Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Open Informatics

Fighting Games

Alexandra Petrova

Supervisor: Ing. David Sedláček, Ph.D.
August 2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Petrova** Jméno: **Alexandra** Osobní číslo: **475402**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Studijní obor: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Bojové hry

Název bakalářské práce anglicky:

Fighting games

Pokyny pro vypracování:

Analyzujte herní mechaniky používané v bojových hrách typu 'Street Fighter 2D'.
Navrhněte a implementujte prototyp hry využívající poznatků z analýzy a prototyp otestujte.
Vytvořte design dokument pro hru zohledňující výsledky předchozího testu minimálně s těmito požadavky: podpora pro dva hráče na jednom počítači, jednoduché AI pro tréninkovou místnost, minimálně dvě postavy s rozdílným soubojovým systémem.
Navrženou hru imeplementujte, v průběhu implementace postupujte dle metodiky UCD (User Center Design)

Seznam doporučené literatury:

[1] Raph Koster. Theory of Fun for Game Design, 2nd edition, O'Reilly Media, 2013.
[2] Simon Egenfeldt-Nielsen, Jonas Heide Smith, Susana Pajares Tosca. Understanding Video Games, 3rd edition. Taylor & Francis, 2016.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. David Sedláček, Ph.D., katedra počítačové grafiky a interakce FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **12.05.2020** Termín odevzdání bakalářské práce: **14.08.2020**

Platnost zadání bakalářské práce: **19.02.2022**

Ing. David Sedláček, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Studentka bere na vědomí, že je povinna vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studentky

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

I would like to give thanks, first and foremost, to my colleague Markéta Soukupová, who provided all of the graphics for the game, as well as her much needed and appreciated emotional support; and to my other colleague Jan Macháček, who provided us with the music and sound effects. I am also thankful to my supervisor Ing. David Sedláček, Ph.D. for bearing with me, and for giving me this opportunity to design and implement my own game in the first place. Last, but not least, I would like to thank all of the wonderful the people who helped me debug and test the game in the process.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, August 14, 2020

Prohlašuji, že jsem předloženou práci vypracovala samostatně, a že jsem uvedla veškerou použitou literaturu.

V Praze, 14. srpna 2020

Abstract

This bachelor's thesis deals with the analysis of game mechanics of the fighting game genre. I use this analysis to design and implement a prototype of my own fighting game. Afterwards, I propose a design document for the final version of the game and then implement and test it. The game is implemented using the Unity engine.

Keywords: Epic Fight, fighting game, Unity

Supervisor: Ing. David Sedláček, Ph.D.

Abstrakt

Tato bakalářská práce se zabývá analýzou herních mechanik bojových her. Tuto analýzu pak využiji k návrhu a implementaci prototypu mé vlastní bojové hry. Následně vytvořím návrh designu finální hry a hru implementuji a otestuji. Hra je implementována v Unity.

Klíčová slova: Epic Fight, bojová hra, Unity

Překlad názvu: Bojové Hry

Contents

Figures

Tables

Chapter 1

Introduction

The ultimate goal of this project is to create a fighting game with a local multiplayer support, two distinct player characters and a simple training room with an AI opponent. In order to reach this goal, I will first research the game mechanics of the fighting game genre and design and implement a prototype of what the game should look like. After testing out the prototype I will write a technical design document describing the concepts of the final game in detail, and then I will implement the game and test it.

1.1 Term Definitions

In this section I would like to explain the terms which are commonly used in the fighting game community and which I will be using further on in the text.

- When it comes to controls : back and forward - back means in the direction away from the enemy and forward means in the direction towards the enemy.
- Frames - originally refers to animation frames (but is used for 3d games as well) - often used in fighting games as a way of measuring time, i.e. if the game runs at 60 frames per second, one frame refers to 1/60th of a second

Chapter 2
Research

There are a lot of different concepts that need to be understood when creating (or even playing) a 2D fighting game. In this section I attempt to summarize the ones most important both from the implementation point of view as well as for a general understanding of the genre.

Firstly, the premise of any fighting game is two characters fighting against each other in (most commonly) three timed rounds, where the timing can be anywhere from 30 seconds to infinite (and can usually be set in game options before the match). It can be played either as a single-player, where the opponent is an AI (some fighting games have a story-based single-player, some have a training room with an AI which acts as a training dummy) or as a multi-player, where two people fight against each other, usually on the same machine. Fighting games started out in the arcades, so at first they were played with arcade sticks, later on, with the fall of the arcades and the rise of home consoles and PCs, they started to be played with gamepads and keyboards, which is where fighting games are at today [4].

2.1 Attacking

Attacking is probably the most important part to understanding fighting games in general. In most fighting games there are a LOT of different attacks [3, 9] - for basic attacks it is usually three different punches and kicks, each of which has a different animation and hit-box for a standing, crouching and jumping attack, then there is a throw attack, several (even tens of) different specials and, of course, combo attacks.

For each attack, be it a basic attack or a special, there are three distinct attack phases [6]:

- **set up** refers to the beginning of the attack animation (i.e. when the character starts lifting their leg/weapon etc.), it usually lasts around 5 to 10 frames for normal attacks (for specials it's typically longer)

- the actual attack, also called the **active frames** is the part of the animation when the hit-box comes out (it's the point at which the attack can connect with and hurt the opponent), it usually lasts shorter than

the attack's set up for normal moves (it can last a whole variety of times for specials though)

- **recovery** is the part after the actual hitting is done (i.e. the character putting their leg back down from a kick), it generally lasts about twice the amount of frames of the cost of the attack's set up

Also, when it comes to attacks in general, they can be divided into quite a number of different types depending on how they relate to countering the opponent; the one worth mentioning is the anti-air attack - an attack which hits the opponent out of the air when they are jumping, meaning the hit-box of the attack is above the character's hurt-boxes. [2]

2.1.1 Normal Attacks

Each character has the same set of normal moves (also known as normals), although the animations and the frame data in general can be very different for each character. The standard set of normal moves includes three different kicks and punches (light, medium and heavy), each one of these being bound to a single key (be it on a gamepad or a keyboard) and each one having a different animation and hit-box depending on whether the character is currently standing, crouching or jumping.

All normals follow the attack phase pattern described above in 2.1, except for the jumping normals, which have no recovery phase.

As for the speed of normal attacks, they usually last from 15 to even 30+ frames [8]. Light attacks are faster but weaker, whereas heavy attacks are slower but more powerful and medium attacks are somewhere in between. The reach of the attacks is usually fairly short, normals in most fighting games are short-ranged, however, the reach of the heavy attacks tends to be a tiny bit longer (typically by just a few pixels) than that of the light attacks.

Some characters can also have so called command normals, which are performed by pressing a direction and a button - for instance, pushing forward and a medium punch button results in a different animation then when only pressing the medium punch button. Command normals tend to be slightly faster than the common normal attacks. [9]

2.1.2 Throws

A throw (also sometimes referred to as a grab) is a unique type of close ranged attack, usually performed by pushing two buttons at once (for instance a light kick + light punch button), and resulting in (if successful) the character physically throwing the opponent to the ground and the opponent taking damage. If a forward or backward direction is held while throwing, the opponent can be thrown forward or backward.

Since throws were invented as a counter to blocking, its main feature is that it cannot be blocked [5]. However, the opponent can counter it by a throw of their own (also known as a throw tech); in case the counter throw is

successful, neither character is thrown and neither character takes damage nor goes into a stun. In some games, a player can also interrupt the opponent's throw by an attack [9].

Each character (generally) has one throw attack. Some can also have command throws though (i.e. doing a full circle motion and pushing a punch button) which are often impossible to counter.[1]

2.1.3 Special Attacks

Special attacks are where the different archetypes of characters come into play [11] - this is what makes the characters diverse and their play-styles different. These attacks are typically a lot more powerful than normals but they require a more complex input (i.e. a combination of different buttons to be pressed either at the same time or quickly in succession), the most common combination is some complex motion on the joystick (alternatively the arrows or the number-pad on the keyboard) and a punch or a kick button. Some of the common combinations are a quarter-circle (i.e. down, down-forward and forward or down, down-backward and backward), a half circle, a zig-zag motion or a full circle [3]. It is often argued that this complexity of inputs is in large part what makes fighting games so inaccessible to the general public, which is why in some games, there are also alternative, simpler inputs which can be used to trigger some or all of the special attacks, alongside the more complex inputs [7, 9].

In some games, the strength and speed of the special attack can also depend on which attack button was pressed (i.e. was it a light, medium or a heavy attack button) [9], in others the strength of the button pressed makes no difference.

There are also other types of specials. A charge attack is performed by holding a direction and a button, the direction button has to be held for a certain amount of time (at least a second in Street Fighter for instance [3]) to "charge" the attack for it to come out. Other attacks require to simply hold a single attack button - those are similar to charge attacks in that that the attack only comes out after releasing the button. A different sort of special is performed by tapping, i.e. pressing a single button rapidly a number of times. Typically, an attack like this is stronger the longer the button is tapped. Tapping can also be optional for some attacks. [3]

Specials can be both close ranged and long ranged; long ranged attacks effectively have no active frames - the "active" portion of the attack is the projectile they send across the screen which is also where the hit-box is.

Most special moves do chip damage[2], the amount of chip damage taken can be anywhere from 10 to 25 percent of the usual damage of the attack.

[1] An example of a character with command throws would be Zangief from Street Fighter. [3]

[2] Chip damage means the opponent takes a small amount of damage from the attack even when they block it successfully [9].

