Final Year Project Report

Full Unit – Final Report

Building a Game: Practice Makes Painful

Mycal Latchman

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Matteo Sammartino

Department of Computer Science

Royal Holloway, University of London

November 24, 2022

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count:

12,731

Student Name:

Mycal Latchman

Date of Submission:

23/03/23

Signature:

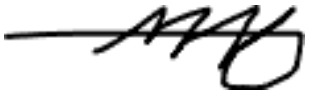


Table of Contents

Abstract

Every form of media has its own genres and games are no different. Ranging from intense shooters to relaxing puzzle games. One such genre that has become very prominent in the games industry is fighting games, which are also known as “fighters”. Fighters typically involve two or more players battling it out against each other whilst utilising a roster of characters. One of the first commercially successful fighters “Street Fighter” was released 35 years ago, marking the solidification of specific characteristics that have become staple features of fighters, separating them from other genres. Examples of this include characters with unique “movesets” (player actions), input-combination-based special attacks, and the player versus player (PvP) format. Due to the innovations and variations in mechanics, modern fighting games have developed into an incredibly distinct subsection of the gaming world with a dedicated fanbase. In this project, I intend to explore what core features are required to create a fighting game and how they are implemented. This will allow me to expand my knowledge on the topic and prove that I can develop the new skills needed to accomplish this, such as learning the C# language and how to utilise the Unity game engine to its full extent.

Project Specification

Aims: To understand the game development ecosystem and to build and deploy a playable game.

Background:

Games development is one of the largest parts of the computing industry. There are many platforms for writing games and indeed, many architectures and frameworks.

In this project, you will build and deploy a game.

In the first term, you will research existing games and their underlying technology platforms. You will also look at game building tools such as Unity3D. You will consider game frameworks such as PyGame and investigate APIs for game development such as Vulkan.

You will need to be familiar with the design patterns required for gaming: Flyweight, publish/subscribe (for message passing), Object Pool, Service, Dirty Flag, Factory, Visitor, Singleton, Game Loop, State, Observer, Command, Event Queue (See perhaps Game Programming Patterns by Robert Nystrom. Also, consider reading some of the material on The Game Designs Patterns Wiki

Your target could be a tower defence game written in Python in which case you will need to understand Threading and Graphics in Python.

Perhaps you will use Blender and other tools and do some maths and produce an excellent Unity game.

Maybe you will program a platform game on an Android device, using the sensors and learning about the Android life cycle, Manifesto, and the mobile approach to gaming.

1.1.1.1 Early Deliverables

1. A simple program displaying an animation using sprites or similar animation technology.

2. A program displaying a simple, interactive user interface. You might want to explore the different means of displaying and writing user interfaces (XML, Object Orientated, Immediate Mode).

3. 3D demo using an API

4. A program that exhibits some 3D graphics using a relatively low-level graphics API (Vulkan, OpenGL, WebGL, JavaFX etc. This might provide a starting point for choosing technology: https://en.wikipedia.org/wiki/List_of_3D_graphics_libraries)

5. A simple game using a game engine / library / framework. Think of it as an opportunity to learn about the technologies you'll use for your main game but keep it simple.

6. A report on design patterns in games, an enumeration of commons ones, the problems they solve and perhaps discuss some specific instances of their use.

7. A report on the specific technologies you've tried / intend to use for your game. You should do deep on you intended tech, show as deep an understanding as you can.

8. A report on animation techniques, enumerate them, perhaps tell a history of them, compare and contrast. The goal is to show a deep understanding of how animation in games is achieved.

9. A report on Threading a User Interfaces. What problems do games experience with user interfaces without threading, how does multi-threading solve these problems and what different ways can this parallelism or concurrency be achieved? Perhaps review how it has actually been achieved in technologies or platform relevant to your game.

10. A report on designing games with graphical user interfaces for multiple platforms.

11. A report on the technologies used to write and draw graphical user interfaces.

1.1.1.1 Final Deliverables

1. Game Source Code: You should use a consistent architectural style. Our suggestion is object orientation, but you could opt for another with sufficient justification.

2. Game Source Code: The game play should make use of animation.

3. The Final Report: An in depth review of background materials you used to learn the concepts and technologies used to write the game, perhaps presenting a historical timeline of those concepts / technologies.

4. The Final Report: A description of the software engineering processes involved in writing your game.

5. The Final Report: A description of interesting techniques, hacks or data structures you used (or could have used) in writing your game.

Chapter 1: **Introduction**

For this project I will be endeavouring to create a 2D fighting game using the Unity engine. The Unity engine is more suited to beginners who are interested in game development compared to more complex ones such as the Unreal Engine.

The only experience I have had so far with making a game was during the Python Game Project in my first year which is why I have decided to use Unity. I feel that this game engine is a more appropriate challenge for me as a beginner, instead of alternatives such as Unreal Engine which is more daunting and complex.

Although my project in my First Year was not a fighting game, it did allow me to gain experience with game object movement and collision, features that are the basis of any good 2D fighting game. I plan for my game to include a multitude of features and it will also ideally support local multiplayer allowing two people to play against each other in real-time. With these objectives, I aim to create a product that can showcase my coding skills and my ability to make consistent progress over time. These skills will be considered strengths by any future employers.

I intend for my game to include certain characteristics that correlate with Roger Caillois' expansion on Huizinga's theories of what distinguishes an activity as play. I will particularly focus on ensuring my fighting game allows a respite from the monotony of the player's usual daily life, an idea that aligns with play being far removed from the daily routine of the consumer. Furthermore, I will strive to assure the player is engaged throughout gameplay by creating a game where there is no assured victory until the end so that the outcome is entirely reliant on the player's initiative. (Bean, 2018)

By creating this fighting game, I intend to expand my skill set to include being able to code in C# and understanding how to efficiently use the Unity Game engine. Accumulating these skills will help me in the future as it is very useful to be proficient in multiple programming languages. As I am particularly interested in the games industry, becoming familiar with the Unity engine will be even more beneficial for the beginning of my career, as it is often used by small game developers.

Examples of games that have inspired my decision to create a fighting game for my Final Year Project include Street Fighter, Dragon Ball FighterZ, and Mortal Kombat. I both grew up with and to this day still enjoy playing each of these games, which makes me believe that I have a strong idea of what constitutes a successful 2D fighting game and an enjoyable final product.

1.1 Aims and Objectives

The aim of this project is to create a 2D fighting game that could be seen as mechanically level with other modern games. In order to achieve this, I must meet the following objectives:

1: Become familiar with the Unity Engine and C#
2: Learn good practices for making games
    a: Key mechanics for fighting games
    b: Common bugs in fighting games that I should avoid
3: Perform User Acceptance Testing and receive feedback to make changes
4: Create and animate sprites

Chapter 2: **Background**

2.1 Game Research

In this section I will be covering research I have done on existing games. From this research, I aim to gain insight into what fundamental pieces must be present in fighting games. As well as features that are good to include, which compel players to return to the game. I will utilise this information to advise what I should implement into my game, in order to get the best results.

Karate Champ (1987): (Technōs Japan Corp.)

Karate Champ, released in May of 1987, is hailed as the first 2D fighter game. The game was very basic compared to more modern titles but laid the groundwork for them through trying out new things; For example, the game system for Karate Champ utilised two joysticks (a design choice that would later be abandoned in favour of button inputs in later 2D fighting games), the left one primarily used for movement inputs and the right for attacks. However, the left stick could also be used to block by pulling away from the opponent, a feature still used in many modern games such as Dragon Ball FighterZ.

Some attacks would require the use of both joysticks at once, for example, to perform a “Foot Sweep” attack you would have to pull both the left and right stick downwards, this allowed for more types of attacks to be implemented by incorporating the movement inputs as part of the attack (SBrainfreeze, 1999). This is a feature still used today, by

separating the movement and attack options to the left and right hand of the player respectively, it allows even a beginner to have an idea of how to play and simple inputs such as down on the movement side and a normal attack are easy inputs to learn.

Street Fighter II (1991): (CAPCOM)

For generations of 2D fighting game enthusiasts, Street Fighter II was their introduction to the genre, by expanding upon the gameplay of its predecessor “Street Fighter”, Street Fighter II was able to gain critical acclaim worldwide. Street Fighter II differed from Karate Champ in that it allowed for local multiplayer, meaning people could play head-to-head against their friends, or competitively, which is a key feature in its success. (Ketonen, 2016)

Furthermore, unlike Karate Champ, Street Fighter did not just have basic inputs for strike variations, but they also had new inputs for special moves, such as the well-known “Hadoken” or the “Shoryuken”, which involved movements known as a “Quarter Circle”. A quarter circle is a common input required for special moves in fighting games to this day, for example, the input for the Hadoken is a quarter circle forward combined with a punch. A quarter circle forward involves the movement input of down, forward-down, and forward; thus rotating through a quarter of a circle on the movement inputs.

I believe what kept bringing players back to Street Fighter II is the fact that it has an easy-to-understand game, with a very high skill ceiling, this means that it is accessible to beginner players whilst also being rewarding to players who put their time into practising features of the game by allowing them to improve.

Mortal Kombat 2 (1993): (Warner Bros Interactive Entertainment)

Mortal Kombat 2 is the second instalment in the Mortal Kombat series, a game that many consider a rival to the Street Fighter game series. This game proved that the main features of the last generation were solidified as core aspects of a 2D fighting game. For example, the Heads-Up Display (HUD) consists of two health bars displaying how much damage each player has received, as well as the timer between them, this allows each player to know off the bat which health bar is their own and which is their opponents as they never move from their respective starting position and the timer is in a position for both players to see easily.

