CHAPTER 4

Implementing Common Components of Video Games

After completing this chapter, you will be able to:

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

Introduction

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

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

The Game Loop

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

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

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

Typical Game Loop Implementations

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

```
initialize();
while(game running) {
    input();
    update();
    draw();
}
```

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

```
elapsedTime = now;
previousLoop = now;
while(game running) {
    elapsedTime += now - previousLoop;
    previousLoop = now;
```

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

In the previous pseudocode listing, `UPDATE_TIME_RATE` is the required real-time update rate. When the elapsed time between the game loop cycle is greater than the `UPDATE_TIME_RATE`, `update()` will be called until it is caught up. This means that the `draw()` operation is essentially skipped when the game loop is running too slowly. When this happens, the entire game will appear to run slowly, with lagging 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.

The Game Loop Project

This project demonstrates how to incorporate a game loop into your game engine and to support real-time animation by updating and drawing the squares accordingly. You can see an example of this project running in Figure 4-1. The source code to this project is defined in the `Chapter4/4.1.GameLoop` folder.

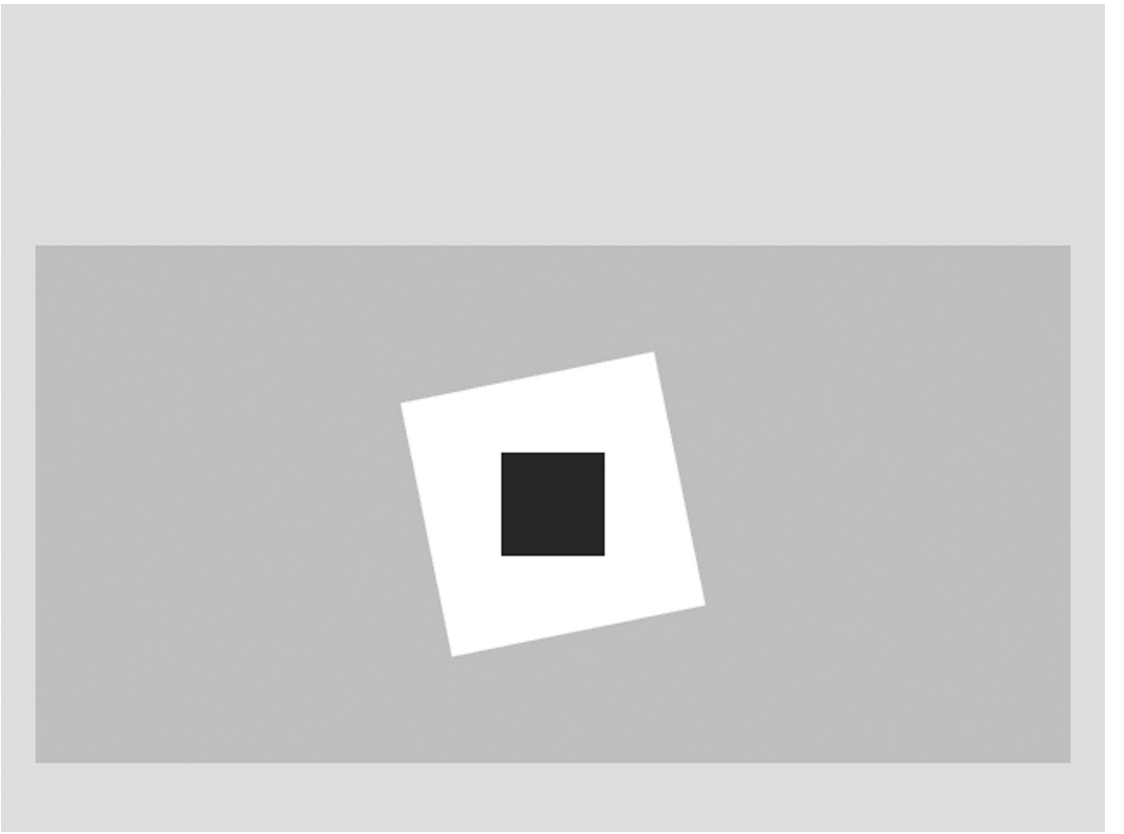

Figure 4-1. *Running the Game Loop project*

The goals of the project are as follows:

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

Implement the Game Loop Component

The game loop component is a core game engine functionality and thus should be implemented similar to `Engine_Core` or `Engine_VertexBuffer`, as a property of the `gEngine` object in a file defined in the `src/Engine/Core` folder. The actual implementation is similar to the pseudocode listing discussed; for clarity it's shown without the input support for now.

1. Create a new file in the `src/Engine/Core` folder and name the file `Engine_GameLoop.js`. Define the `GameLoop` component following the JavaScript module pattern as follows:

```
var gEngine = gEngine || { };

gEngine.GameLoop = (function() {
    var mPublic = { };
    return mPublic;
} ());
```

■ **Tip** This book refers to core game engine functionality as engine components and implements each component as an encapsulated property to the `gEngine` object based on this same pattern.

---

2. Add the following instance variables to keep track of frame rate, processing time, and referencing to the currently running game:

```
var kFPS = 60;           // Frames per second
var kMPF = 1000 / kFPS;  // Milleseconds per frame.

// Variables for timing gameloop.
var mPreviousTime;
var mLagTime;
var mCurrentTime;
var mElapsedTime;

// The current loop state (running or should stop)
var mIsLoopRunning = false;

// Reference to game logic
var mMyGame = null;
```

Notice that `kFPS` is the frame per second discussed and that `kMPF` is milliseconds per frame. It is important to maintain the game at an update interval of `kFPS`.

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

```
// This function assumes it is sub-classed from MyGame
var _runLoop = function () {
    if(mIsLoopRunning) {
        // Step A: set up for next call to _runLoop and update input!
        requestAnimationFrame( function(){_runLoop.call(mMyGame);} );
```

```
            // Step B: compute elapsed time since last RunLoop was executed
            mCurrentTime = Date.now();
            mElapsedTime = mCurrentTime - mPreviousTime;
            mPreviousTime = mCurrentTime;
            mLagTime += mElapsedTime;

            // Step C: update the game the appropriate number of times.
            //      Update only every Milliseconds per frame.
            //      If lag larger then update frames, update until caught up.
            while ((mLagTime >= kMPF) && mIsLoopRunning) {
                this.update();      // call MyGame.update()
                mLagTime -= kMPF;
            }

            // Step D: now let's draw
            this.draw();    // Call MyGame.draw()
        }
    };
```

Notice the similarity between the pseudocode examined and the steps B, C, and D of the `_runLoop()` function shown previously. The main difference is that the functionality of the outermost `while` loop is implemented with the `requestAnimationFrame()` function call at step A, where the `_runLoop()` function is set up to be called continuously. More specifically:

a. The `requestAnimationFrame()` function registers the `_runLoop()` function with the browser where the browser will call on the next available frame. Notice that each call to the `requestAnimationFrame()` function will result in one execution of the corresponding `_runLoop()` function.

b. The syntax of the function passed to `requestAnimationFrame()`, `_runLoop.call(mMyGame)`, binds the `'this'` context to `mMyGame`. In this way, the `this.update()` and `this.draw()` function calls within the loop are those defined in the `mMyGame` object.

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

---

4. Add the function to start the game loop as follows:

```
var start = function(myGame) {
    mMyGame = myGame;

    // Step A: reset frame time
    mPreviousTime = Date.now();
    mLagTime = 0.0;
```

```
        // Step B: remember that loop is now running
        mIsLoopRunning = true;

        // Step C: request _runLoop to start when loading is done
        requestAnimationFrame(function(){_runLoop.call(mMyGame);});
    };
```

This function initializes the game loop internal state by setting `mMyGame`, initializing the frame time, and setting the loop running flag to true, and then it starts the game loop by passing the `_runLoop()` function to `requestAnimationFrame()`.

5. Remember to add the `start()` function to the public interface of the `GameLoop` object.

```
var mPublic = {
    start: start
};
return mPublic;
```

Using the Game Loop

To test the game loop implementation, a game class must define the `update()` and `draw()` functions. In this case, the `MyGame` object will also define a constructor and an `initialize()` function.