2.1.4 Combo Attacks

Combo attacks were originally invented by the players by exploiting a bug in Street Fighter 2 and became so popular that the developers left it in [7]; later, combos became an important feature of fighting games in general [10].

Combos are performed by move cancelling - meaning the recovery phase of the attack is cancelled (effectively skipped) by pressing the button(s) for another attack during the active phase of the previous attack which results in the next move's set up animation starting immediately [3]. Cancelling does not work with all moves, however. In some games normals can only be cancelled into some specials (Street Fighter [2]), in other games they can also be cancelled into some of the other normals and thus make combo attacks a lot more common (Killer Instinct [9]).

There can also be so called combo breakers (Killer Instinct [9]), which are attacks that can break an opponent's combo if timed correctly or punished by even more damage if not. In other games, combos are pretty much unstoppable, leaving the opponent to helplessly watch until the attacker messes up, in these kinds of games, however, combos are usually a lot harder to perform.

2.2 Blocking

A player can block high or low, where blocking high is generally performed by holding **back** (i.e. for the player on the left it would be 'A' or left arrow on a keyboard and left on a joystick) and blocks standing and jumping attacks, while blocking low is performed by holding **back + down** and blocks standing and crouching attacks [9]. For the block to register, the defender has to be facing the attacker.

There are certain attacks that cannot be blocked either high or low (i.e. throws, a feature introduced to fighting games as a block counter - for more info on throws see 2.1).

A successful block occurs when the enemy attack hit-box intersects with the defender's hurt-box (for more on hit-boxes and hurt-boxes see 2.4), the attack is blockable and the defender is blocking correctly. Successfully blocking an attack generally results in taking no damage (or a reduced amount of damage for specials or supers) and getting sent into a block-stun (for more on stuns see 2.3).

As for implementation, how the back button is registered depends on how far the opponent is from the defender and on whether the opponent is currently attacking - if the opponent is in an attack state and the defender is within a certain distance, the back button registers as a block, otherwise it registers as walking backwards.

2.3 Stuns

Stun in a fighting game is a state in which a character can get stuck - the character is stuck in a single animation and can't move or attack for the duration of the stun. There are two main types of stuns in 2D fighting games - **hit-stuns** and **block-stuns**. The character goes into a stun after being hit; what kind of stunned state they enter depends on whether the attack was successfully blocked (block-stun) or not (hit-stun). Block-stun usually lasts a shorter amount of time than a hit-stun but makes the character invulnerable to damage for the duration of the stun. Hit-stun sends the character into a reeling animation, while in block-stun the character is stuck in the blocking animation [2]. During either a hit-stun or a block-stun the character can be invincible to throws (see 2.1 for more on throws).

A knockdown can also be considered as a certain type of stun. When an opponent is knocked down, they can't perform any actions until their character stands up. In some games, a character can stand up from a knockdown faster if the player either presses an attack button, or mashes buttons or even shakes the controller [2]. The most common attacks to knock an opponent down are throws, some crouching normals (like heavy kick), some specials and combo enders.

2.4 Hit Boxes

There are various different types of hit-boxes but the main two are [6]:

- **hurt-boxes** - boxes which specify the area in which the player can be hurt - they are usually situated on the character's body and extremities, but not for instance on hair (see 2.1, hurt-boxes are green)

- actual **hit-boxes** - boxes which specify the area which, when intersecting with the enemy's hurt-box, can hurt the other player - it only comes out during the active phase of an attack animation and is usually situated on limbs and/or weapons (see 2.2, hit-boxes are red)

Figure 2.1: Skullgirls - Filia, idle pose

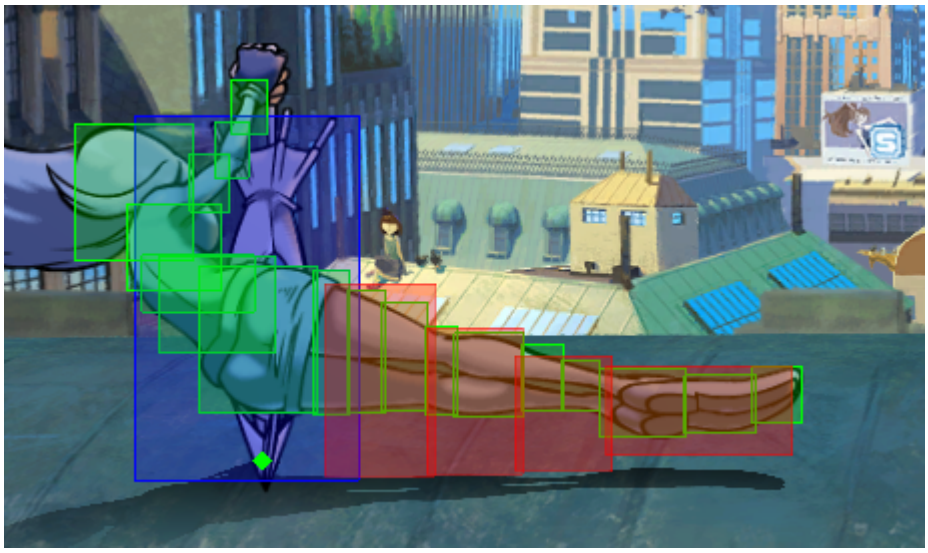

Figure 2.2: Skullgirls - Parasoul, crouching kick

The blue box which can be seen on the pictures is the collision box. The player center point (the green dot at the bottom of the characters in the pictures) is used to determine which character is standing on which side of the screen. It is useful for instance when blocking, to determine if the defending player is blocking in the correct direction.

2.5 Controls

Because fighting games started out in the arcades, the very first way of controlling them was with an arcade controller. Later, when fighting games moved to consoles and PCs, one could play them with a video game controller or with a keyboard. Although arcade controllers still exist, and are a way to play fighting games, they are very expensive and generally only the most dedicated players invest in them [7]. For that reason, I will focus mostly on keyboards and game controllers in my project.

Player movement is controlled with either a joystick or a d-pad (which is equivalent to the arrows or WASD keys on the keyboard), a player can only move forward or backward, they can crouch using the down direction and jump using the up direction (there is no other button for jump in fighting games, like for instance the space button which is the typical choice for jump in most other games [7, 2]).

Blocking an attack in fighting games is performed either by holding the back direction while standing, this is called blocking high, or by holding the back direction when crouching (or simply the down backward direction), also known as blocking low [7, 2]. Since blocking uses the same input as when walking backwards, the player only goes into the blocking state if the opponent is currently attacking. Blocking high blocks standing and jumping attacks, while blocking low blocks standing and crouching attacks[3].

When it comes to the attack buttons, the most commonly used schemes are a four-button and a six-button layout. The six button layout implements three different kicks and three different punches - light, medium and hard, while the four button one only implements two kicks and two punches - light and hard. More on this in 3.2.

[3]There are some exceptions to this rule with some characters having standing attacks which hit through a crouching block; these attacks typically have a slightly more complex input, like holding a direction and then pushing an attack button. [9]

Chapter 3

First Game Prototype

I have decided to implement my fighting game in Unity, mostly because I am familiar with the engine. Although it's not exactly the ideal engine for the development of 2D games, it still gives me plenty of options to work with. The implementation so far is only the base of what the final game should look like - the current character and their animations is only a stand-in, meant to represent the basic principles of what the fighting system will look like.

I have decided to consider the player as a state-machine[1] - which is one of the things that works naturally in Unity, because the way Unity deals with animations is through an Animation Controller, which is exactly that - a state-machine, see 3.1.

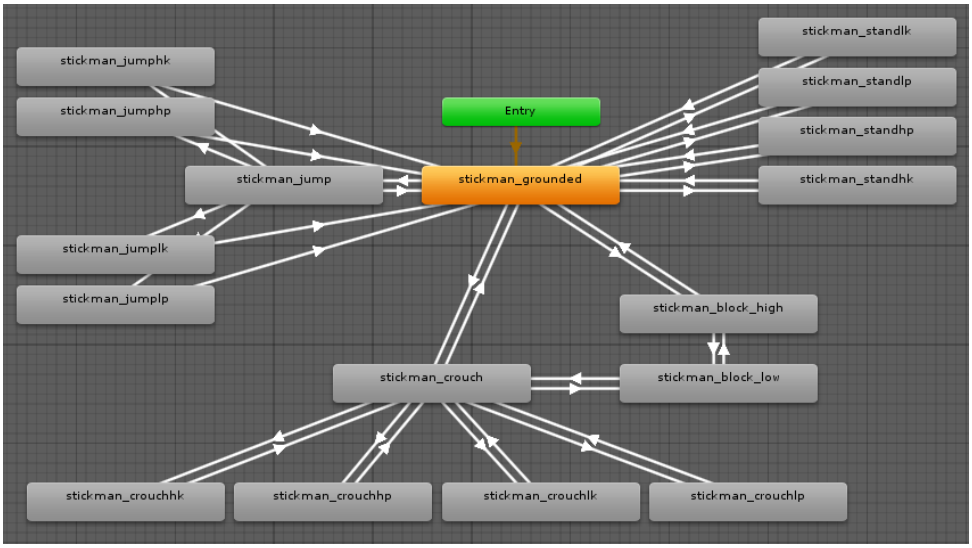

Figure 3.1: Animation Controller for the current stand-in character

One of the problems I have to tackle is the enormous amount of animations that each of the characters will have (beside movement, which includes walking, jumping, crouching and dashing, there is blocking, normal attacks, specials, etc.). So far, I've done most of the work manually, dragging each animation

[1] This idea was taken from the *I wanna make a fighting game! – A practical guide for beginners* article by Andrea "Jens" Demetrio [1].

into the Animator Controller object and creating all of the parameters and transitions by hand.

