KUMITE (iOS/Android)

C# Unity Game Documentation

Nathan Mayifuila

Table of Contents

1. Abstract **2**
2. Introduction **3**
3. Survey of Literature/Information Sources **4**
4. Requirements **7**
5. Specification, design, algorithms and data structures **8**
6. Implementation/Testing **17**
7. Critical Appraisal **31**
8.Conclusion **34**
9. Bibliography and Citations **35**
10. Appendix **36**

1. Abstract

My aim for this project is to create a fully functioning mobile smart phone fighting game which allows users to interact with characters in the game by choosing which character they play as from the main character select screen, this game will have an array of different characters to suit different user personalities as well as allowing me to gain a more diverse audience, I will be making the game's user interface very simple to use as well as nicely organised.

When it comes to character creation I plan on making each character unique with their very own unique fighting styles, different voices, different attack sounds, different animations and different animation speed. I plan on having an arcade mode so that users can continue to progress and I want different difficulty levels from beginners to experts. The arcade mode will keep track of how many points the user achieves from each fight and will total these points to compare later on against the default high score or high score set by another player.

2. Introduction

In this dissertation I will be discussing my C# mobile application, I will be covering what I have done within these last few months and I will explain in full detail what my project is and how it works. I will start off with the aims and objectives following on from the plan as well as the interim report.

2.1 Aims

I aim to create a fully functioning mobile phone fighting game, this game should be able to run on at least an iOS device however I plan on also making it multi platform later on. The user should be given the ability to choose different characters and opponents as well as stages. The opponent should be a perfectly working AI which moves and attacks the players without command and each attack must be selected at random. There should be a win and lose feature in order to make this a working game therefore each player and opponent must have their separate health bars.

2.2 Objectives

I believe the aspects within this project which will make the Kumité game stand out more when compared to the rest of the fighting games in the android and iOS app stores is the style of the game, the arcade mode is more of a survival mode rather than a tournament like other fighting games. I want to have Gameboy/Xbox like buttons on each corner of the screen as well as a pause button in the center of the screen, the reason for this is that not many fighting games have on screen buttons and joy sticks.

In my personal opinion they are much easier to use and get used to compared to the modern day swiping gestures. A few aspects which will make my game unique are features such as small cut scenes which will show which characters will be fighting next as well as which stage the fight will take place in.

Another unique feature in Kumité will be the arcade mode where each time a player wins a fight, the game difficulty will increase and the opponent in the next fight will be much stronger. I believe that uniqueness will be a great aspect for the game due to the fact that if users like a certain unique idea which no other game has then this game will stand out a lot more when compared to the other games in the app store.

I want the game to look great on whatever device it is played on so in order to do that I will ensure that the screen resolutions are well scaled automatically for whatever device the game is being played on, Unity has a build feature which will allow me to do this.

I plan on creating different stages for the game so for example there could be a stage where the two fighters are fighting on a boat or another stage where they are in an abandoned air space and many more, I want these stages to have animated backgrounds so some stages could have spectators and others could have a sea in the background with moving waves just to give it a more realistic feel.

To make different scenes I will be using one main Unity scene, however this scene will have multiple animated backgrounds, the only background to be visible in runtime will be the background that the user chose in the character select scene, all the other backgrounds will be disabled. The backgrounds I will be using will be selected from gifer.com (Website where users create and upload free gifs for other users to use) and I will be converting the gifs into MP4 using ezgif.com (a gif to MP4 converter). In Unity each background MP4 will be running in a loop which is kind of like a gif itself.

Another unique feature I will be adding is a scoreboard which records how many wins the user has and how many points they achieve, for example if the user beats the computer three times and loses twice then the scoreboard will say "User 3-2 CPU". Points will be based on the types of wins, so for knock outs the user can be rewarded 10 points, for timeout wins the user can be rewarded 5 points, for draws each player gets 2 points and for losses the user is not rewarded anything.

When creating characters, I will be using a 3D computer graphics software called from Adobe called Fuse CC which will enable me to create 3D character models, and once I upload them to the fuse Maximo website I can then create and choose animations for each character, after the animations have been created I can use a sprite baker software which I purchased in the Unity store to turn each 3D animation into 2D sprites (a **sprite** is a two-dimensional bitmap that is integrated into a larger scene, all my sprites are saved as a .png file) so I can use these 2D sprites in my 2D game.

3. Survey of Literature/Information Sources

3.1 Background research on fighting games

I took a lot of time and research for planning this project because I wanted to ensure that it will be started correctly so that marking progression using a Gantt chart that I previously created will be simple, I wanted to ensure that I understood the concept of a fighting game by researching existing fighting games and techniques on how they are created, my research began with basic mobile phone games so that I could have a feel of different user interfaces in order to generate ideas on how I wanted my UI to be like.

The first step I had to do was gather information and reviews which were already out there so I watched YouTube reviews on fighting games, I read IGN reviews as well as customer reviews in order to see why these existing games were so popular. According to my research most people who play video games on their smart phones are young people aged 12-22, most of which have claimed that they enjoy these games due to the fact that they are amusing, addictive and rewarding (in game points), I used these descriptions as goals to make my own game. The Kumité game must be rewarding, amusing, time consuming as well as addictive in a way.

