous project. In this project, `MyGame` explicitly defines the scene in the `init()` function, while the `BlueScene`, in a manner identical to the case in the previous project, loads the scene content from the `blue_level.xml` file located in the `assets` folder. The content and the parsing of the XML scene file are identical to those from the previous project and thus will not be repeated.

1. Make a copy of the `my_game.js` file from the previous project and named it `blue_level.js`.

2. Rename the `scene.xml` file in the `assets` folder to be called `blue_level.xml`.

3. In `blue_level.xml`, change the background color to blue

```
  <!-- Viewport: x, y, w, h -->
  <Camera CenterX="20" CenterY="60" Width="20"
    Viewport="20 40 600 300"
    BgColor="0 0 1 1.0"
/>
```

*The MyGame Scene*

As mentioned, this scene defines in the `init()` function with identical content found in the scene file from the previous project. In the following section, take note of the definition and calls to `next()` and `stop()` functions. Since `my_game.js` is not going to be loading scene information, we can remove the `load()` and `unload()` methods.

1. Edit `my_game.js` to import from `index.js` and a newly defined file called `blue_level.js` (which we will be creating this file in an upcoming step). Note that with the `Scene` class support, you no longer need to import from the `loop` module.

```
import engine from "../engine/index.js";
import BlueLevel from "./blue_level.js";
```

2. Define `MyGame` to be a subclass of the engine `Scene` class, and remember to export `MyGame`:

```
class MyGame extends engine.Scene {
  ... implementation to follow ...
}
export default MyGame;
```

**Note** The JavaScript `extends` keyword defines the parent/child relationship.

3. Define the `constructor()`, `init()`, and `draw()` functions. Note that the scene content defined in the `init()` function, with the exception of the camera background color, is identical to that of the previous project.

```
constructor() {
  super();
  // The camera to view the scene
  this.mCamera = null;
  // the hero and the support objects
  this.mHero = null;
  this.mSupport = null;
}
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]);
  // Step B: Create the support object in red
  this.mSupport = new engine.Renderable();
  this.mSupport.setColor([0.8, 0.2, 0.2, 1]);
  this.mSupport.getXform().setPosition(20, 60);
  this.mSupport.getXform().setSize(5, 5);
  // Step C: Create the hero object in blue
  this.mHero = new engine.Renderable();
  this.mHero.setColor([0, 0, 1, 1]);
  this.mHero.getXform().setPosition(20, 60);
  this.mHero.getXform().setSize(2, 3);
}
draw() {
  // Step A: clear the canvas
  engine.clearCanvas([0.9, 0.9, 0.9, 1.0]);
  // Step  B: Activate the drawing Camera
  this.mCamera.setViewAndCameraMatrix();
  // Step  C: draw everything
  this.mSupport.draw(this.mCamera);
  this.mHero.draw(this.mCamera);
}
```

4. Define the `update()` function; take note of the `this.next()` call when the `mHero` object crosses the `x=11` boundary from the right and the `this.stop()` call when the Q key is pressed.

```
update() {
  // let's only allow the movement of hero,
  // and if hero moves too far off, this level ends, we will
  // load the next level
  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) { // right-bound of the window
          xform.setPosition(12, 60);
      }
  }
  if (engine.input.isKeyPressed(engine.input.keys.Left)) {
      xform.incXPosBy(-deltaX);
      if (xform.getXPos() < 11) {  // left-bound of the window
          this.next();
      }
  }
  if (engine.input.isKeyPressed(engine.input.keys.Q))
      this.stop();  // Quit the game
}
```

5. Define the `next()` function to transition to the `BlueLevel` scene:

```
next() {
  super.next();  // this must be called!
  // next scene to run
  let nextLevel = new BlueLevel();  // next level to be loaded
  nextLevel.start();
}
```

**Note** The `super.next()` call, where the super class can stop the game loop and cause the unloading of this scene, is necessary and absolutely critical in causing the scene transition.