1. In `MyGame.js`, replace the `MyGame` constructor with the following:

```
function MyGame(htmlCanvasID) {
    // variables of the constant color shader
    this.mConstColorShader = null;

    // variables for the squares
    this.mWhiteSq = null;        // these are the renderable objects
    this.mRedSq = null;

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

    // Initialize the webGL Context
    gEngine.Core.initializeWebGL(htmlCanvasID);

    // Initialize the game
    this.initialize();
}
```

2. Add an initialization function to the prototype as follows:

```
MyGame.prototype.initialize = function() {
    // Step A: set up the cameras
    this.mCamera = new 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 dark gray

        // Step  B: create the shader
        this.mConstColorShader = new SimpleShader(
                "src/GLSLShaders/SimpleVS.glsl",  // Path to the VertexShader
                "src/GLSLShaders/SimpleFS.glsl"); // Path to the FragmentShader

        // Step  C: Create the renderable objects:
        this.mWhiteSq = new Renderable(this.mConstColorShader);
        this.mWhiteSq.setColor([1, 1, 1, 1]);
        this.mRedSq = new Renderable(this.mConstColorShader);
        this.mRedSq.setColor([1, 0, 0, 1]);

        // Step  D: Initialize the white renderable object: centred, 5x5, rotated
        this.mWhiteSq.getXform().setPosition(20, 60);
        this.mWhiteSq.getXform().setRotationInRad(0.2); // In Radian
        this.mWhiteSq.getXform().setSize(5, 5);

        // Step  E: Initialize the red renderable object: centered 2x2
        this.mRedSq.getXform().setPosition(20, 60);
        this.mRedSq.getXform().setSize(2, 2);

        // Step F: Start the game loop running
        gEngine.GameLoop.start(this);
    };
```

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

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

```
// The update function, updates the application state. Make sure to _NOT_ draw
// anything from this function!
MyGame.prototype.update = function() {
    // For this very simple game, let's move the white square and pulse the red

    // Step A: move the white square
    var whiteXform = this.mWhiteSq.getXform();
    var deltaX = 0.05;
    if (whiteXform.getXPos() > 30)  // this is the right-bound of the window
        whiteXform.setPosition(10, 60);
    whiteXform.incXPosBy(deltaX);
    whiteXform.incRotationByDegree(1);

    // Step B: pulse the red square
    var redXform = this.mRedSq.getXform();
```

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

Recall that the `update()` function is called at about 60 FPS, and each time the following happens:

- *Step A for the white square*: Increase the rotation by 1 degree; increase the x-position by 0.05 and reset to 10 if the resulting x-position is greater than 30.
- *Step B for the red square*: Increase the size by 0.05 and reset it to 2 if the resulting size is greater than 5.

Since the previous operations are performed continuously at about 60 times a second, you can expect to see the following:

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

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

```
    // This is the draw function, make sure to setup proper drawing environment,
    // and more importantly, make sure to _NOT_ change any state.
    MyGame.prototype.draw = function() {
        // Step A: clear the canvas
        gEngine.Core.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray

        // Step  B: Activate the drawing Camera
        this.mCamera.setupViewProjection();

        // Step  C: Activate the white shader to draw
        this.mWhiteSq.draw(this.mCamera.getVPMatrix());

        // Step  D: Activate the red shader to draw
        this.mRedSq.draw(this.mCamera.getVPMatrix());
    };
```

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

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

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

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

Keyboard Input

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

The Keyboard Support Project

This project examines keyboard input support and incorporates the functionality into the game engine. The position, rotation, and size of the game objects in this project are under your input control. You can see an example of this project running in Figure 4-2. The source code to this project is defined in the `Chapter4/4.2.KeyboardSupport` folder.

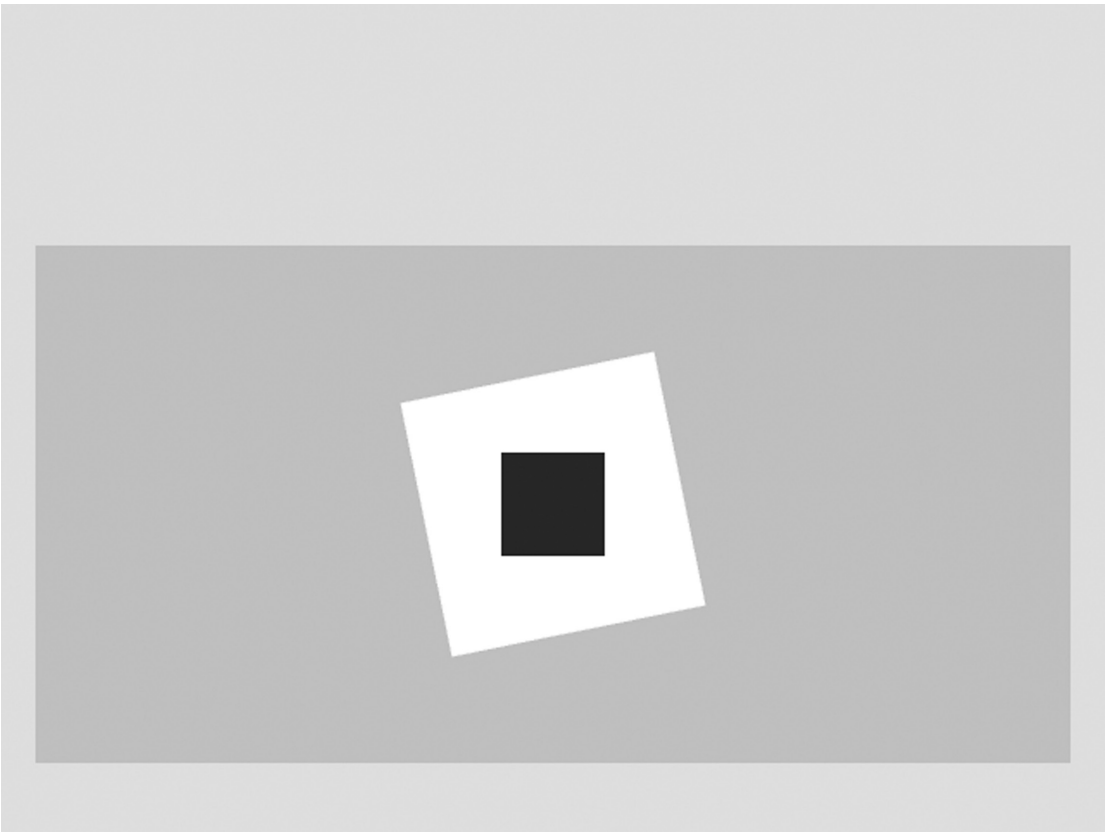

Figure 4-2. *Running the Keyboard Support project*

The controls of the project are as follows:

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

The goals of the project are as follows:

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

Add an Input Component to the Engine

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

1. Create a new file in the `src/Engine/Core/` folder and name it `Engine_Input.js`. Similar to other engine components (for example, `gEngine_GameLoop` or `gEngine_VertexBuffer`), add the following:

```
var gEngine = gEngine || { };

gEngine.Input = (function() {
    var mPublic = { };
    return mPublic;
}());
```

2. Define a set of keyboard keys to map key codes.

```
// Key code constants
var kKeys = {
    // 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,
    R : 82,
    S : 83,
    W : 87,

    LastKeyCode : 222
};
```

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

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

---

3. Create objects to track key states.

```
// 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 = [];
```

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

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

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

```
// Event service functions
var _onKeyDown = function (event) {
    mIsKeyPressed[event.keyCode] = true;  };
var _onKeyUp = function (event)  {
    mIsKeyPressed[event.keyCode] = false; };
```

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

5. Add a function to initialize all the key states and register the key event handlers to the browser.

```
var initialize = function () {
    var I;
    for (i = 0; i<kLastKeyCode; i++) {
        mIsKeyPressed[i] = false;
        mKeyPreviousState[i] = false;
        mIsKeyClicked[i] = false;
    }

    // register handlers
    window.addEventListener('keyup', _onKeyUp);
    window.addEventListener('keydown', _onKeyDown);
};
```

Notice that the `window.addEventListener()` function registers the `_onKeyUp/Down()` event handler functions with the browser such that these functions will be called to register the keyboard state changes.

6. Add an `update()` function to derive key click events.

```
var update = function() {
    var I;
    for (i = 0; i<kLastKeyCode; i++) {
        mIsKeyClicked[i] = (!mKeyPreviousState[i]) && mIsKeyPressed[i];
        mKeyPreviousState[i] = mIsKeyPressed[i];
    }
};
```

The `update()` function uses `mIsKeyPressed` and `mKeyPreviousState` to determine whether a key clicked event has occured.

7. Add public functions for clean inquires to current keyboard states.

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

8. Finally, add the public functions and the key constants to the interface.

```
var mPublic =
{
    initialize: initialize,
    update: update,
    isKeyPressed: isKeyPressed,
    isKeyClicked: isKeyClicked,
    keys: kKeys;
};
```

Modify the Engine to Support Keyboard Input

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

1. For clean and clear initialization of the game engine core, WebGL-specific initializations are collected into a single function. In `Engine_Core.js`, modify the `_initializeWebGL()` function to the following:

```
var _initializeWebGL = function(htmlCanvasID) {
    var canvas = document.getElementById(htmlCanvasID);

    mGL = canvas.getContext("webgl") ||
          canvas.getContext("experimental-webgl");

    if (mGL === null) {
        document.write("<br><b>WebGL is not supported!</b>");
    }
};
```

This function only initializes WebGL.