I began watching and studying YouTube videos based on the popular Capcom fighting game series called Street Fighter in order to see if I could find any flaws within the game as well as if I can get more ideas to implement into the Kumité game. After my research on Street Fighter I then watched a few reviews of every title within the Street Fighter series from the first game released in 1987 and Street Fighter Turbo 2 on the super Nintendo entertainment system to the more modern Super Street Fighter IV and Street Fighter V.

Before creating a fighting game of your own you must understand how these fighting games worked, how they were made and what made them attract their audience. I read a book called "Street Fighter: The Complete History" which was written by Chris Carle in order to gain more knowledge on the subject. I then began researching the Tekken and Mortal Kombat series as well as watching videos and documentaries on both those popular fighting game titles which allowed me to gain more knowledge on the stories and characters behind each series. These fighting games inspired me to choose to create a fighting game project due to the fact I find these games interesting and I personally grew up playing these games.

After gathering more information from documentaries, books such as Street Fighter: The Complete History and Tekken Vol.1 I began planning how I wanted my game to function as well as appear on a user device, I was inspired by the Street Fighter engine from the turbo series so I decided to make a similar 2-dimensional game but for smart phones and tablets instead due to the fact that most people own smart phones and tablets in this day and age but not everyone owns a games console.

I purchased Tekken 7 and Street Fighter V on the PlayStation 4 platform so I could spend some time playing these games and figuring out ways to implement them into a mobile device. This helped me write the plan above. I realized that a lot of work goes into creating these games, from teams of developers to voice actors and directors, I decided to take it upon myself to figure out different methods of carrying out such challenging tasks and in order to carry these tasks out I needed to find a suitable piece of software as well as a programming language.

3.2 Research on game engines and why I chose Unity and MonoDevelop

The next step was to decide which game engine should I use to create this game, since I have never used a game engine or created a game before this was a very hard decision to make, this meant I had to research many different engines, read reviews and find which one suited my project the best.

Blender, Game Maker engine and Unity engine were the three game engines with the greatest amount of positive reviews. I watched a YouTube video by a user called RealTutsGML which gave me some more insight and information on both Game maker and Unity, although Game Maker is a lot easier than both Unity and Blender I decided not to choose that software due to the fact that not only is it too simplistic but the game Maker engine also has a lot more flaws such as it does not allow game creators to take as much controls as pure coding would, this means my options on some feature such as sound, animation, player movement and arcade would be very limited, it also does not teach real programming or computer science concepts which are aspects that I find very interesting.

After researching many game engines, the reason I chose Unity for this project is simply because the Unity game engine is known to run smoothly, this particular game engine will allow the game to be created, tested and ran in different environments, Unity is also great when it comes to multiplatform which gives the user the ability to make an application for both android and iOS as well as other platforms such as windows, mac and Xbox. Unity comes along with it's own built in open source integrated development environment called MonoDevelop which is great for debugging code as well as scripting. It also has an auto formatting feature which makes scripts easily readable and appear clean.

With a project as large and complex as mine it is inevitable to not come across a large number of errors which most developers will find quite challenging and difficult to fix, during my research on Unity I found out that the debugger provided by MonoDevelop will help me locate a lot of the sources for such errors using the debug class built into MonoDevelop so that I can come up with ideas on how to fix them or possibly remove them and implement different ideas.

I was aware that because this project is so large I would most likely end up writing over 15 separate programming code/scripts and possibly thousands of lines of code so I needed a game engine which will allow me to organise all these different scripts well so I that I do not lose track of anything and I find access to the scripts very easily, Unity does just that.

I planned on making a complex game with multiple different features and options, so I decided to go for a more challenging path since Game Maker is too easy to create games and it limits my options a lot. Blender and Unity were my last to options and since I am creating a mobile application I decided to go for Unity due to the fact that it gives me the ability to make my game multiplatform(iOS/Android/Windows) and Unity is known for being the best engine when it comes to 2D design as well as scripting. My final choice for a game engine is Unity, although it is one of the most complex and challenging game engines to learn I decided to accept the challenge and begin studying the software.

3.3 Deciding which object oriented programming language to use

Unity is described as “the ultimate game development platform”, I went into further research to see how games are created with Unity, I watched some videos comparing Unity to other games development software. After I decided to go ahead and use Unity I realized I needed to also choose a programming language to write scripts for my game which is where C# came in.

There are many reasons I chose C# but one key reason is simply because it is known to be cross platform. My project is a mobile phone application, if I chose to code in java I would be mainly limited to android, if I chose objective C or Swift then I would be limited to Apple/iOS. With C# I can make the application available for both operating systems as well as many more. Another reason I chose C# was because after researching it I realised that this language has many constructs which Java lacks, most notable constructs being operator overloading and delegates. My game is quite a complex game so I went for a programming language with a rich library as well as features such as simple cross plat forming.

I decided to learn more about what C# is used for and how it is used before committing to it, I found out that C# is a well known high level multi-paradigm programming language and it is known to work well with Unity so I am using C# for the backend programming in this project. I have used C# to create many scripts for different functions throughout the project. C# is a new language for me however I am confident in the language due to my java and object oriented programming background, I have spent the better half of the year studying this language.

I used pluralsight to watch videos on understanding Unity and writing code for C#. One particular series of video tutorials which I found very useful was the 2D plat former tutorial by Brian Sinasac.

