UNIVERSITY OF HERTFORDSHIRE

School of Computer Science

BSc Honours in Computer Science (Software Engineering)

6COM1053 – Computer Science Project

Final Report

April 2023

Polaris – Determining the Suitability of a Platformer Game Distributed on a PC Platform

A. Williams

19045324

Supervised by: Yonjung Zheng

Abstract

This project investigates the suitability of a PC platform for developing games, or specifically a platformer game developed using Unity and C#. The project explores a litany of research through a literature review necessary to support the project's hypothesis and its success, underpinning the areas of software engineering through game development.

Design and preparations for the project's artefact include materials such as paper design mockups, and class diagrams to assist in the artefact's implementation. Based on the design and preparations, the implementation will form results and conclusions relative to the development approach.

The project was ultimately successful, and a working artefact was produced. Changes and future considerations could be implemented to improve the system further, and the hypothesis for this project can be tested further if needed.

Acknowledgements

I'd like to thank my supervisor for giving me rational criticism and always pushing me to succeed in undertaking this project, even when it may have felt like an impossible task.

I'd also like to thank my parents, to whom I owe everything. Their support has not only guided me through my education but in helping me to become the best version of myself.

Table of Contents

List of Tables

Table of Figures

1.0 Introduction

Inspiration for this project came initially from previous experience developing video games as a hobby. The subject area of game development is an exciting and rich study area. It felt appropriate to combine experience in developing games with the skills and knowledge amassed throughout studying computer science.

In finding a hypothesis to test, it was baffling to see how many games are either developed console-first or seldom provide the same experience when played on PC. This observation provided the basis for the hypothesis. Essentially, in determining the suitability of a PC platform for a developed game and using the platformer genre as an example case to test its validity.

The project's main aim is to develop a solution with a simple design approach – by implementing a working foundation for improvements to be built upon iteratively. Several gameplay mechanics persistent in the platformer genre (such as running and jumping) will be part of the solution's core functionality.

The solution should also include PC-specific features, such as a control scheme that utilises the PC’s peripherals and conscious design decisions to maximise performance across the modular spectrum of PC hardware.

It’s important to note that suitability will not be measured in the gameplay experience alone, but also in the approach towards performance considerations.

This report will outline the development process, with preliminary research shown through a literature review. Sections for a project plan and further design materials outline the project's direction, whilst the implementation and testing sections show the outcomes of the artefact. Finally, two sections to evaluate the solution and conclude the project’s outcome.

2.0 Literature Review

The Literature Review aims to collate, explore, and outline the research relative to the topics associated with the project. The findings will be essential for developing a suitable artefact. From conducting this review, it's crucial to know how to approach a solution by understanding the underlying software engineering principles and other relevant topics. These topics include platformer game design practices, suitable programming languages, game engines, and existing solutions.

2.1 Mechanic Design

A fundamental part of game design is how the player interacts with the game. This interactivity is referred to as mechanics. However, while (Lim *et al.*, 2013) argue that there is no accepted definition, (Boller, 2013) supports this by saying, ‘A game’s mechanics are the rules and procedures that guide the player and the game response to the player’s moves or actions.’. It can be concluded that game mechanics are ambiguously defined and contentious. Concerning platformers, they typically all share Running and Jumping mechanics.

Running mechanics consist of the player’s input controlling a character's position. In their deconstruction of player movement mechanics (Brown, 2019) posits that running mechanics can be split into three parts: *acceleration*, *top speed*, and *deceleration*.

Acceleration asks, ‘When the player moves the input controller, how long is the climb towards top speed?’. Deceleration asks, 'How long after the player stops their input does the character return to a stationary position?’. These curves are important as they dictate how the character feels to move.

(Brown, 2019) continues to argue that steeper acceleration/deceleration results in more abrupt start/stop motions, as seen in games like Mega Man 9 (2008). However, longer acceleration/deceleration makes the character feel frictionless, akin to the character walking on ice, like in Super Mario Bros. (1985).

Jumping mechanics are a counterpart to movement mechanics. Both are fundamental principles of platformer design. (Brown, 2019) mentions that an effective way to measure a character's jump is by comparing how high a character can jump to their body size.

The curve of the character’s jump can be determined similarly to how the running mechanic was calculated by deriving three variables that make up the jump. Those are *climb*, *hang time,* and *fall*. (Brown, 2019) exemplifies this using games like Super Meat Boy (2010), where the character’s jump curve is tall, resulting in a ‘floaty’ feeling when jumping. In contrast, Unravel (2016) has a short jump, with little hang time, making the character feel heavy and leaden.

In summary, game mechanics is a non-descript term for interactivity, with contention of its definition. Every platformer shares two fundamental mechanics: running and jumping. Furthermore, the analysis (Brown, 2019) conducted in deconstructing them is insightful for understanding how they work and how they can be manipulated into providing a different experience regarding player input.

2.2 Level Design

The design of a level concerns the flow of gameplay rather than just where the platforms and obstacles are placed. (Thorson, 2019) likens this to a story the level tells through gameplay. They mention ‘every rectangle is a story’, relating to how each tile in the level can potentially change a player’s approach to that level.

Developers employ techniques such as grey-boxing to build levels quickly when designing levels. (West, 2021) explains that grey-boxing refers to prototyping a level with primitive shapes used as placeholders for game assets. This focuses on the core aspects of game design without being limited by asset production. The idea is, 'If you can make a game fun to play with just blocks, then it will have a solid foundation to build on’. This is especially important for the design approach I intend to take.

Grey-boxing typically refers to 3D-level design. However, the term applies to 2D also. (Thorson, 2019) mentions this in the design process for creating levels in Celeste (2018), documenting the process with screenshots demonstrating the level going from grey-box to fully detailed.

Another example is in more physical planning. In an interview discussing Super Mario Bros. (1985) (Miyamoto and Tezuka, 2015) mentioned using graph paper to design the levels. It would be fair to equate this to its own form of primitive level design, similar to grey boxing.

2.3 Unity Engine

This section evaluates the Unity engine to decide what engine the solution should be built in.

Unity is a game engine whose core elements (such as physics and animation) are written in C++. However, the code that users interact with and should be easy to write is written in C# (Brodkin, 2013). The focus on user development in C# makes Unity incredibly accessible in terms of coding, as things such as memory management are handled automatically (Chen, 2010) and are generally a more hands-off language.

Unity offers a flexible scripting API that uses the C# language. How the API functions are through a series of events, denoted in one of the articles as the ‘Order of execution for event functions’. A diagram illustrating this can be found in **Appendix B (pp. 63)**. Scripting components can be attached to objects in the engine, and applications will require scripts to respond to input from the player and to arrange for events in the gameplay to happen when they should (Unity Technologies, 2023b).

While (Unity Technologies, 2023c) argues that the default Update function is most commonly used in game scripts, they also contest that developers should use methods like FixedUpdate to handle physics as they share the same update frequency. When the

application runs at a lower framerate than the fixed timestep value, each frame's duration is longer than a single timestep (Unity Technologies 2023c). This is important for modelling accurate physics independent of the variable framerate the application might have on different systems.

As well as the core functionality, developers can use tools and external packages. The following consists of articles collated around tools and packages for Unity. These tools focus on level design to support the previous section by applying concepts such as grey-boxing.

ProBuilder is a set of level design tools for creating 3D Unity games that enable users to 'quickly prototype structures, complex terrain features, vehicles and weapons, or to make custom collision geometry, trigger zones, or nav meshes' (Unity Technologies, 2022). The most significant of those features is the ability to prototype levels. This would allow a more iterative process of level design to be achieved, which could be a benefit when developing the proposed solution due to the limited timeframe I have to work with.

This tool was used by (Shouldice, 2016) in designing basic geometry for Tunic (*formerly* Secret Legend) (2022). Similar inclusions of the toolset have been made in other games such as SUPERHOT (2016), Tacoma (2017), and Super Lucky's Tale (2017).

Tilemap provides similar functionality to ProBuilder, providing a flexible level designer for 2D Unity projects. It provides a grid where developers can 'paint' tiles onto the level directly (Unity Technologies, 2023d). Like ProBuilder, it can rapidly prototype levels incorporating grey-boxing level design.

The use of either would depend on developing a 2D or 3D game. Regardless, it reinforces Unity's ability to support both.

2.4 Comparisons to Unity

It is worth mentioning similar engines to compare and contrast them and possibly ensure that Unity may be the better choice.

The main competitor to Unity is Unreal Engine. It has been the source of a variety of successful projects, such as Fortnite (2017), Gears of War (2008), and Sea of Thieves (2018).

(Sanders, 2017), in their book 'An Introduction to Unreal Engine 4', describes its inception as coinciding with the creation of the first-person shooter Unreal (1998). The engine has been designed for first-person shooter games since its inception.

However, as (Sanders, 2017) continues to illustrate, subsequent releases of Unreal Engine recreated engine elements to make it available for a wider user than first-person shooters.

This included rebuilding the core code and rendering engines in Unreal 2 and developing a new lighting engine for Unreal 3.

(Tristem, 2022) argues that the core functionality is similar to Unity but highlights major differences. For one, Unreal uses C++ and a proprietary language, Blueprint (which provides more of a graphical interface to coding). Another is that Unreal allows for a much larger scale due to its support for distributed execution, resulting in higher performance than Unity. However, it would be fair to say the scope of the project I intend to develop is relatively small, and the scalability of Unreal may not be applicable.

Ultimately, Unity and Unreal Engine share a lot in common. The differences are in Unreal's focus on 3D game development and users' freedom to code at lower levels, thanks to Unreal's dependence on C++. However, my experience with C++ is a limiting factor in the decision to use it over Unity to develop this project. On the other hand, Unity proves more accessible using an object-oriented language like C#, with which I have prior experience.

The use of either would depend on the solution's requirements. However, in gauging the timeframe to complete the project, a 2D project would likely be the most achievable through Unity's flexible support of a 2D distribution over Unreal's mainly 3D focus.

2.5 Existing Solutions

This section collates articles relevant to existing solutions to the problem this project aims to solve.

One of the most popular examples is Super Mario Bros. (1985). The game's controls were briefly explained previously, in which (Brown, 2019) remarks that the character's acceleration/deceleration curve is long, which results in feeling like the character is running on ice.

The level design is an improvement from their previous work, Donkey Kong (1981). The game consists of eight worlds with four levels in each over Donkey Kong's one world, three levels. There are collectables in the form of coins, obstacles (such as spike traps and pitfalls), and enemies. This is impressive in being developed when video game technology was limited and primitive.

Compared to Super Mario Bros (1985), a contemporary solution like Celeste (2018) borrows from it but also brings new concepts not found in the former.

Celeste is a platformer developed by Maddy Thorson and Noel Berry. Remade from a PICO-8 game of the same name, the game's premise is more narrative-focused than the initial solution. (Thorson, 2019) mentions this in a conference on the level design of the game, describing how they deconstructed the game into areas (that serve as overarching

stories) and levels (that serve as short, quick stories). It shows how the game's level design is analogous to the overall narrative and how each level tells a story through its design.

The controls consist of a run, jump, climb, and dash, and the latter two recharge over time. The climb uses stamina abstracted from the player and slowly depreciates over time (Brown, 2019). The dash's recharge time is also abstracted. These controls allow traversal of the game's levels, which (Thorson, 2019) specifies that one of the underlying design decisions allowed the player to complete each level in multiple ways.

The importance of this section is in how learning by example can benefit game development. Games like Celeste (2018) contain mechanics with clear inspirations from other solutions that came before them, such as Super Mario Bros. (1985). To specific things like spike traps and pitfalls, as well as more abstract concepts like worlds that contain levels. Since its release, the developers made aspects of the code public (Thorson and Berry, 2019). This will be ultimately helpful in understanding their development approach.

Furthermore, this section overall has been useful in inspiring and informing about the development process. In addition, **Sections 2.1 (pp. 8) & 2.2 (pp. 9)** pair well with this section. Providing a cohesive understanding of the game development process will be useful to the project solution.

2.6 Review Summary

This literature review has helped to build a stronger understanding of the fundamental areas my problem explores.

Regarding game design principles, (Brown, 2019)'s findings on platformer mechanics, the importance of running and jumping, and the effect of tweaking their variables have put these mechanics at the forefront of my project going forward.

Furthermore, research into the specificities of game engines has reaffirmed my decision to build the solution in Unity based on accessibility and previous experience.

The existing solutions showed an interesting correlation between Super Mario Bros. (1985) and Celeste (2018). In both their design and their structure. Celeste adapts the 'worlds made up of levels' idea that Super Mario Bros. created.

In closing, this review helped to inform a solution to my project's problem. Celeste (2018) is a clear example of a game that fits the 'suitable PC platformer' mould, being available on PC with PC-specific considerations, such as keyboard-specific controls.

The most important part has been how the solution has been shaped by it. What was initially decided as a 3D platformer has now been narrowly focused on creating a 2D platformer. My conclusions are drawn from how 3D is typically more time-consuming and

include a third dimension to consider, unlike 2D. Ultimately, a 2D platformer produced in Unity will yield more substantial results than attempting to develop a 3D platformer.

3.0 Project Plan

The Project Plan will outline a plan for the solution's implementation, discuss the potential risks pre-implementation and post-implementation, and identify stakeholders involved in my solution. It will also use a Gantt Chart found in **Appendix A (pp. 62)**

3.1 Proposed Solution

The proposed solution will be to create a platformer game for a PC distribution. As per my findings in the literature review, the game will be developed using the Unity engine, consequently using C# as the given programming language. The game will be created in a 2D format instead of the initially proposed 3D format considering time management, reflecting the summary of the literature review and reaffirming the initial proposal's aim to make the game simple.

The development will focus on first getting the solution to a playable and testable state and then refining it through the testing outcomes. The literature review findings regarding game mechanics design emphasise these aims' importance. The running and jumping mechanics referenced in the literature review will make up the core functionality of the Player's movement implementation. Further mechanics, such as a dash and wall jump, will be derived from this implementation when refining the code.

The player will also use Unity's RidigBody2D component to simulate physics. This is supported by the literature review's findings that Unity uses fixed timesteps to simulate physics. This would mean that the solution would function relatively the same regardless of hardware, which is a concern when developing for a PC distribution.

Considerations have been made for more technical aspects of the solution. There will be seamless transitions between levels, where level loading/unloading will be handled through a level handler script. Due to the project's scope, individual level files seemed unnecessary and time-consuming when everything could be handled within one level file.

Beyond that, appropriate time will be allocated to design and detail levels, create interfaces such as menus, and develop art assets such as sprites and tilesets for the game if time allows. In addition, if there isn't sufficient time for asset production, free assets will be sourced and added to the game, or the solution will feature primitive shapes representing the game objects.

3.2 Preparations

The following section details the resources needed to develop the solution.

Paper

As part of the pre-production phase of development, a paper-based method will be used to illustrate early concepts of the game's level design. This was derived from **Section 2.2 (pp. 9)** in the literature review, showing that Nintendo developers used this to plan the game's levels. As a result, a similar approach seemed appropriate to draft the initial level design.

Git

Git will allow version control to be a part of the project by creating a local repository. Changes can then be recorded and rolled back, providing robust management tools for the project. Having previously used Git in academic and extra-curricular projects, I'm confident in how to set up local repositories and push changes to GitHub. GitHub will be used to host the repository and will be available at the following URL:

https://github.com/ascwnyc/Polaris

Unity Engine

Unity will be the engine the solution is developed in. This was decided in the summary of the literature review and reflected the engine's flexibility, as well as the accessibility it has over engines written in more complex languages such as Unreal Engine. With experience developing games as a hobby, Unity is familiar territory and uses object-oriented design in its engine, which is also an area I have academic experience in.

Tutorial projects are available inside Unity to inform me more about the engine. This will come in handy in furthering my understanding of the engine.

Extension plugins can speed up the development process by adding new features to the editor. These can be authored either by Unity themselves or by developers. As I lack experience developing these extensions, the project will use extensions developed by other users.

CineMachine

This is a free plugin available in Unity that offers greater control over the camera system. It will be used to follow the player primarily. Furthermore, its confiner component will allow me to set the bounds of the camera so it only shows the contents of the level the player is in. This should help in creating the seamless level transitions described previously.

C#

The project will be developed using C#. Considerations were made for using C++ and the aforementioned Unreal Engine for its versatility. However, due to the project's scope, it was decided Unreal would over-engineer it. C# is also a closer approximation to Java than C++, which is an object-oriented language that I'm familiar with from using in multiple academic modules. Having used Unity to produce games in the past, my experience using C# further ensures that it is a more appropriate choice over C++.

JetBrains Rider IDE

This is a cross-platform integrated development environment for coding in .NET languages, including C#. It has built-in support for Unity, making it a desirable choice of IDE for the project's coding. Like most modern IDEs, it supports Git's version control through a GUI.

In terms of costs, the IDE requires a paid-for license. However, this is not required if the user is a student.

Level Design

The project will be developed using the level design techniques outlined in the Literature Review. Previous experience developing games and subsequent skill in designing levels for those games will ensure that the level design is adequate and designed gameplay-first.

As well as this, Unity includes object prefabbing (for creating many entities from a template) and the Tile Mapping component to handle tileset assets such as the ground, hazards, and obstacles.

Assets

The project's assets (sprites, tilesets, and graphical user interface elements) will be created using Aseprite. Aseprite is a graphics-manipulation program built to create sprites and tileset graphics. The program does cost £15.49 to purchase, but as I already own the program, there will be no cost incurred for the development of the artefact.

Considerations were made to source royalty-free/open-source assets for the project due to time constraints. Time will be allocated for asset production, which would normally be accounted for during commercial game development and exemplifies the typical game development process.

Furthermore, with experience developing 2D games, the time allocated should be more than enough to develop the assets for this project.

If additional assets are required to complete the project, they will be sourced from free asset packs available on the Unity Asset Store.

3.3 Stakeholders

This section will outline the various stakeholders involved in the project's outcomes.

- **User** – The user or ‘end-user’ is a stakeholder involved in playing the game produced from the project development.
- **Project Supervisor** – The supervisor for this project will be a stakeholder in the sense that they will be involved in reviewing the supporting documentation for the project (i.e., the report and code).
- **Developer** – The developer for this project (myself) is also a stakeholder. The developer will oversee the development process and test and maintain the produced game.
- **Project Manager** – The project manager is a stakeholder responsible for time management and planning. This will be a role also assumed by myself.

3.4 Risk Assessment

This section will give an overview of the risks during and following implementation, presented as tables. The likelihood of a risk occurring will be measured as Low/Medium/High.