2. The engine core initialization can now be cleanly defined as the initialization of each essential component, as follows:

```
// initialize all of the EngineCore components
var initializeEngineCore = function(htmlCanvasID) {
    _initializeWebGL(htmlCanvasID);
    gEngine.VertexBuffer.initialize();
    gEngine.Input.initialize();
};
```

3. Remember to include the correct functions in the public interface.

```
var mPublic = {
    getGL: getGL,
    initializeEngineCore: initializeEngineCore,
    clearCanvas: clearCanvas
};
```

4. Finally, it is important to update the input states from within the game loop. In `Engine_GameLoop.js`, within the `_runLoop()` function, add a call to update the input before each update to the game in step C, as follows:

```
var _runLoop = function () {
    if(mIsLoopRunning) {
        // ... rest of the code is identical ...
        // Step C: Make sure we update the game the appropriate number of times.
```

```
        while ((mLagTime >= kMPF) && (mIsLoopRunning)) {
            gEngine.Input.update();
            this.update();      // call Scene.update()
            mLagTime -= kMPF;
        }
        // ... rest of the code is identical ...
```

Test Keyboard Input

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

1. First you must modify MyGame to work with the new engine core initialization function. In MyGame.js, within the MyGame constructor, call the new initialization function as follows:

```
// Initialize the webGL Context
function MyGame(htmlCanvasID) {
    // ... identical code as previous project ...
    gEngine.Core.initializeEngineCore(htmlCanvasID);
    // ... identical code as previous project ...
}
```

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

```
MyGame.prototype.update = function() {
    // For this very simple game, let's move the white square and pulse the red
    var whiteXform = this.mWhiteSq.getXform();
    var deltaX = 0.05;

    // Step A: test for white square movement
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Right)) {
        if (whiteXform.getXPos() > 30)  // the right-bound of the window
            whiteXform.setPosition(10, 60);
        whiteXform.incXPosBy(deltaX);
    }

    // Step  B: test for white square rotation
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Up))
        whiteXform.incRotationByDegree(1);

    var redXform = this.mRedSq.getXform();
    // Step  C: test for pulsing the red square
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Down)) {
        if (redXform.getWidth() > 5)
            redXform.setSize(2, 2);
        redXform.incSizeBy(0.05);
    }
};
```

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

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

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

Resource Management and Asynchronous Loading

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

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

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

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

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

The Resource Map and Shader Loader Project

This project guides you to develop the `ResourceMap`, an infrastructural object for resource management, and demonstrates how to work with this object to load shader files asynchronously. You can see an example of this project running in Figure 4-3. This project appears to be identical to the previous project, with the only difference being how the GLSL shaders are loaded. The source code to this project is defined in the `Chapter4/4.3.ResourceMapAndShaderLoader` folder.

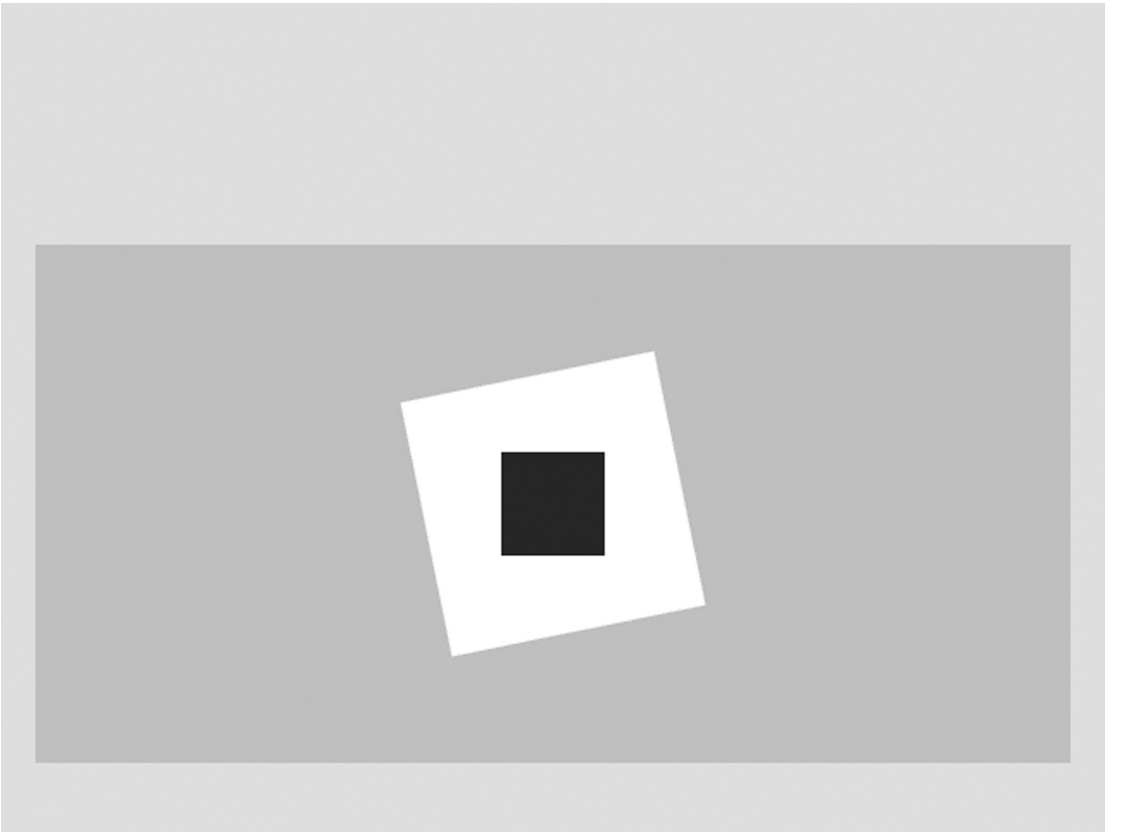

Figure 4-3. *Running the Resource Map and Shader Loader project*

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

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

The goals of the project are as follows:

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

Add ResourceMap Component to the Engine

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

1. Create a new folder in `src/Engine/Core/` and name it `Resources`. This new folder is created in anticipation of the necessary support for many resource types and to maintain a clean source code organization.

2. Create a new file in the `src/Engine/Core/Resources` folder and name it `Engine_ResourceMap.js`. Similar to other engine components, add the following:

```
var gEngine = gEngine || { };

gEngine.ResourceMap = (function(){
    var mPublic = { };
    return mPublic;
}());
```

3. Define a simple object for storing each resource.

```
var MapEntry = function(rName) {
    this.mAsset = rName;
};
```

In the constructor, the `mAsset` is initialized to the name of the resource. As will be shown in the following steps, this will be changed, and `mAsset` will store the reference to the loaded resource.

4. Add the following instance variables:

```
// Resource storage
var mResourceMap = {};

// Number of outstanding load operations
var mNumOutstandingLoads = 0;

// Callback function when all textures are loaded
var mLoadCompleteCallback = null;
```

The `mResourceMap` is the hashmap container of `MapEntry` that will support the actual storage and retrieval according to the unique names of resources. Names of resources are typically their corresponding file paths, which are unique. The `mNumOutsandingLoads` is the number of resource load requests issued and not completed. When `mNumOutsandingLoads` becomes zero, the function referenced by `mLoadCompleteCallback` will be called.

5. Define the support for setting and executing the callback.

```
var _checkForAllLoadCompleted = function() {
    if ((mNumOutstandingLoads === 0) && (mLoadCompleteCallback !== null) ) {
        // ensures the load complete call back will only be called once!
        var funToCall = mLoadCompleteCallback;
        mLoadCompleteCallback = null;
        funToCall();
    }
};
// Make sure to set the callback _AFTER_ all load commands are issued
var setLoadCompleteCallback = function (funct) {
    mLoadCompleteCallback = funct;
    // in case all loading are done
    _checkForAllLoadCompleted();
};
```

When all resources are loaded or when `mNumOutstandingLoads` is zero and if the callback function `mLoadCompleteCallback` is defined, it is called. Note that in `_checkForAllLoadCompleted()`, the `mLoadCompleteCallback` is reset to ensure each callback can be called only once.

6. Define functions to record asynchronous loading requests and completions.

```
var asyncLoadRequested = function(rName) {
    mResourceMap[rName] = new MapEntry(rName);
        // place holder for the resource to be loaded
    ++mNumOutstandingLoads;
};
var asyncLoadCompleted = function(rName, loadedAsset) {
    if (!isAssetLoaded(rName))
        alert("gEngine.asyncLoadCompleted: [" + rName + "] not in map!");
    mResourceMap[rName].mAsset = loadedAsset;
    --mNumOutstandingLoads;
    _checkForAllLoadCompleted();
};
```

The `asyncLoadRequested()` and `asyncLoadCompleted()` functions keep track of the number of outstanding asynchronous loads issued. Notice that `asyncLoadRequested()` allocates a new `MapEntry` and stores it in the `_ResourceMap` according to the resource name. Whereas the `asyncLoadCompleted()` function stored the reference to the loaded asset in `MapEntry.mAsset`, `_checkForAllLoadCompleted()` is always called at the end of the `asyncLoadCompleted()` function to test for the condition where all outstanding asynchronous loadings have been completed, and the callback should be executed.

7. Add functions for testing the load status and retrieving and unloading resources.

```
var isAssetLoaded = function(rName) {
    return (rName in mResourceMap);
};
var retrieveAsset = function(rName) {
    var r = null;
    if (rName in mResourceMap)
        r = mResourceMap[rName].mAsset;
    return r;
};
```

```
var unloadAsset = function(rName) {
    if (rName in mResourceMap) {
        delete mResourceMap[rName];
    }
};
```

8. Lastly, remember to add all public functions to the public interface.

```
// Public interface for this object.
var mPublic = {
    // asynchronous resource loading support
    asyncLoadRequested: asyncLoadRequested,
    asyncLoadCompleted: asyncLoadCompleted,
    setLoadCompleteCallback: setLoadCompleteCallback,

    // resource storage
    retrieveAsset: retrieveAsset,
    unloadAsset: unloadAsset,
    isAssetLoaded: isAssetLoaded
};
return mPublic;
```

Notice that `ResourceMap` is not designed to perform the actual loading of any resources. Instead, it is a manager that allows an asynchronous resource loader to register loading commands issued and to store and retrieve loaded results.