With the way the Animation Controller looks like presently (3.1) it is getting difficult to get oriented in, and the states don't even include the special attacks yet. That's why, in the future, I would like to automate this process by writing a Unity editor script which would set up the Animation Controller, together with the parameters and the transitions, for me.

Another thing I found I have to do manually is setting up the hit-boxes on characters for each animation, frame by frame. So far, I didn't think of a reliable way of making this step automatic. I did, however, create a few custom gizmos in Unity, which should make setting up the hit-boxes a bit easier - I modified the default look of a Collider to represent the different hit-boxes (see figure 3.2).

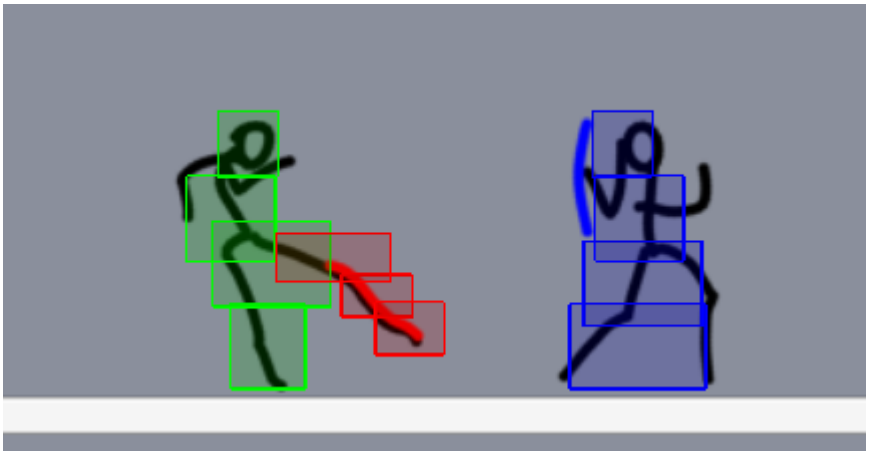

Figure 3.2: Differently colored Colliders for the hit-boxes - green is a hurt-box, red is a hit-box and blue is a hurt-box when the player is in a blocking state

3.1 Character Object

Most of the main logic so far is on the player character object, displayed in figure 3.3. It has a dynamic Rigidbody2D[2] component, a BoxCollider2D[3], an Animator[4] and three scripts - PlayerController, PlayerMovement and PlayerCombat. The PlayerController sets up a few variables key to the player object, namely the Character class (which will later be used to determine which character the player is currently playing as), the state the player is currently in (i.e. blocking, attacking), etc., and handles the damage dealt to the player. The PlayerMovement script handles movement by subscribing a Move function to the Move input action (input actions are a feature of the

[2] A component which controls physics simulations of the object and is used to move the player through script. [12]

[3] A component responsible for controlling collisions of the player with other objects. [12]

[4] A component which handles player animations, it holds a reference to the Animator Controller state-machine which contains a large part of the logic of how the player character behaves. [12]

new Unity Input System, see figure 3.4 for an illustration). The PlayerCombat script subscribes simple attack functions to their respective input actions and these attack functions set a trigger in the Animator Controller for the specific attack animation to play.

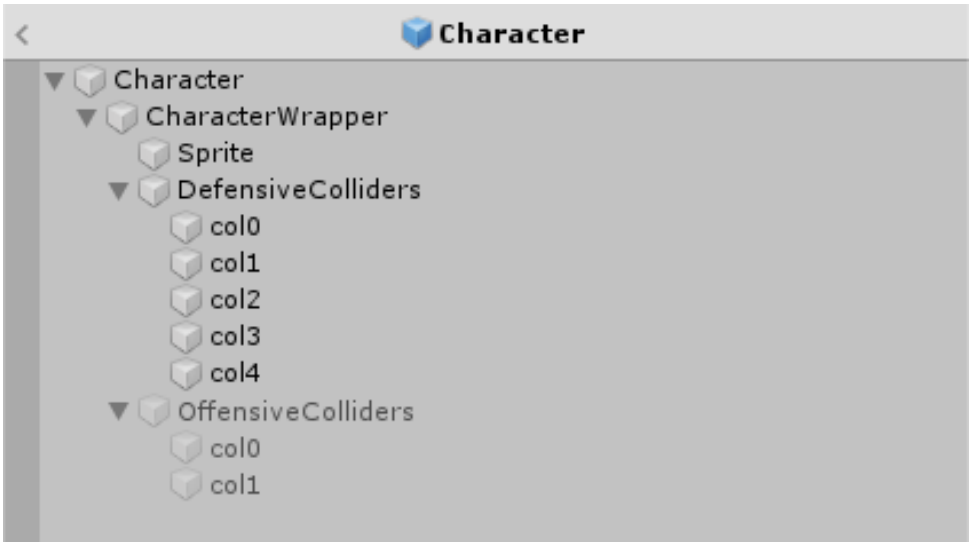

Figure 3.3: The character prefab and its hierarchy as shown in the Unity editor

The CharacterWrapper is an empty object which serves a single purpose which is offsetting the player Sprite and Colliders within the Character object during animations. The Sprite object contains the player sprite, which is controlled by the Animator Controller. The DefensiveColliders and OffensiveColliders each contain a set of hurt-box and hit-box objects respectively, each of them having a kinematic Rigidbody2D, a BoxCollider2D set up as a trigger[5] and a script that sends information about its collisions upwards to the PlayerController script on the Character object. The PlayerController then determines whether the received collision is relevant (i.e. whether it's a collision between the attacker's hit-box and the defender's hurt-box) and deals damage to the player accordingly.

There is no multiplayer so far; the opponent is a very primitive AI, created primarily for debugging purposes. The AI object contains a SimpleAIBehavior script, which puts the AI in one of a few different states (idle, blocking, attacking, etc.) according to an enum property only changeable from inside the Unity inspector.

3.2 Player Input

For the input implementation I have decided to use the new Unity Input System [13] (which is only included as an official package as of writing this

[5] A trigger does not register a collision with a Rigidbody, it only sends messages when the Rigidbody enters or exits the Collider. [12]

but should become a part of the engine in its 2020 version), the main reason being that it has a much better support for different input devices, and in the future, I would like my game to support playing with both a keyboard and a gamepad. So far, the game was only tested on a keyboard though.

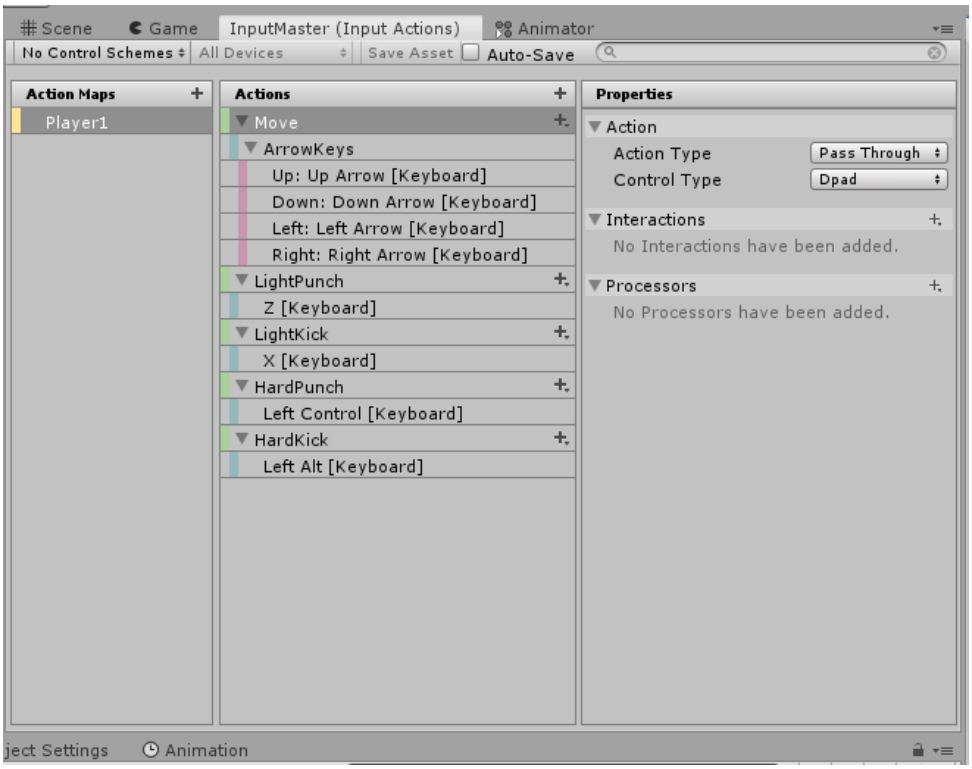

Figure 3.4: The new Unity Input System, showing the action map for Player 1

Player movement can be controlled by the arrow keys, and attacking is performed by pressing ctrl, alt, z or x. As I've mentioned in 2.5, the most common button layout for fighting games is the six-button one, I have, however, decided to use the four-button layout instead, the reason being primarily simplicity (both for the potential audience and from the implementation point of view) and the fact that the six-button layout would require a lot more animations, thus a lot more time spent on the development which, I think, would not be realistically achievable in the scope of this project.