Additionally, the “Best of Three” system was maintained in this instalment. This means that whichever player is first to win two matches will win the set overall. Due to this, the developers of Mortal Kombat decided to implement a round win counter under the health bar in the form of circles with the game’s logo in them. This is not to be confused with the game win counter, which is above the health bar, this counter would only increment if you won a best of three, adding to the competitive nature of 2D Fighters as it will make players want to increase their win counter.

2.2 Game Theory

The Nash Equilibrium:

In any game that exists where two people are put up against each other, there will be some degree of interaction. Both players are aiming to win, but there can be only one winner. This means that in each interaction there will be a Nash equilibrium, “one of the most basic concepts in game theory”. The concept of the Nash Equilibrium is about a “steady state” of a game, and not the events that lead to each state. In this steady state, each player is aware of all possible actions their opponent could take and acts based on this (Osborn and Rubinstein 27-42).

Nash Equilibrium could be applied to a fighting game as exemplified through the following scenario - imagine two players are not quite in range to attack each other. This is known as the “neutral” zone. Since they are not in range to attack each other, neither player has an advantage over the other. This leaves them with three options. Firstly, a player could “Move in” to range with their opponent, allowing them to attack. Secondly, a player could launch a “Preemptive” attack, by anticipating their opponent entering their attack range they can launch an attack to catch them on their way in before their opponent could strike. Finally, a player could look to make their opponent miss their initial attack (by waiting) and then “Punish” them while they are off-guard with an attack of their own.

Each of these approaches to the neutral range has its own risks and benefits, which can be shown well in the table below:

	Move in	Preemptive	Punishing
Move in	0, 0	-1, 1	1, -1
Preemptive	1, -1	0, 0	-1, 1
Punishing	-1, 1	1, -1	0, 0

This table displays the Nash Equilibrium through the chances a player will land or receive attack damage, +1 when landing a hit and -1 when receiving a hit. For example, if player 1 decides to move in for an attack, whilst player 2 intends to punish player 1 by anticipating him preempting a move and waiting for an attack before doing anything. Player 1 would take the advantage and land a strike on player 2 by moving in before attacking, thus landing a hit.

Situations, where both players adopt the same approach to the neutral zone, will result in neither player landing an attack. For example, if both players wait to punish one another for preempting an attack, neither will accomplish their goal because they will both be waiting for something that will not happen. If both players intend to preempt one anothers’ attacks, neither will enter the range to actually hit the other with their attack. Finally, if both players move in to attack, they will both hit each other, resulting in both dealing and receiving the same amount of damage and creating no benefit to either player. (Cornell University)

By researching this topic I have realised that a fighting game can be made more interesting by expanding the options of the Nash Equilibrium. In the example above there are only 3 actions that can occur when in neutral, but by introducing actions such as grabbing, blocking or allowing for long range attacks my game would have greater

possibilities compared to more basic games. Thereby enriching the experience for my players.

2.3 Unity

Unity is a game engine that I'm using to create my project. This is because it supports a multitude of features that come in handy when making things such as GameObjects, which are "fundamental objects in Unity" (Unity Technologies, 2022) these can be used as players and the environment (key components of fighting games). Additionally, GameObjects can act as containers for components, used to implement functionalities. For example, all GameObjects have a built-in "transform" component used to represent their position in the scene as well as the direction they are facing.

Rigidbody:

A component that will be useful in making my game is "RigidBody", due to my game being a 2D Fighter, I will be using RigidBody2D. Rigidbody2D is a component that gives control of the object's motion to Unity's built-in physics engine, as a result of this, the object will be pulled down by gravity automatically. Another thing the Rigidbody2D component allows is the application of forces on an object. This would allow for movement or knockback forces to impact the players in my game by applying a force upon the trigger of pressing the movement keys or the pressing of an attack key of a player when in range of their opponent; this could be achieved using the AddForce() method.

It is recommended that a FixedUpdate method is used to apply forces to and change the features of a Rigidbody, this is because updates made to the physics system are carried out at a different frequency than those in Update(). This means that the script in FixedUpdate is called before each physics update, allowing it to be processed directly at the correct rate. (Unity Technologies, 2022)

Collider:

A collider component defines the shape of an object that will be used to detect physical collisions, this can be used in conjunction with Rigidbody to apply a force on a GameObject upon the detection of a collision. For example, this could be used to detect when a player is hit and apply a force to them, creating a knockback effect. It could also be used to make sure some features could only be used when in a collision by using OnTriggerEnter() and OnTriggerExit(). This is useful for preventing a player from jumping in the air unlimited times by making it say the player can only use the jump feature when in contact with the ground. (Unity Technologies, 2017)

Tags and Layers:

Tags are words that can be used to reference GameObjects, I will use this to distinguish between player and environment objects I create. Then, in order to distinguish between Player1 and Player2 I will use layers, this will enable me to inflict effects such as damage and knockback to the GameObjects that I intend to, based on actions taken.

Design Patterns:

Design Patterns are solutions to common problems that appear when coding. In Unity, there are a number of integrated design patterns. Through the use of these and the design patterns I learned in the Software Engineering module, I will be able to design a well functioning game.

One example of a pattern that is automatically within Unity is the “Update Method”. The Update Method is a method that has the purpose of making the game aware of an implementation, Update is called every frame that MonoBehaviour is enabled. For example, this would be useful for me when creating a round-timer. I would be able to count down consistently by calling a reduction by 1 to the timer in the Update method, this would cause the clock to decrement, counting down each frame until the timer hits 0.

Another Pattern that is already implemented within Unity is the “Game Loop". This is a loop that is at the core of all games, which functions independently of the hardware used to run the game. This allows Unity games to account for all kinds of computers, which run at different speeds, allowing a consistent count of how much time passes between each frame and therefore maintaining consistency of performance between devices. (Unity Technologies)

A design pattern that will be central to my game is the Singleton. A Singleton is an example of a creational design pattern. A creational design pattern deals with mechanisms of object creation, this is helpful because uncontrolled object creation can create inefficiency. A Singleton is restricted to one object, which allows for simple usage, and becomes a global point of access for unique objects. In the case of my game, it would be helpful for accessing information about each player, such as their position or their level of health.

Another design pattern that would be helpful in my game is the Factory. A Factory is a creational design pattern, which is used for the creation of objects behind the scenes. This would allow me to create instances of player objects with their own statistics, which I can give parameters for. For example, if no player should have more than 100 health, I can set the creation of a player object to only be allowed if the player has under 100 health.

2.4 Literature Review

The background research I have done as part of my report has given me vital information on things that will aid in the creation of my own fighting game project. For example, by looking into other successful fighting games such as Karate Champ, Street Fighter 2, Mortal Kombat 2 and “Karate Champ - Move List and Guide” on the gamespot website; as well as their features I was able to find important features that could be considered the foundation of a good fighting game, as well as features that are good to include to improve the user experience.

Moreover, research into features of Unity such as RigidBody and Colliders will grant me the ability to immediately create features and solve issues that I would otherwise have trouble with, if I did not have prior knowledge about the game engine and its integrated features; I found information on these features within the several Unity Manual sources I have cited. Additionally, research into the application Aseprite’s documentation was useful in understanding how the tool worked in order to create sprites and other graphics for my game. Finally, the research I did into game theory will grant me the ability to have a deep understanding of why certain features of games would be well received by users. This means that I can now create new features and balance them properly to be enjoyable to my game, as well as make sure they are not game breaking. The sources that aided my understanding of this information include Cornell University’s “Fighting, Games, and Game Theory” blog as well as “A Course in Game Theory”.

Chapter 3: **Build**

3.1 Milestones

When creating games, it is critical that testing is done at each stage of development, to make sure everything works before moving to the next stage. Although this is true, standard forms of testing such as unit tests are not the best for testing a game built with Unity. This is because games are an art form, which means more cold and calculated ways of testing would not necessarily produce a game that feels good to play. Unity allows you to run your game at each stage of development. As a result, I have decided to focus on a more manual approach to testing. Prior to developing each component of my game, I planned out Milestones that I intend to accomplish during each stage, this allowed for a form of test driven development (TDD).

Test Number	Test Description	Expected Outcome	Actual Outcome
1	Movement - Can players move in the correct direction using the movement keys?	To be able to move the player in each direction I intend after pressing the corresponding key.	I was able to write a script that makes it so the W, A and D keys move the player left right and jump.
2.1	Dash - Can players dash when they press the dash key?	My player should be able to dash wherever they want by facing a direction and pressing the dash key.	The player was able to dash, but only in their starting direction
2.2	Dash - Can players dash when they press the dash key, in the direction they are facing?	The player should be able to change the direction they dash in between left and right.	The player can now dash in whatever direction they please because of the Invert() function.
3	Attack - Can I detect that someone has been hit when the punch key was pressed while in range of an enemy?	When the player presses the punch button the Debug Log, I put in the Punch method should be displayed.	When the player is punched the Debug Log showing that the attack hit is displayed.
4	Health - When the player hits their enemy, will the enemy's health go down?	When the player attacks their opponent, the opponent's health should be reduced by the correct amount.	When the player hit the opponent, their opponent's health was reduced by the damage variable I set.
5.1	Punch Knockback - Will a player be knocked back if hit with a punch?	When the player punches their opponent, the opponent should be knocked back.	When the player punches their opponent, the player who threw the punch is knocked back.

