MSC DISSERTATION

Developing a Video Game Prototype to Test the Effectiveness of Intuitive Level Design

Abstract

Designing a game to be intuitive can allow new players to immediately engage with the gameplay, rather than becoming bored or frustrated because they don't understand it. This study delves into the theory behind what makes games intuitive and explores different methods for designers to subtly convey information to their players. A game prototype was created to test whether intuitive level design affected player's performance and enjoyment. Twelve participants tested the game but no statistically reliable difference was shown between the two levels. Despite this, the study showed a lot of promise and discovered many useful recommendations for future work in this field. Significant differences in player behaviour were observed between the two levels and recommendations have been provided on how to better capture these differences as concrete data.

Stuart Paterson

Sp96@hw.ac.uk

Declaration of non-plagiarism

I, Stuart Paterson,
confirm that this work submitted for assessment is my own and is expressed
in my own words. Any uses made within it of the works of
other authors in any form (e.g., ideas, equations, figures, text,
tables, programs) are properly acknowledged at any point of their use.
A list of the references employed is included.
Signed: Stuart Paterson
Date: 15/08/2019

Acknowledgements

Stefano Padilla

Keith Nimmo – For creating all the art assets used in the game prototype.

All participants who playtested the game.

Contents

Abbreviations & Definitions

2D – Two dimensional

3D – Three dimensional

AWS – Amazon Web Services

FPS – First person shooter

GDD – Game design document

GUI – Graphical User Interface

MoSCoW – Must, Should, Could, Won’t requirements analysis

NES – Nintendo Entertainment System

PC – Personal Computer

Platformer – A genre of video game where the player character must travel from platform to platform, usually by jumping. They often include additional obstacles such as enemies or hazards.

SEM – Standard error about the mean

UE4 – Unreal Engine 4

UI – User Interface

1 Introduction

1.1 Overview

An intuitively designed game is easy for new players to jump straight into and play without long-winded introductions. Long tutorial-sections or walls of instructional text can cause players to become bored before they even get the chance to play the game. Alternatively, players may struggle to understand a game that is not properly explained which will add unintended difficulty and may make them lose interest or grow bored. Designing a game to be intuitive means the designer can convey information and instructions to the player without them even realising it is happening. This helps the player stay engaged and immersed in the gameplay, rather than having to worry about the technicalities of the game.

Intuitive design is most important in the early stages of a game where the player is unfamiliar with the game mechanics. It can also be used later in the game when introducing new mechanics the player has yet to encounter. Many games develop intuitive designs by playtesting and observing how new players interact with the game (Fullerton, 2008). These games develop iteratively from this feedback to repeatedly improve how intuitive they are. This is effective but time consuming. This study aims to test whether applying intuitive design principles can offer immediate improvements to level design without the need for iterative development.

The study involved developing a small video game prototype and then using it to test intuitive level design. Two levels were created within the game, one acted as the control level and the other used intuitive level design. Twelve people tested the game (six per level) and data was collected based on their experience. Completion time and number of failures for each participant were measured for each section of the levels. After completing the game participants were given a short survey to gauge how much they enjoyed the game.

This dissertation will cover:

- The aims and objectives of this study.
- The different aspects of intuitive design for games and the benefits each of them can provide.
- Techniques for implementing intuitive design.
- Case studies of games making use of intuitive design.
- The design and development of a game prototype for testing intuitive design.
- The test procedure used for gathering data from the game prototype.
- An evaluation and discussion around the results data gathered.
- Conclusions drawn from this study and recommendations for future work investigating intuitive design.

1.2 Project Aim

The aim of this study was to test the effectiveness of intuitive level design techniques. Both the player's performance in the game and their enjoyment was used to determine how effective the intuitive design was. A game prototype was developed to use for the testing and a level designed using intuitive practices was tested against one without to establish if it offered any noticeable improvement.

1.3 Project Objectives

- Create a stable game prototype.
- Create a well thought out game design for the prototype. The design should provide the tools to allow testing intuitive level design with ease. The game mechanics, premise and technology will be included in this design.

- Create two levels within the game. One using intuitive design and one with a simpler layout.
- To test if intuitive level design allows players to understand the game faster.
- To test if intuitive level design increases player enjoyment.
- To test if intuitive level design increases how intuitive the player finds the game.
- To test if intuitive level design decreases the player's perceived difficulty of the game.
- To analyse the data collected and evaluate whether there are any noticeable trends.
- To calculate whether any trends in the data are statistically significant.
- To formulate insights and design recommendations based on the work completed for this dissertation that could aid future work in this field.

1.4 Summary

This dissertation will investigate what intuitive design is and how it can be applied to video games to improve player experience. A game prototype will be developed to test several hypotheses related to intuitive design. Intuitive design was hypothesised to improve player performance and increase player enjoyment. The data collected was analysed and design recommendations and insights were formulated from the experience gained during this study. These design recommendations could be used to improve future work in this field.

2 Literature Review

2.1 What Intuition Means for Games

2.1.1 Defining Intuition for Games

If a game is designed to be intuitive it can allow the player to quickly understand the controls, mechanics, layout and many other features of the game. Allowing the player to quickly grasp the technicalities of the game allows them to focus on actually playing and enjoying the game. If the player has difficulty understanding the game this can cause frustration. It also may mean the player never develops an interest in the game and stops playing because they are bored. Toshio Iwai stated "*When I design something, I am trying to create it so that it is very attractive at first sight. If players touch it, it can be understood instinctively and the players can be pulled into it very strongly*" about his game Electroplankton (Centaur Communications Ltd, 2006). He acknowledges that designing a game that is immediately understood by the player allows them to dive straight into engaging with it rather than spending time puzzling over it.

What makes a game intuitive is a complex combination of many factors. Shirinian(2013) divided the intuitive "mapping" of controls to actions into 4 characteristics:

1. *Physical or spatial analogies*: The location in space gives away the function. For example, the left arrow key being used to move the player character left in the game.
2. *Cultural standards*: Preconceptions a user is likely to have based on societal norm. For example, the QWERTY keyboard is viewed as more intuitive than the Dvorak keyboard because QWERTY keyboards have been so widely used for many years (Shirinian, 2013).
3. *Biological aspects*: A parallel with a real biological relation may make certain controls intuitive.
4. *Principles of perception*: Physical arrangement can show relation. For example, grouping functionality relating to one object separately from other functionality.

Although Shirinian(2013) applied these principles purely to controls they can also be applied to gameplay mechanics.

2.1.2 Physical or spatial analogies

Physical analogies are important but are often so intuitive that they are not considered. For example, if the player triggers an explosion to their left they expect to be thrown right, due to their understanding of realistic physics. The Gestalt principles of proximity shows that people perceive objects as being grouped based on their physical location (Kubovy & Pomerantz, 2017). Take the example in Figure 1. There are 3 buttons and 3 objects. Most players would assume that the buttons correspond to the objects above them. The designer will often create physical analogies without even considering it, but if they are neglected this can result in extremely confusing features.

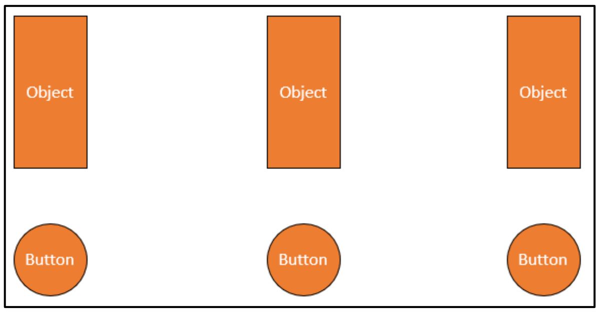


Figure 1: An example of the Gestalt principle of proximity

2.1.3 Cultural Standards

Cultural standards are important to consider in video games, especially for appealing to more experienced players. For example, it is almost universal in first person shooter games on PC that WASD is mapped to moving the player and left click is used to shoot (Shirinian, 2013). If a game were to be released with differing control schemes this would immediately be an additional obstacle for players trying the game for the first time. Adhering to these standards when possible can smooth the learning process for players who have these prior experiences.

Cultural standards are also important for the gameplay, not just the control schemes. For example, 2D platformer games such as *Super Mario Bros* (Nintendo Creative Department, 1985) have established a standard that the player should run from left to right to progress through levels (Brown, 2018). The game need not strictly follow these standards but instead the designer can make use of them. On beginning a game of *Metroid* (Nintendo R&D1 and Intelligent Systems, 1983) the player will likely run to the right as is common in other platformer games. They will then reach an impassable object and must retrace their steps to the start to explore the other direction. This immediately teaches the player they will encounter dead-ends throughout the game and will then have to explore, discover a new item and use it to overcome these previous obstacles.

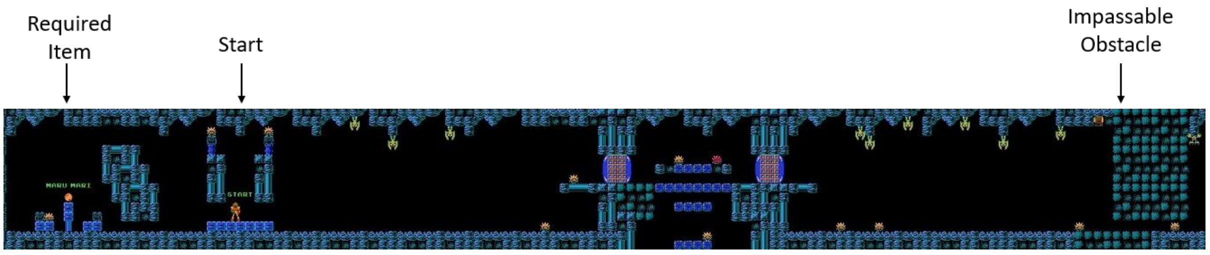


Figure 2:The first area of the map in Metroid (NES Maps, 2013)

2.1.4 Biological Aspects

The example Shirnian(2013) gives of biological aspects is volume vs frequency. When concerned with volume, most people instinctively understand that more volume means a louder sound. When it comes to frequency however, more is not necessarily so easily understood. People are less likely to associate more frequency with a higher pitched noise, despite that being the mathematical definition. Biological aspects are the least well understood of these mappings, and therefore least applicable to typical game developers.

2.1.5 Player perception

Player perception is very important for ensuring player understanding in video games. If the player cannot quickly make sense of what they're seeing they must spend time deciphering this rather than actually playing the game. The Gestalt principle of similarity states that people will assume objects which are similarly displayed will behave similarly (Kubovy & Pomerantz, 2017). Taking the example from Figure 1, if different colours are applied to each button in the same layout there is suddenly a shift in the perceived functionality. Figure 3 shows the updated image, which now gives the impression that a button interacts with the object of the corresponding colour, regardless of physical grouping.

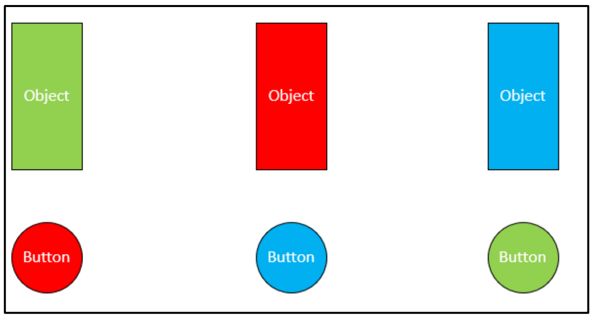


Figure 3: An example of the Gestalt principle of similarity

In *Portal* (Valve Corporation, 2007) the player must often trigger buttons to open doors. A visual path tracing from the button to the door is shown within the game to ensure the player can easily tell what each button does. This allows the player to concentrate on solving the puzzle rather than introducing needless confusion that may frustrate them. This is another example of perception being used to guide the player and an example is shown in Figure 4.

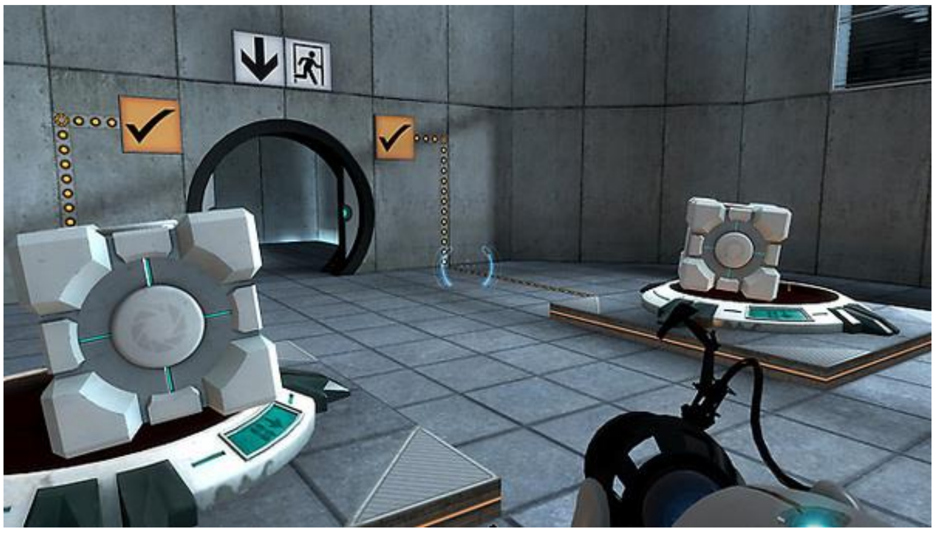


Figure 4: Screenshot from Portal showing buttons connected to door (Steam, 2007)

2.2 Intuitive Level Design

Intuitive design can be applied to several different aspects of game design. Some of these aspects are listed below.

- Controls/Input
- Game mechanics
- Level design
- User interface (Both gameplay overlays and menus)

This study specifically focuses on intuitive level design. Many of the practices explored later in this literature review are also applicable to other areas of game design. However, this study only aims to test their effect when applied to level design. The designer can use intuitive level design to convey information to the player and smooth their playing experience. This is particularly important at the beginning of games, or when introducing new mechanics (Desurvire & Wilberg, 2008).

The way a level is laid out can teach the player about the game without them realising. Take the first level of *Super Mario Bros* (Nintendo Creative Department, 1985) as an example. When the player first encounters the mushroom power up, they will be unsure whether this will be positive or negative. The designer ensured due to the confined space and high movement speed of the mushroom that even if they player tries to avoid it, they will not be able to (Anthropy & Clark, 2014). Being forced to grab the mushroom the player will then see that it is a power-up and the game has taught this without any explicit instructions.

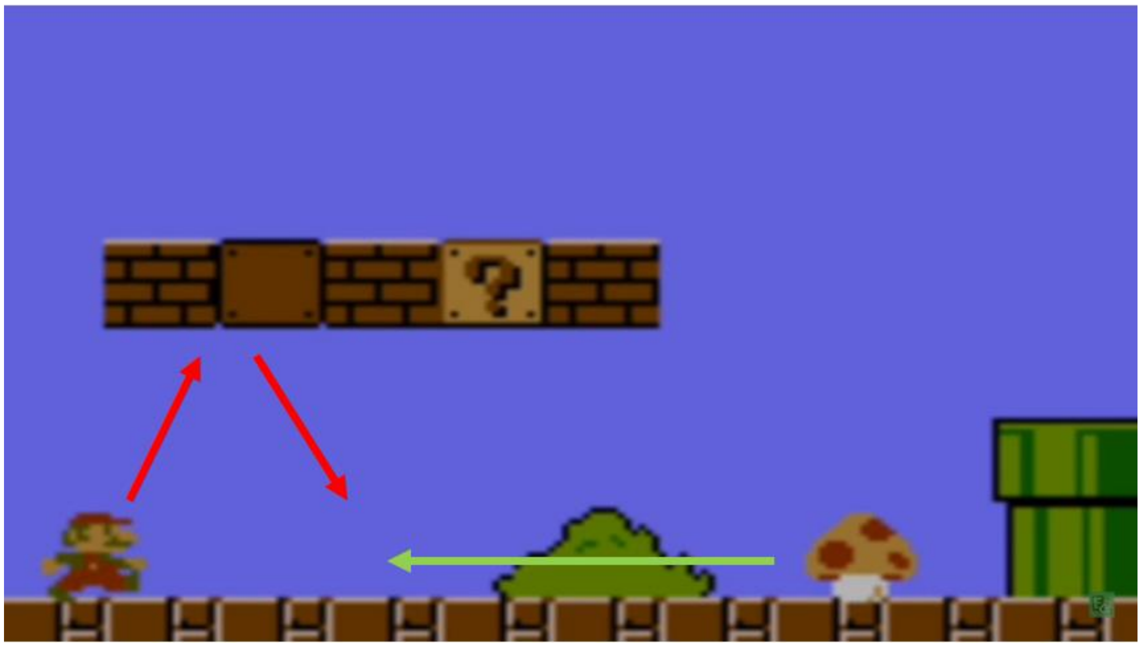


Figure 5: The beginning of the first level of Super Mario Bros. The blocks above ensure the player cannot jump over the mushroom and will likely collide with it. Image taken from (Iyer, 2017) and modified

Super Mario 3D Land (Nintendo EAD Tokyo, 2011) uses intuitive design to introduce a new core mechanic in each level and uses a 4-step system to do so (Nutt, 2012).

1. Introduce the mechanic in a simple situation.
2. Use the mechanic to complete a more complex challenge.
3. Throw a very different challenge at the player that requires them to utilise the mechanic in a different way.
4. Challenge the player to use everything they have learned in combination and demonstrate their skill.

This shows how intuitive level design can be important throughout the entire game rather than just the early stages. None of the levels in *Super Mario 3D Land* feature lengthy textual descriptions explaining the new mechanic. The simple introduction of the mechanic teaches the player the basics and then they build upon this in the more complex challenges.

2.3 Benefits of Intuitive Design in Games

2.3.1 Initial Player Engagement

Intuitive design eases the player into understanding the technicalities of the game like the controls and game mechanics. In many games these are addressed with text-based tutorials or dedicated tutorial levels that instruct the player at each step. This type of tutorial is often slow and boring, running the risk of disrespecting the player's intelligence by explaining every simple detail (Anthropy & Clark, 2014). When a player decides they wish to try a game, they have been enticed in by snippets of the gameplay. On starting the game they want to jump right in, not sit and read a wall of text or move through a slow and unchallenging tutorial level.

2.3.2 Challenge

Challenge is a very important part of player engagement and tutorial levels often offer no-risk to the player which can lead to boredom (Hoffman & Nadelson, 2010). Hoffman & Nadelson(2010) discovered that balancing the degree of challenge is very important for keeping players interested, as both too easy or too difficult a game resulted in a negative response from players. Figure 6 shows how challenge must be based on player ability to keep the player interested (Chen, 2007). This is not only true for the first levels, it also means challenge should increase as the player's ability increases when the game progresses. Intuitive design can play a key part in making the game easy to understand, which means the designer can control player challenge more easily. If the player understands the tools they can use in the game then the challenge can come from the game's content rather than "unintended difficulty" created by the player not understanding the game (Desurvire & Wixon, 2013).

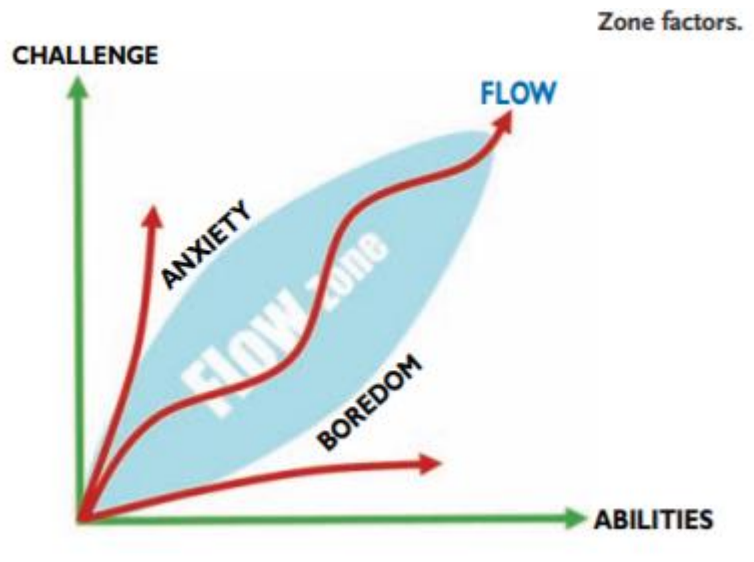


Figure 6: A graph depicting how challenge and player ability must be balanced to maintain flow (Chen, 2007)

2.3.3 Flow

Intuitive design increases player engagement by balancing challenge but it also encourages players to enter a flow state. Csikszentmihalyi (2014) describes flow as a state of "deep concentration" where the person is only responding to a "limited set of stimuli". A common sign of flow is the person losing track of time due to their complete absorption in the task (Finneran & Zhang, 2003). Player engagement and immersion are essential parts of achieving flow for video games and intuitive design can help both. Reading tutorial text or receiving instructions can break the flow of the game and interrupt the player's immersion. These types of instruction remove the player from the game world and remind them they're playing a game. Reducing the player engagement like this is bad for achieving flow whereas making the game easy to understand intuitively will allow the player to be drawn in and remain immersed. It has been shown people who reach this flow state show enhanced learning so players in flow will understand the game faster (Webster, et al., 1993) (Kiili, 2005).

2.3.4 Accessibility

Intuitive design is not only good for engagement but also accessibility. It provides benefits for both hardcore gamers and more casual players. Intuitive design can allow hardcore gamers to feel immediately familiar with the game while also making the game easier to understand for new players. Allowing less-experienced players to understand the game faster makes the game accessible to a larger audience by allowing the player to feel competent immediately, regardless of skill (Shirinian, 2013). Casual players are more likely to quit the game if they struggle to understand it, so intuitive design can ensure better player retention and a larger player base (Desurvire & Wilberg, 2008).

Removing textual instructions is not only advantageous for player immersion, it also transcends language barriers. Without as much text essential to the game more players from different cultures across the world will be able to access the game. The first level of *Shovel Knight* features no textual instructions but still teaches the player how the main mechanics work. Figure 7 shows the screen where the player must use the downwards aerial attack for the first time. The player cannot progress without using the ability, and the breakable blocks that have been previously introduced make it clear the player must somehow attack downwards. This combined with the fact that players lock into a down attack if they press down at any point in the air means the player will quickly discover this ability even if they are not aware it exists (Grow, 2014). By increasing accessibility, intuitive design can allow a game to reach larger markets and appeal to a wider player demographic.

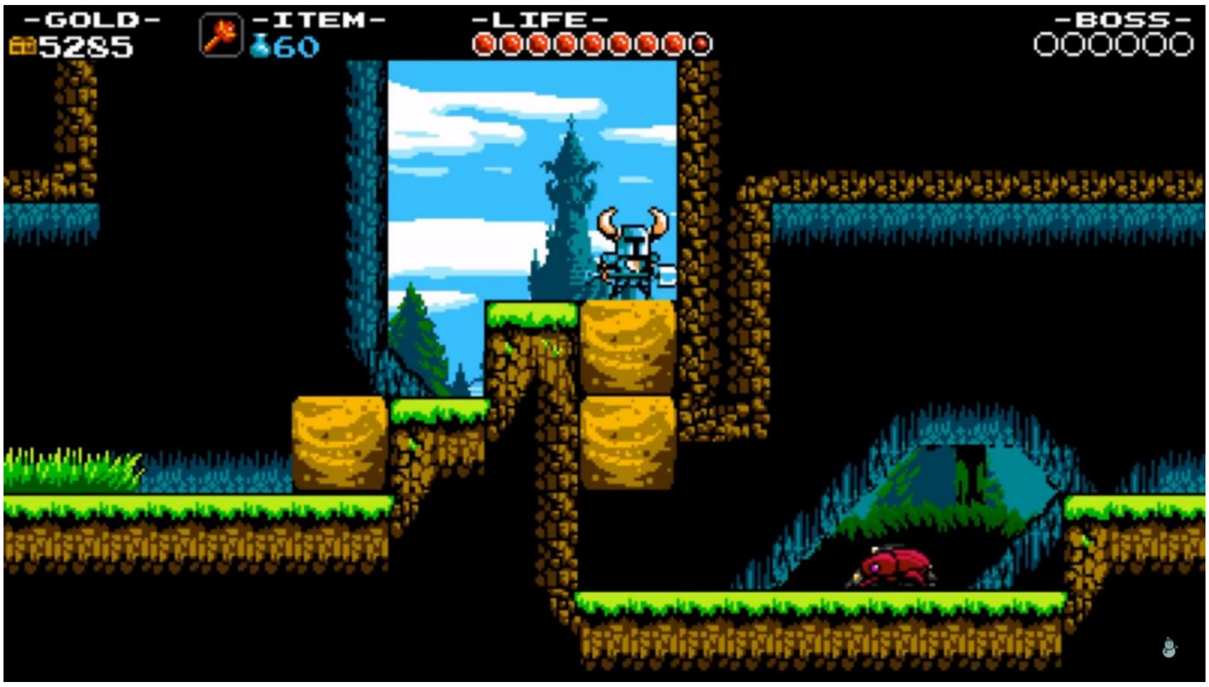


Figure 7: The first level of shovel knight requires the player to use the downwards aerial attack to progress (Grow, 2014)

2.4 Implementing Intuitive Game Design

Intuitive game design offers a lot of benefits but implementing it comes with challenges. Different techniques can be used to make the game intuitive and to teach the player about the game.

2.4.1 Experiential Learning

Experiential learning means learning from past-experiences (Kolb, 2014). Experiential learning is useful in games as the designer can control what the player experiences. Kolb (2014) theorises that the optimal learning pattern is when people; experience something, reflect on their experience then form conclusions based on those experiences.

Experiential learning can be very useful for introducing game mechanics and allowing the player to understand the overall structure of the game. *Super Mario Odyssey* (Nintendo EPD, 2017) uses experiential learning in each level to introduce new mechanics in a way that is easy for the player to understand. Each level begins by introducing the mechanic and its basic functionality in a very simple situation. This lets the player gets some practice using the mechanic and ensures they understand it's basic function. Then the difficulty is ramped up as the stage progresses and the mechanic is often used in multiple different ways (Brown, 2017). If immediately after acquiring the new mechanic, the player had to use it in several different ways to solve a complex puzzle this would be extremely difficult. By controlling the player's previous experience with the mechanic, Nintendo ensure that the player is never overwhelmed, and by ramping the difficulty up they keep the player engaged.

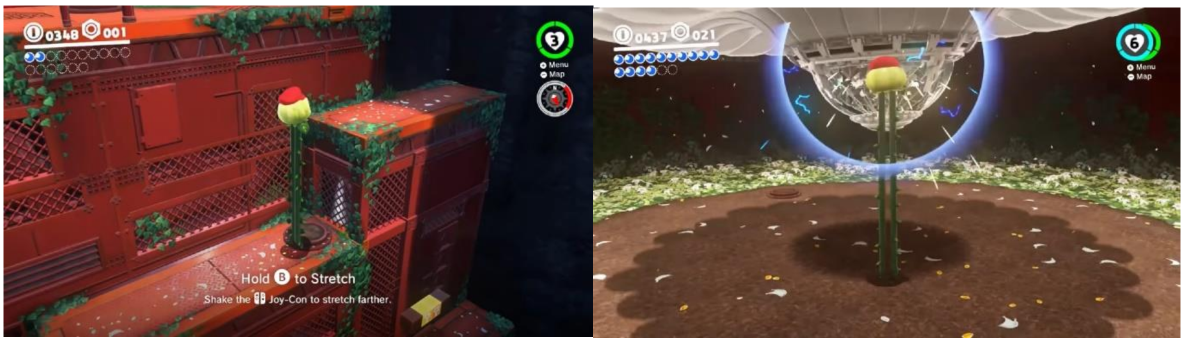


Figure 8: The uproot mechanic in Super Mario Odyssey is initially used for climbing simple steps (left). Later in the level it must be used to attack the underside of the level boss(right). Images from (Brown, 2017).

2.4.2 Productive Negativity

Productive negativity is a type of experiential learning and can be thought of as learning from mistakes (Gauthier & Jenkinson, 2018). By allowing the player to fail, the designer can allow the player to draw their own conclusions from this failure.

Controlling the impact of failure is essential when using productive negativity. If failure is too significant then the player will avoid it too much. If failure is not significant enough then the game may struggle to engage the player. As discussed earlier, failure must remain significant enough to balance the difficulty of the game. If the game is too easy players may lose interest (Hoffman & Nadelson, 2010) so steps must be taken to prevent this when encouraging productive negativity.

Super Meat Boy (Team Meat, 2010) is an example of a game that allows the player to use productive negativity and maintains suitable challenge. It achieves this by keeping levels short and increasing the number of hazards as the game progresses. Early levels in super meat boy are extremely short and only feature a small number of hazards for the player to pass. In these early levels if the player fails, returning to the start of the level is not much of a setback. Later in the game the levels feature many more hazards and returning to the beginning becomes a much larger penalty. An increased number of hazards means more challenges the player must overcome successively to reach the end of the level, greatly increasing the difficulty. Reducing the significance of failure in the early levels ensures the player won't become frustrated when failing to new hazards. Instead they learn about the game's mechanics through these failures, which provides them the skillset they need to complete the more complex challenges later in the game.

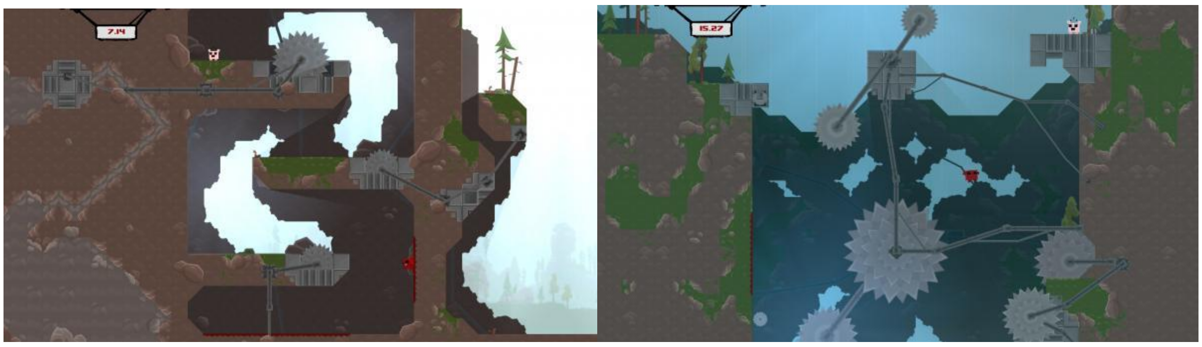


Figure 9: A comparison of two levels in Super Meat Boy. (Left) A simple level from early in the game to introduce hazards. (Right) A more complex level later in the game (Valve, 2010)

Portal (Valve Corporation, 2007) also has an excellent example of productive negativity in a different manner to *Super Meat Boy*. There is a mechanic known as the "fling" in portal where the player falls through a portal and then comes out of another, retaining their forward momentum (RPS, 2013). When the player enters the first room where the fling mechanic is required they will not have any prior experience of it. The level guides the player to the solution by limiting where the player can place portals and keeping the layout very simple. The player is immediately confronted with a diving board styled platform over a pit with a surface they can place portals on at the base. The only other area they can place portals in the room is a vertical wall above the door they entered through. A single panel protrudes from this wall, hinting the player should use it. The use of productive negativity comes into play if the player jumps into the pit. It is likely many players will either jump or fall into the pit when entering the room; either mistakenly or while attempting to explore. To get out of the pit the player must place a portal on the floor and on the only other surface they can see outside the pit, the protruding wall panel. When passing through this portal the player will be slightly flung forwards and see the exit door in front of them. The player can then conclude they could use the same technique to reach the other side if they had a little more speed. To add to this, the player's portals are now in the correct position for them to do so. The player can then jump from the platform into their existing portal and reach the goal, as shown in Figure 10.

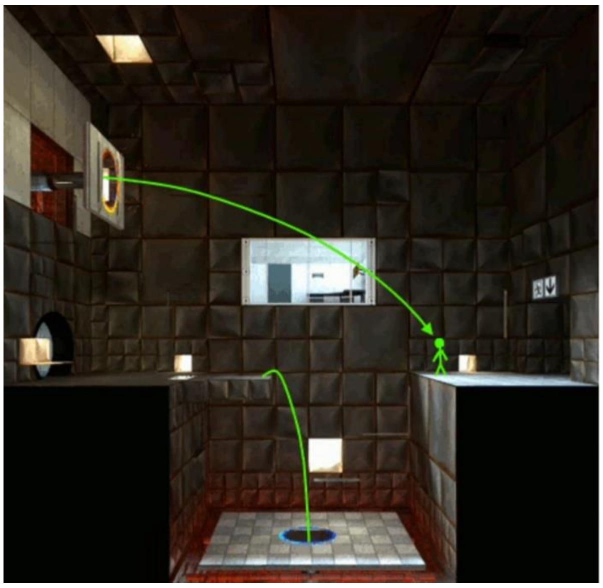


Figure 10: The first level the "fling" mechanic is required in portal (RPS, 2013)

Super Meat Boy takes away the significance of failing in early levels to allow the player to make mistakes and learn from them without getting frustrated. In contrast *Portal* cleverly lures the player into failing and in the process of returning from this failure shows them exactly how to complete the puzzle. Both techniques teach the player through failure, but in very different manners.