Chapter 4

Design Document

In this part, I am going to present the concepts of all of the main features which will be included in the final game.

4.1 Multiplayer

The multiplayer will have a fairly standard form - it will be a best of three, with each round showing the current score, and a timer whose length can be changed in the game options to anywhere from 60 seconds to infinity. Each round starts with a referee announcing the fight, the characters spawning at a distance from each other, each at their end of the screen, and ends when one of the characters is knocked out (when the entirety of their health bar is deplenished). The game will only have support for local multiplayer. Players will be able to control their characters using either a single keyboard, a gamepad and a keyboard, or two gamepads. The game will detect what devices are connected and will prefer to use the gamepad input.

4.2 User Interface

The user interface is going to consist of an intro screen, which will lead to the main menu after pressing any key. The menu will contain a set of buttons - one to start the multiplayer, one to go to the training room, one to show the controls and one to show the game settings. The multiplayer button will lead to the character selection screen, which will have a character card on each side comprised of the character's name and image, and their description. The game will start automatically when both players choose their character and confirm their selection by pressing the button bellow their respective character card. See figure 4.1 for an illustration.

The menu should be controllable with both a mouse and a gamepad.

Figure 4.1: The character selection screen

The in game user interface, also known as the HUD (heads-up display), should be fairly simple; for each player it will contain a character icon (which will be showing the character the player chose to play as), a health bar and a stamina bar. It will also have a timer in the middle. An illustration of what the HUD will look like can be seen in figure 4.2.

Figure 4.2: The game HUD

4.3 Combat Mechanics

The combat mechanics will be more or less straightforward and similar to the other games in this genre. In terms of movement, the player will be able to walk forwards and backwards, jump up, forward and backward, dash forward and backward, and crouch. Each character will have a set of normal attacks - a light kick, light punch, hard kick and hard punch, which can be done when either standing, crouching or jumping, and two special attacks (described in more detail for each character in 4.4). The special attacks will drain the player's stamina; the stamina will be replenished by using normal attacks.

When it comes to blocking, the player will be able to block both high (while standing) and low (while crouching). Special attacks will be unblockable. When a player successfully blocks an attack, they will take a significantly reduced amount of damage but go into a block-stun state rendering them unable to move or attack; however, the player will take no damage for the

duration of the block-stun. If the player fails to block an attack, they will take the full amount of damage and go into a hit-stun state which again renders them unable to either move or attack. When a player is hit, regardless of whether the hit was blocked or not, they will be pushed away from the opponent slightly.

If the player is hit while in the air, they will be knocked down to the ground and go into a hit-stun state - it will behave as a regular hit-stun only with a different animation.

4.4 Game Characters

There will be two different characters in this final version of the game, one of them a close combat fighter, the other a long ranged zoner[1]. The characters' normal attacks are going to be very similar; although there will naturally be some differences in range and in hit-boxes. The main difference in the characters' play-styles will lie in the characters' special attacks, as described in the following subsections.

4.4.1 Terry

Terry will be the close combat fighter, although she will have some longer ranged flexibility in the form of one of her specials. She is supposed to fill the spot of the main character, akin to the legendary Ryu from Street Fighter. [3] Her normal attacks will be fairly fast and will have slightly longer range than normals for Eunice, to compensate for her near lack of other long ranged options. Her first special, the "Pistol Smash" will be a simple short ranged attack with a massive damage swing and a stun, whose main significance will be to punish an opponent getting too close, while rendering the user invincible for a short amount of time. Her second special will be a medium to long ranged attack called "Tear Gas Execution", an attack during which she throws a tear gas grenade which explodes when hitting the ground, dealing a fairly high amount of damage and stunning the opponent.

4.4.2 Eunice

Eunice, as mentioned earlier, will be our zoner. Her normal attacks will have a relatively short range, meant to be used as a last resort. Her main strategy should be to keep the opponent at range by using her special attacks, and escaping by dashing when the opponent gets too close. Her first special attack, the "Tyre Burst" is a projectile attack which sends a tyre rolling across the battlefield. The tyre explodes when connecting with the opponent's hurtbox, although the opponent should be able to jump over it. Her second special will be another long ranged attack called "Crying Rain" it will be a projectile with a rather large hitbox covering most of the screen before the character,

[1] A zoner is a character "whose core strategy relies on the use of projectiles to keep opponents at bay" [11]

travelling slowly across the screen towards the opponent. The opponent will only be able to counter it with a "Crying Rain" of their own or a different short to long ranged special ("Tear Gass Execution" in the case of Terry). Countering the attack in this case means ending both the original and the counter attack's animation which (usually) results in neither of the characters taking any damage.

4.5 Simple AI

There will be a training room in the game, serving mainly for the player to try out what each of the characters can do. In the training room the player will have an opponent in the form of a simple AI. The AI will behave similarly to a fairly experienced player, only it will have two different difficulties. The behavior for the harder difficulty AI should be as follows:

- when the opponent is attacking block the attack using the correct block type (i.e. block low if the opponent uses a crouching attack, block high if the opponent uses a jumping attack) and respond with an attack of your own
- when the opponent blocks an attack, your next attack should be an attack type that would break the type of block they have just used (e.g. if the opponent blocks an attack high, use a crouching attack next)
- when the opponent is close and jumps, use an anti-air attack (see 2.1 to find out more about anti-air attacks)
- if the opponent is far away randomly choose whether to move towards them or to use a long ranged attack (if a long ranged attack is available)
- if the opponent is in range for a medium ranged attack, use a medium ranged attack if available, similarly with short ranged attacks
- if pushed against a wall, jump over the opponent and attack them from behind

The easier difficulty AI will be fairly similar, with the only differences being the following:

- never block opponent's attacks
- do not attack as often
- when pushed against a wall only jump over the opponent, do not attack them from behind

The player will be able to chose the AI difficulty inside of the Options menu in the game.

Chapter 5
Implementation

5.1 Character Object

The character object and its hierarchy have stayed very similar to what they looked like in the game prototype (described in detail in section 3.1), see figure (5.1) for an illustration. The main changes are as follows: I have replaced the box collider with a capsule collider in order to avoid players sometimes getting stuck after one of them tries jumping over the other. Because I have decided to handle player input a little differently from what it looked like in the game prototype, the Character object now contains the native Unity Input System component called PlayerInput (a more detailed description of this component can be found in section 5.4) which sends messages to the Character GameObject when a certain action is performed. The PlayerInputController script is responsible for listening to these messages and calling the respective functions in other scripts on the Character object. Another new script, the PlayerAudioController, contains a set of functions to play player specific sounds (for instance the sound effects for when the player takes damage in combat and so on).

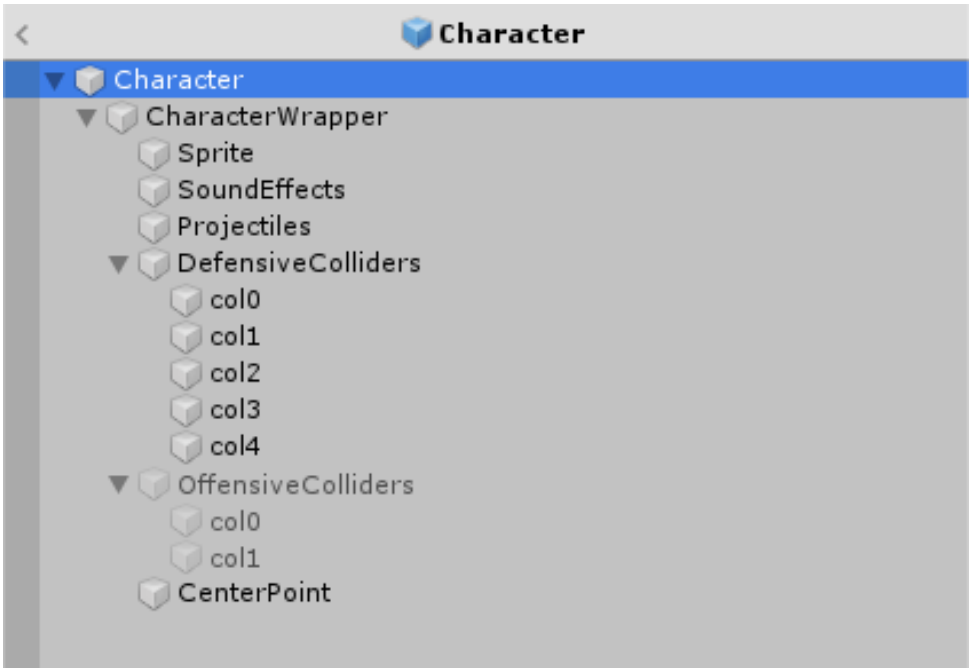

Figure 5.1: The character prefab and its hierarchy as shown in the Unity editor

There are also a few new objects in the Character hierarchy. The SoundEffects object contains two AudioSource components which are controlled by the PlayerAudioController script. The Projectiles object is an empty object, whose only purpose is to determine the Transform position when spawning projectiles in combat. The CenterPoint object determines the position of the player center, which is used when computing the distance of the players, and to establish which way each player should be facing.

Since a lot has changed inside of the core scripts on the Character object, I am going to describe these scripts in more detail in the following subsection.