Risk	Likelihood of Occurrence	Consequences of Occurrence	Mitigating Actions
The project fails to meet deadlines or is unfinished.	Medium	Failure of the project module, less time to produce a report.	Plan each stage of development thoroughly.
Too much time is spent developing assets (sprites, tilesets, GUI elements).	Medium	Wasted development time, the foundations of the project aren’t substantial, and there will be less to write about in the report.	Develop basic systems first, and add extra detail later. Source free assets from Unity Asset Store.
The project lacks a definition or documentation of the scope of its requirements.	Low	Project development becomes uncontrolled and has too many unnecessary features (scope creep).	Clearly define the project’s requirements, ensure a basic prototype before developing further, and tackle the requirements in order of importance.
The project has compilation errors, bugs etc.	High	Less time spent developing the project and more time spent fixing these problems.	Allocate sufficient time in the project plan for testing.
The project incorporates new features not defined		More time will be spent implementing features not agreed upon, and less time	Clearly define what is required of the project implementation, and

in the requirements after the requirements have been satisfied (gold-plating).	Low	will be spent on testing the project. Technical issues can arise from implementing more features for little gain.	ensure any extra features support the existing features and are added with reason.
Lack of effective time management leads to consistently long work days (commonly called crunch).	Medium	The developer ends up not being able to work to the best of their abilities, and there isn't enough time to implement all of the proposed requirements of the project.	Ensure the management accounted for the appropriate time to finish tasks. This is one of the most common reasons for crunch in the games industry.

Table 1 – Risks and mitigation during project implementation.

Risk	**Stakeholder Affected**	**Likelihood of Occurrence**	**Consequences of Occurrence**	**Mitigating Actions**
The project contains features not defined in the project requirements after the requirements have been satisfied (gold-plating).	User, Project Supervisor	Low	The project fails to resemble what is decided in the requirements and is not the game the user wants. This also goes against what was agreed between the developer and project supervisor.	Scale back development scope so that new iterations of the game ensure what was outlined in the requirements is present. Ensure extra features support the requirements rather than creating new ones altogether.
Problems during development resulting in added workload to patch the project post-release and longer work hours (crunch)	User, Developer, Project Manager	Medium	This affects the user's perception that the released project is unfinished and not worth the money (if the project was sold for profit). Added stress on the developer to address these issues that weren't accounted for during development.	As this is usually a problem at a management level, proper planning and time management are the best solutions to mitigate the problems of strained development.
The project uses assets that, upon further inspection, are not royalty-free or free to use.	Developer	Low	The developer is required to remove the assets or is sued for copyright infringement	Ensure a thorough investigation is made into where all assets are from and whether they can be licensed in the project.

Table 2 – Risks and mitigation after project implementation.

3.5 Statement of Success

Putting aside the artefact's creation, the importance of learning from my experiences and measuring the success of academic achievement throughout the project is paramount. With the project putting me in a considerably real-world scenario, it will assess my approach to this issue and what can be gained from it.

The project will be useful in looking into the software development cycle as a whole and using the waterfall method through a Gantt Chart to plan the project. Having more experience using the agile methodology during a group project in the previous academic year, it will be interesting to see how the two differ. As well as bolstering knowledge of both methodologies.

Through the use of .NET languages like C#, object-oriented programming is a fundamental requirement for developing this project. If successful, my understanding of object-oriented design will be reflected in the project's artefact.

The importance of this project towards my final grade has helped me focus and complete the project to the best level possible. While I seldom take a serious approach to any project I develop, this seems like a great opportunity to demonstrate how I can dedicate time to something important and hopefully develop a new approach to future project.

4.0 Design

This section shows the early development stages of the project. It contains design information in the form of level design concepts and class diagrams that were used as a foundation for the solution to be built from.

4.1 Paper Concept

The paper concept was intended to design how the game would look visually. This was an approach taken from an article in the literature review in which Nintendo demonstrated using graph paper to design levels for Super Mario Bros. (1985). An example of the early level design concepts is shown in **Figure 1**.

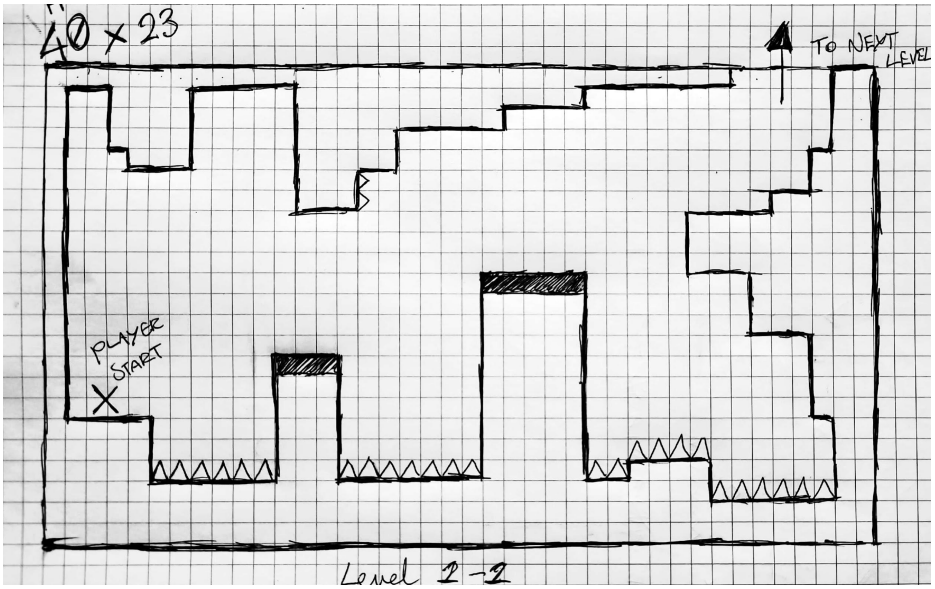


Fig. 1 – Paper diagram of initial level design concept.

The general findings of this phase were that drafting level design was a simple and effective way to quickly produce level design concepts for the project. While these concepts were less detailed than Nintendo's implementation, the purpose was to provide primitive design ideas for levels, which was achieved. Furthermore, when developing the levels in Unity meant there was a wealth of design ideas to choose from.

Additional design concepts can be found in **Appendix C (pp. 64)**

4.2 UML Class Diagrams

These diagrams will show how I plan to implement the individual objects that make up the game. The Unity documentation will be used as a reference to convey the system's logic. All classes in Unity are represented as objects or components, so the class hierarchy illustrates inheritance between the classes. For brevity, only the relevant properties/methods have been shown.

Player

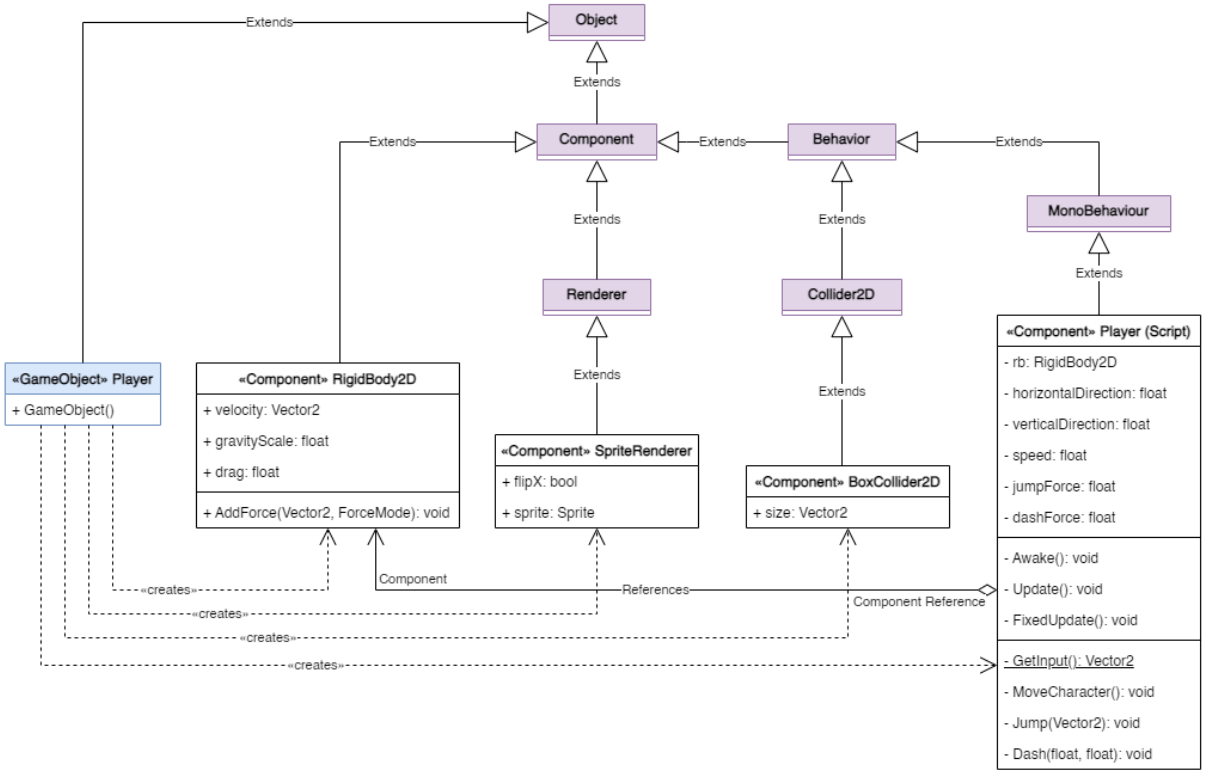


Fig. 2 – UML Class Diagram for the Player GameObject.

Figure 2 shows that the player object will comprise several components that modify the object's behaviour. For example, the `RigidBody2D` component will allow control of the object's position through physics simulation.

The Player (Script) component will influence object behaviour through C# code. **Figure 2** also shows several properties relating to movement, which will be used in the `MoveCharacter()`, `Jump()`, and `Dash()` methods. These will provide some of the mechanics identified in **Section 2.1 (pp. 8)** of the literature review.

This component will also have to reference the RigidBody2D to make movement possible. Because the script exists as a component, references must be made to other components to invoke their methods and affect their properties.

LevelHandler

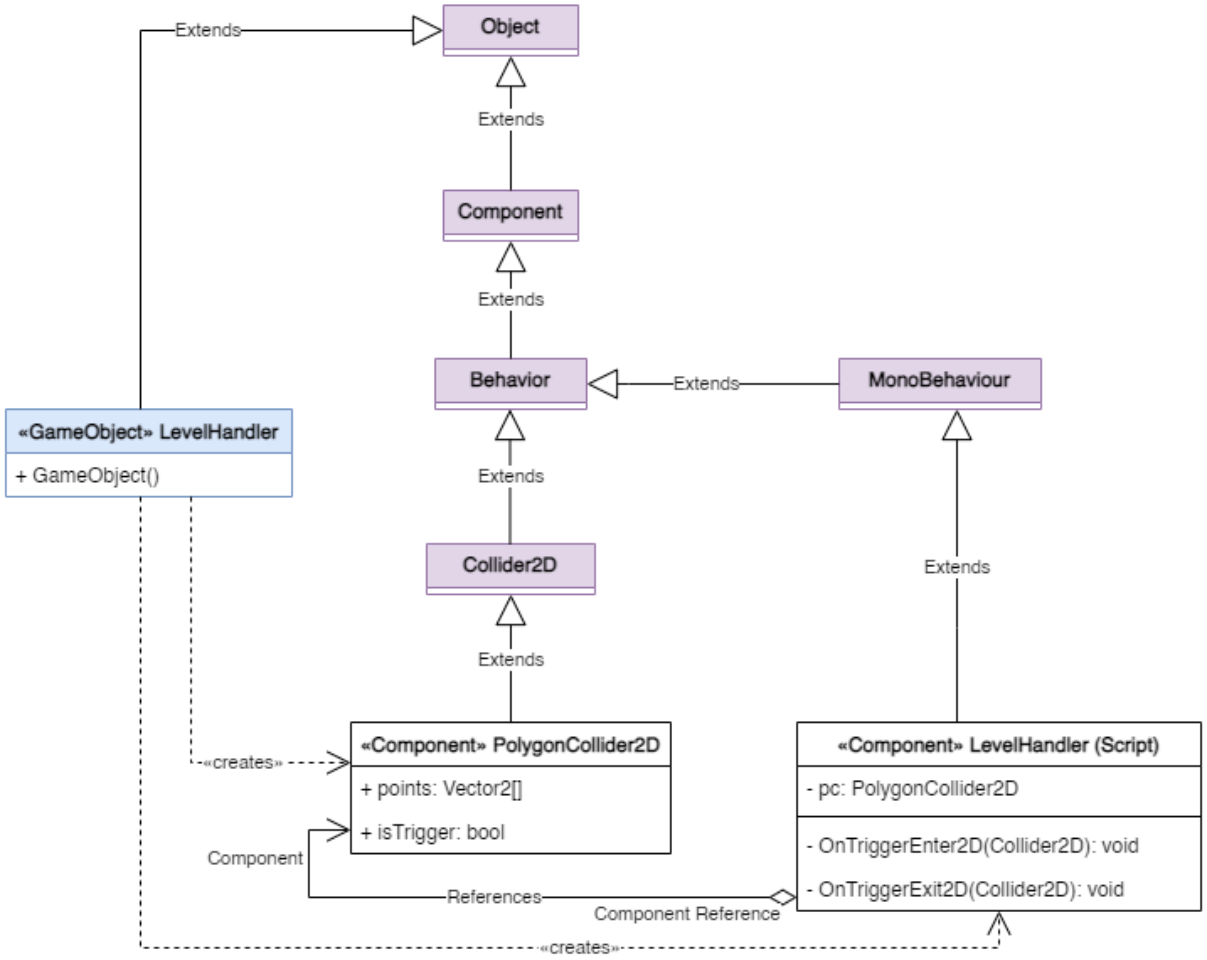


Fig. 3 – UML Class Diagram for the LevelManager GameObject.

The LevelHandler will feature a script that controls the level using a `PolygonCollider2D` component. The `PolygonCollider2D` component has a property '`isTrigger`', which, when set to true, will allow the LevelHandler script to control what happens when the player enters and exits the `PolygonCollider2D` component. This will support loading and de-loading levels based on the level the player is currently in.

5.0 Implementation

The following section will detail the project's development. It will highlight the relevant areas, using figures of screenshots, code, and supporting information to illustrate the development process.

The first step in creating the artefact was the setup and configuration. This was done by creating a new project in the Unity Hub application. New projects can be built from a variety of templates. For the artefact, the 2D (URP) template was chosen, as shown in **Figure 4** below.

This provides a blank project with a renderer pre-configured for use with Unity's universal rendering pipeline. This was chosen over Unity's built-in renderer as the documentation suggested the URP is more flexible for developing graphics across many platforms (Unity Technologies, 2023e). Despite the project's focus on PC distribution, it seemed the right choice for future extensibility.

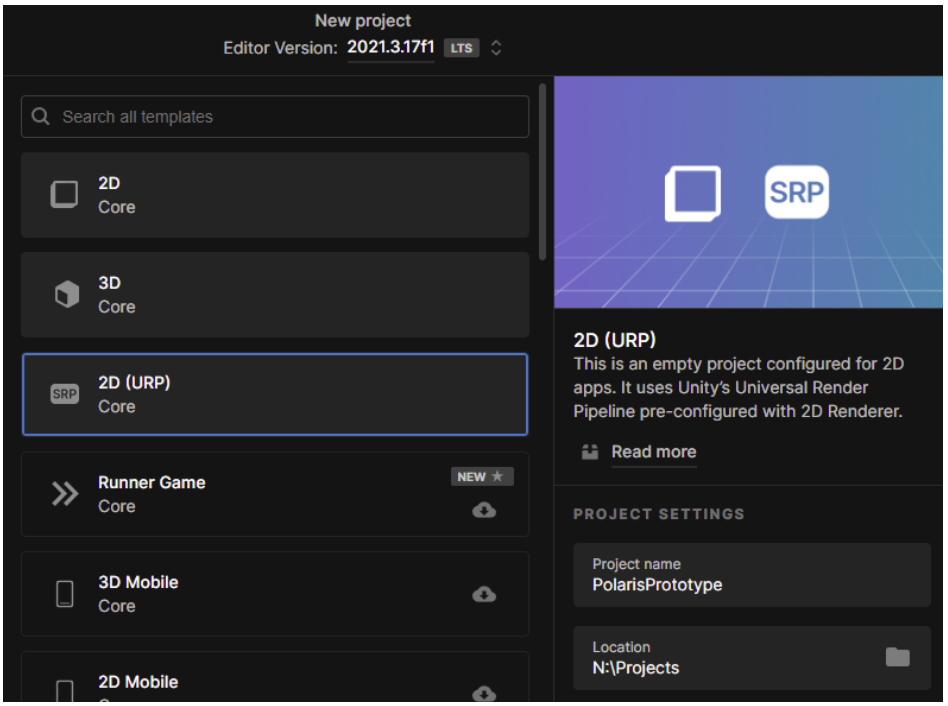


Fig. 4 – Creating a new project in Unity Hub.

The first decisions made about the solution were in its presentation. The resolution was decided to be 320x180. This was because it was a factor of the 16:9 aspect ratio that most modern computer monitors share. The lower base resolution was also intentional, making asset production easier by working with less space (8x8 tiles etc.) and being infinitely scalable to the aspect ratio.

There was also some configuration of the Input Manager, which handles keybindings and general input from the user. The following figure shows the configured WASD movement

keys and some secondary input keys. Detailed screenshots of the configuration can be found in **Appendix D (pp. 65)**

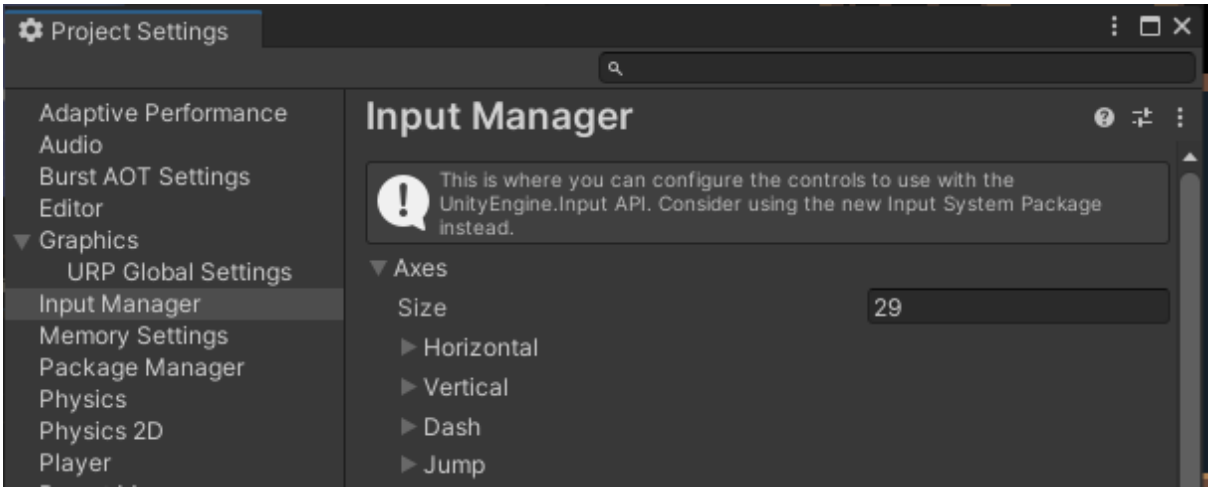


Fig. 5 – Input Manager showing primary and secondary input entries.