2.4.3 Reinforcement Learning

Reinforcement learning involves letting the player experiment to determine which actions offer a reward or penalty (Abe, et al., 2011). The player can view the outcome of actions they take then adjust their understanding of the game based on the result. There are two types of reinforcement learning; model-free and model based.

Model-free reinforcement learning involves the player only drawing conclusions from the results of their own actions. Model-based reinforcement learning extends this to include the player drawing conclusions based on observing the environment. This allows them to deduct the hypothetical result of an action without taking it (Abe, et al., 2011). Using level 1-1 of *Super Mario Bros* (Nintendo Creative Department, 1985) as an example, at the start of the level when the player first encounters an enemy they may run into it and fail. They can then draw the conclusion that that enemy is dangerous which is model-free reinforcement learning. When the player encounters the first mushroom from a question-mark block it appears on a platform above the player. It then travels forwards, falls and bounces off the green pipe. After bouncing off the pipe it moves towards the player. The player witnesses the mushroom change direction when colliding with the pipe and can draw the conclusion that any moving object will change direction when they collide with other objects (Iyer, 2017). This is true for the enemies too and so the designer has used model-based reinforcement learning to teach the player how moving objects behave.

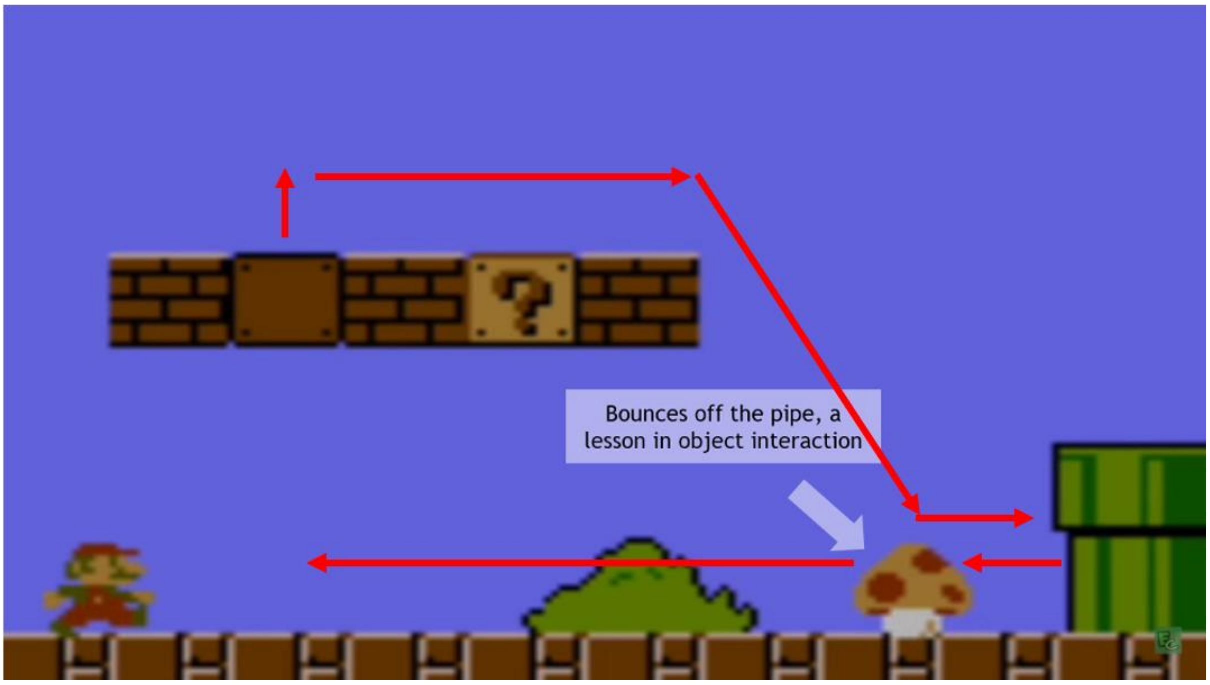


Figure 11: The first mushroom power up in Super Mario Bros demonstrates how moving objects interact with obstructions (Iyer, 2017)

2.4.4 Transformative Learning

Transformative learning involves taking previous knowledge and then adjusting it to reveal something not previously perceived (Podleschny, 2012). It takes prior experience and not only builds upon it but changes the player's understanding of the game. As discussed earlier, *Super Mario 3D Land* (Nintendo EAD Tokyo, 2011) uses a 4-step level design pattern which involves introducing a mechanic then twisting it to surprise the player (Nutt, 2012). Transformative learning is used for this to show the player the mechanic can be perceived differently. For example, in World 1-4 the player must use tilting platforms to travel along rails through the level (Nintendo EAD Tokyo, 2011). At the start of the level the mechanic is that if a player is standing on one side of a platform it moves in that direction. Part way through the level it is revealed to the player that the platforms can in fact move up and down, adding a new dimension to the gameplay. The player must change their existing understanding of the platforms being constrained to their rails and utilise rising water spouts and falling to complete the remaining puzzles. After their first encounter with vertically moving from one set of rails to another the player's perception of the mechanic is permanently changed to include the additional movement dimension.

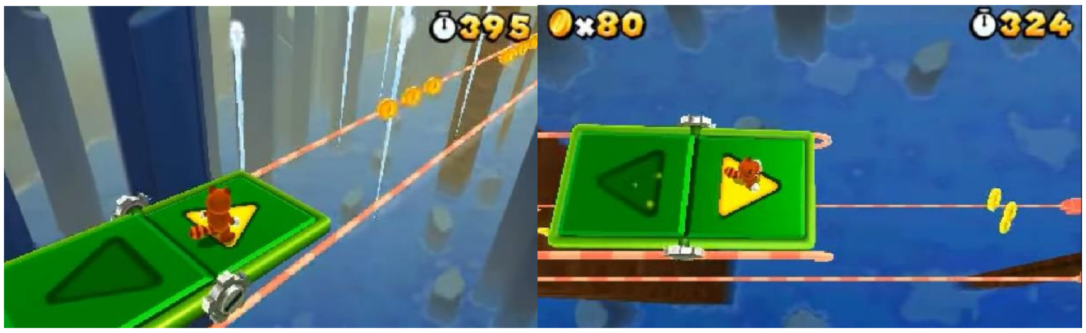


Figure 12: Tilting platform mechanic in Super Mario 3D Land. Initially the player is constrained to a set of rails (left) but later they realise they can move between rails by falling or riding water spouts (right). Images taken from (JapanCommercials4U2, 2012)

2.4.5 Iterative Learning

Iterative learning (also called recursive learning (Mitgutsch & Schirra, 2012)) is another useful tool for intuitive game design. Iterative learning involves introducing mechanics to the player a small amount at a time (Fischer, 2016). A typical iterative learning design uses two repeating steps; a new mechanic is introduced that builds on the player's previous knowledge (if any) and then the player gets to use it and learn how it works. Portal (Valve Corporation, 2007) makes an obvious use of iterative learning to introduce the Portal Gun and the steps are outlined below.

1. Player must exit first room through static portals that appear.
2. Player must solve puzzle using moving portals controlled by game.
3. Player must solve puzzle using a portal gun capable of shooting one colour of portal. The other portal is already placed in level.
4. Player gains ability to shoot both colours of portal and must solve the remaining puzzles using this.

Recognising that moving through space using the portals may be difficult for the player to understand, Valve cleverly introduced the mechanic in iterations, with each building upon the last and adding new depth.

2.4.6 Antepieces

An antepiece is a simple task that precedes a more complex challenge to give the player hints on how to solve it (tvtropes, 2019). The antepiece itself offers no challenge but by completing it the player gains knowledge they can use later. This can be used in game design to make difficult challenges easier to approach. In *Portal* (Valve Corporation, 2007) the player is forced to incinerate the "weighted companion cube" before approaching the final boss GLaDOS. When the player fights the final boss they must use an identical incinerator to destroy parts of GLaDOS, a mechanic they have just learned to use through the antepiece (RPS, 2013).

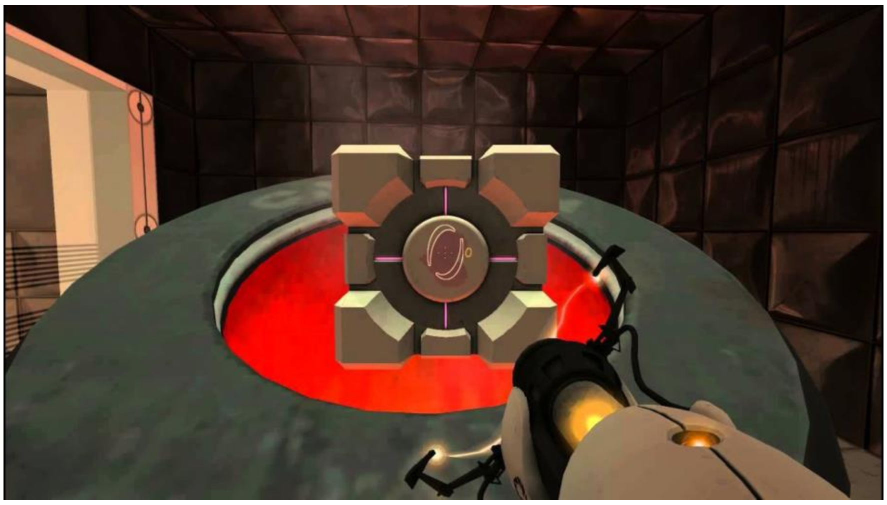


Figure 13: The player incinerating the weighted companion cube in Portal. This teaches the player how to use the incinerator which they will require in the upcoming boss battle. Image taken from (astral, 2014)

2.4.7 Designing around a core mechanic

Designing a game around a single core mechanic can help make it more intuitive to the player. In this model the core mechanic is the central feature of the game and other mechanics and the narrative are all built around it (Kim, 2012). *Super Mario Odyssey* (Nintendo EPD, 2017) is an excellent example of this structure of game design. In *Super Mario Odyssey* the core mechanic is the hat throw (Nintendo EPD, 2017). Mario's hat can be used to jump, attack enemies, possess creatures, collect coins and many more actions. To add to this, possessing creatures is an extremely detailed mechanic. There are a huge number of different creatures Mario can possess and each one has different abilities that are used to solve different challenges. The many different mechanics in *Super Mario Odyssey* would confuse the player if Mario was always capable of doing them. The game's design choice to make Mario interact with everything through his hat means if the player encounters an object or situation where they don't know what to do, they know that they must use the hat and this helps them to understand the situation quicker (Brown, 2017).

Although the hat in *Super Mario Odyssey* is a single core mechanic, it still holds a large amount of complexity due to the many different interactions. Designing around a core mechanic does not mean the mechanic must have a huge number of different possibilities and interactions. The game *VVVVV* (Cavanagh, 2010) uses the core mechanic of being able to flip gravity. The game's creator describes it as *"a platform game all about exploring one simple mechanical idea"* (Cavanagh, 2010). The game prides itself on using a simple single mechanic and introducing complexity through level design. The wide range of possibilities result from combining the games limited number of mechanics together to result in differing challenges in each level (Kayali & Schuh, 2011).

2.4.8 Signifiers

Signifiers are a technique used to highlight objects of importance to the player. They can show the player that something is interactable and encourage them to investigate further. For example, the question mark blocks in *Super Mario Bros* (Nintendo Creative Department, 1985) use signifiers to make it obvious to the player that these blocks differ from typical bricks. The combination of the question mark symbol and the flashing animation make these blocks stand out compared to the other static elements in the scene (Iyer, 2017).

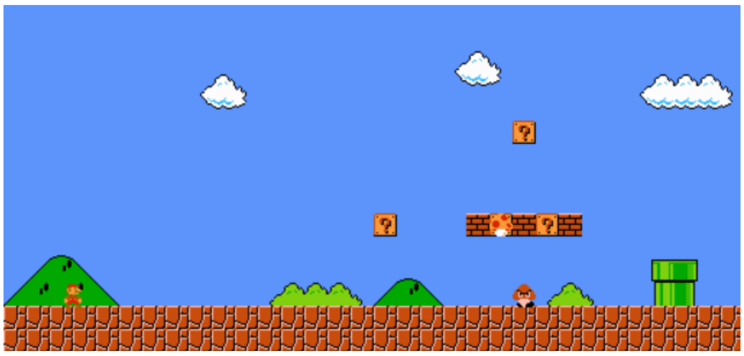


Figure 14: The question-mark blocks stand out in the initial scene of Super Mario Bros. Their flashing animation and mysterious question-mark symbol pique the player's interest. Image from (NES Maps, 2008)

2.5 Integrating Intuitive Design

The techniques discussed previously are excellent at teaching the player how to play the game without explicitly instructing them. However, these techniques must be applied in ways that integrate them into the game naturally. If they are not, then rather than feeling intuitive to the player, the player will see that the game is leading them. This may break their immersion or reduce their satisfaction when they make progress in the game. Therefore, methods must be devised for integrating intuitive design into the gameplay seamlessly.

Intuitive design should give the player hints and point them in the correct direction for understanding the game. The goal is to give the player the necessary tools they require to work it out for themselves. Spelling it out too obviously will decrease the challenge of the game which will mean players might start to lose interest (Fischer, 2016). *Half-Life 2* (Valve Corporation, 2004) uses an excellent example of integrating intuitive design into the narrative naturally so that the player will not realise the designer is guiding their hand. In *Half-Life 2* the player possesses a gravity gun which can be used to move objects around. The gravity gun can be used to fire saw blades at enemies to defeat them, and Valve used a very clever technique to introduce this mechanic.

The first time the player encounters saw blades they are embedded in a doorway and block the player from passing. When the player uses the gravity gun to remove one so that they can pass, this triggers a zombie to stumble around the corner. The player, who is already aiming at the zombie due to its placement, will most likely hit the fire button and shoot the saw blade through the enemy (Brown, 2015). This is a much more eloquent solution than say, locking the player in a room with a zombie and a saw blade and not letting them progress until they figure it out. This type of seamless design makes the game truly intuitive as the player create conclusions on their own and doesn't realise the designer is leading them.

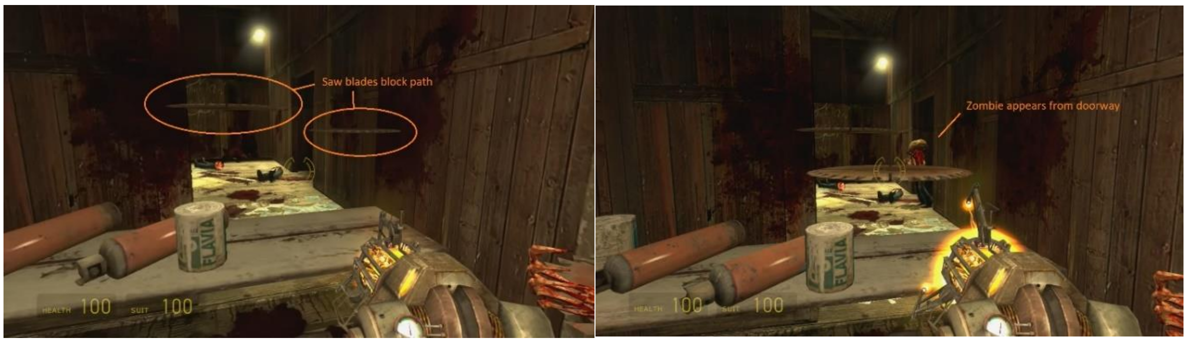


Figure 15: The introduction to the saw blade mechanic in Half-Life 2. Images taken from (Brown, 2015)

2.6 Selection of Technology

2.6.1 What is a Game Engine?

Intuitive design has been discussed in depth, but in order to implement these design techniques a prototype is required. There are many technological options when developing a game, and this section describes the best methods for this study.

The game could either be created entirely from scratch, or by using a game engine. There are many game engines and they offer a wide range of different functionality. However, they all have the same core purpose, to provide commonly used functionality so that the developer doesn't have to create it themselves. This usually includes support for; physics, 2D and/or 3D graphics, animation, sound, GUI, networking and many other features. Using a game engine saves a lot of time compared to programming the game from the ground up (Gregory, 1970).

2.6.2 Selecting a Target Platform

A lot of different platforms support video games and using an engine can make many of them easy to develop for. Table 1 shows a comparison of the three main platforms for video games. Players typically use one platform to play. Therefore releasing a game on multiple platforms will allow it to reach the largest market. Porting the game to more than one platform requires additional development time so this should only be done when the benefit outweighs the cost.

Table 1: A comparison of different video game platforms

Platform	Advantages	Disadvantages
PC (Windows, Linux, Mac)	• Allows keyboard or controller for input. • Can be portable if run on a laptop. • Easy to save data (e.g stats for evaluation). • Easy to develop for. • Can be most computationally powerful platform with correct hardware.	• Performance can vary a lot due to wide range of hardware specifications. • Separate builds needed for different operating systems. • Compatibility depends on operating system version.
Mobile (Android, IOS)	• Allows touch controls. • Portable for testing. • Most people have mobile phones.	• Only touch controls. • Computing power may be a limitation.
Console (Xbox, Playstation, Switch)	• A lot of players only use consoles. • Platform is dedicated to games.	• Difficult to develop for. • Not easily portable. • Difficult to save statistics on. • One type of controller per platform.

Developing for PC is the simplest and most versatile platform, but care must be taken to ensure compatibility. The huge range in performance specifications, operating systems and software versions means usually games will need to be designed to be adaptable to ensure they run. Mobile offers a completely different experience with portability and touch controls. Mobile games typically appeal to a more casual audience than PC or consoles. Consoles can be difficult to develop for, but the advantage is that a large audience use them exclusively to play games. Therefore, developing for console makes the game available to a large number of potential players.

2.6.3 Selecting a Game Engine

It can be very beneficial to use a game engine to produce a game, saving time and effort. Some game engines are very high level and may even require no programming, such as Scratch. At the other end of the spectrum, Unreal Engine 4 supports coding in C++ and the engine source code can be accessed by the developer to tweak the behaviour to their exact specification. The engines that offer the most flexibility and complex features often have the steepest learning curve whereas the higher-level options can be easier for beginners.

A game engine usually has its own strengths and weaknesses meaning different engines can suit different games. To give an extreme example, Ren'Py is an engine which can be used to easily create visual novels and only requires some basic python programming (Ren'Py, 2019). However, it cannot be used to produce any other type of game. The more common engines like Unity and UE4 can be used to make a wider variety of game genres. Table 2 shows the advantages and disadvantages of some of the most popular game engines.

Table 2: A comparison of game engines (Christopoulou & Xinogalos, 2017) (Burrows, et al., 2019) (GameDesigning.org, 2019)

Engine	Advantages	Disadvantages
Unity	• Exports to a lot of platforms. • Free to use. (Only charged when game makes over a $100k a year) (Unity, 2019). • Good documentation. • Large number of features. • Suited to small teams. • 2D and 3D.	• Performance may be an issue (higher overhead cpu cost and less space for optimisation). • Source code isn't open. • Inferior graphics to UE4.
Unreal Engine 4 (UE4)	• Excellent audio-visuals. • Free to use. • Open source. • 2D and 3D. • Non-programmers can use blueprints to edit code. • Programmers use C++ so easier to optimise code.	• Suited to larger teams with individuals for each role. • Engine is complex and takes time to learn. • Documentation is good but inferior to Unity.
Lumberyard	• Amazon web services (AWS) integration. • Good for cloud AI and multiplayer. • Free to use.	• Web services cost money. • Less features than UE4 and Unity.
GoDot	• Open source and free to use. No royalties. • 2D and 3D.	• Less features than UE4 and Unity. • Documentation inferior to UE4 and Unity.
GameMaker	• Doesn't require programming knowledge so more accessible.	• Imposes a lot more limitations than programming engines. • Free version doesn't include most features.

The two most powerful engines with the most comprehensive list of features available are Unity and UE4. These two are distinguished by their complexity and ease of use. Unity is excellent for rapid prototyping and developing with small teams. There is excellent documentation and the engine is easy to understand and use. UE4 is more difficult to use than Unity but provides additional development power for this cost. UE4 is best suited to a game that requires the ability to tweak the engine code and have full control of the technology behind the game to achieve the required results. UE4 works best with larger teams that consist of individuals specialised in different areas (Christopoulou & Xinogalos, 2017). This is because the additional complexity requires more effort and time. To summarise, Unity is well suited to teams where manpower and time are limited, and the game specifications are within the functionality that Unity provides. UE4 is suited to larger teams and projects that require modification to the engine tools in order to achieve the precise desired results.

Rather than attempting to compete with Unity and UE4, most other engines try to offer a more specialised niche. For example, Amazon Lumberyard's main marketing point is the AWS integration and GoDot is like Unity but open source. These engines can save even more development time than a generic engine like Unity when making games that require their niche role. However, this specialisation makes these engines poorly suited to use for other types of games.

2.6.4 Technology for this Study

An engine was used for the prototype required in this study due to the timescale of the project. The tools provided by an engine will be more than adequate provided the right engine is picked, meaning there is little to no benefit to creating an engine from scratch. A lot of time was saved not developing the engine which allowed more time for developing the game and carrying out the testing.

Based on the findings in Table 2, Unity was the most appropriate engine for the prototype. The application was be developed by a single developer which means Unity was more appropriate than UE4. The lack of access to the engine's source code was not an issue for this prototype because the supplied features in Unity were adequate. The game didn't include any revolutionary features and the default Unity physics system was be suitable. Although UE4 offered better audio-visuals than Unity these were also not important for this project, especially for a 2D game. Unity is easily capable of providing the necessary audio-visuals.

Unity is free for any game making less than $100000 per year (Unity, 2019). The prototype in this study will not be published commercially so Unity can be used free of charge. Unity required a greater performance overhead than making the game without an engine, but this was not an issue. The game was run on one machine rather than being distributed to many users, so the computing power was known beforehand. During development the performance was monitored and tweaked to ensure the game runs adequately for testing.

Windows was the ideal platform to build the prototype for. This made development of the game simpler and ensured frequent testing was easy. The issues with compatibility and performance were circumvented because the game was only required to run on one machine. The game was built to run on the laptop that was used for collecting test data, meaning most issues were discovered during development and fixed before testing. Running the game on a laptop allowed it to be portable which made travelling to test participants to collect data easy. Windows allowed a range of input devices to be used to control the game. The game was designed for controller and an xbox one controller was used for testing.

2.7 Review Conclusions

This review has explored what intuitive design means for video games. Allowing the player to immediately understand a game by providing the tools for them to work it out themselves can provide many benefits. The main goal is to ease new players into the mechanics of the game and allow them to focus on the gameplay rather than the technicalities. In a similar vein, intuitive design can be used later in the game when introducing new mechanics. Several intuitive design techniques have been explained and a summary is shown in Table 3.

Table 3: A summary of the intuitive design techniques explained

Design Technique	**Summary**
Experiential learning	Shape the player's experiences to teach them about the game.
Productive negativity	Allow the player to fail and learn from their failures. Control the impact of failure to prevent frustration or boredom.
Reinforcement learning	Let the player experiment in the game and provide positive or negative feedback to their actions.
Transformative learning	Take the player's existing understanding and re-shape it to include additional information.
Iterative learning	Introduce features in small pieces at a time, building off previous knowledge.
Antepieces	Let the player complete trivial tasks that teach them the basis for solving a more difficult task.
Designing around a core mechanic	The game is based upon a single mechanic. The player then knows when they encounter a new situation that they must use this mechanic somehow.
Signifiers	Use visual or audio cues to give the player hints about important objects.

The importance of seamlessly integrating intuitive design into the game was explained. For the design to be truly intuitive the player must not see the designer's intent and should instead think they are working out the game themselves. The discussed techniques can be applied to many aspects of game design, but this study will be focusing on level design. To test the effectiveness of these techniques a game prototype was developed. Technological options were explored for both the platform and game engine. Windows was selected as the target platform due to the simplicity of development and flexibility of input devices it provides. Unity was used to develop the game prototype. Using a game engine saved time developing the prototype and freed up this time to perform testing and collect more data.

3 Project Implementation

3.1 Project Lifecycle

3.1.1 Overview

The lifecycle of the study was split into three high-level stages; design, implementation and gathering results. The design stage involved designing the gameplay and technical aspects of the game prototype, along with establishing a playtesting procedure. The implementation stage involved programming the game within the game engine Unity. The game was implemented incrementally and tested by the developer frequently to ensure it functioned as intended. Once the prototype was finished the game was playtested by volunteers and result data was collected from each playtesting session. These results were evaluated to analyse whether there were any trends in the data.

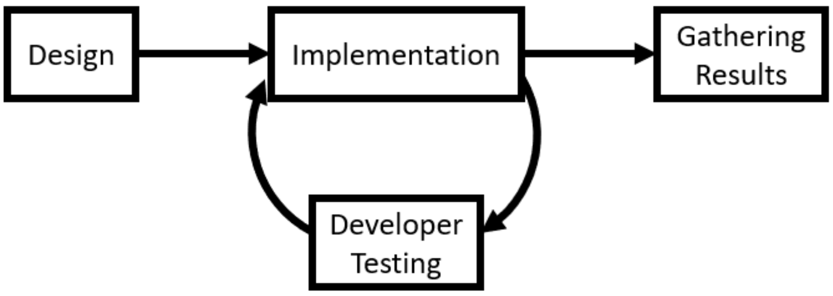


Figure 16: A high-level of the project lifecycle

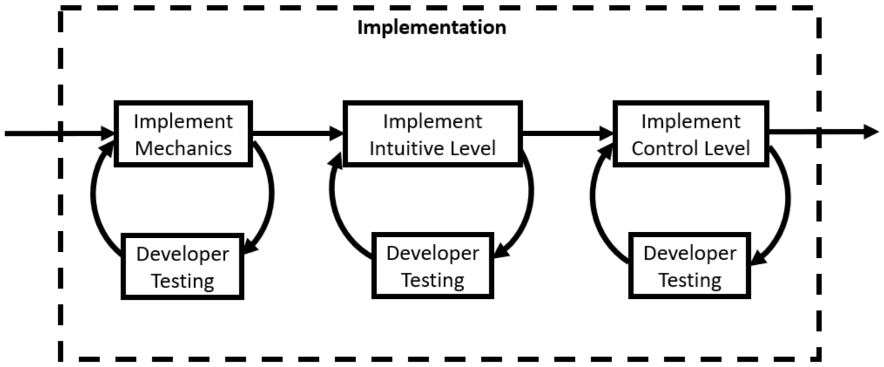


Figure 17: A breakdown of the individual stages of the implementation phase

A plan was developed which broke down the project into the main tasks involved. Timescales were estimated for each task and they were collated into a Gantt chart which can be seen in appendix A. As with any software project, some modifications to the plan were required throughout the implementation to adapt to unforeseen circumstances. In these cases, steps had to be taken to ensure the project still achieved its aims and objectives. Deviations from the plan occurred due to many reasons but having a plan in place helped to show whether the project was ahead or falling behind. Being aware of the project falling behind the plan allowed measures to be taken to bring the project back on track. The actual timescale of the project and reasons behind any deviations from the plan are discussed in section 3.1.6.

3.1.2 The Design Stage

The design stages of the project involved expanding on the simple concept review proposed in the research report to plan out all aspects of the game prototype. The design included:

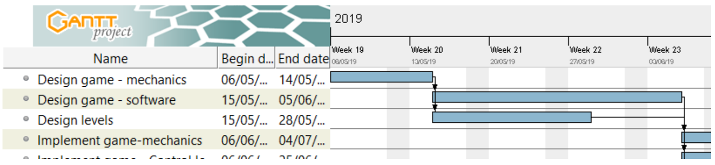
- The game prototype’s mechanics, actions and interactions.
- The game prototype’s premise and setting.
- The technical software design of the game prototype.
- The level design of the control and intuitive levels.

A requirements analysis was created as part of the research stages of the project and this defined the scope of the design. These requirements and how the final game prototype compared to them can be found in section 4.1.

To document the design as it evolved a game design document (GDD) was created (De Nucci & Kramarzewski, 2018). The game design document contained information about all aspects of the game's design, including technical specifications for the software design. The GDD kept all information about the game prototypes design in one place making it easy to reference. If the design changed then the GDD was updated to reflect these changes. This is especially beneficial when working as part of a team. Before the implementation of this project began an artist called Keith Nimmo agreed to create all the art assets required for the game prototype. Other than creating the art he was not involved in the design process, meaning he did not know anything about the structure of the game. A copy of the GDD was given to him after the mechanics and premise designs were finished to allow him to read about the design of the game. This helped him understand the structure of the game, as well as including a list of the required art assets. Creating the game design document was important as the game design is a key part of the experimental design. If the scope and functionality of the game prototype were not carefully considered then the study could have failed completely. For this reason, a well thought out design was essential to ensure the game provided enough depth to support the study, while also being completed on time.

Game design documents are excellent tools for collaboration and allow easy sharing of information. A single master copy of the design document can be made available to all members of the team, which they can then change if they update the design. This allows all team members to convey and receive updates to the game. It also provides a reference point if someone needs information about the game. The downside of this is the time and effort required to create and update the GDD. The usefulness of the GDD is lost if care is not taken to ensure it remains up to date. If members of the team think the GDD is a reliable source of information but it is outdated this may cause issues and waste development time.

The design of the game mechanics had to be completed before the software design and level design could begin. The software must be designed to support all the intended game mechanics and therefore any changes to the mechanics may have a knock-on effect on the software design. The same can be said for the level design, the levels are designed around the mechanics of the game to provide a challenging but possible experience for the player. Changes to the mechanics will affect how the player can approach the level and may either cause undesired side-effects or force the level to be changed. It was therefore decided the fastest design process would be to finalise the mechanics design and then design the software and levels afterwards. Figure 18 show the design stages displayed in the project plan Gantt chart.



Name	Begin d...	End date
Design game - mechanics	06/05/...	14/05/...
Design game - software	15/05/...	05/06/...
Design levels	15/05/...	28/05/...
Implement game-mechanics	06/06/...	04/07/...

Figure 18: A section of the Gantt chart showing the design stages

This design order avoided any extra time being spent on reflecting changes to the mechanics in the software and level designs. This approach was optimum for a single developer application but may not be suited so well to larger teams. In a larger team with multiple developers working on the design it can be more efficient to run the entire design process simultaneously, to avoid the longer lead time associated with sequential design. For a single developer the same number of tasks must be completed regardless of whether they are sequential or parallel. However, in a team parallel tasks would ensure that all developers always have work to do which makes sure no time is wasted waiting for others. In this situation having to mirror changes to another part of the design would likely be worth the advantage gained by running the design processes simultaneously.

The level design was done in sections rather than designing each level in its entirety. By splitting the two levels into sections it simplified comparing them when it came to the playtesting stage. Each level has the same number of sections and between the two levels each section is the same length. Each section was designed for both levels at once and then the complete level was assembled from the various sections. Designing the sections together helped ensure they were similar in difficulty and required use of the same mechanics. The first sections of the level introduce the game mechanics one at a time to ease the player into the game. The last few sections then put the player to the test, requiring them to use all the mechanics in combination to pass them.

3.1.3 The Game Prototype Design

This section provides a short synopsis of the game prototype's design and the reasons behind some of the design choices. The full game design document can be viewed in appendix B and contains more detail about the initial design.

The game prototype developed for this study was a short 2D platformer game. The player controls a character which can shoot projectiles out of their feet. This shooting mechanic generates a knockback effect which sends the character in the opposite direction, making the shooting both an attack and a movement mechanic. The player must progress the character through the level, passing complex movement-based challenges and avoiding or defeating enemies. The game is split into sections marked by checkpoints and if the player is defeated at any point, they will return to the last checkpoint they passed.

The game is set in space and has an overarching sci-fi theme. The player character is a human female wearing an advanced space suit equipped with rocket boots which allow the player to shoot projectiles. The hostile enemies in the game are aliens and there were two varieties implemented in the prototype. Set on an unknown planet, the level moves back and forth between the interior of a building and the planet's surface as the player progresses. An example transition between the interior and outside is shown in Figure 19. Implementing a premise is important to help engage players in the game. Without these dramatic elements the game would be too abstract and most players would struggle to remain interested (Fullerton, 2008).

