# The Development of a 3D Game with a Time Loop Mechanic

**Jamie Fergus**
**ID: 190018054**
**AC40001 Honours Project**
**BSc (Hons) Computing Science**
**University of Dundee, 2023**
**Supervisor: Dr Craig Ramsey**

***Abstract -*** *The aim of this project was to create a unique 3D game, utilising a time loop mechanic, and to understand what goes into game development. Inspiration was taken from other game featuring time loop mechanics e.g., Outer Wilds, The Legend of Zelda, and games such as Superhot and Neon White.*

*The game was developed using the Unity game engine. The final game is a first-person puzzle platformer with 24 levels to complete. Where a player can progressively learn from their mistakes and change their approach accordingly. Levels are split into two stages. The aim of each level is to reach a finish area. However, if the player dies, they will be reset back to the start of the loop. User testing revealed that this was successful, but further improvements could be made.*

## 1 Introduction

The primary aim of this project was to create a 3D game. This could involve either a low-level library like OpenGL to create realistic 3D lighting, shading, and other effects, or to utilise a higher-level platform/ game engine if preferred. The student chose to focus on the latter. A secondary aim was for the student to gain a better understanding of game development and how a 3D game is made. It was agreed with the project advisor that the specific objectives of the work were adaptable to the interests of the student. The project provided the student with an open problem to investigate and explore. The student took inspiration from a range of sources. Some from their own experiences and knowledge of games and the games industry, from different games and genres. But also took some inspiration from other sources like film.

The video games industry is a global industry now worth billions, with some titles selling billions of copies. With a significant number of players, of many ages across the world. The industry has also seen tremendous growth over the past decade, with consoles in many homes being a household item, the rise of mobile games industry and of professional esports gaming. With games even making use of emerging technologies like Virtual Reality, Augmented Reality and many more. In short there is no shortage of people who play and care about video games and the industry [1].

The student wanted to try to add something new to the field. A new type of game or experience. Perhaps a combination of multiple ideas and genres from a range of games. The student ultimately settled on a novel game utilising a time loop mechanic. Since the student has always been fascinated by this idea in the entertainment industry. In films like Groundhog Day (1993) and games like Outer Wilds (2019). In both cases feature some form of time loop where the main character relives the same time loop with events playing out in the same or at least predictable ways.

This idea's relevance in the larger scope of the games industry was always an interest to the student. And believed that there were many avenues of design and ideas that could be explored. Especially across genres and in utilising different mechanics from multiple different games or genres. Inspiration was also taken from other non-time loop games, such as 3D platformers, time based, puzzle-based and rhythm-based games.

The student settled on the idea of a 3D platformer/puzzle hybrid, with a time loop mechanic. With multiple levels of increasing difficulty, as well as different challenges and themes to them. The student also wanted to the final version of the game to have many finished polished features including menus, sound, options and no major bugs and glitches.

Ultimately the goal is to explore this concept, see how it works in practice. Have the gameplay go through user testing. Where players can play a build of the game and provide their thoughts on the game in general and the time loop mechanic.

In the remainder of this report, is structured as follows. In Section 2, the background context to the project is explored. Section 3 will then go on to describe the design choices that were made. Section 4 the requirements for the project are presented. Section 5 covers the implementation of the code.

In Section 6 examines how testing and user feedback are evaluated. Section 7 provides a description of build of the game. Section 8 an appraisal of the project. Section 9 Summary and conclusion from the project. Section 10 any future work that could be done.

## 2 Background

### 2.1 Inspirations and Similar work

One of the original inspirations of this game's core idea is the 1993 movie Groundhog Day, starring Bill Murray. A story about a self-absorbed, egocentric man who is stuck reliving the same day over and over, until he gets it right [2].

"Well, what if there is no tomorrow?
There wasn't one today!"
- Phil Connors, Groundhog Day

Another film utilising a similar time loop effect but in a different context Edge of Tomorrow (2014) starring Tom Cruise, with the slogan "Live. Die. Repeat.". About a man stuck reliving the same day fighting off an alien invasion, until he finds a way to win [3].

There are also many games that utilise time loop mechanics such as The Legend of Zelda: Majora's Mask (2000) by Nintendo, Outer Wilds (2019) by Mobius Digital [4], 12 Minutes (2021) by Luis Antonio [5], The Forgotten City (2021) by Modern Storyteller [6]. In many of these games the time loop mechanic is necessary in solving some sort of mystery or puzzle.

For example, in the Bafta Award winning game: Outer Wilds (2019) by Mobius Digital. Which is set in a solar system with strange and bizarre planets. The player takes the role of a newly recruited alien astronaut, who is sent out to explore the solar system. However, during the player's first adventure into the cosmos, the sun suddenly goes supernova, wiping out the player along with the entire solar system. The player then wakes up back at the start of their adventure, before they set foot in their spaceship, while everyone else is oblivious to the impending doom. The player then must use these 22-minute time loops to explore the planets and find out clues to solve the mystery of what's going on and figure out a way to save the solar system [4].

Figure 1 - screenshot of Outer Wilds

Other non-time loop games also served as an influence for game idea. SUPERHOT (2016) by SUPERHOT Team, a game where time moves faster, when the faster the player moves [7]. As well Neon White (2022) a game which takes the core ideas of speedrunning an makes a game around it [7]. Both these games also have a similar and simple minimalist art style.

Figure 2 - Screenshot of SUPREHOT

1st person 3D platformers like Neon White but also Mirrors Edge (2008) by also served as inspiration that would influence the and help shape the movement of the game. Research into common movement systems often contain systems like coyote timers and jump buffers. Which are probably not obvious to the player but if removed they would probably notice their absence.

Speedrunning is an interesting and unique way to experience a game. Where the goal is to complete the full game, or part of a game, in the fastest possible time. This can often be achieved by abusing bugs or exploits, as well as by following specific route through the game. These tend to be the currently known fastest routes that a player can be expected to pull off.

Speedrunners will often playthrough a game multiple times, sometimes restarting a run if they fail a part that costs them significant time. Or they will continue the playthrough. The end up playing and replaying a stage or the full game countless times. Over multiple playthroughs, they end up perfecting the techniques, learning from their mistakes, making small time saves. And overtime hopefully improving their overall time.

Random Number Generation or simply RNG is a staple of many games. From procedural generation of a world like Minecraft, or randomly selecting which card the player will draw next in collectable card game like Hearthstone. One benefit of randomness in games is it often adds replay ability. Since for example if a player was to replay a level, different events or scenarios might play out even if the player attempted to do the same thing. Resulting in a unique experience.

Later in development, a ghost mechanic was developed which the player would use to help them time there moves. This mechanic was influenced by similar mechanics that can be seen in racing games. Which is often used to visualise where the player was in a previous attempt at a race. Another similar mechanic was the replay mechanic in Super Meat Boy (2010) by Team Meat. On successful completion of a level, a replay of your successful attempt along with every failed attempt would play out.

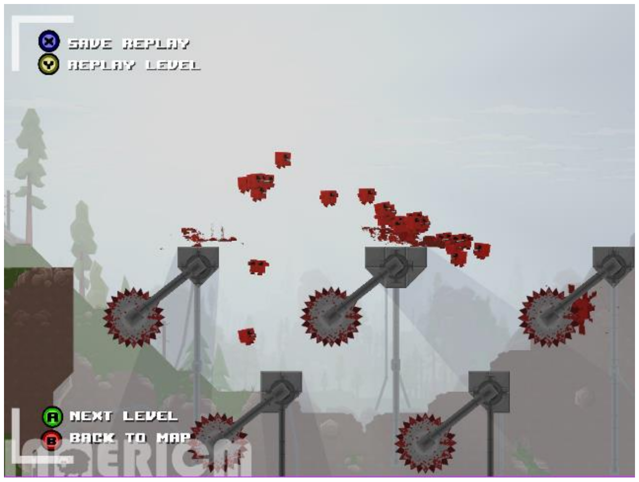


Figure 3 - Screenshot of the replay mechanic from Super Meat Boy

The different time loop games and film inspired the student to explore a time loop mechanic in their game. Although the student wished to do something different than previous titles. Instead focusing on smaller loops rather than one big loop to solve some big puzzle or mystery. Instead, it was to be utilized in a level-based approach like Neon White or SUPERHOT. Where a player would have to learn the level and its patterns in-order to beat it. Replaying the same level multiple times until they can finally beat it, to a similar extent to how a speed runner would play through a game. 3D platforming elements were also inspired from 1st person 3d platformers like Neon White and Mirror Edge. And the ghost mechanic was inspired from racing games and the replay mechanic in Super Meat Boy.

## 2.2 Choosing a Game Engine

To prepare for this project, it was essential to conduct research on the available game engines. It was crucial to decide about the game engine early in the project as it would have a significant impact on the design of the game's features and mechanics going forward.

The choice of a whether to use a game engine was decided very earlier on. The student decided that it would be more appropriate to use and engine as it would ultimately allow for a game of a much bigger scope and polish to be created. If the no engine was used the final product would be much simpler and would require implementation of many systems like graphics, physics, etc that game engines usually provide.

The main two engines that were considered for the project was the Unity [9] and Unreal engine [10]. Both see widespread use with Unreal being the more popular of the two within the industry, with many big well-known games such as Fortnite (2017) developed by Epic Games, Star Wars Jedi: Fallen Order (2019) developed by Respawn Entertainment and published by EA, and Sea of Thieves (2018) developed by Rare and published by Microsoft.

But also seeing use in the indie scene and smaller games studios with games like Stray (2022) by BlueTwelve Studio, Abzu (2016) by Giant Squid, Satisfactory (2019) by Coffee Stain Studios.

Then there was Unity which sees a lot of games from the indie scene. Including Hollow Knight (2017) by Team Cherry, Subnautica (2018) by Unknown Worlds Entertainment, GTFO (2019) by 10 Chambers. Neon White and SUPERHOT were also both made in Unity.

Overall Unreal seems to see more use by bigger studios or games looking to have very nice graphics. Whereas Unity sees a lot of use from hobbyists and indie developers. The Unreal Engine mainly uses C++ whereas Unity uses C# for its programming language.

The student had no prior experience in Unreal however they had some experience with Unity. The student also had slightly more experience with the C# than C++. And due to fact that graphics were not important to the goal of this project. The student decided it would make more sense to go with the Unity Engine.

## 2.3 Unity Game Engine

Figure 4 - Unity logo.

Even though the student had prior experience with this engine, their overall knowledge was still lacking. The student early in development decided to conduct research into Unity in general. To learn more about the engine and its features. Through participating in tutorials on the Unity Learn platform [11]. But by also watching videos online through sites like YouTube. The students had also learned from their previous Unity project.

Unity is a popular cross-platform game engine and development tool used to create both 2D and 3D games, simulations, and interactive experiences. Unity sees use by both small indie developers and large game studios to create games for desktop, mobile, and console platforms. With Unity, developers have access to a wide range of tools and resources including an asset store [12], a robust scripting API, and a large community of developers sharing tips and techniques [9]. Unity has become a go-to engine for game developers of all levels due to its versatility, accessibility, and extensive support.

The Unity IDE offers many tools to help develop a game. Including tools such as a physics engine, asset store, an inspector to tweak the properties of an object. Can create multiple scenes which are separate instances or parts of a project, often used for different levels or parts of a game. Game objects are the building blocks which represent an entity in the game. These can be used for the geometry of level, the player character, etc. Can render the project in real time, which allows a developer to instantly evaluate their most recent changes. Can create C# scripts, written in the C# programming language, to get much more control over the project. Which can be used to implement new features and game mechanics [9].

In a previous project, the student had run into some issues with saving and loading game data. So, the student conducted research into a new way to save data. The student found that Unity had a system called "PlayerPrefs". This simply stands for player preferences, and it is a very convenient way to store integer, float, or strings data with key value pairs, to save data such as player settings. These could then be fetched when the game is launched to get the previous session values loaded into the new session [13].

Prefabs were another unutilized feature. Prefabs allow a user to create their own template game object then save them as a template. Then they can simply use the created prefab like a regular game object. Prefabs of prefabs can also be created. One of the benefits of using a prefab is that if the original template is changed it gets applied to every single object created from that prefab. Allowing a user to update every instance of an object across multiple scenes. However, a game object created from a prefab can also have its properties overwritten, allowing a user to override the original template. If the template is changed the overridden settings would not change [14].

Scriptable objects were also discovered through this research. These are simply a way to store data, which can get saved as an asset in the project. They are created with a C# script then can accessed by an object or other C# scripts by referencing an instance of the scriptable object. These would be used to save data between loops of the game [15].

Unity by default way to handle input [16] is simple and it is harder to scale to incorporate rebinding and handle multiple input methods. Which is why research into a better system was done. The student discovered that a new input package can be installed through the package manager. Although it is more complicated and requires more work to set up. It is much more scalable and allows for much easier handling of multiple input devices and would make it much easier to implement a system to rebinding inputs [17].

Cinemachine is another package which allows for much more complex control and provides more in-depth options over a camera than Unity offers by default. This works by creating a virtual camera which is used to set up or customize any properties that a user might want to code or configure in the Unity editor. Then these get applied to the default Unity camera by attaching a Cinemachine component that gets to it [18].

Unity's ECS framework was investigated by the student. However, it was decided that it would not be used since at the beginning of the project, the framework was still in beta. The reason this was researched was because the Entity Component Systems use in the gaming industry has been growing over the years. And even in Unity it can see performance improvement when dealing with very large amounts of objects. Even though there would be a reasonable amount of game objects on screen at one time, there was not enough to really warrant the use of the ECS system. The student also concluded that it would be safer not to use it since the resources for the framework were lacking compared to the standard approach more object-oriented approach [19].

## 2.4 Other tools

Visual Studio was chosen as the text editor/IDE to use for all the coding for the project. This was chosen as it was the preferred text editor that the student used when coding in C#. GitHub, the version control tool, was used to manage the project and to restore the previous version if necessary.

### 2.5 Agile

An Agile like approach was taken for the project where the student could work on a single feature in a single sprint. Where the feature would be set a certain time to complete, designed, then implemented, then tested locally and finally fixing any problems before moving onto the next feature.

At the beginning of a Sprint a feature or section of the game would get chosen to be worked on. Then a proposed amount of time would be set to completely implement this feature. Once complete the next feature would then be started.

## 3 Specification

The main features of the project are:

- Within each level there is a time loop, that is restarted every time you die.
- The random elements remain the same each time the time loop restarts.
- When the level is restarted or replayed, the random elements are regenerated using a random seed.
- There are multiple levels.
- There will be a UI for level selection, allowing the user to go back and replay levels.
- The game will be in a first-person perspective.
- Basic character movement including jump.
- Each level will contain obstacles/enemy AI that the player will have to overcome.

For more requirements see requirement specification in the appendices.