Following the initialisation, work on creating the basic GameObjects required for the artefact to function can begin. As defined in **Section 4.2 (pp. 20)**, these are the *Player* and *LevelManager* objects. Additionally, a Tilemap object will be required for designing levels.

5.1 Player Initialisation

The Player game object requires *RigidBody2D*, *BoxCollider2D*, *SpriteRenderer*, and *Script* components to function. **Figure 6** shows the Player GameObject in the inspector view and the assigned components. The Script component is perhaps the most important, as it instructs the behaviour of the Player object and its components.

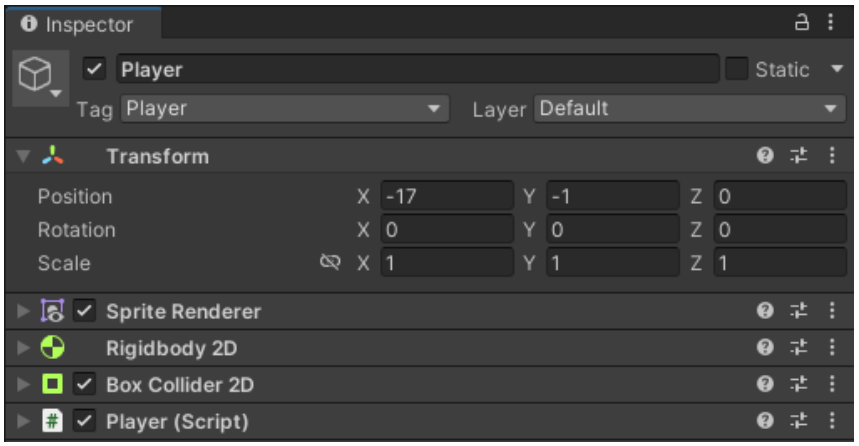
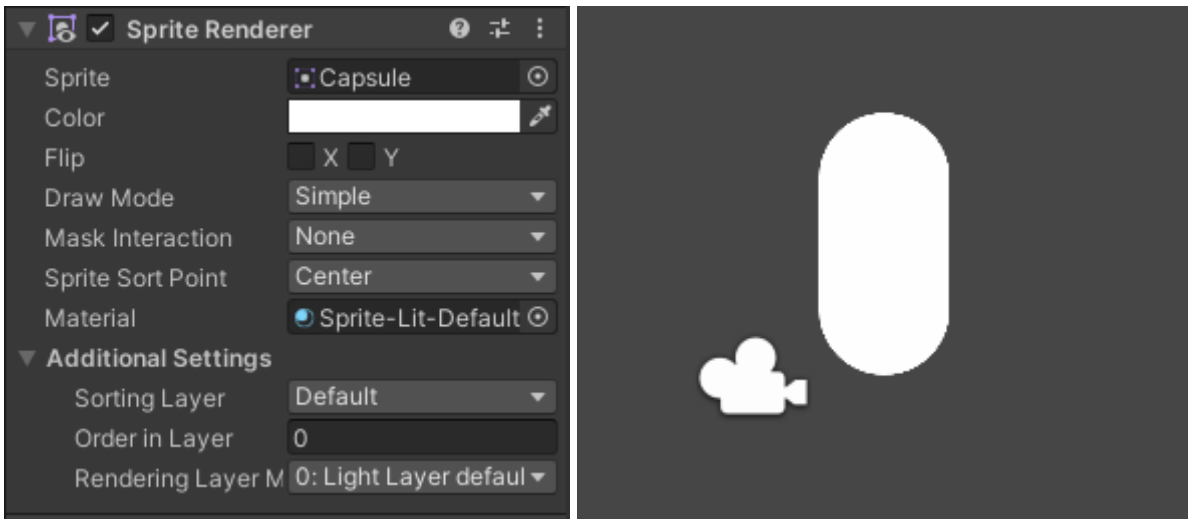


Fig. 6 – Screenshot of Player GameObject in the Inspector.

As the inspector abstracts a component's code from the developer, the configuration of these components mostly dealt with assigning values and references to the component's properties.



Figs. 7 & 8 – Configuring the SpriteRenderer (left). Resulting sprite in the Scene View (right).

Firstly, the SpriteRenderer needed to be configured to visualise the object better. **Figure 7** shows the SpriteRenderer in the inspector. A capsule sprite has been selected as a placeholder. Later in development, this will be replaced with the player sprite. **Figure 8** shows the results of setting the Sprite property in the scene view. The simplicity of this means that the capsule sprite can be replaced with custom graphics later in development without hassle.

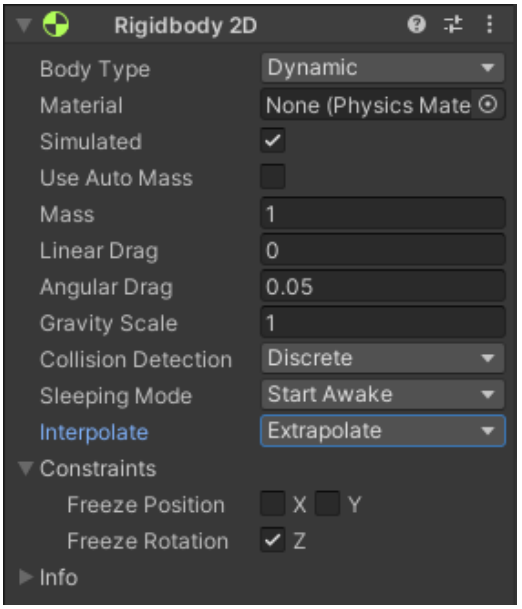


Fig. 9 – Configuring the RigidBody2D component.

The RigidBody2D component required little configuration (see **Figure 9**). The Interpolate property has been set to Extrapolate, which was needed to smooth out the object in motion. This is because the physics in Unity runs in discrete time steps, while graphics are rendered at a variable frame rate, causing the object to have jittery motion due to the physics and graphics not being entirely in sync (Unity Technologies, 2023f). The Z-axis also has its rotation frozen. This will stop the object from falling over when too much force is applied on the X-axis (i.e: high friction values).

Figs. 10 & 11 – Configured BoxCollider2D component (left). BoxCollider2D size properties in the inspector (right).

The BoxCollider2D component provides collision detection for the object. For the Player object, the collider's bounds fit the sprite's dimensions. **Figure 10** shows this as a green rectangular outline around the object, and the collider's size is shown in the inspector in **Figure 11**.

The Script component didn't need configuration like the other components, as the developer defines its properties. The class diagram in **Figure 2 (pp. 20)** referenced the methods required for the player's movement. The following section will show how that was implemented in C#.

5.2 Primary Mechanics

```
private void Awake()
{
    rb = GetComponent<Rigidbody2D>();
}
```

Fig. 12 – Creating a reference to the RigidBody2D component.

As stated in the **Section 4.2 (pp. 20)**, components require a reference to be modified by scripts. The code above stores a RigidBody2D reference into a `private RigidBody2D` variable named '`rb`' using the `GetComponent<>();` method. It's declared in the Awake method, as according to **Figure 12**, it is the first method that runs in a Unity class, making it ideal for initialising.

```
private static Vector2 GetInput()
{
    return new Vector2(Input.GetAxisRaw("Horizontal"),
    Input.GetAxisRaw("Vertical"));
}
```

Fig. 13 – Creating a method to obtain the user's input.

Next, it was important to have access to the user's input to create methods for moving the Player object. **Figure 13** shows a method which returns a 2-dimensional vector of the user's

horizontal and vertical inputs. The `GetAxisRaw()` method normalises a user's WASD input to a magnitude of -1 to 1. For example, if the user presses the D (right) key, the game will register that as a positive x value.

```
private void Update()
{
   horizontalDirection = GetInput().x;
   verticalDirection = GetInput().y;
}
```

Fig. 14 – Invoking the `GetInput()` method in update.

These inputs can be stored in the relevant variables and called within the Update method. This is shown in **Figure 14** above. Following this, physics can be attached now that the script checks for the user's input.

```
private void MoveCharacter()
{
   rb.AddForce(new Vector2(horizontalDirection * speed, 0),
   ForceMode2D.Force);
}
```

Fig. 15 – Creating a method to move a character using physics.

In **Figure 15**, the `MoveCharacter()` method has been implemented. The code shows the RigidBody2D reference to invoke the AddForce method, multiplying the relevant input direction with the `speed` value. The `speed` variable will allow me to control how fast the object should move. The method is called the `FixedUpdate()` method, as it deals with physics simulation.

The jump was implemented similarly, using the RigidBody2D's `AddForce()` method in a given direction.

```
private void Jump(Vector2 dir)
{
   rb.AddForce(dir * jumpForce, ForceMode2D.Impulse);
}
```

Fig. 16 – Code for the `Jump()` method.

As shown in **Figure 16**, the `Jump()` method passes the direction of the jump as a Vector2 parameter and applies force in that direction. This was achieved similarly to the previous implementation by multiplying the direction by the `jumpForce` value.

Because the `Jump()` method doesn't explicitly state the direction parameter as upwards, the method is flexible and reusable later in development.

Gravity also had to be accounted for. Without applying a higher force of gravity on the jump's descent, it caused the fall time to be exceptionally long and unrealistic. To remedy

this, a `FallMultiplier()` method was created to handle gravity. The following figure shows its initial implementation.

```
private void FallMultiplier()
{
    if (rb.velocity.y < 0)
    {
        rb.gravityScale = fallForce;
    }
    else
    {
        rb.gravityScale = 1f;
    }
}
```

Fig. 17 – Code for the `FallMultiplier()` method.

While the value of the player's velocity on the Y axis is less than zero (meaning that the player is falling), the gravity scale should increase. Once a jump is complete, the gravity should reset so the player doesn't face inconsistent gravity when jumping a second time. This method is then called in the FixedUpdate(), as it manipulates physics components.

The `Jump()` method is called within the Update() method, not the `FixedUpdate()` method. This is because input data is determined for each frame, but `FixedUpdate()` uses fixed time increments. This would cause an inconsistent input response. As discussed, a conflict exists with ensuring physics calls are made using FixedUpdate.

A solution could be to cache the input in `Update()` and handle the physics calls in `FixedUpdate()`, which is mentioned in **Section 6.2 (pp. 46)**.

```
private void Update()
{
   if (Input.GetButtonDown("Jump"))
   {
      Jump(Vector2.up);
   }
}
```

Fig. 18 – Implementing `Jump()` into the `Update()` method.

In **Figure 18**, the condition is checked, and if true, invokes the `Jump()` method in an upwards direction. The condition has to be specified as `GetButtonDown` because the jump should occur once per key press, and simply using `GetButton` would continuously check for the corresponding input.

It's worth briefly mentioning the use of the Input Manager here. The `"Jump"` string corresponds with an Input Manager entry. This avoids hardcoding the input key and allows for versatility in modifying the input (i.e., through re-bindable controls).

With this, the initial mechanics have been implemented. The following section will detail the implementation of secondary mechanics such as the Dash and Wall Jump.

5.3 Secondary Mechanics

For the Wall Jump, it was important to understand the logic of its implementation. From a jump, when the player touches a wall and then hits the jump key again, the player should jump in the direction opposite the wall.

To achieve this, there first needs to be a condition check for if the player is touching a wall. This can be done using OverlapCircle colliders that detect when a given collider enters its radius.

```
private void CheckCollisions()
{
   var position = transform.position;

   isWalled = onRightWall || onLeftWall;

   onRightWall = Physics2D.OverlapCircle((Vector2)position + rightOffset,
   collisionRadius, wallLayer);

   onLeftWall = Physics2D.OverlapCircle((Vector2)position + leftOffset,
   collisionRadius, wallLayer);
}
```

Fig. 19 – Collision detection code for wall jumping.

In **Figure 19**, the `CheckCollision()` method creates two OverlapCircle colliders named onRightWall & onLeftWall. Their positions are determined by the sum of the collider's position and the respective offset values. Along with the radius, these values are stored in variables for flexibility in adjusting them. Finally, the collider type they're checking against is any object with the Wall layer attached.

In Unity, layers and tags are a system that can be used to group objects together and have them referred to in code. When invoked in `FixedUpdate()`, this code enables collision detection for the player touching the wall.

```
private void Update()
{
   if (Input.GetButtonDown("Jump") && isWalled)
   {
      WallJump();
   }
}
```

Fig. 20 – Jump input implementation in the Update() method.

With the collisions implemented, this can be checked alongside the jump key being pressed in `Update()`. This figure shows a conditional statement asking if the jump key is being pressed *and* the player is touching a wall.

The implementation of the physics is found in **Figure 21**. The `onRightWall` variable is reused here to determine the direction of the jump depending if the player is touching the right wall or not.

```
private void WallJump()
{
    wallDir = onRightWall ? Vector2.left : Vector2.right;

    Jump(Vector2.up / 1.5f + wallDir / 1.5f);

    wallJump = true;
}
```

Fig. 21 – Code for the `WallJump()` method.

The `Jump()` method was reused in implementing the Wall Jump. The direction of the jump combines a `Vector2.up` with the value of `wallDir` so that the player jumps diagonally. The values are divided by 1.5 to lessen the power of the jump as it applies the value of `jumpForce` on both the up and left/right vectors.

Firstly, a boolean to determine if the player has dashed that will be set to true when a dash occurs. Secondly, a boolean to determine if the player is dashing which will be set to true while the dash is occurring.

The method takes two inputs which during this point in development was the horizontal and vertical directions mentioned in the `MoveCharacter()` method implementation. These inputs are used to determine the direction of the dash. The dash direction is then normalised, meaning its value can be of a magnitude between -1 and 1, and then multiplied by the value of `dashForce`.

```
private void Dash(float x, float y)
{
    hasDashed = true;

    rb.velocity = Vector2.zero;
    dashDir = new Vector2(x, y);

    rb.velocity = dashDir.normalized * dashForce;

    StartCoroutine(DashWait(.15f));
}
```

Fig. 22 – Code for the `Dash()` method.

This method also marked the introduction of coroutines. They can be used to create pauses in execution. In this instance, a coroutine has been used to handle the time spent in the `isDashing` state before the player can regain control.

```
private IEnumerator DashWait(float time)
{
    rb.gravityScale = 0;
    isDashing = true;

    yield return new WaitForSeconds(time);

    isDashing = false;
}
```

Fig. 23 – Code for the `DashWait()` coroutine.

Setting `gravityScale` to 0 was important here, as the player shouldn't be affected by gravity while dashing. `isDashing` is also set to false after the coroutine's pause has finished ensuring the player doesn't perpetually dash.

5.4 Level Handler

The Level Handler was a proud moment in development as it demonstrates how easy it can be to create a powerful system. As **Section 4.2 (pp. 20)** mentions, the LevelHandler object would include a PolygonCollider2D and a LevelHandler script component.

The implementation would then use the OnTriggerEnter2D and OnTriggerExit2D methods to handle loading and de-loading level objects. According to the Unity documentation, the triggers are sent when another object enters/exits a trigger collider attached to this object (Unity Technologies, 2023g).

```
public class LevelHandler : MonoBehaviour
{
    public GameObject vc;
    public GameObject obs;
    public GameObject cp;

    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player") && !other.isTrigger)
        {
            vc.SetActive(true);
            obs.SetActive(true);
            cp.SetActive(true);
        }
    }
    private void OnTriggerExit2D(Collider2D other)
    {
        if (other.CompareTag("Player") && !other.isTrigger)
        {
            vc.SetActive(false);
```

```
            obs.SetActive(false);
            cp.SetActive(false);
        }
    }
}
```

Fig. 24 – Code implementation of the `LevelHandler` script.

The implementation was similar, except Unity's tag feature profiles objects with the 'Player' tag as the object to track. This meant that when an object with that tag leaves the collider's space, the other objects in the level are disabled, and vice versa.

What was useful about this implementation was its extensibility. If new object types need to be added (i.e., Enemies or collectable items), they can be declared as a `public GameObject`, and references can be assigned in the editor, as shown in **Figure 25** below.

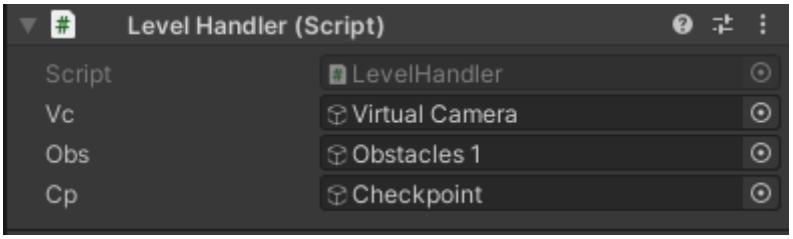


Fig. 25 – Assigning object references into the Level Handler script.

5.5 Levels

The level design was implemented using Tilemap, a tool mentioned in the literature review. The tile graphics' creation and implementation are discussed further in **Section 5.8** below.

The levels were initially designed using the mockup designs discussed in the **Section 4.1 (pp. 18)**. Initially, the tiles used were very primitive to coincide with the grey boxing concept introduced in the literature review, and an example of that can be found in **Figure 26**.

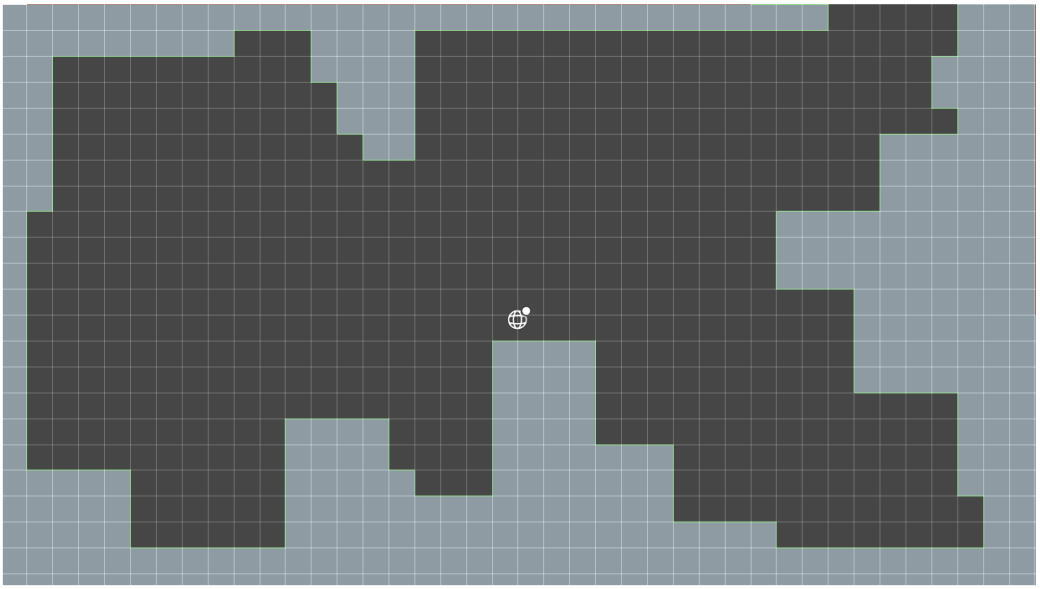


Fig. 26 – Screenshot of early grey-box level design.