Pluralsight helped me understand and pick up on Unity and C# very rapidly, I went onto the app store and downloaded a few more games made with C# and Unity, I was impressed by how powerful the Unity tool really was, the games were highly rated in the app store receiving great feedback for user interfaces.

I decided to make my own form of user interface and I have been showing it around to people to see what they think and most really like the joystick rather than directional buttons(which is what most games include for player movement), they say that it is a lot easier if the user keeps their finger on the screen rather than having to take it off to press different buttons all the time, I got this idea from my previous research of Tekken 7 on the PlayStation 4, I realised that no one complains about PlayStation 4 controllers and there is a reason, because it compliments your hands as well as the controls correctly considering the buttons are all placed in areas your fingers can easily reach

4. Requirements

The Kumité game is a mobile phone game first and foremost therefore the game must be fully functioning and ready for a smart phone device, this game must run on at least an android device or an iOS device due to the fact that those are the two leading operating systems in modern day smart phones.

After the game is fully functioning on a mobile phone the next requirement will be to add audio due to the fact that almost every game in the iOS and android app stores contain audio in them. The user interface for the game should be very simplistic so that a user of any age can see and understand how the game functions. One key requirement within the UI is colour and image as well as text, all text must be visible especially when a video is playing in the background.

Not only must all text be visible, all buttons should be visible and they should all fit the screen of the user's device which means the resolution must automatically scale down depending on what device the user is using. The user should be given options when it comes to creating new game sessions so for example they should always be able to choose how many rounds each fight lasts for and how long should each fight last. In the fight now/exhibition stage the user must be given freedom when it comes to choosing the opponent as well as the stage.

In training mode, the user must have the ability to turn music on and off as well as make the health and shield bars unlimited if they wish to do so. The fight scene must play sound effects such as punch and kick sounds so that the user will be aware of when the opponent is attacking them if they are blocking. One very important requirement is that every single character must have their own animations so that in the game they look and act differently, characters cannot share the same animations or animation controllers. Every fight button must fit the screen naturally, the pause button must freezer the game and it must also give the user an array of options such as settings, quit to main menu, resume and return to character select.

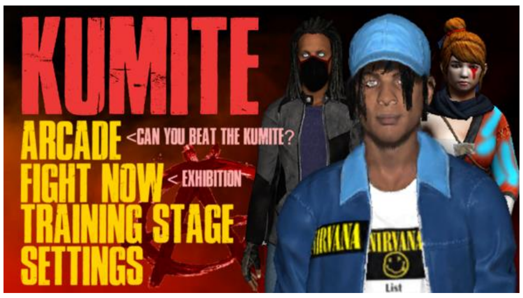
The game must also have an arcade feature above all, this will be the key aspect of the game. The most popular video games in the market have some sort of arcade, story or campaign mode. This is because users like getting to know certain characters as well as challenging themselves against other people, the arcade mode must have a level system where after every fight the user is promoted to the next level and the user will earn xp points in order to beat a default high score.

5. Specification, design, algorithms and data structures

When creating a game in Unity we need to start off with a scene, for the main opening scene I began by adding a Unity video player component onto the background and rendering an image onto it in order to attach a video clip to the video player so that it could play on the image, this was all very simple to do as it required no coding what so ever. The canvas and camera components are probably the most important components in Unity, the camera can be rendered to the user's preferences and the canvas is the area that holds all UI elements, In the canvas I used Text UI's to make a title and a few text components with button components attached to them.

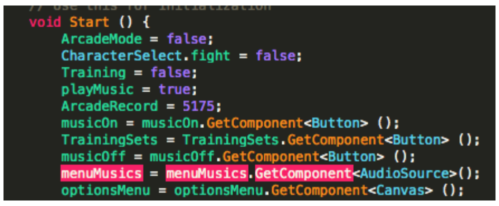
5.1 Main Menu

In the main menu scene there are four options which are Arcade, Training, Settings and Fight now as shown in the image below. The settings button opens up a translucent Unity image component with buttons attached, these settings include music on and off, fight now settings which are settings for the exhibition fight mode (turning sound effects on and off as well as music).



The training settings allows the user to have options for unlimited health meaning both the player and opponent's health bar do not decrease in percentage of fill, sound effects and music. The last option within the settings menu is control/UI overview which opens up a short video which I created by recording myself playing the game on a mobile device, this setting will show the user everything they need to know about the fight mode's UI i.e. what each button is used for and how the joystick functions. New users can use this feature to gain a better understanding of the game before proceeding to begin playing it. A Unity UI raw image component is used to act as a sprite renderer for the video player in order for the video to be played on top of the UI rather than in the background.

The menu scene contains an audio player component which is used to add music so when the game launches, it will launch with music playing in the background, the music chosen for this scene is a free instrumental from YouTube.com provided by Beats Era YouTube channel. This was downloaded in the format of MP3 which was then imported into the Unity project's assets folder.



```
    // Use this for initialization
    void Start () {
        ArcadeMode = false;
        CharacterSelect.fight = false;
        Training = false;
        playMusic = true;
        ArcadeRecord = 5175;
        musicOn = musicOn.GetComponent<Button> ();
        TrainingSets = TrainingSets.GetComponent<Button> ();
        musicOff = musicOff.GetComponent<Button> ();
        menuMusics = menuMusics.GetComponent<AudioSource>();
        optionsMenu = optionsMenu.GetComponent<Canvas> ();
```

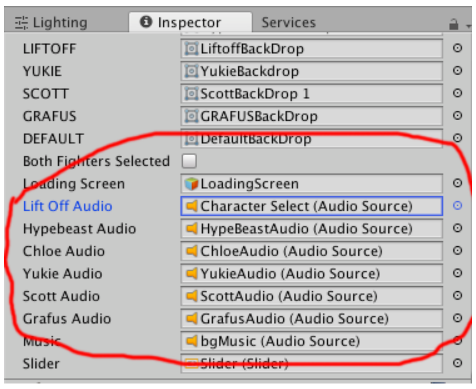
In order to implement the music into the scene, MenuScript.cs has an audio source component declared as a public attribute so that it could be manipulated within the Unity inspector later on, then it was created within the script's start function.