Define a Text File Loader as an Engine Component

This section will define a `TextFileLoader` object to work with `ResourceMap` to load text files asynchronously. This object serves as an excellent example of how to take advantage of the `ResourceMap` facility and how to replace the synchronous loading of GLSL shader files.

1. Create a new file in the `src/Engine/Core/Resources` folder and name it `Engine_TextFileLoader.js`.
2. Define the `TextFileLoader` similar to other engine components.

```
var gEngine = gEngine || { };

gEngine.TextFileLoader = (function() {
    var mPublic = {};
    return mPublic;
}());
```

3. Define the different types of text files to be loaded, such as XML or plain text (for example, GLSL shaders).

```
var eTextFileType = Object.freeze({
    eXMLFile: 0,
    eTextFile: 1
});
```

■ **Note** `Object.freeze()` disables modification and makes an object immutable.

4. Add a function to asynchronously load a given text file.

```
var loadTextFile= function(fileName, fileType, callbackFunction) {
    if (!(gEngine.ResourceMap.isAssetLoaded(fileName))) {
        // Update resources in load counter.
        gEngine.ResourceMap.asyncLoadRequested(fileName);

        // Asyncrounsly request the data from server.
        var req = new XMLHttpRequest();
        req.onreadystatechange = function () {
            if ((req.readyState === 4) && (req.status !== 200)) {
                alert(fileName + ": loading failed!
                     [Hint: you cannot double click index.html to run this
                     project. " +
                    "The index.html file must be loaded by a web-server.]");
            }
        };
        req.open('GET', fileName, true);
        req.setRequestHeader('Content-Type', 'text/xml');

        req.onload = function () {
            var fileContent = null;
            if (fileType === eTextFileType.eXMLFile) {
                var parser = new DOMParser();
                fileContent = parser.parseFromString(req.responseText, "text/xml");
            } else {
                fileContent = req.responseText;
            }
            gEngine.ResourceMap.asyncLoadCompleted(fileName, fileContent);

            if ((callbackFunction !== null) &&(callbackFunction !== undefined))
                callbackFunction(fileName);
        };
        req.send();
    } else {
        if ((callbackFunction !== null) && (callbackFunction !== undefined))
            callbackFunction(fileName);
    }
};
```

With the listing, notice the following:

a. The `fileName` parameter is the path to the text file, and it will be used as the unique resource name.

b. The `fileType` parameter is of `eTextFileType` indicating whether the load request is meant for an XML file or a simple text file.

c. The `ResourceMap.isAssetLoaded()` function is first called to ensure that the requested resource is not already loaded before an actual loading command is issued. This ensures that the same resource will be loaded only once and will be shared.

d. The `ResourceMap.asyncLoadRequested(fileName)` function is called to register a new asynchronous `XMLHttpRequest` loading request. Take note of how these two classes collaborate; `ResourceMap` registers that there is one outstanding load operation, while `TextFileLoader` is performing the actual loading operation.

e. The `ResourceMap.asyncLoadCompleted()` function is called and passed the loaded results when it becomes available. In this way, the `ResourceMap` entry identified by `fileName` will have its `MapEntry.mAsset` set to reference `fileContent`, the loaded asset.

f. Finally, when loading is done and if `callbackFunction` is defined, it will be called.

5. Complete the loader object by defining an appropriate unload function.

```
var unloadTextFile= function(fileName) {
    gEngine.ResourceMap.unloadAsset(fileName);
};
```

6. Lastly, remember to add public functions to the public interface.

```
// Public interface for this object.
var mPublic = {
    loadTextFile: loadTextFile,
    unloadTextFile: unloadTextFile,
    eTextFileType: eTextFileType
};
return mPublic;
```

Load Shaders Asynchronously

The `TextFileLoader` object can now be used to load 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. The callback function in the `ResourceMap` serves as the excellent indicator that loadings are completed and resources can be used.

Implement Default Resources Support

There are two interesting observations on the `SimpleShader` object, and together these observations lead to the definition of an engine-wide default resource storage support, or the `DefaultResources` component. First, to avoid loading the GLSL shader files synchronously, the files must be loaded before the creation of a `SimpleShader` object. Second, the `SimpleShader` object serves as a conduit for passing information and can be shared by `Renderable` objects for supporting the drawing of different constant color squares. Based on these observations, a `DefaultResources` engine component can be created to load GLSL

shader files during engine initialization and to create a sharable instance of `SimpleShader`. In general, the `DefaultResources` component can serve as the infrastructure that supports all future loading and sharing of game engine resources.

1. Create a new file in the `src/Engine/Core/Resources` folder and name it `Engine_DefaultResources.js`.

2. Define the `DefaultResources` similar to all other engine components.

```
var gEngine = gEngine || { };

gEngine.DefaultResources = (function() {
    var mPublic = {};
    return mPublic;
}());
```

3. Define the constant file paths to the GLSL shaders and the variable and assessor to the `SimpleShader` object.

```
// Simple Shader GLSL Shader file paths
var kSimpleVS = "src/GLSLShaders/SimpleVS.glsl";  // Path to the VertexShader
var kSimpleFS = "src/GLSLShaders/SimpleFS.glsl";  // Path to the simple
                                                  // FragmentShader

var mConstColorShader = null; // variable for the SimpleShader object
var _getConstColorShader = function() { return mConstColorShader; };  // assessor
```

4. Define the initialization function to initiate the GLSL shader loadings and set the target callback function to create the `SimpleShader` when all loading has completed.

```
// callback function after loadings are done
var _createShaders = function(callBackFunction) {
    mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);
    callBackFunction();
};
// initiate asynchronous loading of GLSL Shader files
var _initialize = function(callBackFunction) {
    // constant color shader: SimpleVS, and SimpleFS
    gEngine.TextFileLoader.loadTextFile(kSimpleVS,
        gEngine.TextFileLoader.eTextFileType.eTextFile);
    gEngine.TextFileLoader.loadTextFile(kSimpleFS,
        gEngine.TextFileLoader.eTextFileType.eTextFile);

    gEngine.ResourceMap.setLoadCompleteCallback(
        function() {_createShaders(callBackFunction);});
};
```

The `_initialize()` function issues the two loads for the GLSL vertex and fragment shader files and sets the `_createShaders()` to be the callback when the load operations are completed. The `_createShaders()` function in turn calls the `SimpleShader` constructor with the shader file paths to create the constant color shader. In the following, you will see how these file paths are used as resource names for retrieving the contents of the shader files.

5. Once again, remember to add all public functions to the public interface.

```
var mPublic = {
    initialize: _initialize,
    getConstColorShader: _getConstColorShader
};
return mPublic;
```

With DefaultResources, the game engine can now load and create constant color shaders during internal initialization and offers the mConstColorShader as a shared SimpleShader for all instances of Renderables. In the following, the required modifications to the SimpleShader implementation and the overall integration of all components are examined.

Modify SimpleShader to Retrieve Shader Files

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

1. Since no loading operations are required, you should change the _loadAndCompileShader() function name to simply _compileShader() and change the file-loading commands to content retrievals.

```
SimpleShader.prototype._compileShader = function(filePath, shaderType) {
    var gl = gEngine.Core.getGL();
    var shaderSource = null, compiledShader = null;

    // Step A: Access the shader textfile
    shaderSource = gEngine.ResourceMap.retrieveAsset(filePath);
    // ... identical to previous code ...
```

Notice that the synchronous loading operations are replaced by a single call to ResourceMap.retrieveAsset() to retrieve the file content based on the filePath or the unique resource name for the shader file.

2. Remember that in the SimpleShader constructor, the calls to _loadAndCompileShader() functions should be replaced by _compileShader() functions, as follows:

```
function SimpleShader(vertexShaderPath, fragmentShaderPath) {
   // ... identical to previous code ...

// Step A: load and compile vertex and fragment shaders
var vertexShader = this._compileShader(vertexShaderPath, gl.VERTEX_SHADER);
var fragmentShader = this._compileShader(fragmentShaderPath, gl.FRAGMENT_SHADER);
```

Recall that the instance of SimpleObject is created from the DefaultResources._createShaders() function where the path to the vertex and fragment shaders are passed in as parameters to the constructor. In the previous code, the file paths are used as unique resource names to retrieve the corresponding GLSL shader file contents from the ResourceMap.

Initialize Engine Core

The DefaultResources component must be initialized when the engine core is first initialized. In addition, an appropriate callback function must be provided for the DefaultResources._createShaders() function to call after the SimpleShader object is created.

1. Initialize DefaultResources in Engine_Core.js.

```
var initializeEngineCore = function(htmlCanvasID, myGame) {
    _initializeWebGL(htmlCanvasID);
    gEngine.VertexBuffer.initialize();
    gEngine.Input.initialize();

    // Inits DefaultResoruces, when done, invoke startScene(myGame).
    gEngine.DefaultResources.initialize(function() { startScene(myGame); } );
};
```

2. Define the startScene() function to initialize myGame and start the game loop.

```
var startScene = function(myGame) {
    myGame.initialize.call(myGame); // Called in this way to keep correct context
    gEngine.GameLoop.start(myGame); // start the game loop after initialization
};
```

With these modifications, Engine.Core initialization will call the DefaultResources initialization, causing the asynchronous loading of the GLSL shader files. DefaultResources in turn will call the startScene() function after the SimpleShader object is created. In this way, MyGame initialization will be properly called, and the system game loop will be properly started.