## 4 Design

### 4.1 Game Engine

As discussed in Section 2 previously, Unity was selected as the engine that would be used to develop the game. Unity was chosen as the engine that would be used to develop the game. This main reason being the student had some prior experience with the engine in a previous project developing a smaller 2D game. Lot of lessons were learned from that project such as unutilised features like prefabs, “PlayerPrefs”, input actions manager. However, there were many other reasons.

### 4.2 Visuals

The student decided that graphics were not important to the project. Since the student was more interested in the mechanical and gameplay side of the project. So, it was decided the environment of the levels would be left untextured and would just be made from standard objects provided in unity. Such as cubes, spheres, capsules. The colour of the levels would be set to white. Which would give the game a minimalistic look, reminiscent of SUPERHOT or Neon White. Games like Thomas was Alone (2012) have proven that a game does not need good graphics to be engaging.

### 4.3 UI

#### 4.3.1 Menus

Designs for the user interfaces of the many menus of the game would be kept quite simple and would follow general trends that are seen in the game games industry. For the games menus the game will consist of a many menus including a main menu, level select menu, options menu, pause menu and a menu for end of level screen.

The main menu would consist of title and multiple buttons for navigating to the other necessary menus and one for closing the game. The level select menu would consist of buttons for each level and a back button. The options menu would consist of settings for including sliders for volume, camera field of view, mouse sensitivity and a toggle inverting sensitivity and options to rebind keys. Later in development the rebinding section of the options menu, was decided that it needed its own menu.

Pause menu would follow standard game norms with similar layout to main menu with buttons for resuming the game, restarting the level, navigating to the options menu, and quitting to main menu. Then there would be a final menu that would pop up on level completion. Which would provide the player with buttons for the next level and returning to menu.

Each menu’s elements will be overall centred vertically where appropriate, with space to either side. With a consistent theme across all menus and elements. A grayscale colour scheme was chosen to keep it simple and easier to create. This should ensure that it will be completely visible to users with colour-blindness. Since there are no colours that could get confused.

#### 4.3.2 Heads Up Display

The game heads up display was to be as minimalist as possible practically no HUD. The only thing the HUD will contain is a timer, positioned at the top of the screen, so a player can have an easier time timing there moves and to help them learn the timings of the events in a level. The reason for this as there is not many standard HUD elements for information like a crosshair, health, score, mini-map etc, are not needed in this game. For example, health, as it has been decided that a player to die will die in one hit to any incoming damage. Since the intended idea is for a player to learn the patterns of a level and perfect the level instead of simply scraping by. Minimalistic HUDs can often be more immersive for a player. This has the added benefit avoiding a bad or cluttered HUD since there is so little information to be shown to a player. It will also make it easier for a player to concentrate on the elements of the game since they'll be practically no HUD elements that could cover or obscure them.

## 4.4 Levels and Structure

The game will consist of multiple stages with multiple levels. Each stage having a particular theme or mechanic being explored. First stage is for dodging obstacles that are coming in from the sides. The second stage will be about traversing across platforms with some platforms being fake and others being real solid platforms which can be safely traversed across. Another stage for navigating across an area with a series of traps. A stage where a player would have to navigating across or climb up through multiple moving platforms, to reach the end.

Other less flesh out ideas were also explored, ideas including avoiding enemies. However, the student never found the time to really research and design these features.

The geometry of the levels would be built using various game objects shaped to make platforms, walls, etc for the various levels.

In these levels the player goal is to reach a finish area. A seed will be generated which will be used to randomise the spawning of the obstacles. So that every time the player dies or restarts the level the seed will be reused, and the random elements will stay the same.

For the obstacle levels a Unity empty game object to indicate the position of where an obstacle will spawn will be used. Then a C# script will be used to randomly spawn the obstacles.

For the fake platform levels, the platforms themselves will be a standard cube game object shaped accordingly. It will have a Collider which will be turned on or off to ensure that only one platform has a Collider in each row.

## 4.5 Player Object

An object for the player will be created built using unity components. Containing a camera and the player object. With C# scripts to move and rotate the player.

## 4.6 Movement

A movement system would be developed which would include movement for moving forward, backward, left, and right. Then also the ability to jump. The movement would be tweaked to feel responsive and quite quick to accelerate to allow the player to suddenly stop and start so that they could avoid obstacle sufficiently. But not too quick to accelerate and decelerate that it would feel unnatural. See section 5.5.2.7 for more in-depth details on the movement.

For the player to jump, the system would first need to check if the player is touching the ground. If they were then they would jump. However, this is quite a simple solution. A better solution is to account for circumstance when a player input is either slightly too early or late. For example, if a player was falling and about to touch the ground but pressed the jump button a fraction of a second too early. Or if a player pressed the jump input just after they had walked off the edge of a platform.

Even though in these circumstances that player is not technically in a grounded state. To them they might think that they were and felt cheated that the character did not jump. So, to make the game feel more responsive and fairer, a coyote timer and jump buffer would be implemented into the game.

So instead of checking if the player is grounded, it checks if the player was grounded within a short period of time. This is often called a coyote timer in the industry. In the case when a player presses the input too early before they touch the ground. The game well waits a short period of time to see if a granted state is achieved. If it does within the time, then the player will jump.  This is called a jump buffer. See the flow chart to see a visualisation of this in the appendices.

To get the player to move or jump the unity physics engine will be used to add a force to the player. See section 5.5.2.8

## 4.7 Camera

The player camera will get attached to the player object. Giving it first person perspective, then using player mouse input will be used rotate the camera and player object accordingly.

## 4.8 Player Input

Al player input will be retrieved using the new input actions manager. Creating control schemes for both the keyboard and mouse and controller/gamepad. Then an input manager script will be created to manage this input. Which will be used by any script that needs input player input. See section 5.2.1 for more details.

## 4.9 Ghost mechanic

For the ghost mechanic three things will have to be recorded when the player is playing. The current time, their position, and their rotation. This should be recorded at certain intervals up until the player dies and the ask current loop is reset. Then when a player starts the new loop, we should see a ghost of their previous attempt with them. To store this a suitable data structure will be chosen, to store all the ghost data and then another data structure to store all the ghosts containing the data. See section 5.4.1 for more details.

Then while a player is playing through of the level if there are any ghosts that have been stored. Then their movement would need to be played. The student decided there would only be a maximum of 10 ghosts at one time. For this a ghost object prefab would be created then a script would spawn this object for each ghost. Then move these ghosts through the previous positions and rotations that are stored at the correct time. Interpolating the positions/rotations if required. See section 5.4.2 for more details.

### 4.10 Options

Options for volume, mouse sensitivity, field of view, mouse inversion and key rebinding. These settings will be saved by using "PlayerPrefs" so that on game launch they will be loaded in and be the same as what the player last set them to. They will also be applied at runtime to the appropriate object or script. See section 5.5.3 for more details.

### 4.11 Audio

Audio clips for when the player walks/runs, initiating a jump and landing back on the ground will be added. These audio clips will be sourced from the asset store. See section 5.6.1 for more details.

### 4.12 Plan for Development

The plan and rough order of execution for development and testing of the game can be summarised in the following way:

- Development and testing of the movement system and camera system
- Implementation of the different level mechanics and stages
    - Obstacle levels
    - Fake platform levels
    - Floor trap levels
    - Moving platform levels
- Ghost Mechanic
- UI
- Options
- Audio
- Enemy ai
- More levels

## 5 Implementation and testing

In this section, the implementation stages of the project shall be described, covering the following aspects: the initial prototype, player input, movement and camera, implementation of the levels, ghost mechanic, user interface and in-game settings, and then any final touches.

### 5.1 Prototype

During the December-January break the development of a basic prototype of the game was started. This prototype was created to validate the basic design and ideas of the game to see if they would work in practice. The overall prototype would be very simple so that it could be created in a small amount of time.

The main feature that was to be developed and tested was the random spawning of obstacles that a player would need to avoid. As well as resetting the level when hit, ensuring that the Obstacle spawned the same times each loop.

The prototype consisted of a single scene with large flat game object for the ground, a static camera, a player game object. The player game object was just a simple capsule with a simple movement script attached to it, which only allowed forwards, backwards and side to side movement. As well as a player collision script which would restart the scene if the players collider came into contact with an obstacle's collider. A prefab of a simple obstacle was created, a simple 2x2 cube with a collider and script attached to give it a velocity on the x-axis.

Another script was needed to control the spawning of objects. This script worked by first initialising a list of spawn positions, each with the same x and y coordinate but different z coordinates to space them out. Then the randomisation needed to be seeded this was achieve with the method "Random.InitState(int seed)". This method Initializes the random number generator state with a seed parameter. This was required so that every time the scene is loaded the random generators would output the same values. The seed for testing purposes was not randomised and manually set, to ensure that it was working as expected. Then for each spawn position a coroutine was started which would start an endless loop, that contained code spawn a new obstacle after a random amount of time. This script was then attached to the camera object in the scene, so it would run.

After a bit of playtesting this prototype, although simple it proved that the original concept does work in practice. Through dying and resetting the loop, a player could learn the patterns and make it past the obstacles.

### 5.2 Player Movement and Camera Control

After the prototype was developed, the plan shifted to developing a full game. The student decided the most important thing to start with was to fully implement the movement system.

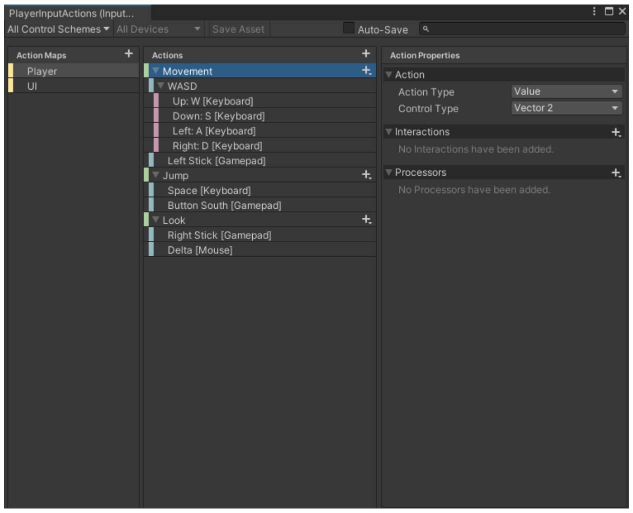
To achieve this the prototype was altered the spawn obstacle script was removed and more game objects and platforms were placed in the scene to create an environment to test the movement system.

### 5.2.1 Player Input

The prototype movement system just used Unity default input system to handle player input. This system works by using functions, like the "Input.GetKey()" and "Input.GetAxis()" methods for keys and mouse input respectively. This makes it quick and simple to set up and why it was chosen for the prototype. However, this system is not easily expandable and makes it difficult to account for multiple input devices or rebinding of keys and controls. So instead, the full game movement system would make use of Unity's new input system.

#### 5.2.1.1 The New Input System

This package does not come installed as default, so it needs to be added and download in the package manager. To us the new input system, an input action asset needs to be created. This is a Unity asset which is used to define all the input actions and there corresponding bindings. Within the input actions window, the window is split into three groups: Action Maps, Actions and Action Properties. See Figure 5 below.



PlayerInputActions (Input...
All Control Schemes ▾ | All Devices ▾ | Save Asset | Auto-Save

Action Maps
- Player
- UI

Actions
- Movement
  - WASD
    - Up: W [Keyboard]
    - Down: S [Keyboard]
    - Left: A [Keyboard]
    - Right: D [Keyboard]
  - Left Stick [Gamepad]
- Jump
  - Space [Keyboard]
  - Button South [Gamepad]
- Look
  - Right Stick [Gamepad]
  - Delta [Mouse]

Action Properties

Action

| | |
|---|---|
| Action Type | Value |
| Control Type | Vector 2 |

Interactions

No Interactions have been added.

Processors

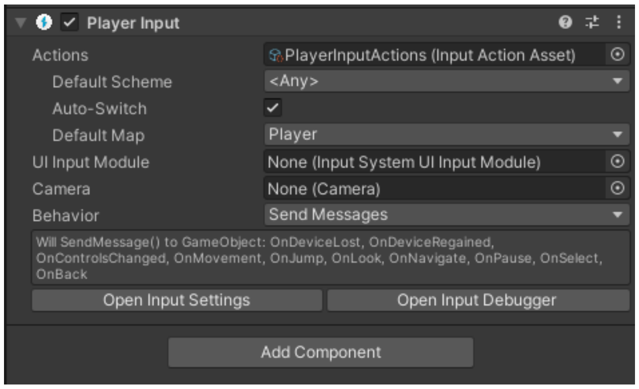
No Processors have been added.

Figure 5 - Input Actions Window in Unity

Action Maps are simply containers used to organise your different actions. Actions are the things a player can do as a result of input, for example movement, jumping, moving the camera, etc. Individual actions can be given multiple bindings, and from different input devises, like a mouse, keyboard, gamepad etc. Action Properties are where the properties for a specific action or binding is configured. Control Schemes can also be set up to organise your bindings to specific input devices.

The student used the Input Actions system to create a Player Action map that contained Movement, Jump and Look (camera look) actions, as seen under the actions tab in figure 5. Then set up bindings for two control schemes Mouse and Keyboard, and Gamepad, so that players can choose to play with keyboard or with a controller. Dead zones where also set up for the gamepad bindings for Movement and Look, to avoid stick drift if applicable.

Once the input action Asset has been created and set up it can be integrated into the rest of the project in a few ways. Two different ways it can be used is that the input action asset can be used to generate a C# script with all the actions and settings that were configured, then simply referencing that script in the corresponding scripts that it is needed. It can also be attached to a game object by using the Player Input component, as seen below in Figure 6. Then within a script using "GetComponent()" method to access actions.



Player Input

| | |
|---|---|
| Actions | PlayerInputActions (Input Action Asset) |
| Default Scheme | <Any> |
| Auto-Switch | ✓ |
| Default Map | Player |
| UI Input Module | None (Input System UI Input Module) |
| Camera | None (Camera) |
| Behavior | Send Messages |

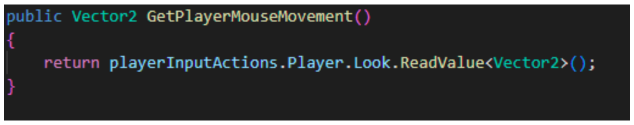
Will SendMessage() to GameObject: OnDeviceLost, OnDeviceRegained, OnControlsChanged, OnMovement, OnJump, OnLook, OnNavigate, OnPause, OnSelect, OnBack

Figure 6 - Player Input Component

The student used the first method to use and access the input actions, however later needed to change their code, and use the other method to allow for rebinding of controls. As the generated C# script could not be regenerated at runtime, whereas with the Player Input component the Input Action Asset could be accessed directly and changed at runtime. Examples of these scripts are discussed below.