The GetComponent function labelled in pink means we set whatever is sent to the inspector as the component for the menuMusic we have declared above the start function, which is a default C# function which runs once and is always the first function to run by default in C# unless an awake function is declared along side it. In Unity an audio source component was created and the MP3 file was dragged into this component, the audio source was then dragged into the MenuScript which is attached to the Unity inspector, now when we run this scene, the audio should be playing in the background.

The video chosen for the background of this scene is also a free video from YouTube, this video is provided by a YouTube channel called Colors do anything. This video was

downloaded in the format of MP4, it was then imported into Unity and then a video player component was created.
An image component that the video could be rendered onto was used by the video player in order to render the video onto something.

5.2 Character Select



LIFTOFF	LiftoffBackDrop
YUKIE	YukieBackdrop
SCOTT	ScottBackDrop 1
GRAFUS	GRAFUSBackDrop
DEFAULT	DefaultBackDrop
Both Fighters Selected	☐
Loading Screen	LoadingScreen
Lift Off Audio	Character Select (Audio Source)
Hypebeast Audio	HypeBeastAudio (Audio Source)
Chloe Audio	ChloeAudio (Audio Source)
Yukie Audio	YukieAudio (Audio Source)
Scott Audio	ScottAudio (Audio Source)
Grafus Audio	GrafusAudio (Audio Source)
Music	bgMusic (Audio Source)
Slider	Slider (Slider)

In the character select scene the user is greeted with more music, this time the music playing is another free instrumental from YouTube produced by a user called KillerOnTheBeat, the video in the background of this scene is a free to use background video provided by GoodaDesign on YouTube.

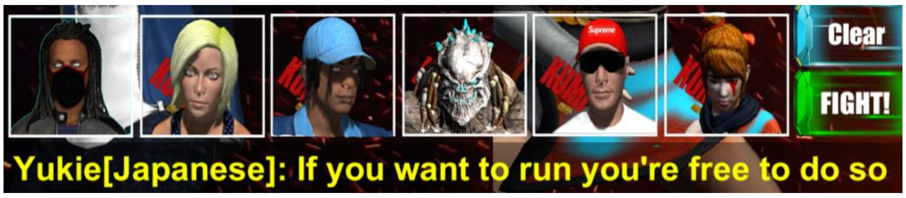
The next two components to be implemented to this scene were the music component as well as the video component using a similar method to the main menu scene although instead of a raw image for a video renderer, this scene is playing video via a normal sprite which was created on photoshop as it's main renderer.

The video player and audio source components then had their respective file formats MP4 and MP3 as their attachments before being inserted into the Unity inspector, then they were dragged and dropped into their respective Unity components.

The Unity inspector is shown in the image on the left hand side along with the music audio source as well as the audio sources for each character. In the character select script(CharacterSelect.cs) there are methods which control the music meaning it can be turned on or off, the video player has been set to play the video in a loop so that as long as the user is on this scene, the background video will continue to play.

The main goal for the character select screen is for it to be an exhibition setting, so that the user can select which character they want to use and which character they want the CPU to use. This scene is very user friendly as it shows clear instructions via text components, one component which really stands out is the 'Please select' component which prompts the user to choose a character and then choose an opponent, if neither are chosen and the user presses the next button then this text will not allow them to go past this scene and will issue a warning message.

Each character has a picture with an invisible button attached to it so for example if a user clicks on one character's face in the select bar, a picture of the character will appear on one side of the screen with the player's name and national flag on the right side of the character, the character will also say a small catchphrase(via audio player component, similar to music however it only plays when a certain button is pressed, in this case the player's button), the catchphrases are from a text to speech website called natural readers, I typed in text then used one of the many voices available and then recorded the voice and added it to the game as an MP3 file.



Above is the select bar, each character picture is a button which will play the character's catchphrase. Below that is the subtitles text component which the C1Select function also controls, here as you can see the user selected Yukie Kunimitsu as their opponent of choice meaning the audio played was this character's catchphrase and the subtitles play to this information.