Modify index.html for Proper Engine Core Initialization

The last step in integrating all the changes is to ensure that the initializeEngineCore() function is properly activated, before the initialization of MyGame. To accomplish this, the index.html file's onload event handler must be modified to create an instance of MyGame and to call the initializeEngineCore() function with the instance of MyGame as a parameter.

```
<body onload="
    var myGame = new MyGame();
    gEngine.Core.initializeEngineCore('GLCanvas', myGame);
">
```

Logically, the modified onload event handler initializes the engine core by passing the area upon which the game should output to, GLCanvas, and a game object upon which the game should begin, myGame. In the following sections, the MyGame parameter to initializeEngineCore() will be generalized into any arbitrary game levels.

Test the Asynchronous Shader Loading

With `DefaultResources` handling the storage of the shaders and `index.html` initializing the engine core, `MyGame` object does not need to be concerned with any special initialization calls to engine core, and it can access `DefaultResources` for the sharable instance of `SimpleShader`.

1. Remove the call to `gEngine.Core.initializeEngineCore()` from the constructor.

```
function MyGame() {
    // variables for the squares
    this.mWhiteSq = null;        // these are the renderable objects
    this.mRedSq = null;

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

2. Remove the creation of `SimpleShaders` and reference to `DefaultResources` for the sharable `SimpleShader` to create `Renderable` objects as follows:

```
// Step  C: Create the renderable objects:
var constColorShader = gEngine.DefaultResources.getConstColorShader();
this.mWhiteSq = new Renderable(constColorShader);
this.mWhiteSq.setColor([1, 1, 1, 1]);
this.mRedSq = new Renderable(constColorShader);
this.mRedSq.setColor([1, 0, 0, 1]);
```

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

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

Game Level from a Scene File

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

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. You can see an example of this project running in Figure 4-4. This project appears and interacts identically to the previous project with the only difference being that the scene definition is asynchronously loaded from a file. The source code to this project is defined in the `Chapter4/4.4.SceneFile` folder.

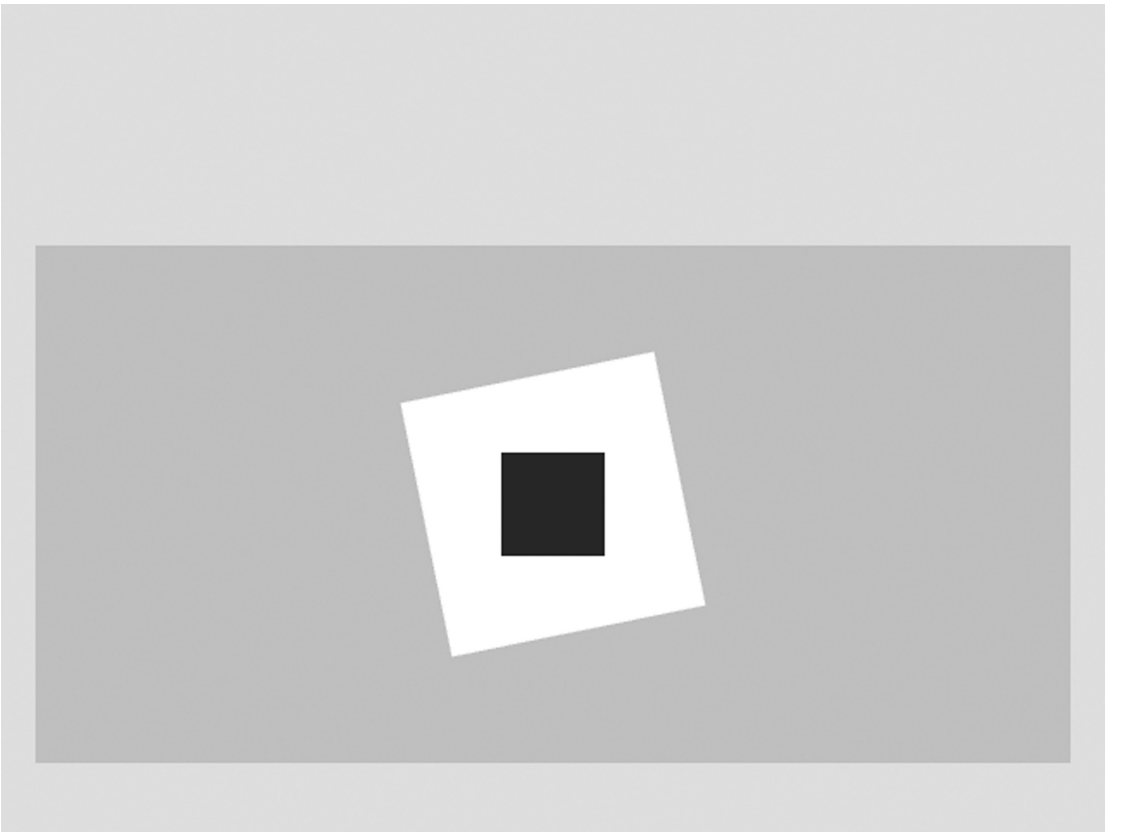

Figure 4-4. *Running the Scene File project*

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

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

The goals of the project are as follows:

- To 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

Keep in mind the ultimate goal of this project is to define the public interface methods between the game engine and a game level, or the client. While the definition/loading of a scene file is interesting, in this case it is but a vehicle. The following describes the definition and parsing utility for the scene file. It is important to remember these are only the tools for examining the required public methods for interfacing to the game engine.

The Scene File

Instead of hard-coding the creation of all objects to a game in the `initialize()` 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 all 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.

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/Core/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/GLSLShaders` 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:

```
<MyGameLevel>
    <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>
```

The listed XML content describes the same scene as defined in the `initialize()` functions from previous `MyGame` objects.

■ **Tip** Delimiting attributes with commas is not supported.

Parser for 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 MyGame folder.

1. Create a new folder in the src/MyGame folder and name it Util. Add a new file in the Util folder and name it SceneFileParser.js. This file will contain the specific parsing logic to decode the listed scene file.

2. Define the constructor for the SceneFileParser object.

```
function SceneFileParser(sceneFilePath) {
    this.mSceneXml = gEngine.ResourceMap.retrieveAsset(sceneFilePath);
}
```

Note that sceneFilePath is the complete path to the scene file. In this case, the file path is once again used as a unique resource name for retrieving the loaded asset from the ResourceMap.

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

---

3. Add a function to parse for contents of an XML element.

```
SceneFileParser.prototype._getElm = function(tagElm) {
    var theElm = this.mSceneXml.getElementsByTagName(tagElm);
    if (theElm.length === 0)
        console.error("Warning: Level element:[" + tagElm + "]: is not found!");
    return theElm;
};
```

4. Add a function to retrieve the definition of a camera.

```
SceneFileParser.prototype.parseCamera = function() {
    var camElm = this._getElm("Camera");
    var cx = Number(camElm[0].getAttribute("CenterX"));
    var cy = Number(camElm[0].getAttribute("CenterY"));
    var w = Number(camElm[0].getAttribute("Width"));
    var viewport = camElm[0].getAttribute("Viewport").split(" ");
    var bgColor = camElm[0].getAttribute("BgColor").split(" ");
    // make sure viewport and color are number
    for (var j = 0; j<4; j++) {
        bgColor[j] = Number(bgColor[j]);
        viewport[j] = Number(viewport[j]);
    }
    var cam = new 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 colors are arrays of four numbers. These are input as string 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 all strings are converted into numbers.

5. Add a function to parse for squares.

```
SceneFileParser.prototype.parseSquares = function(sqSet) {
    var elm = this._getElm("Square");
    var 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 Renderable(gEngine.DefaultResources.getConstColorShader());
        // make sure color array contains numbers
        for (j = 0; j<3; j++)
            c[j] = Number(c[j]);
        sq.setColor(c);
        sq.getXform().setPosition(x, y);
        sq.getXform().setRotationInDegree(r); // In Radian
        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.

Integrate Game Resource Loading

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

Public Methods of MyGame

At this point, it is established that `MyGame` should define the following:

- *Constructor*: For declaring variables and defining constants
- `initialize()`: For instantiating the variables and setting up the game scene
- `update()/draw()`: For interfacing to the game loop with these two functions being called continuously

With the requirement of loading a scene file, two additional public methods will be defined.

- `loadScene()`: For initiating the asynchronous loading of external resources, in this case, the scene file
- `unloadScene()`: For unloading of external resources when the game has ended

The implementations of these functions are as follows:

1. `MyGame` constructor:

```
function MyGame() {
    // scene file name
    this.kSceneFile = "assets/scene.xml"
    // all squares
    this.mSqSet = new Array();        // these are the renderable objects

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

The constructor defines the scene file path, the array `mSqSet` for storing the `Renderable` objects, and the camera.