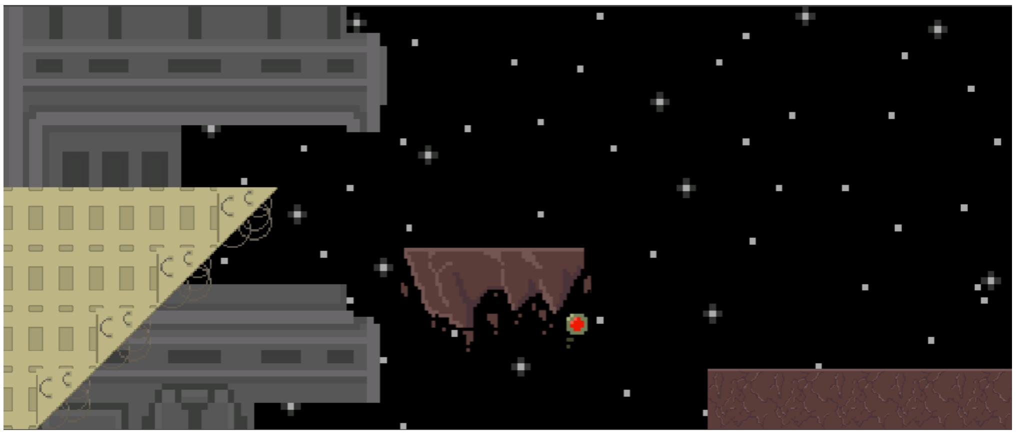


Figure 19: A transition between interior and outdoors in one of the game prototype's levels

A platformer was selected as the genre of game for the prototype because they offer the player some freedom while still enforcing linearity in the gameplay. At any stage the player can move their character anywhere within the bounds of the level and use the actions available to them however they choose. In some situations, there may be several ways the player can approach an obstacle or different actions they can use to pass a challenge. However, they must still progress through the level linearly, completing each stage in order to reach the end. This freedom within the game's constraints, or "play" as it is sometimes referred to (Fullerton, 2008), is what ensures there is enough variation in the gameplay for it to be possible to gather meaningful results. This by no means guarantees that the results will be meaningful, but without sufficient play there would be no variation in each playtest and the study could not demonstrate anything. The linearity of the game's overall structure makes comparing results from different playtests easier. Each player must complete the sections in the same order, meaning they will have had the same experience playing at each stage.

Many of the games referenced in the literature review fall into the genre of platformer. This is because intuitive level design can help to guide the player within the freedom of the game, but the linearity of the level structure makes it easier for the developer to understand how the player will approach each stage. This is also why the game was designed as 2D instead of 3D, it limits the number of ways the player can approach each situation. A 2D game was also less complex to implement and therefore faster, so it was well suited to a short timescale project. 2D assets are also simpler than 3D assets. Sprites and animations generally take much less time to implement than animated 3D models. Intuitive design techniques are not limited to 2D games, they can equally be applied in 3D games. However, a much more complex 2D game could be implemented in the timescale of this project with only one developer. A 3D game would have offered little or no advantage in this study.

The character's ability to shoot from their feet can be used in several ways to attack or move in the game. Figure 20 shows the basic actions the player can take. If they shoot on the ground, there is a small delay between their input and the character firing. This is shown by an animation of the character raising their leg and was intended to make shooting while grounded less powerful. Unlike shooting on the ground, shooting in the air is when the mechanic becomes both a movement tool as well as an attack. The character can shoot directly downwards in the air to jump a second time. Shooting forwards in the air will send the player character diagonally backwards, away from the direction they fired in.

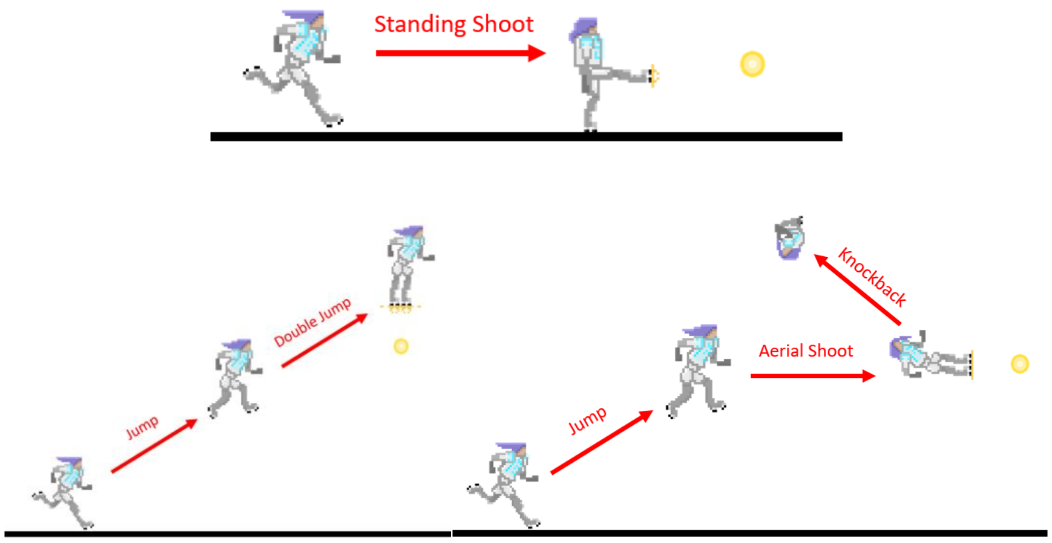


Figure 20: The basic actions the player character can take in the game prototype

The core mechanic of the character being able to shoot projectiles from their feet was designed to give some depth that the players could explore and use to ease their playing experience. The mechanic provides enough different actions that the player will have several ways to tackle each situation. For example, when encountering an enemy on flat terrain the player's options would be:

- Shoot while standing and account for the associated delay.
- Jump then shoot and account for the associated knockback.
- Jump over the enemy.
- Jump above the enemy, then use double jump to shoot them from above.

If the player has a good understanding of the mechanics, then they will use the most appropriate action for the situation. So in the situation above, they would choose not use the jump and shoot if there is no platform behind them, as the knockback will send them off the edge. Creating two effects from an action means the player must consider the advantages and disadvantages of each ability at their disposal. This makes it easier to test the player's understanding of how the actions behave. The goal was to create a mechanic that not only gave the player the tools to complete the level, but also created the ability for higher skilled players to use additional potential to improve their performance.

3.1.4 Implementing the Game Prototype

After completing the game design and software specification, work began on implementing the prototype. The functionality for all the game mechanics was created first, followed by the levels being implemented once the mechanics were finalised. Implementing the levels did not begin until the mechanics were finished because the level layout is dependent on how the mechanics function. There is more information about these interactions in section 3.1.6. Table 4 shows the main mechanics that had to be implemented.

Table 4: A description of the main mechanics that had to be implemented in the game prototype

Mechanic	**Description**
Player Character	State-machine based behaviour, reading player inputs, movement, jumping and double jumping, shooting on ground and in air (with knockback). Precise and responsive control with predictable behaviour.
Basic enemy	Movement, attacking player, responding to collisions with terrain, reset when level resets
Armoured enemy	Same as basic but different sized colliders and invulnerability to attacks from front
Switches	Link to a switchable object and change state, activated by player shooting, reset when level resets
Doors	Connected to switches, toggle between open and closed, reset when level resets
Pit/kill zone	Reset the level if the player falls out of bounds, return the player to previous checkpoint
Level	Manage setup of objects, resetting all objects when player fails, managing checkpoints and resetting player
Camera	Lazily follows player (movement is damped so the camera does not respond immediately, prevents jittering and motion sickness), movement limited between certain coordinates, movement limits updated by checkpoints
Game Controller	Manages initial menu and starting the game, loads the level the player selects
Stat logger	Writes statistics about the players performance to a file, stores completion time and number of failures per section and overall, logs which level a player has selected and current date, appends to log file if it exists so can run multiple tests in succession

As the game was implemented using Unity most of the work was done within the Unity editor. The C# scripts required to create the game logic were written in Visual Studio and work based on the game objects or project settings was done within the Unity editor. Figure 21 shows a labelled example image of the Unity editor. The key components of the editor are explained in Table 5. Section 3.3.1 discuss the reasons Unity was used to develop the game prototype in detail, as well as explaining some of Unity's features in more depth.

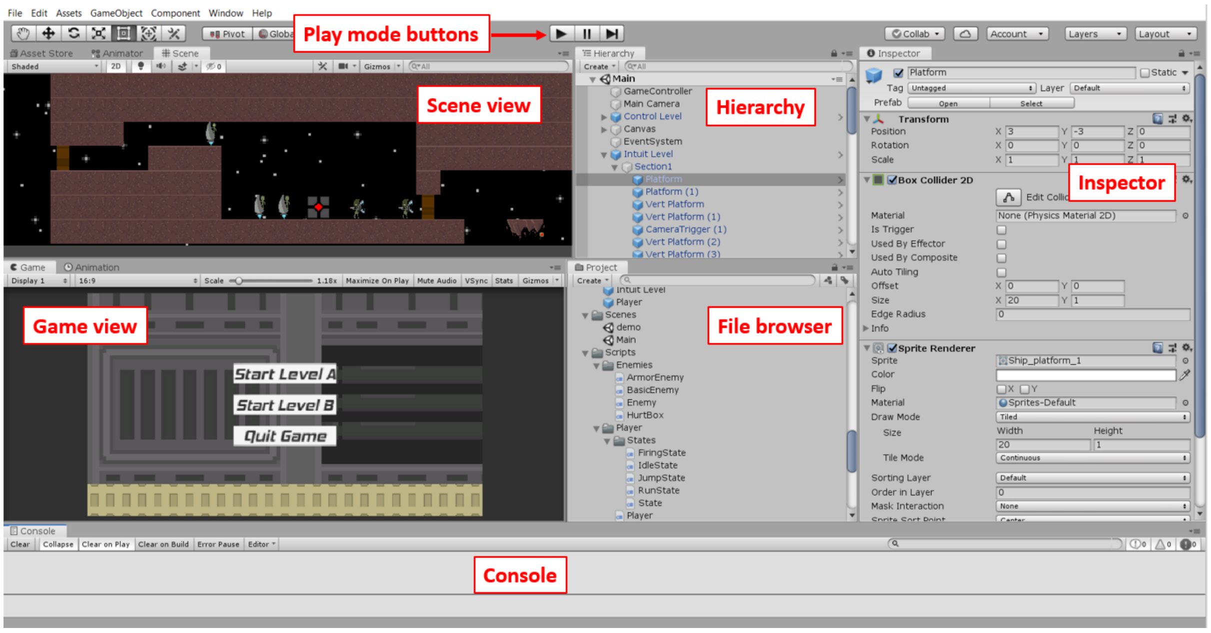


Figure 21: A labelled image of the Unity editor with the game prototype project open

Table 5: A description of the main components of the Unity editor

Component	**Description**
Scene View	Shows a visual representation of the current scene. The display can be freely panned and zoomed to view the entire scene.
Game View	Shows the output of the main camera. This is what will be displayed if the game is built to an executable and run.
Hierarchy	Shows all the game objects in the current scene.
Inspector	Allows the developer to inspect detailed information about any game object. Can also be used to edit properties
File Browser	Shows the file hierarchy within the project directory. Used to access assets such as spritesheets, C# scripts and prefab objects.
Play Mode buttons	Used to activate "Play Mode" which allows the developer to test the game in its current state
Console	Displays any error or warning messages. The developer can also add statements in their scripts to log messages to this console.

The prototype was developed incrementally, meaning that one at a time the features were implemented and then tested. Testing after adding each feature ensured that they functioned as intended and helped to locate bugs in the software. Bug-fixing is more efficient using incremental development because there is less new code added before each test, making it easier to track down the cause of new bugs. Testing was done by the developer in Unity's "Play Mode". "Play Mode" allows the developer to play the game and test any changes to the project immediately, without building the game to an executable (Unity, 2018). This allowed fast testing when adding new features or checking bug-fixes were successful.

As the final levels were not implemented until the end of the development process, features were tested in a small development level created in the Unity editor. As new features were added to the game, they were inserted into the level so they could be tested. The goal when testing was not to just play the game as a normal player but to try and create situations where the mechanics may function incorrectly. When new mechanics were added it was important to test both if the mechanic functioned properly in isolation and if it interacted with other existing mechanics as expected. The development level contained instances of all the functionality present in the game to allow everything to be tested in conjunction with one and other. Incremental development is good for testing how each mechanic works with other components of the game, but the effectiveness is dependent on how thorough the testing practices are.

The downside of incremental development is that it theoretically requires more testing time than implementing several features at once and then testing them together. In reality, incremental development may be faster because of the efficiency of making changes to the software and bug-fixing. Adding one feature at a time makes it easier to identify the cause if a mechanic does not behave as intended or a bug is affecting the game. This makes it faster for the developer to fix the issues, whereas if a lot of code has been added before the test it may take a while to root out the cause. Incremental development is less risky because it greatly reduces the chances of large delays during the testing phase

Programming the game prototype went smoothly due to the design work completed in advance. A high-level class hierarchy was planned out in the design stage and shown in a class diagram (as described in section 3.2.2). Once all the functionality was finalised, work began developing the two levels. The level using intuitive design was developed and completed first. Once the intuitive level was completed it was duplicated and then modified to create the control level. A full description of each level and the design differences is shown in appendix H.

The levels were also developed incrementally by creating one section at a time. This development style was well suited because the levels had to be divided into sections for gathering results anyway. The level was built within the Unity editor using prefab objects (discussed in section 3.3.1.5). This allowed sections to be built and edited quickly and easily, while also providing a real-time visual display of the level. Development-only functionality was added that allowed the position where the player character started to be set to a specific section. This made testing faster and more efficient because the developer didn't have to waste time playing other sections. The sections could be tested in play mode to check if they worked as intended and to assess the difficulty. Fine tuning the difficulty was important and most of the changes during testing were because of this. Care was taken to ensure the level was possible but provided enough challenge to require the player to use the correct mechanics to progress through it. If the game was too easy there was a risk there would be no difference in the results between the two levels because players could cruise through either level without a challenge.

After finalising both levels the game was exported to an executable file. This greatly improved the performance in comparison to the Unity editor used for testing. It was important the version of the game used for testing ran smoothly as performance issues would interfere with the test results. The laptop used for testing had no issues running the final version of the game smoothly, so this was not an issue.

3.1.5 Gathering Results

Gathering results for the study could not begin until the implementation of the game prototype was completed. All test participants had to play an identical version of the game for the results to be compared fairly. This was identified early in the planning stage as an area for possible delays, so it was important the test procedure was established in advance to ensure gathering results did not encounter problems.

People to participate in the study and playtest the game were recruited in advance, in parallel to the design and implementation stages. This helped prevent delays acquiring participants during the result gathering stage of the project. Six people tested each level, resulting in a total of twelve participants in the study. This was because each participant could only play one level as a new player. If participants were to play both levels, they would already understand the game when beginning the second level they played.

To make sure the test conditions were identical each participant in the study had to follow the same steps when playtesting the game. Care was taken to ensure all participants received the same instructions before playing the game, so that they all had equal understanding of the game when beginning the playtest. The steps taken are described below:

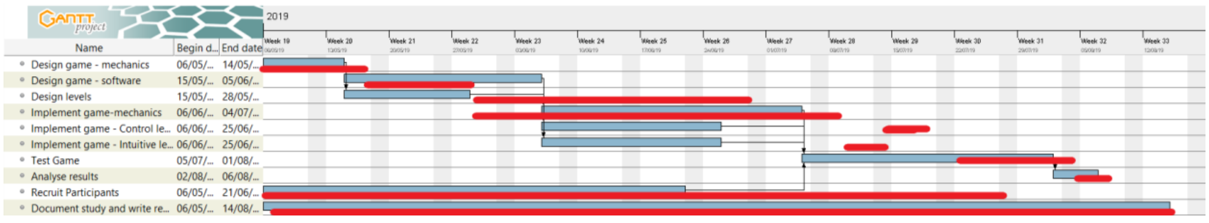
1. Before beginning the playtesting, participants were presented with the consent form shown in appendix D. The consent form describes what participating in the playtesting involves and how the data collected will be used. Participants had to agree with the conditions before they could proceed to any further steps.
2. Before playing the game, participants were shown a simple instructions graphic that showed the basic controls and actions in the game. These instructions did not include detailed information about mechanics. For example, there was no indication of the delay when shooting on the ground or the knockback when shooting in the air. For consistency all participants were shown the same image. This image is shown in appendix E.
3. Next the participant began the game and played through one of the two levels. The levels were simply labelled as A and B, so that participants were not aware of which level they were playing. The participants played the level from start to finish without any interruptions so that the time data gathered was accurate.
4. After completing the game, a log file was generated containing all that participants test data.
5. The participant was then asked to complete a short survey about the game as honestly as possible. The questions in this survey can be seen in appendix C.

All participants played the game prototype on the same laptop so that the performance was the same. The game prototype ran very smoothly on the test machine which was important as performance issues would have inconsistent effects on the gameplay during each test, which could interfere with the results. It was important for the collected data to be based on the player’s ability alone.

While the participants played through the game it automatically logged their time and number of failures in each section to a text file. When they completed the game it also logged the overall time taken to complete the level and the total number of failures. Having the game automatically log the data removes any human error in taking the measurements and ensures all the collected data is in the same format. The downside of automatic logging is it cannot be paused which is why care was taken to ensure that playtests were only carried out when there was no chance of interruption.

3.1.6 Deviations from the Project Plan

Overall the tasks involved in implementing the prototype were mostly completed on different dates to the project plan. Despite the actual dates of the tasks differing from the plan, the project was completed on time and met all the key requirements. This chapter discusses the causes of the main deviations from the project plan and the actions that were taken to correct these and ensure the project was still completed on time. Figure 22 shows the actual dates of the tasks in comparison to the plan. The unmarked initial plan can be seen in appendix A.



Name	Begin d...	End date
Design game - mechanics	06/05/...	14/05/...
Design game - software	15/05/...	05/06/...
Design levels	15/05/...	28/05/...
Implement game-mechanics	06/06/...	04/07/...
Implement game - Control le...	06/06/...	25/06/...
Implement game - Intuitive le...	06/06/...	25/06/...
Test Game	05/07/...	01/08/...
Analyse results	02/08/...	06/08/...
Recruit Participants	06/05/...	21/06/...
Document study and write re...	06/05/...	14/08/...

Figure 22: The actual dates of each tasks (shown in red) in comparison to the initial plan (in blue)

The initial game mechanical design took longer than expected. The main reasons for this was due to underestimating the scope of this section. Not only did the game mechanics need to be designed but all aspects of the game such as the premise, art style, required assets and animation storyboards also needed to be designed. These extra aspects of the design process caused the first design stage to overrun it's intended end date. To make up for falling behind so early in the project extra hours were put into the software design stage to get the project back on track. In combination with this the level design was also postponed. Although initially planned to be completed at the same time as the software design, there was no need to complete the level design early in the project life. The levels could not actually be implemented in the game until all the mechanics had already been programmed. To take advantage of this and help keep the project on time the level design was instead completed simultaneously to the implementation of the game mechanics.

The initial plan intended to implement the two levels of the game simultaneously to programming the mechanics. However, this was not possible as building the level required all the mechanics to be in place already. The level layout depends on both what mechanics are in the game and exactly how they behaved. Therefore, the levels had to implemented after the mechanics had been finalised. The initial plan also showed both levels being developed simultaneously, whereas they were completed one after another. Because the two levels had to be very similar to make the comparison fair, the levels were instead developed by creating one level and then modifying it to create the second level. This also saved a lot of time compared to building both levels from scratch.

Despite beginning early, the implementation of the game mechanics overran and caused the largest delay to the project. Due to unforeseen circumstances unrelated to the project, a lot less time than intended was spent implementing the game in late June. The commit history in Github is shown in Figure 23 and shows a large drop in work around this time. This lack of development hours meant the implementation stage ran over the intended finish date and pushed back the playtesting time. One measure taken to lower the duration of this stage was reducing the scope of the game. In the game design document, a minimum viable product (MVP) was specified and can be seen in appendix B. This MVP included a list of which planned mechanics could be cut if required to reduce the development time. The result of this was that the intended grappling mechanic and upwards shooting were removed. The number of different enemies was also reduced from three to two. These mechanics would have added additional depth to the prototype but they were not necessary and the prototype was still sufficiently complex without them.

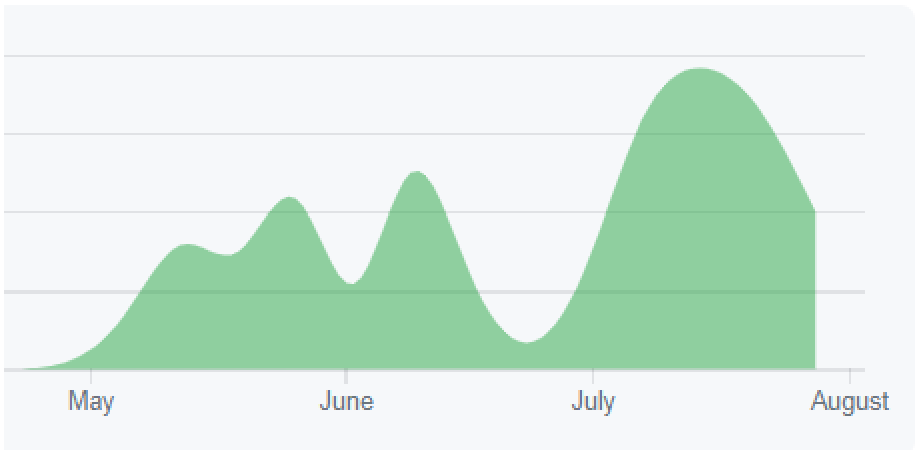


Figure 23: The Github commit history showing a large drop in productivity in late June

Although implementing the game prototype was completed almost two weeks later than intended this was not disastrous for the project. The reasons for this are that when the project was initially developed, the time required for playtesting the game and gathering results was consciously overestimated considerably. There were two reasons for doing this:

1. It was expected there would be delays in the implementation of the game. Therefore, it made sense to plan for the testing being delayed.
2. If there were no delays in the implementation the playtesting stage is one of the few stages that would benefit from the additional time. A longer playtesting stage would allow more results to be gathered, which would have enhanced the findings of this study.

It is important to build contingency into any software development project because unexpected delays often arise. Not accounting for some delay could result in the project failing or not completing all its objectives. Overestimating the testing time results in a win-win situation, as it either makes up for time lost to other delays or allows the gathering of extra results to improve the study. In this case the time was required for making up the delay in the implementation stage, and the actual time spent testing the game was reduced to approximately two weeks instead of four.

3.2 Software Design Methods

3.2.1 Overview

This section discusses some of the methods used to design the technical aspects of the game prototype. Like any piece of software, it is important to create a well thought out class structure to make the code readable and easy to maintain. Designing the software in advance helped minimise dependencies between classes to create a loosely coupled system. Low coupling makes it easy to update the software in the future or develop it further into a larger game.

Designing the software first not only resulted in a better structure but also made implementation faster. Not having to consider the overall structure during implementation made programming the game faster and more efficient by reducing the need for refactoring during development. Because the structure was well thought out the software was less prone to bugs or unexpected behaviour.

If the scope of the game prototype had changed significantly during development then the software design work may go to waste as the game's requirements change. During this project the implementation was carried out after the design was complete, meaning the design did not change. Therefore, completing the software design in advance was worth the time required. For more complex projects with changing requirements these techniques may be less suited. The advantages and disadvantages of each design method are discussed in their relevant chapter.

3.2.2 Class diagram

A UML class diagram was created during the planning stage to help understand the class hierarchy and dependencies within the program. The class diagram was created after the game mechanics had been designed, so that it accurately reflected the intended functionality. A labelled version of the initial class diagram is shown in Figure 24 and the unlabelled diagram can be found in appendix F. Table 6 provides explanations for the labels shown in Figure 24. During the implementation of the game the initial class diagram was used as the basis for the class structure but was modified where required. Changes to the class diagram occurred because of improvements, changes to the prototype scope and unforeseen issues.

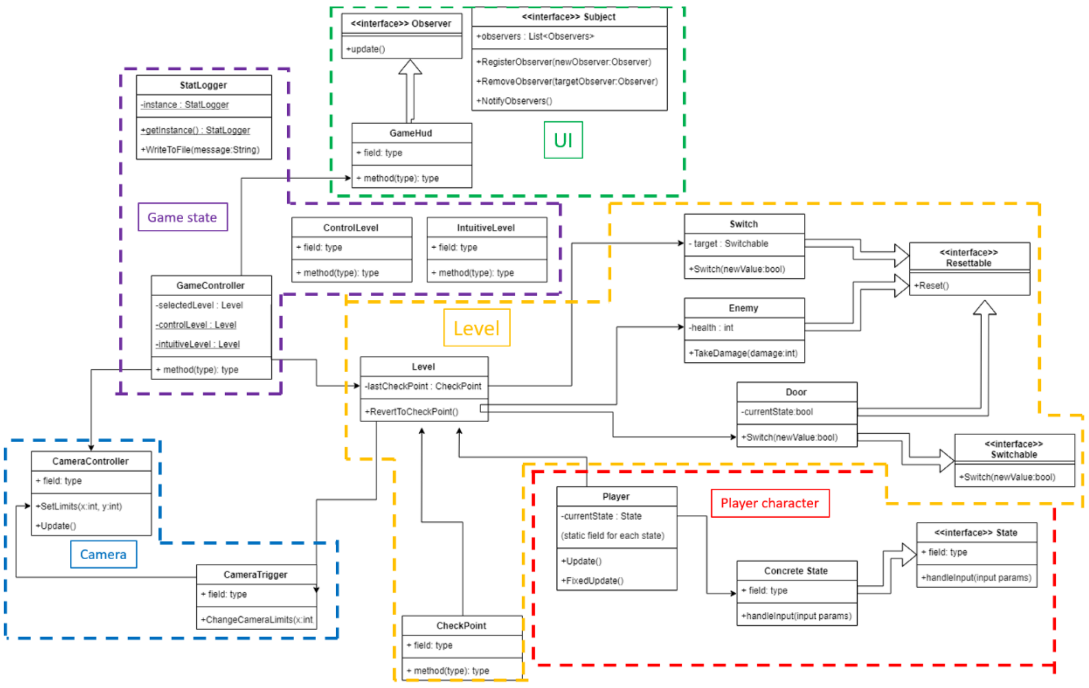


Figure 24: The initial class diagram created during the design stage of the project with labels to show key functionality

Table 6: High level explanations for the labels shown on the class diagram

Game State	Manages the overall state of the game: which level is active, the players statistics, writing to the log file, creating the player and starting the game, closing the game
Camera	CameraController controls camera movement, CameraTriggers update the camera's movement limits
Level	Level class handles resetting game when player fails and storing latest checkpoint. Objects in the scene which have functionality need their own script: enemies, doors, switches
Player Character	Defines the character functionality, state classes to create the player state machine, handles player input to make character respond appropriately.
UI	Updates any UI components that are used to display information to the player while the game is running.

By planning the class dependencies before implementing any code it made it easy to make changes to the class structure and optimise it. Showing the dependencies on a diagram helped to highlight and eliminate bad practices such as tight coupling. In the initial class diagram only key fields and methods which defined the classes main functionality were included. The time to plan out every field and method would not have yielded better results and the time taken would be better used somewhere else. It was much easier to work out the small details while implementing the program and to only use the class diagram for planning the high-level structure.

The class diagram did not drastically change between the initial plan and the final implementation, which shows it was well designed. Most of the changes were simply expanding on the original high-level design. For example, the Enemy class was replaced with two concrete classes and a superclass. More detail was added to the final diagram including; important fields for each class, important methods for each class and every player state class. Methods such as simple getters or setters have been omitted. It appears there are many public fields in the class diagram but many of these used auto-implemented properties. Auto-implemented properties create a private field along with a getter and/or setter method (Microsoft, 2015). Different access modifiers can be applied to these get and set methods to control access to the field. The code snippet below shows some example auto-implemented properties used in the Player class.

```
public class Player : MonoBehaviour
{
   ...
    //Components
    public Rigidbody2D AttachedRigidBody { get; private set; }
    public SpriteRenderer PlayerSprite { get; private set; }
    public Animator PlayerAnimator { get; private set; }
    public AudioSource PlayerAudio { get; private set; }
   ...
}
```

In some cases, functionality was no longer required such as the heads-up display (HUD), so the classes required for this were cut from the diagram. The labelled final class diagram is shown in Figure 25 and the unlabelled diagram is shown in appendix F.

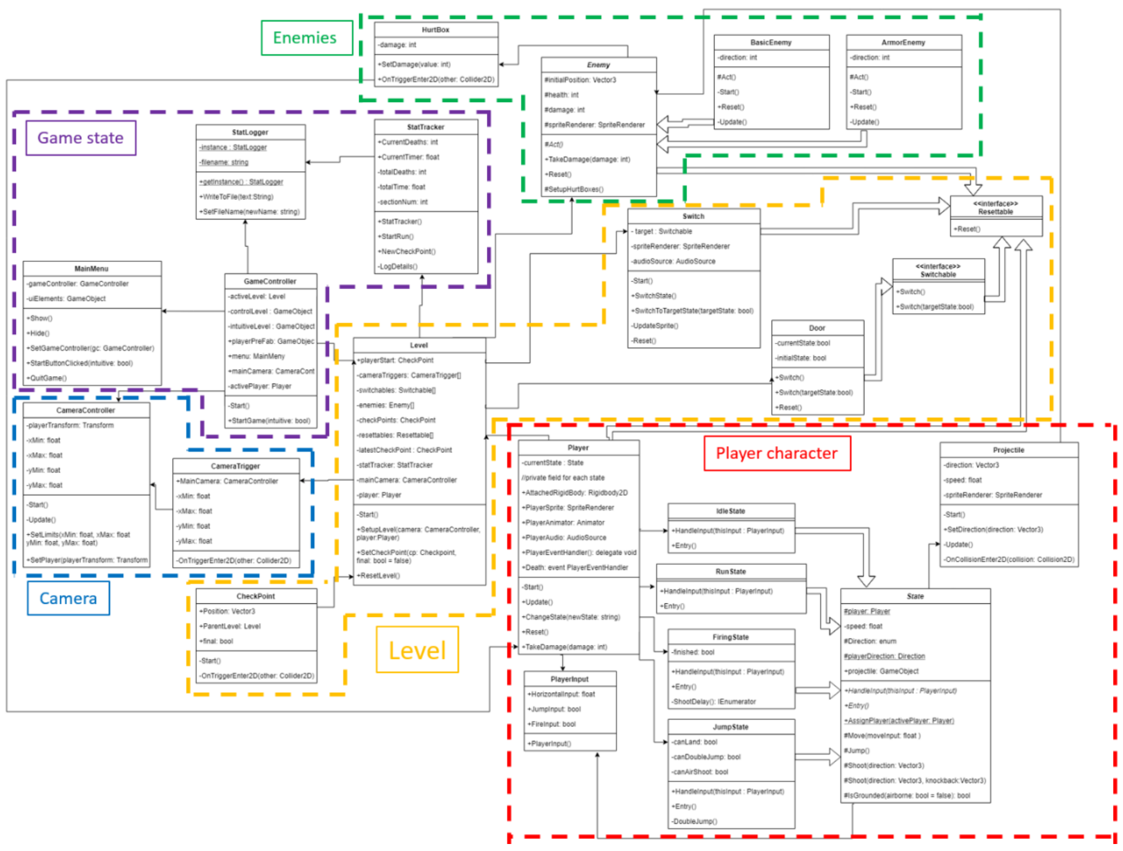


Figure 25: The final class diagram for the game prototype

The planned class structure made implementing the game faster for two reasons:

1. Much less time had to be spent considering the design during the implementation. It was easier to focus on the small details while programming the game because the overall hierarchy was mostly established.
2. It was much faster to refactor the design on a diagram as no code had been written yet. Any changes to the structure just required manipulating the class and dependencies on the diagram. Refactoring during the implementation would involve deleting or rewriting code which would mean much more time went to waste.

The disadvantage of creating a class diagram is the time required to create it. A considerable amount of time is required to plan the class hierarchy and display it on a diagram. If the game scope were to change considerably then the class hierarchy would also have to change, meaning a lot of the work done on the class diagram would be wasted. Therefore, creating diagrams such as this for a project where the scope is likely to change is a bad idea. The design and implementation were sequential in this project, so the game changed very little in scope over its lifecycle. This is shown by the fact the final class diagram does not greatly differ in structure from the original. The class diagram in this project was a powerful tool that helped created a clean class hierarchy with well thought out dependencies.

3.2.3 Software design patterns

3.2.3.1 The State Pattern

An important design pattern used for the game prototype was the state pattern (Nystrom, 2014). The state pattern was used to create the player character’s behaviour. This pattern helped to simplify how the game handled the complex combination of variables that can affect which actions the player can take. To give some examples

- The player can both jump from the ground and double jump in the air. The jump in the air should shoot a projectile downwards but the ground jump should not.
- The player can only shoot forwards one time in the air, and double jump one time in the air. This should reset on landing.
- The player can't take any actions while firing on the ground but there is no delay when firing in the air.
- The player should be knocked back when firing in the air but not on the ground.

Many difficulties arise even with a simple player character that can only move, shoot and jump. If further mechanics were added these complexities would increase exponentially.

