Unity First-Person Platformer: Feasibility Report

Ted Singleton

# Introduction

The intention of this report is to examine the feasibility of my planned project over the next 10 weeks. I plan to create a first-person platforming video game using the Unity Engine, with a heavy focus on scripting in C# and using other tools to enhance the game experience.

# Background

During my final year at university, my final project was to create a 2D platformer game using Python and the Pygame library. This project saw me create this game from the ground up, writing the engine for the game, handling player input and character physics, as well as basic visual output. During this project, I realized that many of my original plans were out of scope due to a lack of options presented by my choice to create the engine for the game from the ground up, combined with my inexperience with the tools I had chosen. While this project was a strong showcase of my programming ability, it ultimately created a technical demo rather than an actual game that a customer would want to purchase and play. My primary goal with this project is to create a portion of a full game, with the intent of providing it as a demo to potential customers.

My research at the time of creating this project showed that there were several similar games on the market, leading me to the conclusion that a simple 2D platformer would struggle to stand out against competitors. Because of this, I decided to develop my game around a single core mechanic – the use of a grappling hook to grant players more freeing movement. My implementation of this felt somewhat unrefined, and I intend to make a game with a more satisfying flow using the Unity Engine and a first-person perspective.

# Outline

Through the course of this project, I plan to develop a full demo for a first-person platformer game. This game will, like my original 2D platformer project, include the use of a grappling hook for player movement, but in a 3D space. I also plan to implement other movement modes like mid-air jumps and sliding to allow players free, flowing movement. Following my current plan, the demo will be launched from a main menu, which will include options for level selection and input settings like camera sensitivity and key bindings. Each level will feature the same main goal – reaching the goal zone at the end of the level. Some levels will include sub-objectives like collectible items or doors that need to be opened using buttons in the level. Upon completion of a level, the player will be given a grade based on their time, and their fastest time will be recorded in a persistent database, with the intention of allowing players to compare times on a global leaderboard.

I am choosing to develop a demo rather than a full game because this will allow me to focus more on the core gameplay and surrounding systems during the time that I have for the completion of this project. If I were to commit to development of a full game, I would need to spend a significant portion of the time developing individual levels to reach a full game’s length, which could be better spent refining the gameplay itself. In this same vein, I am placing visual presentation and audio as a lower priority.

# Methodology

Throughout the requirements gathering and design phases of development, I investigated multiple approaches to how I could develop the game. This process began with researching multiple engines in which to develop my game. Much of this research focused on the Godot Engine, an open-source alternative to many popular commercial engines for game development. I learned that while Godot primarily uses its own scripting language, it does also support use of C# which would make it suitable for this project. However, as outlined in the Project Design document, I ultimately decided to use the Unity Engine, for its native C# support, access to an extensive Asset Store, and compatibility with a range of target platforms.

Another way I investigated the project’s feasibility was through the creation of prototypes. These are also described in detail in the Project Design document, and I will also describe the process here. My prototyping approach started with an empty environment, with simply a player character and a platform for them to stand on. As I developed the functionality of player movement, I added extra objects in the environment to test each piece of functionality, until I had a full test space where I could easily test any of the player’s core movement modes. This allowed me to assess the feasibility of movement features and changes to said features in a consistent space, without having to interact with other game systems to navigate between real levels. I also prototyped menus using Microsoft PowerPoint, allowing me to get a feel for how my planned menus flowed between each other. Ultimately, I decided to make several changes compared to these initial plans, particularly on the Level Complete screen, due to concerns over scaling and the display becoming cluttered.

During development of the systems that persisted or were called by multiple game levels, much of the development was done with a UI-first approach. I would develop the UI visually, ensuring that it was not too cluttered or confusing, and then wrote scripts that would give each element functionality. This can best be seen in the development of the Leaderboard. I began with the creation of the leaderboard table, converting a single entry into a Prefab in Unity, allowing me to instantiate more such entries through code, and tested the UI with dummy information. Then, when it came to writing scripts for this functionality, it was quite simple to understand the database queries I would need to run, as well as how to populate the leaderboard table with modified instances of the entry Prefab for each of the top 100 times on the selected level.

One issue I faced during the development of my project was one of insufficient technical knowledge on certain aspects of the Unity Engine. At an early stage, I decided to use the PlayerInput component as it would simplify the development process of handling the player’s inputs. In code, each relevant script could subscribe to specific inputs by their action rather than a specific key, allowing for flexibility in how inputs were handled, which would allow me to implement key rebinding. This led to some issues when attempting to implement input rebinding, which I will cover in the next section.