5.2	Punch Knockback - Will the correct player be knocked back if hit with a punch?	When the player punches their opponent, the opponent should be knocked back.	When the player punches their opponent, the opponent is now knocked back.
6	Kick Attack - Will the kick deal the expected amount of damage to the opponent?	When the player kicks their opponent, they should take less damage than when hit with a punch.	When the player hit their opponent, they got damaged by the correct amount.
7	Kick Knockback - Will the player be knocked back if hit with a kick?	When the player kicks their opponent, they should be knocked back by a greater amount than they were by the punch.	The kick knocked the player's opponent back further than the punch.
8	Health Bar - Will the player's opponent's health bar be reduced by the correct amount, based on the damage of a punch?	When the player punches their opponent, their opponent's should be reduced by 10 hit points on their respective health bar.	The opponent's health was reduced by 10 hit points, which was visually on their health bar.
9	Health Bar - Will the player's opponent's health bar be reduced by the correct amount, based on the damage of a punch?	When the player punches their opponent, their opponent's should be reduced by 20 hit points on their respective health bar.	The opponent's health was reduced by 20 hit points, which was visually on their health bar.
10	Health Bar - Will the first player's health bar be reduced when player two hits them	When the first player is hit by player one, their health bar should be reduced by an appropriate amount	The first player's health is reduced by 10 when they have been punched
11.1	Timer - Will the timer count down at an appropriate rate	When the round starts the timer should begin counting down 1 second at a time.	The timer counted down at an appropriate rate. But the timer would count below 0 to negative numbers, which it should not
11.2	Timer - Will the countdown timer end at 0?	The timer should now count down to 0 and then stop.	The timer now stops at 0.
12	Win Counter - Does the player's win counter increase by 1 when the enemy player's health is reduced to 0?	When player 1 hits player 2 to the point that their health is reduced to 0, player 1's win counter should be incremented by 1.	The win counter is incremented by 1 at the appropriate time; however, it will continue to increase if the player hits their enemy while they are down.

13.1	Player Two - Does the Second player move properly?	Player two should move the same as player one.	Player two moved the same as player one. But, both players would be moved at the same time.
13.2	Player Two - Can each player move independently?	Player Two moves with different controls than player one.	Player Two was controlled with arrow keys, while player one was controlled with WASD keys.
14.1	Player Two - Does the second player attack properly?	Player two should attack properly and independently of player one.	Player two can attack using the full stop and comma keys, and attacks with the appropriate amount of power + damage. But, Invert() was being called at the opposite time that it should have been, because player two is spawned facing player one.
14.2	Player Two - Does player two invert at the correct time?	Player two should be attacking in the direction they last moved in.	Player two only inverts when they are moving in a direction they are currently not facing.
15	Sprites - Does the sprite sheet for idle stance look good?	When played, the sprite sheet should look like an actual idle stance.	The idle stance looks like what a fighter would do when idling.
16.1	Sprites - Does the sprite sheet for walking look good?	When played, the sprite sheet should look like an actual walk cycle.	The sprite looks like he is shuffling, which is more appropriate for a fighter.
16.2	Sprites - Will the walking animation play when appropriate?	Walk animation should play when a horizontal directional key is pressed, and should stop when the player stops moving.	The walk animation plays when moving horizontally, and stops after the player's velocity hits 0. But, sprites would behave strangely.
16.3	Sprites - Will the sprites move naturally?	When walking, the sprites should move smoothly, and not glitch.	By cutting the sprites manually, the strange behaviour of the sprites ended.
17	Game Manager - Will the players' position reset	When a player is hit to the point that their health is	Both players were sent to their start position if

	when a player's health hits 0.	reduced to 0, both players should be reset to their original position.	either player defeated the other.
18.1	Game Manager - Will the players' health be reset when a player's health hits 0.	When a player is hit to the point that their health is reduced to 0, both players should have their health bars set to full.	When a player is defeated, only the defeated character's health was set to full. This must be fixed
19	Game Manager - Does the game over screen appear when a player has won twice?	When a player wins two times out of the set of a potential three games, the game over screen should be displayed.	If a player wins twice, the game over screen is displayed.
20	Game Manager - Does the exit game button work?	Once a game set is finished, the exit button on the game over screen should close the game.	The Exit button closes the game when pressed.
21	Game Manager - Does the reset game button work?	Once a game set is finished, the reset button on the game over screen should close the game.	The Reset button resets the game to the beginning when pressed.
22	Game Manager - Does the correct player win screen display once a game set is concluded?	Once a player wins twice, the correct win screen should be displayed.	If player one wins twice in a set, then "Player 1 Wins" is displayed on the game over screen. If player two wins twice in a set, then "Player 2 Wins" is displayed on the game over screen.
18.2	Game Manager - Do both player's health bars reset after a player is defeated?	When a player knocks out their opponent, both players health bars should be restored.	If player one or two had received damage before defeating their opponent, their enemy and their own health bar was set to full.

3.2 Movement

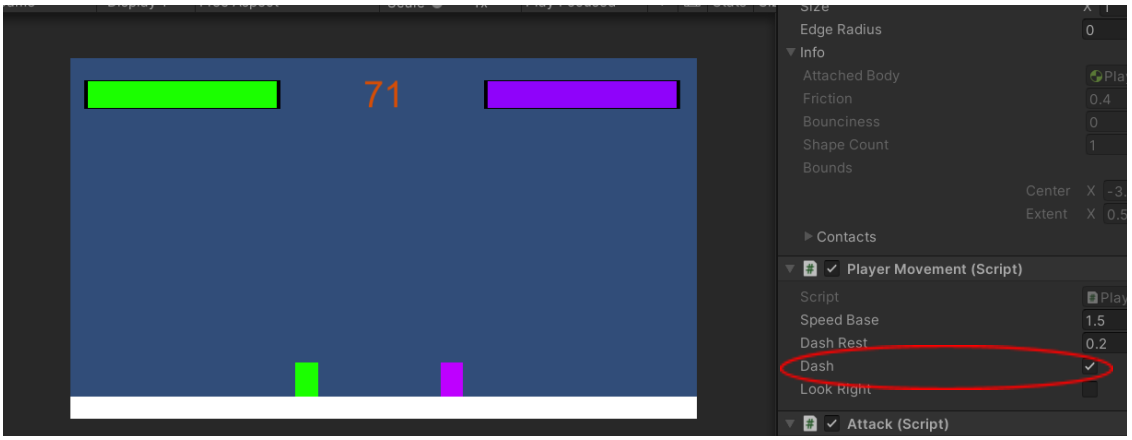
One of the most essential parts of any fighting game is the player movement, this is because weaving in and out of combat is the only thing that stops a fighting game from being a basic and monotonous “button masher”, (i.e., a game where the only objective is to repeatedly hit buttons as the only aim). The code below shows how I implemented movement using the AddForce feature of RigidBody2D:

```
if(moveHoz > 0.1f || moveHoz < -0.1f)
        {
                    player.AddForce(new Vector2(moveHoz * moveSpeed, 0f),
ForceMode2D.Impulse);
        }
```

moveHoz is a variable I used to detect which direction the player was intending to move. By multiplying this by the speed variable of the player, it was possible to add an appropriate force to the player in the correct direction based on their inputs.

Allowing more versatility in play will bring more scenarios that the fight could enter, which results in a much more interesting game; to take advantage of this, alongside basic horizontal and vertical movement, I made a dash method as displayed below:

```
private IEnumerator Dash()
    {
        dashAvail = false;
        isDash = true;
        rb.AddForce(new Vector2(rb.transform.localScale.x * dashSpeed, 0f),
ForceMode2D.Impulse);
        yield return new WaitForSeconds(dashTime);
        isDash = false;
        yield return new WaitForSeconds(dashRest);
        dashAvail = true;
    }
```

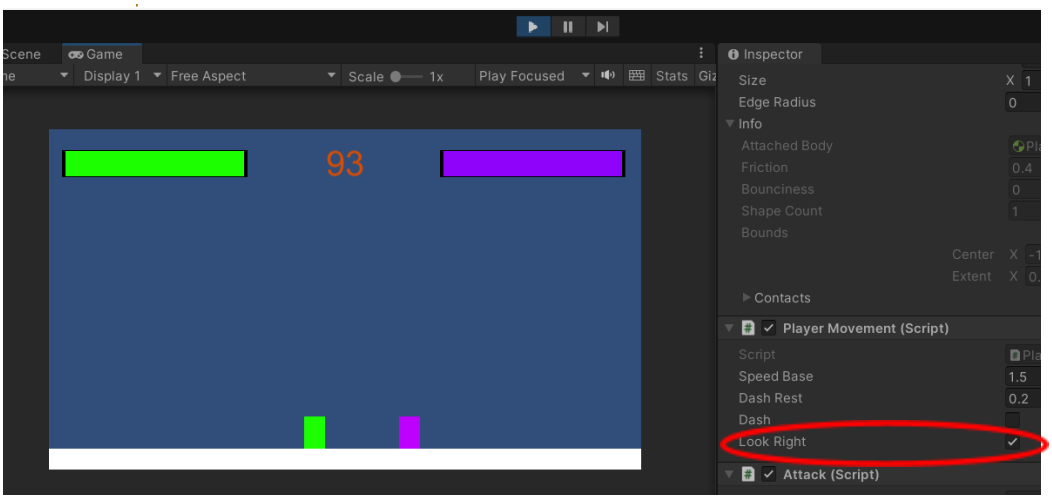


In the beginning of this method, I set the variables “dashAvail” and “isDash” to false and true respectively to allow the game to know when this method is run the dash is no longer available and the player is currently dashing. Then it adds a Vector2 force to the Rigidbody2D attribute of the player object. The method knows which direction to send the player when dashing due to “transform.localScale.x” returning the direction our player is facing, by multiplying this by our float variable “dashSpeed” it sends the player at dashSpeed in that direction, it also keeps the y component at 0, because we do not want the dash to move the player anywhere but in the x axis. Then after some seconds, it sets

“isDash” back to false, the delay is to account for the time the player is dashing, by putting a delay here it will prevent the player from dashing while in the dash. Finally, after some seconds it sets “dashAvail” to true to allow the player to dash again, the delay is to stop the player from dashing immediately after a dash.