As the approach to level design that was chosen was iterative, the levels went through several changes based on playtesting. The graphics also improved, from single-colour to multi-coloured tiles to denote the different terrain (See **Figure 27**).

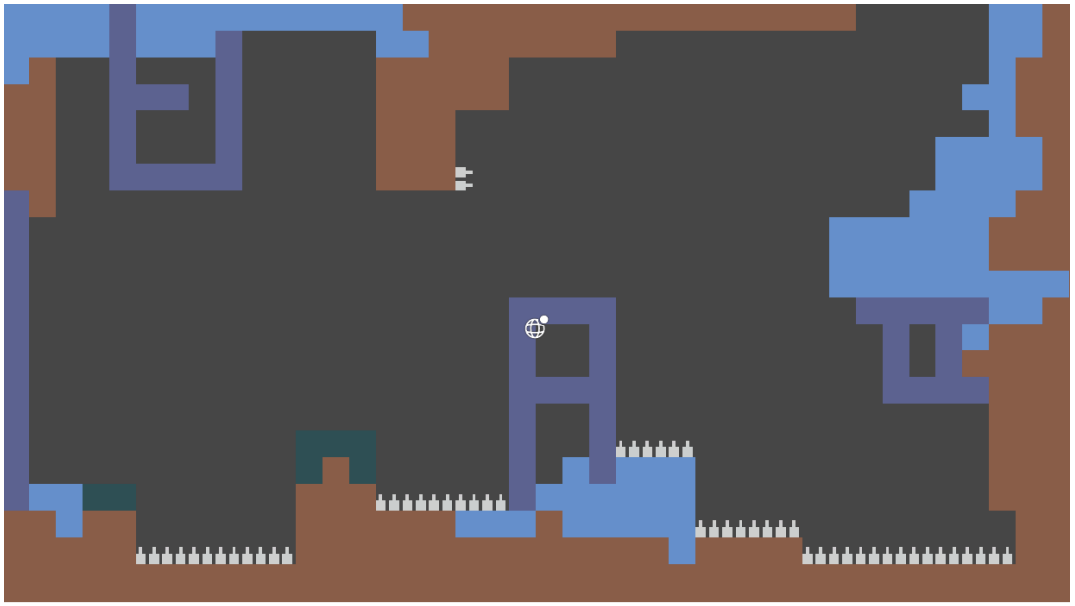


Fig. 27 – Screenshot of the tweaked level design following playtesting.

Basic obstacles were also added as development progressed. They became a particularly strong part of development by adding a sharp difficulty curve to the levels.

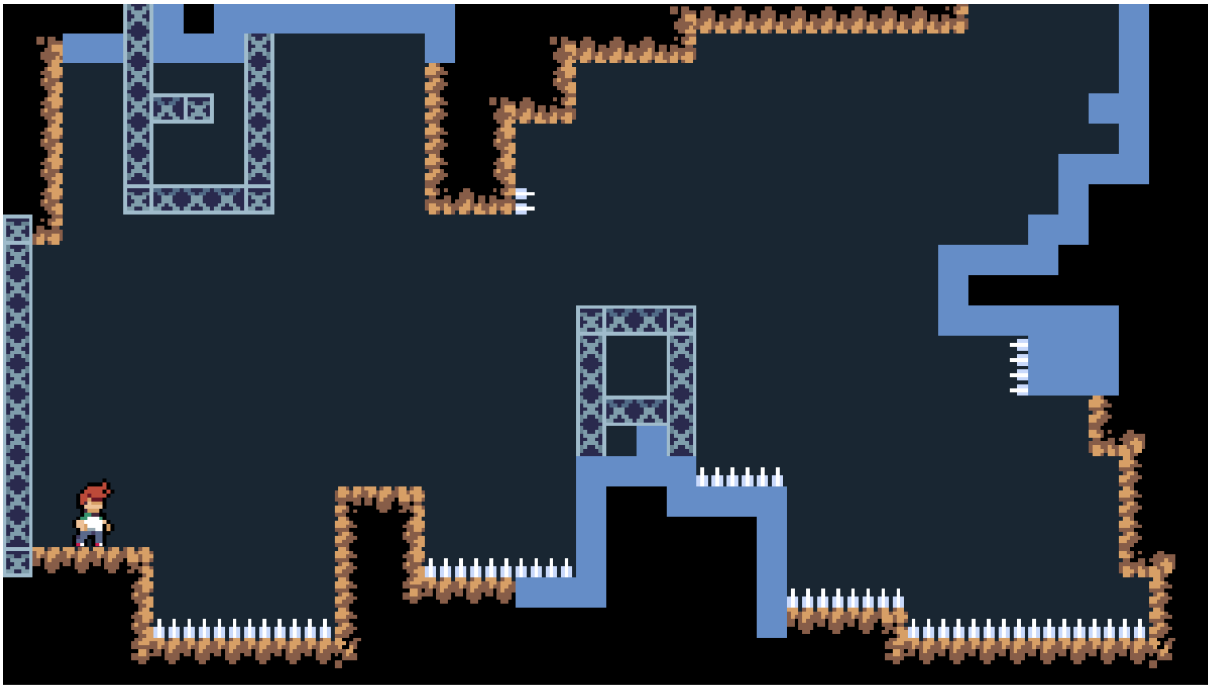


Fig. 28 – Screenshot of the close-to-complete level design.

Figure 28 shows the close-to-finished level, following improvements to the tile graphics and refinements to the level design.

Screenshots of the finished levels can be found in **Appendix E (pp. 66)**.

5.6 Respawning

In **Section 5.4 (pp. 30)**, the Level Handler touched on objects in a level, such as the obstacles and checkpoints. This coincided with the implementation of player death, respawn, and checkpoints. Logically, when a player touches an obstacle (like the spike traps seen in the previous section), they should die or receive a game over, and a short time afterwards, respawn at a checkpoint.

To code this, a similar approach can be made to how the LevelHandler dealt with level loading/unloading. Using the `OnTriggerEnter2D()` method in the Player class, it can check for when the player collides with an obstacle. Using a conditional statement, if the object the collider belongs to has the “Obstacle” tag, the death state will initiate. The code for this is shown in the figure below.

```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.tag == "Obstacle")
    {
        // Check for Respawn Point
        while (!cp.activeSelf)
        {
            cp = GameObject.FindWithTag("Respawn");
        }

        Die();
    }
}
```

Fig. 29 – Code for the death state initiation when an obstacle collision occurs.

A while loop is used to find a checkpoint to store. As each level has a checkpoint, this loop needs to run to find a new one when the previous level is unloaded in the LevelHandler script. It achieves this using the activeSelf property, which checks if a game object is currently enabled.

The Die() method initiates the death state, which plays the death animation and respawns the player. The following figure shows the code for this method.

```
private void Die()
{
    // Freeze player
    rb.constraints = RigidbodyConstraints2D.FreezePosition |
    RigidbodyConstraints2D.FreezeRotation;

    // Disable all animations
    an.SetBool("IsJumpingFast", false);
    an.SetBool("IsJumpingSlow", false);
    an.SetBool("IsFalling", false);
    an.SetBool("IsClimbing", false);
    StartCoroutine(DisableMovement(1f));

    // Fire Death animation
    an.SetTrigger("Death");
}
```

Fig. 30 – Code for the `Die()` method.

The method first freezes a player's position modifying the RigidBody2D's constraints. It then cancels all other animations and triggers the Death animation. This is required as it would otherwise cause the player to get stuck in a loop of animations, and the death state would never occur. Admittedly this isn't the most efficient way of doing this, but it serves its purpose.

What might be noticeable is that none of the code in the `Die()` method places the player at the checkpoint location. An issue I found in implementing a death animation was that the player would respawn before the animation finished. To solve this, I looked into Unity's documentation and found a solution using animation events. The documentation shows that these can be used to call functions at specific points in an animation's timeline (Unity Technologies, 2023h).

With this information, a `Respawn()` method was created in the Player class to set the player's position to the checkpoint position using their `transform` properties. The method's code is shown in the figure below.

```
private void Respawn()
{
    // Reset player
    rb.constraints = RigidbodyConstraints2D.FreezeRotation;
    sr.enabled = true;

    // Move player position to respawn point
    transform.position = cp.transform.position;
}
```

Fig. 31 – Code for the `Respawn()` method.

This method was then assigned to a point in the death animation, which will be called when it reaches that point. The following figure shows a screenshot of the animation event.

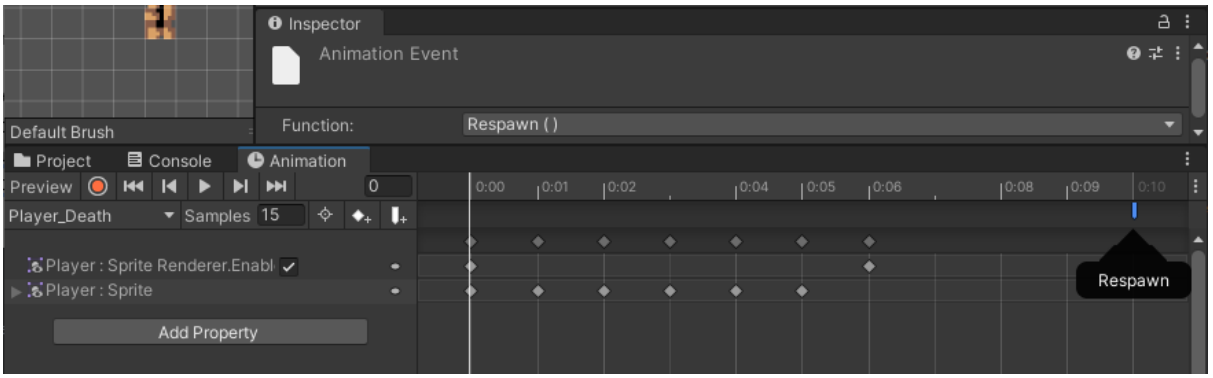


Fig. 32 – Screenshot of the animation event.

5.6 Collisions

In previous sections, it is understood that different colliders were used to create a collision detection system between the Player, the Level, and the Obstacles. This section will outline how this was achieved, using Unity's documentation to convey how it works.

Collisions in Unity are calculated either discretely or continuously. Discrete collision detection happens at discrete time steps without considering the continuous movement of objects during the time step. Continuous collision detection subdivides the time step and uses the Time of Impact (TOI) algorithm with swept colliders to detect collisions accurately

(Unity Technologies, 2023i). CCD is most useful in situations with fast-moving objects that require accurate collision detection, such as the Player object. This prevents issues in which objects colliding at high speeds can cause one object to become lodged in another. CCD was used in the Player object's RigidBody2D component to prevent this issue.

In the Player script, an OverlapCircle was used to solve an issue with the jump mechanic. The problem was that the player could infinitely jump because there was nothing to check whether the player had landed after their jump. Because OverlapCircles were used to detect walls in implementing the wall jump, it felt appropriate to extend the `CheckCollisions()` method to include ground collision.

```
private void CheckCollisions()
{
    var position = transform.position;
    isGrounded = Physics2D.OverlapCircle((Vector2)position + bottomOffset,
    collisionRadius, groundLayer);
}
```

Fig. 33 – Extending the `CheckCollisions()` method to include ground.

Figure 33 shows how this was implemented similarly to wall detection. `isGrounded` refers to a boolean that returns true if a collision enters the radius of the OverlapCircle. This is checked before a jump occurs, as shown in **Figure 34** below.

```
if (isGrounded && Input.GetButtonDown("Jump"))
{
    Jump(Vector2.up);
}
```

Fig. 34 – Code to check if the player is grounded in the `Update()` method.

Like the wall jump implementation, it also required declaring a LayerMask type variable called `groundLayer` to select the corresponding layer in the editor. This is shown in **Figure 35**.

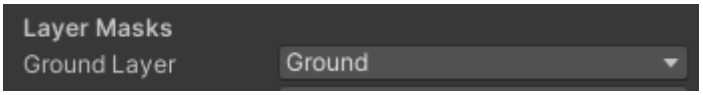


Fig. 35 – Setting up the layer mask in the inspector view.

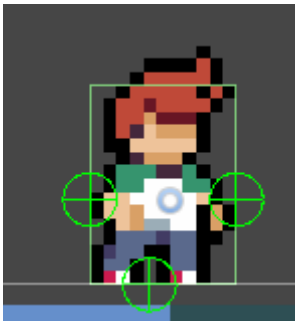


Fig. 36 – The OverlapCircle colliders shown on the Player object.

Unity employs various shapes to define its colliders. When developing the artefact, it was important to use appropriate shapes to set an object's boundaries. An example of this is in the Player object using a BoxCollider2D component (see **Figure 36** for reference). As shown in the **Section 4.2 (pp. 20)**, the Player's class diagram specifies that the box collider uses a Vector2 to get its width and height. It then computes the shape's area using the equivalent mathematical formulae ($Area = width * length$).

In other instances, PolygonCollider2D components, for example, behave differently. As they don't have a set number of points, an array of points is used, and the distance between each point is calculated to determine its area.

5.7 Interface/Main Menu

The interface for the artefact consisted of a main menu when the application is started. It was created to make the game more presentable. Unity provides a Canvas game object for designing UI elements, which was used accordingly.

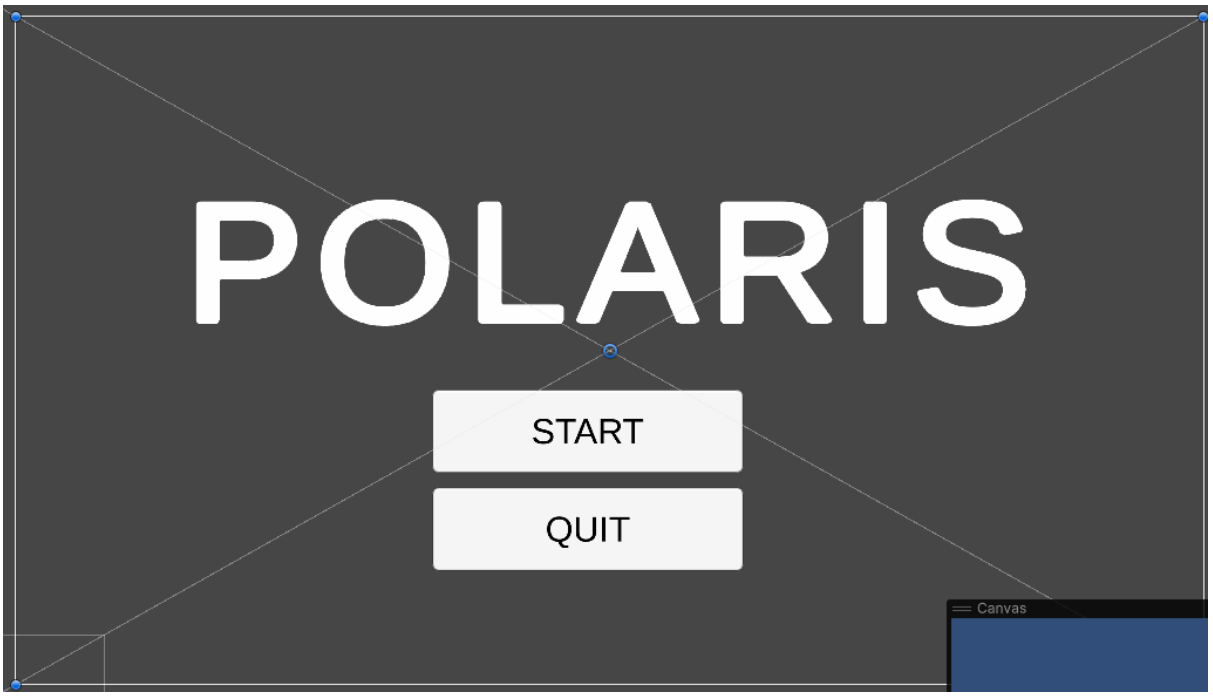


Fig. 37 – Basic implementation of UI elements as children of the Canvas object.

Figure 37 shows the basic implementation of a text object and two buttons for 'Start' and 'Quit'. The Canvas object also contains several components that control how the UI elements are viewed. For example, the Canvas Scaler component was adjusted to scale with the screen size, so the UI would always look clear and readable regardless of screen resolution.

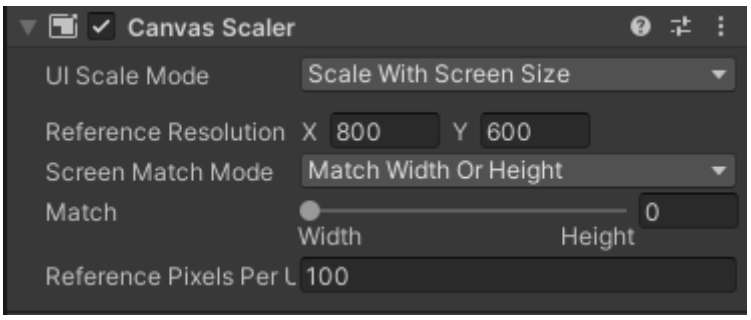


Fig. 38 – Screenshot of the configured Canvas Scaler component.

For creating functional buttons, a script was used to handle the `OnClick()` events that occur when a button is pressed. The following **Figure 38** shows the code behind this.

```
public class MainMenuHandler : MonoBehaviour
{
    public void PlayGame()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex +
        1);
    }

    public void QuitGame()
    {
        Application.Quit();
    }
}
```

Fig. 39 – Code for the `MainMenuHandler` script.

While the `QuitGame()` method is self-explanatory, it's worth mentioning how the `PlayGame()` method works. The SceneManager class provides scene management at runtime, allowing new scenes/levels to be loaded. The `LoadScene()` method takes either the name or index of the scene to load.

In the code, however, it uses the current scene's (`GetActiveScene()`) build index (meaning the order it falls in the build of the game) and adds 1 to that index, which will always load the following scene in the build index. Evidence is shown in **Figure 40** below.

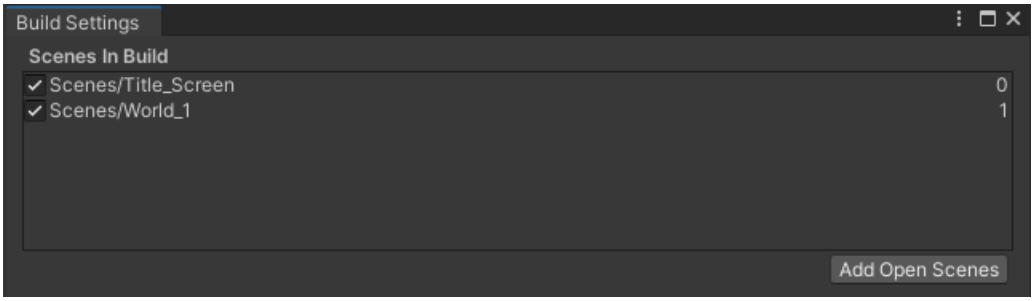


Fig. 40 – Screenshot of the Build Settings menu.

The relevant methods are called in the button's `OnClick()` events, using the Main Menu Handler script as a reference for these methods.