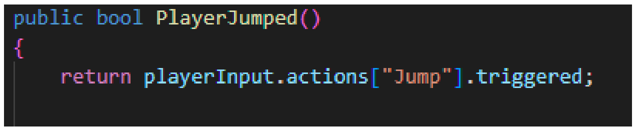
#### 5.2.1.2 The Input Manager script

These actions can be referenced in the code to return the value of an action like player movement or look, by using the "ReadValue()" method, as seen in figure 7, or when a button is pressed, i.e., jump action, with the triggered property, as seen in figure 8.



```
public Vector2 GetPlayerMouseMovement()
{
    return playerInputActions.Player.Look.ReadValue<Vector2>();
}
```

Figure 7 - code snippet from the Input Manager script returns the value from the mouse look input.

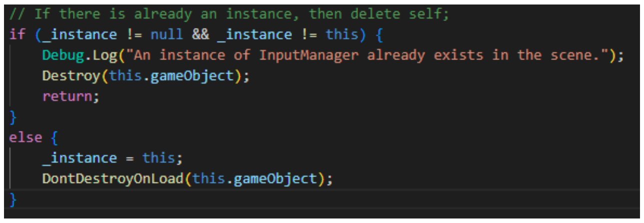


```
public bool PlayerJumped()
{
    return playerInput.actions["Jump"].triggered;
```

Figure 8 - code snippet from the Input Manager script that returns true when the jump input is pressed.

Instead of placing this code within the player controller a separate input manager C# script was created. This script would act a middleman between the Input Action asset and the player controller or any other script that would need player input. This was done to separate input logic from the code that requires it, which makes it easier to change anything with the input logic or the input actions asset without having to change any of the scripts that use it. This input manager script contains functions that return the values of the movement and look actions as well as a function that returns true when the jump action is triggered. Then any script that requires input would just need a reference to an instance of this input manager class and call one of these functions. This script was then attached to a new empty game object called input manager.

This script is also an example of the Singleton design pattern, this was done to ensure that only one instance of the input manager could exist at one time, and any scripts that uses this class would only use this instance. To make sure that one instance of this class could exist at one time in the class's "Awake()" method, there is an if statement to check if there are any other instances, and if true then it would destroy the new instance. Otherwise, if false it sets this game object as the instance as well as ensuring the object does not get destroyed on load. This is to ensure that this instance persists when changing levels. As seen in Figure 9.



```
// If there is already an instance, then delete self;
if (_instance != null && _instance != this) {
    Debug.Log("An instance of InputManager already exists in the scene.");
    Destroy(this.gameObject);
    return;
}
else {
    _instance = this;
    DontDestroyOnLoad(this.gameObject);
}
```

Figure 9 – code snippet from Input Manager Class Awake() method, showing that only one instance can exist at one time.

### 5.2.2 The Player Controller Script

A new player controller script was set up to utilise the new input system and input manager. This script is the largest and most complex piece of code in the project, and took longer to develop than the student thought it would.

Many global variables are needed for this script as the logic is split across the Update function as well as the Fixed Update function. This is because the player input needs to be taken in once per frame. However, this script also makes use of the player game object's Rigid body component, to apply forces to the player. It is recommended to do all "Rigidbody" calculations inside the "FixedUpdate()" function as physics updates are carried out immediately after the "FixedUpdate()" functions are run, so any updates can be processed directly after. So, to ensure accurate and consistent physics calculations all uses of the player's "Rigidbody" are contained within the "FixedUpdate()" function.

There are also many constant variables initialised within the global variables, such as speed, max speed, jump force, etc, as well as constants for the various timers.

Most of the script's logic is split between the "Update" and "FixedUpdate" Methods. These are built-in Unity methods. The "Update()" method is called every frame, while the "FixedUpdate()" method is frame independent and is called at a set interval, which is currently set to its default rate of 0.02 seconds. However, within these methods, are multiple instances of methods that were created by the student, which will be examined first.

#### 5.2.2.1 Is Grounded Method

To check if the player is grounded, the "IsGrounded()" method is called. This function returns a bool indicating the player object is currently in contact with ground. The student went through multiple different versions of this function when developing the player controller, as there are multiple different ways to check if a player is grounded.

The first thing that the function does is it returns false if the player is jumping. This is done by checking the value of the bool "isJumping", if it is true the function returns false. "isJumping" is a global bool variable that is set to true when the player jumps. If "isJumping" is false, then the function continues with the ground check.

One of the simplest ways to do this is to use a "Raycast", to draw a line (ray) straight down from the centre of the player object and checks if it intersects with anything by returning a Boolean. This works well most of the time, however there is a major problem with this approach. If the player moves towards an edge of a platform so that their centre is no longer over the hitbox of the platform, but the player is still on the platform. The "Raycast" would return false, as there is nothing to intersect, however the player is still grounded.

Another few methods that the student tried was by using a "BoxCast" and "SphereCast". These functions are essentially a type of "Raycast", however instead of casting a line, they cast a box and sphere along a ray. However, even though these would not run into the previous edge case problem, they ran into other problems, where they would often return the wrong result. These results are most likely caused by the fact that they will not detect colliders when they overlap the collider.

So, the student abandoned the different "Cast" methods in favour of the "CheckSphere()" method, this method is slightly similar to sphere cast, but it instead checks if there are any colliders overlapping the sphere, defined in the function. This method worked consistently and returned the desired result once the sphere radius and position were specified.

For the position of the sphere a new child object was added to the player Object. This object position was set to be slightly lower than the centre of the bottom sphere that makes up the capsule. The radius of this object was also set to be slightly smaller than the player's capsule radius. So, the resulting sphere would only slightly stick out of the bottom of the players game object and not the sides. This was to ensure that it would not collide with any walls, only game objects below the player. And to ensure a ground state could not be achieve when not close to the ground. See figures 10 and 11 for virilisation of this sphere.

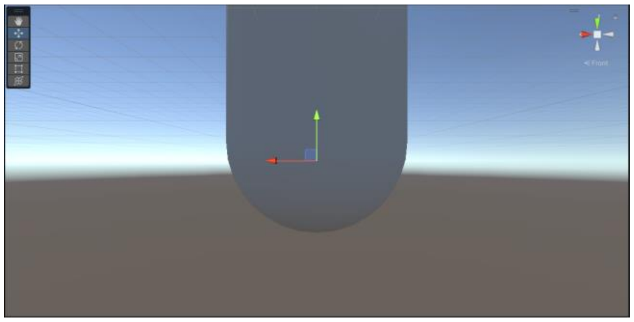


Figure 10 - Shows bottom of player object.

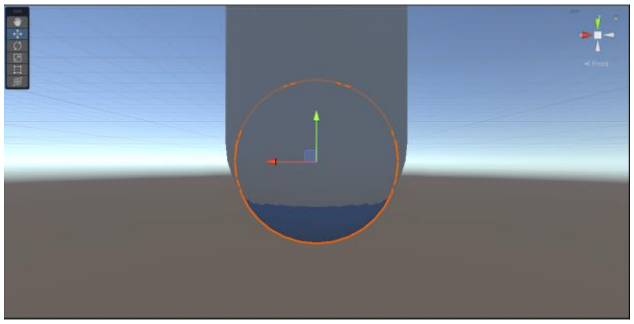


Figure 11 - Shows bottom of player object with a blue sphere object used to visualise the position of the sphere in the "CheckSphere()" method.

The "CheckSphere()" method requires a "LayerMask" input parameter. This is what layer, game objects are in. This is used to selectively ignore certain colliders. So, a ground Layer was created and added to every object that is to be treated as ground.

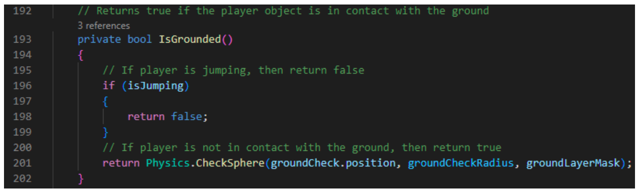

```
// Returns true if the player object is in contact with the ground
3 references
private bool IsGrounded()
{
    // If player is jumping, then return false
    if (isJumping)
    {
        return false;
    }
    // If player is not in contact with the ground, then return true
    return Physics.CheckSphere(groundCheck.position, groundCheckRadius, groundLayerMask);
}
```

Figure 12 - Final version of the "IsGrounded()" Method.

A lot of testing was done at this stage to ensure that the "IsGrounded()" function was working as intended and for its finetuning. By printing this method's value in the debug console, as well as using the "Debug.DrawRay()" method to make the different ray casts visible. A sphere object with no collider was also temporarily added to the player object to visualise the position of the sphere for the CheckSphere() function, as seen in figure 10 and 11.

### 5.2.2.2 On Slope Method

The "OnSlope()" function is similar to the "IsGrounded" function in that it returns a Boolean, true if the player is on a slope and false if not. This function first calls "IsGrounded()" to check if the player is grounded. Then a "BoxCast" is used to get information on the collider that is hit. This is done by using the out parameter "hitInfo", which returns a "RaycastHit". A "RaycastHit" is a data structure used to get information from "Raycast". In this case to get the normal of the collider that is hit. A "BoxCast" is used here instead of "CheckSphere()" since it is not a type of "Raycast", it does not have "hitInfo" parameter. The "hitInfo" is saved as a global variable and is then used to check if it's normal is in the up direction. If it is not, then the function returns true.

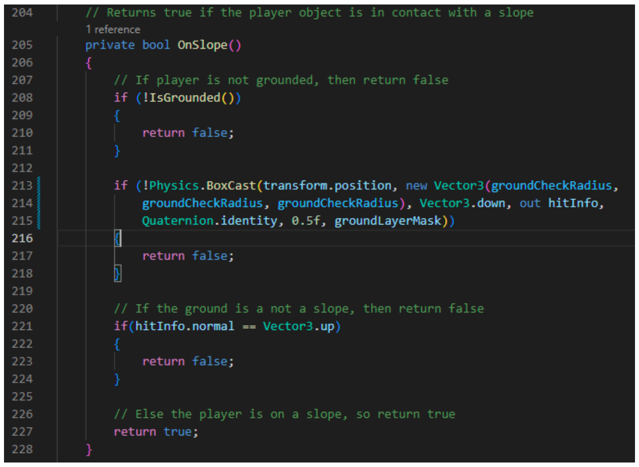
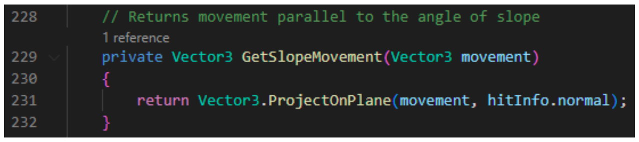

```
// Returns true if the player object is in contact with a slope
1 reference
private bool OnSlope()
{
    // If player is not grounded, then return false
    if (!IsGrounded())
    {
        return false;
    }

    if (!Physics.BoxCast(transform.position, new Vector3(groundCheckRadius,
        groundCheckRadius, groundCheckRadius), Vector3.down, out hitInfo,
        Quaternion.identity, 0.5f, groundLayerMask))
    {
        return false;
    }

    // If the ground is a not a slope, then return false
    if(hitInfo.normal == Vector3.up)
    {
        return false;
    }

    // Else the player is on a slope, so return true
    return true;
}
```

Figure 13 – Final version of the code for the "OnSlope" Method.

This function also went through a lot of testing and iterations similar to the "IsGrounded()" method. In one version, when the player stood over the edge or corner of a platform. Each normal of the multiple sides, were being linearly interpolated together into one normal. Which caused this method to return true on corners and edges of platforms, which caused many unintended effects. This was resolved by updating how the normal was received and the size of the “BoxCast” used.

### 5.2.2.3 Get Slope Movement Method

This function takes in the movement as a parameter, then returns the adjusted movement parallel to the angle of the slope. This is done by using the “ProjectOnPlane()” function. This function projects a vector onto a plane that is determined by a normal vector perpendicular to the plane.



```
// Returns movement parallel to the angle of slope
1 reference
private Vector3 GetSlopeMovement(Vector3 movement)
{
    return Vector3.ProjectOnPlane(movement, hitInfo.normal);
}
```

Figure 14 – Final version of the code for the "GetSlopeMovement()" Method.

Without this method, the movement was only in the X and Z axis. If the player was on a slope and wanted to move down the slope, they would instead move away from the slope, along the x and z-axis, then fall due to gravity. As shown in figure 15. If the Player decided they wanted to move up the slope, they would be pushing into the slope, which would make them still climb it but at a slower velocity. So, this was function added to ensure that a player could smoothly move up and down slopes, at the intended speeds.

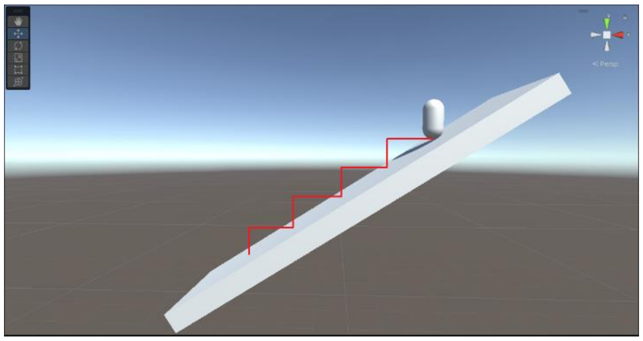


Figure 15 - Visualization of the movement path the player would take, when moving down a slope, without the “GetSlopeMovement()” method.

Through testing it was discovered that this method had the unintended effect of causing the player to sometimes bounce while moving up a slope. However this is later resolved in the “FixedUpdate()” method. See section 5.2.2.8 for solution to this problem.

### 5.2.2.4 Is Touching Wall Method

This function checks if the player is currently in contact with a wall, similar to the “IsGrounded()” function, this function checks if there is a wall using the “CheckBox()” and “CheckCapsule()”. These functions work the same as the “CheckSphere()” function the only difference is the shape used to check for colliders, being a cuboid and capsule. The reason why two functions are needed, is so that the area around the cylinder part of the players capsule object is checked, so only checking for colliders around the sides of the player object and to ignore any colliders above or below. No “CheckCylinder()” function exist in Unity. However, when a cuboid with sides of length X intersects with a capsule of equal sides and radii, the resulting shape is a cylinder. The Radius is set to be slightly larger than the players objects radius, to leave a small gap to detect the wall. So, overall if both the “Check” functions are true, the player is touching a wall and the Function returns true, otherwise returns false.

### 5.2.2.5 Get Wall Movement Method

This takes in “movement” and “wallMod” as input parameters then returns an adjusted “movement”, and “wallmod” as an out parameter. This method adjusts movement so that the movement in the direction of the wall is set to zero. This is done through a series of if statements that check the direction of movement and the normal of the wall, then set the corresponding direction to zero. The “wallNormal” variable is a global variable that is updated in “OnCollisionStay()” method. Next the “wallMod” variable is calculated. This variable is used later in the “FixedUpdate” method to limit the max velocity. The value of “wallMod” is set to the value of the final movement magnitude divided by its initial magnitude.