I ran into issues with the dash function in that it would always send the player in the same direction, this is because the direction the player was facing did not change despite walking in different directions. In order to solve this, I created an Invert method as displayed below:

```
void Invert()
    {
        Vector2 startDirection = rb.transform.localScale;
        startDirection.x *= -1f;
        rb.transform.localScale = startDirection;
        lookRight = !lookRight;
    }
```



This method takes the initial direction the player is facing in the variable “startDirection” by taking the Rigidbody2D’s localScale. Then I took the x-axis of the start direction and multiplied it by -1, this would invert the player’s direction no matter where they’re facing because 1 * -1 = -1 and -1 * -1 = 1. Next, I made the localScale into the new startDirection, which is the opposite of the initial start direction. Finally, the boolean variable “lookRight” is set to the opposite of whatever lookRight currently is, this is because the start direction of player 1 is to the right, so from the start of the game. This invert method allowed me to change the direction the player was facing, which meant the player could dash left and right, instead of just the start direction.

An issue I realised would arise while making the movement for my player is that they would be able to jump whenever they want, which means they could stay in the air forever by holding down or pressing the jump button multiple times. In order to stop this, I wrote the code below:

```
  void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.tag == "Surface")
        {
            isJump = false;
        }
    }

    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.gameObject.tag == "Surface")
        {
            isJump = true;
        }
```

The methods OnTriggerEnter2D() and OnTriggerExit2D() detect when two game objects collider with one another, in this case, I used them to detect when the floor of the game which I tagged as “Surface” came into contact with a small hitbox I placed on the bottom of each player. This allowed me to change the boolean variable “isJump” to true when the bottom of the player is not in contact with the surface and change isJump to false if the bottom of the player is in contact with it. The following if the statement shows how I used this variable in the FixedUpdate method to make the player jump:

```
if (!isJump && moveVert > 0.1f)
        {
            rb.AddForce(new    Vector2(0f,    moveVert    *    jumpForce),
ForceMode2D.Impulse);
        }
```

Here I checked if the variable isJump is false and if the player has chosen to move vertically using the up button for a value higher than 0.1.

3.3 Game Manager

A Game Manager is an integral part of any game. The Game Manager can be accessed from any class in my code, which allows for the control of all methods that would be considered vital to the game.

For example, the Reset() function is a core function that allows my game to function properly. Without the reset function, a player would beat their opponent down to 0 health but no winner would be declared. This is because Reset() handles the movement of players back to their start states, as well as the regeneration of their health statistics back to full. Additionally, Reset() will bring up the game over panel by calling GameOver() as well as displaying the winner by using the Winner() method. This can be seen below:

```
public void Reset()
    {
        if (playerOneHealth.Health <= 0 || playerTwoHealth.Health <= 0)
        {
            HealthZero();
        }
        else if (timeUp)
        {
            TimeUp();
        }
        if (winsOne == 2 || winsTwo == 2)
        {
            Winner();
            GameOver();
        }
    }
```

This is accomplished by checking each player object's health stat to see if they have become equal to or less than 0. In the case that this has happened, both players will be relocated to their start position to allow for a new round to begin and their health bars will be set to full. Additionally, the correct player's win count will be incremented. This was accomplished by checking which player's health hit 0, then incrementing their opponent's win counter. Additionally, Reset() is invoked if the timer runs out, in the event that this occurs, the health stats of both players are compared. The player with the higher health stat is declared the winner and their win counter is incremented. GameOver() is a method that displays the game over screen and grants the player the option to either continue playing the game by pressing the reset button, or exit the game using the exit button.

The Winner() method will activate the correct player's win screen, the code for this can be seen below:

```
 private void Winner()
    {
        if (winsOne >= 2)
        {
            winScreenOne.SetActive(true);
        }
        else if (winsTwo >= 2)
        {
            winScreenTwo.SetActive(true);
        }
    }
```

This accomplishes its job by checking if either player's win counter has reached or exceeded 2. In the event that this does happen, their win screen is displayed.

The GameManager is responsible for handling the use of each player's objects however, this is only possible by calling the PlayerHealth class as displayed below:

```
public PlayerHealth playerOneHealth = new PlayerHealth(100, 100);
public PlayerHealth playerTwoHealth = new PlayerHealth(100, 100);
```

The PlayerHealth class is a Factory, which is responsible for the creation of player objects, this is displayed below:

```
public class PlayerHealth
{
    public int currentHealth;
    public int currentFullHealth;

    public int Health
    {
        get
        {
            return currentHealth;
        }
        set
        {
            currentHealth = value;
        }
    }

    public int FullHealth
    {
        get
        {
           return currentFullHealth;
        }
        set
        {
            currentFullHealth = value;
        }
    }

    public PlayerHealth(int health, int fullHealth)
    {
        currentHealth = health;
        currentFullHealth = fullHealth;
    }

    public void TakeDamage(int damage)
    {
        if (currentHealth > 0)
        {
            currentHealth = currentHealth - damage;
        }
    }
}
```

The PlayerHealth class creates properties that can be manipulated in order to alter the fields within the methods PlayerHealth and TakeDamage(). When called, this allows for the manipulation of each player's health by setting them to an amount or reducing it by an amount set by the value you put for each value.

3.4 Combat

Combat is an idea central to the fighting game genre. In order for this to be realised in my game, I created methods to apply damage and knockback to a player who is hit with an attack. This allows for each player to have a reason to be alert during the game as they want to apply damage to their opponent, whilst also avoiding being damaged themselves.

In order to detect when the player has hit his opponent with a punch I created a Punch() method, which is shown below:

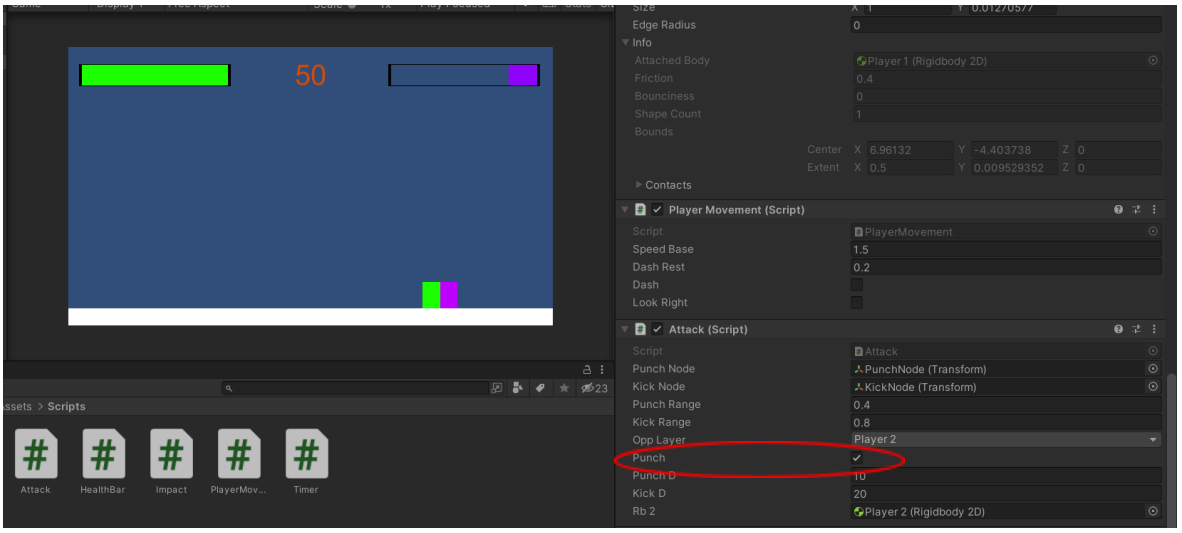
```
//Method called to apply appropriate damage and knockback to an enemy who
is punched.
    private void Punch()
    {
                                                Collider2D[]      punched      =
Physics2D.OverlapCircleAll(PunchNode.position, punchRange, oppLayer);
        foreach (Collider2D opp in punched)
        {
            opp.GetComponent<PlayerTwoImpact>().PlayerPunched(punchD);
            punchKnockback(opp);
            break;
        }
    }
```

In this method, I created an array (called “punched”) of Collider2D elements that are within the circle created from the punchRange, which is the radius around the PunchNode I created in Unity to be used as the point the player’s punches will come from. The issue that this would cause is that the player themselves would be within the range of the punch, meaning that they would be hit too, in order to mitigate this, I created an “oppLayer”, which I assigned as the Player 2 layer in Unity, so Player 1’s punches would only hit Player 2.

In order to apply damage to the people hit by the punch I used a foreach loop to go through each element in the array “punched”. Within this loop I called the Punched() method from my Impact class, which is shown below:

```
  public void PlayerPunched(int damage)
    {
        GameManager.gameManager.playerTwoHealth.TakeDamage(damage);
        GameManager.gameManager.playerTwoHealth.Health -= damage;

healthBar.SetHealth(GameManager.gameManager.playerTwoHealth.Health);
        if (GameManager.gameManager.playerOneHealth.Health <= 0)
        {
            KO();
        }
    }
```



In Punched() I deduct an amount of damage assigned in the variable “punchD” from the player’s overall health and check if their health becomes equal to or less than 0, which implies they have been Knocked Out, so it calls the method KO().