Fig. 41 – Screenshot of the implemented `OnClick()` event.

Further interface elements were developed in a similar way, such as a pause menu when the game was in progress. The following figure depicts a screenshot of the game's pause menu.

Fig. 42 – Screenshot of the pause interface.

The implementation was similar to the main menu, using a Canvas with Button object, and matching the `OnClick()` functionality to defined methods in a script. However, as this was a menu active during gameplay, it required pausing, so the player didn't move while the menu was active.

```
private void Update()
{
    if (Input.GetButtonDown("Cancel"))
    {
        if (pause)
        {
            Resume();
        }
        else
```

```
        {
            Pause();
        }
    }
}
```

Fig. 43 – Code for the pausing toggle, using the Cancel button (Escape Key).

To achieve this, a `pause` boolean to check if the game is paused can be declared in the PauseMenu script. Then, to create a toggle for the pause, a simple conditional statement can check if the pause is true, pause the game, and otherwise unpause it. The code for this is displayed in **Figure 43**.

The individual methods `Resume()` and `Pause()` create the toggle by setting the menu object active/inactive using `SetActive()`. This is a similar approach to how the LevelHandler script toggled level objects in **Section 5.4 (pp. 30)**. The timescale is also set to 0 whilst paused, in order to freeze the game. The implementation is shown in **Figure 44** below.

```
public void Resume()
{
    pm.SetActive(false);
    Time.timeScale = 1f;
    pause = false;
}

public void Pause()
{
    pm.SetActive(true);
    Time.timeScale = 0f;
    pause = true;
}

public void MainMenu()
{
    pause = false;
    Time.timeScale = 1f;
    SceneManager.LoadScene("Title_Screen");
}

public void QuitGame()
{
    Application.Quit();
}
```

Fig. 44 – Rest of the code for the PauseMenu script.

While an interface wasn't required for a functional solution, it did improve the game's presentation. Aspects of the interface, such as the pause menu, also gave the player more control over the game's state.

5.8 Graphics

The artefact's graphics were one of the last features produced. However, it was important to show the role of asset production in the development process.

They were created using Aseprite, as the **Section 3.2 (pp. 14)** mentioned. Sprites were produced for the Player object, first with static sprites acting as placeholders and then fully animated for each of the player's actions.

Fig. 45 – Collection of sprites showing the Player's run cycle.

Figure 45 shows these results in the player's 12-frame run cycle. In practice, the animation looks smooth and was another proud point in development.

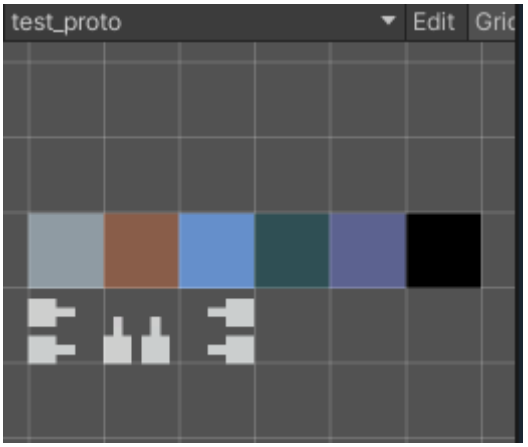
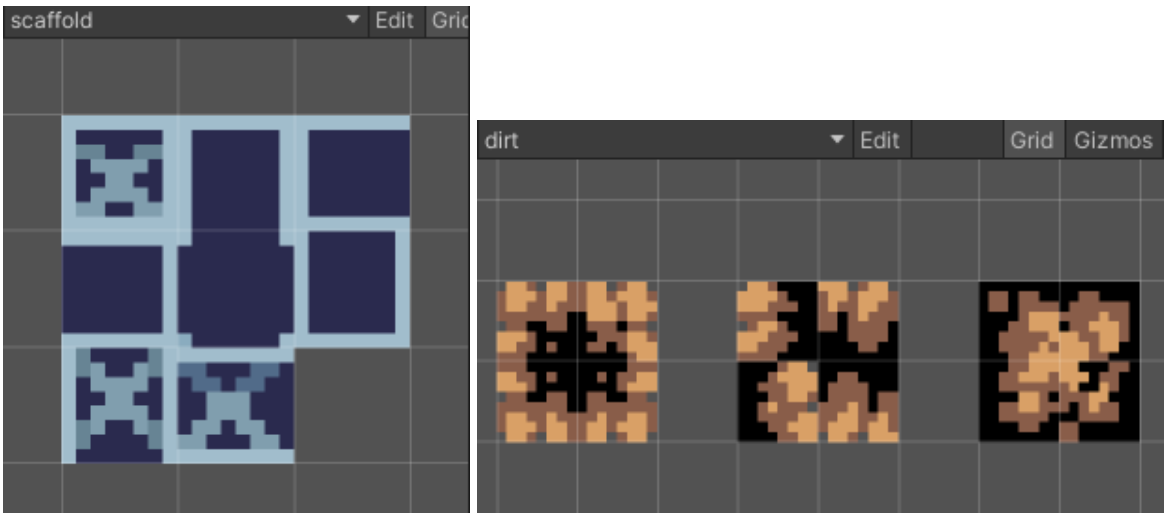


Fig. 46 – Examples of the early development tilesets.

Beyond sprites, tilesets were also produced for the game's levels. In the initial stages, coloured blocks were used to differentiate between terrain. As development progressed, more detail was added. Ultimately, they were left somewhat unfinished by the end of development. However, they were only intended to provide a base level upon which to improve. Examples of the tiles are shown in **Figures 47 & 48** below.



Figs. 47 & 48 – Examples of the finished tilesets.

As is evident from the figures above, the tile graphics here are much more detailed than the initial assets. New tiles were incorporated to handle corners and different types of terrain (such as the scaffolding tiles). Overall, I'm proud of how both the tilesets and player sprites turned out, especially with the limited space I had to work with.

5.9 Animation

Animations were created using Unity's Animator tools. After importing the sprites and creating the relevant animation files ("`Player_Running`", "`Player_Jumping`", etc.), the sprites can be dragged into the Animator panel. **Figure 49** shows this process, highlighting the sprites in the individual keyframes of the animation. Samples have been reduced to 10 to have the animations play slower.

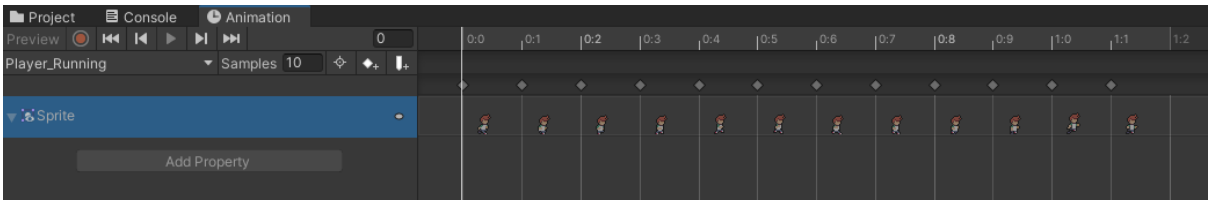


Fig. 49 – An example of the Player_Running animation in the Animator.

Once the animations were completed, an Animation Controller was required to handle transitions between the animations. This is comparable to a state machine which changes animations based on conditions. **Figure 50** illustrates the finished animation controller with arrowed lines denoting the transitions.

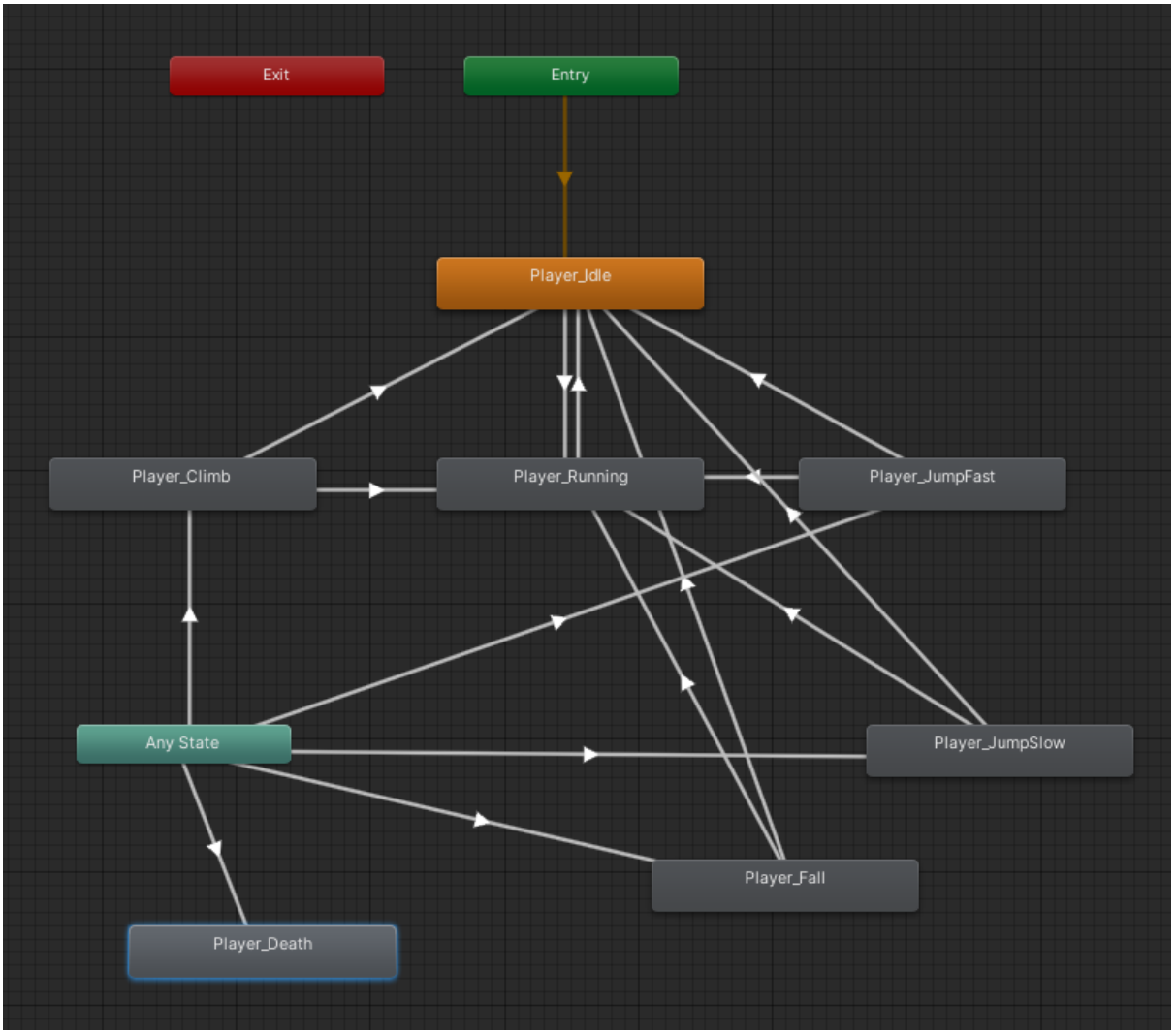
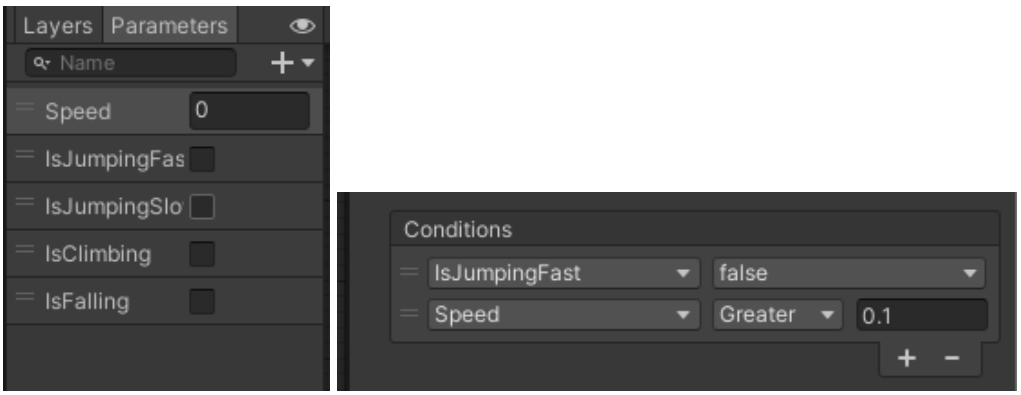


Fig. 50 – Screenshot of the finished animation controller.

The controller first enters into an idle animation node. From here, transitions between it and the running animation node can be seen. Note that animation nodes such as the climb, jump and fall all feedback into the running and idle nodes. However, these nodes don't transition into the climb, jump, and fall. This is because the Any State node is being used, which allows nodes connected to it to be transitional from any state. This solved issues where the player often gets stuck in animations that should only play once per action (such as the Jump animation).

It was mentioned earlier that transitions were achieved using conditions. These are defined in the animation controller and then handled in the Player script. **Figures 51 & 52** show examples of the variables used in the conditionals and the implementation of the conditionals in the behaviour menu of the animator.



Figs. 51 & 52 – Declaration of variables and conditionals in the animator.

For these to work, they need to be instantiated at the relevant points in the code. Below are examples of how this was implemented.

```
private void Jump(Vector2 dir)
{
    // Code omitted

    if (!Input.GetButton("Jump"))
    {
        an.SetBool("IsJumpingFast", true);
    }
    else
    {
        an.SetBool("IsJumpingSlow", true);
    }

    // Code omitted
}
```

Fig. 53 – Example of animation code in the Jump() method.

```
private void Update()
{
    // Code omitted

    an.SetFloat("Speed", Mathf.Abs(horizontalDirection));

    // Code omitted
}
```

Fig. 54 – Example of animation code for the player's running animation.

Figure 53 shows the implementation shows that when the player jumps, the relevant animation boolean is set to true to play the animation. The two different jumps are part of a revision to the code explained in **Section 6.4 (pp. 48)**. **Figure 54** shows a slightly different implementation. The horizontalDirection is set as the Speed parameter in the code, so when a player moves, the Animator checks the value of horizontalDirection and then plays the running animation.

6.0 Code Revisions

Following the initial implementation of the system, time was spent revising shortcomings and issues with the code. This section documents any refinements made to the system.

6.1 Running Improvements

One of the first issues that started to crop up in development was how static the run mechanic felt. Because of its simple implementation, the player always moved at a fixed speed. Having reviewed articles on how games achieve good run mechanics during the literature review, it felt appropriate to improve this mechanic.

The literature review findings outlined that a run can be divided into acceleration, top speed, and deceleration. The implementation involved declaring these as variables and using them to refactor the `MoveCharacter()` method.

```
private void MoveCharacter()
{
    if (Mathf.Abs(rb.velocity.x) > topSpeed)
    {
        rb.velocity = new Vector2(Mathf.Sign(rb.velocity.x) * topSpeed,
        rb.velocity.y);
    }

    rb.AddForce(new Vector2(horizontalDirection * acceleration, 0),
    ForceMode2D.Force);
}
```

Fig. 55 – Code of the revised `MoveCharacter()` method.

The figure above shows the implementation of this. The logic it provides is that while the player's velocity is greater than topSpeed, topSpeed should be applied. Otherwise, the player should build up to topSpeed by multiplying the horizontal input by the acceleration value. Note that the absolute value of a player's velocity is checked because the condition should be precise. Sign is also applied to velocity, so the value can be either -1 or 1 and doesn't calculate the assigned `Vector2` with the absolute value.

The deceleration aspect of the revision was handled in a separate method. The following figure shows that implementation.

```
private void ApplyDeceleration()
{
    if (isDashing)
    {
        rb.drag = 0f;
    }
    else
    {
        if (Mathf.Abs(horizontalDirection) < 0.4f || changingDirection)
```

```
        {
            rb.drag = deceleration;
        }
        else
        {
            rb.drag = 0f;
        }
    }
}
```

Fig. 56 – Code for the `ApplyDeceleration()` method.

When the player's input stops, the deceleration value is applied to the RidigBody2D's drag property. Drag causes an object to slow down gradually. Drag is only applied when the player has let go of the movement keys, demonstrating deceleration. There is also the absence of drag when a player is dashing. Further considerations were made for the change in direction. The corresponding variable is a boolean that has the following configuration.

```
private bool changingDirection => (rb.velocity.x > 0f &&
                                   horizontalDirection < 0f) ||
                                  (rb.velocity.x < 0f &&
                                   horizontalDirection > 0f);
```

Fig. 57 – Configuration of the `changingDirection` boolean.

This essentially checks whether the player is moving in a direction opposite to the current velocity, and the drag allows for snappier direction changes.

6.2 Input Caching

An issue mentioned during **Section 5.2 (pp. 25)** was how the implementation of the jump mechanic mixes physics calls with the `Update()` method, which Unity argues should be reserved for the `FixedUpdate()` (Unity Technologies, 2023j). A briefly suggested solution was to cache the input in the `Update()` and execute the physics call in the `FixedUpdate()`.

```
if (Input.GetButtonDown("Jump") && isGrounded)
{
    queueJump = true;
}
```

```
if (queueJump)
{
    Jump(Vector2.up);
    queueJump = false;
}
```

Figs. 58 & 59 – Refactored jump mechanic code following input caching revision.

This can be achieved through booleans and a conditional. The boolean queueJump is introduced instead of the `Jump()` method call in `Update()` and is set to true. Then, in the

`FixedUpdate()`, a conditional statement checks if queueJump is true and executes the `Jump()` method if it is.

This can be applied elsewhere in the system where physics calls were handled in the `Update()` method. For example, the figures below show a similar implementation for the dash mechanic.

```
if (Input.GetButtonDown("Dash") && !hasDashed)
{
    queueDash = true;
}
```

```
if (queueDash)
{
    Dash(horizontalMouse, verticalMouse);
    queueDash = false;
}
```

Figs. 60 & 61 – Refactored dash mechanic code following input caching revision.

6.3 Coyote Time

A feature in existing solutions like Celeste (2018) was a mechanic coined by developers as coyote time. Normally, the conditions for a jump can only occur when the player is touching the ground. However, it can feel like the player initiated the jump after leaving the platform, but the game's logic doesn't read the input, making the jump feel static and less responsive.

Coyote Time solves this idea by allowing players to jump quickly after leaving a platform. This felt like an appropriate way to improve the responsiveness of the jump.

The easiest solution is to implement a timer that counts down after the player has left a platform. Then a condition will check to see if the timer is greater than 0. If it is, the player will still be able to jump. Otherwise, they will not.

```
private void CoyoteTime()
{
    if (isGrounded)
    {
        coyoteTimeCounter = coyoteTime;
    }
    else
    {
        coyoteTimeCounter -= Time.deltaTime;
    }

    if (Input.GetButtonUp("Jump"))
    {
        coyoteTimeCounter = 0;
```

```
    }
}
```

Fig. 62 – Implementation of the coyote time mechanic.