The state pattern focuses on modelling the player's behaviours as a finite state machine (FSM), each state of which will become a class in the game code. The FSM created for the player character is shown in Figure 26.

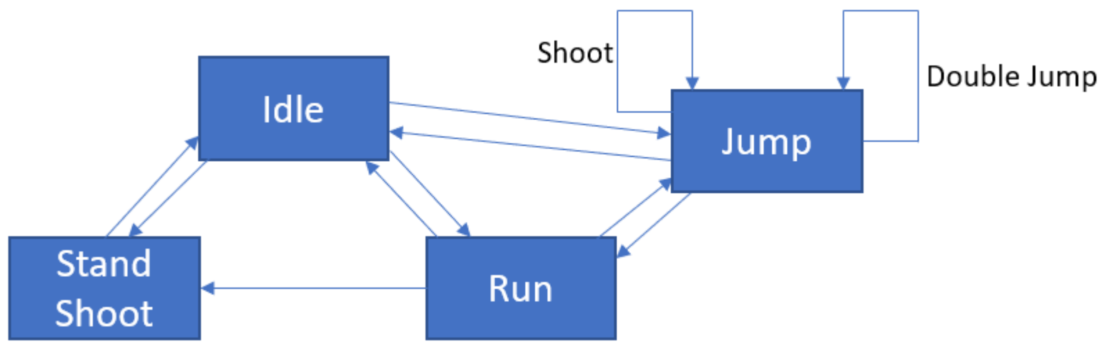
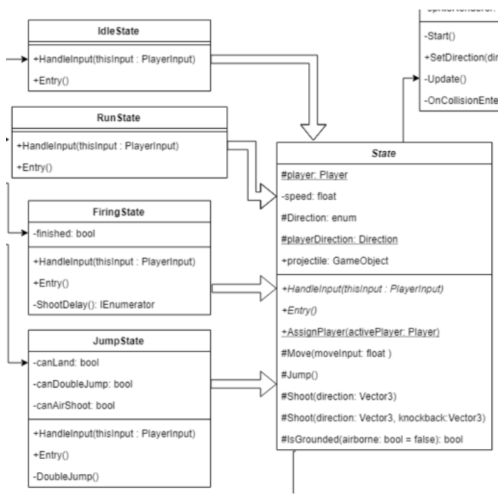


Figure 26: The Finite State Machine that controls the player character's behaviour

Initially the shoot and double jump actions that can be performed in the jump state were intended to be their own states. However, these are instantaneous actions and so are better suited to being methods that can be used in the jump state. The standing shoot is different as it has an associated delay. By having a state to represent the standing shoot this can: prevent the player from acting during the shoot delay, ensure the shoot occurs after the correct time and transfer back to the idle state when finished.

An abstract State class defines the abstract methods that any child must implement. These were HandleInput and Entry. The HandleInput method can be called generically on whichever state is currently active that frame so that it can react to that frame's inputs as required. The Entry method is called when the state becomes the current active state and is a tidy place for operations like triggering animations. An Exit method could also have been created if required. The complete State superclass is shown in appendix G.

The State superclass also defines any behaviours each state may use. This is the player character's actions such as move, jump and shoot. Each child class can access the operations it needs, and all the action functionality is stored in one place where it can be easily edited. The concrete state subclasses then implemented whatever functionality that state required. Figure 27 shows a section of the class diagram described in 3.2.2 containing the State superclass and all the subclasses in the final version of the game prototype.



IdleState
+HandleInput(thisInput : PlayerInput)
+Entry()
RunState
+HandleInput(thisInput : PlayerInput)
+Entry()
FiringState
-finished: bool
+HandleInput(thisInput : PlayerInput)
+Entry()
-ShootDelay(): IEnumerator
JumpState
-canLand: bool
-canDoubleJump: bool
-canAirShoot: bool
+HandleInput(thisInput : PlayerInput)
+Entry()
-DoubleJump()
State
#player: Player
-speed: float
#Direction: enum
#playerDirection: Direction
+projectile: GameObject
+HandleInput(thisInput : PlayerInput)
+Entry()
+AssignPlayer(activePlayer: Player)
#Move(moveInput: float)
#Jump()
#Shoot(direction: Vector3)
#Shoot(direction: Vector3, knockback:Vector3)
#IsGrounded(airborne: bool = false): bool
-Start()
+SetDirection(dir
-Update()
-OnCollisionEnte

Figure 27: The State superclass and its concrete subclasses

The State pattern makes the player character's behaviour easily scalable. New states can be added to the FSM with ease and without affecting the existing behaviours. States can also be removed with ease for the same reason. Adding and removing states is also easy because the State pattern decouples player actions from the top-level Player class. The Player class simply collects the inputs each frame and passes them to the current state to handle. It does not require any knowledge of what actions can be taken and how the input should be handled. This means changes to state classes, actions or behaviours will not require any changes in the Player class. The code below shows how the player class handles player input each frame.

```
//Called once each frame
void Update() {
    //Capture inputs
    PlayerInput thisFrameInput = new PlayerInput();
    thisFrameInput.HorizontalInput = Input.GetAxisRaw("Horizontal");
    thisFrameInput.JumpInput = Input.GetButtonDown("Jump");
    thisFrameInput.FireInput = Input.GetButtonDown("Fire1");

    //Have current state handle inputs
    currentState.HandleInput(thisFrameInput);
}
```

The State pattern does require some additional programming overhead initially to create the class structure and properly define how to transition between states. However, this additional complexity in the hierarchy removes complexity from the code itself and is therefore less prone to bugs and unintended behaviour. Another issue with the State pattern can be the scalability in certain situations. The State pattern scales easily when adding additional states but adding other conditions may not scale as well. For example, if the ability to pick up weapons was added to the player character this would double the number of states because each state would need a version for carrying a weapon or not (Nystrom, 2014). One way around this would be to swap between two FSMs, one based on carrying a weapon and one without. However, the more separate FSMs there are the more programming overhead there is, which may also result in code duplication.

3.2.3.2 The Singleton Pattern

The Singleton pattern involves applying two rules to a class (Nystrom, 2014). These are:

- Only one instance of that class should exist in the program scope.
- That instance should be globally accessible by other classes.

This pattern is useful in situations where multiple instances of the same class may cause conflict issues, most commonly when the class' purpose is to access a single resource.

In this prototype the StatLogger class implemented the Singleton pattern. This class' job was to output the statistics and playtesting data to a text file while the game was running. A singleton was selected for this class for 2 reasons:

1. Only one class may write to a single file at one time. Simultaneous attempts to write to the same file would lead to exceptions and objects having to wait for others to finish.
2. Many classes throughout the program may need access to the StatLogger class, to output information relevant to the playtesting. What information needs to be output may easily change in the future so changing what data is logged should be quick and simple.

The StatLogger class implementing the Singleton pattern means that any object that calls for data to be written to the output file goes to the same StatLogger instance. This makes sure there are no conflicts with file opening/writing and ensures all data is written to the file in the correct order. To ensure there is only one instance of the class the constructor is made private. Instead of using the constructor, a static GetInstance method provides access to the StatLogger object for other classes. Lazy initialisation was used, meaning the StatLogger object isn't loaded until the first time an object calls GetInstance. The code snippet below shows how this was implemented in the StatLogger class:

```
public class StatLogger
{
    private static volatile StatLogger instance;
    ...

    private StatLogger(){    }

    public static StatLogger GetInstance()
    {
        if (instance == null)
        {
            lock (mutex)
            {
                if (instance == null)
                {
                    instance = new StatLogger();
                }
            }
        }
        return instance;
    }
```

The lazy initialisation of the StatLogger class presents an issue because the program is multithreaded. If multiple objects call the GetInstance method in short succession before the instance has been instantiated, two instances may be instantiated. To prevent this a lock has been used. The lock ensures the objects attempting to call the GetInstance method are forced to execute the locked section one after another in the order they first called it. This lock ensures that if an object has already instantiated the class it cannot be instantiated again. In order for this lock to work in C# the static instance of the StatLogger class must be volatile which prevents the compiler from caching the value of the field (Semmle, 2019). The advantage of lazy initialisation is that the instance is never loaded if it is not required. It saves time when the game first loads as many other objects will be instantiated at first.

The singleton pattern should be used carefully as global access to a class can have negative impacts on a projects structure. A global access method like the GetInstance method makes it difficult to keep track of dependencies between the singleton class and other classes. Because the singleton becomes coupled to every class that retrieves the instance of it, each of these classes will need to be updated if the singleton class changes. Because the dependencies are difficult to track this may lead to bugs or unexpected behaviour when updating the singleton.

3.2.4 High Level Programming Features

3.2.4.1 Resettable Interface

A very important type in the game's class hierarchy was the resettable interface. This interface was used by objects which need to be reset when the player fails. As the player progresses through the level they reach checkpoints. These checkpoints mark the start of each section and are where the character resets to if they fail. When the player fails and they return to the start of the section, the section should not stay in its current state. Instead the section should return to the initial state it was in when the game loaded. Some examples of this are defeated enemies returning to life and switches returning to their initial state.

Any object that had the possibility of being in a different state when the player failed, used the resettable interface. This meant the class had to implement a Reset method that returned it to its initial state. The concrete implementation of this Reset method would differ for each class because different actions are required to return it to its original state. For example, to reset a switch it must be set to the "off" state whereas to reset an enemy it must be moved back to its initial position and returned to life if it had been defeated.

The Level class stored a reference to every resettable object that existed within it. Every time the player failed this collection could be iterated over to call the Reset method of every object, ensuring the level returns to its initial state. In the future this functionality could be improved to only Reset the objects in the current section, but this was not required as the game had no performance issues.

3.2.4.2 Switchable Interface

One of the important mechanics in the game was switches which were activated when the player shot them. The Switchable interface was created to define objects which were capable of being connected to a switch. To connect to a switch, an object had to have two states it could swap between. To enforce this functionality the Switchable interface meant any class using it required two Switch methods. One method to swap the object to the opposite state and one method to set the object to a specified state.

In the final version of the game prototype the only Switchable object was the door object. The doors switched between being open and closed. A moving platform was also developed which would only move when its paired switch was set to on, but this was not used in the final levels.

Despite its limited use within the game, the Switchable interface is essential for developing the game further. The Switchable interface creates a standard practice for pairing objects to switches and allows new switchable objects to be added easily. This interface makes adding new objects faster and ensures the new code will be standardised and therefore easy to understand.

3.2.4.3 Enemy Superclass

During the design stages of the game prototype it was specified that there would be multiple types of enemy. These enemies had individual behaviours or functionality but had the same high-level purpose, to defeat the player. It therefore made sense to create an abstract superclass that each individual enemy type could inherit from. This allowed different types of enemies to be grouped together and treated the same way, despite their small differences in functionality. The Enemy superclass was made abstract because it should never be instantiated in the game. Instead subclasses for each type of enemy should be created and inherit from the superclass.

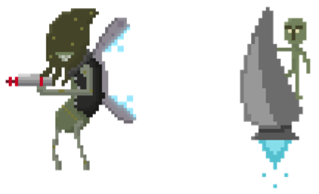


Figure 28: The two types of enemy present in the final version of the game prototype

The Enemy class defined an abstract Act method that had to be implemented by all its subclasses. The Act method for each subclass contained the logic which the enemies activated each frame. This allowed all the enemies to be stored in a collection and then iterated over to make them act. Enemies had to be resettable, so the Enemy class implemented the resettable interface and provided a concrete Reset method with basic reset functionality. This method was virtual because some specific enemy types required more complex functionality to correctly reset them. The ArmorEnemy is an example of a concrete enemy class that needed to override the Reset method and the code snippet below shows how this was implemented.

```
public override void Reset()
{
    base.Reset();
    if(direction == 1)
    {
        body.Rotate(new Vector2(0.0f, 180.0f));
    }
    direction = -1;
}
```

In this case the ArmorEnemy just needed to add extra functionality to the superclass' Reset method so it first called "base.Reset()" and then took an additional step to ensure the enemy was facing the correct direction when reset.

The Enemy superclass makes it simple to add new types of enemies and allows all types of enemies to be grouped together. It provides some standard functionality such as the basic Reset method so that every enemy class does not need to reimplement the same methods. This reduces the total lines of code and makes the code base more readable. It also ensures that common functionality is being executed in a standard way across all enemy types.

3.3 Tools and Software Required for this Project

3.3.1 Game engine - Unity

3.3.1.1 Overview

As outlined in the literature review, a game engine can save a large amount of time and effort during the development process of a game. Unity was used to create the game prototype for this study and many benefits were seen from this throughout the development process. There are also some disadvantages to using game engines and although none of them affected this project they will be discussed briefly.

The main advantage of a game engine is the large amount of time and effort it saves in the initial stages of the development process. Unity provides many systems that the game prototype for this study required such as physics, animation, collisions and audio. Developing all these systems from the ground up would take a long time, whereas using a game engine means they can be used from the beginning. This meant once the game prototype had finished being designed work could begin immediately on implementing the game mechanics. Particularly for a project with a short timescale like this one, game engines can allow the creation of a much more complex and interesting game because no time must be spent implementing common systems like physics. These systems can also be very complex. If a physics system had been developed in this timescale rather than using Unity, then the result would probably have been far less advanced.

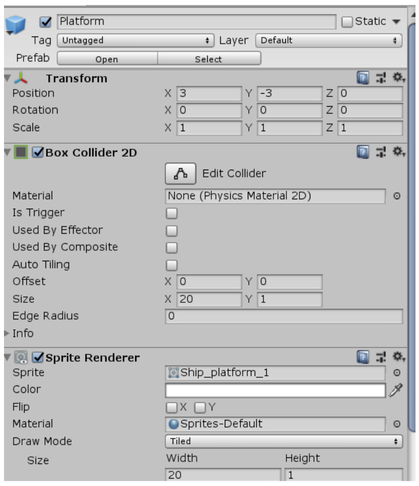
One difficulty when working with game engines is that in order to make the best use of the engine the developer must understand how it works. As discussed in section 2.6.3, Unity has excellent documentation available which makes it easy to understand how the engine functions. Unity provide both a manual to describe how to use the engine at a high level, and a scripting API which goes into full technical detail about the available packages, classes and methods. The Unity documentation was used extensively during the implementation of the prototype for this study.

Unity provided a well-structured platform to build the game logic upon, helping keep the codebase tidy and well-structured. Unity uses several well-known games programming patterns which help the developer to interface with the Unity logic cleanly. Unity uses the “update method” pattern to help the developer add functionality that should be activated every frame. Each frame the game will loop through every gameobject and call their Update method if they have one. The Update method is the perfect place to add any real-time functionality such as detecting player-inputs or activating enemy behaviours. The code snippet below shows how the Player class uses the Update method to capture the player’s inputs then pass them to the current player state so that the character acts accordingly.

```
//Called once each frame
void Update() {
    //Capture inputs
    PlayerInput thisFrameInput = new PlayerInput();
    thisFrameInput.HorizontalInput = Input.GetAxisRaw("Horizontal");
    thisFrameInput.JumpInput = Input.GetButtonDown("Jump");
    thisFrameInput.FireInput = Input.GetButtonDown("Fire1");

    //Have current state handle inputs
    currentState.HandleInput(thisFrameInput);
}
```

Another powerful pattern Unity uses is the “component” pattern. The component pattern decouples the many aspects of the engine’s functionality from each object within the game. This means that a basic gameobject within the game does not use any of the engine’s features such as physics, rendering or audio. This functionality can be added by adding components to the object. Therefore, all the gameobjects become simply collections of components. Components frequently have properties that can be edited individually for each object to specialise their functionality. For example, to create the player character first a SpriteRenderer component had to be added to the object so that the player could see their character. Next a Rigidbody2D component was added so that the character was affected by physics and a Collider component was added to define the characters shape when calculating collisions. Finally, to add the correct game logic a Script component was added which contained the Player class. This pattern is said to decouple each component because rather than having one large Player class which defines the physics, sprite rendering and game logic these features are separated into components (Nystrom, 2014). This pattern makes it easy to apply selected functionality from the engine to build the desired objects within the game. To contrast the complex player character object example, a simple wall object would only contain a sprite and a collider to define its shape.



```
☑ Platform                                   ☐ Static
Tag    Untagged                  Layer  Default
Prefab         Open          Select
Transform
Position            X 3        Y -3       Z 0
Rotation            X 0        Y 0        Z 0
Scale               X 1        Y 1        Z 1
☑ Box Collider 2D
                    Edit Collider
Material            None (Physics Material 2D)
Is Trigger          ☐
Used By Effector    ☐
Used By Composite   ☐
Auto Tiling         ☐
Offset              X 0        Y 0
Size                X 20       Y 1
Edge Radius         0
Info
☑ Sprite Renderer
Sprite              Ship_platform_1
Color
Flip                ☐ X  ☐ Y
Material            Sprites-Default
Draw Mode           Tiled
   Size             Width          Height
                    20             1
```

Figure 29: An example of the components of a platform gameobject in the game

A game engine was an excellent choice for this project and implementing the game prototype went without issue. However, using a game engine is not ideal for every game as they have some key limitations. The most important limitation of Unity is that the developer is constrained to the engine's behaviour. The Unity source code is not open, meaning that the developer cannot customise or change the functionality that Unity provides. In more complex or larger games some of Unity's systems may not be adequate but for the prototype created in this study Unity was perfectly suitable. Game engines also have an associated performance overhead when used. The engine functionality will require extra CPU power and little can be done to control this. Therefore, game engines may be unsuitable for games where efficiency and performance are key concerns. The simple functionality in this game prototype meant that even with the engine overheads the game ran very smoothly on the development laptop.

Overall, using Unity to develop the game prototype for this study saved a lot of time that would've been spent implementing systems like physics and sprite rendering. This time was used to further develop the game logic, creating a more ambitious and more polished final prototype. Unity provided all the functionality required and did not limit the prototype at any stages. Unity is excellent for small projects such as this but may have limiting factors when used for larger and more complex games.

This chapter will now explore in-depth some of the most important Unity systems that were used during this project.

3.3.1.2 Scene Hierarchy

The hierarchical scene structure in Unity helped to organise objects within the game, allowing for grouping, parent-child relations and hierarchy traversing. This hierarchy allowed many efficient operations such as deactivating a top-level object which will disable all its children too.

Figure 30 shows an example section of the level gameobject, which had a multi-level hierarchical structure.

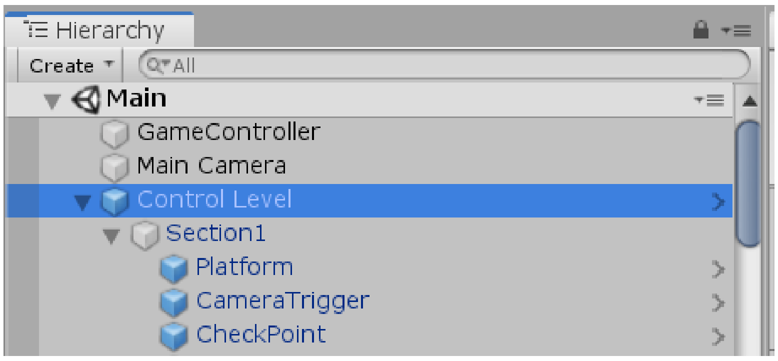
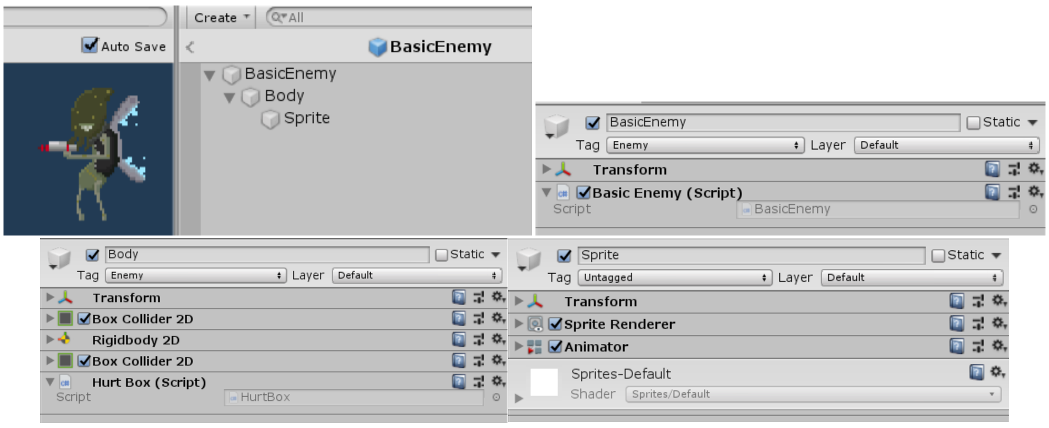


Figure 30: An example of the hierarchical structure of the level gameobject

A frequently used method from the Unity API was GetComponentInChildren<T> which extended the functionality of the GetComponent<T> method. The GetComponent<T> method searched for a component of the type T on the current object. GetComponentInChildren<T> searched the attached object, then every one of its children using a depth-first search (Unity, 2019). Creating an object hierarchy can decouple functionality within an object, making it easier to change in the future.

For example, the enemy units in the game prototype had the game logic script, collider and sprite renderer split over three separate levels. The script that defined the enemy functionality sat at the top level. This allowed the script to disable the enemy's physical presence in the game when defeated by the player and then reactive them if the level was reset. Disabling and then resetting the enemies is much more efficient than destroying them and re-creating them. The sprite renderer was a child of the body object which contained the collider definition, decoupling the art asset from the size and shape of the enemy. If later in the development process the art changed it would be easy to modify the enemy object to support these changes.



BasicEnemy
Tag Enemy Layer Default
Basic Enemy (Script)
Script BasicEnemy

Body
Tag Enemy Layer Default
Box Collider 2D
Rigidbody 2D
Box Collider 2D
Hurt Box (Script)
Script HurtBox

Sprite
Tag Untagged Layer Default
Sprite Renderer
Animator
Sprites-Default
Shader Sprites/Default

Figure 31: Images taken from the Unity editor to show the hierarchical structure of an enemy. ***Top left****: the hierarchical game object structure of a basic enemy.* ***Top right****: The top-level object of the enemy containing the game logic.* ***Bottom Left****: The body object containing physical representation of the enemy within the game.* ***Bottom Right:*** *The lowest level object containing the sprite to represent the enemy.*

Unity's hierarchical structure allows for efficient operations like disabling objects. It also provides organisation to the scene structure, making it easy to navigate through all the objects within the game. This was especially important when developing the levels as they were constructed from many gameobjects. Each section could be represented as a child of the level making it easy to work with.

3.3.1.3 Physics and Collisions

The Unity physics system was used to handle movement and collisions. Specifically, the Physics2D module (Unity, 2019) because the game only required two-dimensional movement. Unity uses a rigid body based physics system, meaning that objects do not deform and instead behave rigidly under force (Unity, 2019). An object with a rigid body component attached will be affected by physics, giving it properties like velocity and acceleration. Rigid bodies in Unity have both linear and angular movement components to support linear movement and rotation. Colliders are used to define an objects shape and size and Unity supports both basic shapes and more complex polygon colliders (Unity, 2019). To summarise, an object with a rigid body will be affected by physics and an object with a collider has a defined shape which allows collisions with other objects.

In this game, objects that moved and required collisions used both a Rigidbody2D and at least one collider. A good example is the player character who must; move in response to the player inputs, be affected by gravity and collide with other objects such as the enemies. Objects that were capable of colliding but did not move were simply given a collider. The ground and walls are examples of objects with colliders but no rigid bodies. This allowed the player to run along the ground and be blocked by walls.

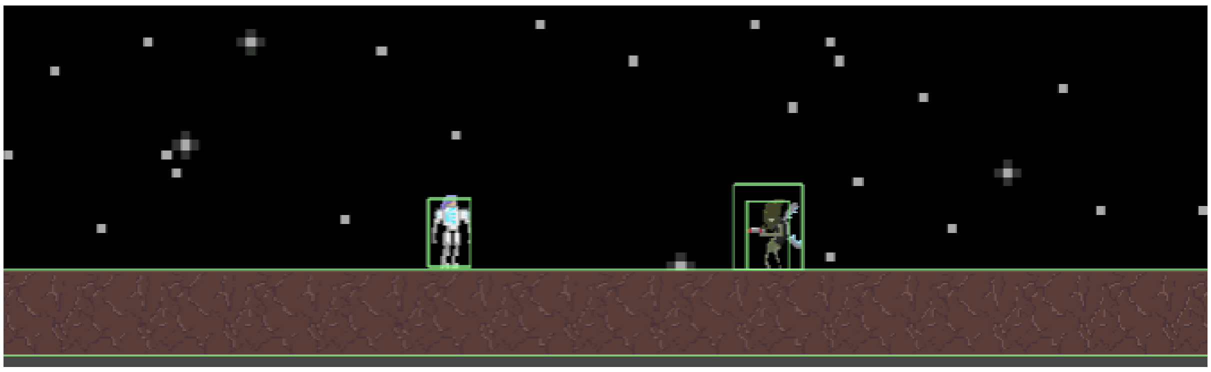


Figure 32: The player and enemy have both colliders and rigid bodies, the ground has just a collider

There are several ways to move a rigid body. The game prototype used two different methods for different purposes when creating the player movement. To cause the player to jump an upwards impulse force was added to the player's rigid body. Using an impulse ensured the player character jumped instantly when the player entered the jump input (Unity, 2019) making the game responsive. The force created an acceleration on the rigid body, affecting its velocity and as a result its position. However, when creating the horizontal movement of the player instead of using forces the position of the player was directly edited. Altering the position bypassed the physics system and meant the character's rigid body did not have any resultant velocity or acceleration. The benefit of this was extremely precise movement controls. The player character moved immediately when the player inputted a direction and stopped immediately if they released it. Precise controls allowed the player to feel in control and meant they were less likely to feel cheated when they failed. The code snippet below shows the Move method directly altering position and the Jump method adding a force to the rigidbody.

```
protected void Move(float moveInput)
{
    Vector3 moveVec = new Vector3(moveInput, 0.0f, 0.0f) * Time.deltaTime * speed;
    transform.position += moveVec;
   ...

}

protected void Jump()
{
    player.AttachedRigidBody.AddForce(Vector2.up * 10.0f, ForceMode2D.Impulse);
   ...
}
```

Rigid bodies are much simpler than deformable bodies and are therefore have lower performance requirements. The drawback of rigid bodies is that objects cannot change shape based on forces and collisions which is unrealistic. Therefore, rigid bodies are suited to games that do not require realistic physics behaviour, but instead only need basic physics and collisions. This suited the game prototype for this project perfectly, as deformable bodies would not have added anything other than slight visual flair to the game. The simpler nature of rigid bodies kept the performance requirements down and ensured the game ran smoothly with a consistent high frame rate.

3.3.1.4 Sprite Rendering and Animation

As the game was 2D it was created entirely from sprites rather than 3D models. Unity provides a sprite renderer component that can be used to display sprites in both 2D and 3D games (Unity, 2019). Sprites offer better performance than 3D graphics but are of course constrained to two dimensions, making them only suitable for certain applications.

The sprite renderer components can be accessed via C# scripts in order to manipulate it while the game is running. For example, the “flip x” property of the sprite renderer on the player was toggled on and off to turn the player sprite to the direction the player was running. The code snippet below shows how the sprite renderer is assigned when the object is first loaded and then how the sprite is flipped to match the player’s direction when they move.

```
void Start()
{
   ...
    PlayerSprite = GetComponentInChildren<SpriteRenderer>();
}

protected void Move(float moveInput)
{
    ...

    if (moveInput > 0.0f)
    {
        PlayerSprite.flipX = false;
    }
    else if (moveInput < 0.0f)
    {
        PlayerSprite.flipX = true;
    }
}
```

A useful feature built into the Unity sprite renderer is sorting layer. Sorting layer allows different layers for containing sprites to be defined. After creating a layer any number of sprites can be assigned to it, and the order of the layers determines which sprites will be rendered over the top of others. This was used in the prototype to make sure the characters were rendered over the top of the background images.

Animations can be created within Unity as sequences of sprites. For this project the artist created spritesheets containing all the images in the animation. Within Unity these sprites were then assembled together into an animation which could be played by an animator component (Unity, 2019). The animator component is linked to a sprite renderer and updates which sprite is being displayed to match the ones defined in the animation.

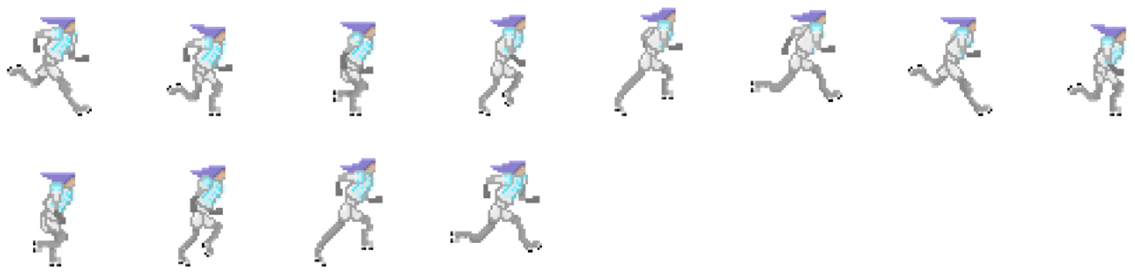


Figure 33: An example animation spritesheet which was used to create the player character running animation

Animator controllers can be used in Unity to store multiple animations an object can use. The controller defines which animations should be used, when they should be played and how transitions between them should occur (Unity, 2019). A controller is defined as an animation state machine which can contain parameters that are used to trigger transitions between states. This mapped well to the player character's behaviour which was also mapped to a similar state machine as described in section 3.2.3.1.

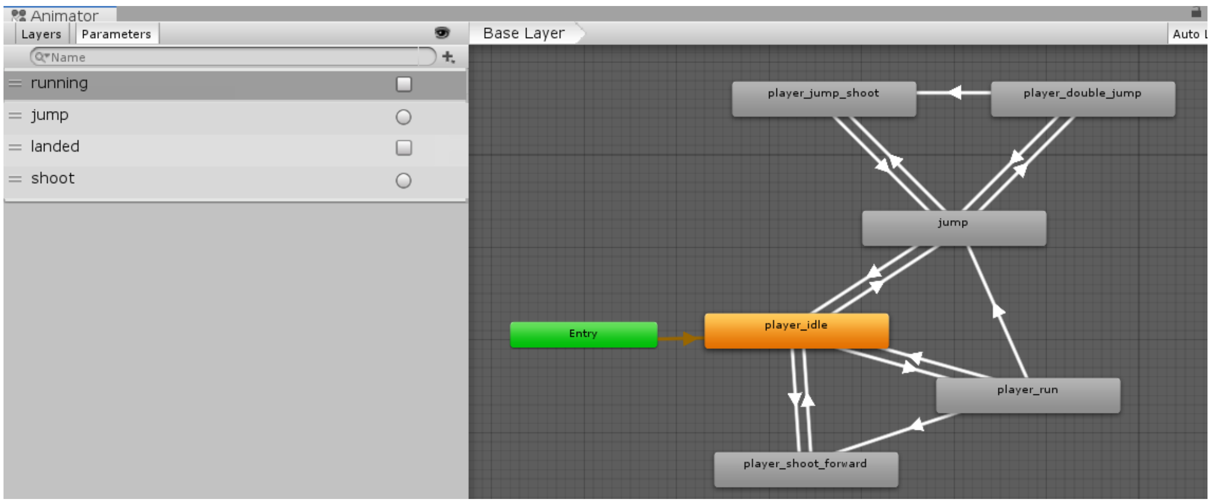


Figure 34: The player character animation state machine

3.3.1.5 Prefab Objects

Unity allows gameobjects that have been created in the editor to be saved as "prefab objects". Prefab objects are an efficient way to create multiple independent copies of the same object within a single scene. When creating a prefab, a gameobject is saved as a file within the project and can then be referenced by scripts or added to other objects within the editor. These two different methods of using prefabs were both used within the game prototype for different purposes.

Accessing a prefab using a script was used for dynamically created objects like the players projectile. During the game the projectiles need to be instantiated wherever the player's character was which cannot be known in advance. To achieve this, the script finds the projectile prefab and then creates a copy of it based on the players current position. This functionality is shown in the code snippet below.

```
protected void Shoot(Vector3 direction)
    {
        //Instantiate the projectile
        GameObject tempProjectile =
              (GameObject)Instantiate(Resources.Load("Projectile"),
              transform.position + direction * 0.3f,
             Quaternion.identity);
        //Set the direction of the projectile
        tempProjectile.GetComponent<Projectile>().SetDirection(direction);
        player.PlayerAudio.PlayOneShot(player.shootSound);
    }
```

Instantiating objects at run time can be liable to performance issues and so should be used with care. It was suitable in this case because the projectile object was very small and simple, and the player could not fire very quickly. To reduce the performance requirements object pooling could be used, which involves creating all the required objects when the game is loaded and then simply taking one from the pool instead of creating one when needed. This was not required as the game ran smoothly throughout the entire level.