After detecting the damage that should be inflicted on the opponent, Punch() applies knockback by calling the method punchKnockback(), which is shown below:

```
   public void punchKnockback(Collider2D player)
    {
                    Vector2   trajectory   =   (player.transform.position   -
transform.position).normalized;
        body.AddForce(trajectory * punchP, ForceMode2D.Impulse);
    }
```

This method calculates the trajectory I want to send the opponent after hitting them by normalising the position the opponent is in and subtracting the position the attacker is in, by normalising this value it returns the direction I want to send the opponent with the value of 1. Then, the normalised trajectory is multiplied by “punchP”, a variable used to store the float value of punch power.

I had an issue with my knockback feature because when the player would punch or kick their opponent, instead of knocking them back the knockback would be applied to themself. In order to solve this issue, I first narrowed down the code that the error was appearing in, then I realised that the call to move was being performed on the wrong entity. In order to remedy this I removed the line of code where I assigned the Rigidbody2D variable “body” upon start up (in the method Start()) and made the variable public. This allowed me to manually assign the variable to the player’s opponent, thus correcting the issue of attacks knocking back the wrong player.

After I programmed the player's punches, I added similar functionality for kicking. However, the variable used to store kick power is greater but the damage is less. This is to represent how “teep” kicks are a more potent form of attacking than punching.

I ran into another issue with hit detection after implementing a kick function. The issue was that when kicking their opponent, for some reason the player would be able to land their attack twice per press of the kick button. This was not an intended feature of the game so, in an attempt to stop this from happening, I broke the foreach loop after each application of damage and knockback.

```
//Method called to apply appropriate damage and knockback to an enemy who
is kicked.
    void Kick()
    {
        Collider2D[] kicked = Physics2D.OverlapCircleAll(KickNode.position,
kickRange, oppLayer);
        foreach (Collider2D opp in kicked)
        {
            opp.GetComponent<PlayerTwoImpact>().PlayerKicked(kickD);
            kickKnockback(opp);
            break;
        }
    }
```

As a result of this, the player would only be able to hit their enemy player with one instance of damage and knockback before their opponent would no longer be in the foreach loop. This meant that the unfair bug had been fixed.

3.5 HUD

The Heads Up Display or “HUD” is an important part of the game experience. It allows the player to know how much damage they have dealt, as well as the amount of damage they have received through the health bars. It makes the player aware of the amount of time left in the round through the round timer and it reminds the player of the number of wins or losses to their opponent; allowing them to know how close they are to winning or losing. Without a HUD, the players would be attacking each other with no visual representation of the effect they are having. Therefore making the game less rewarding to play.

I created a HealthBar as part of the Heads Up Display (HUD) of my game, which is utilised by my GameManager. The HealthBar class consists of the methods SetFullHealth() and SetHealth(), which are shown below:

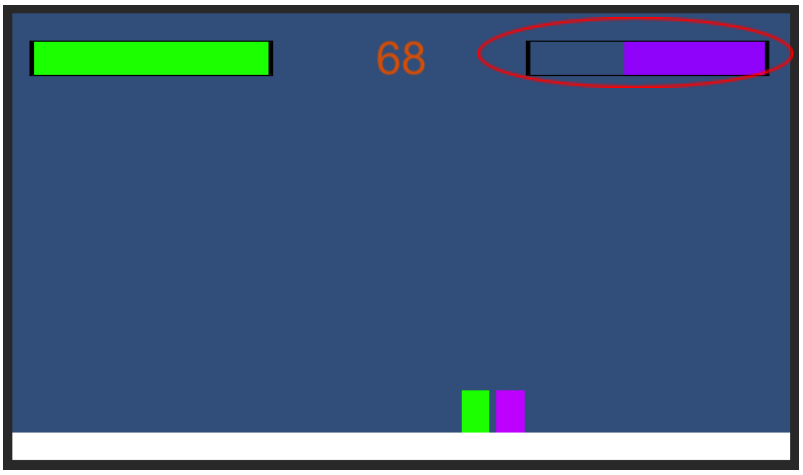
```
    public void SetFullHealth(int health)
    {
        healthSlider.maxValue = health;
        healthSlider.value = health;
    }

    public void SetHealth(int health)
    {
        healthSlider.value = health;
    }
```

SetFullHealth() is used to set the player’s health bar at the start of the match as full. Then, throughout the game SetHealth() is called to adjust the health bar slider to the player’s health at each frame. The HealthBar class works closely with the Impact class to achieve this, as shown below:

```
  public void PlayerKicked(int damage)
    {
        GameManager.gameManager.playerTwoHealth.TakeDamage(damage);
        GameManager.gameManager.playerTwoHealth.Health -= damage;

healthBar.SetHealth(GameManager.gameManager.playerTwoHealth.Health);
        if (GameManager.gameManager.playerTwoHealth.Health <= 0)
        {
            KO();
        }
    }
```



SetFullHealth() is called in Start() to make sure that at the beginning of the game, the health bar is set to the maximum value, which I set in the variable “fullHealth”. Then, in Kicked(), SetHealth() is called and given the value stored in the variable “health” after deducting the kick damage. A similar method to Kicked() is used in the Punched() method, but with greater damage.

SetHealth() is also called within the Game Manager in order to set the health bars of each player to an appropriate level after a game should be reset. This can be seen in the following methods:

```
private void HealthZero()
    {
        player1.transform.position = new Vector3(-2.37f, -1.97f, 0);
        player2.transform.position = new Vector3(4f, -1.97f, 0);
        if (playerOneHealth.Health <= 0)
        {
            winsTwo++;
            healthBarTwo.SetHealth(fullHealthTwo);
            healthBarOne.SetHealth(fullHealthOne);
        }
        else if (playerTwoHealth.Health <= 0)
        {
            winsOne++;
            healthBarTwo.SetHealth(fullHealthTwo);
            healthBarOne.SetHealth(fullHealthOne);
        }
        FullHealth();
        time = startTime;
    }

    private void TimeUp()
    {
        player1.transform.position = new Vector3(-2.37f, -1.97f, 0);
        player2.transform.position = new Vector3(4f, -1.97f, 0);
        if (playerOneHealth.Health < playerTwoHealth.Health)
        {
            winsTwo++;
            healthBarTwo.SetHealth(fullHealthTwo);
            healthBarOne.SetHealth(fullHealthOne);
        }
        else if (playerOneHealth.Health > playerTwoHealth.Health)
        {
            winsOne++;
            healthBarTwo.SetHealth(fullHealthTwo);
            healthBarOne.SetHealth(fullHealthOne);
        }
        FullHealth();
        time = startTime;
        timeUp = false;
    }
```

The methods HealthZero() and TimeUp() are called in the Reset() method at appropriate times. HealthZero() is called when a player's health is reduced to zero and TimeUp() is called when the timer hits 0; both methods reset the players to their starting positions and resets their health to full.

3.6 Handbook

In order to run my game, you must have PracticeMakesPainful.exe, UnityPlayer.dll as well as the files Practice Makes Painful_Data and MonoBleedingEdge in the same directory. Then run Practice Makes Painful.exe

In the event that the executable does not work, in order to create a new executable you will need to have Unity downloaded. Open the Practice Makes Painful folder, go to File -> Build and Run.

The controls of Practice Makes Painful are as follows:

Player 1:

W = Jump

A = Walk Left

D = Walk Right

Q = Punch

E = Kick

Left Shift = Dash

Player 2:

Up Arrow = Jump

Left Arrow = Walk Left

Right Arrow = Walk Right

Comma = Punch

Kick = Kick

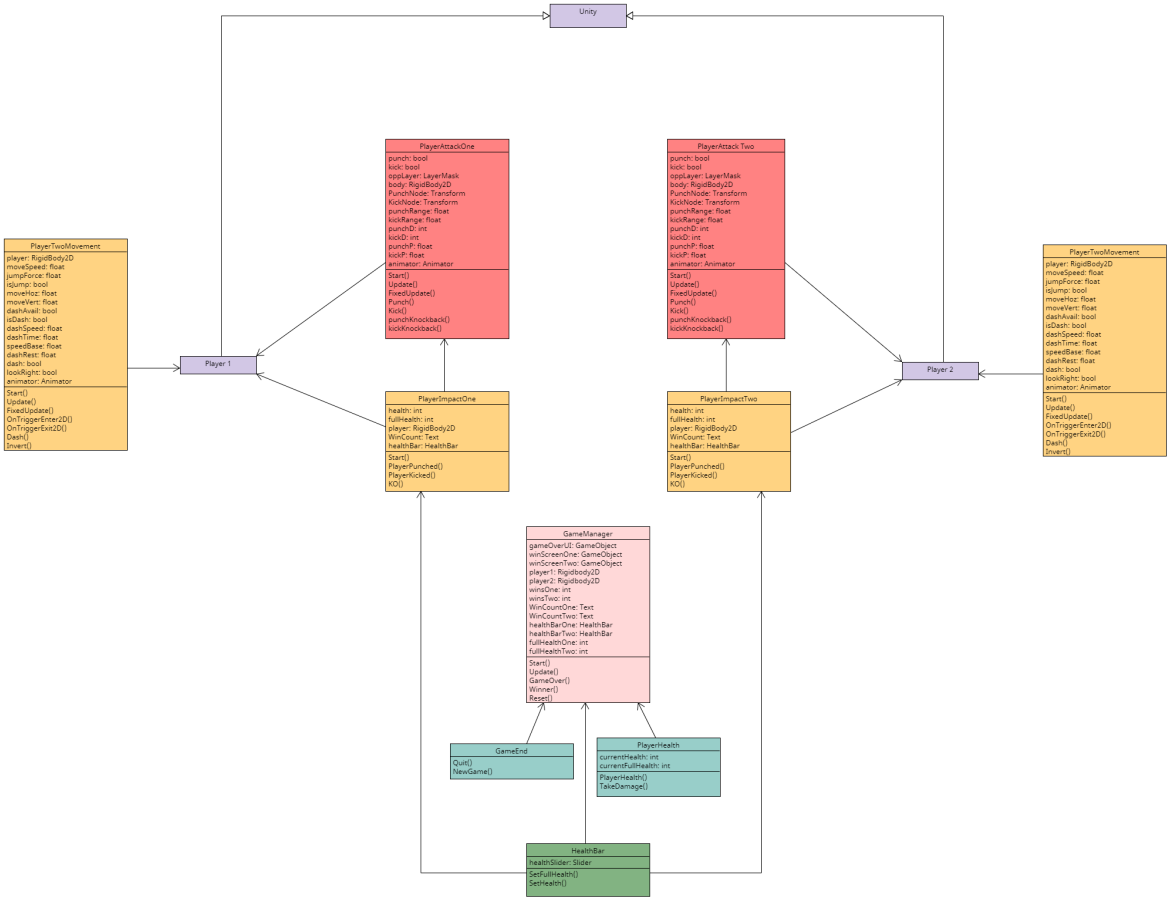
Right Shift = Dash

When a game set is finished, you will be presented with the option to either “Exit” or “Reset” the game. If you want to continue playing, press Reset. If you want to stop playing, press Exit.