5.1.1 Core Character Scripts

The main function of the PlayerController is to hold all of the information pertaining to the character the player has chosen to play as, as well as some information about the current state of the player. The Character class, whose instance is saved in the PlayerController, contains the character name, the amount of health it has, the sounds associated with the character and a list of its Attacks. The Attack class then contains the animation state name of this attack, the characteristics of the attack (like the attack type, range, damage, stun duration and so on) and a list of sounds one of which is to be picked at random when performing this attack. It also has a function which sets up a default attack, with its characteristics depending solely on the name of this attack's animation state.

The PlayerMovement script stores variables pertaining to player movement such as whether the player is currently crouching or jumping, or the player movement speed. It handles player movement by modifying the velocity of

the Rigidbody2D for walking forwards and backwards and leaves the rest (jumping and dashing) to the Animator Controller by setting some animation triggers. PlayerMovement also handles blocking, simply because it is done by pressing the backward button; the player goes into the blocking state if they are pressing the backward button and the opponent is currently in an attacking state.

The PlayerCombat script defines a couple of variables describing the player combat state (i.e. whether the player is attacking, blocking, etc.). It also contains methods for attacking, spawning projectiles, sustaining an attack and taking damage. Attacking, again, simply means setting an animation trigger and the player combat state to attacking. Spawning a projectile creates a projectile object in the scene and adds a force to it which moves it towards the other player. The spawned projectile object has the ProjectileController script attached; this script defines how much damage the projectile does, and handles the projectile collisions. On colliding with the player the projectile deals damage and is destroyed; it is also destroyed when colliding with another projectile or with a wall. The method handling sustaining an attack first determines whether the attack was blocked successfully and then subtracts the appropriate amount of damage from the player's health pool and applies a knock-back force to the player.

5.2 Character Animator

The Animator component of the Character object contains the entire logic of the character behavior contained within its Animator Controller state-machine. It holds a list of parameters responsible for deciding when to transition form one state to another, like the player's current speed, whether the player is currently crouching or what attack did the player currently trigger. Each animation state contains a reference to its respective animation clip. The animation clip controls most importantly the character's current sprite and its offset within the Character object, and the current size and offset of its hurt-boxes and hit-boxes.

Some of the animation states have StateMachineBehaviors[1]. The TriggeredAnimationBehavior is added to every non-looping animation state - it cleans up (i.e. resets) all of the triggers and resets some of the player state variables (like whether the player can take damage) when exiting the state.

Because some of the player movement (namely jumping and dashing) is performed by the animation sprites, rather then by moving the Rigidbody, there needed to be a way for the Rigidbody to catch up after one of these animations is finished; this is solved by adding the ShiftPositionBehavior to the affected animation states, which figures out the offset by which the animation sprites move during the animation and shifts the Rigidbody's position accordingly when exiting the animation state.

[1] A StateMachineBehavior is a script which can be added to any animation state; it can be used to modify the behavior of the state throughout its duration. [12]

Every attack animation in the state-machine has the AttackBehavior component. When entering the state it sets the current attack variable in the PlayerCombat script to this attack; it is also responsible for calling the methods in the PlayerAudioController which should play a sound at the beginning of the attack.

The StunnedBehavior is added to both the blocking and the hurting animation state in order to handle the length of the stun duration after getting hit. The behavior thus first determines what the stun duration should be, based on the opponent's current attack, and is then responsible for setting a trigger which exits the state when the stun time is up.

5.3 Character Generation

Figuring out how to automate as much of the character generation as possible was one of the greatest challenges when designing this game. However, in the end, I have managed to automate almost everything about it, including generating all of the different colliders, and fit it into one giant Unity script called GenerateCharacterPrefabWindow which can be run from inside the Unity editor. It takes in as parameters the name of the character to be generated, the name of the folder containing the character sprites, and the character initials, and returns the created character as a GameObject with an Animator Controller (described in detail in 5.2), all the necessary scripts and an entire hierarchy of child Gameobjects (described in 5.1).

First, there are some steps which need to be taken for the character generation script to work properly. Each character's sprites come in a separate folder with the character name, the name of this folder is passed into the script as a parameter from the Unity editor. The main folder contains more folders, each one representing one animation and containing the animation frames numbered in ascending order. The animation frames' folder naming convention is "anim_<character initials>_<animation name>", where character initials is a parameter passed into the script and animation name is one of several predefined names denoting the different character states (i.e. idle, crouching, jumping, etc.). The animation frame sprites follow the same convention with a "_<frame number>" added at the end. For the attack animations, the attack active frames are denoted by an additional "_act" at the end of the name of the frame; some attacks also have an additional "_up" suffix - this indicates that it is an anti-air attack (see 2.1 on the description of what an anti-air attack is). There are also some additional parameters that can be changed, see figure 5.2 for an illustration. The sprites' properties need to be slightly modified before the character generation can begin (namely changing the sprite pivot to bottom center and the filter mode to none), however this is done automatically inside of the character generation script.

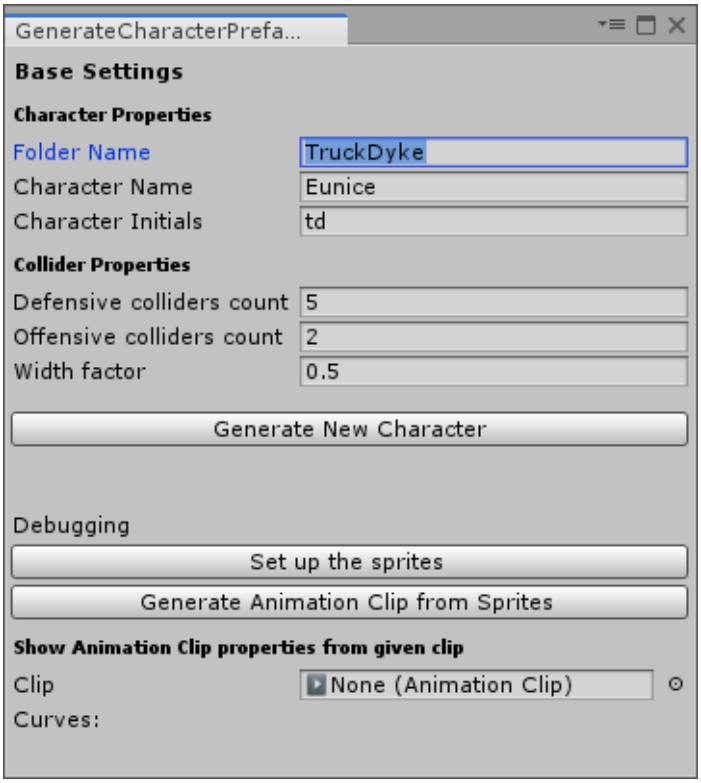

Figure 5.2: The character generation window inside of the Unity editor

Now, to the character generation itself. The GenerateCharacterPrefabWindow script first creates a new character as a GameObject, adds the required components and sets up its hierarchy (the hierarchy is described in 5.1). The character generation process is finished by saving the GameObject as a prefab[2] into the project's Assets folder. The most important component here is the Animator with its Animator Controller, the generation of which I will describe in more detail in the following section.

5.3.1 Generating the Animator Controller

First, the AnimatorController object is created and saved into the Assets folder of the project. Then, parameters are added to the controller which will later be used to trigger transitions between states. Next, the root state machine is created and populated with animation states and, lastly, the animation transitions are set up.

The animation state generation is where the main logic of each character state is set up and where the animation clip for the respective state is created. Creating the clip consists mainly of setting up the animation curves; the script creates a curve for the animation frames, for the main collider, for the position of the character center point, for the hurt-boxes and hit-boxes (which are also represented by colliders) and for the offset of the character wrapper.

[2] A prefab is a reusable asset which stores a GameObject with its components, property values and hierarchy. [12]

When setting up the main collider curve a separate method is called to not only determine the main collider's size and offset, but to also collect information which later helps set up the hurt-box and hit-box colliders.

This method first creates a list of vertices of the sprite of the current frame; it does this simply by temporarily assigning a PolygonCollider2D to the sprite object in the character hierarchy and retrieving the collider paths. It computes the width and height of the desired capsule collider and its offset from these vertices - this is done in accordance with the following simple algorithm:

1. order the list of vertices by its x coordinate in ascending order
2. save the minimum and maximum x values (*minX* is the x coordinate of the first element of the list, *maxX* the last)
3. repeat this process for the y coordinate
4. the size of the collider is equal to *maxX* - *minX* in width and *maxY* - *minY* in heigth
5. the offset of the collider is equal to *minX + (maxX - minX) / 2* on the x axis and *minY + (maxY - minY) / 2* on the y axis

This method then returns the size, offset and the list of vertices in a struct. The struct is then used to compute the sizes and offsets of all of the other animated properties mentioned above. The hurt-box colliders are generated by splitting the vertex list into the same number of parts as the number of colliders to be generated (a parameter which can be set in the character generator window) and using the same algorithm defined above to figure out their sizes and offsets. The hit-box collider is computed using a very similar method, only with some restrictions on how far the collider can stretch. The character wrapper curve offsets the wrapper in order to shift the character center position at the origin of the character object.

The hurt-box and hit-box collider generation is not perfect; it works for all of the normal attacks sufficiently well (see figure 5.3), however, when it comes to special attacks, the variation in where the hit-boxes are supposed to be is too great to efficiently automate it, therefore the colliders generated for the special attack animations have to be corrected manually.

5.3.2 Other Utility Scripts

There are some more minor utility scripts complimentary to the character generation. The SpriteVertexGizmos script is used to show the vertices of the sprite of the object on which this script is placed, in the form of sphere gizmos situated at the positions of the vertices inside the scene. It also displays some information about the vertices in the form of a text object onscreen.

