# Scene Object: Client Interface to the Game Engine

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

* The window.onload function initialize the game engine and calls the loop.start() function passing in MyGame as the parameter.
* The loop.start() function, through the resource\_map, 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 loop.start() function to initiate the loading of 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.scene\_objects 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
* Q key: Quits the game

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

The goals of the project are as follows:

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

### The Abstract Scene Object

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

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

"use strict";

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

import engine from "./index.js";

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

1. Define JavaScript Error objects for warning the client in cases of miss-use.

const kAbstractClassError = new Error("Abstract Class")

const kAbstractMethodError = new Error("Abstract Method")

1. Create a new class named Scene and export it.

class Scene { … }

export default Scene;

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

constructor() {

if (this.constructor === Scene) {

throw kAbstractClassError

}

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

async start() {

await loop.start(this);

}

next() {

loop.stop();

this.unload();

}

stop() {

loop.stop();

this.unload();

engine.cleanUp();

}

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

init() { /\* to initialize the level (called from loop.start()) \*/ }

load() { /\* to load necessary resources \*/ }

unload() { /\* unload all resources \*/ }

// draw/update must be over-written by subclass

draw() { throw kAbstractMethodError; }

update() { throw kAbstractMethodError; }

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

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

### Modify Game Engine to Support the Scene Object

The game engine must be modified in two important ways. First, as you will continue to observe when new functionality is introduced the game engine access file, index.js, must be modified to export the newly introduced symbols to the client. Second, the Scene.stop() function introduce the possibility of stopping the game, and, when that happens, it is necessary to cleanup all system components and free up allocated resources.

#### Export the Scene Class to the Client

Edit the index.js file to import from scene.js and export the default Scene class identifier for the client.

// … identical to previous code

// … identical to previous code

export default {

// … identical to previous code

Camera, Scene, Transform, Renderable,

// … identical to previous code

}

#### Implement Engine Cleanup Support

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

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

// … identical to previous code

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

// … identical to previous code

// … identical to previous code

export default {

// … identical to previous code

init, cleanUp, clearCanvas

// … identical to previous code

}

**Note** Similar to other core engine internal components, such as gl, or vertex\_buffer, loop should not be accessible by the client. For this reason, loop module is imported but not exported by index.js. Imported such that game loop cleanup can be invoked, not exported such that the client can be shield off from the unnecessary complexities of the engine.

Notice that none of the components have define the corresponding cleanup functions. You will now remedy this.

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

// … identical to previous code

function cleanUp() {

if (mLoopRunning) {

stop();

// unload all resources

mCurrentScene.unload();

mCurrentScene = null;

}

}

export {start, stop, cleanUp}

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

// … identical to previous code

function cleanUp() {} // nothing to do for now

export {keys, init, cleanUp,

// … identical to previous code

1. Edit shader\_resources.js to define, and export a cleanUp(), function to clean up the created shader and unload the shader source code.

// … identical to previous code

function cleanUp() {

mConstColorShader.cleanUp();

text.unload(kSimpleVS);

text.unload(kSimpleFS);

}

export {init, cleanUp, getConstColorShader}

1. Edit simple\_shader.js to define thefor the SimpleShader class release the allocated WebGL resources
2. Edit vertex\_buffer.js to define, and export, a cleanUp() function to delete the allocated buffer memory.

// … identical to previous code

function cleanUp() {

if (mGLVertexBuffer !== null) {

glSys.get().deleteBuffer(mGLVertexBuffer);

mGLVertexBuffer = null;

}

}

export {init, get, cleanUp}

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

// … identical to previous code

function cleanUp() {

if ((mGL == null) || (mCanvas == null))

throw new Error("Engine cleanup: system is not initialized.");

mGL = null;

// let the user know

mCanvas.style.position = "fixed";

mCanvas.style.backgroundColor = "rgba(200, 200, 200, 0.5)";

mCanvas = null;

document.body.innerHTML += "<br><br><h1>End of Game</h1><h1>GL System Shut Down</h1>";

}

export {init, get, cleanUp}

### Test the Scene Object Interface to the Game Engine

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

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

#### The MyGame Scene

As mentioned, this scene defines in the init() function the identical content found in the scene file from the previous project. In the following, take note of the definition and calls to next() and stop() functions.