Chapter 4: **Design**

4.1 Overview

UML



My project is based around the manipulation of GameObjects, as such my scripts work together to manipulate their attributes.

For example, my PlayerMovement class of both players take in button inputs from the player as triggers to apply forces to the GameObject of their character, allowing them to travel left and right, as well as jump. Additionally, my PlayerMovement classes contain a dash feature, which allows players to quickly move towards or away from their opponent, allowing them to evade or move in to inflict damage.

I also have player Attack and Impact classes; these allow for the player to deal damage as well as detect when a player has been hit and received damage. My game has both a punch and kick function, with the kicks dealing more knockback but less damage. This is because the kick is designed as a push kick to make space more so than to deal damage.

My HealthBar class works closely with the Impact classes, this is because the damage received by the enemy player is displayed through the bars on the top of the screen. The amount of damage applied by the Impact class, which varies depending on the type of

attack the player is hit with, is displayed through the manipulation of the slider on their health bar.

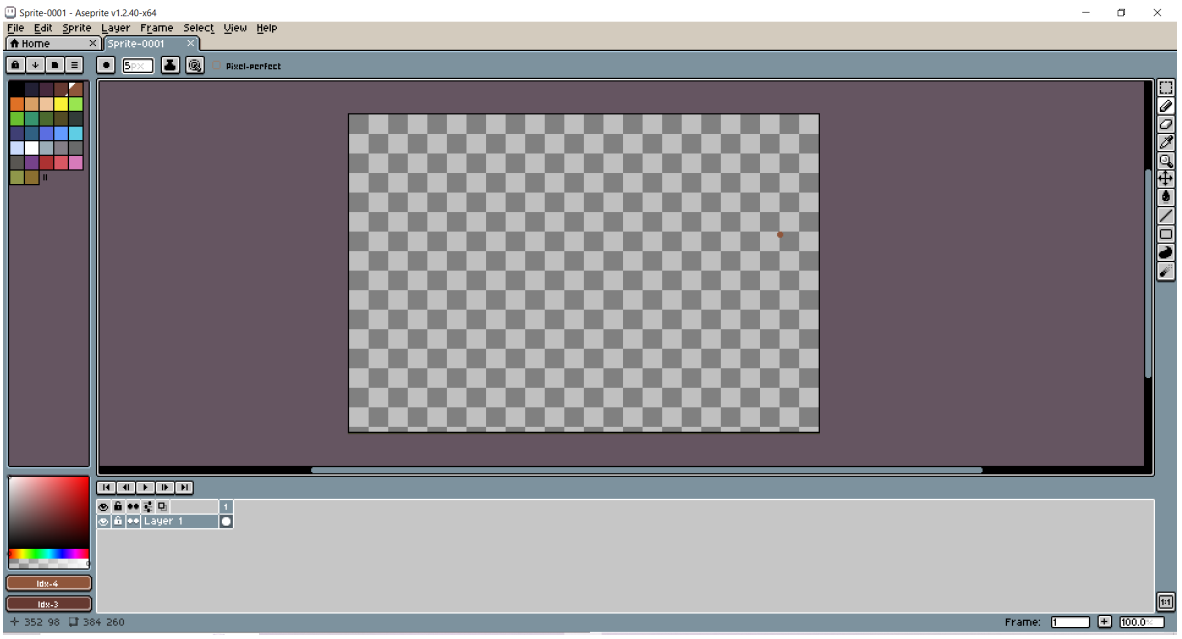
I have created a countdown style timer that is displayed in between the health bars at the top of the HUD, this is achieved in the Timer class by ticking down the timer every real second and displaying this number through a string.

Between all of this is my GameManager. The Game Manager is a class used to control the flow of the whole game. For example, it detects when rounds or game sets have been completed and declares a winner. It is also in control of the win counters and the timer for the game.

The classes for PlayerMovement, Attacks and Impact are similar and I am aware that in cases like this it is best to utilise inheritance. However, due to the way Unity works, code is attached to GameObjects as if they were components. As a result, I found it necessary to created separate scripts for each player's movement, attacks and impact due to them requiring different inputs.

4.2 Aseprite

Aseprite is an application that can be used to create 2D pixel art, as well as sprite sheets for animation. These sprite sheets are exportable, so they can be sliced and used in conjunction with Unity in order to animate the sprite sheets when appropriate. For example, the player can move in their idle stance but once they move, their idle animation will be swapped for their walking animation in response to the movement key being pressed.

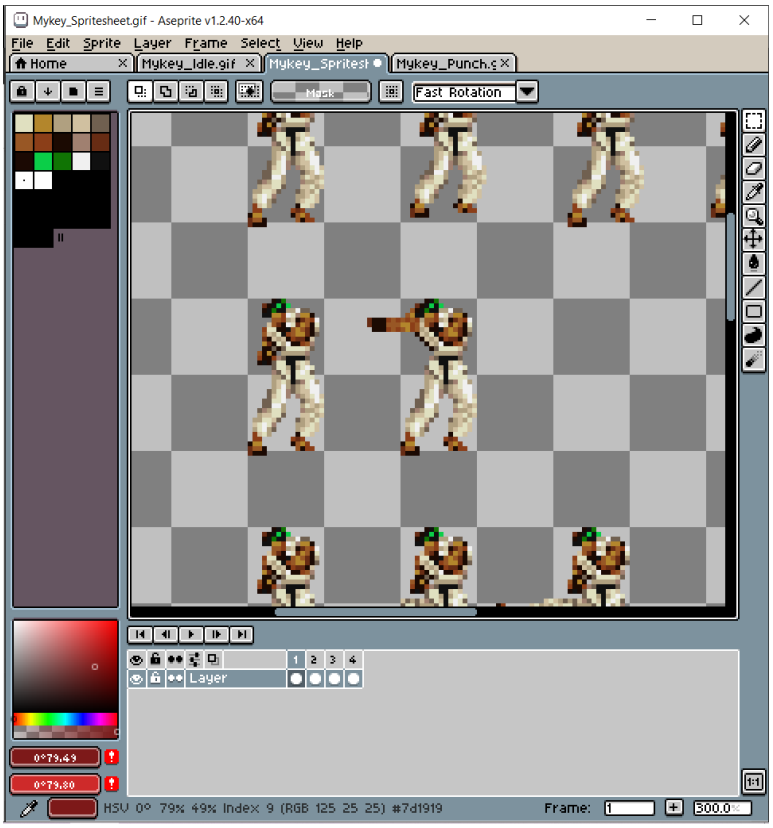


Aseprite contains several features that make the creation of pixel art and sprite sheets easier. Therefore I decided to use it to create both the background and the sprites for the floor as well as the characters that are usable in my game.

4.3 Sprites

Good fighting games are not only mechanically fun to play but are also aesthetically pleasing. In order to accomplish this I utilised an application called Aseprite to create sprite sheets of characters. As well as, Unity's built in animation feature to put these sprite sheets into motion. By animating my sprites players will receive an immediate feeling of feedback upon pressing a button. This will give the player a sense of achievement that their inputs are being received and are working as intended.

Aseprite is an "animated sprite editor and pixel art tool", this tool allowed me to create 2D sprites in the form of sprite sheets. The pencil and eyedropper tools allowed me to select colours I wanted to be present in my sprite and draw them on a canvas. Then, using the rectangular marquee tool, I could copy my sprite over to aid in creating the next frame of movement. Erasing pieces that didn't belong using the eraser tool, and adding new ones, an example of this can be seen below. (Aseprite)

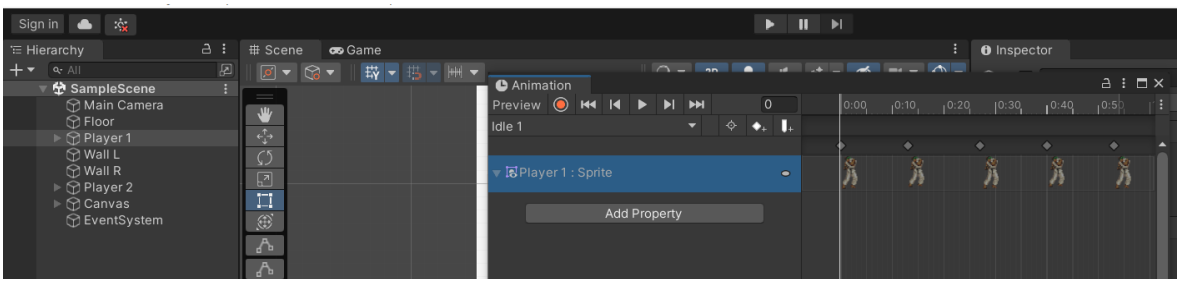


In my punch animation, the idle stance was correct. However, in order to throw the jab, the character's left arm was in the way. To solve this issue, I removed the current arm and created a new one that was outstretched to the right position.