6. Lastly, modify the `window.onload()` function to replace access to the `loop` module with a client-friendly `myGame.start()` function:

```
window.onload = function () {
  engine.init("GLCanvas");
  let myGame = new MyGame();
  myGame.start();
}
```

*The BlueLevel Scene*

The `BlueLevel` scene is almost identical to the MyGame object from the previous project with the exception of supporting the new Scene class and scene transition:

1. Edit `blue_level.js` file in the `my_game` folder to import from the engine `index.js`, `MyGame`, and `SceneFileParser`. Define and export `BlueLevel` to be a subclass of the `engine.Scene` class.

```
// Engine Core stuff
import engine from "../engine/index.js";
// Local stuff
import MyGame from "./my_game.js";
import SceneFileParser from "./util/scene_file_parser.js";

class BlueLevel extends engine.Scene {
  ... implementation to follow ...
}

export default BlueLevel;
```

2. Define the constructor to define the scene file path, create an array to store the `Renderable` objects and a variable to store the camera.

```
constructor() {
  super();

  // scene file name
  this.mSceneFile = "assets/blue_level.xml";
  // all squares
  this.mSqSet = [];        // these are the Renderable objects

  // The camera to view the scene
  this.mCamera = null;
}
```

3. Define the `init()`, `draw()`, `load()`, and `unload()` functions to be **identical** to those in the `MyGame` class from the previous project.

**TASK 4.5A** This level is called **Blue_Level**. Modify the `draw()` function so that the background is blue instead of grey.

4. Define the `update()` function similar to that of the `MyGame` scene. Once again, note the `this.next()` call when the object crosses the `x=11` boundary from the right and the `this.stop()` call when the Q key is pressed.

```
update() {
  // For this very simple game, let's move the first square
  let xform = this.mSqSet[1].getXform();
  let deltaX = 0.05;
  /// Move right and swap over
  if (engine.input.isKeyPressed(engine.input.keys.Right)) {
      xform.incXPosBy(deltaX);
      if (xform.getXPos() > 30) { // right-bound of the window
          xform.setPosition(12, 60);
      }
  }
  // test for white square movement
  if (engine.input.isKeyPressed(engine.input.keys.Left)) {
      xform.incXPosBy(-deltaX);
      if (xform.getXPos() < 11) { // this is the left-boundary
          this.next(); // go back to my game
      }
  }
  if (engine.input.isKeyPressed(engine.input.keys.Q))
      this.stop();  // Quit the game
}
```

5. Lastly, define the `next()` function to transition to the `MyGame` scene. It is worth reiterating that the call to `super.next()` is necessary because it is critical to stop the game loop and unload the current scene before proceeding to the next scene.

```
next() {
  super.next();
  let nextLevel = new MyGame();  // load the next level
  nextLevel.start();
}
```

You can now run the project and view the scenes unloading and loading and quit the game at any point during the interaction. Your game engine now has a well-defined interface for working with its client. This interface follows the well-defined protocol of the `Scene` class.

- `constructor()`: For declaring variables and defining constants.
- `start()`/`stop()`: For starting a scene and stopping the game. These two methods are not meant to be overwritten by a subclass.

The following interface methods are meant to be overwritten by subclasses.

- `init()`: For instantiating the variables and setting up the game scene.
- `load()`/`unload()`: For initiating the asynchronous loading and unloading of external resources.
- `draw()`/`update()`: For continuously displaying the game state and receiving player input and implementing the game logic.
- `next()`: For instantiating and transitioning to the next scene. Lastly, as a final reminder, it is absolutely critical for the subclass to call the `super.next()` to stop the game loop and unload the scene.

Any objects that define these methods can be loaded and interacted with by your game engine. You can experiment with creating other levels.

**Figure 4-6. Two Scenes Completed Project**

## Audio

Audio is an essential element of all video games. In general, audio effects in games fall into two categories. The first category is background audio. This includes background music or ambient effects and is often used to bring atmosphere or emotion to different portions of the game. The second category is sound effects. Sound effects are useful for all sorts of purposes, from notifying users of game actions to hearing the footfalls of your hero character. Usually, sound effects represent a specific action, triggered either by the user or by the game itself. Such sound effects are often thought of as an audio cue.

One important difference between these two types of audio effects is how you control them. Sound effects or cues cannot be stopped or have their volume adjusted once they have started; therefore, cues are generally short. On the other hand, background audio can be started and stopped at will. These capabilities are useful for stopping the background track completely and starting another one.