# Overview of Alternatives

During the attempted implementation of input rebinding, I realised that a proper implementation would require significant changes to how I handled PlayerInput’s input events in each script and deemed it out of scope in the time I had remaining. This was an unfortunate decision to have to make, but I believed that fundamentally reworking input handling would be a risky decision to make at such a late stage in development. Based on knowledge I gained from developments made after cutting this feature, I would make use of a singleton PlayerInput script attached to a relevant GameObject, which would persist between scenes in Unity through the use of Unity’s DontDestroyOnLoad() method. This would allow me to rebind actions in a menu, and those new binds would persist between scenes. I could then also make use of a local save to ensure that these binds persist between gameplay sessions.

There were also multiple other instances where I had multiple ideas for ways to implement features. For example, the late addition of Doors required multiple iterations to ensure that they worked in a way that felt right as a player. Initially, I only implemented timed doors, that would open on a button’s activation and close after a set duration. I then decided to implement toggle doors, which would open on activation, and remain open until a second activation. This process required me to investigate how asynchronous methods work in Unity, through methods that return an IEnumerator. I originally attempted to implement every type of door opening and closing in the Door script itself, before I realised that this was quite limiting in terms of extensibility for new types of buttons, for example ones that swap the open status of multiple doors. I decided to instead keep the functionality of simply opening and closing the door within the Door class and move the timer or toggle functionality over to the respective types of buttons.

There were also multiple approaches made in terms of visual design for many aspects of the game. Early in development, I decided that visual design was not a priority for my game, as the purpose of this project was to display my ability to create an application in C#. However, it did become necessary at a later stage to provide some kind of visual feedback to the player when it came to level design. Ultimately, I decided to reflect the purpose of each object in the game using a colour. For example, the goal zone of each level is green, the walls are grey, and the pit is near-black. Objects the player can interact with started as simple platforms in white, and I decided to keep grappleable objects white as well. The intent there was to keep visual consistency with other objects the player will have learned they can walk on, while keeping the grappleable object out of reach to indicate that the player can use it to move. Doors are also represented with colours that match their respective buttons, and buttons turn red while a door is moving, to indicate that it is inactive. I made these decisions with full knowledge that this is not a perfect solution to readability for my game, as it does not consider accessibility for colourblind people, but with the tools and knowledge I had during development, I felt that this was a good approach, while not going too far out of my original scope.

During development of the database functionality of my game, I had already decided on an approach for each table, and mostly stuck to it. I realised I needed to include an ID for each entry in the PlayerTimes table, but other than that my database design functioned properly. Additionally, when the game is run in Unity’s Editor, database queries function properly, directly interacting with the database. However, when running the game in a built state, no database queries are executed. The same goes for the player’s progress save file. I have attempted to investigate these issues but have not been able to come up with a solution in the time I had after discovering them.

# Conclusion

In summary, my research has shown that the gameplay elements and systems I plan to implement will be feasible in the time frame I have. I have already put together a small prototype for moving around a game world in first-person and will be able to expand on this prototype until I have developed a full suite of gameplay options. Research has also shown that Unity projects can be created with database support for leaderboards, and many of the specific gameplay elements I have planned have been implemented by others in their own products.

The implementation of my game proved the conclusion from my research to be accurate, as I have successfully developed a demo for my game making use of code from the prototype. My game includes menus that tie together each element of the UI in a reasonable way, and my game has database support. Issues have arisen when attempting to run my game as a built product, however I can prove that the functionality does work in the Editor, as well as the ability to read a save file. If these issues had been found earlier in development, I may have had time to fix them, but at this stage there is not sufficient time.

# Recommendation

Writing this section retrospectively provides a comprehensive perspective on the Unity First-Person Platformer project. The project successfully met nearly all of its original requirements and introduced additional features, such as interactable doors, demonstrating its versatility and adaptability. Performance metrics indicate satisfactory results, although broader hardware testing was constrained by external factors.

Reflecting on the project, several valuable lessons emerged. Testing in a built version of the game, rather than solely relying on the Unity Editor, would have revealed nuanced issues earlier in development. Additionally, adopting a more structured approach to persistent systems, such as implementing a true singleton PlayerInput class, would enhance code maintainability and extensibility.

For future endeavours, it is recommended to incorporate these reflections into the development process. Emphasizing thorough testing and refining implementation processes can mitigate challenges and streamline project progression. While this project was deemed feasible from the outset, the complexity of certain implementations and the importance of comprehensive testing highlight the need for a careful approach to project execution.

In conclusion, this Unity First-Person Platformer project was successful in its execution and highlights areas for growth in future endeavours. By remembering lessons learned and maintaining a commitment to thorough development and testing practices, future projects can build upon this foundation of success.