Prefab objects were added as children to another object when creating the levels for the game prototype. The level contained many instances of duplicate objects such as platforms, enemies and switches. Despite being duplicates of the same object each instance had unique traits such as position or references to other objects. Prefabs allowed the level to be created without having to define every platform or enemy as a new game object. Instead a copy of the prefab was added to the level and then modified to create each specific object. This saved huge amounts of time when creating the level. Prefabs also decouple the individual objects from the level which allows a change to the prefab to be reflected in every copy of it. So, if the sprite for the enemy changes the developer would simply have to update the prefab and every enemy in the level would be updated. Figure 35 shows an example section of the control level and the many instances of the platform objects used to build it.

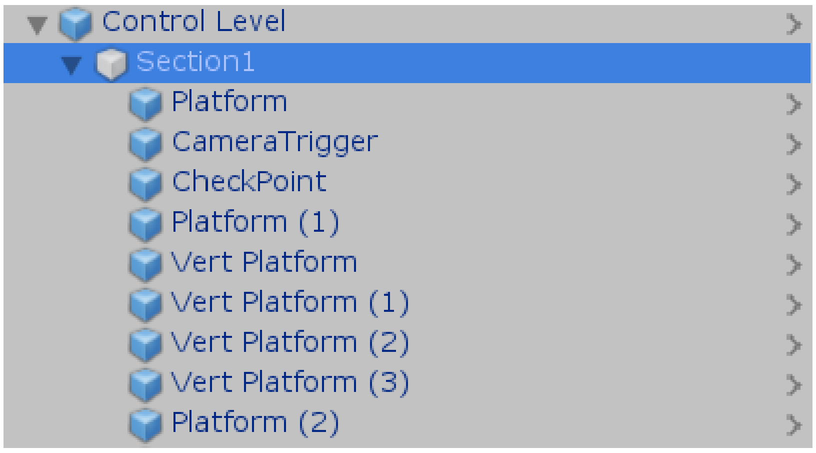


Figure 35: An example section of the control level showing the multiple instances of prefab objects used to build it

Prefab objects have a higher storage requirement than defining the equivalent object in a script, but they offer many benefits over this approach. Prefab objects can be opened and edited in the Unity editor, giving the developer a real time visual representation of the object. This gives the developer immediate feedback on any changes they are making. Prefab objects are very fast to create, the developer simply selected a game object they had created in the game and chose to save it as a prefab. Defining the same object in a script would've been time consuming and changes would also take a lot longer. Because storage space was not an issue for the small prototype created in this project, prefab objects were a fast, simple and powerful tool for dynamically creating objects and building the level from a small collection of components.

3.3.2 Version Control

Version control software was used throughout the project to capture the changes made at each stage of development. The version control software allowed the developer to revert single files, or the entire project to a previous stage if required. This is a powerful tool that can both save time and protect the developer from losing data. For example, if a file becomes corrupted then that file can be reverted by one stage which means minimal data is lost. Version control is far superior to a simple backup of the data because it tracks changes and allows individual reversions of files as well as reverting the entire project.

Version control software's strength is not just backing up data, it is also excellent for facilitating collaboration. Version control helps collaborators; to prevent overwriting each other's work, manage any conflicts created by changes to the same file, track other people's work and work simultaneously on a file without interfering with each other. Although this was an individual project most real-world software will be created by teams of developers. Using version control in this project from the beginning makes it very easy for new developers to become involved in the project and begin contributing simultaneously. The version control system would help them get up to speed with the previous work on the project and allow them to easily begin contributing.

The specific version control used for this project was git. Git is a distributed version control system, meaning anyone working on the project takes a clone of the main repository, makes changes to their local version then "pushes" the changes to the shared repository (git, 2019). Github was used as a repository hosting platform to store the master copy of the project remotely. Hosting the project on Github meant even if the local development hardware completely failed the project was backed up and could be retrieved on another computer.

Making use of version control software does require some additional effort from the developer(s). Changes should be committed to the repository often and using descriptive commit messages will make it easy to view the change history of the project. Committing when individual features are completed creates logical restore points in the project's history, making it easy to revert specific features. The extra effort to use version control is far outweighed by the benefits; it creates a backup of the project, allows reversion to previous states for one or many files, stores a history of the changes that have been made to a project and supports collaboration between multiple developers on the same files.

3.4 Summary

The implementation of this study was split into three stages: designing the game prototype, implementing the game prototype and gathering results. A game design document was used to track the design of the game mechanics and the technical software design. Several design techniques such as programming patterns were used to improve the software design of the prototype. The game prototype was created using the game engine Unity, which involved programming the game logic in C#. This chapter went into detail about the key aspects of Unity's functionality that were used in the game prototype. Other tools required for this project such as version control were also discussed.

A testing procedure was developed to gather data using the game prototype and has been explained in depth. This procedure was used to gather the data required for this study which will be discussed in full in section 4.

The initial project plan was compared to the real completion dates of each task. There were some deviations from the plan and the cause of these has been explained, along with the measures taken to get the project back on track. The game prototype was finished on time and all planned tasks were completed fully. Professional practices were used in the design and development of the game to ensure a high-quality piece of software was produced. This means the game could easily be developed further in the future.

4 Results and Evaluation

4.1 Completeness of Game Prototype

During the research stage of this study a requirement analysis was created to define the scope for the game prototype. The MoSCoW method was used to assign priorities to each requirement which determined their importance when implementing the prototype. Requirements were not only created for the game prototypes functionality, they also extended to the design process, implementation and results gathering stages of this study. Tables 7 to 11 evaluate the final game prototype in comparison to the defined requirements.

Table 7: A comparison of the game prototype to the sub-requirements of "Have a playable prototype to play"

Requirement	**Prioritisation**	**Explanation**	**Requirement Met?**	**Details**
Have a playable prototype to play	Must	There must be a playable game prototype that is capable of supporting the study of intuitive level design	Completely	
Prototype uses game engine to save time	Must	Using a game engine rather than developing from scratch will greatly reduce development time	Completely	
Prototype uses Unity as the game engine	Could	Unity is a game engine well suited to small scale projects produced by small teams or solo developers	Completely	
Basic sound effects to accompany mechanics	Should	Required to give player feedback on gameplay and aid their understanding	Completely	Sound effects: Jump, shoot on ground, shoot in air, enemy
Basic animations to accompany player mechanics	Should	Required to give player feedback on gameplay and aid their understanding	Completely	Animations: idle, run, jump, shoot on ground, shoot in air, double jump
Have a complete set of sound effects	Could	Enhance player engagement and immersion	No	No music, no variation in sound effects, no ambient sounds, no footsteps
Have a complete set of animations	Could	Enhance player engagement and immersion	Partially	Player animations are complete, enemies are lacking, some ambient animations
Be available on major distribution platforms	Won't	Players will need to be interviewed while playing. Must use same machine for all tests	No	
Be multiplayer	Won't	Only testing on one individual at a time	No	
Be monetised	Won't	Not necessary	No	

Table 8: A comparison of the game prototype to the sub-requirements of "Have at least 2 level designs to compare"

Requirement	Prioritisation	Explanation	Requirement	Details
Have at least 2 level designs to compare	Must	Require comparison to show whether altering level design has an affect	Completely	
Have a control level that doesn't use intuitive design	Must	This level will be used as baseline	Completely	
Have an intuitively designed level	Must	This level will attempt to use intuitive design techniques to improve player learning and performance	Completely	
2 Levels to be compared have similar length and difficulty	Should	Will ensure results are fairer and data is valid	Completely	
Have multiple consecutive levels	Won't	Only one level for each test is required	No	

Table 9: A comparison of the game prototype to the sub-requirements of "Have a suitable number of playtesters to collect data"

Requirement	Prioritisation	Explanation	Requirement	Details
Have a suitable number of playtesters to collect data	Must	More playtesters means the data is more likely to be statistically valid	Completely	
Playtesters should be from target audience	Should	This will ensure they have similar experience and interest in the game	Completely	
Same number of playtesters for each level	Must	Same amount of data for each level means fair comparison	Completely	
Candidates are interviewed on their experience	Must	Must establish candidate experience level is within a given range so that all candidates are similar in experience	Completely	Survey used to gather data

Table 10: A comparison of the game prototype to the sub-requirements of "Game is thoughtfully designed"

Requirement	Prioritisation	Explanation	Requirement	Details
Game is thoughtfully designed	Should	Thoughtful design of the game will enhance experience and be more representative of a full game	Completely	Time was taken to create design doc and refine design
Game has a basic premise which is implemented in game	Should	Premise can make game more engaging and help players get immersed.	Completely	Character, setting and conflict established through graphics
Be designed around a core mechanic	Could	Designing around a core mechanic could help make the main mechanics easy to understand	Completely	Shooting
Use design patterns to structure software	Should	Design patterns lead to better structures and more efficient code	Completely	Design patterns implement in logic: state pattern, singleton patter. Also design patterns implemented by game engine
Integrate feedback early on to improve game design	Could	Can help address issues designer may be missing	No	No time, this could've benefitted project

Table 11: A comparison of the game prototype to the sub-requirements of "Game collects data for analysis"

Requirement	Prioritisation	Explanation	Requirement	Details
Game collects data for analysis	Must	Data is required to compare results after testing is done	Completely	Time and deaths for sections and overall were logged to a text file
Game stores data about playtest for analysis later	Should	Would be more efficient and accurate than inspecting the test and manually recording data	Completely	Text file auto generated in same format each run. Stores which level, times and number of deaths for each section and times and number of deaths overall
Have fully automated exporting and analysis of test data from game prototype	Won't	Not necessary for a test of this scale. Development time would be greater than time saved	No	Data manually analysed using excel

The game prototype and the implementation process met all the key requirements. Every requirement with "must" or "should" priority was met, along with several of the optional "could" requirements. The requirements that were not completed were due to time constraints. These requirements were assigned "could" priority because it was suspected there would be insufficient time to complete them. One of these requirements was to integrate feedback into the development of the game by having user testing early on. This could be done iteratively to continuously build upon player feedback and improve the game (Fullerton, 2008). The benefits of this are discussed in more detail in section 5.2.3 but the time constraints of this study did not allow for iterative development and testing.

The requirements were created to ensure the scope of the game prototype was sufficiently complex to support the investigation into intuitive level design. Requirements for the implementation process were defined to ensure the study made efficient use of the time and was completed to a professional standard. The final comparison shows the requirements were completed to a high standard and the study was executed well. By meeting all its key requirements, the final game prototype was sufficiently complex to support the investigation, well designed and robustly implemented.

4.2 Analysis of Test Data

4.2.1 Overview

The goals defined during the research stage of this study were:

- To establish if intuitive design allows players to understand the game faster.
- To establish if intuitive design increases player enjoyment.
- To establish if intuitive design increases how intuitive the player finds the game.
- To establish if intuitive design decreases the player's perceived difficulty of the game.

The game prototype was developed to investigate these goals and two types of data were collected to analyse the participants' performance. Twelve people were recruited to participate in the test, meaning six unique participants played each level. Statistical data about the player's completion time and number of failures was recorded for each section of the level the player completed. This statistical data was used to evaluate how well the player performed in the game. On completing the game each participant then filled in a short survey. The survey asked the player some questions about how much they enjoyed the game and how difficult they found it. The full survey can be viewed in section 0. The survey aimed to evaluate each player's experience when testing the game to determine whether the intuitive design had any effect on the players perception of the game.

The data gathered was compared between the two levels using t-tests to determine if there was a statistically significant difference between the two sets of data. The t-test compares the variation within the data for each level to the variation between the two levels to determine whether any differences are statistically reliable or could be caused by randomness. The result of the t-test is the p-value, which must be less than the α-value for the difference to be statistically reliable.

If the resultant p-value was less than an α-value of 0.05 then the two groups had a statistically reliable difference. Care had to be taken with the section specific data being compared because multiple t-tests are being carried out. When running several t-tests the probability that a statistically significant result is returned due to randomness instead of actual statistical significance increases. To account for this the α-value was reduced using the Bonferroni correction (Napierala, 2012). The Bonferroni correction requires dividing the α-value for each individual test by the number of tests. As the game was split into 14 sections and each was individually analysed, the α-value was divided by 14. This gives an α-value of 0.00357 to 3 significant figures.

4.2.2 Time Per Section

Null hypothesis: Intuitive design has no effect on completion time.

Alternative hypothesis: Intuitive design results in a faster completion time.

The alternative hypothesis of this study expected that players of the intuitive level would have a faster completion time than players of the control level. In Figure 36 the average time for all the participants has been plotted against section number for each level. The error bars on the graph show the standard error of the mean (SEM). The data collected shows that in 7 out of 14 sections the intuitive level had a higher average time than the control level. However, there is significant overlap in the error bars for each section which shows that the difference may be due to errors in the data.

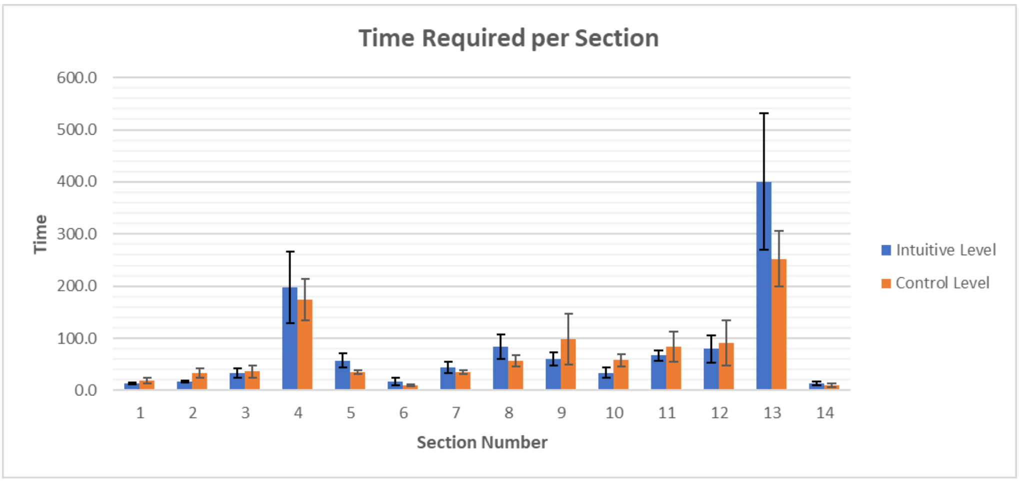


Figure 36: The average time per section for both levels, plotted on a bar chart

To determine whether there was a statistically significant difference between the levels a t-test was carried out. Table 12 shows the resultant p-value for the times collected for each section. None of these values are below the target p-value of 0.00357 meaning they are not statistically significant.

Table 12: The p-value for each section when comparing completion time

Section	1	2	3	4	5	6	7
p-value	0.594	0.399	0.549	0.603	0.113	0.342	0.309
Section	8	9	10	11	12	13	14
p-value	0.412	0.693	0.416	0.344	0.450	0.271	0.731

4.2.3 Number of Failures per Section

Null hypothesis: Intuitive design has no effect on number of failures.

Alternative hypothesis: Intuitive design results in fewer failures.

The number of failures per section was a combined total of how many times the player fell out of the level or was defeated by enemies in the game prototype. The alternative hypothesis predicted that players of the intuitive level would have fewer failures than players of the control level. Figure 37 is a bar chart showing the average number of deaths per section for each level. In 6 out of the 14 sections the players of the intuitive levels had a lower average number of deaths than the players of the control level. This means in most sections the alternative hypothesis was incorrect. The error bars show the standard error about the mean (SEM) and there is significant overlap between the two levels in most cases. This means the differences between the two levels may simply be due to errors in the data.

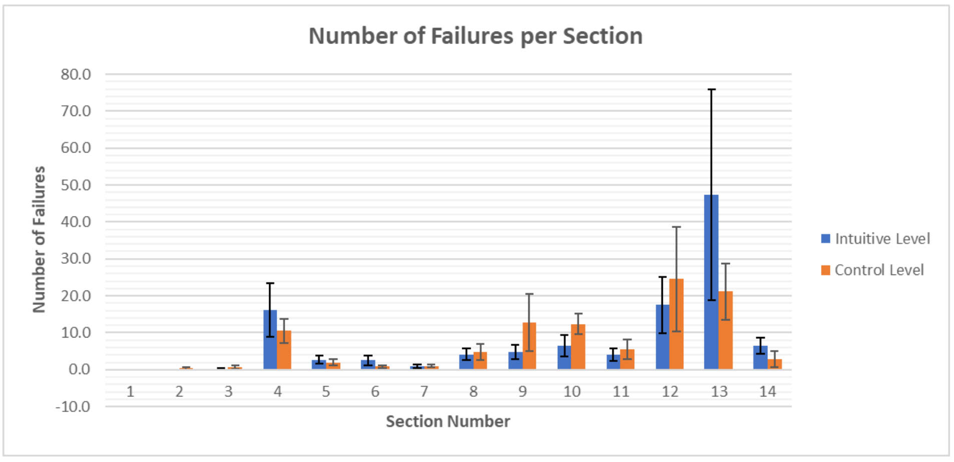


Figure 37: The average number of failures per section for both levels, plotted on a bar chart

A t-test was carried out on each section to analyse whether there was a statistically significant difference between the data sets for the two levels. Table 13 shows the resultant p-value for each section calculated by these t-tests. None of the sections had a p-value below the target of 0.00357 so this data is not statistically significant.

Table 13: The p-value for each section when comparing number of failures

Section	1	2	3	4	5	6	7
p-value	N/A	N/A	0.791	0.397	0.373	0.269	0.797
Section	8	9	10	11	12	13	14
p-value	0.472	0.606	0.392	0.764	0.785	0.322	0.530

4.2.4 Survey Result

Null hypothesis: Intuitive design has no effect on player experience.

Alternative hypothesis: Intuitive design improves player experience, making the game easier to understand and more engaging.

The survey was designed to try and analyse each participant's opinion of the game prototype after playing. The questions in the survey and their corresponding answer formats are shown in Table 14.

Table 14: The questions and their respective answer formats in the survey given to test participants

Question	Answer Format
How difficult to understand did you find the game mechanics?	Rating from 1-10 1 = I didn't understand, 10= I understood everything in the game
How difficult did you find the gameplay?	Rating from 1-10 1 = I found the game almost impossible, 10= I found the game easy
How much did you enjoy playing the game?	Rating from 1-10 1 = I did not enjoy it, 10= It was very enjoyable
How engaging did you find the game?	Rating from 1-10 1 = It did not hold my attention at all, 10= I was very absorbed in the game
Having played the game, do you now feel you have a good understanding of how it works?	Yes or No

The questions with responses that required a rating from 1-10 had labels assigned to the 1, 5 and 10 star ratings. The aim of this was to standardise player response because it is important that each participant understands the rating system to try and make the comparison as fair as possible. Figure 38 shows an example question in the survey that demonstrates these scale labels.

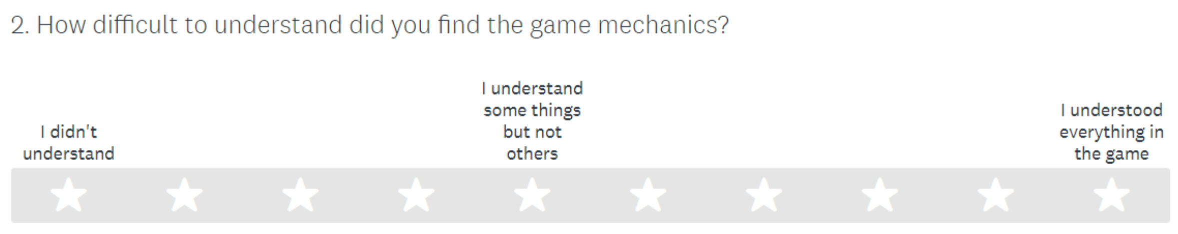

2. How difficult to understand did you find the game mechanics?
I didn't understand
I understand some things but not others
I understood everything in the game

Figure 38: An example question for the survey completed after participants played the game

The participants who played the intuitive level where hypothesised to perceive the game as easier to understand because of the intuitive design techniques. This was predicted to make the game more enjoyable and more engaging because the player could become immersed in the game more quickly. Section 2.3 discusses how intuitive design can improve player experience in more depth. The mean survey results for each level are shown in Table 15.

Table 15: The mean survey results for each players of each level

Level	How difficult to understand?	How difficult?	How enjoyable?	How engaging?	Understood Overall?
Intuitive	9.33	5.33	8.50	8.50	All yes
Control	8.67	5.17	7.17	7.33	All yes

As expected, players of the intuitive level found the game easier to understand, more enjoyable and more engaging. Every player who participated in the study answered that they understood the game overall after playing it. Players found the difficulty between the two levels very similar which was expected. The two levels were designed to be similar in difficulty to give a good comparison. Although the data shows trends towards the intuitive level as expected, this data must be scrutinised to define whether these trends are statistically reliable or not. A t-test was carried out on the results for each survey question and the resultant p-values are shown in Table 16.

Table 16: The p-values for each question in the survey

Question	How difficult to understand?	How difficult?	How enjoyable?	How engaging?	Understand Overall?
P-value	0.431	0.875	0.199	0.559	N/A

This t-test reveals that the survey results are not statistically reliable and therefore the trends in the data may be caused by randomness.

4.3 Summary

It was shown the game prototype met all the key requirements set out in the original requirements analysis. The game completely satisfied all the “must” and “should” priority requirements, while also completing some of the optional “could” requirements. This shows the prototype met or exceeded the initial planned functionality.

Result data was gathered from twelve test participants, six for each level in the game. For each player their completion time and number of failures were recorded for each section of the level. The averages for each level were then compared but no noticeable trends were observed. A t-test showed that no statistically reliable difference was observed between the two levels for completion time or number of failures.

Each test participant was asked to complete a survey after finishing the game. The purpose of the survey was to measure how much the player enjoyed the game and how difficult they perceived it to be. The survey results showed that on average players of the intuitive level enjoyed the game more and found it more engaging. However, the t-test showed there to not be a statistically reliable difference between the two levels.

Section 5 discusses some of the challenges faced when gathering results. Insights and design recommendations have been formulated to help future work in this field build on this study and overcome these challenges.

5 Discussion

5.1 Overview

The data recorded during this study was calculated to not be statistically significant in section 4. Despite this, the results showed promise and a lot can be learned from this investigation. A sample size of six people per level is small, which means a t-test would only show statistical significance for very large differences between the two levels (a high effect size). The survey results showed that players of the intuitive level enjoyed the game more. Using some of the design recommendations presented in this chapter a future study may be able to prove this to be statistically reliable. These design recommendations are based on challenges that were encountered during this study. This chapter also explores why completion time and number of failures may have issues accurately representing the player's performance.

Although the statistical data did not align with the hypotheses, notable differences where observed between the intuitive and control levels during the playtesting stage. Section 5.3 discusses some of these observations and explains what can be learned from them for future studies. If the differences in player decision making between the two levels could be captured this would be more representative data of the effectiveness of the intuitive design.

5.2 Challenges when Collecting Results

5.2.1 Mechanical Skill

One of the biggest factors affecting the results is the player's mechanical skill. Each test participant came into the playtesting with different amounts of experience and skill when it came to the game. This creates a learning bias because each player has different experiences which may help them better understand the game. Many factors affect this such as;

- How much prior experience they have playing any video games.
- How much prior experience they have playing other platformer games with similar mechanics.
- How familiar they are with using a controller.
- Skills that may have been trained anywhere that affect the game, e.g reaction time.

Mechanical skill is difficult to account for because it can't be easily quantified. It would be challenging to try and test each participant's skill before they played the game because of all the different factors that affect it. The ideal scenario would be to have each participant play both levels and compare their performance between the two. However, the experience the participant gains when playing the first level invalidates the data gathered in the second level.

The mechanical difficulty can unfortunately mask the desired results in unintended ways. For example, a player may immediately understand the mechanic required to complete a section because of the intuitive design. However, if they then find that section very mechanically challenging and struggle to complete it, the results for time and number of failures show they performed poorly. If the section is mechanically easy, then the player's understanding will be the limiting factor; if the section is mechanically hard then the player's mechanical skill will be the limiting factor.

Insight: If mechanical skill limits the player's progression this may hide how well they understood the game in the collected data.

In one case the mechanical difficulty proved too much for a participant to be able to complete the level, despite the fact they were playing the intuitive level. The contrast between some players completing the game with relative ease and others being unable to shows how heavily the players mechanical skill and learning bias affected the results.

A mechanically easier game would help to ensure that the player's understanding of the game was the limiting factor during the tests and not the mechanical difficulty. A better genre of game would have been something like a puzzle game.

Design recommendation: Select a genre for the prototype where the player's progression through the game is limited by their understanding of the game rather than mechanical skill. E.g a puzzle game

5.2.2 Playstyle

Playstyle means the way in which somebody plays a game. An individual's playstyle encompasses how they approach the game, the type of decisions they make and the techniques they priorities when playing a game. There are multiple playstyles that players can adopt when playing a platformer game. Some player will approach the game cautiously, taking their time when entering new sections of the level to explore and observe what is coming next. Other players are more reckless, attempting to move quickly through the level despite not knowing what is ahead. A cautious player may take longer to complete the game but have fewer failures in the process whereas a reckless player may fail many times more but complete the level quicker.

The significance playstyle can make to completion time is shown by section 1 of the level. Section 1 is the only section that is identical between the two levels. It is a short flat section at the very beginning of the game that was created to isolate data about the players behaviour when immediately starting the game. There are no challenges or obstacles so if each player were simply to move through the level they should have similar times. However, the data shows that some players took up to four times as long as others. Table 17 shows the completion time for section 1 for all twelve participants.

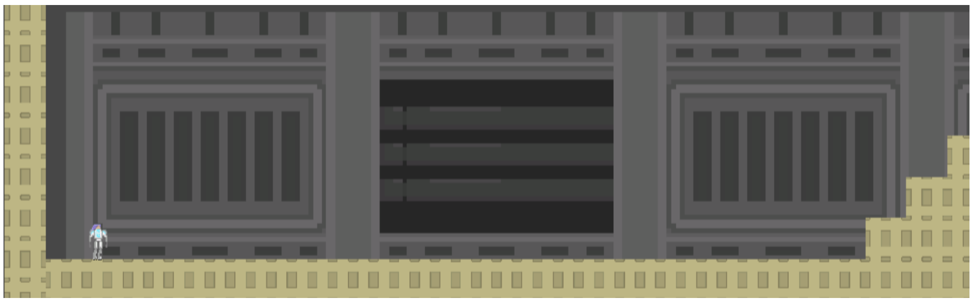


Figure 39: Section 1 of both the control and intuitive levels

Table 17: All the completion times for section 1 of both levels

Level	**Intuitive Level**						**Control Level**					
Section 1 Completion Time (s)	17.8	7.9	9.8	9.4	11.6	22.6	17.5	23.4	12.1	8.7	42.6	12.1

When observing the playtest, the cause of this was obvious. When the game first started some players immediately ran to the right and began progressing through the level. Other players began by staying on the first screen and testing out the mechanics. They would jump and shoot to get a feel for the game's actions before they moved anywhere. This shows how differently players can behave in the same circumstances.

Insight: Different players approach a game in different ways. Therefore, care must be taken when comparing their behaviour.

Another factor to consider is that players may alter their style in the testing environment. The participants involved in this study knew they were playing the game prototype for research purposes and that some sort of data was being collected. People may feel pressure to play the game in a different manner to how they would in a casual environment. For example, while playtesting some players expressed they were unsure if they should be defeating every enemy in the game or if it was ok to simply run past some. If a player attempts to defeat every enemy rather than passing some by this will affect their completion time and possibly number of failures. This shows playstyle can have a direct impact on results gathered.

It was hypothesised there would be a correlation between completion time and number of failures. The more times a player failed the longer they would take to progress through the level. However, the data shows players with similar completion times do not have similar numbers of failures. For example, the two participants shown in Table 18 have less than 10 seconds difference in completion time but one failed twice as many times as the other. Both players played the intuitive level, so the conditions were identical. This shows that time and number of failures were not the ideal statistics to gather when trying to gauge how well a player performed in the game. Section 6.2 discusses how future studies could better analyse player performance when playtesting a game prototype. Looking at how the player acted and how optimal the decisions they made were, could yield better results than difficult to control factors like time. Another option is to combine multiple statistics like completion time and number of failures into an overall performance rating.

Table 18: A comparison of total time and number of failures for two players who tested the intuitive level

Subject Number	**Time**	**Deaths**
3	1493.8	104
5	1500.7	284

Design recommendation: Be aware that different playstyles may affect the results. The best measure of performance may be to combine multiple statistics into an overall performance rating.

5.2.3 Player Testing

As described in section 3.1.4 the game was incrementally developed and tested by the developer. Developer testing is extremely beneficial during the implementation phase as it helps identify bugs early. The problem with developer testing is that the developer becomes very familiar with the game due to their extensive experience playing and implementing it. This experience means the developer struggles to see the game through the eyes of a new player who is unfamiliar with the game. For this reason, there are issues related to the design and usability of the game that are very hard for the developer to identify. One issue is that bugs may slip through because they do not affect the way the developer plays the game. The solution to this is to playtest the game with people who are unfamiliar with it during development to gather feedback.

Several small bugs were present in the game prototype and were identified when observing the playtesting sessions. To ensure the result data gathered could be compared fairly all participants played the same version of the game. This means that bugs that were only discovered during the result gathering stage could not be fixed. The list of known bugs that were present in the prototype during testing is shown in Table 19.

Table 19: The known bugs that were present in the game prototype

Description	Severity	Frequency	Solution
Some unintended edges stop the player from following but do not allow them to jump.	Low	Medium	Some platforms need to be restructured in level to remove edges
Armoured enemies inconsistently fail to block the player's projectiles.	High	Low	The interaction between the projectiles and armoured enemy needs to be debugged to locate the fault.
The player can occasionally jump off the invisible kill zone at the bottom of the screen.	Medium	Low	The raycast to check if the player is on the ground should ignore killzones
The enemies hitbox (the collider that defines the size of the enemy and where they can deal damage in the game) was slightly larger than the sprite making the player sometimes take damage when they should not have.	Low	High	Hitbox size should be adjusted to better match the sprites

None of these bugs were severe or common enough to considerably affect the results. However, an ideal prototype would be free of any bugs to give the most reliable platform for collecting data. Another option is that if a bug affects a player's performance in a section take note of it and discard the results from that section.

Design recommendation: Test with target users during the development phase to help identify bugs and improve the game prototype before testing begins.

5.3 Interesting Observations

During the playtesting stage some trends in behaviour were noticed when watching the participants play through the game prototype. Unfortunately, these were not captured in the data being measured and are therefore purely anecdotal. Despite this, these observations provide an interesting basis for future testing of intuitive design. If these observations could be measured accurately, they provide a more reliable method of measuring the effectiveness of intuitive design.

The situation where the player first encounters a switch in the game prototype was different in the two levels. In the intuitive level several blocks prevented the player from accessing the switch at first, meaning they had to move to the far side of it which revealed the first door in the game. When the player triggered the switch, the door would open and they could progress. In the control level these blocks did not exist, and the player could trigger the switch immediately. This is shown in Figure 40. Some players of the control level triggered the switch when they could not see the door. This confused many players as they did not immediately see feedback from the action they had performed. Several players triggered the switch multiple times to see if they could work out the effect it was having. In support of this, this section of the level was one of the few where the intuitive level had a faster average completion time than the control level. Recording data such as how many times this switch was activated would show a distinct difference between the two levels, as no players of the intuitive level felt the need to trigger the switch more than once.

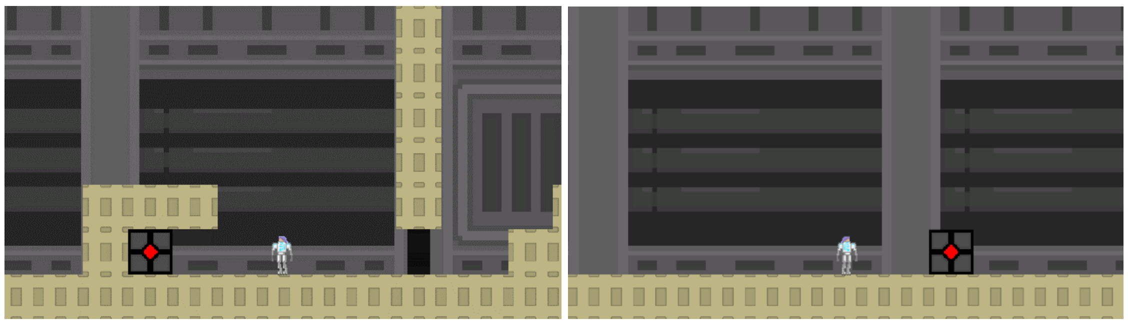


Figure 40: A comparison of the first encounter with a switch between the intuitive level (left) and control level (right)

Figure 41 shows a comparison of the same challenge in the control and intuitive levels where the player must reach a platform beneath themselves. Players of the control level used the double jump to reach the platform beneath, whereas players of the intuitive level would use the aerial shoot to knock themselves backwards through the gap. In the intuitive level players could easily have used the double jump to reach the platform beneath, then shot the switch from a standing position but only one of the six players opted to do this. Using the double jump is a much more mechanically complex manoeuvre than simply shooting while falling but as the control level did not go to the same lengths to teach the player this, it appeared fewer of them understood. In a section like this, a future study could record which action the player took as the test data. They could then compare whether players chose the optimal action or a sub-optimal action in each situation to gauge how well the player understands the game.