### Lab 4.6
### The Audio Support Project

This project has identical `MyGame` and the `BlueLevel` scenes to the previous project. You can move the front rectangle left or right with the arrow keys, the intersection with the left boundary triggers the loading of the other scene, and the Q key quits the game. However, in this version, each scene plays background music and triggers a brief audio cue when the left-/right-arrow key is pressed. Notice that the volume varies for each type of audio clip. The implementation of this project also reinforces the concept of loading and unloading of external resources and the audio clips themselves.

The controls of the project are as follows:

**Left-/right-arrow key**: Moves the front rectangle left and right to increase and decrease the volume of the background music
**Q key**: Quits the game

The goals of the project are as follows:

To add audio support to the resource management system
To provide an interface to play audio for games

While both `mp3` and `wav` files are supported, audio files of these formats should be used with care. Files in `.mp3` format are compressed and are suitable for storing longer durations of audio content, for example, for background music. Files in `.wav` format are uncompressed and should contain only very short audio snippet, for example, for storing cue effects.

*Define an Audio Resource Module*

While audio and text files are completely different, from the perspective of your game engine implementation, there are two important similarities. First, both are external resources and thus will be implemented similarly as engine components in the `src/engine/resources` folder. Second, both involve standardized file formats with well-defined API utilities. The Web Audio API will be used for the actual retrieving and playing of sound files. Even though this API offers vast capabilities, in the interests of focusing on the rest of the game engine development, only basic supports for background audio and effect cues are discussed.

**Note** Interested readers can learn more about the Web Audio API from www.w3.org/TR/webaudio/.
The latest policy for some browsers, including Chrome, is that audio will not be allowed to play until first interaction from the user. This means that the context creation will result in an initial warning from Chrome that is output to the runtime browser console. The audio will only be played after user input (e.g., mouse click or keyboard events).

1. In the `src/engine/resources` folder, create a new file and name it `audio.js`. This file will implement the module for the audio component. This component must support two types of functionalities: loading and unloading of audio files and playing and controlling of the content of audio file for the game developer.

2. The loading and unloading are similar to the implementations of `text` and `xml` modules where the core resource management functionality is imported from `resource_map`:

```
"use strict";
import * as map from "../core/resource_map.js";
// functions from resource_map
let unload = map.unload;
let has = map.has;
```

3. Define the decoding and parsing functions, and call the `resource_map loadDecodeParse()` function to load an audio file. Notice that with the support from `resource_map` and the rest of the engine infrastructure, loading and unloading of external resources have become straightforward.

```
function decodeResource(data) { return data.arrayBuffer(); }
function parseResource(data) {
  return mAudioContext.decodeAudioData(data); }
function load(path) {
  return map.loadDecodeParse(path, decodeResource, parseResource);
}
```

4. With the loading functionality completed, you can now define the audio control and manipulation functions. Declare variables to maintain references to the Web Audio context and background music and to control volumes.

```
let mAudioContext = null;
let mBackgroundAudio = null;
// volume control support
let mBackgroundGain = null; // background volume
let mCueGain = null;        // cue/special effects volume
let mMasterGain = null;     // overall/master volume
let kDefaultInitGain = 0.1;
```

5. Define the `init()` function to create and store a reference to the Web Audio context in `mAudioContext`, and initialize the audio volume gain controls for the background, cue, and a master that affects both. In all cases, volume gain of a 0 corresponds to no audio and 1 means maximum loudness.

```
function init() {
  try {
      let AudioContext = window.AudioContext ||
                         window.webkitAudioContext;
      mAudioContext = new AudioContext();
      // connect Master volume control
      mMasterGain = mAudioContext.createGain();
      mMasterGain.connect(mAudioContext.destination);
      // set default Master volume
      mMasterGain.gain.value = kDefaultInitGain;
      // connect Background volume control
      mBackgroundGain = mAudioContext.createGain();
      mBackgroundGain.connect(mMasterGain);
      // set default Background volume
      mBackgroundGain.gain.value = 1.0;
      // connect Cuevolume control
      mCueGain = mAudioContext.createGain();
      mCueGain.connect(mMasterGain);
      // set default Cue volume
      mCueGain.gain.value = 1.0;
  } catch (e) {
      throw new Error("...");
  }
}
```

