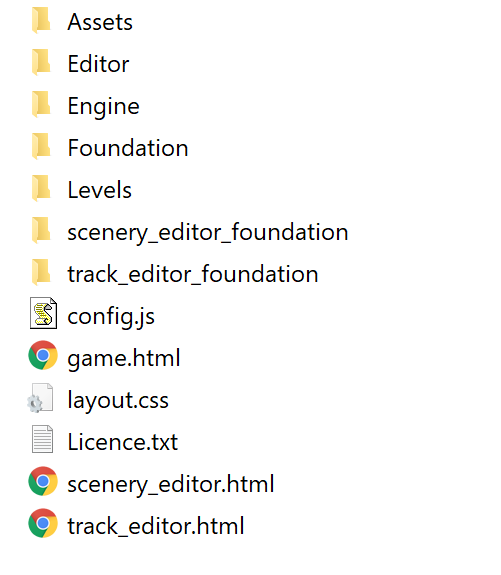
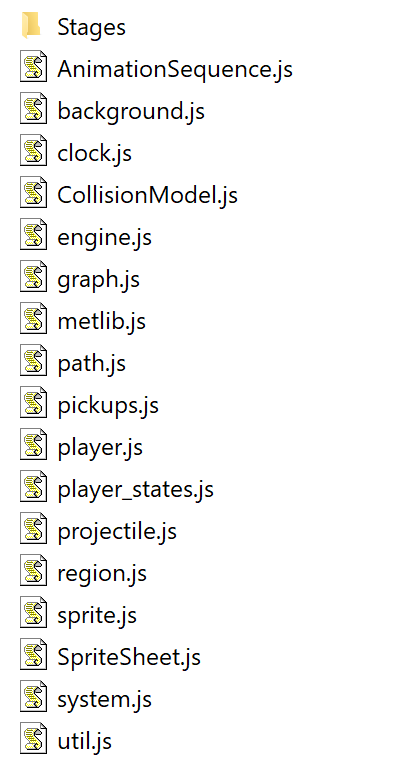
**OverDrive Game Engine - Overview**

This document outlines the structure of the OverDrive game engine and highlights key points in the code where you can change the game behaviour. Figure 1 shows the main project folder structure.



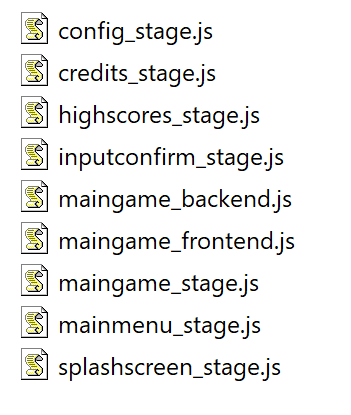
*Figure 1 - Project folder structure*

The game.html file contains main game demo. This sets up the main web page and loads a series of JavaScript files to setup the game engine and load the track / level data. The Engine folder contains the main OverDrive game engine. If you look inside the Engine folder, you’ll see a src folder and two JavaScript files. These are minified files. Minification in JavaScript is a way of stripping down the code so it loads quicker. It’s not typically human readable so you can ignore these for now. Inside the src folder is the main engine. The contents of the folder are shown in figure 2 below.



*Figure 2 - OverDrive Engine folder – main files*

The Stages folder is where most of the action happens. Figure 3 shows the different stages used in the OverDrive game demo given out in the Welcome pack.



*Figure 3 – Stages used in OverDrive*

The OverDrive engine uses stages to represent different activities in the game. This is a little like the different stages you show when storyboarding. For the OverDrive engine, the stages are linked together as follows…

*Figure 4 – Stages and how they’re linked*

The “maingame” stage is where most of the action happens. For OverDrive, this has been split into 3 files (maingame\_stage, maingame\_backend and maingame\_frontend). This is to better separate out concerns so we can see what’s happening at different levels of abstraction!

Each stage file is build around the Module programming pattern common in JavaScript. This is intended to emulate C++ style namespaces and keep each library’s types isolated from the global namespace.

Each stage has a constructor function (for example, maingame\_stage has the stage.MainGame constructor. The constructor function in JavaScript essentially creates a JSON object, which can contain data types and functions! Each stage has a number of functions – typically these are…

**Pretransition**

This is called before the stage starts – usually by another stage prior to transitioning.

**init**

This performs initial setup of resources needed for the stage. You might load game images and sounds here for example.

**initPhaseInLoop**

**phaseInLoop**

The “phase in” stage is intended to act as an introduction to the player / user entering the stage, but not part of the main game / stage activity. For example, if your screen fades into view for a few seconds, the phase-in function for the stage would handle this.

**mainLoop** (or mainLoopActual)

This is the actual main activity for the stage. For maingame\_stage this represents the main game loop activity.

**initPhaseOutLoop**

**phaseOutLoop**

The “phase out” part of the stage controls what happens when you’re just about to leave the stage. For example, if you win and the game displays your victory dance for a few seconds before proceeding to the next screen, the phaseOutLoop function controls this.

**leaveStage**

This is the last function called on the stage before the game engine moves onto the next stage.

The above functions encapsulate a sort of internal “state model” for each stage. You don’t have to implement these functions of course, but they keep functionality neatly separated.

**Implementing the game loop**

Each game is centred around the concept of a game loop. Very simply this is structured as follows…

While (!end-of-game) {

update\_game\_state();

render\_game\_state();

}

Modern games programming (particularly on the web and mobile) don’t implement this directly. Instead the host OS (or browser) controls updates / refresh. In JavaScript, at the end of each stage function, you’ll notice our code request a call back to the next function. This is done with the requestAnimationFrame function. The browser will at some point call back to the specified function. This is how we’ll implement the game loop in OverDrive.

Why do this? Well, if we just wrote the game loop as above, we may end up generating frames of animation quicker than the device can draw them (typically every 1/60th second). Everything our game does draws power and consumes CPU cycles, so the intent is to say “look, I know you can run at 200fps, but the user’s never going to see most of that, so let’s let the host OS / browser run the show so we don’t waste CPU cycles!”

**Configuring the game**

The main points you will want to focus on to change the game behaviour are the **init** function in **maingame\_stage.js** and the **config.js** file, located in the main project folder. Here you can setup pickups, change physics properties and define how the physics environment should behave. Check out the files – much of the code has been commented, but we can look through these in the sessions throughout the week.