2. Define the functions to load and unload the scene file.

```
MyGame.prototype.loadScene = function() {
    gEngine.TextFileLoader.loadTextFile(this.kSceneFile,
            gEngine.TextFileLoader.eTextFileType.eXMLFile);
};
MyGame.prototype.unloadScene = function() {
    gEngine.TextFileLoader.unloadTextFile(this.kSceneFile);
};
```

The `loadScene()` function initiates the asynchronous XML text file loading, while the `unloadScene()` function frees the resources.

3. Change the `initialize()` function to create objects based on the scene parser, as follows:

```
MyGame.prototype.initialize = function() {
    var sceneParser = new SceneFileParser(this.kSceneFile);

    // Step A: Parse the camera
    this.mCamera = sceneParser.parseCamera();

    // Step B: Parse for all the squares
    sceneParser.parseSquares(this.mSqSet);
};
```

Once again, notice that the file path to the scene file is passed into the constructor of `SceneFileParser` and will be used as the resource name for retrieving the scene file contents from the `ResourceMap`.

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

```
MyGame.prototype.draw = function() {
    // ... identical code to previous project ...
    // Step  C: draw all the squares
    for (var i = 0; i<this.mSqSet.length; i++) {
        this.mSqSet[i].draw(this.mCamera.getVPMatrix());
    }
};
MyGame.prototype.update = function() {
    // work with the xform of the white square ...
    var xform = this.mSqSet[0].getXform();
    // ... identical code to previous project ...

    // work with the red square ...
    xform = this.mSqSet[1].getXform();
    // ... identical code to previous project ...
};
```

Integration with the Game Engine

With the load and unload functionality defined in `MyGame`, the important task for the game engine is to ensure that `MyGame` initialization is called only after the load operation has completed. The game engine can ensure this sequence by coordinating the operations of engine core initialization and starting the game loop.

1. In `Engine_Core.js`, at the end of `initializeEngineCore()` after `DefaultResources` initialization is completed, the callback `startScene()` function needs to be modified to initiate the loading of the external resources of `MyGame` with the call to the `myGame.loadScene()` function.

```
var startScene = function(myGame) {
    myGame.loadScene.call(myGame);  // Called in this way to keep correct context
    gEngine.GameLoop.start(myGame); // call initialize() only after async
    loading is done
};
```

2. In `Engine_GameLoop.js`, always wait for any potential `MyGame` asset loading to be completed before initializing the game and starting the loop.

```
var _startLoop = function() {
    // Step A: reset frame time
    mPreviousTime = Date.now();
    mLagTime = 0.0;
    // Step B: remember that loop is now running
    mIsLoopRunning = true;
    // Step C: request _runLoop to start when loading is done
    requestAnimationFrame(function(){_runLoop.call(mMyGame);});
};
```

```
    var start = function(myGame) {
        mMyGame = myGame;
        gEngine.ResourceMap.setLoadCompleteCallback(
                function() {
                    mMyGame.initialize();
                    _startLoop();
                });
    };
```

In the previous listing, the `_startLoop()` function implements the continuous cycling of the game loop. The `start()` function registers a call with the `ResourceMap` to wait for the completion of `MyGame`-loading operations before calling the game `initialize()` function, and then it begins the game loop. In this way, the game loop will begin only after all asynchronous loading operations are completed and the game is properly initialized.

You can now run the project and experience the identical behaviors with the previous two projects. Though less than overwhelming, it is important to remember the purpose of this project and recognize that through the process of supporting asynchronous loading of external resources, the public methods and calling sequence between the game engine and the client have been defined.

Before continuing, you may notice that the `MyGame.unloadScene()` 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 two projects.

Scene Object: Client Interface to the Game Engine

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

- The `EngineCore` component defines the `startScene()` function that will initiate the loading process of `MyGame` and call to start the game loop.
- The `GameLoop.start()` function registers with the `ResourceMap` to wait for the completion of all asynchronous loading operations before it calls to initialize `MyGame` and starts the actual game loop cycle.

In the previous 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 `startScene()` function to load a new scene. This section pursues this observation by introducing the `Scene` object for interfacing the game engine with its clients.

The Scene Objects Project

This project defines the `Scene` object 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 well-defined and uniform manner. You can see an example of this project running in Figure 4-5. The source code to this project is defined in the `Chapter4/4.5.SceneObjects` folder.

Figure 4-5. *Running the Scene Objects project with both scenes*

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

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.

The goals of the project are as follows:

- To define the abstract `Scene` object 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 Object

Based on the experience from the previous project, an abstract `Scene` object for encapsulating the interface to the game engine must define these functions: constructor, `initialize()`, `loadScene()`, `unloadScene()`, `update()`, and `draw()`.

1. Create a new JavaScript file in the `src/Engine` folder and name it `Scene.js`.
2. Implement the following functions:

```
function Scene() { // constructor }
Scene.prototype.initialize = function() {
    // Called from GameLoop, after loading is done
};
Scene.prototype.loadScene = function() {
    // called from EngineCore.startScene()
};
Scene.prototype.unloadScene = function() { };
Scene.prototype.update = function() { };
Scene.prototype.draw = function() { };
```

Notice that the Scene object is an abstract object because all the functions are empty. 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 objects. The language does not prevent a game programmer from instantiating a Scene object. However, the created instance will be completely useless.

Modify Game Engine to Support the Scene Object

The game engine core components can now be modified to support the explicitly defined Scene object.

1. To facilitate inheritance of the Scene prototype methods, the inheritPrototype() function is defined in the Engine_Core.js as follows:

```
var inheritPrototype = function(subClass, superClass) {
    var prototype = Object.create(superClass.prototype);
    prototype.constructor = subClass;
    subClass.prototype = prototype;
};
```

The inheritPrototype() function is a utility function that passes a reference to the prototype of one object to another. This is called *prototypal inheritance* and allows the inherited object to access the prototype functions of the original object. Note that the subclass's constructor is explicitly saved to prevent it from being overwritten.

2. In the Engine_GameLoop.js file, include support to stop the game loop and remember to add this function to public interface:

```
var stop = function() {
    mIsLoopRunning = false;
};
var mPublic =  {
    start: start,
    stop: stop
};
```

3. To support orderly unloading of the currently running scene, the game loop must be stopped first, and then the unload function of the currently running scene can be called. With the previous stop() function, the unloading can be implemented by modifying the _runLoop() function in Engine_GameLoop.js to call the unloadScene() function when the game loop has stopped.

```
var _runLoop = function () {
    if(mIsLoopRunning) {
        // ... identical to previous code ...
    } else {
        // the game loops has stopped, unload current scene!
        mMyGame.unloadScene();
    }
};
```

Test the Scene Object Interface to the Game Engine

With the abstract Scene object definition and the simple modification to the game engine core components, it is now possible to stop an existing scene and load a new scene at will. This section uses the cycling between two subclasses of the Scene object, BlueLevel and MyGame, to illustrate the loading and unloading of scenes.

The BlueLevel Scene

Define a BlueLevel object that inherits from the Scene object and loads the scene from an external XML scene file.

1. Create a new XML scene file in the assets folder and name it BlueLevel.xml. Add a scene definition for the BlueLevel as follows:

```
<MyGameLevel>
    <Camera CenterX="20" CenterY="60" Width="20"
             Viewport="20 40 600 300"
             BgColor="0 0 1 1.0"/>
    <Square PosX="20" PosY="60" Width="5" Height="5" Rotation="30" Color="1 1 1 1" />
    <Square PosX="20" PosY="60" Width="2" Height="3" Rotation="0"
        Color="1 0 0 1" />
</MyGameLevel>
```

Besides minor size, position, or color differences, this file defines a scene that is similar to the one defined by the scene.xml file from the previous project.

2. Create a new file in the src/MyGame folder and name it BlueLevel.js.

3. Define a constructor for BlueLevel as follows:

```
function BlueLevel() {
    // scene file name
    this.kSceneFile = "assets/BlueLevel.xml";
    // all squares
    this.mSqSet = [];  // these are the renderable objects
    // The camera to view the scene
    this.mCamera = null;
}
gEngine.Core.inheritPrototype(BlueLevel, Scene);
```

Note the inheritPrototype() function is called right after the constructor. This overrides the prototype of the BlueLevel with that from the Scene and effectively defines all the behaviors of Scene on BlueLevel. It is important to call the inheritPrototype() function right after the constructor before any new prototype methods are defined for BlueLevel.