6. Define the `playCue()` function to play the entire duration of an audio clip with proper volume control. This function uses the audio file path as a resource name to find the loaded asset from the `resource_map` and then invokes the Web Audio API to play the audio clip. Notice that no reference to the `source` variable is kept, and thus once started, there is no way to stop the corresponding audio clip. A game should call this function to play short snippets of audio clips as cues.

```
function playCue(path, volume) {
  let source = mAudioContext.createBufferSource();
  source.buffer = map.get(path);
  source.start(0);
  // volume support for cue
  source.connect(mCueGain);
  mCueGain.gain.value = volume;
}
```

7. Define the functionality to play, stop, query, and control the volume of the background music. In this case, the `mBackgroundAudio` variable keeps a reference to the currently playing audio, and thus, it is possible to stop the clip or change its volume.

```
function playBackground(path, volume) {
  if (has(path)) {
      stopBackground();
      mBackgroundAudio = mAudioContext.createBufferSource();
      mBackgroundAudio.buffer = map.get(path);
      mBackgroundAudio.loop = true;
      mBackgroundAudio.start(0);
       // connect volume accordingly
       mBackgroundAudio.connect(mBackgroundGain);
       setBackgroundVolume(volume);
  }
}
function stopBackground() {
  if (mBackgroundAudio !== null) {
      mBackgroundAudio.stop(0);
      mBackgroundAudio = null;
  }
}
function isBackgroundPlaying() {
  return (mBackgroundAudio !== null);
}
function setBackgroundVolume(volume) {
  if (mBackgroundGain !== null) {
      mBackgroundGain.gain.value = volume;
  }
}
function  incBackgroundVolume(increment) {
  if (mBackgroundGain !== null) {
      mBackgroundGain.gain.value += increment;
      // need this since volume increases when negative
      if (mBackgroundGain.gain.value < 0) {
          setBackgroundVolume(0);
      }
  }
}
```

8. Define functions for controlling the master volume, which adjusts the volume of both the cue and the background music:

```
function  setMasterVolume(volume) {
  if (mMasterGain !== null) {
      mMasterGain.gain.value = volume;
  }
}
function  incMasterVolume(increment) {
  if (mMasterGain !== null) {
      mMasterGain.gain.value += increment;
      // need this since volume increases when negative
      if (mMasterGain.gain.value < 0) {
          mMasterGain.gain.value = 0;
      }
  }
}
```

9. Define a `cleanUp()` function to release the allocated HTML5 resources:

```
function cleanUp() {
   mAudioContext.close();
   mAudioContext = null;
}
```

10. Remember to `export` the functions from this module:

```
export {init, cleanUp,
    has, load, unload,
    playCue,
    playBackground, stopBackground, isBackgroundPlaying,
    setBackgroundVolume, incBackgroundVolume,
    setMasterVolume, incMasterVolume
}
```

*Export the Audio Module to the Client*

Edit the `index.js` file to import from `audio.js`, initialize and cleanup the module accordingly, and to export to the client:

```
... identical to previous code ...
import * as audio from "./resources/audio.js";
... identical to previous code ...
function init(htmlCanvasID) {
  glSys.init(htmlCanvasID);
  vertexBuffer.init();
```

```
    shaderResources.init();
    input.init();
    audio.init();
  }
  function cleanUp() {
    loop.cleanUp();
    audio.cleanUp();
    input.cleanUp();
    shaderResources.cleanUp();
    vertexBuffer.cleanUp();
    glSys.cleanUp();
  }
  ... identical to previous code ...
  export default {
    // resource support
    audio, text, xml
    ... identical to previous code ...
  }
```

*Testing the Audio Component*

To test the audio component, you must copy the necessary audio files into your game project.

1. Create a new folder in the assets folder and name it `sounds`.

2. Add the `bg_clip.mp3`, `blue_level_cue.wav`, and `my_game_cue.wav` files into the `sounds` folder (your instructor can provide you these files).

You will now need to update the `MyGame` and `BlueLevel` implementations to load and use these audio resources.