This figure shows the implementation of a `CoyoteTime()` method. If the player touches the ground, the timer value is set. Otherwise, the timer decreases using the interval in seconds from the last frame to the current one (`Time.deltaTime`). Finally, to ensure the player cannot repeatedly jump, the timer is set to zero when the jump key is released (`Input.GetButtonUp`).

Further revisions needed to be made to how the game checks for jumping. Before, the condition checked if the player was touching the ground, and the jump key was pressed. Now, the condition should check if the timer is greater than zero *and* the jump key is being pressed.

```
private void Update()
{
    if (coyoteTimeCounter > 0 && Input.GetButtonDown("Jump") && !hasDashed
    )
    {
        queueJump = true;
    }
}
```

Fig. 63 – Refactored conditional statement for the jump mechanic.

Note that this implementation doesn't check if the player is grounded anymore. This is because negating multiple jumps occur in the `CoyoteTime()` method by setting the timer to zero when the jump key is released.

6.4 Variable Jump Time

In testing the solution against existing solutions, I found that the jump mechanics in games like Celeste (2018) and Super Mario Bros. (1985) allowed players to vary their jump height based on how long the jump key is pressed. Tapping the key resulted in shorter, snappier jumps, whereas holding it retained upwards momentum for longer. This seemed like an improvement over the solution's static jump, giving players more control over their jumps.

In implementing this, a timer was employed similarly to the Coyote Time revision. While the timer counts down, the refactored `FallMultiplier()` method applies the appropriate gravity depending on if the jump is short or long. This is illustrated in the following figure and explained in the following paragraph.

```
private void FallMultiplier()
{
```

```
    if (jumpTimer < 0.01f)
    {
        rb.gravityScale = fallForce;
    }
    else if (rb.velocity.y > 0 && !Input.GetButton("Jump"))
    {
        rb.gravityScale = lowJumpFallForce;
    }
    else
    {
        rb.gravityScale = 1f;
    }
}
```

Fig. 64 – Code revisions made in the `FallMultipler()` method.

While the jump key is held, the player will continue to jump until the jump timer hits 0.01, in which the first condition applies gravity to pull the player back down.If the player is jumping but lets go of the button, gravity is applied at a smaller degree than in the previous case. Otherwise, the gravity scale will be reset.

6.5 Mouse Dash Implementation

A problem that needed addressing was the control scheme. The dash originally took whatever movement keys were being pressed as its direction parameters. This led to issues where the control scheme felt archaic, with the majority of control on the left side of the keyboard.

It felt appropriate to improve this and incorporate the rest of the computer's peripherals, chiefly the mouse - further embracing the PC distribution with PC-specific controls. It also showed that precise movement on a controller typically uses analog sticks, and the mouse seemed like the closest approximation for a PC platform.

Improving the dash controls involved setting up a way to include mouse input. It also had to use the player as an anchor point because the dash originates from the player. As the player's position is relative to the world space, and the mouse's position is relative to the screen space, I had to find a way of translating the two positions.

In researching how to acquire an object's position, I was met with the `WorldScreenToPoint()` method. This method transforms the `position` property of an object from world space into screen space (Unity Technologies, 2023k).

```
private Vector2 GetMouseInput()
{
```

```
    // Find mouse position relative to player object
    playerPos = cam.WorldToScreenPoint(transform.position);
    mousePos = Input.mousePosition;
    playerToMouse = (mousePos - playerPos);

    return new Vector2(playerToMouse.x, playerToMouse.y);
}
```

Fig. 65 – Code for getting the mouse input.

A method similar to `GetInput()`'s implementation was used to return a Vector2 with the values needed. Within the method, the player's position is calculated by invoking the `WorldToScreenPoint()` method on the camera reference and passing the object's position through it. Getting the mouse position was simpler, as it already used screen space. Finally, to anchor the position to the player, the player position is subtracted from the mouse position to find the distance between them.

To verify this works, the line `Debug.Log(dashDir.normalized)` can be added in the `Dash()` method to list the mouse direction when dashing. The following figure shows a sample of the results.

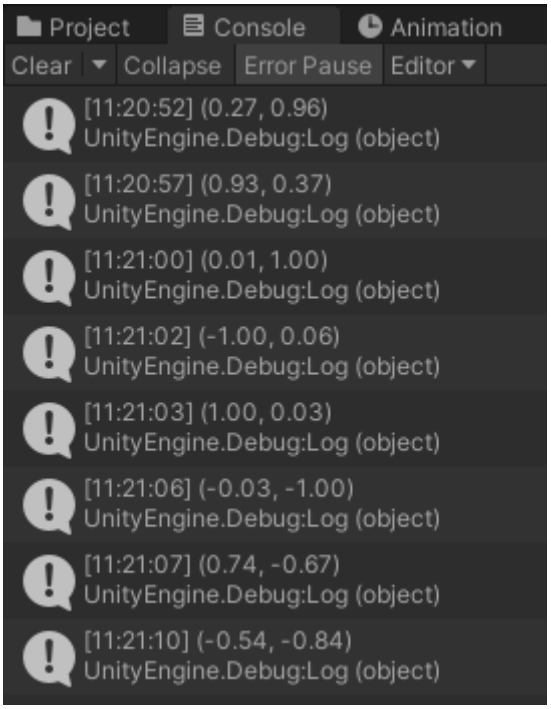


Fig. 66 – Debug Log evidence for mouse dash revision.

6.6 Wall Climbing

During development, it was decided to include a wall climbing feature to complement the other mechanics. Inspiration for the feature came from Celeste (2018), which used a similar mechanic for more methodical level traversal.

The implementation remained ultimately unfinished, but significant progress was made on the mechanic, however, did not include all of the intended features. The following figure shows the functional parts of the code. The full code is viewable in **Appendix F (pp. 68)**.

```
private void WallGrab()
{
    if (!canMove)
    {
        return;
    }

    rb.gravityScale = 0f;

    rb.velocity = new Vector2(rb.velocity.x, verticalDirection *
    climbSpeed);

    // Decrease player's stamina during climbing
    if (rb.velocity.y == 0)
    {
        stamina -= staminaCost * Time.deltaTime;
    }
    else
    {
        stamina -= staminaCostClimbing * Time.deltaTime;
    }

    if (stamina <= 0)
    {
        wallGrab = false;
    }
}
```

Fig. 67 – Code for incomplete Wall Climb mechanic.

The method's logic is to lock the player's horizontal direction while allowing them to move vertically up/down a wall. The gravityScale property is set to 0 to stop affecting the player with gravity.

The mechanic also features a stamina resource, which depreciates over time at varying speeds depending on if the player is climbing or stationary. Checks have also been included to ensure the player doesn't climb if they cannot move.

Although left unfinished, it was still an interesting look at the ideas that appear naturally as a part of the development process.

7.0 Testing

This section concerns testing various parts of the solution to identify mistakes, errors, and bugs. Several methods have been used, including white-box static analysis and black-box unit testing.

7.1 Static Code Analysis

As part of the testing procedure, it was necessary to use static analysis tools to identify problems in the code before runtime. The built-in analyser provided by JetBrains Rider IDE was used for this testing.

The report revealed several issues of varying degrees of severity. A few issues regarding inefficient coding practices were highlighted in the Player class.

One of the more persistent issues was a repeated property access error. These were used throughout the code, typically in conditional statements and when carrying out operations and declarations of built-in variables. The static analysis counted 3 occurrences of this.

Fig. 68 – An example of an inefficient property access issue.

Figure x shows the repeated use of `rb.velocity` and how the analyser highlights the inefficiency. The solution given by the static analyser was to introduce a `velocity` variable that stores `rb.velocity` so that calls would be made to the `velocity` variable and not `rb.velocity` directly.

Another issue highlighted was inefficient property lookup when modifying animation states. This was accounted for 18 times in the Player class. The following figure shows the static analysis report for this issue.

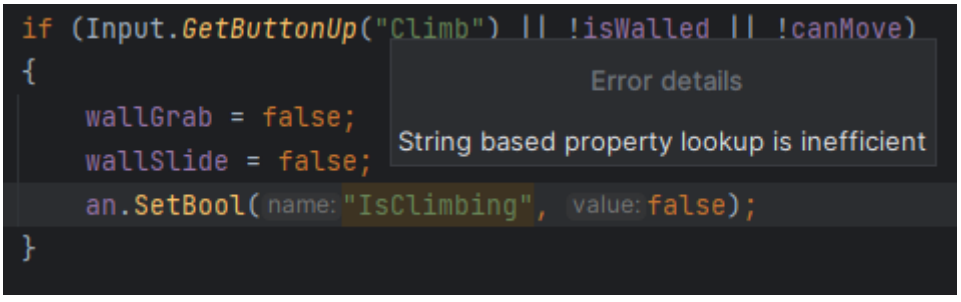


Fig. 69 – An example of a string-based property lookup issue.

However, the solution was simple to fix by removing the string quotations and fetching the ID of the animation parameter instead. This used what the static analyser described as a ‘cached property index’, which would explain why it was an efficient solution than string-based property lookup.

7.2 Performance Profiling

For this part of the testing, Unity's Profiler tool will generate performance information about the solution. **Table 3** shows the hardware specifications that will be used to carry out the testing.

CPU	Intel(R) Core™ i7-10700K CPUI @ 3.80GHz
GPU	GIGABYTE AORUS Master RTX 3080
RAM	32.00GB
Operating System	Windows 10 Home
NVME Storage	2TB

Table 3 – System specifications to test the game with.

Component	Usage Time
Rendering	0.11ms
Scripts	0.24ms
Physics	0.08ms
Animation	0.16ms

Table 4 – Profiler's CPU Usage results after playtesting (averaged).

The results represent the time a given component uses on the CPU thread throughout a play session. These figures were then averaged to give the results seen in **Table 4**.

The results are as expected, with scripts taking up most CPU usage times. However, the results are still relatively low, which is understandable when considering the hardware specifications used to test the solution.

8.0 Evaluation

To evaluate the project, the hypothesis must be assessed to determine if the project was successful. In building a game for a PC distribution, I felt that the project succeeded in producing an adequate solution to this problem. Within the timeframe given, all requirements outlined in the Project Plan were implemented, including new features such as the wall climb adapted from the initial mechanics.

In assessing the suitability of a PC platform for this solution, the results were concise, albeit limited by the scope of the project's testing. The modularity of PC hardware and the lack of ethics approval for this project made it hard to source results beyond my computer's specifications and ways I could manipulate that (i.e., by using the onboard graphics over a graphics card). I also faced difficulties finding cases to unit test that didn't amount to more than what could be achieved by logging debug messages in the console.

Despite this, there were ways in which this was achieved through the conscious design decisions made during development. For example, performance decisions about the screen's resolution, the choice of 2D over 3D, and having the levels load/unload through a level manager script. Also, in embracing PC-specific controls to implement dash mechanics that took a mouse's pinpoint accuracy to control the dash's direction.

The graphical aspects of the project were mostly successful too. I felt specifically the player sprites and animations turned out best. The level detailing was partially finished, only missing a few tile graphics here and there. It's fair to say I struggled with asset production due to my artistic limitations. However, the decision on the resolution meant that it was ultimately less work which was useful in quickly producing assets.

As for the coding, implementing this part of the project took longer than expected. It meant that the code ended up more robust and flexible, especially in making methods designed to be reusable. However, in other areas, such as the Update() and FixedUpdate() methods, the code is more archaic as there is little organisation within the methods. A positive change introduced when writing the code was using regions to organise the code into sections (Setup, Physics, Animation etc.). The results of the static analysis testing revealed a wealth of inefficient coding practices, but following the advice of these tools, improvements were made. Overall, the code had to be simple and iterative in its revisions, which I felt I achieved in that respect.

In implementing mechanics, while I managed to cover everything I set out to do in the project plan, some features presented issues and incompleteness. The wall climb, for example, was one such feature that wasn't completely implemented in the timeframe given. This was largely due to it being a decision made during development. With concerns that the mechanic would lead to feature creep (a risk outlined in the risk assessment), I felt it necessary to keep what was finished and abandon subsequent functionality not yet

implemented. Even though it remains unfinished, it was interesting to understand how ideas develop organically throughout development.

There were also issues I was unable to resolve in the implementation. For example, the dash would often have inconsistencies in the distance it would move the player. Essentially, dashing upwards would send a player further than dashing horizontally or diagonally. While I couldn't figure out the reason for this problem, I suspect it was a forces issue (drag, gravity, etc.).

Regarding project management, I felt the planning was adequate for the tasks I set out to complete. I delegated the tasks with appropriate time frames to complete them within and even accounted for extra time should I have needed it. Certain aspects of the project, such as the code implementation, took longer than expected. This delayed other aspects, such as testing, which couldn't be done until the code was implemented. Tasks like asset production, which I originally planned to carry out alongside the code implementation, ended up being about a week behind schedule, but this was again due to underestimating the time needed for coding. However, the code was written to be reusable in many cases. One such example was reusing the Jump() method within the WallJump() method. This reduced the time spent coding and reinforces an earlier point made on the conscious design decisions, in this case, to meet deadlines.

The Gantt Chart was also useful in guiding my project's progress. However, I feel this style of software development would've benefited greatly from an agile plan rather than the waterfall plan. This is because game development is largely iterative, which agile is effectively built for. As stated previously, the artefact's development was largely iterative through implementation and code revisions, further supporting the use of agile in hindsight.

9.0 Conclusions

The main aim of this project was to assess the suitability of a video game for PC distribution. Undertaking this project proved this hypothesis, albeit in a way that was hindered by a lack of options in testing the use case of a PC platform. Despite this, the conscious design decisions made to reduce potential performance issues later on and considerations made for the type of game (2D vs 3D) ensured that the project was somewhat successful. Decisions made towards using PC-specific controls, such as the mouse-controlled dashing, also helped to achieve a successful outcome.

The project benefitted from reusable code, which reduced time spent on coding and showed good coding practices in an object-oriented environment. While some features were implemented poorly, they served the purpose they were created for, and further changes will be discussed in the following sub-section.

Ultimately, this project was an exploration of the many facets of game development from the perspective of a computer science student. Considering that, I feel the product touches on these significant areas (gameplay, level design, graphics, and user interface) well. It also involved software engineering topics such as a waterfall methodology in the project plan, object-oriented design using C# to code, and an adequate testing approach in static analysis and performance profiling.

From this, it's accurate to say that the project went well. In so much as developing an application that fulfils the requirements. Some improvements will be needed in the code's unfinished areas, but overall the code is robust and stable, posing only a few noticeable edge cases. In proving the hypothesis, I feel it achieves this through performance, design, and controls, although further changes could be made to satisfy the hypothesis further.

9.1 Future Considerations

The future considerations of my artefact are aspects, details, and features necessary in future work to develop an effective software application.

Firstly, I recognise the importance of addressing incomplete or inconsistent features, especially those outlined in the artefact’s design. The wall climbing, for one, is still lacking in player animations and an incomplete climbing jump. Although this wasn’t a design-specific feature, I would still treat it with the same focus and effort to implement the remaining implementation. Beyond this, it would be important to understand the inconsistencies plaguing the dash functionality. As mentioned previously, I believe the issue lies in the forces being applied during and after a dash, which would be the starting point in fixing the issues.

Another consideration I’d make is refactoring some of the weakest parts of the code. For example, the aforementioned Die() method is inefficient in cancelling other animations

before starting the death animation. I would likely make use of the Animation.Stop() function to individually stop the animations over modifying the conditions that affect them playing. This may not solve the edge cases where the player is stuck looping between the death and the previous animation. Still, it would at least create uniformity in using appropriate methods to stop animations.

A penultimate consideration would be finding a better way to define collisions for the obstacles. An Obstacle game object currently holds numerous BoxCollider2D components that make up the obstacle collisions in a given level. The problem is in its extensibility, as each collision consists of a new BoxCollider2D component, making the current system unmanageable with large numbers of collision components. One potential solution could be to use polygon colliders to create a singular freeform shape instead of multiple BoxCollider2D components.

A final consideration would be to develop more levels and improve the assets for the levels. By the end of development, the graphics were some of the final assets produced for the game; therefore, only the necessary amount of time was spent working on them. Because of this, the solution featured a mix of detailed tilesets and the more primitive ones featured in **Figure 46 (pp. 41)**. In terms of the level design, while I'm happy with how the levels turned out, there are persistent issues with collisions, as mentioned previously. As well as this, managing multiple levels inside of a single scene proved unwieldy at times, despite my initial assessment that the scope of the game would be small enough to accommodate all the levels in such a way. In future, it would be appropriate to rethink this design approach or at least improve the current implementation.

10.0 References

1. Aleem, S., Capretz, L. F., and Ahmed, F.. (2016) 'Critical success factors to improve the game development process from a developer's perspective', *Journal of Computer Science and Technology*, 31(5), p. 1. doi: 10.007/s11390-016-1673-z

2. Boller, S. (2013) 'Learning Game Design: Game Mechanics', *The Knowledge Guru,* 17th July. Available at: http://www.theknowledgeguru.com/learning-game-design-mechanics/ (Accessed: 15th November 2022).

3. Brodkin, J. (2013) 'How Unity3D Became a Game-Development Beast', *Dice,* 3rd June. Available at: https://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/ (Accessed: 15th November 2022).

4. Brown, M., Game Maker's Tool Kit (2019) *Why Does Celeste Feel So Good to Play?* 31st July. Available at: https://youtu.be/yorTG9at90g

5. Burke, Q., and Kafai, Y. (2014) 'Decade of Game Making for Learning: From Tools to Communities'. Handbook of Digital Games. 689-709. doi: 10.1002/9781118796443

6. Bycer, J. (2019) *Game Design Deep Dive: Platformers.* Place of publication: CRC Press.

7. Chen, H., 2010. Comparative Study of C, C++, C# and Java Programming Languages. p. 44 Available at: https://www.theseus.fi/bitstream/handle/10024/16995/Chen_Hao.pdf

8. Lim, T; Louchart, S; Suttie, N; Ritchie, J.M.; Aylett, R.S.; Stanescu, I.A.; et al. (2013). "Strategies for Effective Digital Games Development and Implementation". *Cases on Digital Game-Based Learning*. IGI Global. 12: 168–198. doi: 10.4018/978-1-4666-2848-9.ch010

9. Miyamoto, S., Tezuka, T. Nintendo of America (2015) *Super Mario Bros. 30th Anniversary Special Interview ft. Shigeru Miyamoto & Takashi Tezuka*, 13th September. Available at: https://youtu.be/DLoRd6_a1CI

10. Sanders, A. (2017) *An Introduction to Unreal Engine 4*. Available at: https://doi.org/10.1201/9781315382555 (Downloaded: 17th November 2022).

11. Shouldice, A., Uploaded by Gabriel Williams (2016) *Secret Legend - ProBuilder Developer Showcase*. 18th October. Available at: https://www.youtube.com/watch?v=i2IT4VBqfGI

12. Spolsky, J. (2002) 'The Law of Leaky Abstractions', *Joel on Software*, 11th November. Available at: https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/ (Accessed: 16th November 2022).

