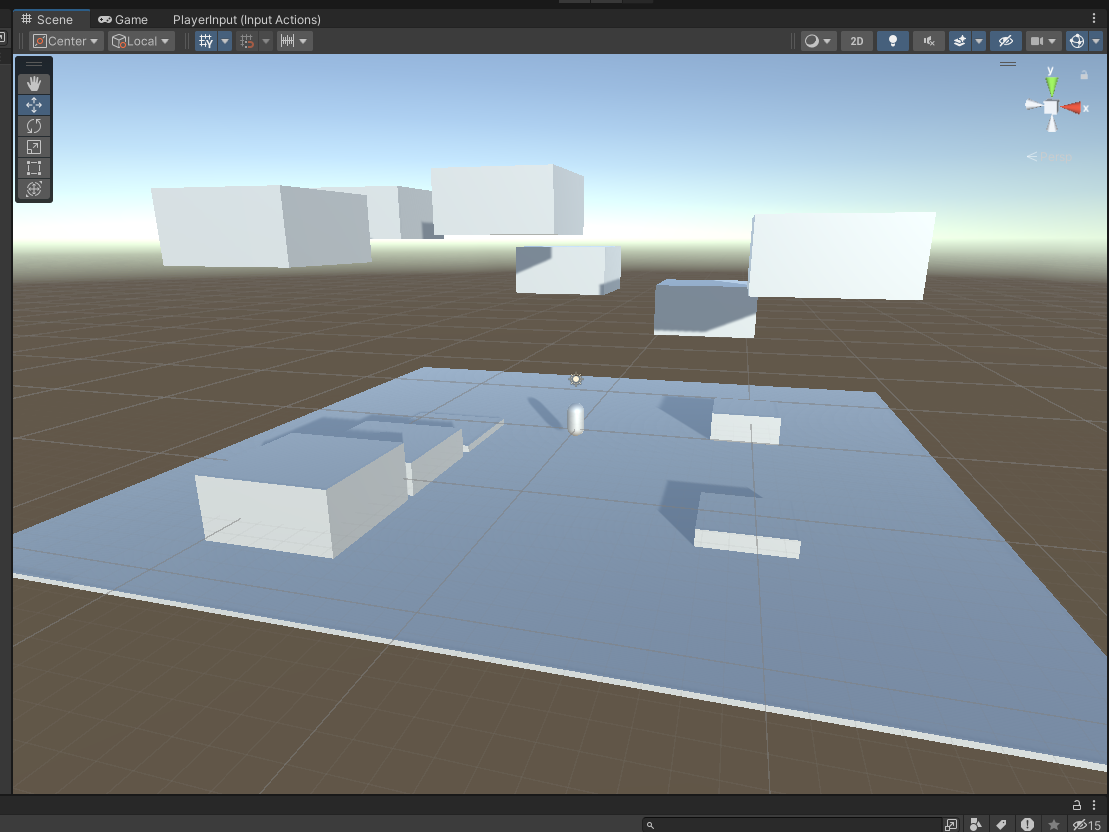
**Project Design**

The purpose of this document is to cover my design methodology for my project. It will cover prototyping, platform selection and my programming approach, providing details on each and explaining my approach.