*Change MyGame.js*

Update `MyGame` scene to load the audio clips, play background audio, and cue the player when the arrow keys are pressed:

1. Declare constant file paths to the audio files in the constructor. Recall that these file paths are used as resource names for loading, storage, and retrieval. Declaring these as constants for later reference is a good software engineering practice.

```
  constructor() {
    super();
    // audio clips: supports both mp3 and wav formats
    this.mBackgroundAudio = "assets/sounds/bg_clip.mp3";
    this.mCue = "assets/sounds/my_game_cue.wav";
    ... identical to previous code ...
  }
```

2. Request the loading of audio clips in the `load()` function, and make sure to define the corresponding `unload()` function. Notice that the unloading of background music is preceded by stopping the music. In general, a resource's operations must be halted prior to its unloading.

```
load() {
  // loads the audios
  engine.audio.load(this.mBackgroundAudio);
  engine.audio.load(this.mCue);
}
unload() {
  // Step A: Game loop not running, unload all assets
  // stop the background audio
  engine.audio.stopBackground();
  // unload the scene resources
  engine.audio.unload(this.mBackgroundAudio);
  engine.audio.unload(this.mCue);
}
```

3. Start the background audio at the end of the `init()` function.

```
init() {
  ... identical to previous code ...
  // now start the Background music ...
  engine.audio.playBackground(this.mBackgroundAudio, 1.0);
}
```

4. In the `update()` function, cue the players when the right- and left-arrow keys are pressed, and increase and decrease the volume of the background music:

```
update() {
  ... identical to previous code ...
  // Support hero movements
  if (engine.input.isKeyPressed(engine.input.keys.Right)) {
      engine.audio.playCue(this.mCue, 0.5);
      engine.audio.incBackgroundVolume(0.05);
      xform.incXPosBy(deltaX);
      if (xform.getXPos() > 30) { // right-bound of the window
          xform.setPosition(12, 60);
      }
  }
  if (engine.input.isKeyPressed(engine.input.keys.Left)) {
      engine.audio.playCue(this.mCue, 1.5);
      engine.audio.incBackgroundVolume(-0.05);
      xform.incXPosBy(-deltaX);
      if (xform.getXPos() < 11) {  // left-bound of the window
          this.next();
      }
```

```
    }
    ... identical to previous code ...
}
```

*Change BlueLevel.js*

The changes to the `BlueLevel` scene are similar to those of the `MyGame` scene but with a different audio cue:

1. In the `BlueLevel` constructor, add the following path names to the audio resources:

```
constructor() {
    super();
    // audio clips: supports both mp3 and wav formats
    this.mBackgroundAudio = "assets/sounds/bg_clip.mp3";
    this.mCue = "assets/sounds/blue_level_cue.wav";
    ... identical to previous code ...
}
```

2. Modify the `load()` and `unload()` functions for the audio clips:

```
load() {
    engine.xml.load(this.mSceneFile);
    engine.audio.load(this.mBackgroundAudio);
    engine.audio.load(this.mCue);
}
unload() {
    // stop the background audio
    engine.audio.stopBackground();
    // unload the scene file and loaded resources
    engine.xml.unload(this.mSceneFile);
    engine.audio.unload(this.mBackgroundAudio);
    engine.audio.unload(this.mCue);
}
```

3. In the same manner as `MyGame`, start the background audio in the `init()` function and cue the player when the left and right keys are pressed in the `update()` function. Notice that in this case, the audio cues are played with different volume settings.

```
init() {
    ... identical to previous code ...
    // now start the Background music ...
    engine.audio.playBackground(this.mBackgroundAudio, 0.5);
}

update() {
    ... identical to previous code ...
```

```
    // Move right and swap over
    if (engine.input.isKeyPressed(engine.input.keys.Right)) {
        engine.audio.playCue(this.mCue, 0.5);
        xform.incXPosBy(deltaX);
        if (xform.getXPos() > 30) { // right-bound of the window
            xform.setPosition(12, 60);
        }
    }
    // Step A: test for white square movement
    if (engine.input.isKeyPressed(engine.input.keys.Left)) {
        engine.audio.playCue(this.mCue, 1.0);
        xform.incXPosBy(-deltaX);
        if (xform.getXPos() < 11) { // this is the left-boundary
            this.next(); // go back to my game
        }
    }
    ... identical to previous code ...
}
```

You can now run the project and listen to the wonderful audio feedback. If you press and hold the arrow keys, there will be many cues repeatedly played. In fact, there are so many cues echoed that the sound effects are blurred into an annoying blast. This serves as an excellent example illustrating the importance of using audio cues with care and ensuring each individual cue is nice and short. You can try tapping the arrow keys to listen to more distinct and pleasant-sounding cues, or you can simply replace the `isKeyPressed()` function with the `isKeyClicked()` function and listen to each individual cue.

