Game

The game I re-created is COD zombies clone as a top-down 2D shooter. It is an endless survival game where the object is to try and survive waves of zombies whilst they get increasingly more difficult.

Shadow Engine

The game was created using my own engine (called *Shadow*). *Shadow* follows an *Entity Component System* (ECS) as well as independent systems (example Rendering and Collision) for certain tasks.

The individual *components* communicate with the needed system to create the desired outcome. Example, a '*SpriteRenderer*' *component* would communicate with the rendering system to load, render and destroy the sprite.

I did this to follow the '*single responsibility*' principle. A *component* is meant to be attached to a *game object* to give it behaviour or characteristics, and not be a system on its own. Keeping the systems separate results in being able to use these systems in other parts of the codebase without having to use the *component*.

Lastly, I tried to ensure that the engine and game is expandable. An example of this is with the gun system. Guns are loaded in from a file, meaning that it would be easy to add more guns. As well as this, this allows for a process where (if the engine was more refined and had a GUI) these methods could be serializable or exposed in the engine so that designers, in industry, could tweak the values to find the right feel of the game.

A brief description of ECS

An ECS is an architectural pattern where there are abstract data containers called *Entities* which contain a list of *components*, which give the *entities* certain behaviours or characteristics. This is advantageous as it avoids things like the *diamond inheritance* as well as improves codebase due to a component having a single responsibility.

Concerns with ECS

The main concern with ECS is performance. ECS can be created simply using inheritance – which is the approach I took; however, this can cause performance issues with larger projects. If we consider the *update* method, each game object and component have this method. Not all components or game objects need an *update* method, however, during execution, each *update* method will be called. For example, if I look at my walls, I have approximately 700 walls, each wall has 2 components, meaning that there are a total of 2100 *update* method calls that aren't needed. This is not a massive issue with this size project, however, can become an issue with larger projects.

Lessons learnt

The main lesson that I have learnt is better architecture for an engine as well as some overall coding improvements.

The biggest challenge that I had was a bug in my ECS causing the game to crash. This was caused by a game object being deleted in the middle of the game loop, and still being used at a different part in the game loop. This lead me to the different architecture approach.

Example, for change in architecture, have a boolean to determine if an object needs to be *deleted* at the end of the frame – after the update. And for codebase, having a method for a single responsibility instead of one massive function. This generally occurred in the Update or Initialize methods as they start small, but soon do a lot.