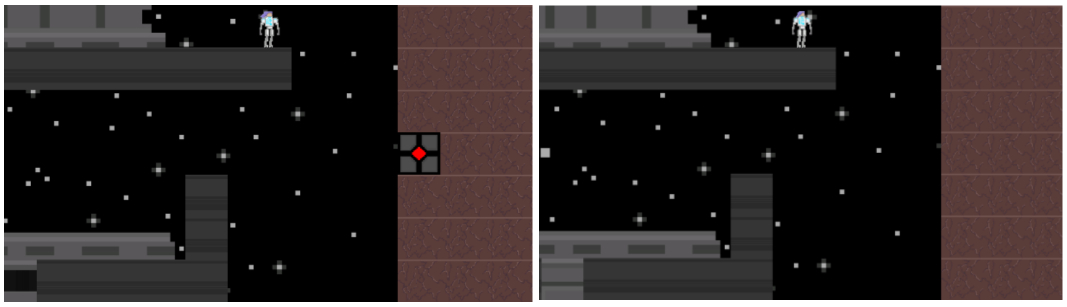


Figure 41: A comparison of a challenge where the player must reach a platform below themselves between the intuitive level (left) and control level (right)

These observations occurred frequently during testing, even in the relatively small sample pool. Unfortunately, there is no concrete data to back them up because completion time and number of failures were influenced by too many other factors. User testing during the development process (as described in section 5.2.3) could be used to identify areas of the game where this type of data could be collected rather than the developer simply guessing.

6 Conclusions and Future Work

6.1 Conclusions

6.1.1 Overview

This dissertation investigated what intuitive game design is and whether using it can improve players experience in video games. Intuitive level design was hypothesised to both: improve player performance and increase player enjoyment when compared to a level that did not use intuitive design. A game prototype was developed to test these hypotheses by comparing data gathered from two different levels created in the game. One level used intuitive design technique and the other acted as a control. People were recruited to play one of the two levels and data was recorded about their performance in the level. Participants also answered a short survey about how much they enjoyed the game and how difficult they perceived it to be.

No trends were observed in the data collected for completion time or number of failures and there was no statistically reliable difference between the data for the two levels. Despite this, clear differences were observed between the behaviours of the players of each level. Participants who played the intuitive level were found to enjoy the game more than players of the control level, but this difference was shown to not be statistically reliable. Some of the challenges that affected the results for this study were explored in detail. Insights were drawn from these challenges in order to provide design recommendations for future work in this field.

6.1.2 Literature Review

The literature review went into depth about what intuition means for video games, and what benefits it can offer. The goal of intuitive design is to allow new players to easily understand games and to give them the tools to work things out for themselves. Unintuitive games can lead to players becoming frustrated or losing interest. Some intuitive design techniques were explored with relevant examples from commercial games. Several intuitive design techniques were discussed and case studies that use them were analysed.

As part of this study a game prototype was developed and used to test intuitive level design. The different development technology and build platforms available were compared to evaluate which best suited this project. Unity was selected as a game engine to build the prototype in due to its comprehensive features, good documentation and its compatibility with solo developers. Windows was chosen as the build platform as the game will be tested and used to collect data on a single Windows laptop.

6.1.3 Implementation

The lifecycle of the game prototype was described in section 3. A thorough design process defined all the mechanics within the game prototype along with the technical software specification. A game design document was created which held all the design information in a single point of reference. Techniques such as software design patterns were used to ensure the prototype was a well-structured piece of software which could be easily developed further in the future.

A project plan was created which divided the implementation process into tasks and estimated their timescales. Despite the implementation of the game prototype deviating from the project plan significantly it was still completed on time. How the project deviated from the plan and what measures were taken to counteract this are discussed in depth in section 3.1.6. The design methods and tools/software used for the implementation were also described in detail.

6.1.4 Results and Evaluation

Twelve participants were recruited to play the game prototype and data was gathered about how they performed. Statistics about their completion time and number of failures were measured for each section of the level and they were asked to complete a short survey after completing the game. T-tests were used to evaluate whether there was a statistically reliable difference between the data collected for each level. The Bonferroni correction was used to adjust the target p-value because there was a t-test for each section meaning a much higher chance of a positive result being due to randomness.

There was no trend observed between the two levels for completion time or number of failures and these results did not show a statistically reliable difference for the two levels. The survey results showed that players found the intuitive level easier to understand and more engaging. However, these results were also found to not show a statistically reliable difference.

6.1.5 Discussion

This study explored intuitive level design and showed promising differences between the two levels in the prototype. The challenges that affected this study were discussed and future design recommendations were given based on the insights gained in this study. Recommendations were given for both creating a game prototype and improving the test procedure in any future work in this field. Clear differences were observed between the levels during the testing of the game prototype and some of these were described in detail. Creating a test procedure that better measured these differences could be a much more reliable way of proving the effectiveness of intuitive design.

6.2 Future work

I believe this study showed promise and further work investigating this topic could build upon knowledge learned during this study to collect more meaningful results.

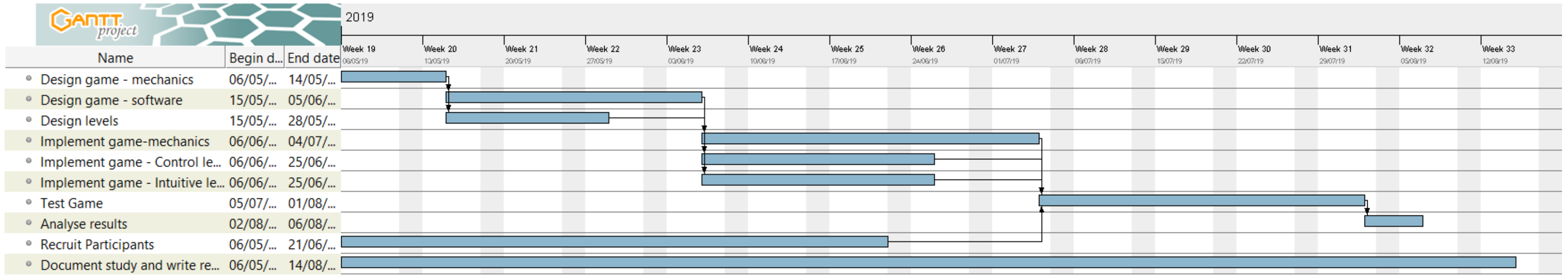
Future studies can learn from some of the challenges faced during this study to help produce a game that is better suited to testing intuitive design. The game created for this study relied too much on mechanical skill, which prevented the results from accurately demonstrating player understanding. Opting for a less skill-based game genre would help to ensure the limiting factor during testing is player understanding, which would yield better results.

This study also offered insight into how to analyse the effectiveness of intuitive design. Completion time and number of failures proved to be affected by too many factors to show significant results. This study showed that players with different styles achieved a large variation in number of failures for the same completion time and vice versa. Looking at the decisions a player makes or which action they choose in a certain situation could give a better understanding of how well they understand the game. A game used to test intuitive design should always have enough depth that players with a better understanding can make more optimal choices, while allowing players with a lesser understanding to still progress less efficiently.

The techniques presented in this study do not have to be applied exclusively to games. Many of the discussed principles are shared with user interface (UI) design and could be applied to help design more intuitive UIs for a large variety of applications. These techniques could also be applied in other fields which have been gamified to create more intuitive results. This could be especially useful in gamified training or tutorial packages as intuitive design can improve the speed and efficiency of learning (as discussed in section 2.3.3).

7 Appendix

7.1 Appendix A – Gantt Chart



Name	Begin d...	End date
Design game - mechanics	06/05/...	14/05/...
Design game - software	15/05/...	05/06/...
Design levels	15/05/...	28/05/...
Implement game-mechanics	06/06/...	04/07/...
Implement game - Control le...	06/06/...	25/06/...
Implement game - Intuitive le...	06/06/...	25/06/...
Test Game	05/07/...	01/08/...
Analyse results	02/08/...	06/08/...
Recruit Participants	06/05/...	21/06/...
Document study and write re...	06/05/...	14/08/...

7.2 Appendix B – Game Design Document

Game Design Document

Overview

The game will be a short 2D platformer game with pixel art graphics and unique movement mechanics. The core gameplay will stem around the player being able to shoot projectiles from their feet. Firing these projectiles will create a knockback that can be used as a movement mechanic by the player. For example, shooting downwards in the air to double jump or backwards in the air for a boost of speed forwards.

The game will have a sci-fi theme and will be set in space. The game features a female protagonist fighting against an unnamed group of hostile aliens. The goal of the game is simply to reach the end of the level. The player will reach checkpoints as they progress through the level and being "killed" in the game will return the player to the most recent checkpoint.

The game will be both mechanically and mentally challenging. Precision and speed will be required to manoeuvre through the level but the player must also understand the game mechanics and use them to clear sections of the level that require using them in a specific way.

Premise

Location: Space, foreign fictional galaxy.

Player Goal: Make their way through entire level, bypassing obstacles and defeating enemies.

Villains: A gang of cyber-terrorists operating an intergalactic black-market. Members consist of many different races of aliens and includes humans too.

Conflict: Main conflict between policing force and terrorist gang. Villains are immediately hostile to player.

Look and feel

2D pixel art graphics. Low pixel count, similar style to NES games.

Crisp, responsive controls.

Mechanics

The game is a 2D platformer based around the core mechanic that a player can shoot a projectile out of the soles of their feet.

The player will progress through a linear level, bypassing hazards and defeating enemies. There will be some simple puzzles/challenges to solve to progress and some challenge of mechanical skill. The player can be defeated and will respawn at the last checkpoint they passed.

Player Mechanics

Move, jump, shoot forwards on ground, shoot upwards on ground, shoot downwards in air, shoot forwards in air, shoot upwards in air.

As stated, the game will be designed around this core mechanic, meaning other mechanics must build upon it. Below is a list of the secondary mechanics planned, and how they interact with the main mechanic.

- Move: Left/Right movement, affected by gravity.
- Jump: Short hop, can retain forward momentum or jump on spot.
    - Jump higher the longer button is pressed?
- Shoot(grounded): Movement must stop. Slight delay to raise leg before firing.
- Shoot upwards(grounded): Movement must stop. Slight delay to go with animation.
- Shoot downwards(air): Boosts player upwards, effectively giving them double jump. Projectile travels downwards.
- Shoot forwards(air): Boosts player backwards and slightly up. Projectile travels forwards.
- Shoot upwards (air): Doesn't affect air velocity, projectile travels up.

Enemies

Some enemies will be invincible from certain directions so that the player has to attack them a specific way. Character art should hint at where they are invincible/vulnerable.

Enemy 1, no invincibility.

Enemy 2, invincible sides.

Enemy 3, invincible top.

Switches and Switchable Objects

Switch block that flicks between on and off when shot by player.

Objects that are connected to a switch and switch on and off with it.

Would be good to have a visual indication of what switch is connected to like a light trail or something.

<u>Objects</u>

- Door: open and closed

Player Death

Player can be defeated by enemies or environmental hazards (e.g pits).

On death player returns to last checkpoint. This should reset all level features (enemies, switches etc.)

Player will have unlimited lives.

Level

- **Ground:** Doesn't have to be flat, will have step ups and downs to form ledges and pits.
- **Holes:** Player can fall out of level and die.
- **Floating platforms:** Can jump on top and run along.
-

Aesthetics

Main Character (Player)

Appearance: Human female. Androgynous space suit/armour with no helmet. Shoulder length hair. Rocket boots.

Personality: TBD (not required)

Story (not implemented due to time constraints)

Recover a missing ally who previously attempted to infiltrate villain's gang. Their fate is unknown but will be discovered at end of level. Also aiming to damage our destroy the villains' organisation if possible.

Main conflict between policing force and terrorist gang. Villains are immediately hostile to player.
Villains: A gang of cyber-terrorists operating an intergalactic black-market. Members consist of many different races of aliens and includes humans too.

Minimum Viable Product (MVP)

If time becomes and issue some features will have to be cut so that evaluation can begin.

Essential Mechanics (high priority)

- Crisp/responsive player movement and jumping.
- Player shooting mechanic on ground and in air.
- Switches and switchable objects.
- Some type of enemy.
- Basic graphical assets for level, player and enemy/enemies.
- Projectile reflectors

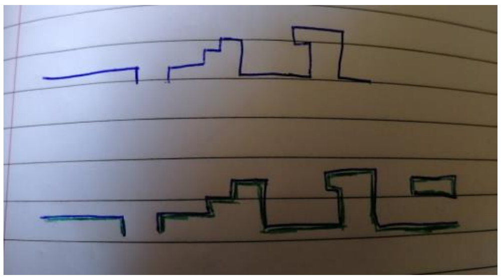
Non-essential Mechanics (low priority)

- Grappling points.
- Shooting upwards.
- Multiple enemy varieties.
- Audio Assets
- Higher quality graphical assets.
- Story and character development

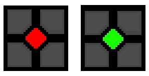
Assets List

Level (2D)

- Modular tile assets to build level. Flat ground, pits, building blocks
    - The theme is really up to you. I was either thinking like spaceship interior or some sort of alien planet, which basically gives you the freedom to do what you want within a scifi setting. Is that cool with you?

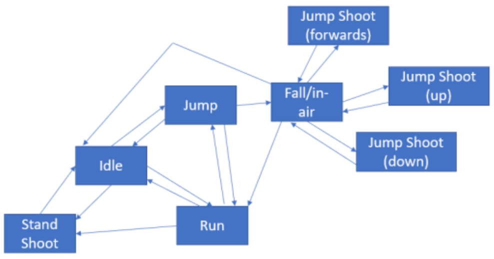


- Floating platforms (Could have supports in background if you want)
- Switch block to flick between 2 states when shot by player. Could create the states using a light in Unity rather than changing the asset if you want. I used this in the prototype I made:



Player(2D)

- Description is in characters section.
- 2d pixel art
- Animations to match each player state (in the state diagram below).



Projectile(2D)

- Little 2d projectile to be shot from players feet

Enemies(2D)

- In the end 3 different enemy types. Enemy art should show their invincibility and vulnerabilities. See description in enemies section.
- Just start with 1 enemy. Multiple types can be cut if it becomes an issue for time so don't prioritise multiple enemies.
- 2d art like character

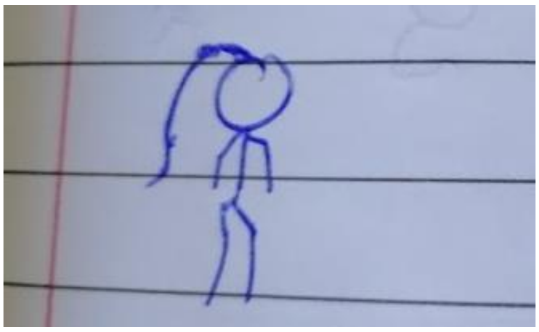
Background

- Scrolling background to go behind level. Not crucial to gameplay, could use a solid colour so this is not a high priority at all.

Player Animations

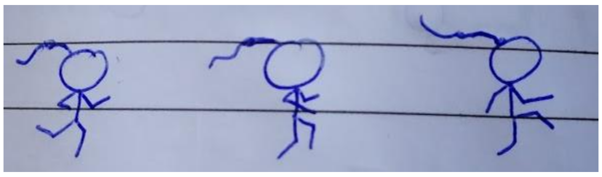
Idle

Usual thing, slight movement. Maybe slightly swaying hair in wind.



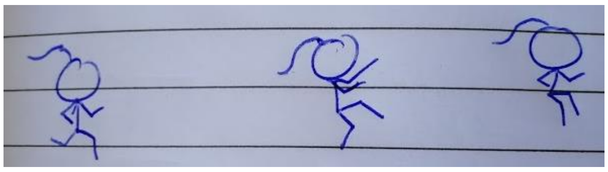
Run

Hair trailing behind



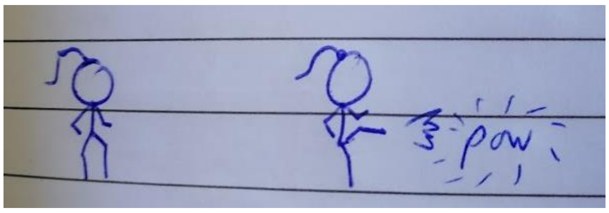
Jump

First jump doesn't use rocket boots



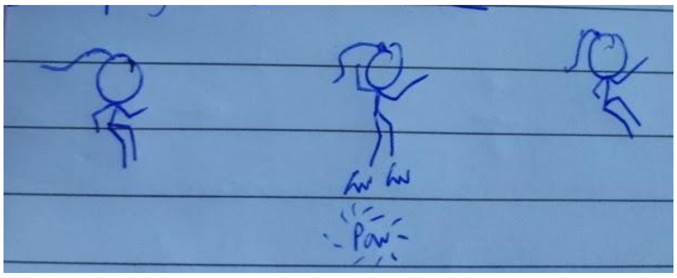
Standing Shoot

Character will be stationary, not running.

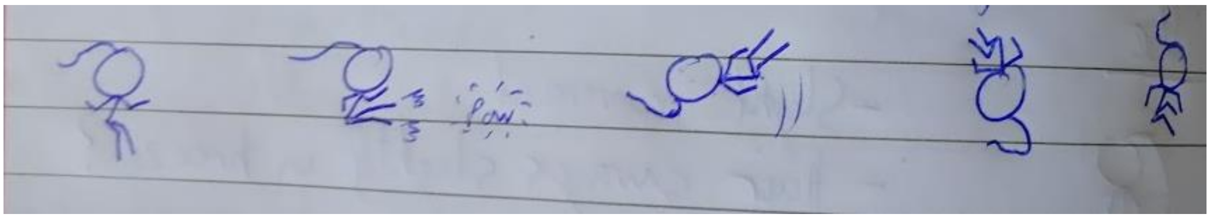


Jumping Shoot (Down)

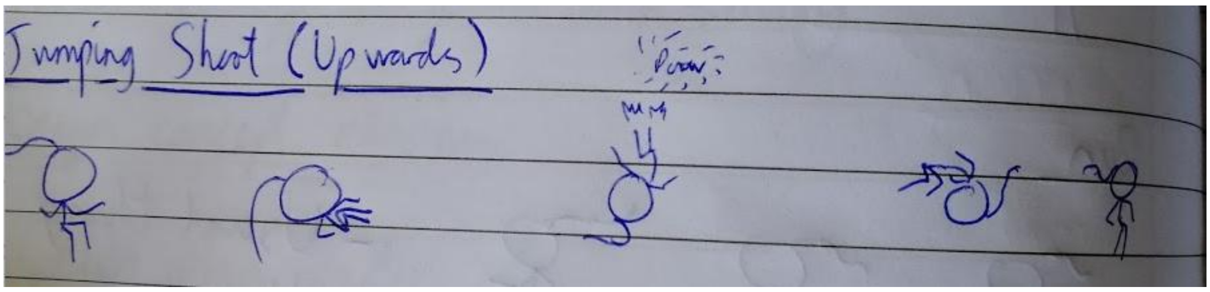
Basically a double jump



Jumping Shoot (forwards)



Jumping Shoot(Upwards)



Software Technical Specification

Engine: Unity

Unity Version: 2019.1.4f1

Unity will handle: physics, graphics rendering, game loop(update methods), GameObject and Component structure, detecting inputs, camera

New code for: game logic, game cycle(menu-new game-game over), handling user input, logging statistics

Design Patterns and High Level Structures

Observer pattern for UI (not used)

Singleton for tracking session data and logging stats.

Switch class (with reference to switchable)

Switch-able objects interface (Door etc.)

State machine for player.

State interface with class for each state using that interface.

Use static states to swap between them.

Entry and exit actions in each state allow changes to happen when transitioning

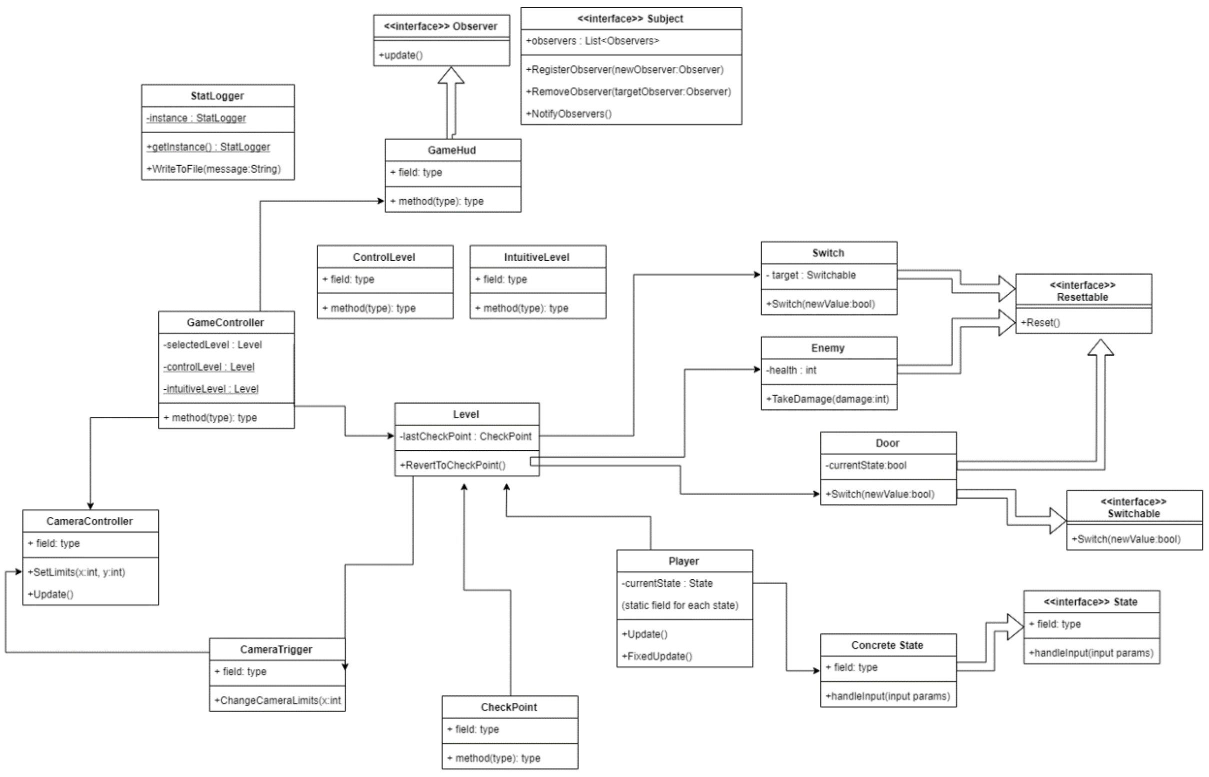
We are using Game Loop, Update and component patterns by using Unity

Could use “Type Object” pattern for enemies but probably not worth it due to limited number. Better with inheritance. Enemy superclass

Object pool projectiles? (probably won’t be needed)

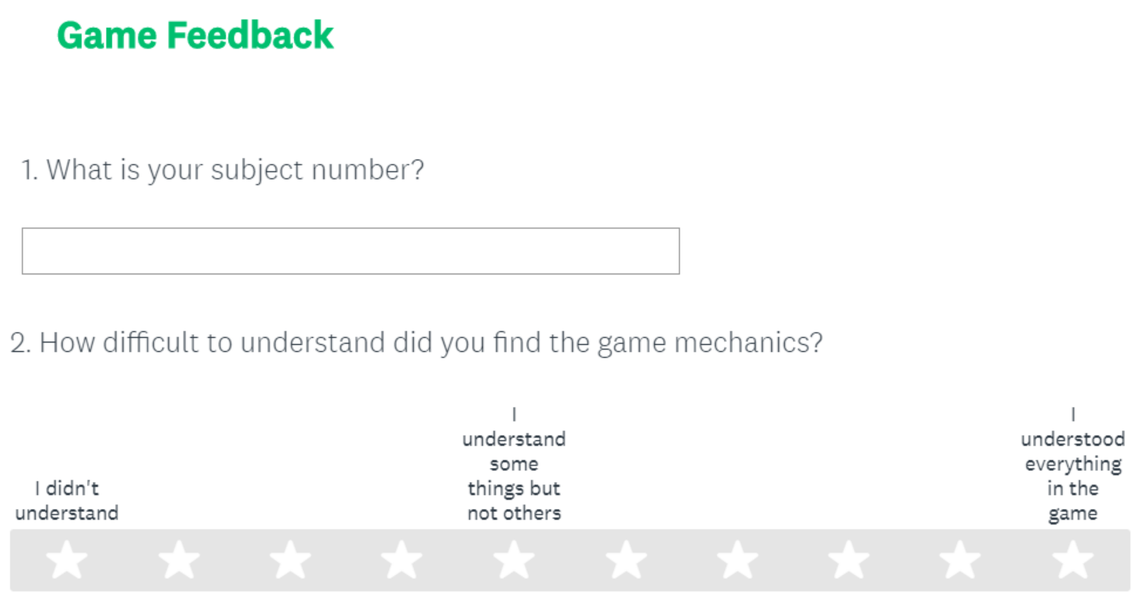
Resettable interface for anything in level

Preliminary Class Diagram



7.3 Appendix C – Survey

This survey was implemented using SurveyMonkey, a free online tool for hosting surveys.



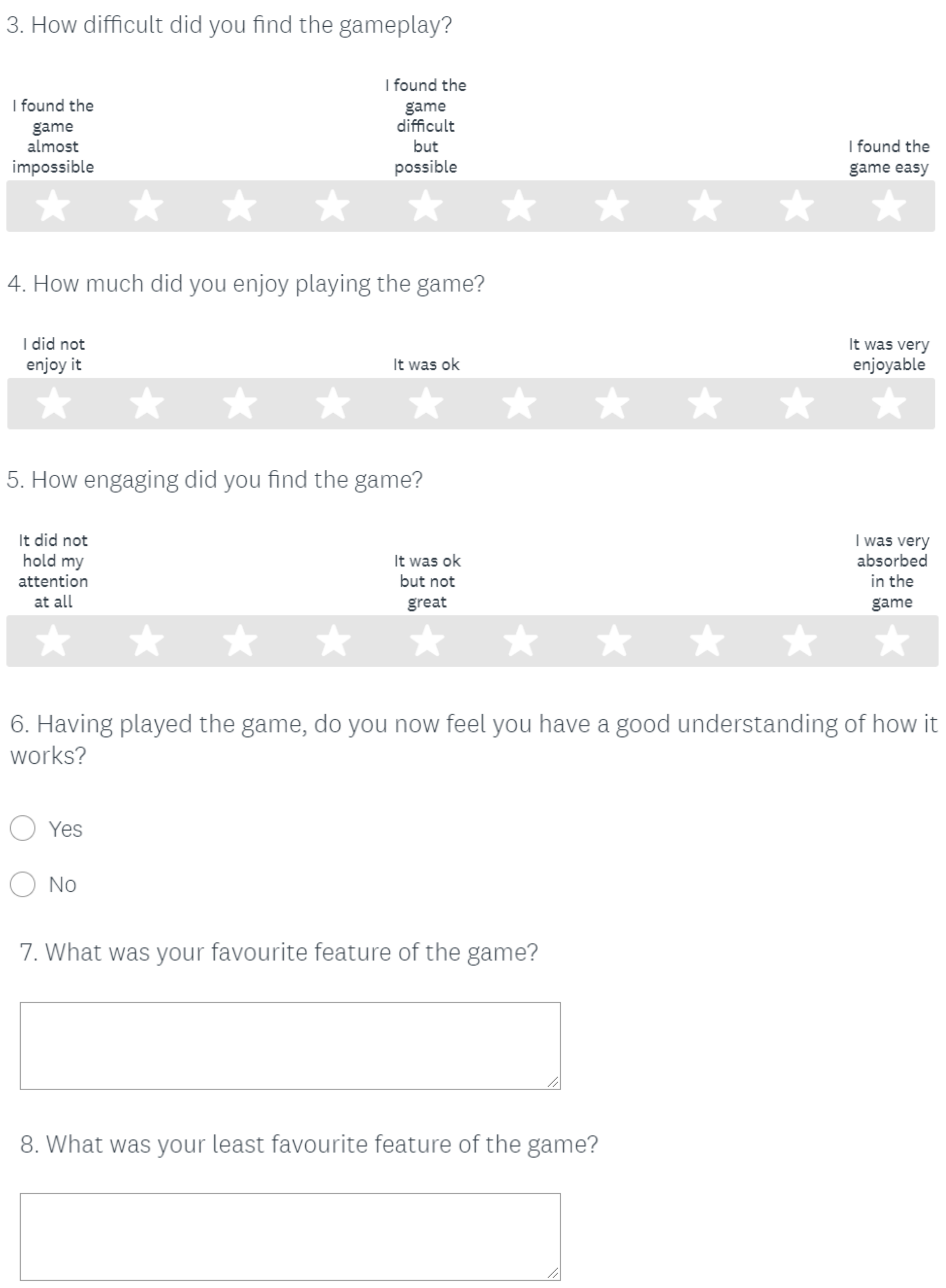
Game Feedback

1. What is your subject number?

2. How difficult to understand did you find the game mechanics?

I didn't understand | I understand some things but not others | I understood everything in the game

★ ★ ★ ★ ★ ★ ★ ★ ★ ★



3. How difficult did you find the gameplay?

I found the game almost impossible				I found the game difficult but possible					I found the game easy
☆	☆	☆	☆	☆	☆	☆	☆	☆	☆

4. How much did you enjoy playing the game?

I did not enjoy it				It was ok					It was very enjoyable
☆	☆	☆	☆	☆	☆	☆	☆	☆	☆

5. How engaging did you find the game?

It did not hold my attention at all				It was ok but not great					I was very absorbed in the game
☆	☆	☆	☆	☆	☆	☆	☆	☆	☆

6. Having played the game, do you now feel you have a good understanding of how it works?

○ Yes

○ No

7. What was your favourite feature of the game?

8. What was your least favourite feature of the game?

7.4 Appendix D – Consent Form

Stuart Paterson MSc Project

Game Prototype Consent Form

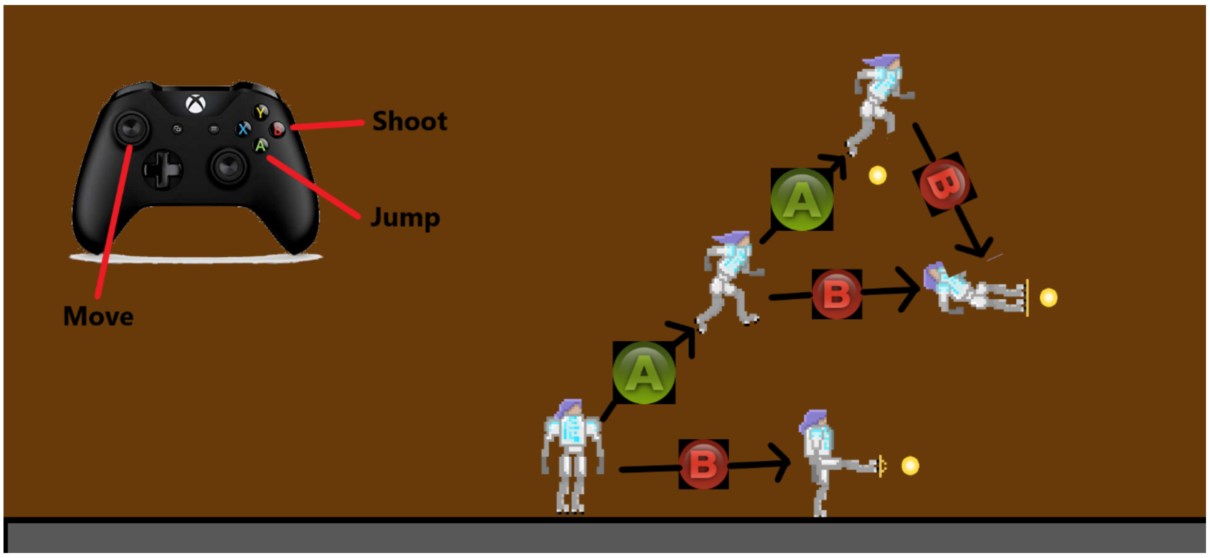
By signing this consent form I agree to the following statements.