The reason this method was added, was due to a strange movement anomaly discovered during testing when the player was moving while in contact with a wall. When the player moved into a wall but not perpendicular to it. The player would still gradually accelerate to their maximum speed, even if they were almost perpendicular to it. This was probably a biproduct of removing friction from the walls.
This method also stops problems with the player’s collider getting stuck on some parts of the wall colliders, since parts of the wall are made up of multiple wall segments with their own colliders. This problem is sometimes referred to as ghost vertices [20].

One problem with this method is that it only accounts for walls that run along the X and Z axis, however this is not a problem as currently every wall in the final version of the game is parallel to these two axes. If the game was to be expanded with more walls this method may need to be updated.

### 5.2.2.6 On Collison Stay Method

This is a built-in method in Unity that is called once per frame for every Collider or Rigidbody that touches the player’s Collider or Rigidbody. This method was simply added to get the normal of a wall from a collision. This was done by using the “ContactPoint” from the “Collision” input parameter. A “ContactPoint” describes a point where the collision occurs and contains many useful properties including the normal of the collider from the collision. So, to determine a wall's normal, one simply needs to check if the magnitude of its normal in the X and Z axes is equal to one, therefore it is a wall and not slope or a floor. Then the “wallNormal” global variable is set to this normal so that “GetWallMovement()” method can make use of it. This if statement only works for walls that are parallel to the y-axis, so only normal vertical flat wall, which are the only walls in the game.

To ensure that this function was working as intended another “Debug.DrawRay()” method was used to visualise the wall normal. Which allowed the student to clearly see if it was working properly. See figure 16.

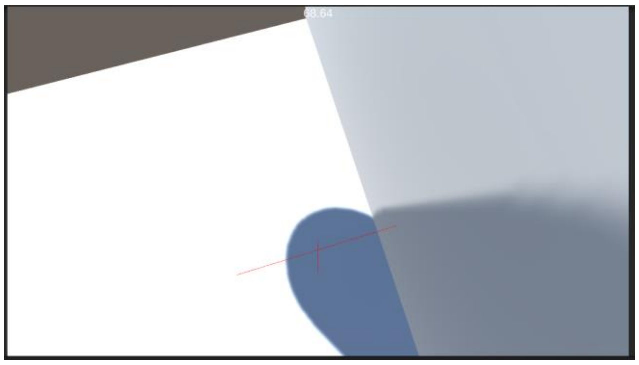


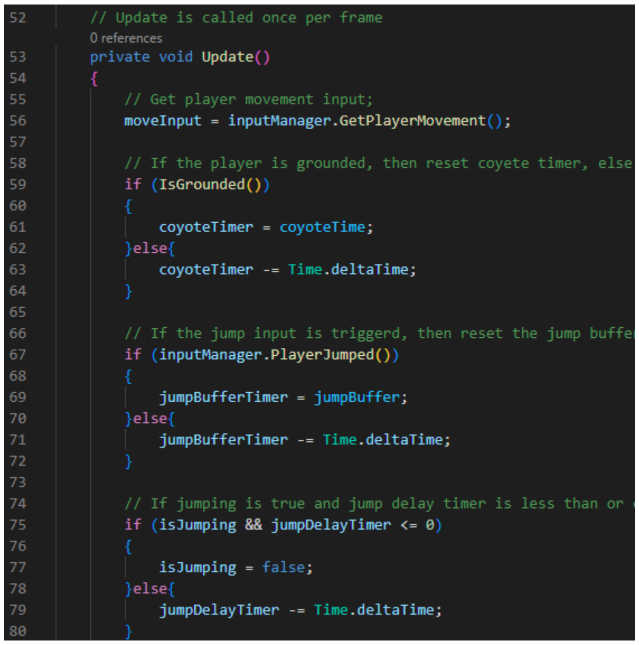
Figure 16 - Screenshot showing debug rays for the normals of the surfaces the player object is in contact with.

#### 5.2.2.7 Update() method

In the “Update()” method the players movement input, from the Input Manger, is received and stored in the “moveInput” variable. Then if the Player is grounded the coyote timer is reset, if not the coyote timer is started.

After checking if the player grounded to start the coyote timer. Next there is another if statement to check if the player has pressed the jump button, from the input Manager. If true, the jump buffer timer is reset, if false, the jump buffer timer starts counting down. This is followed by another if statement that checks if “isJumping” is true and that the jump delay timer is zero or less. If true is jumping is set to false, if false, decrement jump timer.
The jump delay timer and “isJumping” variable were added after testing, to prevent the possibility of multiple jump inputs triggering jump multiple times before the player object left the ground. Before this was added it was possible for the player to jump more than once before leaving the ground if they pressed the button in quick succession. Now there is a 0.2 second delay between when a jump can be allowed to be executed, to prevent this problem.

So, this if statement essentially checks the delay timer has run out. If it has the player is allowed to jump again.



```
    // Update is called once per frame
    0 references
    private void Update()
    {
        // Get player movement input;
        moveInput = inputManager.GetPlayerMovement();

        // If the player is grounded, then reset coyete timer, else
        if (IsGrounded())
        {
            coyoteTimer = coyoteTime;
        }else{
            coyoteTimer -= Time.deltaTime;
        }

        // If the jump input is triggerd, then reset the jump buffe
        if (inputManager.PlayerJumped())
        {
            jumpBufferTimer = jumpBuffer;
        }else{
            jumpBufferTimer -= Time.deltaTime;
        }

        // If jumping is true and jump delay timer is less than or
        if (isJumping && jumpDelayTimer <= 0)
        {
            isJumping = false;
        }else{
            jumpDelayTimer -= Time.deltaTime;
        }
```

Figure 17 - Code snippet from Player Controller script showing the Update() Method.

#### 5.2.2.8 Fixed Update

Next is the “FixedUpdate()” function where all of the physics calculations are carried out, in order to move the player object. First two local variable are set to the value of the “IsGrounded()” and “OnSlope()” functions;

Then another local variable “movement” a Vector3 is set equal to the “moveInput” converted from a Vector2 to a Vector3, by set the Y component equal to zero. Then this gets combined with the direction of the camera, so that if the player was to move forward it would move the player in the direction the camera is facing. This is achieved by multiplying the z-axis of the camera by the Z component of the movement and by multiplying the x-axis of the camera by the X component of the movement. Then the result is normalised.

Next are two if statements, one for if the player is on a slope the movement adjusted using the “GetSlopeMovement()” function. A new float variable “wallMod” is initialised to one. Then if the “IsTouchingWall() function returns true then convert movement to wall movement with The “GetWallMovement()” method.

Next the player can finally be moved using the player’s Rigidbody component. This is done by using the “AddForce()” method, which has parameters “force” and “forceMode”. The force is set to be the result of the “movement” variable multiplied by the “speed” global constant. And for the “forceMode” is set to “ForceMode.Force”. This applies a continuous force to the rigid body.

However, since friction was set to zero for all surfaces, the player would continue to slide after the input key is released. So, to resolve this artificial friction was added by applying a force in the opposite direction of the player movement when the player is grounded and there is no move input. This allows the player to move, by applying a force when a move key is pressed and the when the key is released the force will be taken away and the player will gradually slow down.

However, the players max velocity is too fast, so it gets limited with the global constant “maxSpeed” by checking if it player velocity is greater than “maxSpeed” multiplied by the “wallMod” variable. Then the player velocity set equal to this max velocity. The Y component is excluded from this calculation so that it only effects player movement and not the players fall speed.

Later in development an if statement was added here to check if the player is grounded and moving which plays a movement audio clip.

Next, we can deal with the player jump. This is done by having an if statement that checks if “isJumping” is false and both coyete and jump Buffer Timers are greater than 0. These values were set up back in the update function, but this basically means the player jump is off cooldown and the player is grounded or has just left the ground and player has press jump in the last few milliseconds.

If this statement is true, then “isJumping” is set to false and player’s Y-component of the velocity is set to zero. Then by using “Rigidbody.addForce” again but with the “jumpForce” constant and Impulse forceMode instead, an upward force is applied. An Impulse force is a force that an instant force that is applied to an object, this allows for a quick and responsive feeling jump. Then the three timers (delay, coyote, buffer) are reset.

Later in development an audio clip is also played here for a jump sound.

Next is an if statement that solve a problem where when a player is moving up a slope they will sometimes bounce. This if statement check if the player is on a slope, if they are then a downward continuous force is applied to keep them from bouncing.

Later in development an if statement was added here to checks if the player has just land back on the ground, by checking if the player is grounded and was not grounded last update. Then a jump land audio clip is played.

#### 5.2.2.9 Changes in the Unity Inspector

A few changes were also made in the Unity inspector. A new material was created that was then added to all the game objects that made up the geometry of levels: floors, walls etc. This was done to resolve a problem where the player could stick to wall mid-air if they jumped into them while moving towards them. However, this caused the player to slide about on the ground as though they were on ice. So artificial friction was added when the player is grounded as mentioned earlier.

#### 5.2.2.10 Testing and fixing problems

During the development of this the script on-going testing was conducted and fine tuning of the script was done to ensure that the new features worked as intended and that the player movement felt responsive and good to control.
The constant variables, that set the various movement properties; speed, jump force, coyote time etc. They were constantly being changed and fine-tuned so that the movement and jump had a nice and responsive feel to it, the jump force was also adjusted to make sure the player could jump small gaps and reached high enough to jump over small objects.

There were multiple problems that were discovered at this stage. Some that have not been mentioned yet. The “IsGrounded()” and “OnSlope()” methods went through multiple different versions as mentioned earlier however the position of the ground check child object had to be moved and the constant for its radius, had to be adjusted a few times. This was due to the player reaching a grounded or on slope state in unintended situations.

### 5.2.3 Camera Look

The next thing that was done was to configure the camera so that the player uses the look input to move the camera. Unity’s default camera is quite simple so to get more settings and control over the camera another package called Cinemachine was installed. This package adds much more tools for cameras in the Unity inspector without the need for complex scripts.

Once added a Cinemachine virtual camera game object is added to the scene. This object has multiple components that have lots of settings that then are applied to the main camera. So instead of configuring the main camera, the virtual camera is set up instead. Another empty game object is also created called “CamPos”. Then both these objects are added as child object to the player object. Then “CamPos” position is set to roughly three quarters of the way up the player object and slightly forward from the centre. This object was created to define exactly where the first person is going to sit. Finally, in the virtual camera we set it to follow this object. Now the camera view is first person and moves with the player when they move around.

To get the camera to move with the look input, usually you can configure this in the virtual camera settings. However, with the utilisation of the new input system and later in-development will be adding the ability for a user to customise their sensitivity. A new script was added to override this part of the virtual camera settings.

This code was adapted from code seen in the following video [21]. The code adapted can be seen from 13:53 to 20:25. Using the "PostPipeLineStageCallback()" method, then an if statement to check if the virtual camera is following something. Then another if statement checking that it is the aim stage. This allows for the aim part of the virtual camera settings, to be overridden in this script.

To start a variable for the camera rotation is set to the camera's current rotation if it was null. Then the mouse input is received from the Input Manger instance. The sensitivity is set, and horizontal and vertical sensitivity are set to this sensitivity. Then the mouse input multiplied by the corresponding horizontal or vertical sensitivity is added to the corresponding X and Y component of the camera rotation. The Y component also needs to be clamped so that the camera cannot be rotated more than 89 degrees up or down, to ensure the play cannot look behind them upside down. Then new camera rotation is applied to the camera and horizontal rotation is also applied to the player object to get it to rotate. Originally it was set to 90 degrees, but this would cause a problem as in the player controller script, it needs a direction in order to move the. So, if you looked straight up or down and tried to move, you would not. So, it was changed to 89 degrees instead.

Then later in development, changes were made to this script, to account for changes in other parts of the project. Mainly to deal with the addition of sensitivity settings and pause menu. So, the sensitivity is set to the saved value, using player preferences, instead of the pre-set value. The players settings for inverting the horizontal and vertical sensitivities are also applied here by checking if they are enabled, then setting the corresponding sensitivity negative, again with player preferences. When the pause functionality was added, the player and camera would still rotate when in the pause menu Sensitivity was also set to zero when the game is paused, by checking if the timescale equals zero.
Now that both the movement and player look had been implemented, the user can now use either keyboard and mouse or a controller to control the player character. Allowing them to move and jump around and look around the game from a first-person viewpoint.

## 5.3 Adding Levels

### 5.3.1 First Batch of Levels

The next thing the student decided to work on was implementing some levels. But first a new script was created for a scriptable object for generating and storing a seed. Scriptable objects are a data container in Unity that can be saved as an asset. This will be used to generate a new seed the first time a level is loaded, then every time the level is reset the seed gets reused. This script is very simple, as it has a single integer property called seed. Which is set to public, but with a private set. And then one public method to generate the seed which sets the seed to a random value using a random range method.

Next another script was created. This time a class called Spawner that will be used to store a position of an obstacle spawn position. This is again simple script with to serialised private fields, so that they can be set in the inspector. One for obstacle game object, so that different obstacles can be used and another for the Spawner's position. Then there are there are two public get methods to access these.

Next the old prototype spawn obstacles script was updated. The Scriptable seed object was added and used to get the seed. A list of Spawners object was then added, to replace the manually set positions. Then this list gets iterated through to start a coroutine to spawn the obstacles at each position. Now a list of spawners can be passed in through inspector, allowing different objects and positions across multiple levels. Then two serialised floats were then added to specify the minimum and maximum time between obstacles spawning. This was added so that individual levels could have different values.

The original obstacle prefab was updated and adapted into two prefabs. One for the going in the left direction and one for going in the right direction. The overall side was reduced was reduced to a scale of one in each axis. Then the player object was then added as a prefab to make use of in each level.

Then development begin on the first level. This level would basically be an updated version of the idea from the prototype. A long corridor, with obstacles flying in on both sides. The geometry of the level was built using multiple game objects of different size and shapes. The smooth material was added to each object and its layer mask was set to the ground layer.

Then a new empty game object was created that would be used for the spawner positions, making it easy to place and visualise were objects would spawn. A spawn manager object was created, with the spawn obstacle script attached. Then the obstacle prefabs and the spawners positions were passed into this script, as seen in Figure 18.

The Player and Input Manager prefabs were added into the scene and the level was essentially complete. This process was continued to create twelve levels in total, all different variations of each other. Each level gradually increasing the number of obstacles to avoid and reduce the number of gaps between each group of obstacles, to create a rising difficulty curve.

