# Game Level from a Scene File

The operations involved in initiating a game level from a scene file can assist in the derivation and refinement of the formal interface between the game engine and its client. With a game level defined in a scene file, the game engine must first initiate asynchronous loading, wait for the load completion, and then initialize the client for the game loop. These steps present a complete functional interface between the game engine and the client. By examining and deriving the proper support for these steps, the interface between the game engine and its client can be refined.

## 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.scene\_file 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 towards the right and wraps it to the left of the game window
* Up arrow key: Rotates the white square
* Down arrow key: Increases the size of the red square and then resets the size at a threshold

The goals of the project are as follows:

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

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

### Define an XML Resource Module

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

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

"use strict"

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

// functions from resource\_map

let unload = map.unload;

let has = map.has;

let get = map.get;

1. Instantiate an XML DOMParser, define the decode and parsing functions, and call the loadDecodeParse() function of the resource\_map with the corresponding parameters to initiate the loading of the XML file.

let mParser = new DOMParser();

function decodeXML(data) {

return data.text();

}

function parseXML(text) {

return mParser.parseFromString(text, "text/xml");

}

function load(path) {

return map.loadDecodeParse(path, decodeXML, parseXML);

}

1. Do not forget to export the defined functionality.

export {has, get, load, unload}

1. Lastly, remember to export the defined functionality for the client in the index.js.

import \* as xml from "./resources/xml.js";

// … identical to previous code except for the error handling message …

export default {

// resource support

text, xml,

// … identical to previous code except for the error handling message …

}

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

### Modify the Engine to Integrate Client Resource Loading

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

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

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

async function start(scene) {

if (mLoopRunning) {

throw new Error("loop already running")

}

mCurrentScene = scene;

mCurrentScene.load();

// Wait for any async requests before game-load

await map.waitOnPromises();

mCurrentScene.init();

mPrevTime = performance.now();

mLagTime = 0.0;

mLoopRunning = true;

mFrameID = requestAnimationFrame(loopOnce);

}

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

### Derive the Client Public Interface

Though slightly involved, the details of XML-parsing specifics are less important than the fact that now XML files can be loaded. 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
* init(): For instantiating the variables and setting up the game scene, This is called from the loop.start() function before the first iteration of the game loop.
* draw()/update(): For interfacing to the game loop with these two functions being called continuously from within the core of the game loop, in the loop.loopOnce() function.

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

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

### Implement the Client

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

#### Define a Scene File

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

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

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

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

<MyGameLevel>

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

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

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

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

<!-- cameras -->

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

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

Viewport="20 40 600 300"

BgColor="0.8 0.8 0.8 1.0"

/>

<!-- Squares Rotation is in degree -->

<Square PosX="20" PosY="60" Width="5" Height="5" Rotation="30" Color="1 1 1 1" />

<Square PosX="20" PosY="60" Width="2" Height="2" Rotation="0" Color="1 0 0 1" />

</MyGameLevel>

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

#### Parse the Scene File

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

1. Create a new folder in the src/my\_game folder and name it util. Add a new file in the util folder and name it scene\_file\_parser.js. This file will contain the specific parsing logic to decode the listed scene file.
2. Define a new class, name it SceneFileParser, and add a constructor with code as follows:

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

class SceneFileParser {

constructor (xml) {

this.xml = xml

}

// … additional code to follow …

}

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

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

1. Add a function to the SceneFileParser to parse the details of the camera from the xml file you created.

parseCamera() {

let camElm = getElm(this.xml, "Camera");

let cx = Number(camElm[0].getAttribute("CenterX"));

let cy = Number(camElm[0].getAttribute("CenterY"));

let w = Number(camElm[0].getAttribute("Width"));

let viewport = camElm[0].getAttribute("Viewport").split(" ");

let bgColor = camElm[0].getAttribute("BgColor").split(" ");

// make sure viewport and color are number

let j;

for (j = 0; j < 4; j++) {

bgColor[j] = Number(bgColor[j]);

viewport[j] = Number(viewport[j]);

}

let cam = new engine.Camera(

vec2.fromValues(cx, cy), // position of the camera

w, // width of camera

viewport // viewport (orgX, orgY, width, height)

);

cam.setBackgroundColor(bgColor);

return cam;

}

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

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

parseSquares(sqSet) {

let elm = getElm(this.xml, "Square");

let i, j, x, y, w, h, r, c, sq;

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

x = Number(elm.item(i).attributes.getNamedItem("PosX").value);

y = Number(elm.item(i).attributes.getNamedItem("PosY").value);

w = Number(elm.item(i).attributes.getNamedItem("Width").value);

h = Number(elm.item(i).attributes.getNamedItem("Height").value);

r = Number(elm.item(i).attributes.getNamedItem("Rotation").value);

c = elm.item(i).attributes.getNamedItem("Color").value.split(" ");

sq = new engine.Renderable();

// make sure color array contains numbers

for (j = 0; j < 4; j++) {

c[j] = Number(c[j]);

}

sq.setColor(c);

sq.getXform().setPosition(x, y);

sq.getXform().setRotationInDegree(r); // In Degree

sq.getXform().setSize(w, h);

sqSet.push(sq);

}

}

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

1. Add a function outside the SceneFileParser to parse for contents of an XML element.

function getElm(xmlContent, tagElm) {

let theElm = xmlContent.getElementsByTagName(tagElm);

if (theElm.length === 0) {

console.error("Warning: Level element:[" + tagElm + "]: is not found!");

}

return theElm;

}

1. Finally, export the SceneFileParser.

export default SceneFileParser;

#### Implement MyGame

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

1. Edit my\_game.js file and import the SceneFileParser.

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

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

constructor() {

// scene file name

this.mSceneFile = "assets/scene.xml";

// all squares

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

// The camera to view the scene

this.mCamera = null;

}

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

init() {

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

// Step A: Read in the camera

this.mCamera = sceneParser.parseCamera();

// Step B: Read all the squares

sceneParser.parseSquares(this.mSqSet);

}

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

draw() {

// Step A: clear the canvas

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

this.mCamera.setViewAndCameraMatrix();

// Step B: draw all the squares

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 white square and pulse the red

let xform = this.mSqSet[0].getXform();

let deltaX = 0.05;

// Step A: test for white square movement

// … identical to previous code

xform = this.mSqSet[1].getXform();

// Step C: test for pulsing the red square

// … identical to previous code e

}

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

load() {

engine.xml.load(this.mSceneFile);

}

unload() {

// unload the scene flie and loaded resources

engine.xml.unload(this.mSceneFile);

}

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 by the client, the public methods and calling sequence between the game engine and the client have been defined.

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