4. Override the Scene functions as follows:

```
BlueLevel.prototype.loadScene = function() {
    gEngine.TextFileLoader.loadTextFile(this.kSceneFile,
        gEngine.TextFileLoader.eTextFileType.eXMLFile);
};
```

```
BlueLevel.prototype.initialize = function() {
    // ... identical to MyGame.initialize from previous project ...
    // ... parse the scene from the this.kSceneFile ...
};
BlueLevel.prototype.draw = function() {
    // ... identical to MyGame.draw from previous project ...
    // ... set up the camera and draw all elements in the mSqSet array ...
};
BlueLevel.prototype.update = function() {
    // ... identical to MyGame.update from previous project ...

    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Left)) {
        xform.IncXPosBy(-deltaX);
        if (xform.getXPos() < 11) { // this is the left-boundary
            gEngine.GameLoop.stop();
        }
    }
};
BlueLevel.prototype.unloadScene = function() {
    // unload the scene flie
    gEngine.TextFileLoader.unloadTextFile(this.kSceneFile);

    var nextLevel = new MyGame();  // the next level
    gEngine.Core.startScene(nextLevel);
};
```

Many of the `BlueLevel` functions are similar to the corresponding functions of the `MyGame` object from the previous project, and thus much of the details are not shown. However, notice the following:

a. The `loadScene()` function initiates the asynchronous loading of the scene file and returns. It is important that the game engine wait for the completion of the load process before calling the `initialize()` function. Recall that in your game engine, the `GameLoop.start()` function implements this.

b. The `update()` function is responsible for initiating the level transition by calling the `GameLoop.stop()` function when the transition condition becomes favorable, in this case when the rectangle approaches and touches the left boundary from the right. Recall that the `GameLoop.stop()` function will signal, stop the game loop, and call the `unloadScene()` function.

c. The `unloadScene()` function is called when the game loop has stopped. At this point, since the game loop is not running, no update or draw function will be executed, and all resources are free from being accessed. This is the opportunity to unload and free up `ResourceMap` entries.

d. At the end of the `unloadScene()` function, the next scene, `MyGame`, is created and passed to the `EngineCore.startScene()` function, which will load, initialize, and run the `MyGame` scene.

The MyGame Scene

By this point, it may become clear that the MyGame scene definition is going to be quite similar to that of the BlueLevel where the simple scene will include a camera and two Renderable objects. The major distinction is that MyGame defines its entire scene in the initialize() function and does not load its scene from an external file. As in the case of BlueLevel, because of the similarities, only important differences will be highlighted in the code listings.

1. Change the MyGame constructor to inherit from the abstract Scene object.

```
function MyGame() {
    // ... smilar to previous code ...
    // ... declare a camera and two squares ...
}
gEngine.Core.inheritPrototype(MyGame, Scene);
```

The MyGame constructor is similar to the previous examples, with the exception of not defining the file path to a scene file and the call to the inheritPrototype() function to set Scene as the parent in inheritance.

2. Override the Scene functions as follows:

```
MyGame.prototype.initialize = function() {
     // ... similar to previous code ...
     // ... allocate and define the camera and two Renderables ...
 };
MyGame.prototype.draw = function() {
     // ... similar to previous code ...
     // ... set up the camera and draw the two Renderables
};
MyGame.prototype.update = function() {
    // ... identical to MyGame.update from previous project ...

    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Left)) {
        xform.incXPosBy(-deltaX);
        if (xform.getXPos() < 11) {  // this is the left-bound
         gEngine.GameLoop.stop();
        }
    }
};
MyGame.prototype.unloadScene = function() {
    var nextLevel = new BlueLevel();  // next level to be loaded
    gEngine.Core.startScene(nextLevel);
};
```

Similar to the case of `BlueLevel`, many of the functions are similar to the previous projects with the details not shown in the previous code listing. However, do take note of the following:

a. The `loadScene()` function is not defined. Since `MyGame` does not have any external resources to load, there is no need to override this function.

b. The `update()` function is almost identical to that from the `BlueLevel`. This should be expected because the two levels behave in almost an identical manner. It is once again important to note that when a level transition condition becomes favorable, the `GameLoop.stop()` function should be called to halt the game loop and cause the calling of the `unloadScene()` function.

c. The `unloadScene()` function is defined even though `MyGame` does not have any particular resources to unload. This is because the `unloadScene()` function is responsible for creating and transitioning the game to the next level. In this case, when the rectangle touches the left boundary, the game will transition back to the `BlueLevel`.

You can now run the project and experience the loading and unloading of the two scenes. Your game engine now has a well-defined interface for working with its client. This interface follows the well-defined protocol of the `Scene` object.

- Constructor: For declaring variables and defining constants
- `loadScene()`: For initiating the asynchronous loading of external resources
- `initialize()`: For instantiating the variables and setting up the game scene
- `update()/draw()`: For continuously receiving player input, implementing game logic, and displaying the game state
- `unloadScene()`: For unloading external resources and initiating the loading of the next scene by calling the `EngineCore.startScene()` function

Any objects that define the previous methods can be loaded and interacted with by your game engine. You can experiment with creating other levels and even trying the loading of the `MyGame` level in the `MyGame.unloadScene()` function.

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 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.

The Audio Support Project

This project is identical to the previous one where you can move the front rectangle left or right with the arrow keys, and the intersection with the left boundary triggers the loading of next scene, `MyGame` loads `BlueLevel`, and vice versa. However, in this version, each scene plays background music and triggers a brief audio cue when the left/right arrow key is pressed. The implementation of this project also reinforces the concept of loading and unloading of external resources and the audio clips themselves. You can see an example of this project running in Figure 4-6. The source code to this project is defined in the `Chapter4/4.6.AudioSupport` folder.

Figure 4-6. *Running the Audio Support project with both scenes*

The controls of the project are as follows:

- *Left/right arrow key*: Moves the front rectangle left and right

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
- To optimize and facilitate resource sharing with reference counts of individual resources

You can find the following audio files in the `Chapter4\4.6.AudioSupport\public_html\assets\sounds` folder:

- `BGClip.mp3`
- `BlueLevel_cue.wav`
- `MyGame_cue.wav`

Notice that the previous audio files are in two formats, `mp3` and `wav`. While both 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 AudioClips Component with Web Audio API

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/Core/Resources` folder. Second, both involve standardized file formats with existing 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 `http://www.w3.org/TR/webaudio/`.

---

1. In the `src/Engine/Core/Resources` folder, create a new file and name it `Engine_AudioClips.js`. This file will implement the `AudioClips` engine component.

```
var gEngine = gEngine || { };
gEngine.AudioClips = (function(){
    var mPublic = {};
    return mPublic;
}());
```

2. Declare variables to maintain references to the Web Audio context and background music.

```
var mAudioContext = null;
var mBgAudioNode = null;
```

3. Define a function to create a reference to the Web Audio context and store the results in `mAudioContext`.

```
var InitAudioContext = function() {
    try {
        var AudioContext = window.AudioContext || window.webkitAudioContext;
        mAudioContext = new AudioContext();
    }
    catch(e) {alert("Web Audio Is not supported.");}
};
```

4. Define the function to load an audio file asynchronously.

```
var loadAudio = function (clipName) {
    if (!(gEngine.ResourceMap.isAssetLoaded(clipName))) {
        // Update resources in load counter.
        gEngine.ResourceMap.asyncLoadRequested(clipName);

        // Asyncrounsly request the data from server.
        var req = new XMLHttpRequest();
```

```
            req.onreadystatechange = function () {
                if ((req.readyState === 4) && (req.status !== 200)) {
                    alert(clipName + ": loading failed!
                          [Hint: you cannot double click index.html to run
                          this project. " +
                         "The index.html file must be loaded by a web-server.]");
                }
            };
            req.open('GET', clipName, true);
            // Specify that the request retrieves binary data.
            req.responseType = 'arraybuffer';

            req.onload = function () {
                // Asynchronously decode, then call the function in parameter.
                mAudioContext.decodeAudioData(req.response,
                    function(buffer) {
                        gEngine.ResourceMap.asyncLoadCompleted(clipName, buffer);
                    }
                );
            };
            req.send();
        } else {
            gEngine.ResourceMap.incAssetRefCount(clipName);
        }
    };
```

Note the similarity between the `loadAudio()` function and the `loadTextFile()` function where the `ResourceMap` is first consulted to ensure the requested resource is not already loaded, followed by registering a new asynchronous load, and lastly, when the load is completed, the request for storage of the loaded resource.

a. `ResourceMap.isAssetLoaded()`: Checks whether the requested resource is already loaded. Notice that if a resource is indeed already loaded, there is a call to a `ResourceMap.incAssetRefCount()` function. This function ensures resources are reference counted and can be properly shared across different load requests. Details to reference counting will be described in the next section.

b. `ResourceMap.asyncLoadRequested()`: Registers an outstanding asynchronous loading with the `ResourceMap`.

c. `ResourceMap.asyncLoadCompleted()`: Informs the `ResourceMap` that an asynchronous loading has completed and stores the loaded asset in the `ResourceMap`.

Notice the actual loading of the audio file.

a. The `XMLHttpRequest` retrieves the corresponding data as an `arrayBuffer` object, which holds binary data.

b. The loaded binary data is decoded asynchronously into playable audio format by the `mAudioContext.decodeAudioData()` function. Be aware that the `asyncLoadCompleted()` function is called only when the decoding has been completed.

5. Add a function to unload the audio file.

```
var unloadAudio = function(clipName) {
    gEngine.ResourceMap.unloadAsset(clipName);
};
```

6. Now add a function to play the entire duration of an audio clip.

```
var playACue = function(clipName) {
    var clipInfo = gEngine.ResourceMap.retrieveAsset(clipName);
    if (clipInfo !== null) {
        // SourceNodes are one use only.
        var sourceNode = mAudioContext.createBufferSource();
        sourceNode.buffer = clipInfo;
        sourceNode.connect(mAudioContext.destination);
        sourceNode.start(0);
    }
};
```

The `playACue()` function uses the audio file path as a resource name to find the loaded asset from the `ResourceMap` and then invokes the Web Audio API to play the audio clip. Notice that no reference to the `sourceNode` 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.