The ColliderGizmoColorModifier is a script which can be placed on any object with a collider (the collider can either be on the object directly, or on

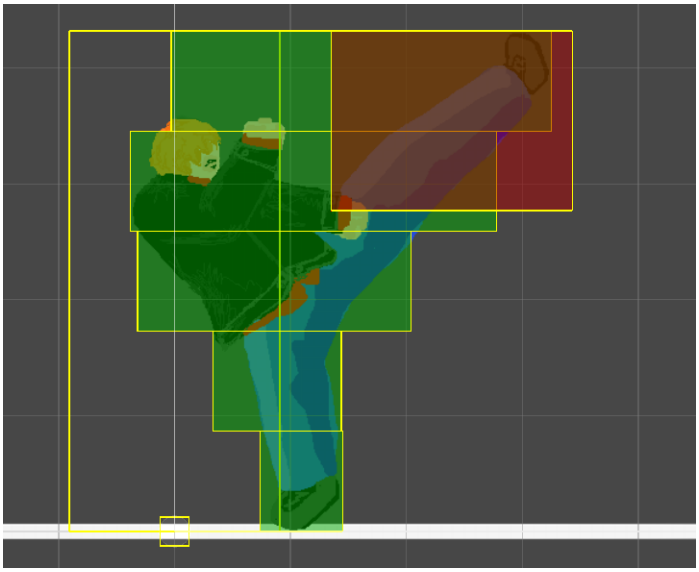

Figure 5.3: The generated colliders for one of the frames from the active phase of the hard kick attack animation, hurt-boxes are displayed in green, hit-boxes in red

one or more of its children). It makes the collider gizmo more easily visible in the scene and can modify its color depending on the passed in parameter. I have used this script for displaying the character hit-boxes in red and its hurt-boxes in green in the Unity editor, to better illustrate the difference between the two types of colliders (see figure 5.3).

There are a few utility scripts in the project which simply store data - they are static classes with readonly variables, storing things such as animation state and parameter names (CharacterAnimationUtility), default character attack values and names of the objects in the Character prefab hierarchy (CharacterUtility), or general data like object tags (Utility).

5.4 Player Input

For handling player input, I ended up using two of the new Unity input system native scripts: the PlayerInput and the PlayerInputManager. The PlayerInput is a wrapper around the new input system which takes care of managing input actions [13] - as was already mentioned in 5.1, it does this by sending messages to the GameObject it is a component of. It takes in as a parameter an input action asset (an illustration of which can be seen in section 3, in figure 3.4). I have three different input action assets in my project - one for handling input from the user interface, and two for handling input when in combat - one for gamepad and the other for keyboard input. The combat input action assets contain the definitions of all actions the player can take during combat, that is movement and attacking, and their respective keyboard or gamepad bindings. One of these assets is passed to

the PlayerInput component attached to the Character object when spawning the player, and whenever a key is pressed which corresponds to one of the bindings defined, it sends a message to the Character object about what action was triggered by the binding. The PlayerInputController script listens to these messages and triggers the according functions in other scripts on the object, as already mentioned in 5.1.

I use the PlayerInputManager script for handling multiplayer input, i.e. the joining and leaving of players [13]. I have set this component up to allow me to join the players manually; which is done from the PlayerManager script (see 5.5.1 for more details on player spawning).

5.4.1 Controls

The user interface can be controlled by both the mouse and by a gamepad, the only menu where the gamepad does not work is the character selection screen (because I was unable to figure it out in time). The confirm function on the gamepad is either the select button or the south button, while the cancel function is bound to the start button.

When it comes to combat, the **gamepad** controls are more or less conventional; player movement is done by moving either the left joystick or the d-pad, where walking is on the x axis and jumping and crouching is on the y. The player can jump up by pressing the up direction, forward by pressing the up-forward direction and backward by pressing the up-backward direction. Crouching is done by holding the down direction. Blocking high is performed by holding the back direction and blocking low by holding the down-back direction. The player can also dash forward and backward - this is done by tapping the desired direction twice. Attacking is controlled by the four buttons on the right - the north button is for light punch, the west button for light kick, the south button for hard kick and the east button for hard punch. Special attacks are triggered by holding one of the gamepad trigger buttons and pressing either the button for hard kick or hard punch.

The **keyboard** controls are bound as follows. For player one, movement is done with the WASD keys and attacking with I, J, K and L keys (I is for light kick, J for light punch, K for hard punch and L for hard kick), while the Shift key serves the purpose of a gamepad trigger. For player two, movement is on the arrow keys and attacking on the NumPad with 8 being for light kick, 4 for light punch, 5 for hard punch and 6 for hard kick. The 0 button on the NumPad serves as a trigger.

5.5 Game Manager

The GameManager is an empty object which contains a collection of scripts responsible for handling various elements of the game. The main script on this object is of the same name; it is responsible for setting up the multiplayer and the training room, as well as cleaning up after the game is over. It begins

and ends the combat rounds, stores the players' current score, and decides which player wins.

The following subsections discuss the other manager scripts, and their respective functionalities, in detail.

5.5.1 Player and AI Spawning

The spawning of characters into the scene is handled by two scripts: the PlayerManager and the AIManager. The PlayerManager is there to handle the player spawning. It first determines what inputs will the player being spawned be able to use to control the game; it does this by querying the Input System on what devices are presently connected and then passes the appropriate input action asset connected to the preferred device into the PlayerInputManager (see 5.4 for more information on how player input is handled). Gamepad and joystick input devices are preferred before the keyboard. The player spawning method also establishes the player's connection to the health and stamina bar in the HUD, and then calls a PlayerController method to reset the player and place them at the appropriate spawn point inside the scene.

The AIManager handles spawning of the AI opponent for the training room mode. It works similarly to the PlayerManager's spawning method except it, of course, does not handle input. Instead of the scripts to handle input which are present on the player, the AI has an AIController component attached defining the AI behavior which depends on the selected AI difficulty. There two difficulties, easy and hard, and a "punching bag" mode in which the AI does nothing. The easy and hard difficulties follow the behaviors described in 4.5.

5.5.2 User Interface Manager

The game's user interface is controlled by the UIManager script; its methods are responsible for switching the different interfaces, from the intro screen, to the menu, to the game HUD. It also has two built in InputActions[13] - one for handling the any key input, which enables the player to exit the intro screen when any key is pressed, and the escape key input, which allows the player to stop the game by pressing it and going to the pause menu. There are some helper scripts which manage smaller parts of the UI; these are the MenuManager, the PlayerSelectionManager and the AnnouncementManager.

The MenuManager is a component added to the main and the pause menu Canvases. Most of the methods which are called when clicking the buttons from these menus can be found here. These methods include showing the Options or the Settings menus, as well as going to the character selection screen, where the CharaterSelectionManager takes over. The CharaterSelectionManager simply controls the switching of the character cards and descriptions, and informs the GameManager to start the game when both players are ready.

The AnnouncementManager is situated on an object called Announcements, which has an Animator and an AudioSource component. It controls the

announcement text pop ups at the beginning and end of the round (such as what round it is, or which player won); these pop ups are simple animated text objects, children of the Announcements object in the scene. It is also responsible for playing the referee voice over which goes along with the announcements.

5.5.3 Audio Manager

The AudioManager script, which sits on an otherwise empty Audio object, is mostly responsible for storing the game sounds which are not specific to a given character; this includes game music, various sound effects and the referee voice lines. Regarding its methods, the script itself only contains two - one for switching to the game music and one for switching to the combat music, which is done by changing the AudioClip in a child object with an AudioSource component.

5.5.4 Camera Manager

The CameraManager script is situated on the main camera object. It controls the movement and the zoom of the camera, which is directly dependent on the position of the players in the scene. The CameraManager finds the center of the player positions in the world and moves the camera so that it always aligns with this center; it then computes the player distance and sets the orthographic size of the camera so that both of the players fit comfortably within its bounds. The camera has a maximum zoom value which it never exceeds; this is ensured by two invisible boundary objects on each side of the screen. These objects contain a collider and the PlayerStopper script, which stops the player from moving in the direction of the boundary when it enters the collider, in order for the player to not push the boundary away; the only thing that can control the movement of these boundaries is the CameraManager.

Chapter 6

Play Testing

There were two rounds of play testing for this game, and both took place in the final stages of the game.

6.1 Round One

In this play testing session, the state of the game was as follows: there was only a single character (Eunice), there was no game sound, there were no special attacks, only normals and there were plenty of bugs. The testing session was divided into several days, each day it was tested with a different set of people (usually only two, so the testing approach had the opportunity of being more individual) and the game was continually being edited in between.

On the first day, we discovered a bug where the player character would sometimes dash off-screen and be stuck there - this turned out to be an issue with the way collisions are handled in Unity and the fact that when dashing the collider of the character changes rapidly in each frame; the issue was solved by enlarging the colliders for the walls of the scene. Then, there were some complaints about the visual of the health bars and how it was not evident which part of the bar represents the remaining health, so the design was changed. Another response was that the character walks too slowly when moving either forward or backward, so the speed of the character was increased considerably.