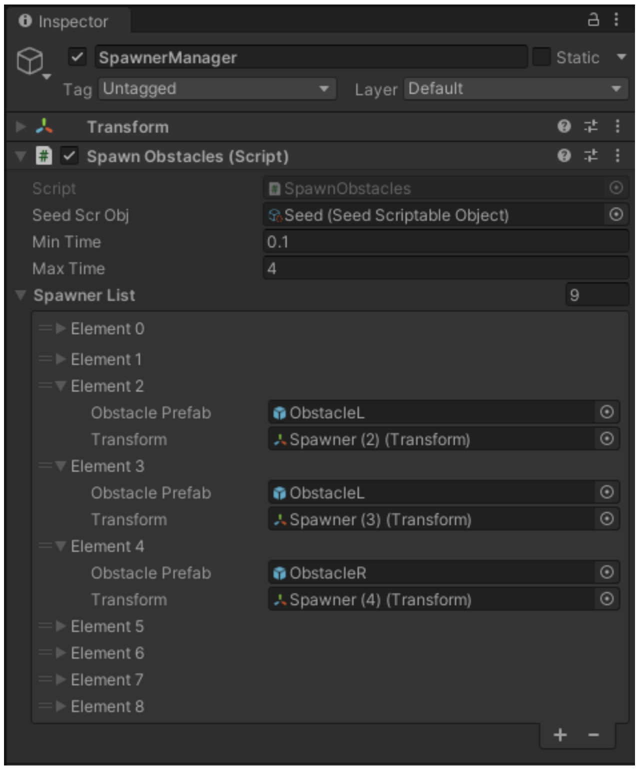

```
Inspector
SpawnerManager                          Static
Tag  Untagged            Layer  Default
Transform
Spawn Obstacles (Script)
Script              SpawnObstacles
Seed Scr Obj        Seed (Seed Scriptable Object)
Min Time            0.1
Max Time            4
Spawner List                                   9
  Element 0
  Element 1
  Element 2
     Obstacle Prefab    ObstacleL
     Transform          Spawner (2) (Transform)
  Element 3
     Obstacle Prefab    ObstacleL
     Transform          Spawner (3) (Transform)
  Element 4
     Obstacle Prefab    ObstacleR
     Transform          Spawner (4) (Transform)
  Element 5
  Element 6
  Element 7
  Element 8
```

Figure 18 - The Spawn Manger's Spawn Obstacles Script component in the Unity Inspector. Showing the configuration for the component in stage 1 level 1.

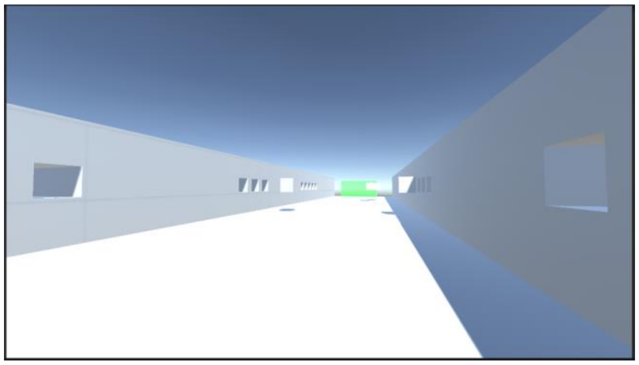


Figure 19 - Screenshot of an obstacle level, from the players perspective, taken in stage 1 level 1.

### 5.3.2 Second Batch of Levels

Then work began on a second group of levels which would utilise a new fake platform mechanic. A new script called platform row was created to hold a list of platform game objects, with methods to return the list, its count and platform at an index. This script was added so that another new script called spawn fake platforms, could make use of it. This script has serialised fields for the seed scriptable object, and int to hold the number of rows of platforms and a list of platform rows. Then in the Start method the random generator is initialised with the seed. And a "foreach" loop is used to loop through the list of Platform Rows and a random platform collider is enabled. This ensures that each row will only have one that will have a collider. So, one true platform and the other being fake for each row.

Then a fake platform prefab was created, a platform with its collider disabled. A new scene was created with, with two large platforms at either end with a series of rows of fake platform between them. An empty Platform Manager object was created with the Spawn Fake Platforms scrip. The number of rows was set, and the seed object and platforms were passed into the script.

Then two more levels were created but with more columns of platforms. Then another nine levels were created using a combination of the fake platforms and the spawning of obstacle. For a total of twelve levels utilising the fake platform mechanic.

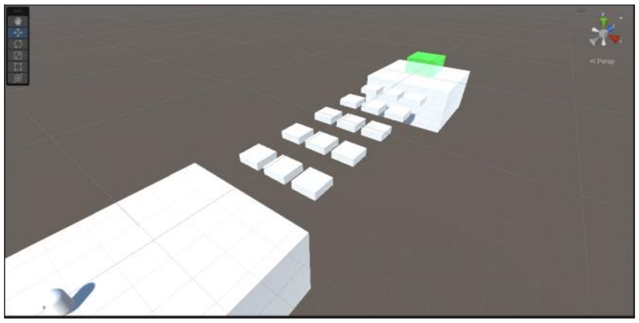


Figure 20 – Screenshot of Stage 2 level 2 in the Unity scene view.

### 5.3.3 Testing and fine tuning of levels

The levels went through testing and multiple changes especially the spawning of obstacles mechanic. This mechanic went through multiple tweaks for example reducing the speed of the obstacles and its spawn rate to ensure that the levels were not too difficult. The number of spawners were changed multiple times as well. Two levels were also cut and replaced due to them being much too difficult to complete.

The spacing and size of the fake platforms went through adjustment to stop player being able to skip platforms by jumping over them. As well as to ensure that a player can always make the jump between each platform row, i.e., platforms on opposite sides. To test that the fake platforms were being correctly set up, the inspector could be used to check which had their colliders enabled or not as well as debug log statements that indicate which platforms had their collider enabled.

## 5.4 Ghost Mechanic

Next the recording and playing of the ghost mechanic was added. First two classes were created. One to hold the ghost data, to hold the timestamp, position, and rotation of a ghost at a particular time. And one to hold a list of this ghost data for a specific ghost. Then another scriptable object for the ghosts was created to store and save a list of each ghost data. This object also had a float variable to set the frequency of recordings, how often per second to record.

Then a new prefab object Ghost Manager was created and added to only the levels that have obstacles to avoid. As well as a prefab for a ghost object which is the same shape and size as the player, but with a transparent material applied to it.

### 5.4.1 Ghost Recorder

Then a script to record the player data and save it to the ghost scriptable object was created. In its awake method, it first checks to see if the ghost manager object exists. If true it sets a Record Boolean to true, if not sets it too false. Then if record it true it adds a new ghost to the scriptable object. If record is false, the list is cleared. Then another if statement that removes the last ghost if the size excides ten. Next in the Start method the time variable gets set to zero.

Now using the “LateUpdate()” method, which is another built in method in Unity that works the same as Update but instead is called after all Update methods have been called. This method was chosen over update to ensure that the player had already moved this frame before this method is called. This is where the recording is done. The time variables are set to the current time. Then an if statement that checks if record is true and the record frequency to check if it is time to record. If true the current timestamp, position and rotation of the player is saved by adding to the lint of the new ghost, in the scriptable ghost object.

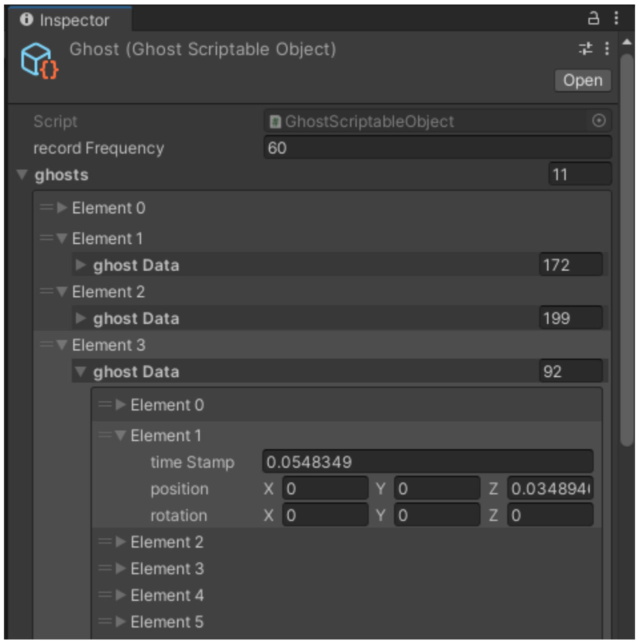

```
Inspector
Ghost (Ghost Scriptable Object)
                                              Open
Script              GhostScriptableObject
record Frequency    60
ghosts                                        11
  Element 0
  Element 1
    ghost Data                                172
  Element 2
    ghost Data                                199
  Element 3
    ghost Data                                92
      Element 0
      Element 1
        time Stamp  0.0548349
        position    X 0     Y 0     Z 0.034894
        rotation    X 0     Y 0     Z 0
      Element 2
      Element 3
      Element 4
      Element 5
```

Figure 21 - The Ghost Scriptable object asset in the Unity inspector. Showing how the ghost data stored.

### 5.4.2 Ghost Player

Then another script to play this data was created. A global list variable is set up to keep track of indexes for each ghost. In its Awake function it checks if there is more than one ghost in the scriptable object. If true then, for each ghost excluding the new one, a new ghost object is added to the scene. Then a variable for the current time is set to zero again in the start function. Then in the Update function the time is set to the current time, then the “UpdateGhostObjects()” method is called.

This method updates the ghosts position based on the current time. For each ghost, excluding the new one, in the scriptable ghost object. If the current index is less that the current ghost’s data count, initialise variables for the current and next ghost data. And then if current time is greater than the current ghost data timestamp and less than the next ghost data timestamp, then update the object position and rotation by linearly interpolating between the two values. Else if current time is equal to next ghost data's timestamp, then set ghost object position and rotation equal to next ghost data values and increment the current index. And then another if statement for if the current time is greater than next ghost data's timestamp, then increment current index. Back to the original if statement if false and else if the ghost object is still active, then deactivate the ghost object since there is no more data as the player has been reset.

Then the ghost recorded script was added to the player prefab and the ghost recorder was added to the ghost manager prefab. Then the ghost manager was tested and then added to each level. And now whenever the player resets a loop, for example by colliding with an obstacle, a ghost will play back the player moves from the previous loops, up to a maximum of ten ghosts.

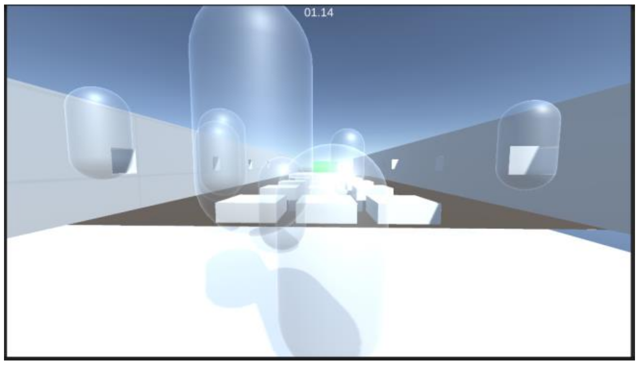


Figure 22 - Screenshot from player's perspective showing player ghosts.

### 5.4.3 Testing and Problems

This mechanic when through a lot of testing to ensure that the recoding and playing was working correctly and to make sure that it did not have a big impact on the performance of the game. One problem the student encountered was in an earlier version of these scripts that recorded the data every single frame. However, this had a noticeable impact on the games performance and the frame rate was significantly reduced. So, to solve this problem the recording frequency was reduced by adding the record frequency variable, which is set to 60 times a second, and checking that instead. As well as simplifying the scripts and how the data was added and stored in data structures, which made the whole mechanic run much more efficiently.

Testing also focused on whether the data was being recorded accurately and at the right times by printing the data to the console as well as checking the ghost scriptable object in the inspector at runtime.

## 5.5 User Interface and Menus

Next the work began on the User Interface. This mostly consisted of multiple menus for the game, including a main, level select, pause, level complete and an options menu.

### 5.5.1 Main and Level Select Menus

The main menu and level select menu were created first by adding a new scene then adding Unity UI elements including a canvas with a background and a panel object child object. Then adding three empty game objects as children of the panel, then renaming these three objects to main menu, level select menu and options menu. Then a title and three buttons were added to the main menu, one to open the level select menu and another for the options menu and one to quit the game.

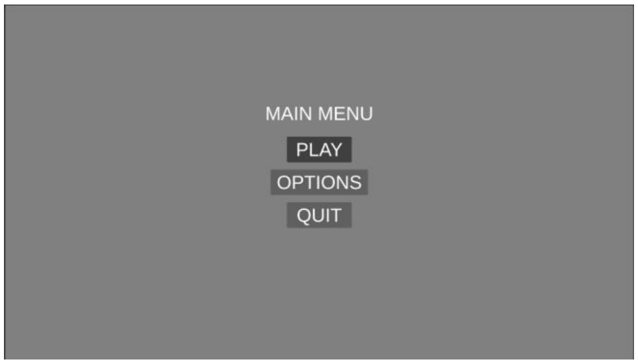


Figure 23 - Screenshot of the Main Menu.

Then for the level select menu another title and a grid of buttons and a grid of checkmark images, to indicate if the level has been completed, for each level as well as a back button.

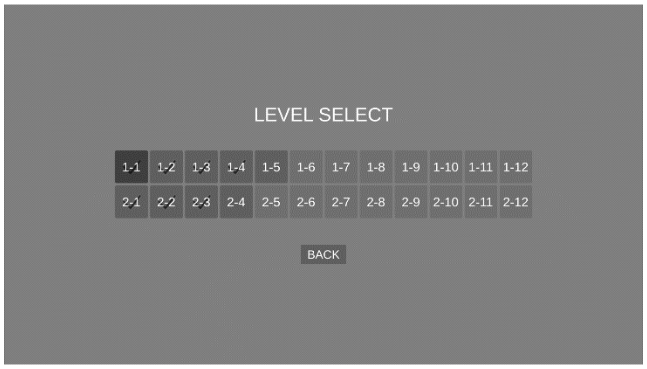


Figure 24 - Screenshot the Level Select Screen. Tick marks show completed levels, dull icons show levels that are not unlocked yet.

The options menu is left mostly blank for now, except for a back button as it will be configured later. Since there currently not settings set up to edit.

A script was added to implement the functionality for these menus. The script would require reference for both the ghost and seed scriptable object to initialise them for a level. As well as references for the three menus and a reference for the first button of each menu. Then in the Awake Method by using "PlayerPrefs" which is a built-in class in Unity that allows for the storage and retrieval of Player data between game sessions. They are stored as key-value pairs, where the key is a string and the value can be a string, int, or float.

Then by checking a "PlayerPrefs" for completed levels, the corresponding image can be enabled or disabled. And by checking "PlayerPrefs" for the number of stage one and stage two levels completed, are used to set which buttons are interactable. Used to indicate which levels are unlocked and to prevent the player accessing levels that they have not unlocked yet.

Then if there are instances of input manager as well as loop counter, which was added later in development, destroy these instances. As they are not needed here.

In the start method, only the main menu is set to be active, and the first main menu button is preselected. This is done to because if a player uses keys or a controller, an element needs to be preselected for them to navigate the menu.
Then there is a method for each of the buttons, these methods are called through and on-click event that is set up in the inspector, as shown in figure 25.