- I understand that data will be gathered about my performance in the game during the playtest session.
- I give consent to this data being stored anonymously and used for the purposes of this research project.
- I understand my personal information will not be stored and any data will be unlinked to my personal information.
- I understand the game will require some mechanical skill and the use of a controller or mouse and keyboard.
- I will make the tester aware of any medical conditions that may be relevant to my own health and safety during the playtesting, before it begins.
- I understand my medical information will not be stored and will only be used to determine whether I am eligible to participate in the playtesting.
- I understand I can stop and leave the experiment at any point and am in no way compelled to finish the game.

Rather than signing your name, please indicate your consent to the above conditions by placing an X in the box below. Make sure to return a saved PDF copy of this consent form with your test data if you are playing remotely!

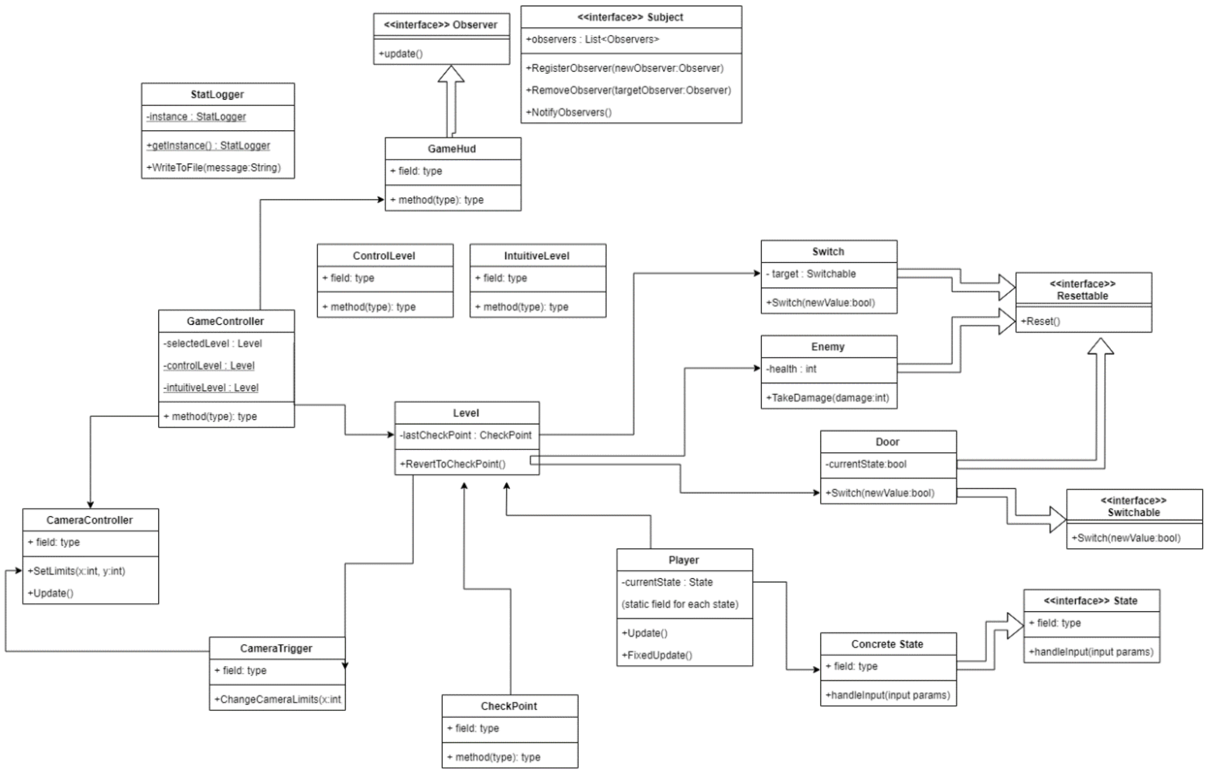
I consent to all terms and conditions on this form ☐

7.5 Appendix E – Instruction Image

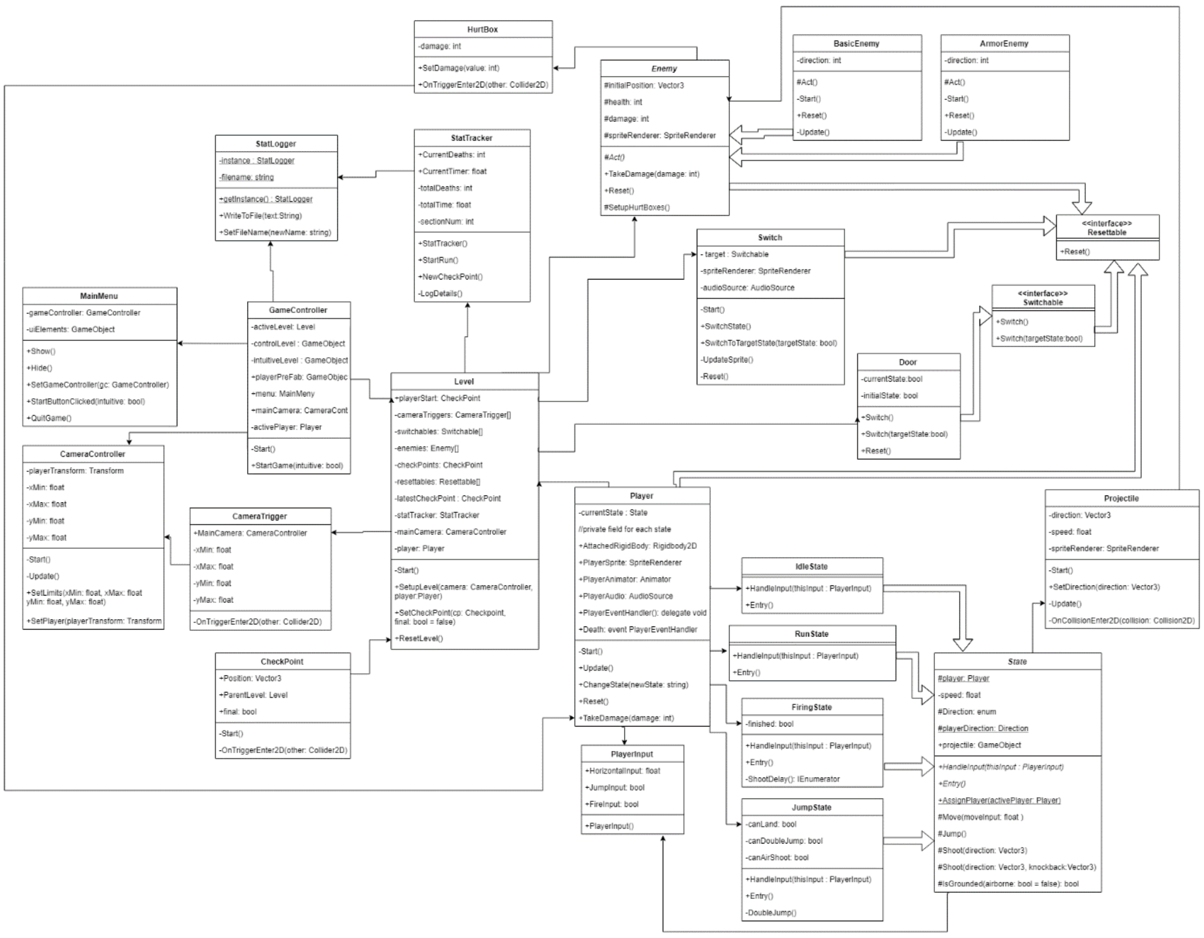


7.6 Appendix F – Unlabelled Class diagrams

Plan Diagram



Final Diagram


HurtBox
-damage: int
+SetDamage(value: int)
+OnTriggerEnter2D(other: Collider2D)
Enemy
#initialPosition: Vector3
#health: int
#damage: int
#spriteRenderer: SpriteRenderer
#Act()
+TakeDamage(damage: int)
+Reset()
#SetupHurtBoxes()
BasicEnemy
-direction: int
#Act()
-Start()
+Reset()
-Update()
ArmorEnemy
-direction: int
#Act()
-Start()
+Reset()
-Update()
StatLogger
-instance : StatLogger
-filename: string
+getInstance() : StatLogger
+WriteToFile(text:String)
+SetFileName(newName: string)
StatTracker
+CurrentDeaths: int
+CurrentTimer: float
-totalDeaths: int
-totalTime: float
-sectionNum: int
+StatTracker()
+StartRun()
+NewCheckPoint()
-LogDetails()
<<interface>> Resettable
+Reset()
Switch
- target : Switchable
-spriteRenderer: SpriteRenderer
-audioSource: AudioSource
-Start()
+SwitchState()
+SwitchToTargetState(targetState: bool)
-UpdateSprite()
-Reset()
<<interface>> Switchable
+Switch()
+Switch(targetState:bool)
MainMenu
-gameController: GameController
-uiElements: GameObject
+Show()
+Hide()
+SetGameController(gc: GameController)
+StartButtonClicked(intuitive: bool)
+QuitGame()
GameController
-activeLevel: Level
-controlLevel : GameObject
-intuitiveLevel : GameObject
+playerPreFab: GameObjec
+menu: MainMeny
+mainCamera: CameraCont
-activePlayer: Player
-Start()
+StartGame(intuitive: bool)
Level
+playerStart: CheckPoint
-cameraTriggers: CameraTrigger[]
-switchables: Switchable[]
-enemies: Enemy[]
-checkPoints: CheckPoint
-resettables: Resettable[]
-latestCheckPoint : CheckPoint
-statTracker: StatTracker
-mainCamera: CameraController
-player: Player
-Start()
+SetupLevel(camera: CameraController, player:Player)
+SetCheckPoint(cp: Checkpoint, final: bool = false)
+ResetLevel()
Door
-currentState:bool
-initialState: bool
+Switch()
+Switch(targetState:bool)
+Reset()
CameraController
-playerTransform: Transform
-xMin: float
-xMax: float
-yMin: float
-yMax: float
-Start()
-Update()
+SetLimits(xMin: float, xMax: float yMin: float, yMax: float)
+SetPlayer(playerTransform: Transform
CameraTrigger
+MainCamera: CameraController
-xMin: float
-xMax: float
-yMin: float
-yMax: float
-OnTriggerEnter2D(other: Collider2D)
Player
-currentState : State
//private field for each state
+AttachedRigidBody: Rigidbody2D
+PlayerSprite: SpriteRenderer
+PlayerAnimator: Animator
+PlayerAudio: AudioSource
+PlayerEventHandler(): delegate void
+Death: event PlayerEventHandler
-Start()
+Update()
+ChangeState(newState: string)
+Reset()
+TakeDamage(damage: int)
Projectile
-direction: Vector3
-speed: float
-spriteRenderer: SpriteRenderer
-Start()
+SetDirection(direction: Vector3)
-Update()
-OnCollisionEnter2D(collision: Collision2D)
IdleState
+HandleInput(thisInput : PlayerInput)
+Entry()
RunState
+HandleInput(thisInput : PlayerInput)
+Entry()
CheckPoint
+Position: Vector3
+ParentLevel: Level
+final: bool
-Start()
-OnTriggerEnter2D(other: Collider2D)
FiringState
-finished: bool
+HandleInput(thisInput : PlayerInput)
+Entry()
-ShootDelay(): IEnumerator
State
#player: Player
-speed: float
#Direction: enum
#playerDirection: Direction
+projectile: GameObject
+HandleInput(thisInput : PlayerInput)
+Entry()
+AssignPlayer(activePlayer: Player)
#Move(moveInput: float)
#Jump()
#Shoot(direction: Vector3)
#Shoot(direction: Vector3, knockback:Vector3)
#IsGrounded(airborne: bool = false): bool
PlayerInput
+HorizontalInput: float
+JumpInput: bool
+FireInput: bool
+PlayerInput()
JumpState
-canLand: bool
-canDoubleJump: bool
-canAirShoot: bool
+HandleInput(thisInput : PlayerInput)
+Entry()
-DoubleJump()

7.7 Appendix G – State Superclass

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public abstract class State : MonoBehaviour
{
    public GameObject projectile;

    protected static Player player;
    protected Vector3 playerSizeOffset = new Vector3(0.0f, -0.51f, 0.0f);
    protected Vector3 horizontalRayOffset = new Vector3(0.25f, 0.0f, 0.0f);
    private float speed = 5.0f;

    protected enum Direction { LEFT=-1, RIGHT=1, };
    protected static Direction playerDirection = Direction.RIGHT;

    public abstract void HandleInput(PlayerInput thisInput);
    public abstract void Entry();

    public static void AssignPlayer(Player activePlayer)
    {
        player = activePlayer;
    }

    protected void Move(float moveInput)
    {
        Vector3 moveVec = new Vector3(moveInput, 0.0f, 0.0f) * Time.deltaTime * speed;
        transform.position += moveVec;

        //Ignore if move input is 0. Don't want to change the player's direction if
they don't input anything
        if (moveInput > 0.0f)
        {
            playerDirection = Direction.RIGHT;
            player.PlayerSprite.flipX = false;
        }else if (moveInput < 0.0f)
        {
            playerDirection = Direction.LEFT;
            player.PlayerSprite.flipX = true;
        }
        player.PlayerAnimator.SetBool("running", true);
    }

    protected void Jump()
    {
        player.AttachedRigidBody.velocity = Vector3.zero;
        player.AttachedRigidBody.AddForce(Vector2.up * 10.0f, ForceMode2D.Impulse);
        player.PlayerAnimator.SetTrigger("jump");
        player.PlayerAudio.PlayOneShot(player.jumpSound);

    }

    protected void Shoot(Vector3 direction)
    {
        //Instantiate the projectile
        GameObject tempProjectile =
(GameObject)Instantiate(Resources.Load("Projectile"),
            transform.position + direction * 0.3f,
             Quaternion.identity);
        //Set the direction of the projectile
        tempProjectile.GetComponent<Projectile>().SetDirection(direction);
        player.PlayerAudio.PlayOneShot(player.shootSound);
    }

    protected void Shoot(Vector3 direction, Vector3 knockback)
    {
        GameObject tempProjectile =
(GameObject)Instantiate(Resources.Load("Projectile"),
            transform.position + direction * 0.55f,
             Quaternion.identity);
        tempProjectile.GetComponent<Projectile>().SetDirection(direction);
        //Add knockback
        player.AttachedRigidBody.velocity = Vector3.zero;
        player.AttachedRigidBody.AddForce(knockback * 8.0f, ForceMode2D.Impulse);
        player.PlayerAudio.PlayOneShot(player.shootSound);
    }

    protected bool IsGrounded(bool airborne = false)
    {
        if (airborne)
        {
            //When player is jumping dont send out side rays to prevent walljumping
            RaycastHit2D hit1 = Physics2D.Raycast(transform.position +
playerSizeOffset,
             -0.05f * transform.up, 1.0f);
            return hit1;
        }
        else
        {
            //Send out 3 rays to detect if player on ground
            //One ray in middle and one at either edge. Allows player to jump when
half off a platform
            RaycastHit2D hit1 = Physics2D.Raycast(transform.position +
playerSizeOffset,
                -0.05f * transform.up, 1.0f);
            RaycastHit2D hit2 = Physics2D.Raycast(transform.position +
playerSizeOffset + horizontalRayOffset,
                -0.05f * transform.up, 1.0f);
            RaycastHit2D hit3 = Physics2D.Raycast(transform.position +
playerSizeOffset - horizontalRayOffset,
                -0.05f * transform.up, 1.0f);
            return hit1 || hit2 || hit3;
        }

    }
}
```

7.8 Appendix H – Level Explanation

Level Breakdown

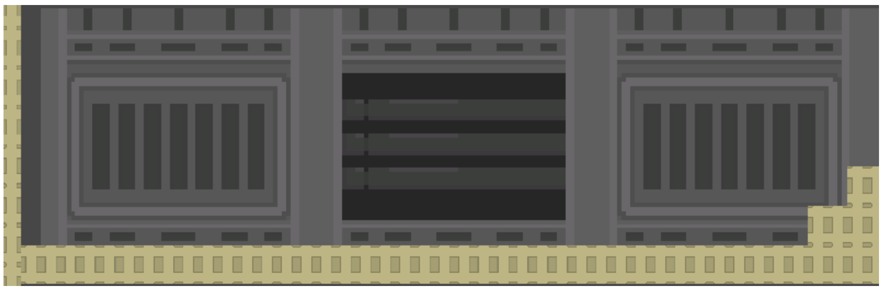
Introduction

This appendix describes the differences between the intuitively designed level and control level for the game prototype. The levels were split into sections for capturing results which also acted as checkpoints for the player. Some sections are identical between the two levels.

Generally, the initial sections were to teach the player the mechanics and these have the most differences between the two levels. The later sections were to test the players understanding of the game and many are identical between the two levels.

Section 1

Identical for both levels. Short flat section, player spawns at on left upon starting the game.



Section 2

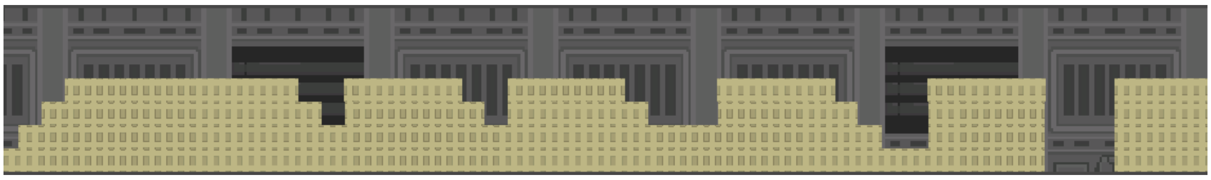
Introduces the player to jumping and double jumping mechanics.

In the intuitive level, double jump is not required for the first 2 jumps. These teach the player basic jumping. Then when they encounter the larger pits they realise the jump is not enough to get them across and must learn the double jump. They have 2 practice attempts with pits that they can climb out of and then one final attempt where they have a chance at failing.

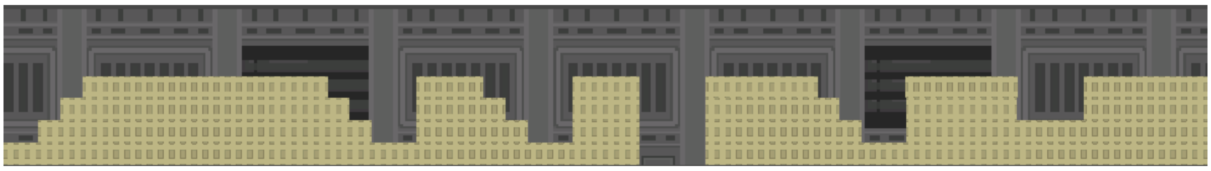
Here experiential learning is being used to teach the player jumping by letting them practice in an environment they can't fail. Iterative learning is being used to introduce the jump and double jump mechanics one after another to not overwhelm the player.

The control level's gaps all require double jump and the pit where they can fail is earlier so they have less experience.

Intuitive



Control



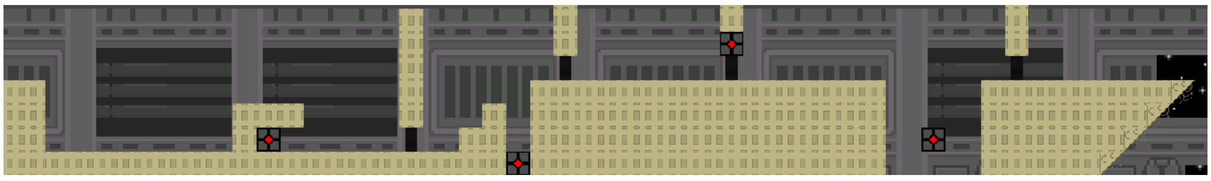
Section 3

This section introduces the switches to the player.

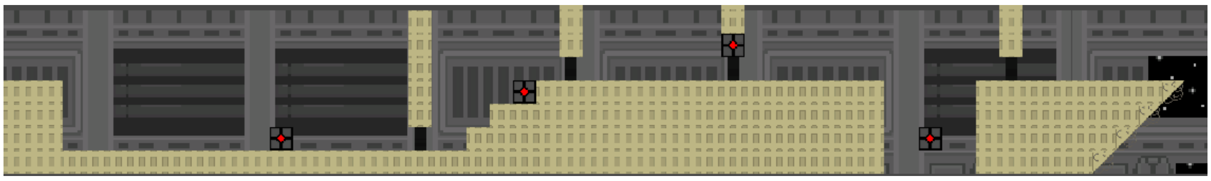
When encountering the first switch the intuitive level ensures that the player is in a position where the door is also visible on their screen before they can trigger the switch. They then get immediate feedback when they trigger the switch that it opens a door. This is an antepiece that teaches the player how switches function. Players of the control level can trigger the switch without seeing the door which may leave them confused.

The intuitive level has a switch in a pit and a switch above ground level. These both serve as antepieces to teach the player they can shoot in the air. They must use the double jump to clear the first one and a jump and shoot forwards to trigger the second. They are then tested in the final challenge to use the double jump to both clear the pit and trigger the final switch. This is transformative learning, showing the player that what they've learned about double jumping and triggering switches can be combined. This aims to teach the player the fundamental concept of the game, that shooting in the air is both an attack and a movement ability.

Intuitive



Control



Section 4

This section introduces enemies to the player.

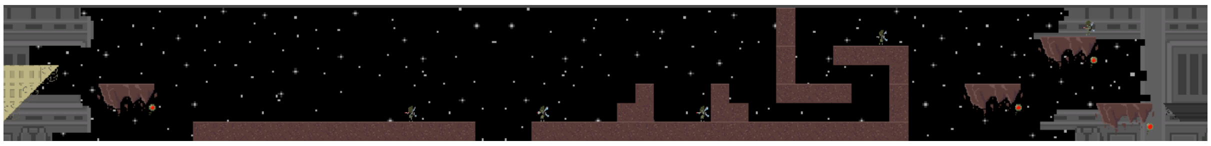
The goal of the intuitive level is to teach the player several things about enemies:

- Enemies die when shot
- Enemies die if they fall out of the level
- Enemies turn when colliding with a wall
- Enemies fall from the end of platforms and continue where they land

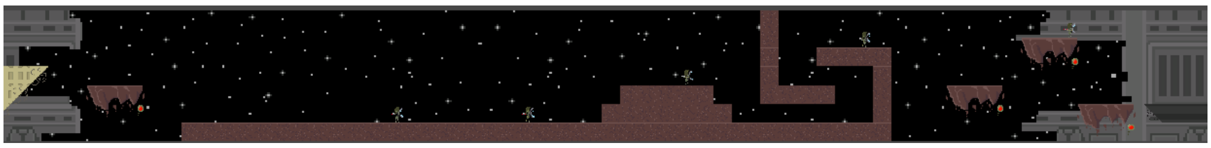
The intuitive level introduces these mechanics one at a time, using iterative learning.

The control level removes some of the situations that allow these mechanics to be demonstrated, leaving the player to guess them for themselves later in the level.

Intuitive



Control



Section 5

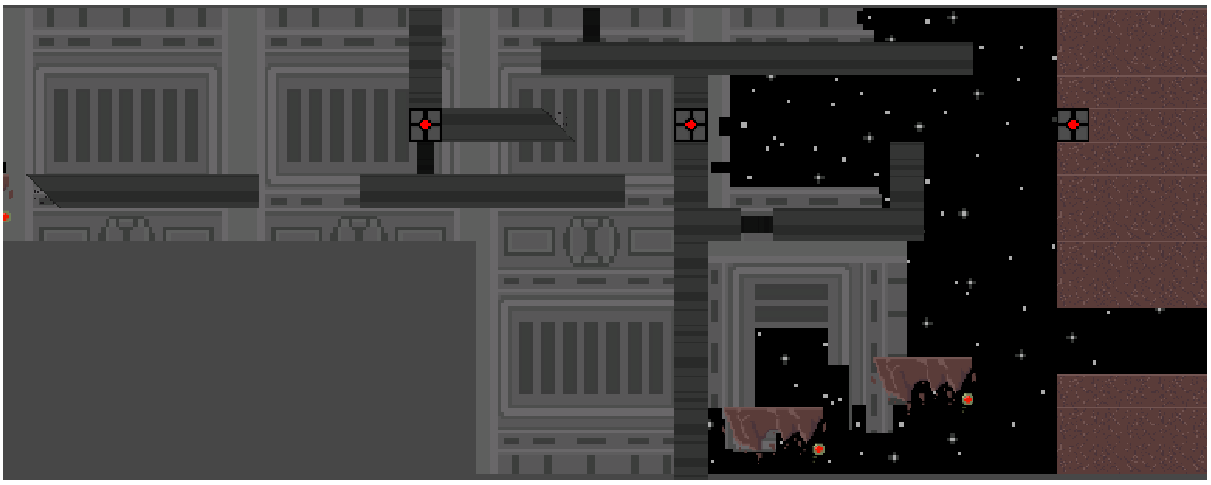
This section calls for the user to use the aerial shoot attack as a movement ability to clear some more complex jumping challenges.

This first switch in this section aims to teach the player about the knockback effect through productive negativity. If the player jumps across the pit, then jumps and shoots the switch they will be knocked back into the pit and fail. This failure teaches them that the knockback can have negative consequences.

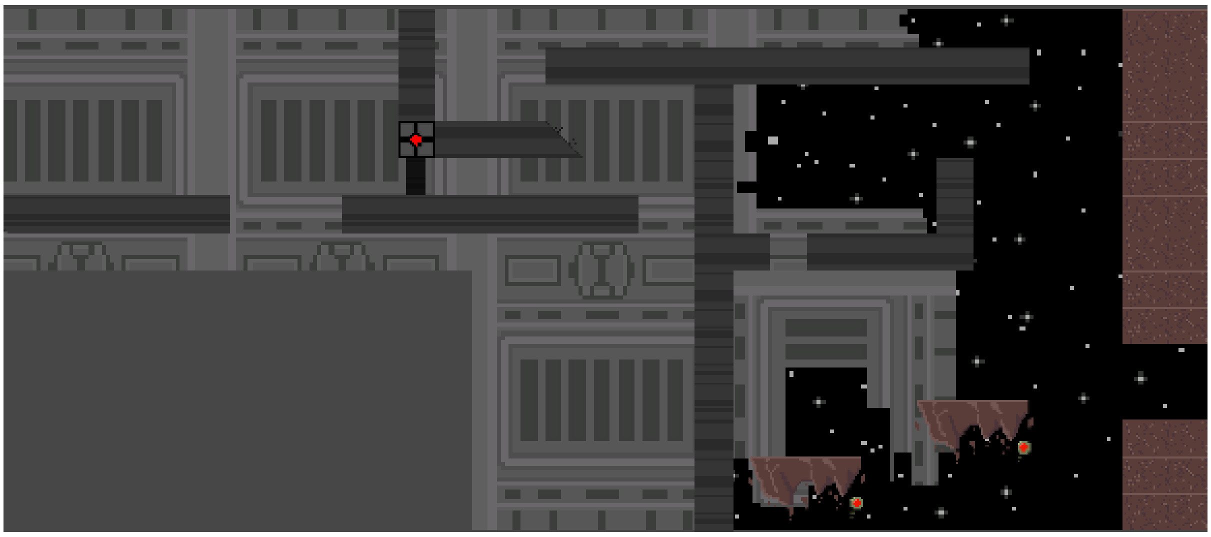
The next two challenges then encourage the player to use the knockback effect to reach places they otherwise couldn't get to. This uses transformative learning to show the player that this knockback can also be positive.

The control level lacks the second two switches which removes the encouragement for the player to use the jumping shoot. Instead, the player can choose between using jumping shoot or double jump to clear this section. Using the double jump is more difficult but the players may use it because they have a better understanding of this mechanic as they haven't been taught about the jumping shoot.

Intuitive



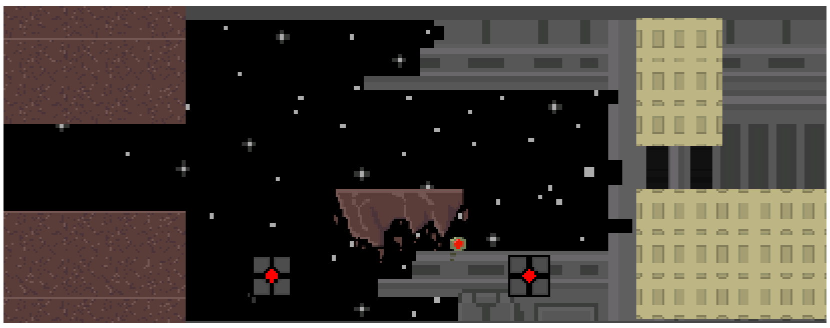
Control



Section 6

This short section just requires the player to use the double jump mechanic as a movement ability and to trigger a switch simultaneously.

Identical

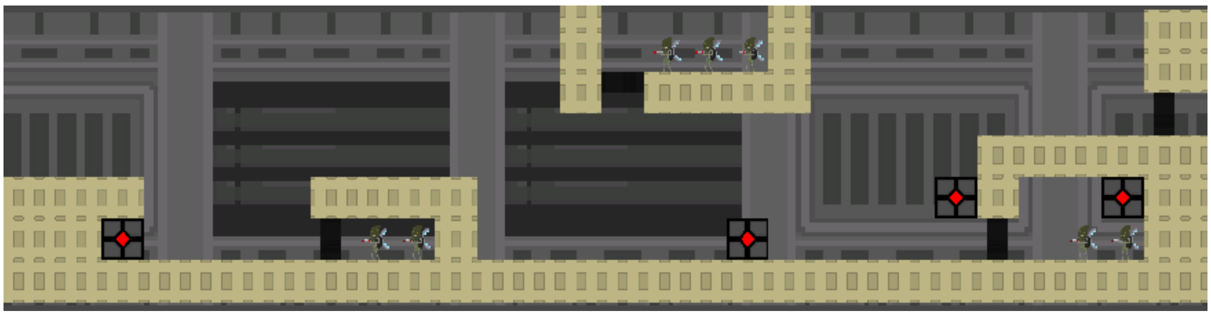


Section 7

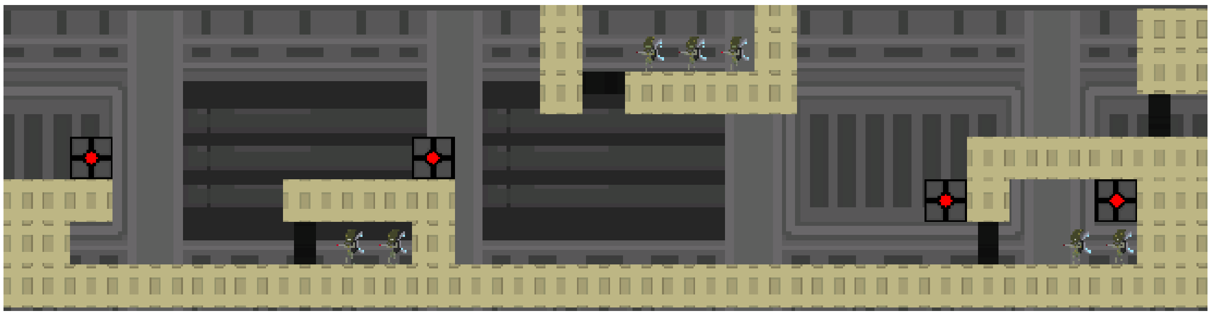
This section presents optional enemies to the player. The intuitive level attempts to show the player a switch and a door with captive enemies to deter them from opening it. They should then hesitate before triggering the next switch which will release enemies from above. For the final challenge they must then trigger the switch to release the enemies so that they can reach the switch to open the doorway. This is transformative learning to show the player that although releasing enemies is negative it is required sometimes.

The control level places the switches before the player has an idea of what they will trigger, meaning most players will trigger them immediately and not understand what they have done.

Intuitive



Control



Section 8

The goal of this section is to introduce a new enemy type. These armoured enemies are invulnerable to attacks from the front so must be defeated from above or behind.

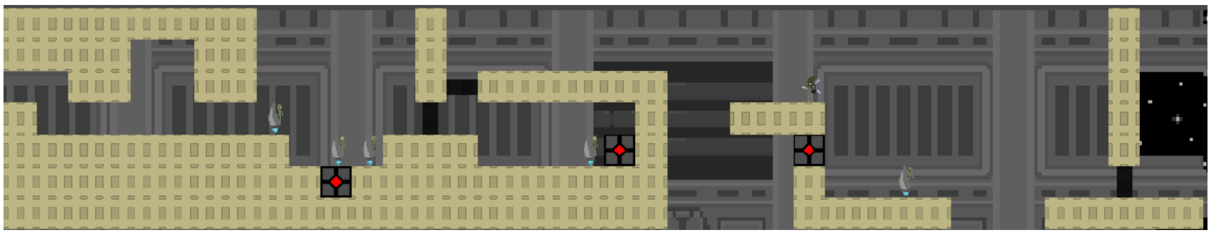
Most players first reaction will be to drop down and shoot the enemy. If they do so the enemy will not die and they will defeat the player. Here productive negativity is used to teach the player about the enemies' defences. Transformative learning is also being used to show the player that unlike the enemies they've encountered so far these ones are not vulnerable to all attacks.

The intuitive level restricts space for the player to jump so that they must jump over the enemy in the middle. To safely clear the enemy, they will either have to wait for the enemy to turn around or double jump over them. If they double jump the downwards projectile will most likely defeat the enemy showing that they are vulnerable from above.

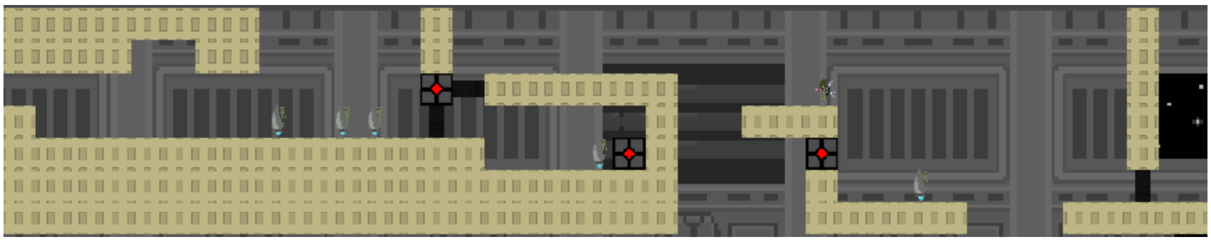
In case the player did not jump over the enemy and defeat them from above, the next switch requires the player to trigger it from above. It is unlikely the player will be able to do this without defeating at least one of the two enemies by accident showing they are vulnerable from above. The third challenge has limited space vertically making it difficult for the player to jump over the enemy. This requires them to shoot the enemy from behind instead. These three challenges in combination require the player to both defeat the enemies from above and behind. This section uses reinforcement learning and productive negativity to allow the player to experiment with the armoured enemies, while requiring each mechanic be used at least once to clear it.

The control level does not use reinforcement learning in the same way and doesn't provide the same guidance to ensure the player uses each mechanic.

Intuitive



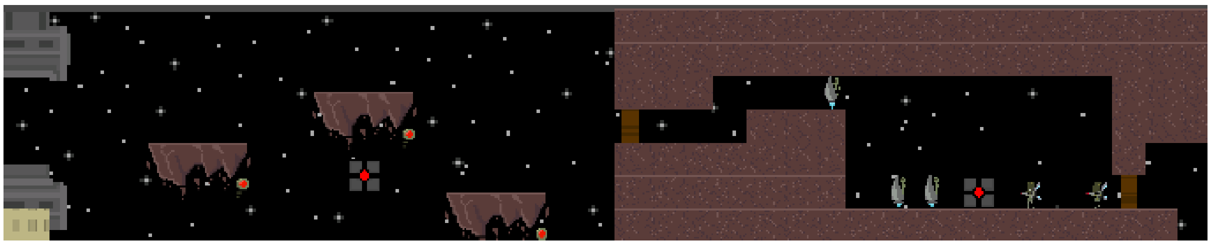
Control



Section 9

Now that every mechanic has been introduced this section tests the player's knowledge of the enemy types and interactions with switches. The final challenge of this section allows the player to trigger the switch which will let the enemies pass through the door and fall off the edge.

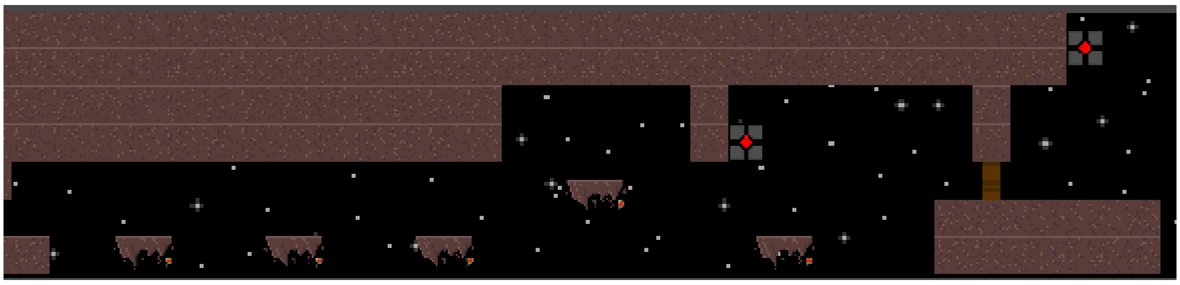
Identical



Section 10

This section tests the players movement abilities. Requiring more complex use of double jumping. The final switch can be triggered with a jumping shoot which will also knock the player back to the ledge they are trying to reach.

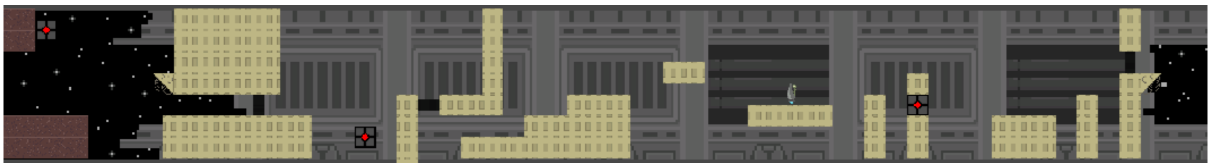
Identical



Section 11

The first few challenges of this section are the most difficult in the level so far. They require use of both jumping shoot mechanics to clear.

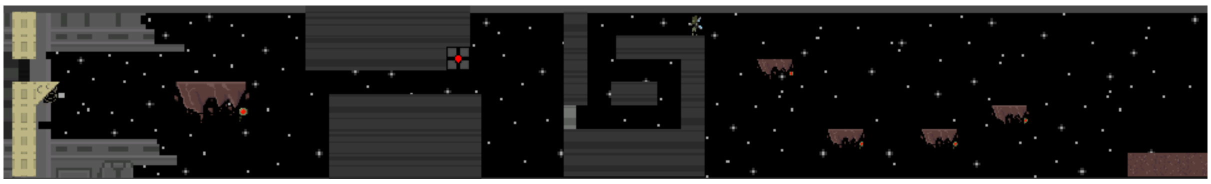
Identical



Section 12

This section requires the use of the knockback from the jumping shoot for the first jump. This was place at the start of the section so that it acted as a barrier until players realised they needed to use this mechanic. They must then use double jump late in the air to get into the short tunnel. The switch and door combo in the middle section require the use of a single jumping shoot to hit the switch and clear the gap.

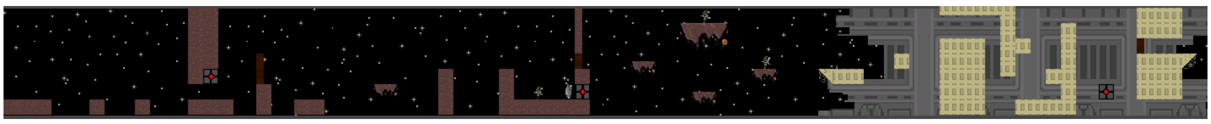
Identical



Section 13

This second last section is the most difficult in the game. It requires knowledge of every mechanic in the game to pass and is quite mechanically challenging. The players must understand the character mechanics, enemies, platforming mechanics and switches to pass the section.

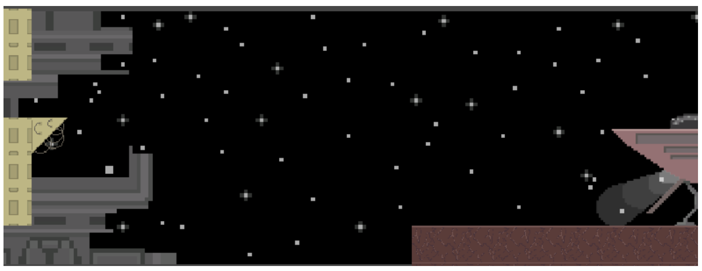
Identical



Section 14

This final section has a very long jump to reach the platform which marks the end of the game. This jump is so far that the target platform is not on the screen from where the character must jump from. Players must take a leap of faith here to discover where the platform is. This is why a checkpoint was placed right before this jump. Once discovering the platform the players should realise they need to use the knockback from the jumping shoot to reach the final platform and complete the game.

Identical



7.9 Appendix I – Initial Risk Assessment

A risk assessment was carried out during the research stage of this study. Due to the mitigation methods used these did not affect the implementation and the study was completed on time.

Table 20: The risks and their corresponding mitigation strategies for this study

Risk	**Severity**	**Probability**	**Mitigation**	**Severity after Mitigation**	**Probability**
Unity licensing changes.	H	L	Unity has been free for personal use since 2009 (Helgason, 2009) and has not announced this will change. No elements of game design should be dependant on Unity specific features so that it could be easily ported to another engine.	M	L
Game is buggy and bugs interfere with testing results.	M	M	Game should be frequently tested during development to identify bugs. These should be fixed whenever possible or mitigated somehow. Worst case players should be informed so they know what to expect.	M	L
Main mechanic is difficult to understand and interferes with level testing.	H	M	Main mechanic should be developed early and iteratively to refine it.	H	L
Hardware failure causes loss of study work e.g. game prototype and level designs.	H	M	Work for this study should be backed up to a cloud service like Github.	H	L
Hardware failure causes loss of test data.	H	M	Test data must be backed up. Care must be taken when backing up test data to make sure that it is still secure.	H	L
Unity does not provide the correct functionality to create the game prototype as intended	M	M	Unity features have been researched in advance to ensure it should be able to support the intended design. Concept prototype was created to test basic functionality/	M	L
Cannot find enough participants to test	H	M	Recruiting will begin at the start of the task. Recruit more participants than required if possible. Leave	H	L

game, result in insufficient data			extra time for testing so that can work around people's schedules		
Underestimated prototype development time	M	M	Prototype has been planned and split it into sub-tasks. Lengths for each sub-task have been estimated and structured with a Gantt char	L	L
Game prototype has poor performance and this affects testing results	M	M	Game should be tested during development on machine that will be used for test participants. Only one machine will be used for collecting data so that performance is always identical	L	L
Recruited participants are unavailable when the time comes for testing.	M	M	Participants will be recruited early and informed of the testing dates. A booking schedule for participants should be set up to ensure everyone fits in. Extra-time should be allowed for testing in plan to account for unforeseen circumstances	L	L

7.10 Appendix J - Concept Prototype

During the research for this dissertation a small test prototype was created in Unity to allow players to experiment with the core mechanic. This lacked some of the secondary mechanics, but the main goal of this prototype was to prove the core mechanic. Free to use assets where used for the sprite art and sound effects and these are not representative of the premise or final game.

The mechanics included in the concept prototype were:

- Player moving and jumping.
- Player shooting projectiles forwards and upwards from standing.
- Player shooting projectile forwards and upwards in air.
- Player shooting projectile downwards in air to double jump.
- Enemies that move left and right and defeat player. No invulnerabilities to any direction.
- A triggerable switch connected to a platform that will appear and disappear.

The concept game could be played with keyboard or controller and was built for Windows. Rather than progressing through a level the player was constrained to one screen and simply had to defeat as many of the endless enemies as possible. A screenshot of the concept prototype is shown in Figure 42.

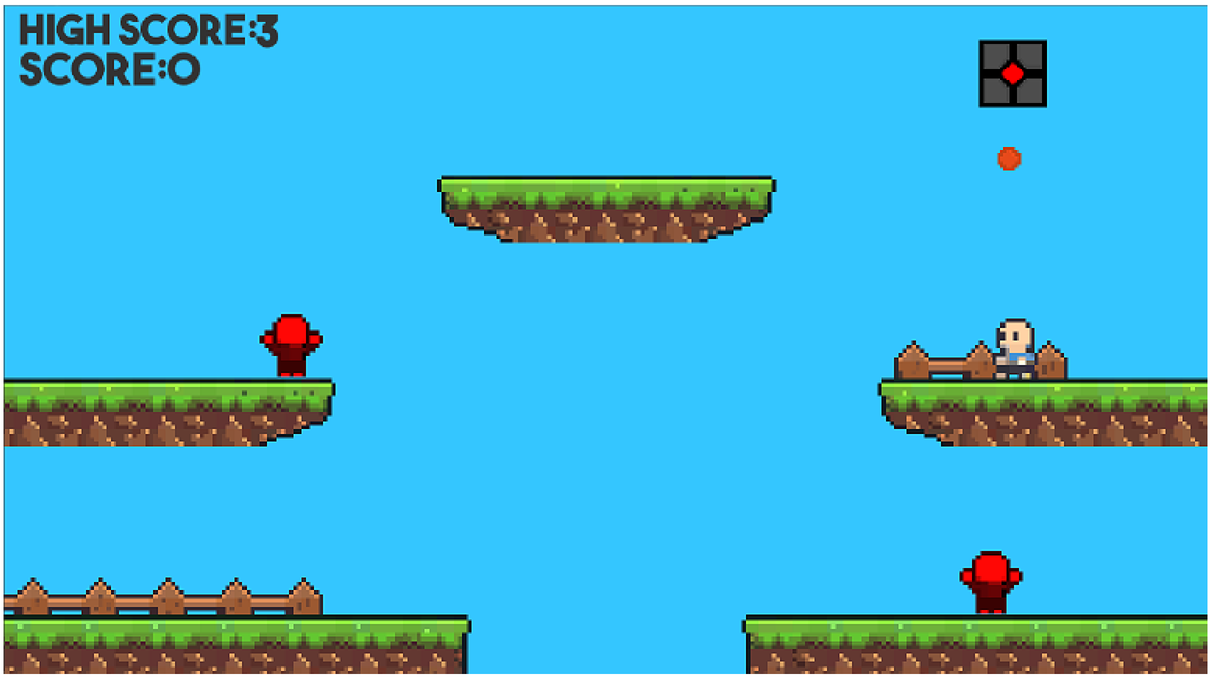


Figure 42: The concept prototype that was tested by the focus group. On the right the player can be seen shooting a projectile upwards towards a switch. The red humanoids are enemies.

7.11 Appendix K – Example Log File