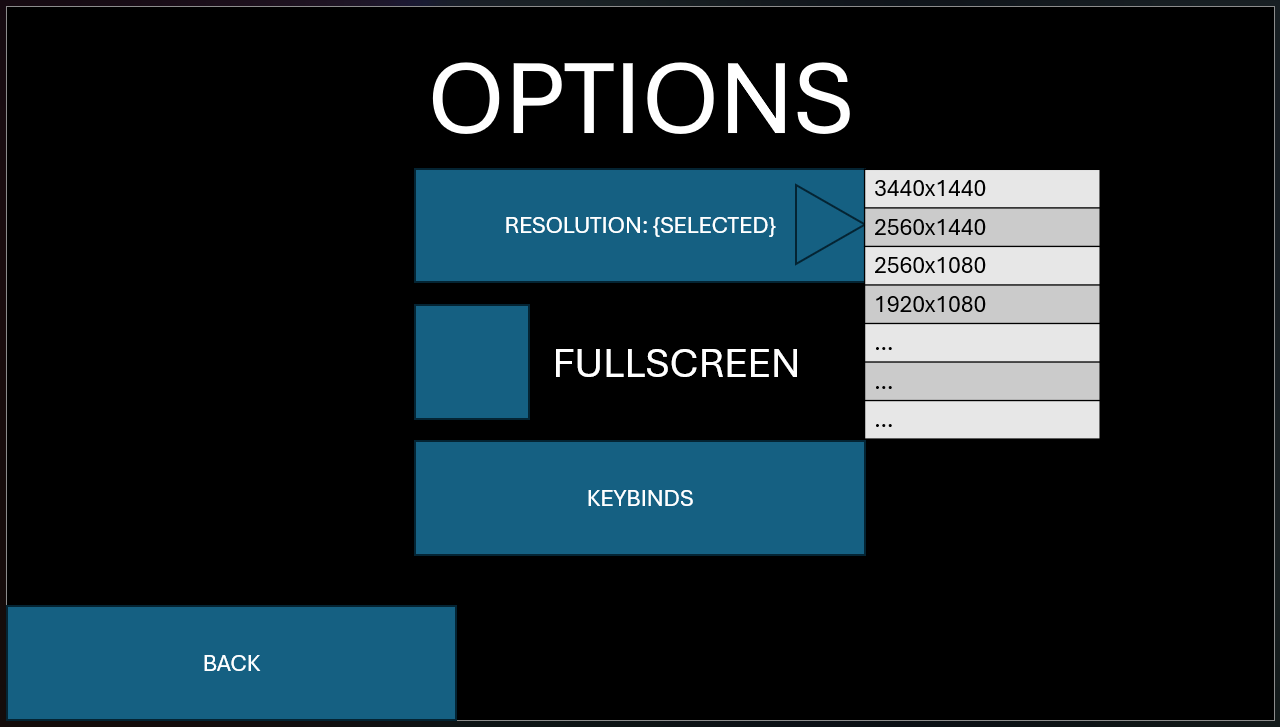
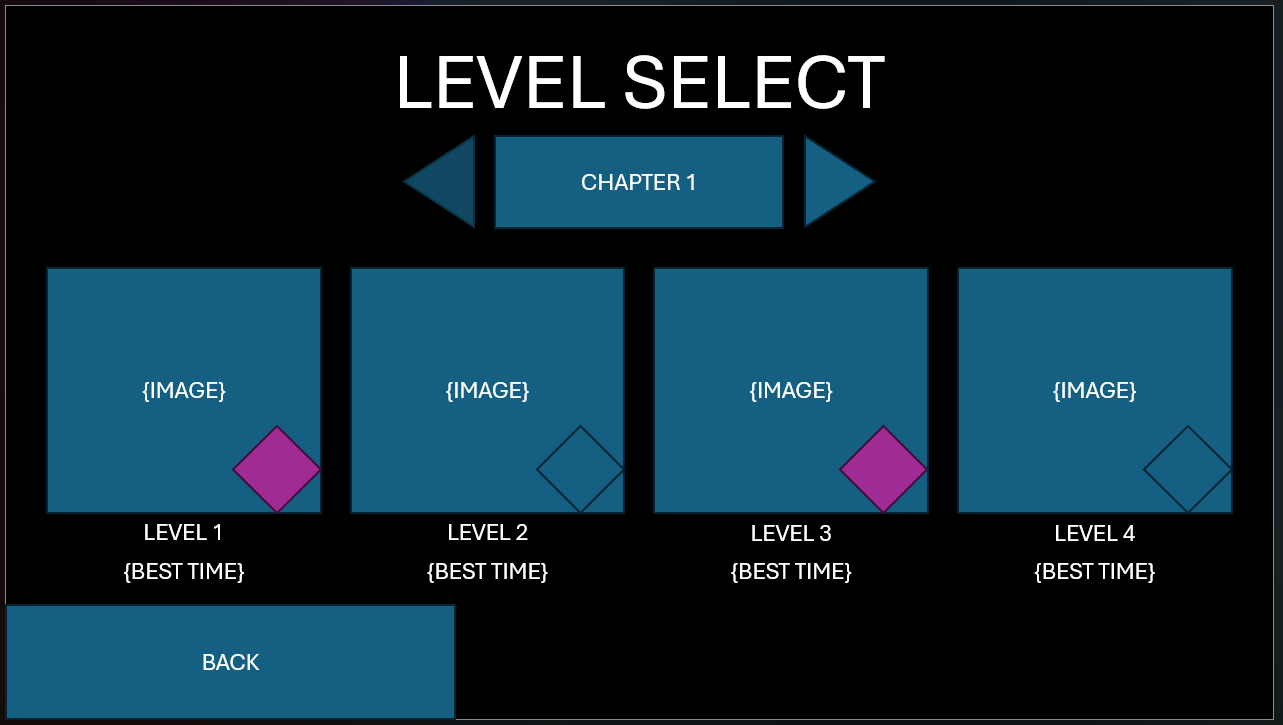
*Prototyping*