After creating each sprite I intend to be in the animation, I split them into separate layers and imported them into Unity. Then, in Unity, I was able to automatically slice the sprites out. There was an issue where the automatic slicing button made it so the character was centred, as the slice would take the dimensions of the sprite. In order to solve this issue I manually adjusted the dimensions of the slice, such that the character model would not seem to be jumping in place. Instead, by giving the sprite a few pixels of space on the appropriate side of the character, it made him seem as though he was jumping from side to side.

4.4 Unity Animation

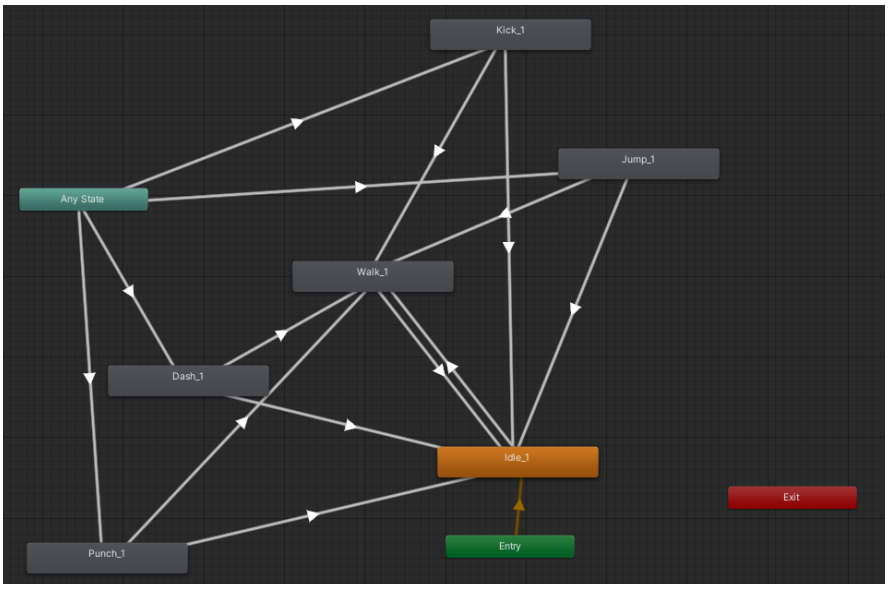
After importing and slicing the sprites, I opened the Unity animation window and dragged them in. The animator presents each frame as a node and allows you to drag them in order to alter when each frame is shown. Because the example below is an idle animation, I made the equidistant.



There was an issue where the animation loop would cut short as a result of the starting position being the final position. In order to solve this issue, I created a copy of the start position and placed it the distance from the penultimate position I wanted. This meant that the loop was completed so, when the idle animation began again there was no jump cut, allowing for a smoother animation.

I was able to utilise animations created using this animation window by using Unity's animator controller, which allows me to arrange my set of animation clips, connected by transitions. These transitions utilised variables that would be manipulated by my code in order to activate different animations at their appropriate time. These variables can be seen below:

These variables would be utilised in transitions to tell the animator controller when it should transition to another animation. An example of this in my code would be the connection between the Idle and Walk animations as shown by the diagram below:



The animator utilised a “State Machine, which could be thought of as a flow-chart of Animation clips” (Unity Technologies). As you can see, there are two transitions between the Idle and Walk clips. This is because the player must be able to shift, not only from idling to walking, but also back to idling once the player stops walking. I accomplished this by using the float variable “horizontal”. When the horizontal variable was set to a value greater than 0.1, the transition from Idle to Walk was activated. However, in the event that the speed of the player is reduced to a value below 0.1, their animation would return to idling from walking. This was achieved by using the code below:

```
animator.SetFloat("horizontal", Mathf.Abs(moveHoz));
```

This accomplishes the goal of manipulating the horizontal variable to be appropriate to the value it should be based on the player’s horizontal movement. This is accomplished by using the SetFloat command on the variable “moveHoz”, which I used in the player movement class to move the player horizontally. I took the variable’s absolute value. This is because, I ran into an issue where in the event, the player is walking left the variable “horizontal” would be set to a negative number. This would be seen as horizontal having a value under 0.1 and would result in the idle animation playing. Despite the player clearly walking left.

The Unity Animator Controller also had a node called “Any State”. This node allowed me to make transitions from it such as punching, which would mean that, no matter what animation was currently playing the player would be able to see that they have thrown a punch. This was helpful in a fighting game like mine because there are situations where you may want to attack in mid-air, which would seem strange if there was no animation played.

Chapter 5: **Testing**

Testing is an important part of the development of any project. It allows you to be certain that anything made as part of the creation process works as intended and greatly reduces the possibility of deeply seeded errors being discovered later. Unfortunately, when creating games standard Unit testing is less than optimal when creating a Unity game so the majority of my testing was accomplished using Milestones during the building process.

Although this is true, I did complete some User Acceptance Testing (UAT) at the end of the development process. This involved getting participants to play my game and give me feedback on their experience and tell me any features they found performed badly or features they would like me to add.

Firstly, I completed testing on the way player inputs interacted with my game. This was accomplished by letting people play the game against each other and point out any features that were not performing properly. For example, there was an issue where player two's KickNode was placed further away from their body compared to player one. This provided an unfair advantage of the range they could attack from. In order to fix this, I altered player two's KickNode to be exactly the same as player one's by inputting the same coordinates.

Secondly, I completed tests on whether the game's Head's Up Display (HUD) performed as intended. While testing this feature, my users notified me that there was a bug where, when a player had sustained damage but ultimately won a round their health bar would be reset to full. However, upon being hit by their opponent their health bar would be sent back down to their health during the last round. This made me aware that there was a bug in how I was making damage display on the health bars. The issue was that there was a variable I was using that was separate to the game's actual health system to alter the health bar. The solution to this issue was to remove those irrelevant variables that were being used and allowed the health bars to be altered directly by the game's health system.

Finally, I completed testing on the graphics of my game. Through UAT I was made aware that the walking animations for the second player was delayed from their actual movement, as well as the first player's punch animation glitching. To solve both of these issues I looked into the Unity Animator Controller and realised that the walk animation of the second player was waiting for an exit time when transitioning from Idling to Walking so I removed this. I also realised in the Animation Window for player one's punch animation that their punch animation was too fast, which resulted in the animation repeating upon a button press and causing the visual glitch so I slowed down the animation by creating a greater space between the keyframes of their animation.

5.1 Requirements

1: Create Players that can move freely and enjoyably.	I accomplished this by giving game objects the RigidBody2D component and applying forces to them based on player inputs. I also created a dash feature, which allows for rapid movements in to attack, or escape from an attack.

2: Create a health system for players, which can be reduced by their opponents attacks.	I accomplished this by using a factory design pattern to construct a player's health. Then I created attack classes that apply an amount of damage to their health. Finally, I created impact classes, which observe the opposing player's attack class and subtracts the appropriate amount of health from the player that was attacked.
3: Create a HUD that contains health bars, a timer and win counters.	I accomplished this by creating a canvas on Unity and adding to it, a text box containing a number, which is manipulated by the Game Manager to count down. Adding two blocks with sliders, which are manipulated by the health system to be reduced with their corresponding player's health. Finally, by adding two text boxes, which contain numbers that are incremented by the game manager when the appropriate player wins a set of rounds.
4: Create different attacks to make the game more versatile.	I accomplished this by creating a kicking feature. The kick is a longer range attack that deals less damage, more knockback but has a greater cooldown before the player can kick again. This creates versatility in the game because players must decide during each neutral interaction whether they would rather move in to deal more damage or, if they'd rather play it safe and deal less damage from the outside of their opponent's punch.
5: Create sprites which are animated.	I accomplished by using the application "Aseprite", which allowed me to draw sprites and utilise these to create sprite sheets. I later used these sheets to slice in Unity and make into animation using Unity's Animation system and Unity's Animator Controller to make sure the correct animation is played at the correct time. This gives players immediate feedback that their actions are working.

Chapter 6: **Diary**

10/8/22
Today I connected GitLab with the Unity Engine, which I will be using to make my game.

11/10/22
Basic placeholder for the floor and player made. Also created and adjusted forces applied due to user input. There was an issue where the user could hold down the jump key, allowing them to be in the air forever, in order to prevent this I added an extra box collider at the bottom of the player object to detect whether they were in contact with a surface or not; then I created if statements to check whether the player was in contact with the ground or not to dictate if they should be able to jump again or not.

16/10/22
Adjusted movement speed and jump height of the player. As well as started creating the dash feature for the game.

19/10/22
Finished off the dash function of my game. Ran into an issue that it would only dash in one direction, so I added a feature that flips the character to whatever direction the player walks, allowing for control over the dash direction.

21/10/22
Created an Attack class. Ran into an issue that I did not have anything to test it on, in order to solve this I made a second character and coded it so that it ran a debug log on impact.

27/10/22
Created a health system using the Impact class and worked on the knockback feature, had an issue that it was knocking back the wrong player, must fix.

03/11/22
Began making a first draft of the interim project report. Abstract and Introduction completed, as well as a list of Objectives.