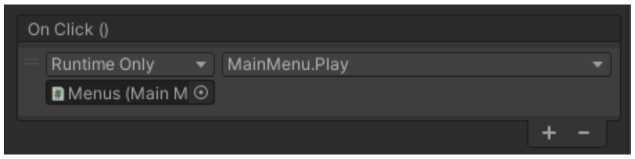

On Click ()
Runtime Only | MainMenu.Play
Menus (Main M

Figure 25 - On Click event configuration for the Main menu play button in the Unity Inspector.

The “Play()” and “Options()” function both disable the main menu and set the corresponding menu to be active as well as preselecting the new first button. Preselection of the first button is done, like this, every time for each of the menus when a new one is opened. The “Quit()” method calls Application.Quit() to close the game session. Then one “Back” function is used for both menus that deactivate both menus and activates the main menu.

Then finally the “LoadLevel()” Method which takes in an index parameter indicating which button in the grid buttons was selected. The list of ghosts in the Ghost Scriptable Object are cleared and the seed is generated in the Seed Scriptable Object. Then using the index, the corresponding level is loaded along with the with pause menu scene. This script was then attached to the panel element of the canvus

### 5.5.2 Pause and Level Complete Menus

Another scene was created which is similar to the main menu’s scene. It has a canvas, with a background, but this time it is transparent to see the game. A Text element for the current loop time to be displayed then three empty child game objects for the pause, options and level completed menus. The options menu is again left black here as it will be developed next. For the pause menu a title and five buttons are added.

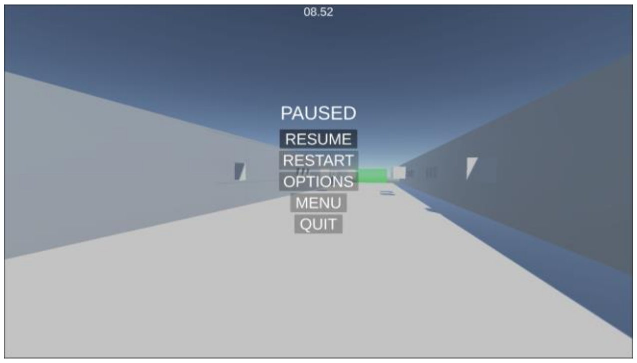


Figure 26 - Pause Menu.

And for the level select menu a title, four text elements and four buttons are added as seen in Figure 27.

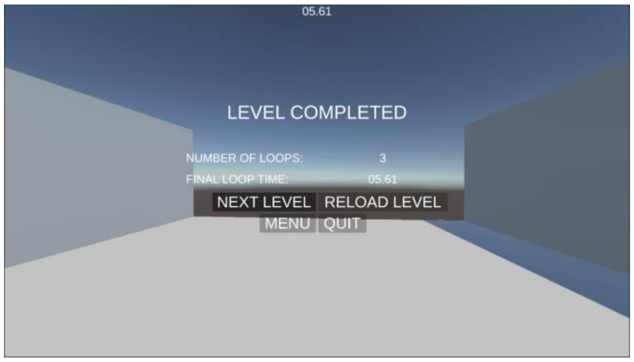


Figure 27 - Level Completed Menu.

Before a script was created for these menus a new singleton script was created call “LoopCounter” which was created to keep track of the total number of loops for the level completed menu. This script awake method is similar to the other singletons, as checks if it is the only instance and destroys itself if there is already an instance. In the start method a global loop count variable is set to zero. Then a public method called “incrementLoopCount()” is created which increments the loop count variable by one. This script is then adder to a new empty game object called loop counter, then added as a prefab and added to each level.

Then the script for the in-game menus was created. This script needs references to both the input manager and loop counter instances, as wells as refenced to the different menus and button like the main menu script before. It also has references to the text element that are used for the timer and the level completed menus loop and time values.

In the Awake method each menu is deactivated then the mouse cursor is locked to the window and its visibility is set to false. So, that it can’t be seen while playing. In the Start method the input manager, loop counter and time are initialised. Then the loop counter is incremented. Then in the Update method, using an if statement to check if the level completed menu is not active and if the pause input has been pressed, using the input manger. Then if the background element is active “ResumeGame()” method is called, else “PauseGame” method is called. Then after the if statement time is set to the current time and the text elements are updated.

Then like the other menu script there is a corresponding method for each button. In the “ResumeGame()” method which is called in when the resume button is pressed or the input key is pressed while the pause or option menu is open. It sets deactivates the menus locks and hides the cursor and sets the timescale to one, so that the game resumes.

Then the PauseGame() method does the opposite. This called when the pause input is pressed when the menus are not active. It sets the timescale to zero to freeze the state of the game, activates the pause menu and background as well as unhiding an unlocking the cursor.

The “RestartLevel” method sets the timescale back to one and restarts the level, as though you had died. The “ReturnToMenu()” method sets the timescale back to one and loads the main menu scene.

The “NextLevel()” method for loads the next level by calling the “LoadLevel()” method, with the current levels index as a parameter. However, if there are no more levels, it instead loads the main menu scene.

The “ReloadLevel()” method also calls the “LoadLevel()” method, with  the current levels index as a parameter. This reloads the level with a new seed.

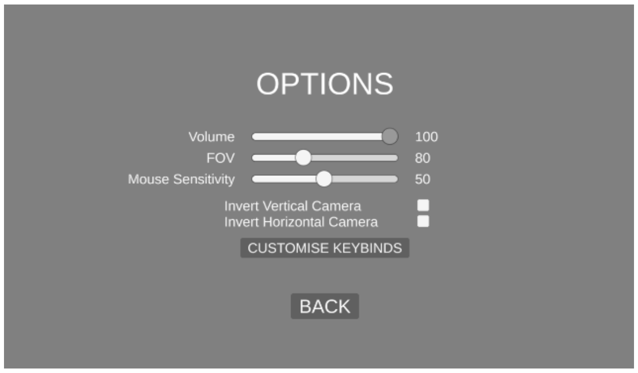
The “LoadLeve()” method destroys the input manager and loop counter instances, then clears the list of ghosts from the ghost scriptable object and generates a new seed in the seed scriptable object. Then the index parameter is used to load the corresponding level along with the pause menu scene.

The “QuitGame()”, “Options()” and “Back()” methods method works the same way as in the main menu script. Then finally the “LevelCompleted()” Method which will be used later when a level is finished to call open the level completed menu. This freezes the state of the game by setting the timescale to zero, then sets an index of the current level. This index is used to update the “PlayerPrefs” for the completed levels, that were used in the level select menu. Then the level completed menu is opened.

This script was then added to the canvas in the pause menu scene since the script need to be active even if the menu is not. The buttons were then set up to call the corresponding functions, and the scene was finished.

### 5.5.3 Options Menu and Player Settings

Next development started on the options menu, which would need three sliders and two check boxes for the various settings and a button that will lead to another menu for the customising key binds. Another empty game object was added for this menu, with a back button, that will be developed after.


OPTIONS

Volume 100
FOV 80
Mouse Sensitivity 50
Invert Vertical Camera
Invert Horizontal Camera
CUSTOMISE KEYBINDS
BACK

Figure 28 - Options Menu.

A new script was created with references to each UI element. As well as the reference to the player object its prefab and their cameras, so the field of view can be changed. An audio mixer which does not exist yet but will be created after for the volume slider. And references to the two menus so that they can be activated and deactivated, as well as the first buttons so they can get preselected like the other scripts.

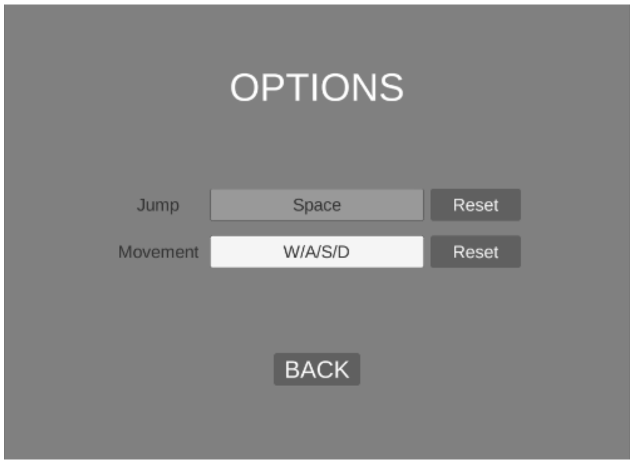
In the “Awake()” method the virtual camera and its prefab are set, using the get component function using the player object and its prefab. The “checkSetDefaultSetting()” method is called. This method simply checks if there are values saved as “PlayerPrefs”. If not, it sets the player “PlayerPrefs” values to the constant default values.

Then there are three functions that are for each slider that get called when a slider value gets changed. The slider values are saved in each method by using “PlayerPrefs”. The “SetVolume()” method sets the value of the audio mixer to $20\,log_{10}(volume)$ where volume is measured in decibels. This is because human perception of sound intensity, is logarithmic rather than linear. The “SetFOV()” method updates the player prefab camera’s field of view and the player object’s camera’ FOV only if it exists in the scene. This is done since this could be done either in a level, where it would exist, or in the main menu, where it would not.

Finally, the “SetMouseSensityity” just sets the “playerPref”. Then two functions for the checkboxes to invert the horizontal and vertical sensitivities. They both take in a bool which is converted to an integer and is saved as a “PlayerPref”. Then the CinemachinePOVExtension script was updated to load the sensitivity and if they should be inverted using these “PlayerPrefs”.

### 5.5.4 Key Rebinding

Then for the key rebinding menu, the rebind component as seen in figure 29, is not a built-in element of Unity and instead is adapted from the key rebinds sample. Which is an optional download that is part of the new input system package. This sample already contained code that handled the rebinding however it does not account for when a player rebinds a key, but the key is already in use. So, the scripts needed to be adapted to prevent this.


OPTIONS

Jump Space Reset
Movement W/A/S/D Reset
BACK

Figure 29 - Rebinding Menu.

## 5.6 Finishing Touches

The next stage of development was dedicated to getting the game ready for user testing. So, a few key features were still missing, mainly audio and a way for a player to complete a level.

### 5.6.1 Audio

First an Audio Mixer was added. This is a Unity asset that is used to mix various audio sources. In this case only a master volume mixer is created to be used in the options menu which controls the volume for all in-game sounds. Next, multiple audio clips were added for movement, jumping and landing. All the sound files were from a free asset form the unity store called Footsteps – Essentials [22]. Then a new empty game object was added to the main menu scene called Audio Manager. Then two new scripts are created one for a Sound Class and Another for the audio manager.

The sound class script has many serialised fields for the multiple settings of single sound, including: name, list of Audio Clips, the audio mixer group, a Boolean to toggle if it should loop, volume and pitch. Then there is one public method "GetRandomAudioClip()" which returns a random audio clip from the list of audio clips.

Then the audio manager script which is another example of a singleton class. A serialised field for a list of the sound class is added. Then in the Awake method, the object is set to don't destroy on load to ensure that the audio manager persists through every scene in the game. Then the checks if this is the only instance. Then a "foreach" loop is used to loop though the list of Sounds, to initialise each one.
Then a public method called "PlaySound()" which uses a string parameter name to play the sound with that name. This is achieved by calling the "SearchForSound()" method then if it returns not null, a random audio clip from that sound is played.

Then another public method "IsPlayingSound()", that returns a bool if a sound with the input parameter name is currently being played. It calls the "SearchForSound()" method, then if the sound is not null, returns true if the sound is playing and false if not.

Then finally one more private method "SearchForSound()". Which returns a Sound, using the name input parameter to find it. This works by looping through each sound until one matches the name, then returns if found. If none are found, then it will return null.

Then the "PlaySound()" and "IsSoundPlaying()" methods are added to the player controller script to play the corresponding sounds when the player moves, jumps or lands.

Then this script was added to the Audio Manager object and the three sounds were added for movement, jumping and landing. Then there corresponding setting were set up and the audio clips added. As shown in Figure.

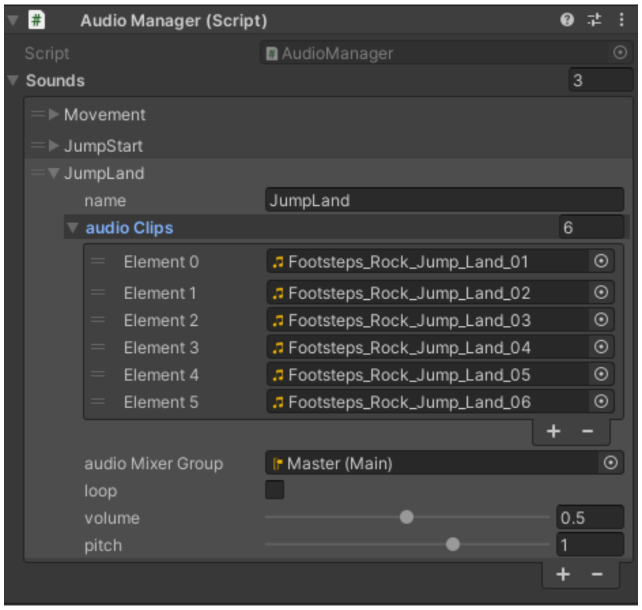

Audio Manager (Script)
Script: AudioManager
Sounds: 3
Movement
JumpStart
JumpLand
name: JumpLand
audio Clips: 6
Element 0: Footsteps_Rock_Jump_Land_01
Element 1: Footsteps_Rock_Jump_Land_02
Element 2: Footsteps_Rock_Jump_Land_03
Element 3: Footsteps_Rock_Jump_Land_04
Element 4: Footsteps_Rock_Jump_Land_05
Element 5: Footsteps_Rock_Jump_Land_06
audio Mixer Group: Master (Main)
loop
volume: 0.5
pitch: 1

Figure 30 - The Audio Manager script component showing how the audio clips are configured for the Jump Land sound effect.

### 5.6.2 Player, Obstacle and Finish bounds

The last few things to be added was a way to reset the player if they go out of bounds or fall off the platforms, and a way to de-spawn obstacle when they go out of bounds, and finally a way for a player to complete a level. These were done were done by creating a prefab called bounds which is just an empty game object with a box collider added. This box collider was set to be a trigger which means it will ignore the physics engine and just be used to trigger events. From this prefab, variants were created, variants inherit the properties of the original. Variants for a player, obstacle and finish bounds were created.

The player bounds were scaled to the correct size to so that it was slightly larger than each level, then a unique tag and layer was added for it, then added to each level. Then in the player collision script an "OnTriggerExit()" method was added which is called if the players collider leaves the trigger. Then an if statement is used to check the tag of collider is the player bounds tag, then the scene is restarted. For the obstacle bounds a few variants were created since the levels are different sizes, then each is given a unique tag and layer, then added to each level. And in the obstacle script another "OnTriggerExit()" Method is added which checks if the tag is the obstacle bounds then destroys the obstacle object.