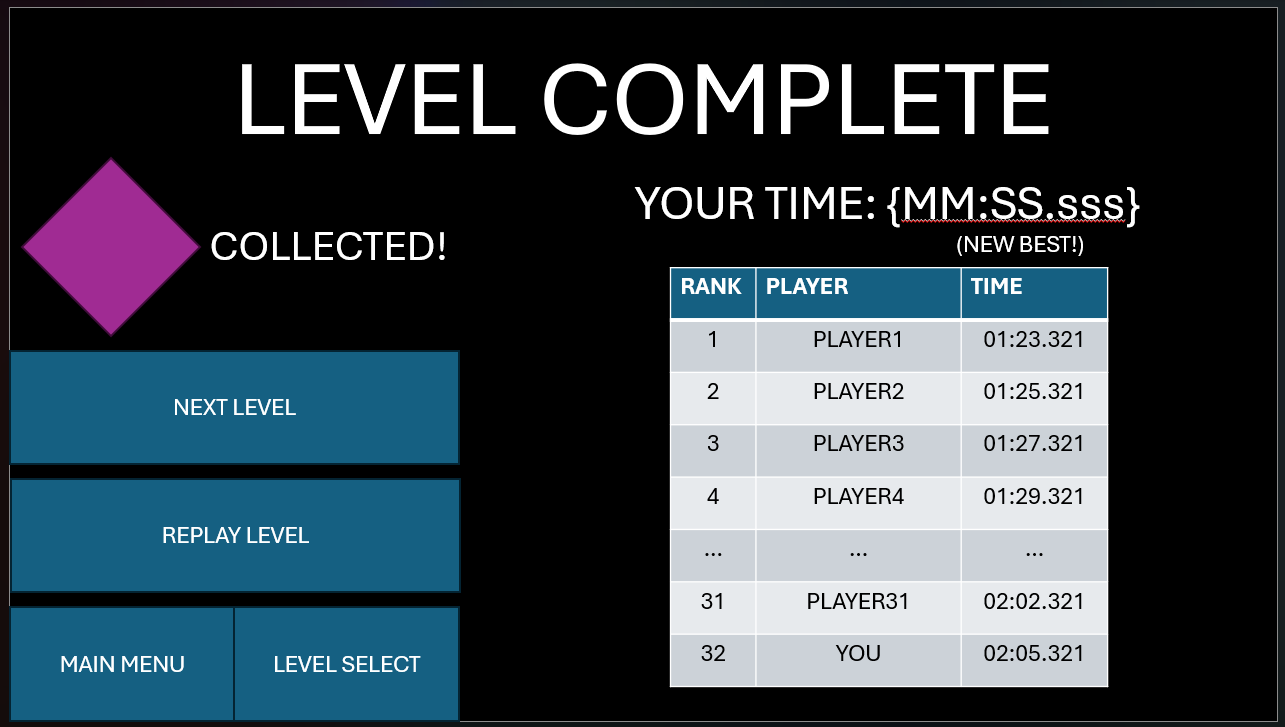
My prototyping approach began in the Unity engine itself. I decided that I would spend some time initially to ensure that my core gameplay systems would actually function, before working on feel and surrounding elements. Below is a screenshot of the environment I created for testing of this prototype, where the player is the capsule in the centre of the environment:



This prototype allows me to enter the game and move around as the player character to test my core gameplay elements. It does not cover things like levels or menus, as I felt it was most important to develop a prototype around the core gameplay first.