```
Starting new run

Level : Control

Section 1 Complete      Time required 12.06523      Deaths 0

Section 2 Complete      Time required 15.56411      Deaths 0

Section 3 Complete      Time required 24.89559      Deaths 0

Section 4 Complete      Time required 294.0755      Deaths 18

Section 5 Complete      Time required 34.60904      Deaths 2

Section 6 Complete      Time required 14.19947      Deaths 1

Section 7 Complete      Time required 31.37589      Deaths 1

Section 8 Complete      Time required 98.78041      Deaths 14

Section 9 Complete      Time required 29.64472      Deaths 1

Section 10 Complete     Time required 89.15054      Deaths 20

Section 11 Complete     Time required 100.0455      Deaths 11

Section 12 Complete     Time required 116.9438      Deaths 39

Section 13 Complete     Time required 419.752       Deaths 30

Section 14 Complete     Time required 7.247428      Deaths 0

Run Completed           Total Time required 1288.349   Total Deaths 137
```

7.12 Appendix L – Raw Result Data

Intuitive	Section																													
	1		2		3		4		5		6		7		8		9		10		11		12		13		14		Total	
Subject	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths
1	17.8	0	21.2	0	25.5	0	64	4	82	6	6.9	0	94.6	3	43.8	0	104	6	59.3	11	65.9	2	111.3	22	197.1	15	3.9	12	929.2	81
3	7.9	0	18.1	0	60	1	231.8	13	102.8	6	33.6	5	49.1	1	160.6	8	36.4	1	52	11	93.5	4	41.6	2	592.3	50	13.8	2	1493.8	104
4	9.8	0	11.6	0	14.7	0	93.7	6	48.1	2	3.9	0	27	0	42.5	2	41.3	4	20.5	1	57.7	4	31.9	2	163.2	9	10.4	2	576.4	32
5	9.4	0	11.9	0	13.1	0	90.8	11	19.5	0	9.75	2	20.7	0	62	5	85.9	13	9.6	0	98.8	12	81.6	41	960.1	186	27.7	14	1500.7	284
10	11.6	0	14.1	0	22.5	0	193.8	11	26.5	0	5.7	0	42.7	1	37.9	1	70.7	5	9	0	37	1	20.2	0	152.7	9	9.3	4	653.8	32
626	22.6	0	22.9	0	62.1	0	510.9	52	65.5	2	42	8	29.6	0	155.4	9	25.2	0	54	16	51.8	1	192.9	38	334.8	15	17.4	5	1590.2	146
Averages	**13.2**	**0.0**	**16.6**	**0.0**	**33.0**	**0.2**	**197.5**	**16.2**	**57.4**	**2.7**	**17.0**	**2.5**	**44.0**	**0.8**	**83.7**	**4.2**	**60.6**	**4.8**	**34.1**	**6.5**	**67.5**	**4.0**	**79.9**	**17.5**	**400.0**	**47.3**	**13.8**	**6.5**	**1124.0**	**113.2**
STDEV	5.76	0.00	4.83	0.00	22.24	0.41	166.97	17.88	32.28	2.73	16.46	3.33	26.93	1.17	58.16	3.76	31.10	4.62	23.52	7.01	24.21	4.15	64.95	18.87	320.42	69.66	8.20	5.21	459.33	94.40
SEM	2.35	0.00	1.97	0.00	9.08	0.17	68.16	7.30	13.18	1.12	6.72	1.36	11.00	0.48	23.74	1.54	12.70	1.89	9.60	2.86	9.88	1.69	26.52	7.70	130.81	28.44	3.35	2.13	187.52	38.54

How difficult to understand	How difficult?	How enjoyable?	How engaging?	Understand?
10	6	10	9	1
10	5	9	10	1
10	6	8	8	1
8	5	7	8	1
10	5	10	10	1
8	5	7	6	1
9.3	**5.3**	**8.5**	**8.5**	**1.0**
1.03	0.52	1.38	1.52	N/A
0.42	0.21	0.56	0.62	N/A

Control	Section																													
	1		2		3		4		5		6		7		8		9		10		11		12		13		14		Total	
Subject	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths	Time	Deaths
2	17.5	0	13.8	0	34.1	1	161.2	12	40.5	2	14.6	2	25.6	0	27.9	0	74.5	14	62.7	13	22.2	0	35.1	8	143	7	26	13	698.7	72
7	23.4	0	20.2	0	13.8	0	45.6	2	28	2	3.9	0	32.3	1	45.2	8	69.2	9	31	6	31.6	1	14.5	0	224.8	18	9.8	3	593.3	50
8	12.1	0	15.6	0	24.9	0	294.1	18	34.6	2	14.2	1	31.4	1	98.8	14	29.6	1	89.2	20	100	11	116.9	39	419.8	30	7.2	0	1288.3	137
404	8.7	0	64.9	0	33.4	0	101.8	3	24.4	0	5.3	0	36.1	2	65.1	5	41.7	3	18.9	4	29.1	0	42.9	10	105.1	8	3.3	0	580.6	35
11	42.6	0	54.8	2	90.8	3	279.5	21	49.3	6	8.2	0	36	0	64.6	2	339.9	50	86.2	20	198.7	16	293.9	89	401.3	55	3.2	0	1948.9	264
13	12.1	0	26.8	0	21	0	163	7	37.7	0	12	2	53.4	2	39.2	0	32.9	0	59.7	11	121.6	5	40.9	1	221.4	9	12.9	1	854.6	38
Averages	**19.4**	**0**	**32.6833333**	**0.3333333**	**36.333333**	**0.6666667**	**174.2**	**10.5**	**35.75**	**2**	**9.7**	**0.8333333**	**35.8**	**1**	**56.8**	**4.8333333**	**97.9666667**	**12.833333**	**57.95**	**12.333333**	**83.8666667**	**5.5**	**90.7**	**24.5**	**252.5666667**	**21.1666667**	**10.4**	**2.8333333**	**994.0666667**	**99.3333333**
STDEV	12.48	0.00	21.75	0.82	27.76	1.21	97.51	7.87	8.95	2.19	4.58	0.98	9.44	0.89	25.19	5.46	119.99	18.97	28.47	6.77	69.86	6.66	105.49	34.66	130.83	18.73	8.51	5.12	536.01	89.01
SEM	5.10	0.00	8.88	0.33	11.33	0.49	39.81	3.21	3.65	0.89	1.87	0.40	3.85	0.37	10.28	2.23	48.98	7.74	11.62	2.76	28.52	2.72	43.07	14.15	53.41	7.65	3.48	2.09	218.83	36.34

How difficult to understand	How difficult?	How enjoyable?	How engaging?	Understand?
9	6	9	10	1
9	4	5	6	1
10	8	8	9	1
6	4	5	6	1
8	4	8	5	1
10	5	8	8	1
8.7	**5.2**	**7.2**	**7.3**	**1.0**
1.51	1.60	1.72	1.97	N/A
0.61	0.65	0.70	0.80	N/A

8 References

Abe, H., Seo, H. & Lee, D., 2011. The prefrontal cortex and hybrid learning during iterative competitive games. *Annals of the New York Academy of Sciences,* 1239(1).

Anthropy, A. & Clark, N., 2014. *A Game Design Vocabulary.* 1 ed. Crawfordsville: Addison Wesley.

astral, l., 2014. *That time when you incinerate the companion cube.* [Online]
Available at: https://www.youtube.com/watch?v=rq1EVrduBDI
[Accessed 25 03 2019].

Brown, M., 2015. *Half-Life 2's Invisible Tutorial | Game Maker's Toolkit.* [Online]
Available at: https://www.youtube.com/watch?v=MMggqenxuZc
[Accessed 10 02 2019].

Brown, M., 2017. *The Design Behind Super Mario Odyssey | Game Maker's Toolkit.* [Online]
Available at: https://www.youtube.com/watch?v=z_KVEjhT4wQ
[Accessed 22 02 2019].

Brown, M., 2018. *The World Design of Metroid 1 and Zero Mission.* [Online]
Available at: https://www.youtube.com/watch?v=kUT60DKaEGc
[Accessed 02 2019].

Bruns, R. N., 2008. *Super Mario Brothers - World 1-1 Labeled Map.* [Online]
Available at: http://www.nesmaps.com/maps/SuperMarioBrothers/SuperMarioBrosWorld1-1Map.html
[Accessed 08 03 2019].

Burrows, J., Jackson, W. & Rideg, M., 2019. Unity versus Unreal: Choosing your Game Engine. *3D World,* Issue 242, p. 86.

Cavanagh, T., 2010. *VVVVV on Steam.* [Online]
Available at: https://store.steampowered.com/app/70300/VVVVVV/
[Accessed 26 03 2019].

Cavanagh, T., 2010. *VVVVV. Game [Microsoft Windows].* London: Terry Cavanagh.

Centaur Communications Ltd, 2006. GAMES: Use your intuition. *Design Week,* 21(23), p. 16.

Chen, J., 2007. Flow in Games & Everything Else. *Communications of the ACM,* 50(4), pp. 31-34.

Christopoulou, E. & Xinogalos, S., 2017. Overview and Comparative Analysis of Game Engines for Desktop and Mobile Devices. *International Journal of Serious Games,* 4(4).

Csikszentmihalyi, M., 2014. *Learning, "Flow," and Happiness.* 3 ed. Claremont, USA: Springer, Dordrecht.

De Nucci, E. & Kramarzewski, A., 2018. *Practical Game Design.* s.l.:Packt Publishing.

Desurvire, H. & Wilberg, C., 2008. *Master of the game: assessing approachability in future game design.* Florence, Italy, CHI '08 Extended Abstracts on Human Factors in Computing Systems.

Desurvire, H. & Wixon, D., 2013. *Game Principles: Choice, Change & Creativity: Making Better Games.* New York, CHI '13 Extended Abstracts on Human Factors in Computing Systems.

Finneran, C. M. & Zhang, P., 2003. A person–artefact–task (PAT) model of flow antecedents in computer-mediated environments. *International Journal of Human-Computer Studies,* 59(4), pp. 475-496.

Fischer, F., 2016. *Why We Need Challenge.* [Online]
Available at:
http://www.gamasutra.com/blogs/FabianFischer/20160224/266405/Why_We_Need_Challenge.php
[Accessed 02 03 2019].

Fullerton, T., 2008. *Game Design Workshop.* 2 ed. Burlington: Morgan Kaufmann.

GameDesigning.org, 2019. *The Top 10 Video Game Engines.* [Online]
Available at: https://www.gamedesigning.org/career/video-game-engines/
[Accessed 19 02 2019].

Gauthier, A. & Jenkinson, J., 2018. Designing productively negative experiences with serious game mechanics: Qualitative analysis of game-play and game design in a randomized trial. *Computers & Education,* Volume 127, pp. 66-89.

git, 2019. *About.* [Online]
Available at: https://git-scm.com/about/distributed
[Accessed 15 07 2019].

Gregory, J., 1970. *Game Engine Architecture.* 2 ed. Boca Raton, Florida: CRC Press.

Grow, B., 2014. *Good Game Design - Shovel Knight: Teaching Without Teaching.* [Online]
Available at: https://www.youtube.com/watch?v=cYvPdEyTXUc&t=1s
[Accessed 10 02 2019].

Helgason, D., 2009. *A free Unity?.* [Online]
Available at: https://blogs.unity3d.com/2009/10/29/a-free-unity/
[Accessed 31 03 2019].

Hoffman, B. & Nadelson, L., 2010. Motivational engagement and video gaming: a mixed methods study. *Educational Technology Research and Development,* 58(3), pp. 245-270.

Intelligent Systems, 2003. *Fire Emblem: The Blazing Blade. Game[Game Boy Advance].* Kyoto: Nintendo.

Iyer, A., 2017. *Analyzing Super Mario's level and tutorial design.* [Online]
Available at: https://medium.com/@abhishekiyer_25378/the-perfect-game-tutorial-analyzing-super-marios-level-design-92f08c28bdf7
[Accessed 20 03 2019].

JapanCommercials4U2, 2012. *Super Mario 3D Land Playthrough Part 1.* [Online]
Available at: https://www.youtube.com/watch?v=wdKnUqaHSmc
[Accessed 17 03 2019].

Kantopia, 2018. *FE7 Blazing Blade Localization: Karel's "Sword" – Physical Blade or a Technique? [JPN vs ENG].* [Online]
Available at: https://kantopia.wordpress.com/2018/06/07/fe7-blazing-blade-localization-karels-sword-physical-blade-or-a-technique-jpn-vs-eng/
[Accessed 02 04 2019].

Kayali, F. & Schuh, J., 2011. *Retro Evolved: Level Design Practice exemplified by the Contemporary Retro Game.* Hilversum, The Netherlands, DiGRA 2011 Conference: Think Design Play.

Kiili, K., 2005. Digital game-based learning: Towards an experiential gaming model. *The Internet and Higher Education,* 8(1), pp. 13-24.

Kim, C., 2012. *Designing around a Core Mechanic.* [Online]
Available at:
https://www.gamasutra.com/blogs/CharmieKim/20120612/172238/Designing_around_a_core_mechanic.php
[Accessed 20 02 2019].

Kolb, D. A., 2014. *Experiential Learning.* 2 ed. New Jersey: Pearson Education Inc..

Kubovy, M. & Pomerantz, J. R., 2017. *Perceptual Organisation.* 1 ed. London: Routledge.

Microsoft, 2015. *Auto-Implemented Properties (C# Programming Guide).* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/auto-implemented-properties
[Accessed 12 08 2019].

Mitgutsch, K. & Schirra, S., 2012. *Subversive Game Design and Meaningful Conflict.* [Online]
Available at: http://gamelab.mit.edu/research/subversive/
[Accessed 05 02 2019].

Napierala, M. A., 2012. What Is the Bonferroni Correction?. *AAOS Now,* Issue April.

NES Maps, 2008. *Super Mario Brothers - World 1-1.* [Online]
Available at: http://www.nesmaps.com/maps/SuperMarioBrothers/SuperMarioBrosWorld1-1Map.html
[Accessed 28 02 2019].

NES Maps, 2013. *Metroid - All of Planet Zebes.* [Online]
Available at: http://www.nesmaps.com/maps/Metroid/MetroidCompleteMap.html
[Accessed 08 03 2019].

Nintendo Creative Department, 1985. *Super Mario Bros. Game [NES].* Kyoto: Nintendo.

Nintendo EAD Tokyo, 2011. *Super Mario 3D Land. Game [Nintendo 3DS].* Tokyo: Nintendo.

Nintendo EPD, 2017. *Super Mario Odyssey. Game [Nintendo Switch].* Kyoto: Nintendo.

Nintendo R&D1 and Intelligent Systems, 1983. *Metroid. Game[NES].* Kyoto: Nintendo.

Nutt, C., 2012. *The Structure of Fun: Learning from Super Mario 3D Land's Director.* [Online]
Available at:
http://www.gamasutra.com/view/feature/168460/the_structure_of_fun_learning_.php?page=4
[Accessed 20 02 2019].

Nystrom, R., 2014. *Game Programming Patterns.* 1 ed. Seattle, WA: Genever Benning.

Podleschny, N., 2012. *Games for change and transformative learning: An ethnographic case study,* Queensland University of Technology: Queensland University of Technology.

Ren'Py, 2019. *What is Ren'Py?.* [Online]
Available at: https://www.renpy.org/
[Accessed 10 03 2019].

RPS, 2013. *Untold Riches: An Analysis Of Portal's Level Design.* [Online]
Available at: https://www.rockpapershotgun.com/2013/09/20/untold-riches-an-analysis-of-portals-expressive-level-design/
[Accessed 14 03 2019].

Semmle, 2019. *Double-Checked Lock is not thread-safe.* [Online]
Available at: https://help.semmle.com/wiki/display/CSHARP/Double-checked+lock+is+not+thread-safe
[Accessed 22 07 2019].

Shirinian, A., 2013. Intuitions, Expectations and Culture. *Game Developer,* p. n/a.

Steam, 2007. *Portal.* [Online]
Available at: https://store.steampowered.com/app/400/Portal/
[Accessed 07 03 2019].

Taito, 1978. *Space Invaders. Game [Arcade Cabinet].* Tokyo: Taito.

Team Meat, 2010. *Super Meat Boy. Game [Microsoft Windows].* Asheville: Team Meat.

tvtropes, 2019. *Antepiece.* [Online]
Available at: https://tvtropes.org/pmwiki/pmwiki.php/Main/Antepiece
[Accessed 0 2019].

tvtropes, 2019. *Antepiece.* [Online]
Available at: https://tvtropes.org/pmwiki/pmwiki.php/Main/Antepiece
[Accessed 16 03 2019].

Unity, 2018. *Unity Manual: The Game view.* [Online]
Available at: https://docs.unity3d.com/Manual/GameView.html
[Accessed 05 08 2019].

Unity, 2019. *The world's leading real-time creation platform.* [Online]
Available at: https://unity3d.com/unity?_ga=2.138526643.1471254040.1553439662-321221326.1538304610
[Accessed 24 03 2019].

Unity, 2019. *Unity Manual - Animator Component.* [Online]
Available at: https://docs.unity3d.com/Manual/class-Animator.html
[Accessed 26 07 2019].

Unity, 2019. *Unity Manual - Animator Controller.* [Online]
Available at: https://docs.unity3d.com/Manual/class-AnimatorController.html
[Accessed 28 07 2019].

Unity, 2019. *Unity Manual - Colliders.* [Online]
Available at: https://docs.unity3d.com/Manual/CollidersOverview.html
[Accessed 28 07 2019].

Unity, 2019. *Unity Manual - Rigidbody Overview.* [Online]
Available at: https://docs.unity3d.com/Manual/RigidbodiesOverview.html
[Accessed 28 07 2019].

Unity, 2019. *Unity Manual - Sprites.* [Online]
Available at: https://docs.unity3d.com/Manual/Sprites.html
[Accessed 21 07 2019].

Unity, 2019. *Unity Personal.* [Online]
Available at: https://store.unity.com/products/unity-personal
[Accessed 25 03 2019].

Unity, 2019. *Unity Scripting API - Component.GetComponentInChildren.* [Online]
Available at: https://docs.unity3d.com/ScriptReference/Component.GetComponentInChildren.html
[Accessed 18 07 2019].

Unity, 2019. *Unity Scripting API - ForceMode2D.Impulse.* [Online]
Available at: https://docs.unity3d.com/ScriptReference/ForceMode.Impulse.html
[Accessed 28 07 2019].

Unity, 2019. *Unity Scripting API - Physics2D.* [Online]
Available at: https://docs.unity3d.com/ScriptReference/Physics2D.html
[Accessed 28 07 2019].

Valve Corporation, 2004. *Half-Life 2. Game [Microsoft Windows].* Belevue, Washington: Valve Corporation.

Valve Corporation, 2007. *Portal. Game [Microsoft Windows].* Bellevue, Washington: Valve Corporation.

Valve, 2010. *Super Meat Boy - Steam.* [Online]
Available at: https://store.steampowered.com/app/40800/Super_Meat_Boy/
[Accessed 13 03 2019].

Webster, J., Trevino, L. K. & Ryan, L., 1993. The dimensionality and correlates of flow in human-computer interactions. *Computers in Human Behaviour,* 9(4), pp. 411-426.