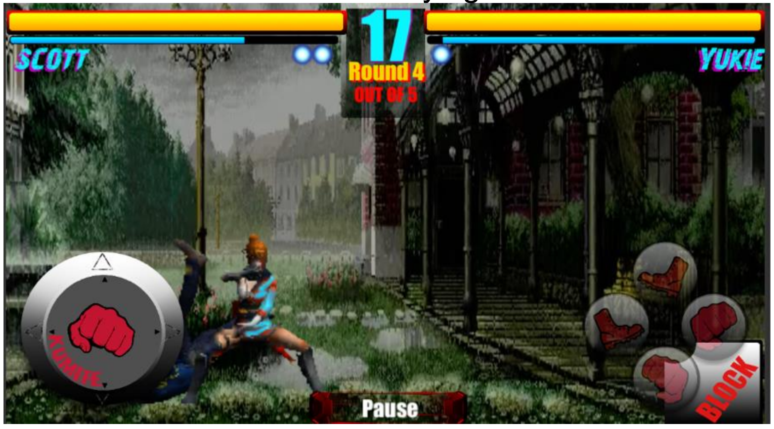
The images I chose for the background of all three of the buttons in this scene are translucent PNG images from pngtree.com, the reason I saved these images as PNG was because I wanted a part of the button to be transparent so that the user can see the part of the background video play behind the buttons, I believe that this makes the scene look so much better and it makes the user experience a lot better too. I recreated all the national flags using adobe photoshop and I took pictures of each fuse character and I imported them into photoshop so that I can make each character's backdrop so that they can appear once the user clicks on their button.

For the voice clips I used voice changers on my own voice so that I could record myself and sound different for each character as well as text to speech websites so that I could generate female voices and monster voices. After all the voice clips were recorded I added them onto these separate audio components and I attached them to the script within the Unity inspector.

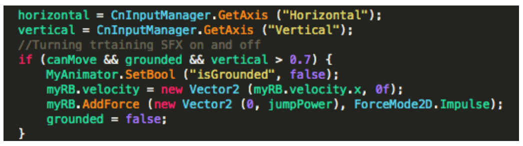
5.3 Training Stage/Fight Stage

When the training option is loaded the user is greeted with a new scene which contains a camera component just like the main scene does and a canvas which contains all the user interfaces components, in this scene I added images in the background with the help of Unity, and I added another image for the floor. In order to make sure the player does not fall through the floor I had to add something Unity calls an "edge collider" which allows me to draw out a collider on the floor and around each wall so that the player does not fall off the scene either, essentially this means I have drawn an invisible box around the screen meaning the two fighters can not escape it or exist the camera space.

The motivation behind building the training stage was so that users can practice before selecting the arcade or fight now options, they can use this training stage to learn new moves, practice old moves and improve their skills within the game by developing new tactics and techniques. The training stage will load every single character at once and then disable the ones we are not using so that there will only be two left, the reason it works this way is simply because I have programmed it to only keep the characters which were selected in the character select screen so a for loop within the SelectFrom.cs file will check which characters were chosen and it will disable the rest at runtime within the SelectFrom.cs start method (this method is a C# process method and it usually comes along with the update function), after so many attempts of having all characters disabled at the start I came up with this new logic since the old logic kept crashing the app and making Unity unresponsive due to the fact that the scripts attached to this scene were trying to access the disabled characters.



As shown in the image above, this is what a typical Kumité fight looks like, on the bottom left is the joystick which was explained earlier, it is made up of two image components which I created on adobe photoshop, in the Player.cs script I use the joystick to manipulate the vertical and horizontal inputs as shown in the code snippet to the right, this helps the rigid body move around the scene carrying the sprites as well as triggering the jump and duck animations.

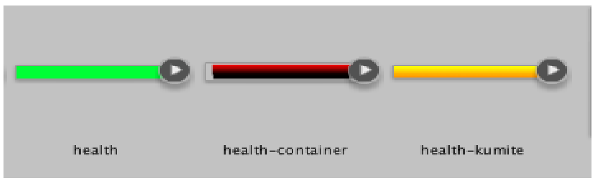


```
horizontal = CnInputManager.GetAxis ("Horizontal");
vertical = CnInputManager.GetAxis ("Vertical");
//Turning trtaining SFX on and off
if (canMove && grounded && vertical > 0.7) {
    MyAnimator.SetBool ("isGrounded", false);
    myRB.velocity = new Vector2 (myRB.velocity.x, 0f);
    myRB.AddForce (new Vector2 (0, jumpPower), ForceMode2D.Impulse);
    grounded = false;
}
```

On the right hand side of the screenshot there are five buttons, each of them trigger animations (which we will discuss later on), these buttons will also run functions in the Player.cs script for example if you press a punch button and the opponent is not blocking then it will take some damage off their yellow health bar or blue shield bar if there the opponent has any shield left. When holding down the block button this simply means the player cannot take damage and the player cannot attack. The blue bar underneath the Yellow bar for both player and opponent is the shield bar, in order to attack the opponent's health, you must first destroy the shield bar by attacking them. The shield bar(blue) and health bar(yellow) were both made the same way within the Healthbar.cs which is a superclass for the PlayerHealth.cs script and EnemyHealth.cs script (both these scripts are attached to each of their respective components).

The subclasses of the health bar are attached to their respective components in the Unity scene, so the PlayerHealth.cs script is attached to the left hand side Shield and Health bars and the right hand side shield and health bar have the EnemyHealth.cs attached to them. These two scripts with will manipulate the values via float of each of the heath and shield bars which they are attached to via the Unity inspector.

I downloaded a free health bar from the Unity asset store to test out before proceeding to create my own health bar, this health bar worked so well and it gave me the idea of having a shield above the health, since this health bar was free I decided to use some classes from it however because they were too simplistic for my game I was forced to go ahead and build on from it so I added more functions and I implemented static references all over my other scripts so that those scripts will be able to manipulate some functions I added to this health bar. I also went onto photoshop to create backgrounds and new health bar images.