I then went about prototyping menus using external tools. Having used it for similar purposes in the past, I simply made these prototypes in Microsoft PowerPoint. This allowed me to quickly mock up UI screens, and model interactions between them. On the next page are the slides I created for mocking up the most important menu UI elements.





*Platform Selection*

I decided quite early to use Unity as the game engine for my game. This was primarily because scripting for Unity is done in C#, a language I am now quite comfortable using. There are alternative engines available that can also support C# scripting, such as Godot Engine, which has been developing significant traction in recent months. However, my research into Godot showed that while it is a very lightweight engine and has strong performance, Unity has more resources available to aid in development, and is more focused on C#. Godot does support C# scripting through the use of a tool called Mono, but I felt my time would be better spent writing C# code directly in Unity.

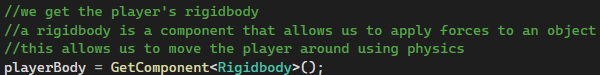
Another factor that led to my decision is one of each engine’s focus. Godot has gained recognition for its support of 2D game development, and it includes many tools made specifically with 2D games in mind. While 3D game development is certainly possible within Godot, Unity was always tailored for 3D games, with strong 3D performance and built-in features like global illumination. Unity also supports an extensive Asset Store, where users can obtain 3D assets for use in their projects, along with all the relevant usage rights. Godot has a growing community, with its members creating a number of potentially useful assets, however I have found it less intuitive to navigate the various locations where assets are shared.

The final factor I considered that impacted my decision was one of compatibility. While both engines offer compatibility with a wide variety of release platforms, my research showed that Godot offers more limited console support compared to Unity. While the aims of this project only include development of a PC version of my game demo, if I were to revisit this in the future and create a full game, it would be in my best business interests to make my game as widely available as possible. Using Godot to create this demo would make this much more difficult.

*Programming Approach*

When it comes to programming my project, I have had to shift my approach compared to how I normally program. Since scripts in Unity are written individually, with little direct interaction, the process of creating new objects has been quicker, with much less coupling between objects. Since GameObjects represent entities in the game world, writing scripts for updating them or moving them has been fairly simple. I can simply create an object, write a script for what I want it to be able to do, and attach that script to the object.

From this, it became apparent that Unity can follow a somewhat-loose MVC design pattern, where the Controller components are the scripts written for objects or handling user input, and the View component contains each of the actual GameObjects they are associated with. The Model component is less clearly defined, but I intend to use it to define my database structure as well as my save files. Additionally, the way Unity is structured makes object-oriented programming somewhat different to how it is often done in other applications. I have been able to implement some of the principles so far, particularly encapsulation, however I have noticed that my Class Diagram in my Requirements document is very flat, since most of the behaviour I have planned or already implemented simply extends Unity’s own MonoBehaviour class. However, once I get further into development, it is likely that I will encounter situations where implementation of other OOP principles are more readily apparent.

One thing that this made very clear was that the commenting of my code would be vital for its understandability. Because Unity has an extensive existing codebase, it is important that whenever I use functionality that I have not written myself, I need to explain what the functionality I am using does. For example, when I used the method GetComponent<Rigidbody>() in my PlayerMovement script, I needed to make it clear what that means, so I wrote the following series of comments:

I often have a tendency to comment almost line-by-line, to the point where it can negatively impact readability, so this is something I will need to work on throughout this project. I aim to write comments that explain functionality I write in a more general sense, while writing with more detail when I use code that I have not written myself.