Before the second day, aside from fixing every bug we found on the day before, I also added a knock-back force to the player when they are hit. This time, one of the participants was actually a fairly experienced player of Tekken. First, there were some issues with responsivity of the input, although after trying out the game on the keyboard we have found it was most likely caused by the gamepads we were using. Another issue that was brought up was the pace of the game - the speed of the animations was judged to be too slow. There was also some confusion as to which player was which, because there was only one character with only one design, it was sometimes difficult to tell, especially when the players' characters switched places during combat. The third day had most of the complaints in a very similar vein, because I did not have the time to fix the issues in between. Thus, after the third day of testing, I have increased the animation frame rate from 15 to 20 to speed

up the pace of the game. Also, another design for the first character was added to differentiate when the players both choose the same character. I have also decreased the knock-back a little bit.

On the fourth day, we discovered that some of the attacks were a bit more over-powered than others - namely the standing and crouching hard kick because of its long reach. This issue was fixed by decreasing the said attack's damage. Another issue that came up was that when one of the players is pushed against the edge of the screen, there is pretty much no way out. I have fixed this by increasing the height of the forward and backward jumps so that one player can always jump over the other, which was previously impossible when the other player was standing up. The last thing we discussed was that after the match ends it was somewhat tedious to have to go back to the main menu and the character selection screen again to replay the match; therefore, instead of going back to the main menu, I have added a rematch button at the end of the match which restarts the match with the same characters that were chosen initially - the players can still choose to go back to the main menu to change their selection instead.

The last day of our first round of testing, we had another participant who had some experience with fighting games, namely Super Smash Bros. A bug that we have only discovered then was that the player could attack when jumping straight up but not when jumping forward or backward. This issue was fixed simply by creating some additional transitions for these jump states in the Animator Controller. Another minor bug was that the player character would sometimes slide when crouching instead of staying still - this was fixed by setting the RigidBody's velocity on the x axis to 0 when crouching. Another thing that was brought up concerns the keyboard layout of the attack controls, so I changed it from keys I, O, J, K to be on the keys I, J, K, L (and similarly for the attack controls on the NumPad, from 8, 5, 9, 6 to 8, 4, 5, 6) which was suggested as a more natural layout.

6.2 Round Two

In the second round of play testing, the game was more or less finished. The testing session took place in one day with a larger group of people. At this stage, the game already had two characters, an AI training room, sounds and special attacks.

There were a few gameplay complaints - one had to do with the camera - its zoom was considered too close and its movement not smooth enough. I have fixed the zoom issue simply by increasing the minimum and maximum allowed zoom of the camera. I was, however, unable to implement the camera smoothing. Another issue had to do with game balance - namely, the Terry character turned out to be a bit over-powered, because some of her animations are a lot faster than for Eunice, yet her damage output was the same - this was fixed by reducing the damage on some of Terry's attacks and by slightly decreasing her life total. There were also some complaints about the AI using too many special attacks, so the frequency of the AI using special attacks

was significantly decreased.

There were also several bugs - sometimes, before the round had the time to end, the winning player would be able to hit the defeated player again which would result in another K.O., thus skewing the score to the winning player's advantage and ending the match early. Another bug was that sometimes the game would start running very slowly all of a sudden, which we first mistook for a performance issue, but it turned out to be caused by the fact that at the end of each round I slow down the time during the transition between rounds and when the K.O. bug happens the time does not get sped back up. Both of these issues were fixed quite easily, by disallowing the players to move or attack before the referee shouts "Fight!" and immediately after the finishing blow is dealt.

Chapter 7

Conclusion

My assignment was to research the game mechanics of fighting games, design a prototype in accordance with what I have found, then write a design document for the game and, finally, implement it.

There are some relatively minor details to the game that I did not have the time to polish, some of which are mentioned in the text. Another issue is that I had no time to implement some of the more advanced elements of a fighting game (like combos for instance). However, I dare say I have accomplished what I had set out; I have successfully managed to design and implement a game in the fighting game genre, with all of the basic mechanics that come with it and with two distinct characters.

Nevertheless, I still do not consider the game to be completely finished - I plan to continue working on it to polish it out and fix the issues I did not manage to fix in the scope of this project; I also plan to add more content to the game in the future, primarily in the form of more characters and more special attacks.

Bibliography

[1] Indiewatch, Andrea "Jens" Demetrio. *I wanna make a fighting game! – A practical guide for beginners*. April 11, 2017.
https://indiewatch.net/2017/04/11/wanna-make-fighting-game-practical-guide-beginners-part/

[2] Shoryuken Wiki,
http://wiki.shoryuken.com/

[3] Street Fighter Wiki,
https://streetfighter.fandom.com/

[4] William L. Hosch. *Electronic fighting game*.
https://www.britannica.com/topic/electronic-fighting-game

[5] Rory Betteridge. *A Beginner's Guide To Street Fighter V*. February 17, 2016.
https://www.kotaku.com.au/2016/02/a-beginners-guide-to-street-fighter-v/

[6] Patrick Miller. *How to play Street Fighter: a fighting game primer for everyone*. July 7, 2014.
https://www.polygon.com/2014/7/7/5876983/how-to-play-street-fighter-fighting-game-primer

[7] Polygon, David Cabrera. *Street Fighter 5 guide*. April 24, 2017.
https://www.polygon.com/street-fighter-5-guide/

[8] Full Meter, a frame assistant tool,
https://fullmeter.com/

[9] Infil, *The Complete Killer Instinct Guide*,
https://ki.infil.net/basics.html

[10] Fighting Game Design Fundamentals,
http://game-wisdom.com/critical/fighting-game-design-fundamentals

[11] GameRant, Paul Disalvo. *The Ten Most Iconic Character Archetypes in Fighting Games.* March 1, 2020.
https://gamerant.com/ten-iconic-character-archetypes-fighting-games/

[12] Unity Documentation,
https://docs.unity3d.com/Manual/index.html

[13] Unity New Input System Documentation,
https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/index.html

Appendix A

Design Document

A.1 Design History

This is the less technical version of the Design Document, version 1.0.

A.2 Game Overview

It is a 2D fighting game in the style of old Street Fighter or Mortal Combat games. The difference is that the characters fighting each other are various lesbian stereotypes commonly (or less commonly, depending on how many characters we end up making) seen in mass media (like films, TV series, even video games).

Figure A.1: The game intro screen

The thing is, most fighting games tend to have a pretty high entry barrier because of their difficulty. So, since the main audience of our game is meant to be the members of the LGBT community (although ideally it should be fun to play for anyone), most of whom we don't expect to be avid fighting game fans, we intend to make the game a little less complicated than you'd

normally expect a fighting game to be (meaning less complex inputs, four attack buttons instead of six etc., see A.3 for a more detailed explanation).

A.2.1 Philosophy

This game is really a paraphrase on the old fighting games - it's not trying to achieve anything new regarding video game mechanics but rather to re-create an old, well-known formula and use it as a sort of a parody of itself, thematically-wise. The reason for that is, well; about every single reasonably playable fighting game I've seen was at least a little bit sexist. In the oldest fighting games, there was usually a roster of a variety of different types of male characters and then there was "the girl", because there was literally a single female character in the entire game, her gender was her only distinguishing quality. And then there are Skullgirls, an indie fighting game with an (almost) entirely female roster; which would be exciting if all of them weren't ridiculously sexualized. So, we want to do something a little different. Even though our game is based on lesbian stereotypes, what we ultimately want is to humanize these stereotypes, our characters.

A.3 Features

The **gameplay** will be similar to most other fighting games, we don't intend to steer away from the genre too much. It should, however, be simpler to play than most fighting games, in order to be more accessible to audiences outside of the niche fighting game community. For this reason, the game will use a four-button layout, instead of a six-button one, and when it comes to special attacks, we will focus mostly on creating ones with the least complicated inputs, like for instance charge moves (which only require the player to hold a direction and push an attack button) or quarter circle moves (e.g. a down, down-forward, forward move on the joystick/keyboard and an attack button).

There will be a **single-player** in the form of a training room with a fairly primitive AI opponent. We have considered also creating a story-mode, something simple, similar to Ryu in Street Fighter, which would also act as a tutorial of sorts. That would, however, require us to come up with a story, which has not been our focus so far. Our main focus is the multiplayer.

The **multiplayer** mode is, obviously, for two players, both of which control a single character. There are fighting games, where one can control multiple characters (in fact, at least a few characters with this option are in almost every fighting game, in other games you can even play as tag teams, switching characters in and out during the round) but we have decided against it, for simplicity's sake. Each player chooses a character they want to play at the beginning of the game and then play a best of three match against each other. The multiplayer will be local only (i.e. two players playing on one machine).

A.4 Game Characters

We have designs for four different characters so far, Terry (A.K.A. The Police Woman), Eunice (A.K.A. The Truck Lesbian), Megan (A.K.A The Good Girl) and Jennifer (A.K.A. The Psycho Lesbian). Each one is modelled after a specific stereotype commonly found in mass media.

There are a bunch of different archetypes of characters/play-styles in fighting games, and we were thinking - how do we use that to our advantage? We thought long and hard about what archetype should go with which stereotypical character and decided, in the end, to try and subvert the stereotype by pairing each one of them with an archetype that would be unexpected for the specific character. In the following section, I'd like to describe the characters, together with their backstories, and with what their role is going to be in the game.

Each section includes the card of the character, which was an illustration made for the early installment of the game, and we will probably stick to it in the final game as well.

A.4.1 Terry, A.K.A. The Police Woman

Terry is a middle-aged woman working as a detective, or a high ranking officer. She doesn't have a lot of lasting relationships, although she often falls in love with a fellow policewoman or a woman she saves during her job. Often longs for a family and takes all relationships with utter seriousness. Great friends with many of her male colleagues who she likes having a beer with.