Between the Health bar there are three text components, the first is the time component which is manipulated by the Timer.cs script, this will allow the GameMode.cs script to know how much time is left and when to start a new round, it also allows the user to check how much time is left. If the timer runs out, then which ever fighter has the most health left wins. Below that are two text components which are used to show the user which round of the fight they are on and how many rounds are left, the Round.cs script is in charge of updating this information and displaying them on the text components in the scene.

Below the shield bar is the character's name which comes from the CharacterSelect.cs script and the blue circles between the name and round number text on each side represents how many rounds that character has one.

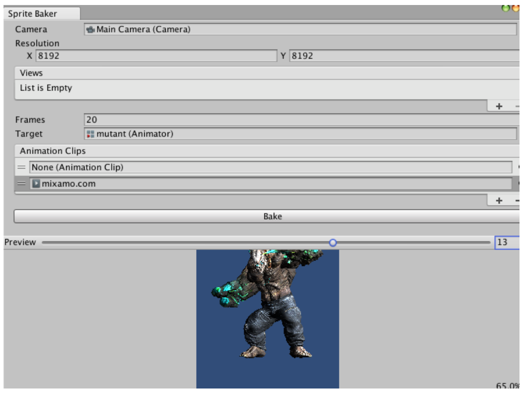
5.4 Map/Difficulty Scene

The canvas within Unity is essentially a template which holds many user interface components such as buttons and images, I added six buttons for each stage and the buttons. Each stage button has a picture attached to it so that you can have a quick preview of what the stage looks like before proceeding to click on it and each button has a function within the script telling the stage scene which background to load on the video player. As described in my objectives, I went ahead and used free animated gifs I found on a popular gif sharing website called gifer, I then converted them into videos and I rendered them onto an image before inserting them into my video player and running them as a loop.

This game has 4 difficulty levels which are easy, medium hard and true warrior, easy being the easiest of course and true warrior being very difficult. The MapDifficulty.cs class contains functions for each difficulty. The method used for difficulty is if you have chosen the hardest level of difficulty then the shield bar, this scene contains the same music and video background as the character select scene and both were implemented using the exact same technique.

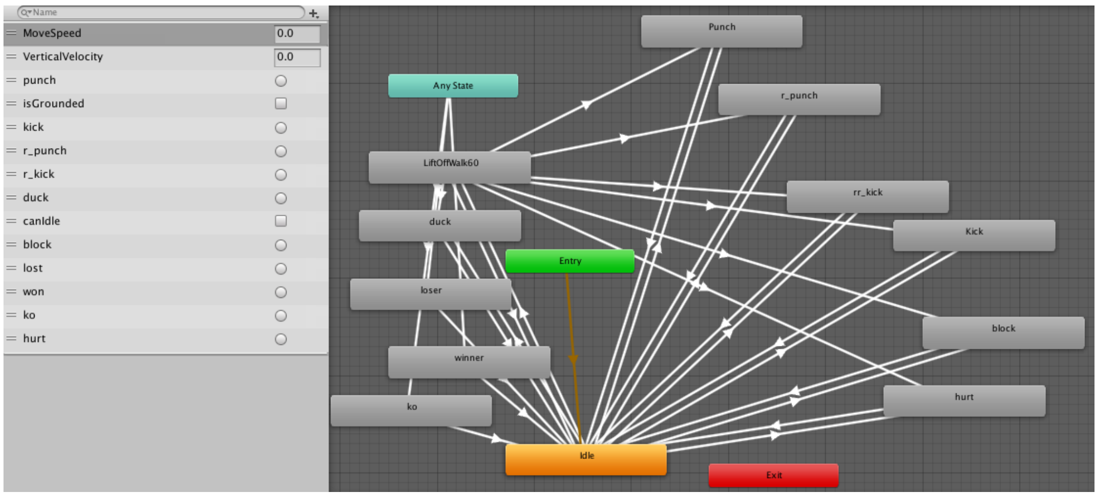
5.5 Player

When adding the player, I started off by using fuse animations, to create the player, I then purchased a “sprite baker” software (image on the right shows the software) which allows me to turn animations into sprites, I began with the idle animation which basically just has the character standing there in a combat stance.


Sprite Baker
Camera: Main Camera (Camera)
Resolution: X 8192 Y 8192
Views: List is Empty
Frames: 20
Target: mutant (Animator)
Animation Clips: None (Animation Clip); mixamo.com
Bake
Preview: 13

I created each animation using fuse and then I used the sprite baker to turn them all into sprites by dragging the animation into the sprite baker software and watching it take snapshots of each frame and then packing into a folder of sprites, after they turned into sprites I converted them into Unity animations, created an animation controller and attached it to one sprite I dragged onto the scene so that could be my default sprite which changes. After I added all the animations to the animations controller within the C# scripts I could now reference the animator the same way I would do with any other game object and within the script I could trigger conditions which will tell the animator which animation to run, Each character has 14 conditions (as shown on the left of the image below) and 15 states (rectangles shown on the right).

The states contain animations attached to them and these states are triggered by the conditions on the right which will be set within the script whilst the game is running. For example, if the player presses the left kick button, the condition “kick will be set to true” meaning the character will go from the default orange “Idle” state to the kick state and then back to the idle state (which is set to loop).