Finally for the finish bounds variants, were scaled to an appropriate size, a green transparent material was added to them, and they were a unique tag. Then it was placed at the end of each level. Then in the player collision script an "OnTriggerEnter()" method was added which is called when the object enters the trigger. Which checks the tag then calls the "LevelCompleted()" method, from the pause menu script. Which then opens the level complete level menu, and the player can finish the level.

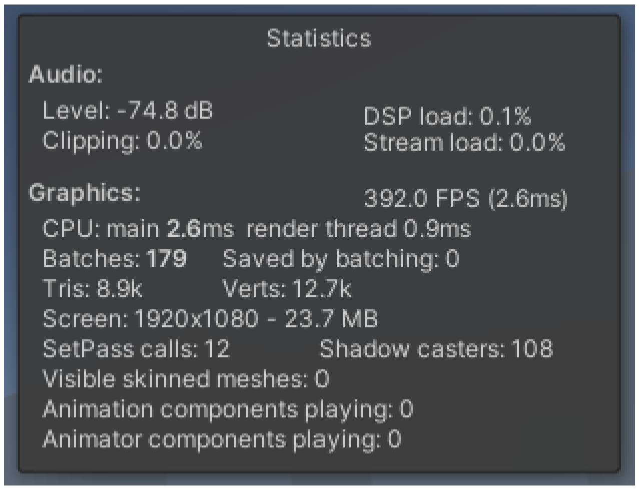
Then the game went through some finally testing to ensure it was working as intended. Then a build of the game was created which was used for user testing.

## 6 Evaluation / Testing

The implementation stage and the testing stage went hand in hand since every new feature was being tested as it was being added. Also considering many mechanics, specifically the ones in the player movement, level implementations and the ghost mechanic. Many had to be constantly fine-tuned when a basic version of it was complete. Then the student would playtest these features. Alongside checking that the intended values of variables or method return results. Such as velocities of game objects, correct Boolean values.  To achieve the intended effect, game feel, responsiveness, or suitable game balance. This often took a significant amount of time much longer than initially anticipated.

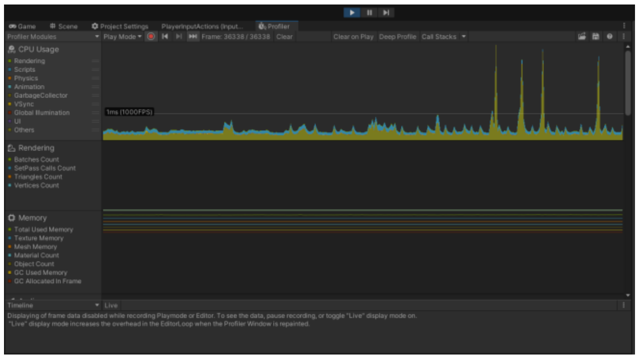
Many of the problems that was run into with sections of code, and their solutions have already been covered in the implementation stage. However, some additional things to note here are as follows.

The use of Unity's Rendering Statistics Window, see Figure 33, during runtime was extremely useful. It was to check that there was no significant or unexpected performance drops with new features. Containing stats on rendering such as well as fps, which was constantly check and compared, to ensure it stayed at expected levels. This was especially useful with the ghost recording mechanic where there was a significant drop in performance. The Unity profiler, see Figure 34, was also used here as it gave a more in-depth insight into performance. Which was used to help identify where the problem with the recording of the ghost data was. Which allowed the student to address the issue and optimise it, minimising the effect on performance.



```
Statistics
Audio:
  Level: -74.8 dB                 DSP load: 0.1%
  Clipping: 0.0%                  Stream load: 0.0%

Graphics:                         392.0 FPS (2.6ms)
  CPU: main 2.6ms  render thread 0.9ms
  Batches: 179    Saved by batching: 0
  Tris: 8.9k      Verts: 12.7k
  Screen: 1920x1080 - 23.7 MB
  SetPass calls: 12           Shadow casters: 108
  Visible skinned meshes: 0
  Animation components playing: 0
  Animator components playing: 0
```

Figure 31- Unity's Rendering Statistics Window.



Displaying of frame data disabled while recording Playmode or Editor. To see the data, pause recording, or toggle "Live" display mode on.
"Live" display mode increases the overhead in the EditorLoop when the Profiler Window is repainted.

Figure 32 - Unity Profiler.

### 6.1 Colourblind Tests

Colourblind tests were conducted on the UI and the game using a colour blindness simulator tool. See appendices. All the in-game menus had no issues due to the lack of colour and all appeared with the same level of contrast. For the in-game levels only one issue was really of note. The green colour of the finish area, in a Monochromacy/ Achromatopsia view could possibly be mistaken as level geometry since it appears as a light grey colour. Making it possible that a player might be unsure of where to go since it does not stand out.

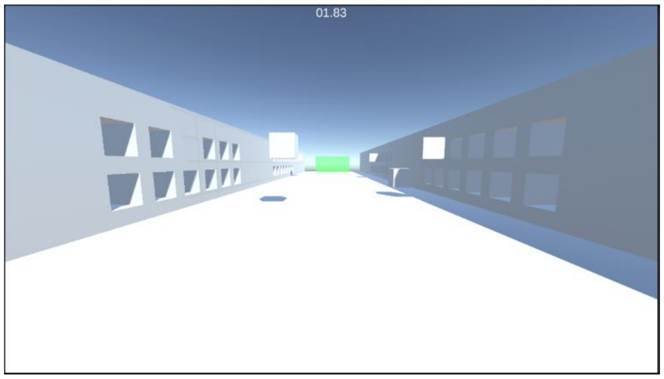


Figure 33 - Normal view.

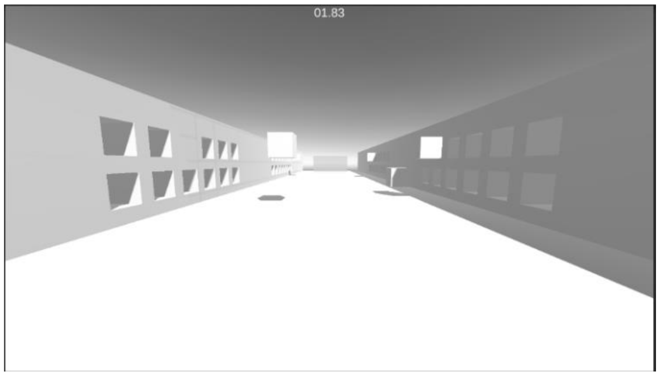


Figure 34 - Monochromacy/Achromatopsia view of the same image seen in Figure 33.

To resolve this a colourblind mode for this type could potentially get developed or the base colour is adjusted so that it perhaps stands out more as a dark object or perhaps some other visual indication could be given such as a cheque mark pattern.

## 6.2 User Testing

The student decided to perform user testing through with two separate playtest that were happening simultaneously. Users had the option to do one or the other. One was observed the other non-observed. In both playtests users were provided with a link to a OneDrive folder. Where they could download the relevant files and a build of the game.

They were instructed to read through the information sheet provided. Then sign the relevant consent form depending on the playtest, observed or non-observed. Then they could play the game for as long as they liked. After they were done with the game, they were instructed to fill in an online anonymous questionnaire. In they observed playtest the student was present when they played, and their gameplay was recorded for future reference.

The players were instructed to play the game as long as the liked with little to no direction. Then were asked to fill in a Microsoft Form Questionnaire. The form consisted of five sections: gameplay, input devices/controls, UI, options, and overall thoughts. With ten questions for the first section and three to four for the remaining sections. See appendices for a copy of the questionnaire.

Most users just opted in to the first type of playtest. While two opted for the recording of gameplay. A total of eight questionnaire responses were received.

## 6.3 Questionnaire responses

The following section goes into the feedback received from the questionnaire. All the responses can be viewed in the excel sheet in the appendices.

### 6.3.1 Gameplay Questions

Overall, users thought the game was easy to understand and follow. One tester did suggest slightly more guidance could be useful. Other players seem to indicate that it was very easy to understand.

The time loop mechanic saw a somewhat mixed response some players liked it and found it straight forward to understand. Others found it hard or slightly confusing to learn from their previous mistakes.

Overall, all testers found the game engaging and it held their interest while playing. One player said, “The simple looping mechanics and easy visualizations of the end goal made the game into one of those addicting puzzle games…you spend hours on”. Although one test did find the game got a bit repetitive. One tester said “…often earlier levels felt too luck dependent…rather than skill or learning.” Suggesting that some parts of the game were more frustrating than other parts.

The responses on the difficulty on the game saw a very mixed response. Some thought it was too easy, adequate, or too hard. Some found some parts fine, other parts too difficult. Overall, much more testing seems to be needed here as most tester's probably only played a single level once i.e., they only played a single seat of the level. It seems that overall, a single level can have quite a wide possible difficulty due to the random nature of the levels. It can be learned from this feedback is that the games difficulty is not very consistent, and highly dependable on the random seed. Overall many players found the game frustrating at different points in the game. Some of this could be chalked up to the inconsistency in the game's difficulty. However, some players did note that in levels with obstacles it was not always clear to them how they died. Some players also found the jump to feel too floaty. Some players also found the fake platforms level frustrating and repetitive.

A mixed response for what players found to be their favourite part of the game. One player favourite part was the loop mechanic, another the movement, some the fake platform levels, others learning from their mistakes and applying them, one how quickly they could complete a level and another the simplicity of the game.

The game seemed to hold the interest of most players but that become repetitive after a while or too difficult in later stages.

Some players fully completed the game. Most players at least completed some levels in both stages while one player only played through levels in stage one.

Many levels stood out to different players in either negative or positive ways. One player noted that it was hard to distinguish one level from another. A few players indicated that they liked the combination of platforming and dodging of the blocks.

Overall, most players didn't see any major bugs some small ones were discovered and later fixed during the user test phase. But there were some noticeable bugs/exploits discovered. The pause menu could be abused to help complete a level. In the fake platform levels two platforms could be tested at once by jumping between them. The FOV slider would not always apply itself immediately and instead apply the change once the pause menu was closed. One player suffered a performance problem when the ghosts reached their maximum number. Player was able to jump over platforms by reaching a grounded state in the side of the platform. Some of these issues have been addressed and were fixed. However, they may still require sufficient user testing to ensure they are indeed fixed.

### 6.3.2 Input Device/Controls Questions

All players played with mouse and keyboard. Only one player attempted to use our controller. Players using mouse and keyboard suffered no issues. Two players indicated the jump felt a bit floaty. The one player who did use a controller noted that the analogue stick for the movement of the player did not work. All players found the controls easy and intuitive.

Overall, more testing could be done here especially for controllers since only a Nintendo switch controller was tested by a player and a problem was discovered.

### 6.3.3 User Interface Questions

Overall users found it easy to navigate the games user interfaces. All users found that the UI elements clearly communicated their function. Although one user indicated that the timer on the hunt was quite small and did not notice it until later levels.

Most users indicated no problems with the UI. One user found their cursor remained when they resumed the game, this problem is believed to be fixed in the final version since it was addressed near the start of user testing and no other users made mention of it. Another user as mentioned previously in section 6.3.1 phoned that when changing the field of view, it sometimes wouldn't apply until the pause menu was closed.

Most players wouldn't change anything with the UI or the layout however a couple users suggested adding a text scale option or larger text. Organ player did note the UI style was a bit plain.

### 6.3.4 Options Questions

Only two players customised their options or Ki bines at some point during their playtest. One experienced no issues the other experienced the FOV issue mentioned earlier in sections 6.3.3 and 6.3.1. The only thing noted by a tester that they felt was missing was an option for text scale and a colourblind mode.

Overall, the options could do with more testing to identify any anymore potential problems due to lack of users interacting with them.

### 6.3.5 Overall Thoughts Questions

There were many different suggestions from users of potential changes to the game or potential new features. Including more colour or textures, an option to crouch or slide, background music, more information on HUD or end screen like overall time to complete, more verticality in level design. One player suggested longer levels and the ability not to die in one hit. Overall, there was a lot of varied and different feedback here.

Most players indicated that they would recommend this game to others or would recommend this game once more features have been added.

### 6.3.6 Questionnaire Conclusions

Overall, the feedback received was positive indicating that the time look mechanic was used in interesting way. However, it did indicate that there are many aspects of the game that needs further testing or further polish.

Also due to the rather unfocused nature of the test and little direction provided to the testers. Certain aspects of the game such as the end game options and multiple input devices we're not thoroughly tested. Overall, much more data and testing need to be done in certain aspects of the game.

## 6.4 Observed Testing

For the two testers that decided that they wanted to participate in the observed testing. They were given the same information sheet, but a slightly different consent form. A time was arranged when they would be available to play the game. At the arranged time they met online with the student. The tester shared their screen while the student watched and recorded their gameplay.

The student encouraged the testers to think out loud while they played and noted anything of interest. The focus of this testing was to watch how they played the game and engaged with its systems. Once done they were instructed to fill in the questionnaire in their own free time like the other testers.

Both playtests were extremely useful and conveyed more information to the student than just the questionnaire. The students got to see what the testers were doing and thinking at different points during the playtest. Also, any minor bugs discovered could be noted down and fixed after they were done.

The only problem with this form of testing is it takes a great deal of time the more testers that you get. However, the information gained is very useful.

## 6.5 Evaluation

Overall, the main takeaways from the testing are that the loop mechanic and game is fun and engaging. However, the balance of the game is not consistent among the various level seeds and players playthroughs.

# 7 Description of Final Product

The Final product is a first-person 3D puzzle platforming game with a time loop mechanic. The game consists of twenty-four levels, a main menu and in game menu. The game supports mouse and keyboard, and gamepad controllers such as Xbox, Play Station etc.

The main menu consists of buttons that navigate to a level select and options menu and a button to close the game. The level select menu displays list of buttons for levels to load. However, the user will only be able to interact with levels that are currently unlocked. Only two levels are initially unlocked, one from each stage. With the rest being unlocked through completion of the previous level in the corresponding stage. Previously completed levels are marked as complete with a tick.

The options menu has settings for in-game options including volume, camera field of view, mouse sensitivity and toggles to invert camera controls. As well as button that navigates to a menu which allows for rebinding of keyboard controls. Which is done by clicking the corresponding rebind button then inputting the new key. Buttons to reset the key bind to the default key are also present. Any changes will be saved and loaded in when the game is relaunched.

Once a level is loaded, the in-game pause menu UI can be opened by pressing the escape key or the start button. This pauses the state of the game and opens a pause menu with buttons for resuming the game, restarting the level, opening the options menu, a button to quit to menu and to close the game. The options menu is the same as the one in the main menu, but the options are applied in real time. The game can be resumed by pressing the pause key again or by pressing the resume button.

The user is given a first-person perspective of a playable character which they can control by with inputs for movement, jumping and camera look. The player HUD is very simple only showing the time since the start of the current loop. If the player dies through collision with an obstacle or if they fall from a platform, the level is restarted from the beginning and all random factors are kept the same. So, the game events will play out in the same order as before.