13. Thorson, M., Game Developer's Conference (GDC), (2017) *Level Design Workshop: Designing Celeste*. 27 February. Available at: https://youtu.be/4RlpMhBKNr0

14. Tristem, B. (2022) 'Unity vs. Unreal: Which Game Engine is Best For You?', *udemy*. Available at: https://blog.udemy.com/unity-vs-unreal-which-game-engine-is-best-for-you/ (Accessed: 13th November 2022).

15. Unity Technologies. (2022) *Build Virtual Worlds From Scratch | Unity ProBuilder*. Available at: https://unity.com/features/probuilder (Accessed: 16th November 2022).

16. West, C. (2021) 'Benefits of Prototyping without Assets', *Medium,* 15th April. Available at: https://gamedevchris.medium.com/white-boxing-grey-boxing-what-6aa9cfa3b0e4 (Accessed: 15th November 2022).

17. Williams, W. R., Mahugh, D., and Gorzelany, J. (1982) 'Joystik Magazine', *Joystik Magazine,* 1(3), p. 54. Available at: https://archive.org/details/joystik_magazine-1982-12/page/n55/mode/2up?q=%22climbing+game%22 (Accessed: 15th November 2022)

18. Unity Technologies (2023a) *Manual: Order of Execution for event functions*, Available at: https://docs.unity3d.com/Manual/ExecutionOrder.html (Accessed 28th March 2023).

19. Unity Technologies (2023b) *Manual: Scripting*, Available at: https://docs.unity3d.com/Manual/ScriptingSection.html (Accessed 28th March 2023).

20. Unity Technologies (2023c) *Important Classes – Time*, Available at: https://docs.unity3d.com/Manual/TimeFrameManagement.html (Accessed 29th Marchl 2023).

21. Unity Technologies (2023d) *Introduction to Tilemaps*, Available at: https://learn.unity.com/tutorial/introduction-to-tilemaps# (Accessed 1st April 2023).

22. Unity Technologies (2023e) *Universal Rendering Pipeline overview*, Available at: https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@16.0/manual/index.html (Accessed 11th April 2023).

23. Unity Technologies (2023f) *Scripting API: Rigidbody.interapolation*, Available at: https://docs.unity3d.com/ScriptReference/Rigidbody-interpolation.html (Accessed 13th April 2023)

24. Unity Technologies (2023g) *Scripting API: MonoBehaviour.OnTriggerEnter2D*. Available at: https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnTriggerEnter2D.html (Accessed 14th April 2023).

25. Unity Technologies (2023h) *Manual: Using Animation Events*. Available at: https://docs.unity3d.com/Manual/script-AnimationWindowEvent.html (Accessed 14th April 2023).

26. Unity Technologies (2023i) *Manual: Continuous Collision Detection*. Available at: https://docs.unity3d.com/Manual/ContinuousCollisionDetection.html (Accessed 15th April 2023).

27. Unity Technologies (2023j) *Scripting API: MonoBehaviour.FixedUpdate*. Available at: https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html (Accessed 17th April 2023).

28. Unity Technologies (2023k) *Scripting API: Camera.WorldToScreenPoint*. Available at: https://docs.unity3d.com/ScriptReference/Camera.WorldToScreenPoint.html (Accessed 17th April 2023).

29. Thorson, M., and Berry, N. (2019) *Celeste Public Archive Repository*. GitHub. Available at: https://github.com/NoelFB/Celeste (Accessed 15th November 2022).

11.0 Bibliography

1. Aaltonen, T., 2018. 3D platformer level design. Available at: https://www.theseus.fi/bitstream/handle/10024/157130/Aaltonen_Toni.pdf?sequence=1&isAllowed=y (Accessed 10th November 2022)

2. Bennedsen, J. and Caspersen, M.E., 2006. Abstraction ability as an indicator of success in learning object-oriented programming?. *ACM Sigcse Bulletin*, *38*(2), pp.39-43. Doi: 10.1145/1138403.1138430

3. Bhosale, T., Kulkarni, S. and Patankar, S.N., 2018. 2D Platformer Game in Unity Engine. *International Research Journal of Engineering and Technology*, *5*(04), pp.3021-3024. Available at: https://www.irjet.net/archives/V5/i4/IRJET-V5I4667.pdf (Accessed: 10th November 2022)

4. Birdwell, K., 1999. The cabal: Valve's design process for creating half-life. *The game designer reader: A rules of play anthology*, pp.212-225.

5. E. F. Anderson and col., Choosing the infrastructure for entertainment and serious computer games - a whiteroom benchmark for game engine selection, 2013 5th Intl. Conf. on Games and Virtual Worlds for Serious Apps, pp. 1–8.

6. Floridi, L., and Sanders, J. W. (2004) 'Levellism and the method of abstraction', *EG Research Report*, Available at: http://www.cs.ox.ac.uk/activities/ieg/research_reports/ieg_rr221104.pdf (Accessed 17th November 2022)

7. Foxman, M., 2019. United we stand: Platforms, tools and innovation with the unity game engine. *Social Media+ Society*, *5*(4), doi: https://doi.org/10.1177/2056305119880177

8. Gajewski, S., El Mawas, N., and Heutte, J. (2022) 'A systematic literature review of game design tools', CIREL – Centre Interuniversitaire de Recherche en Éducation de Lille, F-59000 Lille, France, January 2022, p. 1-2. doi: 10.5220/0011137800003182

9. Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1993, July. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming* (pp. 406-431). Springer, Berlin, Heidelberg. doi: 10.1007/3-540-47910-4_21

10. Gustafsson, A., 2014. *An Analysis of Platform Game Design*. Available at: https://www.diva-portal.org/smash/get/diva2:728079/FULLTEXT01.pdf

11. Haas, J.K., 2014. A history of the unity game engine. *Diss. WORCESTER POLYTECHNIC INSTITUTE*, *483*, p.484. Available at: https://digital.wpi.edu/downloads/2f75r821k?locale=en

12. Hejlsberg, A., Wiltamuth, S. and Golde, P., 2003. C# language specification.

13. Josef, A., Van Lepp, A. and Carper, M., 2022. Scope and Engine Choices. *The Business of Indie Games*, pp.75-92.

14. Minkkinen, T., 2016. Basics of Platform Games.

15. Pichlmair, M. and Johansen, M., 2021. Designing Game Feel. A Survey. *IEEE Transactions on Games*.

16. Šmíd, A., 2017. Comparison of unity and unreal engine. *Czech Technical University in Prague*, pp.41-61.

17. Smith, G., Cha, M., and Whitehead, J. (2008) 'A framework for analysis of 2D platformer levels', *Sandbox '08*, pp. 75-80. doi: 10.1145/1401843.1401858

18. Stroustrup, B. (1988) "What is object-oriented programming?," in IEEE Software, 5(3), pp. 10-20, May, doi: 10.1109/52.2020

19. Wehbe, R.R., Mekler, E.D., Schaekermann, M., Lank, E. and Nacke, L.E., 2017, May. Testing incremental difficulty design in platformer games. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (pp. 5109-5113).

20. Zook, A. and Riedl, M.O., 2014. Generating and adapting game mechanics. In *Proceedings of the 2014 Foundations of Digital Games Workshop on Procedural Content Generation in Games*.

21. Cardelli, L. and Wegner, P. (1985) 'On Understanding Types, Data Abstraction, and Polymorphism', *Computing Surveys*, 17(4), pp. 471-522, Available at: http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf (Accessed: 17th November 2022)

22. Cook, R., Hill, W., and Canning, P. S. (1989), 'Inheritance is not subtyping', *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages,* pp. 125-135. doi: 10.1145/96709.96721

23. Hirshfield, S., and Ege, R. K. (1996) 'Object-Oriented Programming', *ACM Computing Surveys*, 28(1), doi: 10.1145/234313.234415

24. Kramer, J. (2007) 'Is Abstraction the Key to Computing?', *Communications of the ACM*, 50(4), pp. 37-42. Doi: 10.1145/1232743.1232745

25. Pierce, B. C. (2002), 'Types of Programming Languages', *Bulletin of Symbolic Logic*, 10(2), ch.18, doi: 10.1017/S1079898600003954

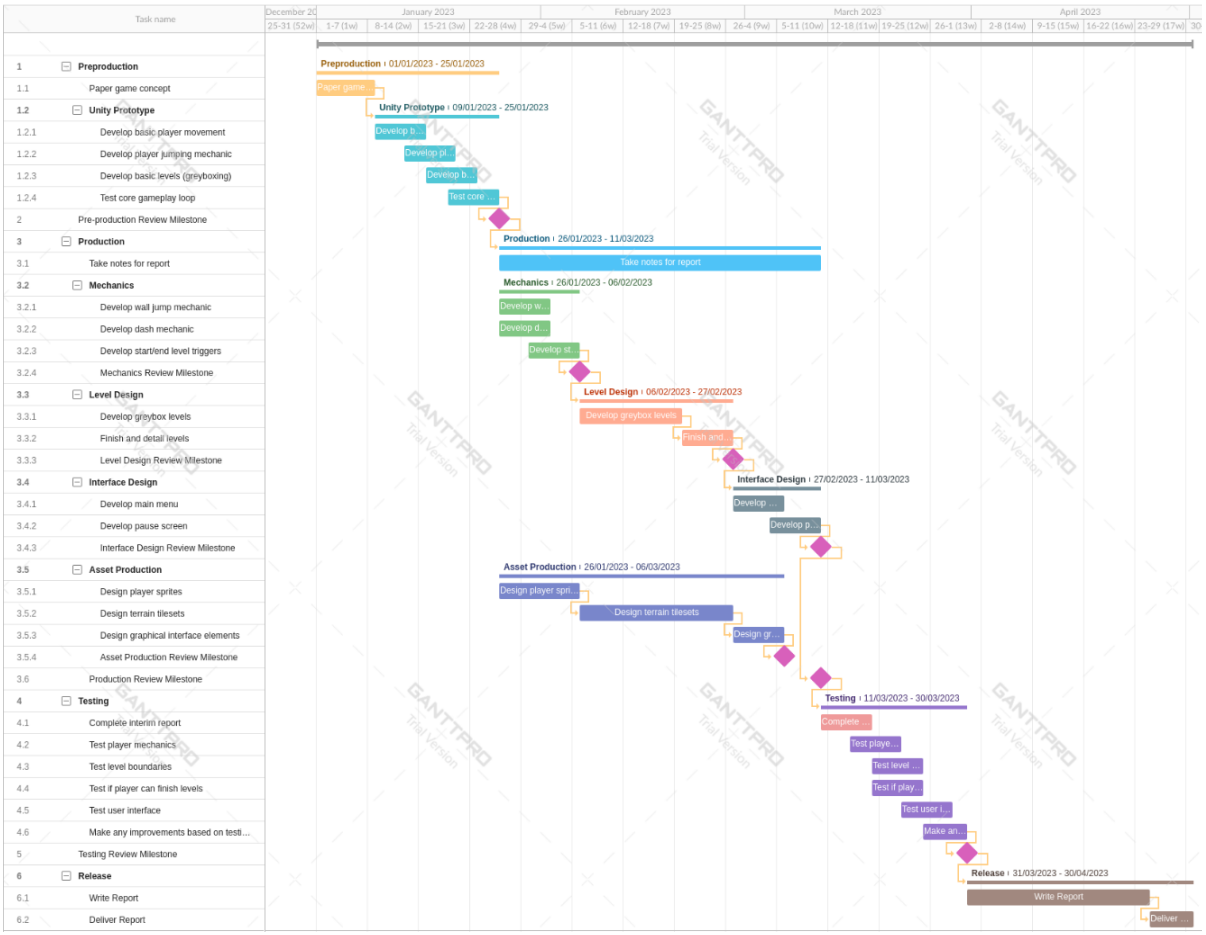
26. Rentsch, T., 1982. Object-oriented programming. *ACM Sigplan Notices*, *17*(9), pp.51-57. Doi: https://doi.org/10.1145/947955.947961

27. Rogers, P. (2001) 'Encapsulation is not information hiding', *InfoWorld*, 18th May. Available at: https://www.infoworld.com/article/2075271/encapsulation-is-not-information-hiding.html (Accessed: 16th November 2022).

28. Snethen, G. (2008) "Complex Collision Made Simple", *Game Programming Gems 7*, pp.165–178

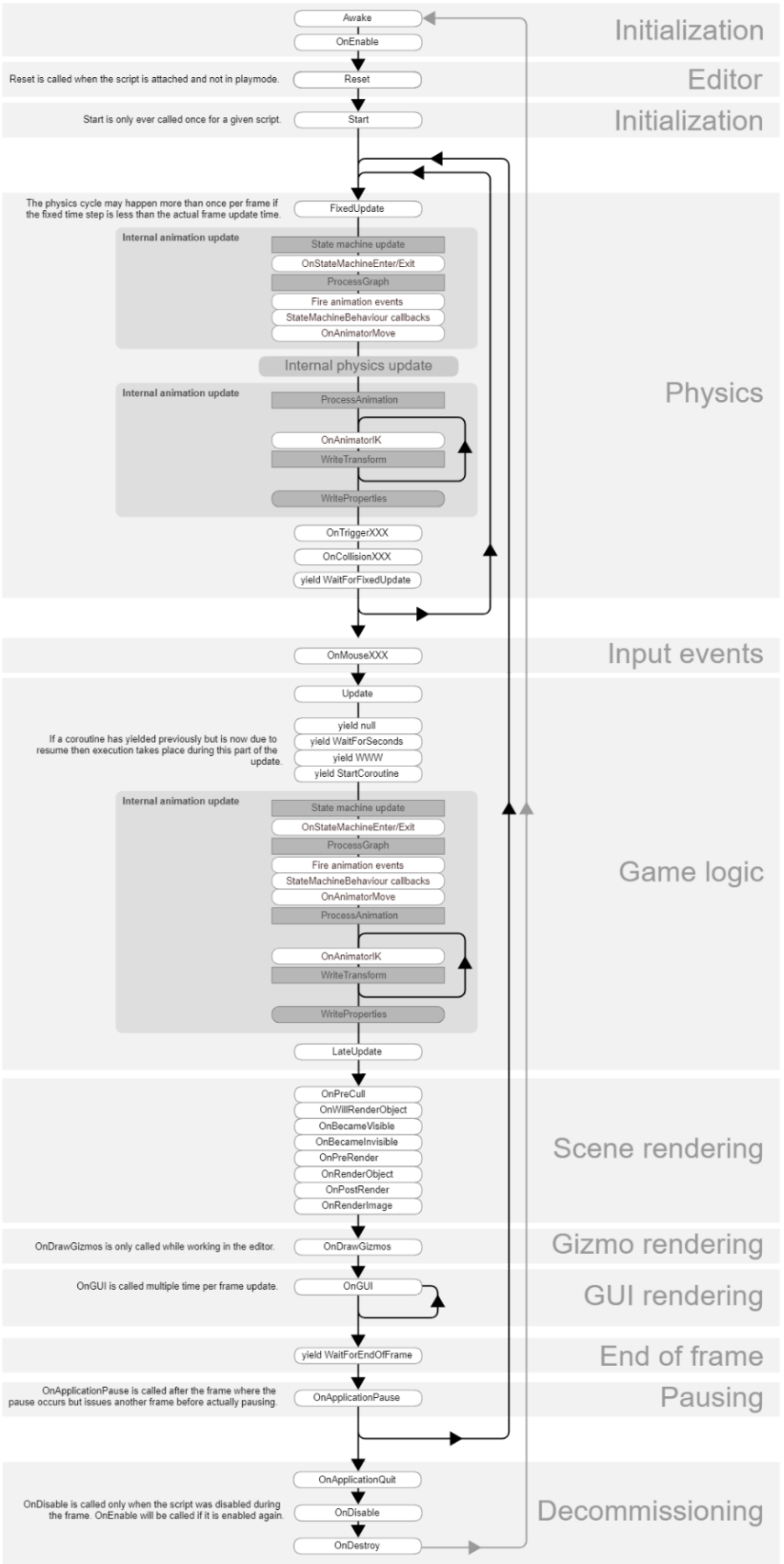
12.0 Appendices

Appendix A: Gantt Chart of Project Plan



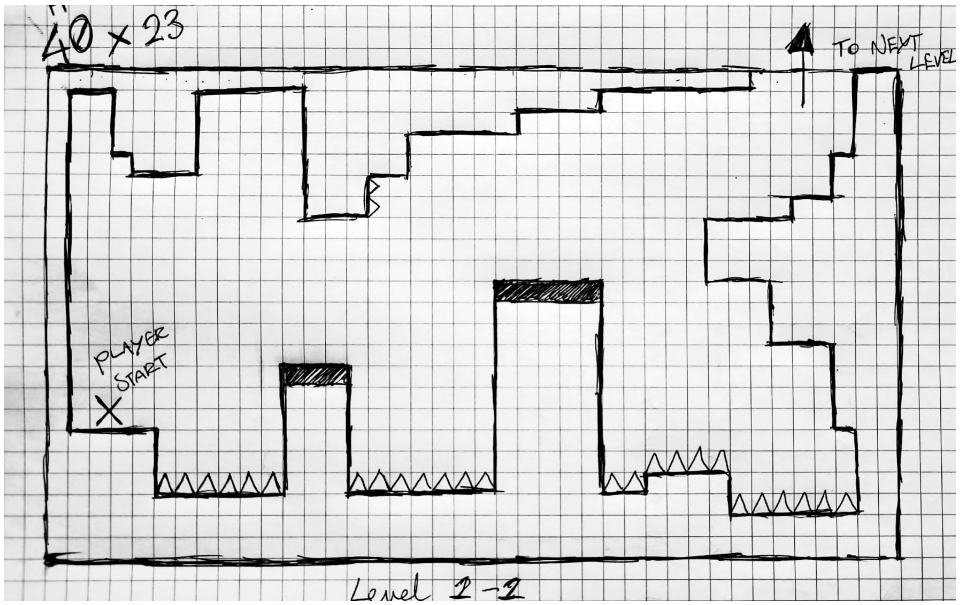
Appendix A – Gantt Chart for Project Plan. Alternate PDF link if the figure isn't clear enough: https://drive.google.com/file/d/1twAXGYcd3sEe-xvPTQBly4--13wrgDN5/view

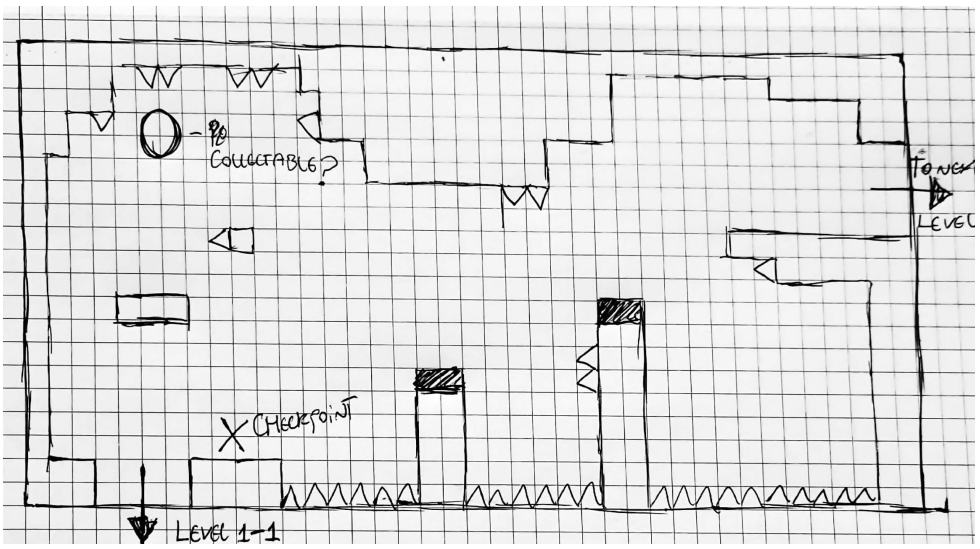
Appendix B: Order of Execution for Event Functions Diagram