MoveSpeed	0.0
VerticalVelocity	0.0
punch	○
isGrounded	☐
kick	○
r_punch	○
r_kick	○
duck	○
canIdle	☐
block	○
lost	○
won	○
ko	○
hurt	○

In order for the player to interact with my edge collider (so that he wouldn’t fall through the scene), I added a 2D box collider to the player character within the Unity inspector in order for it to collide with the edge collider, I then attached a Unity component called a rigid body to the player in order for the player to have gravity enabled so that the player won’t just float in the air, the rigid body is also what actually moves when the inputs are entered, because it is invisible and is a child of the sprite component, it appears as though the sprite is the only thing moving where as it is actually the rigid body.

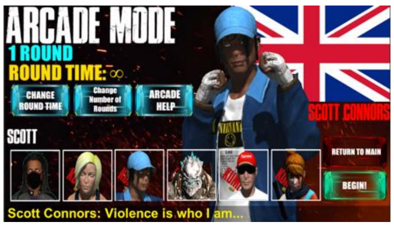
5.6 Opponent/Enemy AI

Creating the Ai was a very challenging task due to the fact that the opponent must think, react and decide upon things just how a human being would if they were playing the game, the opponent has to take into account when to block, attack, move, duck, jump and play certain animations. The first step in creating an AI was to have a standard Idle animation, we want the AI to be visible and moving. The first AI created was an AI version of the character called "LiftOff" I began by creating all the animations using the sprite baker component yet again.

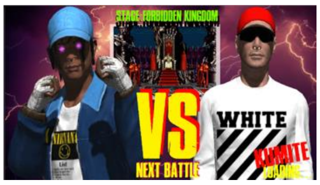
I downloaded more animations from the Maximo Fuse website I then, dragged and dropped these animation files into the sprite baker and began creating a sprite pack (See Appendix 4.0 for how the sprite pack software I purchased functions). Now for the opponent itself, I created an empty game object, added the components needed for the opponent (See implementation part below) and then I added a sprite as the default sprite for when this opponent component is loaded.

5.7 Arcade Mode/Reset Scene

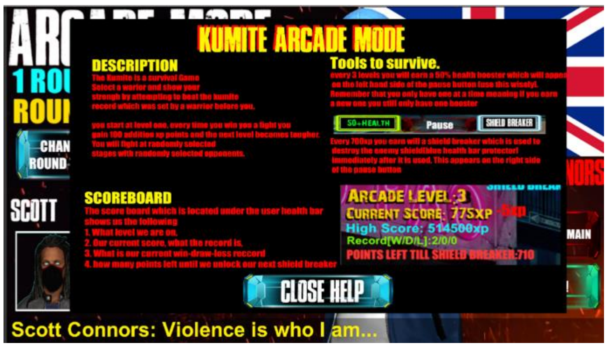
5.7.1 The Arcade scene



Arcade mode is one of the most important features of this game, the Arcade scene in this project is very similar to the character select scene and the Map/Difficulty scene. In fact, the arcade is essentially a combination of both along with it's own separate features. As mentioned it is similar to the character select scene when it comes to the subtitles bar, character voice's and music however, in the Arcade scene the user is only prompted to select which character they want to use, the game will automatically choose the opponents as well as the difficulty and the stage.



The reason the user is only prompted to ask which character they want to use is simply because this will be the character they will use throughout the whole arcade mode. When a user selects a character they want to use and they click to begin, the game will present a versus screen as it does whenever any fight begins. This scene is the "Reset scene". The Reset scene (Shown on the left hand side) is what we use to restart a new fight or set up a new one, it displays both fighters along with the stage in the background, it also works as a loading scene and is not only used in arcade mode but also in training mode and Fight now mode.



KUMITE ARCADE MODE

DESCRIPTION
The Kumite is a survival Game
Select a warior and show your
strengh by attempting to beat the kumite
record which was set by a warrior before you.

you start at level one, every time you win you a fight you
gain 100 addition xp points and the next level becomes tougher.
You will fight at randomly selected
stages with randomly selected opponents.

Tools to survive.
every 3 levels you will earn a 50% health booster which will appear
on the left hand side of the pause button (use this wisely).
Remember that you only have one at a time meaning if you earn
a new one you still only have one booster

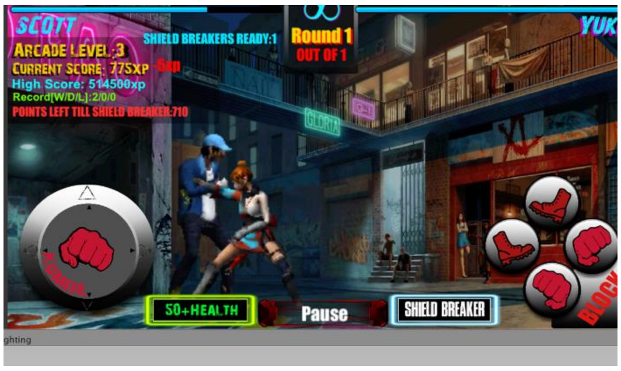
Every 700xp you earn will a shield breaker which is used to
destroy the enemy shield(blue health bar protector)
immediately after it is used. This appears on the right side
of the pause button

SCOREBOARD
The score board which is located under the user health bar
shows us the following
1. What level we are on.
2. Our current score, what the record is.
3. What is our current win-draw-loss reccord
4. how many points left until we unlock our next shield breaker

ARCADE LEVEL :3
CURRENT SCORE: 775XP -5xp
High Score: 514500xp
Record[W/D/L]:2/0/0
POINTS LEFT TILL SHIELD BREAKER:710

In the Arcade scene as you can see there are five buttons in total, the green "begin" button is used to begin the arcade mode, the return to main menu will take us back to the main menu scene. The addition three buttons located above the character select bar are arcade settings, these buttons allow us to change each round time as well as how many rounds each arcade level has, the last button opens up a window (See Kumité Arcade mode help window on the right) to explain how arcade works and what are the rules.