09/11/22
Created a Kick function with different knockback power and damage. However ran into an issue where the kick will hit the player twice at once, which must fix later.

10/11/22 - 15/11/22
Began work on the Background section of my interim report. And created a basic health bar.

17/11/22
Fixed bugs in knockback by assigning the variable for knockback to the enemy player at the start of the game, and solved the multiple kick issue by breaking the loop.

20/11/22
Working on the Build part of my report.

21/11/22
Implemented health bar into the impact health and created second health bar.

23/11/22

Created a timer and fixed a bug that allowed the timer to count down into the negatives by checking if the timer was at 0 or below, then stopping it there.

14/01/23
Implemented initial version of win counter.

21/01/23
Expanding on my report's background section.

28/01/23
Created the "opponent" branch, where I will be working on creating the controls for my secondary character.

While swapping the initial player's movement controls from checking "GetAxisRaw" to "GetKeyDown" I ran into an issue where, if the player pressed a key, the action would continue forever. To resolve this I used else if statements to check if the key was released, if so the value of horizontal and vertical movement was returned to 0.

29/01/23
Created a movement class for the second player and began the second player's attack class by creating a method of punching.

I had an issue where the player would invert at the opposite time they were supposed to, to solve this I altered Update to call Invert() in the opposite moments to player one. This however caused the dash function to be inverted also, to fix this I multiplied the value of the dashSpeed variable by -1.

5/02/23 - 16/02/23
Learning "Aseprite", a tool for creating and animating sprites.

18/02/23
Created sprite sheet for player's idle stance. Had an issue where it automatically sliced the sprites wrongly, which made the player seem as though he was jumping in place instead of side-to-side. Fixed this error by manually adjusting slices.

23/02/23
Worked on the project report. Created an intro for the Milestones section, this gives an explanation of how it is tested. Created a Sprites sub-section for my build section and looked into design patterns Unity uses as part of the software.

25/02/23 - 27/02/23
Began implementing sprites into my game and continued report writing.
28/02/23 Implemented animation for walking, which activates by detecting horizontal movement and then playing the animation.

Ran into an issue where the sprites were positioned funnily and used a feature in unity to choose the centre of the sprite when the slice was a different size from the other slices in the sprite sheet.

Another issue where punch animation would not play when called must fix.
02/03/23

Created a way of resetting my game.

Ran into an issue where it would reset the entire game, including the counts that I needed for counting round wins and game wins. Instead, I will use this to reset the game for a new play session.

Created a new reset method, which returns all players to their right place and filled each player's health bars to solve this.

Ran into an issue where the winning player's win count would reset but not the losing player's.

04/03/23
Created a full reset function called NewGame(), which when called resets everything. Called when a player wins twice.

I also created sprite sheets for jumping and dashing.

11/03/23
Created a game over screen to be displayed after a set is finished. Allowing the players to see the number of wins and health bar over the game over screen. Thus showing the winner.

Ran into an issue where the game would reset immediately, making it impossible to properly see the game over the screen.

To solve this, I used the Invoke method, which allowed for a delay in the call to NewGame().

Realised a bug through User Acceptance Testing. The winning player’s health will not reset upon beating their enemy. Must fix.

12/03/23

Realised a bug through User Acceptance Testing. Winning player's health will not reset upon beating their enemy. Must fix.

Began refactoring code to remove code smells.

14/03/23

Created a player win screen, which displays when the correct player wins twice in a set.

18/03/23

Created and implemented secondary character's sprite animations.

Also fixed bug where only 1 player's health would be reset upon a KO by refilling the winner's health after beating their opponent in the game manager.

Chapter 7: **Professional Issues**

Management is an important process that is relevant throughout the entire development of a project. Management is key at the start of a project, this allows you to make sure you are able to allot the amount of time and resources needed to it before you commit to it. In a situation where you are not able to do this, it may be appropriate to consider reducing the scope of the project. This is because, in order to adjust a project variable such as time or cost to be more manageable, there must be a trade-off with another. For example, if you want a project to take less time, you would have to spend more money to employ more staff or reduce the scope. By reducing the scope you will be aiming to achieve less by the end of the project.

Management is also important to take place during the project. This is because it allows you to make any adjustments to unforeseen issues in production or address shortcomings. An example of the importance of proper management can be seen in my own project. Before beginning work on my game, I created a proposed timetable for when I would like certain parts of my project to be done as part of a plan. This helped me by making me aware of whether I was completing my goals on time, or if I was falling behind schedule. If I was not completing tasks on time I would prioritise working on my project more than my class work and, if I was ahead of schedule on my project, I would be able to focus more on my class work.

Furthermore, as I went further into my project, I realised my end goal was a bit too ambitious. So, in order to reach a satisfactory end product, I reduced the scope of my game. This will allow me to achieve a functional final project within the deadline and, if I end up with spare time in the end, I could use it to begin work on a new build that has more features. Whilst also having the old build to fall back on, just in case there was not enough time to fully implement the new version.

Thanks to the use of management I was able to create a functioning game within the deadline I was given. However, from the information I have discovered about management, I can now see that. If I had managed my time better at the start of the production process I would have been able to create a better product. This is because, I would not have had to spend valuable development time on planning the project and how I was going to go about accomplishing those goals. Additionally, if I had managed the process better I would not have had to spend as much time refactoring my code during development, thus granting me more time to create more features. Although I did create a well functioning program, which has been put through proper testing to make sure it performs well, with better management, I would have been able to create an even more refined game. I planned to have more features such as blocking, long ranged attacks, special attacks, and a choice of characters to select from. However, due to an overly ambitious plan at the start and poor time management the scope was reduced; resulting in my final product not containing these features.

In conclusion, management is an invaluable skill that should be present in all phases of development. Proper management allows for a project to be completed to a satisfying degree in an appropriate amount of time. This is because it minimises the risk of delays occurring thanks to the creation of a plan prior to development and it allows an individual or team working on a project to know when to alter or pivot away from the initial plan.

Chapter 8: **Bibliography**

Reference list

BillWagner (n.d.). *C# docs - get started, tutorials, reference.* [online] learn.microsoft.com. Available at: https://learn.microsoft.com/en-us/dotnet/csharp/ [Accessed 15 Nov. 2022].

Brackeys (n.d.). *2D Animation in Unity (Tutorial)*. [online] www.youtube.com. Available at: https://www.youtube.com/watch?v=hkaysu1Z-N8&ab_channel=Brackeys [Accessed 28 Jan. 2023].

Brackeys (n.d.). *How to use GitHub with Unity*. [online] www.youtube.com. Available at: https://www.youtube.com/watch?v=qpXxcvS-g3g&ab_channel=Brackeys [Accessed 14 Nov. 2022].

CAPCOM (n.d.). *Street Fighter Series | CAPCOM*. [online] Street Fighter Series. Available at: https://www.streetfighter.com/en/ [Accessed 16 Nov. 2022].

Capello, D. (n.d.). *Aseprite*. [online] www.aseprite.org. Available at: https://www.aseprite.org/ [Accessed 25 Feb. 2023].

Cornell University (n.d.). *Fighting, Games, and Game Theory : Networks Course blog for INFO 2040/CS 2850/Econ 2040/SOC 2090*. [online] Available at: https://blogs.cornell.edu/info2040/2021/09/20/fighting-games-and-game-theory [Accessed 21 Jan. 2023].

Ketonen, M. (2016). *Designing a 2D fighting game Tradenomi Tietojenkäsittely*. [online] Available at: https://www.theseus.fi/bitstream/handle/10024/118514/Thesis_Miikka_Ketonen_KAT13PT.pdf?sequence=1&isAllowed=y [Accessed 14 Nov. 2022].

Krossing, D. (n.d.). *HOW TO MAKE A HEALTH BAR IN UNITY | How to Make a Health Bar for Beginners | Learn Unity For Free*. [online] www.youtube.com. Available at: https://www.youtube.com/watch?v=FBDN4b9PGgE&ab_channel=DaniKrossing [Accessed 30 Nov. 2022].

Muthoo, A., Osborne, M.J. and Rubinstein, A. (1996). A Course in Game Theory. *Economica*, 63(249), p.164.

SBrsainfreeze (1999). *Karate Champ - Move List and Guide - Arcade Games - By SBrainfreeze - GameFAQs*. [online] gamefaqs.gamespot.com. Available at: https://gamefaqs.gamespot.com/arcade/583892-karate-champ/faqs/1294 [Accessed 15 Nov. 2022].

Technologies, U. (n.d.). *Unity - Manual: Animator Controller*. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/Manual/class-AnimatorController.html [Accessed 19 Mar. 2023].

Technologies, U. (n.d.). *Unity - Manual: GameObjects*. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/Manual/GameObjects.html [Accessed 22 Nov. 2022].

Technologies, U. (n.d.). *Unity - Scripting API: MonoBehaviour.Update()*. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html.

Technologies, U. (n.d.). *Unity - Scripting API: Rigidbody*. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/ScriptReference/Rigidbody.html [Accessed 23 Nov. 2022].

Technōs Japan Corp (2022). *Karate Champ*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Karate_Champ [Accessed 15 Nov. 2022].

Unity Technologies (n.d.). *Unity - Manual: Colliders*. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/560/Documentation/Manual/CollidersOverview.html [Accessed 22 Nov. 2022].

Unity Technologies (2019). *Unity - Manual: Unity User Manual (2019.2)*. [online] Unity3d.com. Available at: https://docs.unity3d.com/Manual/index.html.

Warner Bros Interactive Entertainment (n.d.). *1*. [online] Mortal Kombat 30th Anniversary. Available at: https://www.mortalkombat.com/en-us [Accessed 16 Nov. 2022].