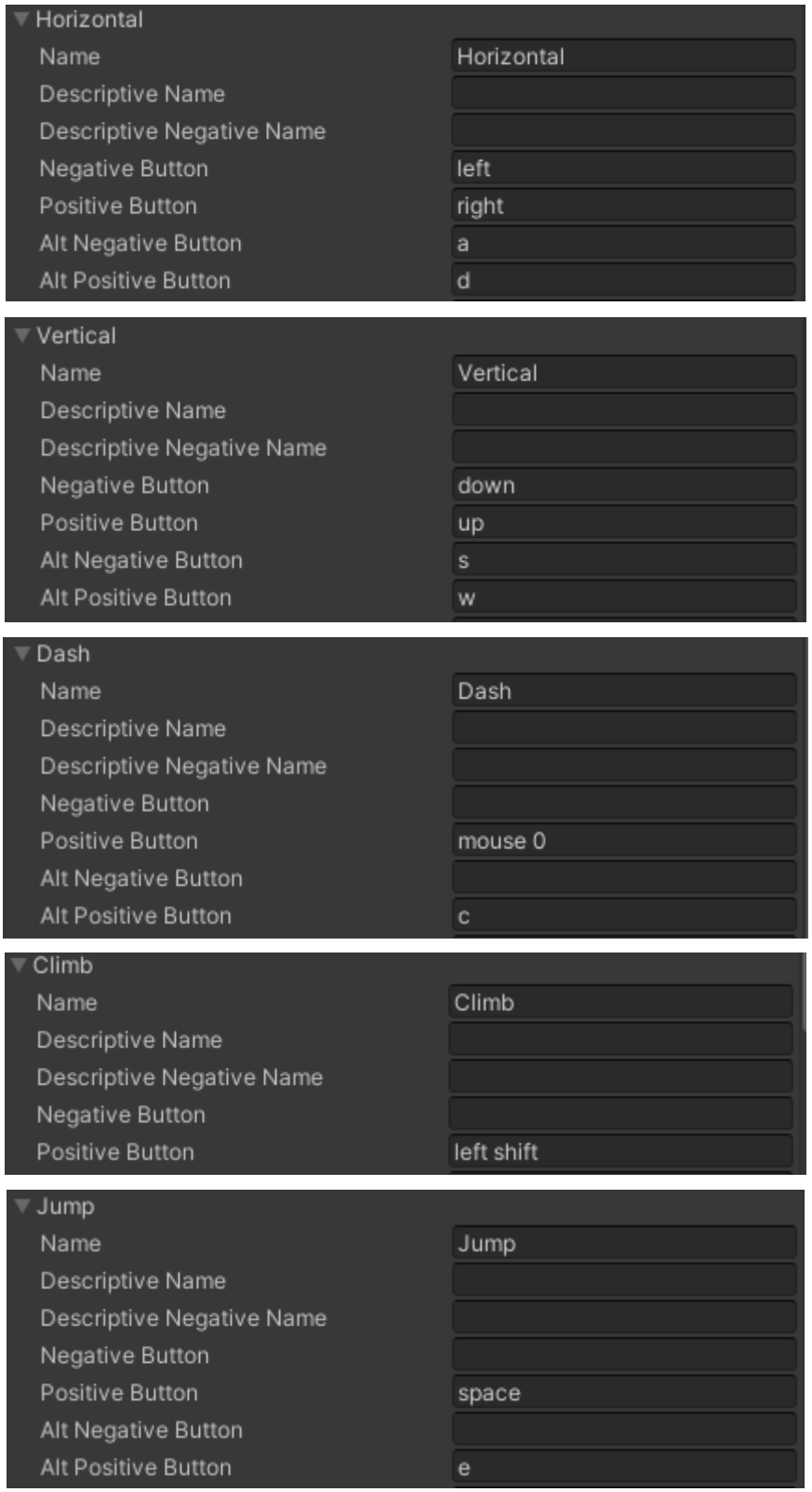
Appendix B – Order of Execution for Event Functions Diagram (Unity Technologies, 2023a)

Appendix C: Level Design Concepts

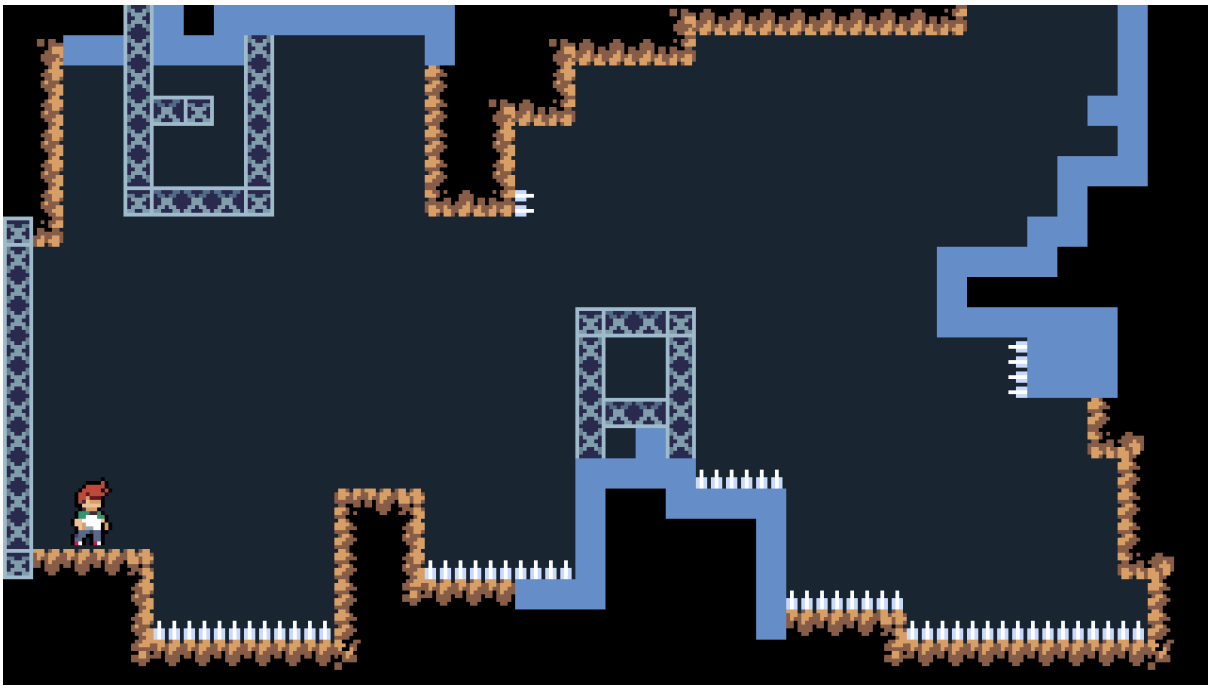


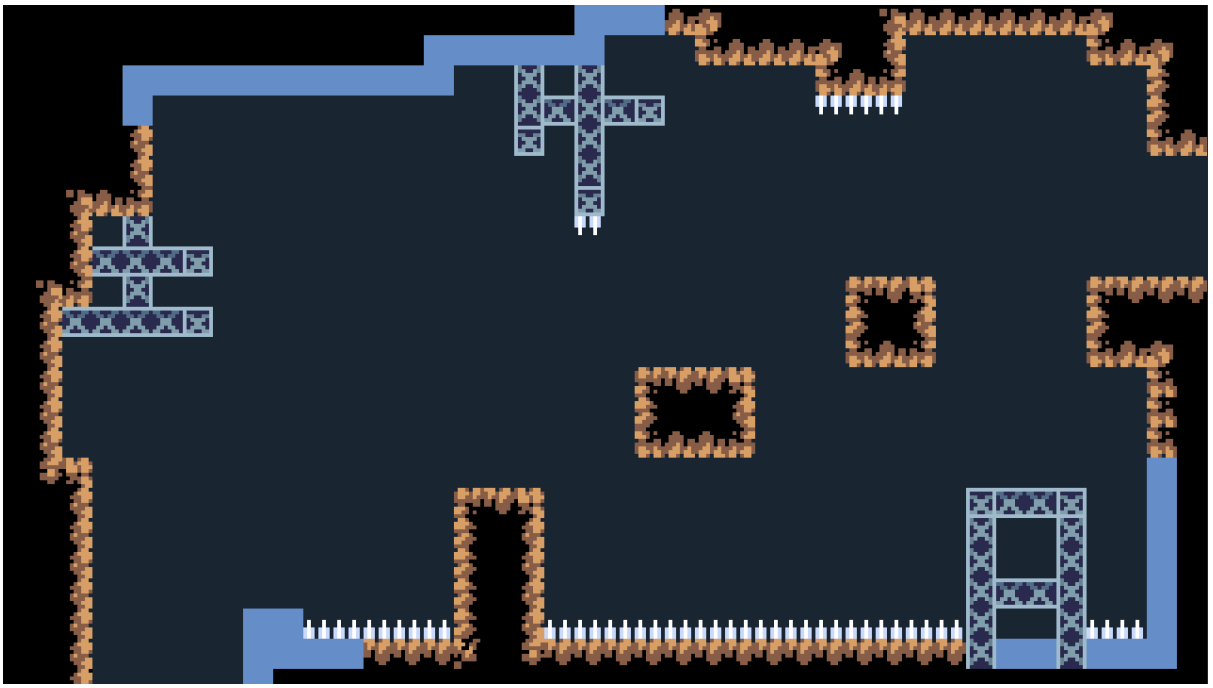


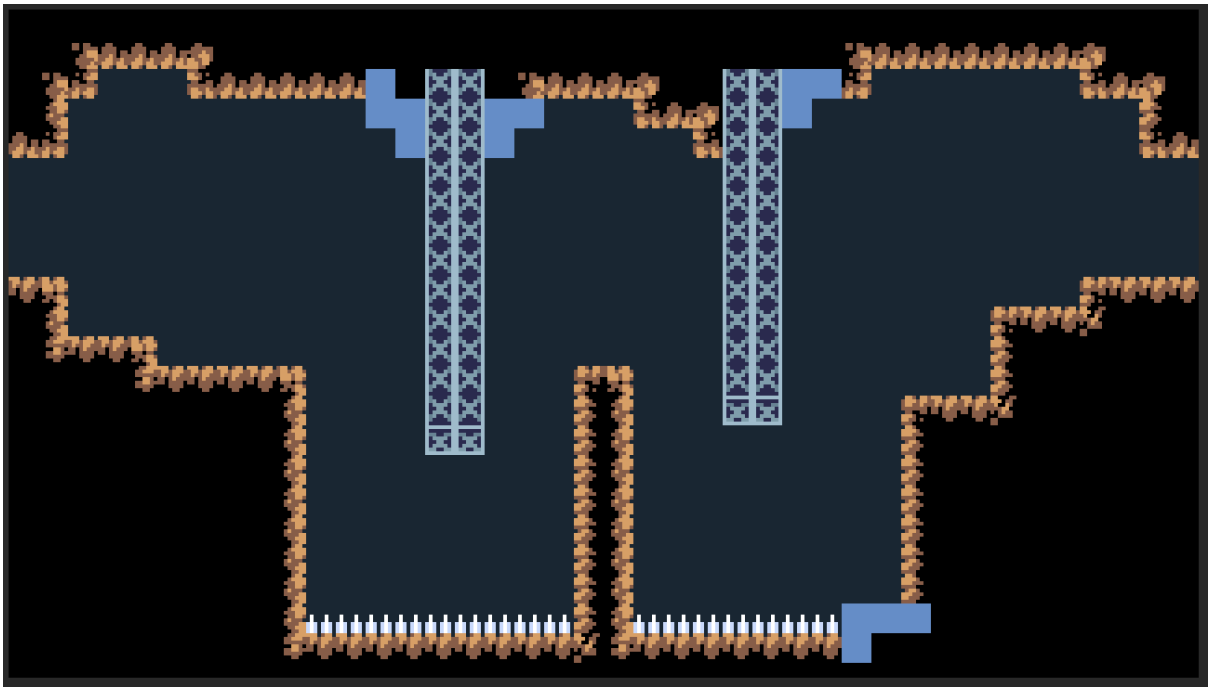
Appendix D: Input Manager Configuration

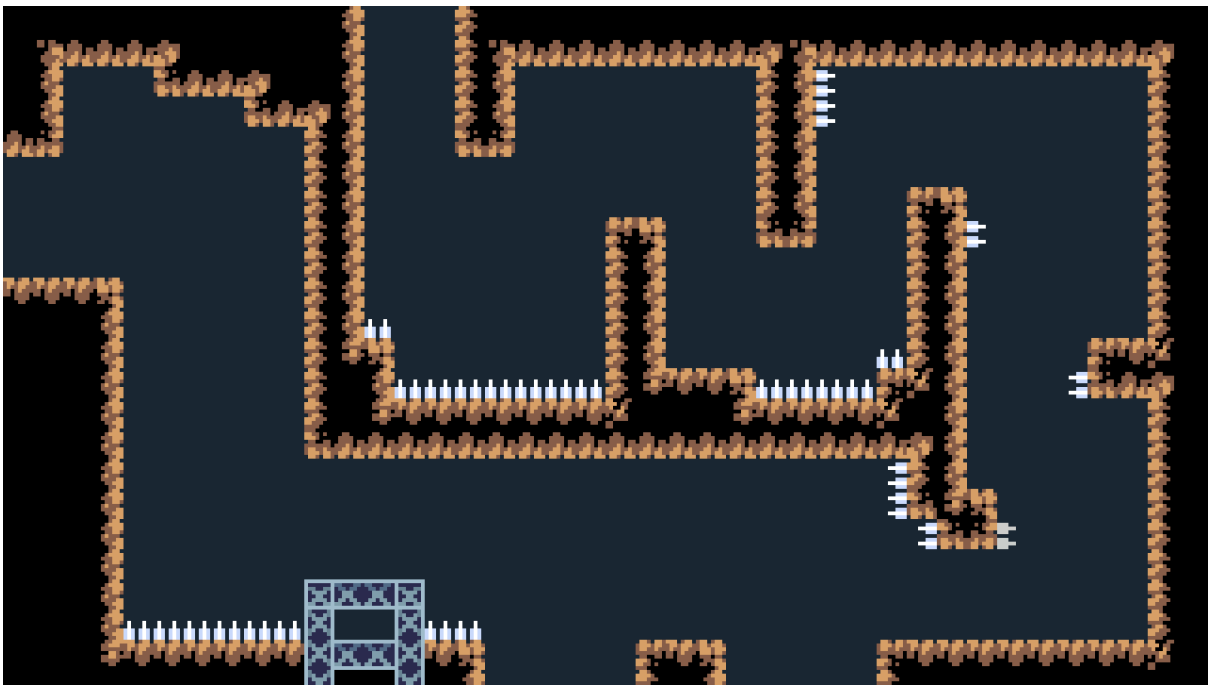


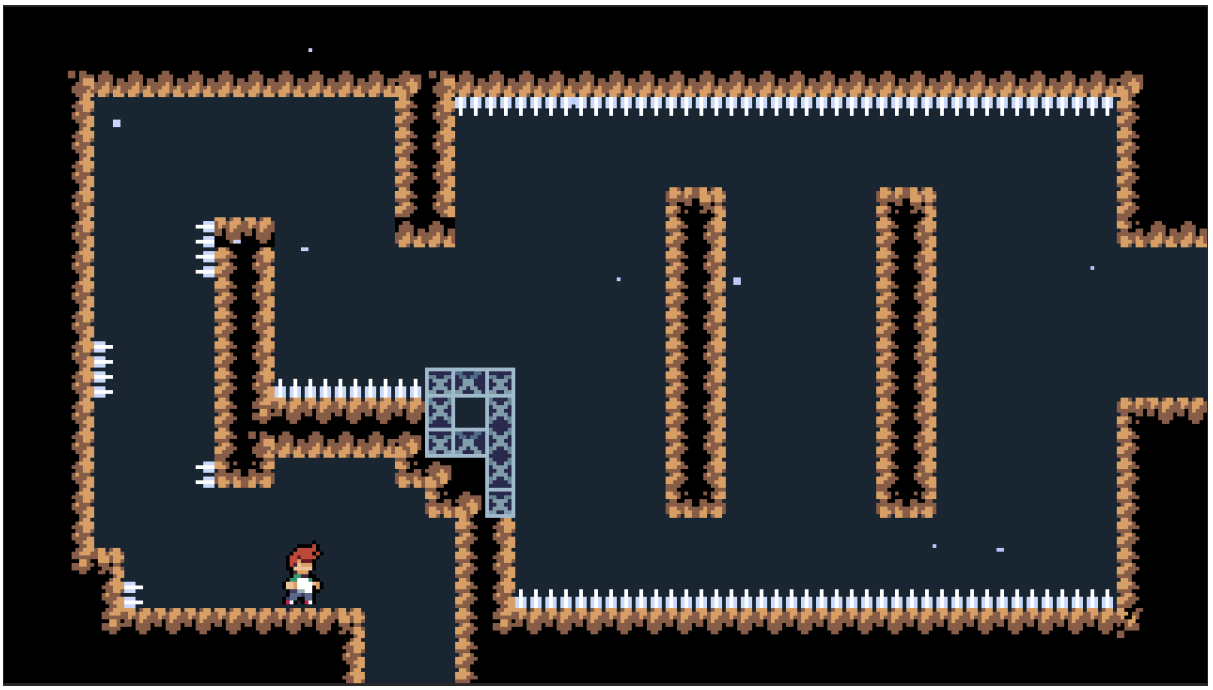
Appendix E: Screenshots of Finished Levels











Appendix F: Unfinished Wall Climb Code

```
private void WallGrab()
{
    if (!canMove)
    {
        return;
    }

    rb.gravityScale = 0f;

    // Unfinished climbing jump
    if (horizontalDirection > .2f || horizontalDirection < -.2f)
    {
        rb.velocity = new Vector2(rb.velocity.x, 0);
    }

    rb.velocity = new Vector2(rb.velocity.x, verticalDirection *
    climbSpeed);

    // Decrease player's stamina during climbing
    if (rb.velocity.y == 0)
    {
        stamina -= staminaCost * Time.deltaTime;
    }
    else
    {
        stamina -= staminaCostClimbing * Time.deltaTime;
    }

    // Animation warning low stamina
    if (isTired)
    {

    }

    if (stamina <= 0)
    {
        wallGrab = false;
    }
}
```