In each level the goal of the player to reach a finish area by navigating through any obstacles in the way. Dying will reset the player back to the start and the level will play out the same as before. Allowing a player to learn from their mistakes and navigate through a level.

The twenty-four levels are split into two stages. The first stage all involves avoiding obstacles coming from the sides to reach the other side. The levels increase in difficult gradually getting more difficult, increasing the number of obstacles to avoid. The second stage of levels introduce platforms that need to be crossed to reach the other side. However, some platforms are fake and will lead to a player death. The later levels in stage two also in corporate the obstacle mechanic as well as the platforms. The levels gradually increase the difficulty by adding more platforms and reintroducing obstacles to avoid.

One a player reaches the end the level is complete, and a menu pops up allowing the player to view their current loop time and number of attempts. With buttons to load the next level, replay current level or return to menu and quitting the game.

# 8 Appraisal

## 8.1 Movement

The movement system was supposed to take about two weeks, the required time ended up being more like double this. Perhaps more if including later changes made in development. The entire system took longer than any other core features. The time to develop the movement system was overall underestimated as well as the constant fine tuning it required, to meet the students' requirements and intended game feel.

The student expectations for this section were based on prior work on experience in developing a 2D movement system. However, the student couldn't account for many problems that the extra access of 3D brought to the project. these unforeseen issues resulted in more time spent here.

A potential time save in this section would have been not to develop the slope movement related functions. The time spent developing sloped movement, ensuring it worked as intended, was time that could have been used elsewhere. Since there are no slopes in the final version. They would have been present in some of the moving platform levels. It would have been wiser to develop this feature when it was needed rather than along with the rest of the movement system.

However even though the movement system took much longer than expected, the result was very good. The student was overall very pleased with how the movement felt in the final build. User testing showed that the testers also felt this, the only thing negative mention was the jump feeling a bit floaty. The student also agreed with this feedback and planned to go back and adjust the player controller script to account for this. However, did not find the time.

## 8.2 Camera

A third person perspective was considered back in the design stage. But due to the student wanting first person perspective and underestimating the potential limitations of the first-person perspective. A third person perspective could make it easier for a player to see how they died. This wouldn't potentially be a huge task to undertake. Since a lot of the logic would be similar. The main changes would be in the Unity inspector itself such as the camera position and Cinemachine settings.

## 8.3 Levels

It could have been better to implement less levels but more variety. Resulting in a potentially shorter game experience but more mechanics and features that could have been potentially implemented in the same amount of time, being tested rather than different levels with the same mechanic. However, this would have resulted in a much shorter game with fewer levels which would have less levels for players to play through. So, the approach the student took may have been the better approach.

## 8.4 Balance and Difficulty

Many issues with the game, either through user testing or the student's own testing, were discovered too late in the project without reasonable time left to address them. The inconsistent game balance of obstacles especially in the harder levels when there was a large amount of them. Making them harder to memorise and predict.

A lot of time was spent implementing the multiple levels. But then even more time was spent trying to get the difficulty of the individual levels feel more balanced. Which due to the random nature of the game was somewhat unsuccessful in certain places. As some levels felt too easy, the difficulty curve spiked at certain points, the random generation was often too random where sometimes a level felt unbelievably easy where you could just randomly just first try walking a straight line and finish and other times it felt like the level would start off with a super high number of obstacles making it much too difficult. There was a large variance of difficulty in a single level. A more complex random algorithm that is centre skewed with less results on the outliers could have resulted in a more consistent level difficulty generation.

Due to the inconsistent difficulty some levels are basically fully implemented had to be cut due to them feeling too difficult. But if a lot of the previous mentioned problems with difficulty had been addressed and obstacles were much easier to protect/avoid. The time wasted on the cut content could have potentially been avoided.

## 8.5 Time Management

Due to falling behind schedule throughout the project, user testing started very late and unfortunately a lot of responses came in later than expected so very few of the feedback could not be taken on board to improve the game in time for the deadline. Most things that were identified from the play testing would have to be saved for future work. A second round of user testing to test new features and changes after the first round to test the new changes have also been extremely useful.

## 8.6 Prefabs

A few places where prefabs could have been used would have probably saved time as sometimes things had to change and helps for further expansion or changes in the future.

### 8.7 Ghost Mechanic

The ghost mechanic worked very well and accurately replayed the moves of the player’s last 10 attempts. It was generally well received but it was sometimes difficult to rely on harder levels. The student experimented with a delay on the ghost but could not decide if it made this problem easier or worse. Perhaps adding this the delay as an option in the options men then getting testers to see what they prefer could have been better.

### 8.8 Streamline Level Building

Another area where time could have potentially been saved was building the individual levels. Use of a Unity's package called “ProBuilder” was not investigated during the research process. Some point after the development of these two stages a way to speed up and streamline the process of building the level geometry was investigated and this tool was discovered. However, at this time the student had moved onto development of UI and never came back to implementing more levels. If the need for this a tool like this had been realised earlier on and research into it had been conducted during the background stage. It could have been potentially used to save time and perhaps more could have been developed.

Streamlining the positioning of spawners could also have been a thing that could have been investigated. This took a reasonable amount of time as they had to be placed manually are their positions specified manually. The Unity inspector did allow some streamlining as the same values could be applied to multiple objects at the same time. And multiple objects could be copy, pasted, or moved together. But in cases where their properties would be different. Perhaps a development tool could have been created which could create a grid of spawners objects by specifying the number of spawners, rows, and columns, spacing etc. Then just simply using this tool to create a grid of spawners then placing.

More importantly this would have saved a lot of time when fine tuning the balance of levels since many levels had to be a changed due to difficulty or other issues. However, development of this tool could take a significant amount of time and it might not be worth it in the short term. However, in the long term this would probably save a lot of time. Especially since many future levels would have used the same spawners that were used for the obstacles present in the game.

### 8.9 UI and Options

The student is overall very happy with how the UI, options menu, level select, and the saving of the data, turned out. All these systems appear to be very robust with very few glitches discovered through the user testing. Overall, the non-gameplay side of the game felt very polished.

### 8.10 In summary

Overall, the project went well, however the students underestimated many parts of the project and how long they would take. There was a lot more the student wanted to do with the project but did not have time for. If the student was to undertake the project again the student would probably do more user testing throughout the project instead of the student testing a lot the features themselves. Then the student could spend more time developing new features.

Although the student might not have achieved everything they wished to, what they did deliver was well received in the user testing. The movement system, options, UI, storing of data and the ghost mechanic were robust and polished and was well received by the testers.

## 9 Summary and Conclusions

In conclusion the project was a success. User testing shows that this novel approach on a time loop mechanic is interesting and can be a fun and engaging game mechanic. However, the final version of this game has some difficulty and balance issues.

Ultimately one of the biggest takeaways that the student has from the project is that they vastly underestimated the impact testing has on development and how's unforeseen consequences are unintended game effects can massively slow down the development of a feature.

Another takeaway is that trying to accurately predict how long something will take to develop in game development is extremely difficult. Even massive game studios struggle with this as games get delayed and realise in unoptimized states all the time. Overall, the student underestimated the scope of many parts of the game and could not predict some the problems faced during development.

However, they were successful in delivering a polished and optimised build of a short game featuring a time loop mechanic. With 3d platforming mechanics, options, and settings, a simple but effective UI, and multiple levels with progression and unlock requirements.

## 10 Future Work

There are multiple mechanics and levels designs that would have been developed if there was more time. This included multiple different ideas utilising new mechanics as well as some present in the final version.

For example, a level with randomly spawning moving platforms that must be traversed over or climbed up to reach the end. Multiple variations of this type of level could be created; some more horizontal or vertical, with different shapes and sizes of platforms, different total number platforms, different movement patterns of the platforms with the goal being different distances away from the player. The platforms themselves could also have hit boxes that could kill the player for example spikes on the ends. Which would cause the player to have to both plans where to move to progress but also must consider avoiding platforms as well.

Adding more levels would greatly improve the game as it would increase the length of the game, allowing players to get more from the game. One new time of level that could be added is a level were the goal is simply to survive for a set amount of time, while obstacle or enemies try to kill the player.

Adding enemy AI would be another great addition. For this, multiple new systems would need to be added for the AI, as well as systems for the player to interact with them. A complex predictable and consistent AI with a complex way for a player to interact with them like combat, was far too large for this project. But there are so many ways it could be implemented and could add a lot of addition playability and complexity to the game. If enemies were to be added combat mechanics for fighting close range and long range could be added as well as weapons for both the player and the enemies

The jump mechanic could be tweaked so that it feels less floaty and more responsive. If the student had more time, they would have reduced the max force that could be applied to the player object rigid body while airborne. Making it harder to change direction so quickly while airborne, giving the jump a more realistic feel. the Unity physics system also could be used to add drag to the player, to simulate air resistance.

A crouch and/or slide mechanic could be added which would give a player more traversal tools to tackle the different levels. This would be especially useful in dodging obstacle as the players hitbox could be shrunk to match the players new crouch/slide height. Along a player to slide under higher obstacles. Also, more levels could be design and developed around this mechanic.

Also improving the games visuals could be another avenue of future development. Could bring more colour and textures or nicer graphics to the game, to make the game more visually appealing. A different colour scheme or theme could be applied to the different stages, A unique visual identity.

More in game settings could also be added especially for features not developed like graphics. But also, many other settings could be added for example: resolution, frame rate, v-sync, controller remapping and many more.

Overall, there are many different possible avenues for future development.

## 11 References

[1]
J. Howarth, “How Many Gamers Are There? (New 2023 Statistics),” *Exploding Topics*, Oct. 07, 2022. https://explodingtopics.com/blog/number-of-gamers (accessed May 01, 2023).
[2]
D. Rubin and H. Ramis, “Groundhog Day,” *IMDb*, May 07, 1993. https://www.imdb.com/title/tt0107048/ (accessed May 01, 2023).
[3]
C. McQuarrie, J. Butterworth, and J.-H. Butterworth, “Edge of Tomorrow,” *IMDb*, May 30, 2014. https://www.imdb.com/title/tt1631867/?ref_=nv_sr_srsg_0_tt_8_nm_0_q_edge%2520of%2520t (accessed May 01, 2023).
[4]
“Outer Wilds,” *Mobius Digital*, 2020. https://www.mobiusdigitalgames.com/outer-wilds.html (accessed May 01, 2023).
[5]
“TWELVE MINUTES – An interactive thriller about a man trapped in a time loop.,” *Twelveminutesgame.com*, 2022. https://twelveminutesgame.com/ (accessed May 01, 2023).
[6]
“Home,” 2021. https://forgottencitygame.com/ (accessed May 01, 2023).
[7]
SUPERHOT sp. z o.o, “SUPERHOT - The FPS where time moves only when you move,” *SUPERHOT - The FPS where time moves only when you move*, 2023. https://superhotgame.com/ (accessed May 01, 2023).
[8]
B. Esposito, “Neon White - Out Now!,” *Neon White*, 2020. https://neonwhite.rip/ (accessed May 01, 2023).
[9]
“Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine,” *Unity.com*, 2023. https://unity.com/ (accessed May 01, 2023).
[10]
“Unreal Engine | The most powerful real-time 3D creation tool,” *Unreal Engine*, 2023. https://www.unrealengine.com/en-US (accessed May 01, 2023).
[11]
“Learn game development w/ Unity | Courses & tutorials in game design, VR, AR, & Real-time 3D | Unity Learn,” *Unity Learn*, 2023. https://learn.unity.com/ (accessed May 01, 2023).
[12]
“Page Redirection To Login,” *Unity.com*, 2023. https://api.unity.com/v1/oauth2/authorize?client_id=asset_store_v2&locale=en_GB&redirect_uri=https%3A%2F%2Fassetstore.unity.com%2Fauth%2Fcallback%3Fredirect_to%3D%252F&response_type=code&state=1c03d42b-1bcf-4122-a869-fffb7873023c (accessed May 01, 2023).
[13]
Unity Technologies, “Unity - Scripting API: PlayerPrefs,” *Unity3d.com*, 2023. https://docs.unity3d.com/2023.2/Documentation/ScriptReference/PlayerPrefs.html (accessed May 01, 2023).
[14]
Unity Technologies, “Unity - Manual: Prefabs,” *Unity3d.com*, 2023. https://docs.unity3d.com/2023.2/Documentation/Manual/Prefabs.html (accessed May 01, 2023).
[15]
Unity Technologies, “Unity - Manual: ScriptableObject,” *Unity3d.com*, 2023. https://docs.unity3d.com/2023.2/Documentation/Manual/class-ScriptableObject.html (accessed May 01, 2023).
[16]
Unity Technologies, “Unity - Scripting API: Input,” *Unity3d.com*, 2023. https://docs.unity3d.com/2023.2/Documentation/ScriptReference/Input.html (accessed May 01, 2023).
[17]
“Input System | Input System | 1.5.1,” *Unity3d.com*, 2023. https://docs.unity3d.com/Packages/com.unity.inputsystem@1.5/manual/index.html (accessed May 01, 2023).
[18]
“Cinemachine Documentation | Package Manager UI website,” *Unity3d.com*, 2017. https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/manual/index.html (accessed May 01, 2023).
[19]
Unity Technologies, “ECS for Unity,” *Unity*, 2023. https://unity.com/ecs#:~:text=ECS%20for%20Unity%20(Entity%20Component,level%20of%20control%20and%20determinism. (accessed May 01, 2023).
[20]
“Ghost vertices - Box2D tutorials - iforce2d,” *Iforce2d.net*, 2013. https://www.iforce2d.net/b2dtut/ghost-vertices (accessed May 01, 2023).
[21]
samyam, “Cinemachine First Person Controller w/ Input System - Unity Tutorial,” *YouTube*. Oct. 17, 2020. Accessed: Feb. 26, 2023. [YouTube Video]. Available: https://youtu.be/5n_hmqHdijM?t=833
[22]
“Footsteps - Essentials,” *@UnityAssetStore*, 2022. https://assetstore.unity.com/packages/audio/sound-fx/foley/footsteps-essentials-189879 (accessed Mar. 27, 2023).

## Appendices

[1] Honours Project Prototype.zip – This is the entire Unity project directory, including source code and all assets.

[2] Final-Build.zip – This contains the executable (Honours Project Prototype.exe) for running and playing the game on Windows.

[3] User-Testing-Responses – This folder contains an excel spreadsheet containing all the responses from the anonymous questionnaire. And contains a blank pdf of the questionnaire.

[4] Documents – A folder containing all the relevant document, including the Ethics Declaration Form, Risk Assessment Form, Mid-Term Report, Project Poster, Blank Consent Forms for both Observed and Non-Observed testing, the participant information sheet, and the original project brief.

[5] GitHub-Repo_Link.txt – A test document containing to the GitHub repository for the project.

[6] Flow chart – A folder containing a flow chart design of the movement system.

[7] Specification – A folder containing an incomplete software requirement document.