**Summary**

In this chapter, you learned how several common components of a game engine come together. Starting with the ever-important game loop, you learned how it implements an input, update, and draw pattern in order to surpass human perception or trick our senses into believing that the system is continuous and running in real time. This pattern is at the heart of any game engine. You learned how full keyboard support can be implemented with flexibility and reusability to provide the engine with a reliable input component. Furthermore, you saw how a resource manager can be implemented to load files asynchronously and how scenes can be abstracted to support scenes being loaded from a file, which can drastically reduce duplication in the code. Lastly, you learned how audio support supplies the client with an interface to load and play both ambient background audio and audio cues.

These components separately have little in common but together make up the core fundamentals of nearly every game. As you implement these core components into the game engine, the games that are created with the engine will not need to worry about the specifics of each component. Instead, the games programmer can focus on utilizing the functionality to hasten and streamline the development process. In the next chapter, you will learn how to create the illusion of an animation with external images.

**Game Design Considerations**

In this chapter, we discussed the *game loop* and the technical foundation contributing to the connection between what the player does and how the game responds. If a player selects a square that's drawn on the screen and moves it from location A to location B by using the arrow keys, for example, you'd typically want that action to appear as a smooth motion beginning as soon as the arrow key is pressed, without stutters, delays, or noticeable lag. The game loop contributes significantly to what's known as *presence* in game design; presence is the player's ability to feel as if they're connected to the game world, and responsiveness plays a key role in making players feel connected. Presence is reinforced when actions in the real world (such as pressing arrow keys) seamlessly translate to responses in the game world (such as moving objects, flipping switches, jumping, and so on); presence is compromised when actions in the real world suffer translation errors such as delays and lag.

As mentioned in Chapter 1, effective game mechanic design can begin with just a few simple elements. By the time you've completed the *Keyboard Support* project in this chapter, for example, many of the pieces will already be in place to begin constructing game levels: you've provided players with the ability to manipulate two individual elements on the screen (the red and white squares), and all that remains in order to create a basic game loop is to design a causal chain using those elements that results in a new event when completed. Imagine the *Keyboard Support* project is your game: how might you use what's available to create a causal chain? You might choose to play with the relationship between the squares, perhaps requiring that the red square be moved completely within the white square in order to unlock the next challenge; once the player successfully placed the red square in the white square, the level would complete. This basic mechanic may not be quite enough on its own to create an engaging experience, but by including just a few of the other eight elements of game design (systems design, setting, visual design, music and audio, and the like), it's possible to turn this one basic interaction into an almost infinite number of engaging experiences and create that sense of presence for players. You'll add more game design elements to these exercises as you continue through subsequent chapters.

The *Resource Map* and *Shader Loader* project, the *Scene File* project, and the *Scene Objects* project are designed to help you begin thinking about architecting game designs from the ground up for maximum efficiency so that problems such as asset loading delays that detract from the player's sense of presence are minimized. As you begin designing games with multiple stages and levels and many assets, a resource management plan becomes essential. Understanding the limits of available memory and how to smartly load and unload assets can mean the difference between a great experience and a frustrating experience.

We experience the world through our senses, and our feeling of presence in games tends to be magnified as we include additional sensory inputs. The *Audio Support* project adds basic audio to our simple state-changing exercise from the *Scene Objects* project in the form of a constant background score to provide ambient mood and includes a distinct movement sound for each of the two areas. Compare the two experiences and consider how different they feel because of the presence of sound cues; although the visual and interaction experience is identical between

the two, the *Audio Support* project begins to add some emotional cues because of the beat of the background score and the individual tones the rectangle makes as it moves. Audio is a powerful enhancement to interactive experiences and can dramatically increase a player's sense of presence in game environments, and as you continue through the chapters, you'll explore how audio contributes to game design in more detail.