7. Add the functionality to start, stop, and test for background audio.

```
var playBackgroundAudio = function(clipName) {
    var clipInfo = gEngine.ResourceMap.retrieveAsset(clipName);
    if (clipInfo !== null) {
        // Stop audio if playing.
        stopBackgroundAudio();
        mBgAudioNode = mAudioContext.createBufferSource();
        mBgAudioNode.buffer = clipInfo;
        mBgAudioNode.connect(mAudioContext.destination);
        mBgAudioNode.loop = true;
        mBgAudioNode.start(0);
    }
};
var stopBackgroundAudio = function() {
    // Check if the audio is  playing.
    if(mBgAudioNode !== null) {
        mBgAudioNode.stop(0);
        mBgAudioNode = null;
    }
};
var isBackgroundAudioPlaying = function() {
    return (mBgAudioNode !== null);
};
```

In this case, notice that the `mBgAudioNode` keeps a reference to the currently running audio, and thus it is possible to stop the clip.

8. Finally, remember to add the public functions to the public interface.

```
var mPublic = {
    InitAudioContext: initAudioContext,
    loadAudio: loadAudio,
    unloadAudio: unloadAudio,
    playACue: playACue,
    playBackgroundAudio: playBackgroundAudio,
    stopBackgroundAudio: stopBackgroundAudio,
    isBackgroundAudioPlaying: isBackgroundAudioPlaying
};
return mPublic;
```

Counting Resource References

It is common for a resource to be shared and reused over the course of a game, for example, an audio cue effect or a background clip that is reused in different levels. In this case, a game may inevitably issue multiple load requests on these resources, potentially each time when a particular resource is needed. As described, loading external resources is a relatively expensive endeavor and should be avoided if possible. For this reason, in both the audio and text file loading, the `ResourceMap` is first consulted with the `isAssetLoaded()` function before the actual asynchronous load command is issued. This way, if a client issues more than one load request for the same resource, the game engine will perform only one actual load operation.

Since the client of the game engine is responsible for always unloading external resources that it has loaded, the support of multiple load requests must be complemented with the corresponding support for unload requests. For example, if there are two requests to load a particular resource, then it can be assumed that this resource is used in two separate cases, and there will be two corresponding unload requests for this resource. In this case, the resource must be kept in the `ResourceMap` until the second unload request is received. To properly support this behavior, individual resources must be reference counted. An integer counter is introduced to keep track of the number of load and unload requests on a resource, and the resource will be removed only when the value of this counter is zero.

To implement reference counting, an integer counter must be added to the `MapEntry` object in the `Engine_ResourceMap.js` file.

1. Open `Engine_ResourceMap.js` and add a counter variable with an initial value of 1 to the `MapEntry` constructor.

```
var MapEntry = function(rName) {
    this.mAsset = rName;
    this.mRefCount = 1;
};
```

Since the `MapEntry` object is created when the first load request is made on a resource, the initial value of 1 represents that there is one load command issued on the corresponding resource.

2. Add a function to increase the reference counter counting the number of times load commands have been issued on the resource, and remember to add this function to the public interface.

```
var incAssetRefCount = function(rName) {
    mResourceMap[rName].mRefCount += 1;
};
```

Each resource loader (for example, `Engine.AudioClip` or `Engine.TextFileLoader`) is responsible for calling this function when it detects a target resource has already been loaded. For example, earlier you saw this function being called from the `loadAudio()` function when an audio clip to be loaded is already present in the `ResourceMap`.

3. Complete the reference counting implementation by updating the `unloadAsset()` function.

```
var unloadAsset = function(rName) {
    var c = 0;
    if (rName in mResourceMap) {
        mResourceMap[rName].mRefCount -= 1;
        c = mResourceMap[rName].mRefCount;
        if (c === 0)
            delete mResourceMap[rName];
    }
    return c;
};
```

Notice that by checking the return value of the `unloadAsset()` function, a client can verify how many more unloads are necessary to free up the associated memory.

Testing the Audio Component

To test the audio component, you must copy the necessary audio files into your game project. Create a new folder in the `assets` folder and name it `sounds`. Copy the `BG_Clip.mp3`, `BlueLevel_cue.wav`, and `MyGame_cue` files into the `sounds` folder. 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.

```
function MyGame() { // audio clips: supports both mp3 and wav formats
    this.kBgClip = "assets/sounds/BGClip.mp3";
    this.kCue = "assets/sounds/MyGame_cue.wav";
    // ... Identical to previous code ...
}
```

Recall that these file paths are used as resource names for loading, storage, and retrieval. Declaring these as constants for later references is a good software engineering practice.

2. Request the loading of audio clips in the `loadScene()` function.

```
MyGame.prototype.loadScene = function() {
    gEngine.AudioClips.loadAudio(this.kBgClip);
    gEngine.AudioClips.loadAudio(this.kCue);
}
```

3. Remember to unload external resources that are loaded.

```
MyGame.prototype.unloadScene = function()  {
    // stop the background audio before unloading it
    gEngine.AudioClips.stopBackgroundAudio();

    // unload the scene resources
    // gEngine.AudioClips.unloadAudio(this.kBgClip);
    //      The above line is commented out on purpose because
    //      you know this clip will be used elsewhere in the game
    //      so you decide to not unload this clip!!
    gEngine.AudioClips.unloadAudio(this.kCue);

    // starts the next level
    var nextLevel = new BlueLevel();  // next level to be loaded
    gEngine.Core.startScene(nextLevel);
};
```

While it is important to unload resources, there can be exceptions. In this case, the background music clip is not unloaded because `BlueLevel` will be using the same resource. In addition, this is an excellent opportunity for testing resources reference counting.

4. Start the background audio at the end of the `initialize()` function.

```
MyGame.prototype.initialize = function() {
    // ... identical to previous code ...
    gEngine.AudioClips.playBackgroundAudio(this.kBgClip);
}
```

5. In the `update()` function, cue the players when the left and right arrow keys are pressed.

```
MyGame.prototype.update = function(){
    // ... Identical to previous code ...

    // Support hero movements
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Right)) {
        gEngine.AudioClips.playACue(this.kCue);
        xform.incXPosBy(deltaX);
        // ... identical to previous code ...
    }

    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Left)) {
        gEngine.AudioClips. playACue (this.kCue);
        xform.incXPosBy(-deltaX);
        // ... 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:

```
function BlueLevel() {
    // audio clips: supports both mp3 and wav formats
    this.kBgClip = "assets/sounds/BGClip.mp3";
    this.kCue = "assets/sounds/BlueLevel_cue.wav";
    // ... Identical to previous code ...
}
```

2. In addition to the scene file, request the loading of the audio clips in the loadScene() function.

```
BlueLevel.prototype.loadScene = function() {
    // load the scene file
    gEngine.TextFileLoader.loadTextFile(this.kSceneFile,
        gEngine.TextFileLoader.eTextFileType.eXMLFile);
    // loads the audios
    gEngine.AudioClips.loadAudio(this.kBgClip);
    gEngine.AudioClips.loadAudio(this.kCue);
};
```

3. Remember to stop background audio and unload all external resources.

```
BlueLevel.prototype.unloadScene = function() {
    // stop the background audio
    gEngine.AudioClips.stopBackgroundAudio();

    // unload the scene flie and loaded Resources
    gEngine.TextFileLoader.unloadTextFile(this.kSceneFile);
    gEngine.AudioClips.unloadAudio(this.kBgClip);
    gEngine.AudioClips.unloadAudio(this.kCue);

    var nextLevel = new MyGame();  // load the next level
    gEngine.Core.startScene(nextLevel);
};
```

4. In the same manner as MyGame, start the background audio in the initialize() function and cue the player when the left and right keys are pressed in the update() function.

```
BlueLevel.prototype.initialize = function() {
    // ... identical to previous code ...
    gEngine.AudioClips.playBackgroundAudio(this.kBgClip);
};
BlueLevel. prototype.update = function(){
    // ... Identical to previous code ...
```

```
        // Support hero movements
        if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Right)) {
            gEngine.AudioClips.playACue(this.kCue);
            xform.incXPosBy(deltaX);
            // ... identical to previous code ...
        }

        if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Left)) {
            gEngine.AudioClips.playACue (this.kCue);
            xform.incXPosBy(-deltaX);
            // ... identical to previous code ...
        }
    };
```

You can now run the project and listen to the wonderful audio feedback. Take note that when transitioning from the `MyGame` level to the `BlueLevel`, the background music reloading request actually triggers an integer increment (instead of a full reload/re-decode) and the music was stopped and restarted. 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 is an excellent example to illustrate 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 and has everything needed in order to create basic games. 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 as well as audio cues.

These components separately have little in common but together make up the core fundamentals of nearly every game. As you implement these common 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 provided by the engine 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 or the technical foundation contributing to the apparent immediate connection between what the player does and how the game responds. If a player grabs 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 that begins 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 object and action responsiveness play 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 actions 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 "chunk" of game play 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 contained completely within the white square in order to complete the level and move on to the next challenge; once the player met the conditions of that causal chain (that is, once the player successfully placed the red square in the white square), the level would complete. This basic mechanic isn't quite enough on its own to create a fun 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 to begin creating 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 Loads 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; as you continue through the chapters, you'll explore how audio contributes to game design in more detail.