Figure A.2: Terry

The Police Woman stereotype, which appears mostly in TV series, is almost always a barely important side character, only appearing in a couple of scenes. Her premise seems to be to diversify the cast, but without sticking out too much, without offending anybody and generally remaining completely unnoticed by the majority of the audience. So, obviously, Terry is our main character. The most common archetype for the main character in a fighting game is a "Jack of All Trades" - a character who can do anything, but doesn't particularly excel at any of their moves. Which is what Terry is going to be, she'll have both short ranged and long ranged attacks (she does carry a gun, after all) and just all in all a well rounded move set, mainly designed to be easy to learn for a beginner.

A.4.2 Megan, A.K.A. The Good Girl

Megan is a high-school girl who is slowly realizing her attraction to women but her family or her religion forbid her from acting on it. From time to time she lets herself be lead astray by a beautiful artist, a punkgirl, or just an out-and-proud girl. When it comes to relationships, however, she usually tries to find a nice boyfriend to show to her parents.

Figure A.3: Megan

The Good Girl can often be seen as the main character of some sub-par teenage movie. She often has very little agency throughout the whole movie (although this is sometimes subverted in the end) and is generally presented as a sweet, innocent girl (who is being corrupted by her attraction to women). Megan, we've decided, is going to be the "Grappler". A grappler character is usually slow and tough, they have a lot of health points and their strongest

suit are their command throws (a special kind of close ranged attack, often performed with a very complex input motion - usually a full circle plus a punch button, which is something we're going to simplify to either a half or a quarter circle). Like Zangief from Street Fighter? That's the archetype Megan is going to represent.

A.4.3 Eunice, A.K.A. The Truck Lesbian

Eunice is a lonely woman in her fifties travelling the land. Crushed by life experiences, she is full of incredible wisdom. She becomes something of a ferryman, saving women running away from their mundane lives from their middle-aged crises, granting them the wisdom she's gained through age and travel. Devoid of sexuality, she has long given up on finding the love of her life.

Figure A.4: Eunice

This is actually a stereotype which also exists in real life (in the sense of gay women embracing the Truck Lesbian stereotype and presenting themselves accordingly); in movies or TV series, Eunice is usually a side character, often appearing in a single scene, often deliberately presented as harsh and unattractive, her sexuality only being mentioned to further other her from the rest of the cast. Eunice is going to be our "Zoner". The zoner is a character with a majority of long-ranged attacks (either projectiles, or physical) which they use to control the battlefield, the play-style is thus rather defensive. We'd like to base Eunice around throwing truck tyres at her opponent from a distance.

A.4.4 Jennifer, A.K.A. The Psycho Lesbian

Jennifer is a girl condemned by her surroundings for her queer behaviour (namely her, either outright or implied, attraction to women). She falls in love with a beautiful polite girl, a love that is often unrequited, which is somehow linked to her psychotic tendencies. Often seduces and kills men, especially if they are romantically involved with her love interest, or alternatively, seeing no other way out of her situation, commits suicide. One way or another, ends up dead by the end of the movie.

Figure A.5: Jennifer

This is probably the most popular stereotype. She appears in a variety of different versions, sometimes an unimportant side character, sometimes the nemesis of the main hero, sometimes the mysterious lady who ends up being the villain.. there are a lot of different versions of the Psycho Lesbian, which is why we still aren't completely sure about what archetype would fit her best. So far, we are going with the "Rushdown", a character that constantly puts pressure on their opponent by using their many close combat attacks.

A.5 Game World

The game world should not be particularly important in our game; it mostly serves as a backdrop, a complement to the fight. We got inspired by Street Fighter here again, and chose to make each of the scenes connected to each of the characters. An example of such a scene can be seen in figure A.6, and in the Characters section above (A.4), on the cards as a background behind each of the characters.

Figure A.6: An illustration of the scene background for Eunice

Terry, the Police Woman, will have either a crime scene or a car accident as her background. Megan, the Good Girl, will probably fight her opponents inside of a church. Eunice will definitely fight at a parking lot full of trucks, her natural habitat. And Jennifer, the Psycho Lesbian, will have her scene inside of a high school, in the the locker room.

The scenes will all be simply static images, no animations in the background like in other fighting games, because we think those are unnecessary and distracting.

A.6 User Interface

The in-game UI should be fairly simple, in order for the player to be able to see the important information right away - it should only contain the round timer, the number of the current round, how many rounds has each won so far, player's portraits and names, and their health-bars. The HUD should, again, be fairly similar to what one is used to from other fighting games, minus the various super meters, guard meters and what-not which we would rather omit, because we think it makes the UI look unnecessarily cluttered.

Figure A.7: The game HUD, very early concept

We have an early concept of the main menu interface - the screen where each player selects which character they want to fight as - but we aren't completely satisfied with it (A.8). We would like to show each characters

backstory next to each of their portraits in the selection screen. In regards to style, or the look and feel, the early concept represents our idea fairly well. The character selection screen should also include stage selection though.

Figure A.8: The early concept of the character selection screen

A.7 Music And Sounds

Music, as in a soundtrack or a musical score, hasn't really been our main focus so far; but we have agreed that we would like the score to be something reasonably cheerful and unobtrusive. As for the sound effects, however, we consider them an important part of the game. So far, in the early installment of the game, we have recorded some fighting sounds at home, they are very unprofessional and not very well edited. What we would ideally like to have as the sound effects are the actual sounds from the old Street Fighter, or just a mix of different sound from various old fighting games. The issues with this would obviously be Copyright, as well as simply getting a hold of the sound clips.

Appendix B

Play Testing Questionnaire 1

1. What was your favorite moment or interaction?
2. What was the most frustrating moment or interaction?
3. Was there anything you wanted to do that the game wouldn't let you do?
4. If you could change, add or remove something from the game, what would it be?
5. How did you feel about the character movement and combat? Was it too fast/too slow? Was it engaging? Were the animations believable?
6. How fun was the game? (scale 1 to 5)
7. How difficult was the game? (scale 1 to 5)
8. How easy was it to learn and use the controls? (scale 1 to 5)
9. How easy was it to navigate the menu? (scale 1 to 5)
10. Extra comments:

Appendix C

Play Testing Questionnaire 2

1. Část 1 - celkový dojem
   a. Co byl váš nejoblíbenější moment ve hře?
   b. Co vás na hře nejvíce frustrovalo?
   c. Bylo ve hře něco, co byste chtěli udělat, ale hra to neumožňovala?
   d. Pokud byste mohli do hry něco přidat (funkcionalitu) nebo ubrat, co by to bylo?
   e. Jak zábavná vám hra přišla? (stupnice od 1 do 5)
   f. Jak složitá vám hra přišla? (stupnice od 1 do 5)
   g. Jak jednoduché / složité bylo zorientovat se v ovládání? (stupnice od 1 do 5)
   h. Jak jednoduché / složité bylo zorientovat se v menu? (stupnice od 1 do 5)
   i. Jak se vám zdála rychlost animací? (stupnice od 1 do 5)
   j. Přišel vám zvukový podkres a zvukové efekty vyvážené?
   k. Jak silné vám přišly speciální útoky? (stupnice od 1 do 5)
   l. Jak zábavné vám přišly speciální útoky? (stupnice od 1 do 5)
2. Část 2 - singleplayer/AI
   a. Jak obtížné vám přišlo bojovat proti AI?
   b. Jak moc předvídatelné bylo chování AI?
3. Část 3 - postavy
   a. Která postava je podle vás lepší?
   b. Byl některý útok moc silný oproti ostatním? Který?
4. Další poznámky, postřehy a nápady pro zlepšení:

Appendix D

User Manual

D.1 Opening the project in Unity

The Unity project is situated in the "EpicFightBC" folder. To run it in Unity first unzip it, then run the Unity Hub, in the "Projects" tab click on "Add" and find the project folder which was previously unzipped and open it. It should now be available in the list of projects in the "Projects" tab. The project was made in version 2019.2.17f1 of Unity and should be opened in this version to ensure it will work properly. The project includes the following packages:

- TextMesh Pro
- Input System

These packages need to be installed before running the game; they can be installed by going to the "Window/Package Manager" menu, selecting "All Packages" in the top left of the editor window, finding the corresponding packages in the list on the left, selecting them and clicking the "Install" button.

D.2 Launching the build

The build only works on Windows. To launch the build simply unzip and open the folder with the name "Epic_Fight_Build" and run the "EpicFight.exe" application.

D.3 Controls

First, the keyboard controls:

- Player 1
  - Movement - WASD keys
  - Dashing - double tap A or D
  - Light Kick - I
  - Light Punch - J
  - Hard Punch - K
  - Hard Kick - L
  - Special 1 - Shift + K
  - Special 2 - Shift + L
- Player 2
  - Movement - arrow keys
  - Dashing - double tap left or right arrow
  - Light Kick - NumPad 8
  - Light Punch - NumPad 4
  - Hard Punch - NumPad 5
  - Hard Kick - NumPad 6
  - Special 1 - NumPad 0 + NumPad 5
  - Special 2 - NumPad 0 + NumPad 6

The gamepad controls are:

- Movement - d-pad or left joystick
- Dashing - double tap left or right direction
- Light Kick - button west
- Light Punch - button north
- Hard Punch - button east
- Hard Kick - button south
- Special 1 - trigger + button east
- Special 2 - trigger + button south

The menu can be controlled by the mouse or by the gamepad (except for the character selection screen) - use the gamepad left joystick to navigate and the select button to select an item. To open the pause menu when in game, use the gamepad start button.