# 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 interface 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 interface 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.

import Scene from "./scene.js";

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

export default {

// identical

text, xml,

// input support

input,

// Util classes

Camera, Scene, Transform, Renderable,

// functions

init, cleanUp, clearCanvas

}

1. Add a function to index.js named cleanUp() as follows.

function cleanUp() {

loop.cleanUp();

input.cleanUp();

shaderResources.cleanUp();

vertexBuffer.cleanUp();

glSys.cleanUp();

}

Notice that none of these functions currently exist. You will remidy this by adding cleanUp() functions to the engines various components and utilities.

1. Add a cleanUp() function to loop.js as follows. Remember since loop.js is a module you also need to export the function.

function cleanUp() {

if (mLoopRunning) {

stop();

// unload all resources

mCurrentScene.unload();

mCurrentScene = null;

}

}

export {start, stop, cleanUp}

1. Add a cleanUp() function to input.js as follows. Remember since input.js is a module you also need to export the function.

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

export {keys, init, cleanUp,

update,

isKeyClicked,

isKeyPressed}

1. Add a cleanUp() function to shader\_resources.js as follows. Remember since shader\_resources.js is a module you also need to export the function.

function cleanUp() {

mConstColorShader.cleanUp();

text.unload(kSimpleVS);

text.unload(kSimpleFS);

}

export {init, cleanUp, getConstColorShader}

1. Add a cleanUp() function to vertex\_buffer.js as follows. Remember since vertex\_buffer.js is a module you also need to export the function.

function cleanUp() {

if (mGLVertexBuffer !== null) {

glSys.get().deleteBuffer(mGLVertexBuffer);

mGLVertexBuffer = null;

}

}

export {init, get, cleanUp}

1. Add a cleanUp() function to gl.js as follows. Remember since gl.js is a module you also need to export the function.

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}

1. Add a cleanUp() function to the SimpleShader object as follows.

cleanUp() {

let gl = glSys.get();

gl.detachShader(this.mCompiledShader, this.mVertexShader);

gl.detachShader(this.mCompiledShader, this.mFragmentShader);

gl.deleteShader(this.mVertexShader);

gl.deleteShader(this.mFragmentShader);

gl.deleteProgram(this.mCompiledShader);

}

}

### 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, 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 blue\_level.xml. Add a scene definition for the BlueLevel as follows:

<MyGameLevel>

<!-- \*\*\* be careful!! comma (,) is not a supported syntax!! -->

<!-- make sure there are no comma in between attributes -->

<!-- e.g., do NOT do: PosX="20", PosY="30" -->

<!-- notice the "comma" between PosX and PosY: Syntax error! -->

<!-- cameras -->

<!-- Viewport: x, y, w, h -->

<Camera CenterX="20" CenterY="60" Width="20"

Viewport="20 40 600 300"

BgColor="0 0 1 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="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.

1. Create a new file in the src/my\_game folder and name it blue\_level.js.
2. Define a new class named BlueLevel that extends your newly created Scene abstract class. Also add imports for the engine and local utilities as well as the class export.

// 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 a constructor for BlueLevel as follows:

constructor() {

super();

// scene file name

this.mSceneFile = "assets/blue\_level.xml";

// all squares

this.mSQSet = []; // these are the Renderable objects

// The camera to view the scene

this.mCamera = null;

}

1. Override the Scene functions as follows:

init() {

let sceneParser = new SceneFileParser(engine.xml.get(this.mSceneFile));

// Step A: Read in the camera

this.mCamera = sceneParser.parseCamera();

// Step B: Read all the squares

sceneParser.parseSquares(this.mSQSet);

}

draw() {

// Step A: set up the camera

this.mCamera.setViewAndCameraMatrix();

// Step B: draw everything with the camera

let i;

for (i = 0; i < this.mSQSet.length; i++) {

this.mSQSet[i].draw(this.mCamera);

}

}

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.stop();

}

}

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

this.stop();

}

// If next() is not defined, then

// the default in the Scenes.next() will be called

// causing the end of the game

//

// next() {

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

// nextLevel.start();

// }

load() {

engine.xml.load(this.mSceneFile);

}

unload() {

// unload the scene flie and loaded resources

engine.xml.unload(this.mSceneFile);

}

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:

1. The load() function initiates the asynchronous loading of the scene file and returns. It is important that the game engine waits for the completion of the load process before calling the init() function. Recall that in your game engine, the loop.start() function implements this.
2. The update() function is responsible for initiating the level transition by calling the loop.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 loop.stop() function will signal, stop the game loop, and call the unload() function.
3. The unload() 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 resource\_map entries.
4. At the end of the unload() 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 init() 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 MyGame to extend the abstract Scene object.

class MyGame extends engine.Scene { … }

1. The MyGame constructor is similar to the previous examples, with the exception of not defining the file path to a scene file.