# Scene Object: Client Interface to the Game Engine

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

* The window.onload function contains the start() function that will initiate the loading process of MyGame and call to start the game loop.
* The loop.start() function registers with the resource\_map 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 start() 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.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

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(), start(), stop(), init(), load(), unload(), update(), and draw().

1. Create a new JavaScript file in the src/engine folder and name it scene.js.
2. Add the following imports:

"use strict";

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

import engine from "./index.js";

1. Add constants to notify the client when they are using an abstract class or method.

const abstractClassError = new Error("Abstract Class")

const abstractMethodError = new Error("Abstract Method")

1. Create a new class named Sceneand export it.

class Scene { … }

export default Scene;

1. Implement the following functions:

constructor() {

if (this.constructor === Scene) {

throw abstractClassError

}

}

async start() {

await loop.start(this);

}

stop() {

loop.stop();

this.unload();

this.next();

}

next() {

// if sub-class does not override this funciton

// then, this is the end of the game

engine.cleanUp();

}

init() {

// initialize the level and load resources (called from GameLoop)

// throw abstractMethodError

}

load() {

// to load necessary resources

}

unload() {

// .. unload all resources

// throw abstractMethodError

}

// update to be called form EngineCore.GameLoop

update() {

// when done with this level should call this.stop()

throw abstractMethodError

}

// draw to be called from EngineCore.GameLoop

draw() {

throw abstractMethodError

}

Notice that the Scene object is an abstract object because all the functions are empty, throw errors, or provide basic start/stop functionality. 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

TEMP TEXT

#### Handle the Unloading and Cleaning Up of Engine Resources

TEMP TEXT

1. Modify index.js to provide access to the Scene object via its import and export.

import Scene from "./scene.js";

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

export default {

// resource support

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);

}

#### Implement Default Resources for Commonly Used Engine Resources

TEMP TEXT

1. TEMP TEXT

"use strict";

import \* as font from "./font.js";

import SimpleShader from "../shaders/simple\_shader.js";

import TextureShader from "../shaders/texture\_shader.js";

import SpriteShader from "../shaders/sprite\_shader.js";

import LineShader from "../shaders/line\_shader.js";

import LightShader from "../shaders/light\_shader.js";

import IllumShader from "../shaders/illum\_shader.js";

import \* as map from "../core/internal/resource\_map.js";

// Global Ambient color

let mGlobalAmbientColor = [0.3, 0.3, 0.3, 1];

let mGlobalAmbientIntensity = 1;

function getGlobalAmbientIntensity() { return mGlobalAmbientIntensity; }

function setGlobalAmbientIntensity(v) { mGlobalAmbientIntensity = v; }

function getGlobalAmbientColor() { return mGlobalAmbientColor; }

function setGlobalAmbientColor(v) { mGlobalAmbientColor = vec4.fromValues(v[0], v[1], v[2], v[3]); }

// Simple Shader

let kSimpleVS = "src/glsl\_shaders/simple\_vs.glsl"; // Path to the VertexShader

let kSimpleFS = "src/glsl\_shaders/simple\_fs.glsl"; // Path to the simple FragmentShader

let mConstColorShader = null;

// Texture Shader

let kTextureVS = "src/glsl\_shaders/texture\_vs.glsl"; // Path to the VertexShader

let kTextureFS = "src/glsl\_shaders/texture\_fs.glsl"; // Path to the texture FragmentShader

let mTextureShader = null;

let mSpriteShader = null;

// Line Shader

let kLineFS = "src/glsl\_shaders/line\_fs.glsl"; // Path to the Line FragmentShader

let mLineShader = null;

// Light Shader

let kLightFS = "src/glsl\_shader/light\_fs.glsl"; // Path to the Light FragmentShader

let mLightShader = null;

// Illumination Shader

let kIllumFS = "src/glsl\_shader/illum\_fs.glsl"; // Path to the Illumination FragmentShader

let mIllumShader = null;

// Default font

let kDefaultFont = "assets/fonts/system\_default\_font";

function createShaders() {

mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);

mTextureShader = new TextureShader(kTextureVS, kTextureFS);

mSpriteShader = new SpriteShader(kTextureVS, kTextureFS);

mLineShader = new LineShader(kSimpleVS, kLineFS);

mLightShader = new LightShader(kTextureVS, kLightFS);

mIllumShader = new IllumShader(kTextureVS, kIllumFS);

}

// unload all resources

function cleanUp() {

mConstColorShader.cleanUp();

mTextureShader.cleanUp();

mSpriteShader.cleanUp();

mLineShader.cleanUp();

mLightShader.cleanUp();

mIllumShader.cleanUp();

core.text.unload(kSimpleVS);

core.text.unload(kSimpleFS);

core.text.unload(kTextureVS);

core.text.unload(kTextureFS);

core.text.unload(kLineFS);

core.text.unload(kLightFS);

core.text.unload(kIllumFS);

font.unload(kDefaultFont);

}

function init() {

let loadPromise = new Promise(

async function (resolve) {

await Promise.all([

core.text.load(kSimpleVS),

core.text.load(kSimpleFS),

core.text.load(kTextureVS),

core.text.load(kTextureFS),

core.text.load(kLineFS),

core.text.load(kLightFS),

core.text.load(kIllumFS),

font.load(kDefaultFont)

]);

resolve();

}).then(

function resolve() { createShaders(); }

);

map.pushPromise(loadPromise);

}

function getConstColorShader() { return mConstColorShader; }

function getTextureShader() { return mTextureShader; }

function getSpriteShader() { return mSpriteShader; }

function getLineShader() { return mLineShader; }

function getLightShader() { return mLightShader; }

function getIllumShader() { return mIllumShader; }

// font

function getDefaultFontName() { return kDefaultFont; }

export {

init, cleanUp,

// default system font name: this is guaranteed to be loaded

getDefaultFontName,

// shaders

getConstColorShader, getTextureShader, getSpriteShader, getLineShader, getLightShader, getIllumShader,

// Global ambient: intensity and color

getGlobalAmbientColor, setGlobalAmbientColor,

getGlobalAmbientIntensity, setGlobalAmbientIntensity

}

### 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.