# 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.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 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 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.

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 initialize() 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** 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 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 object and name it SceneFileParser and add a constructor for the SceneFileParser object 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 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 resource\_map.

**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 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.

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;

### 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
* init(): 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.

* load(): For initiating the asynchronous loading of external resources, in this case, the scene file
* unload(): For unloading of external resources when the game has ended

The implementations of these functions are as follows:

1. Add an import for working with the SceneFileParser.

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

1. Modifiy 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, as follows. 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 resource\_map.

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

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

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

xform.setPosition(10, 60);

}

xform.incXPosBy(deltaX);

}

// Step B: test for white square rotation

if (engine.input.isKeyClicked(engine.input.keys.Up)) {

xform.incRotationByDegree(1);

}

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

// Step C: test for pulsing the red square

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

if (xform.getWidth() > 5) {

xform.setSize(2, 2);

}

xform.incSizeBy(0.05);

}

}

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);

}

#### 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. However, you first need to further expand resource\_map so that your engine can properly support asynchronous loading of xml resources.

1. Add a new file in the src/engine/resources folder and name it xml.js. This file will expand resource\_map to specifically work with xml files similarly to your implementation of text.js.
2. The details of xml.js implementation can be seen as follows:

"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;

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);

}

export {has, get, load, unload}

1. TEMP TEXT

async function start(scene) {

if (mLoopRunning) {

throw new Error("loop already running")

}

// Wait for any async requests before game-load

await map.waitOnPromises();

mCurrentScene = scene;

mCurrentScene.load();

await map.waitOnPromises();

mCurrentScene.init();

mPrevTime = performance.now();

mLagTime = 0.0;

mLoopRunning = true;

mFrameID = requestAnimationFrame(loopOnce);

}

In the previous listing, the start() function implements the continuous cycling of the game loop. The start() function registers a call with the resource\_map to wait for the completion of MyGame-loading operations before calling the game init() 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.

1. TEMP TEXT

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

export default {

// resource support

text, xml,

// input support

input,

// Util classes

Camera, Transform, Renderable,

// functions

init, clearCanvas

}

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.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.