1. Edit my\_game.js to import from index.js and the newly defined blue\_level.js. Note that with Scene class support, you no-longer need to import from the loop module.

import engine from "../engine/index.js";

import BlueLevel from "./blue\_level.js";

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

class MyGame extends engine.Scene {

…

}

export default MyGame;

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

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

constructor() {

super();

// The camera to view the scene

this.mCamera = null;

// the hero and the support objects

this.mHero = null;

this.mSupport = null;

}

init() {

// Step A: set up the cameras

this.mCamera = new engine.Camera(

vec2.fromValues(20, 60), // position of the camera

20, // width of camera

[20, 40, 600, 300] // viewport (orgX, orgY, width, height)

);

this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);

// Step B: Create the support object in red

this.mSupport = new engine.Renderable();

this.mSupport.setColor([0.8, 0.2, 0.2, 1]);

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

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

// Setp C: Create the hero object in blue

this.mHero = new engine.Renderable();

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

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

this.mHero.getXform().setSize(2, 3);

}

draw() {

// Step A: clear the canvas

engine.clearCanvas([0.9, 0.9, 0.9, 1.0]);

// Step B: Activate the drawing Camera

this.mCamera.setViewAndCameraMatrix();

// Step C: draw everything

this.mSupport.draw(this.mCamera);

this.mHero.draw(this.mCamera);

}

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

update() {

// let's only allow the movement of hero,

// and if hero moves too far off, this level ends, we will

// load the next level

let deltaX = 0.05;

let xform = this.mHero.getXform();

// Support hero movements

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

xform.incXPosBy(deltaX);

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

xform.setPosition(12, 60);

}

}

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

xform.incXPosBy(-deltaX);

if (xform.getXPos() < 11) { // this is the left-bound of the window

this.next();

}

}

if (engine.input.isKeyPressed(engine.input.keys.Q))

this.stop(); // Quit the game

}

1. Define the next() function to transition to the BlueLevel scene.

next() {

super.next(); // this must be called!

// next scene to run

let nextLevel = new BlueLevel(); // next level to be loaded

nextLevel.start();

}

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

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

window.onload = function () {

engine.init("GLCanvas");

let myGame = new MyGame();

myGame.start();

}

#### The BlueLevel Scene

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

1. Create and edit blue\_level.js file in the my\_game folder to import from the engine index.js, MyGame, and SceneFileParser. Define and export BlueLevel to be a subclass of the engine.Scene class.

// Engine Core stuff

import engine from "../engine/index.js";

// Local stuff

import MyGame from "./my\_game.js";

import SceneFileParser from "./util/scene\_file\_parser.js";

class BlueLevel extends engine.Scene {

…

}

export default BlueLevel

1. Define the init(), draw(), load(), and unload() functions to be identical to those in the MyGame class from the previous project.
2. Define the update() function similar to that of the MyGame scene. Once again, note the this.next() call when the object crosses the x=11 boundary from the right, and the this.stop() call when the Q key is pressed

update() {

// For this very simple game, let's move the first square

let xform = this.mSQSet[1].getXform();

let deltaX = 0.05;

/// Move right and swap ovre

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

xform.incXPosBy(deltaX);

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

xform.setPosition(12, 60);

}

}

// test for white square movement

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

xform.incXPosBy(-deltaX);

if (xform.getXPos() < 11) { // this is the left-boundary

this.next(); // go back to my game

}

}

if (engine.input.isKeyPressed(engine.input.keys.Q))

this.stop(); // Quit the game

}

1. Lastly, define the next() function to transition to the MyGame scene. It is worth reiterate the critical call to the super.next() to stop the game loop and unload the scene.

next() {

super.next();

let nextLevel = new MyGame(); // load the next level

nextLevel.start();

}

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

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

The following interface methods are meant to be over-written by subclasses.

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

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