5.7.2 Fight scene in Arcade Mode



In Arcade mode we work with a new "levels" feature, this means when when a fight is over, if we win we can progress to the next level which will be more challenging whereas if we lose we will then be asked if we want to retry the current level. When a new level is loaded the game will give us a new opponent and a new stage. Because the arcade is more of a survival mode, this means the user can fight the same opponent multiple times however, each level increases the difficulty meaning that the opponent will be a lot stronger than the previous level. On both sides of the pause button in the image above you can see a "50+ health" button and a "Shield breaker". The shield breaker is rewarded every 3 levels so as we have reached the 3rd level in the shield above (which the scoreboard below the player's name indicates) we have unlocked a shield breaker.

The shield breaker is used to remove the opponent's shield as soon as it is activated/pressed. The "50+ health" button is used to add an additional 50% of health to the player meaning the user can last longer in a fight, this button becomes active every time the user earns 700xp points (In the scoreboard, the last red text will show the player how many points remaining until they unlock a new one). On the scoreboard, underneath the current level you can see a text component which states how many points the player is on and below that we can see what the current high score is. The last text in the scoreboard shows the player's current win/draw/loss record and above the scoreboard we can see how many shield breakers we currently have available. Every time we activate a shield breaker or a 50+ health button a window shatter audio source is played.

5.7.3 Arcade points system

Every punch and kick a player lands will earn them an additional 10xp points, every time the player takes a hit from the opponent the game will remove 5xp points from their total. If a player loses a fight and chooses to retry the level, they will lose 300xp points and every fight they win will gain them an additional 100xp points.

6. Implementation/Testing

6.1 Main Menu

I created a C# script for the main scene(MenuScript.cs) and within that script I added functions for each button. The training stage button has a function attached to it from the menu script, this function includes IEnumerator SelectMenu (); which is a base interface for all non-generic enumerators, this C# enumerator option is very useful when it comes to implementing loading screens as well as loading slider due to the fact that the application loads the Scene asynchronously in the background (at the same time as the current Scene). The enumerator contains the following line.
AsyncOperation asyncLoad = SceneManager.LoadSceneAsync("Select");

Appendix 1 shows this line of code nested inside the Select Enumerator. This line of code instructs the application to go to the character select scene within the Unity inspector, it will look through the build path trying to locate a scene with the name “Select” and once it discovers it, it will load it asynchronously along side the current scene.
This is the best method for creating loading scenes since we do not want to open up a new scene if everything in that scene is not ready or running as this could cause a lot of problems such as Unity crashing resulting in the loss of progress or the application could freeze/crash whilst being played on a mobile device.
This Main menu C# script also includes functions to play and pause the music which are attached to buttons within the settings menu, and it also includes close menu functions which is attached to the buttons which close the mini menu which is opened from the settings button (this also has its own function).

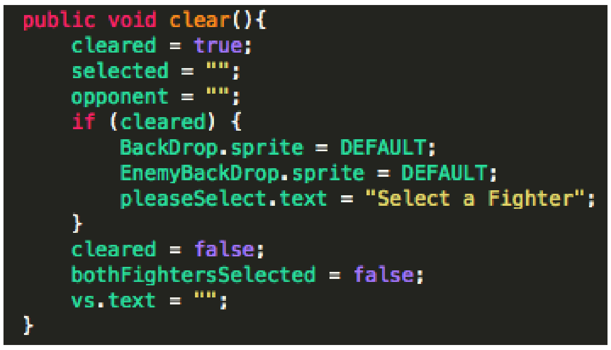
C# has a default Update () function which is called every frame the game is played and this function checks for which button the user is pressing and it then performs the function which the button is attached to. As mentioned above there is also a Start () function which is used here to set certain attributes and variables up with values, the Start () function will run the music, start the video in the background, disable the settings menu in order for it to not be visible as soon as the program is launched. Every button and canvas has its own public variables in the script and in Unity where the script is attached meaning it is possible to drag buttons to specify which button is which so that Unity and C# will understand which button is which.

6.2 Character Select

in the character select script I made two public text objects and I referenced them into the script on Unity via the Unity inspector as shown below so that the script could access them whilst they were in the Unity scene, one of the text is the please select and the other is the subtitles text component (Appendix 2 shows how these two public fields along with other public fields from the CharacterSelect.cs script appear in the Unity inspector after being declared in code).

As mentioned above, once a character has been selected the character will say a short sentence for example Scott Connors will say “Violence is who I am”. In order to create a character’s voice, multiple audio components (on for each character) were added within the CharacterSelect.cs script and they were named whatever the character's name is along with the word "audio" as shown in Appendix 3.1. I then went into the C1Select method (Appendix 3.2) in the script, within each if statement I stated which audio to play and when using the Play () method. I then made a public text SubTitles attribute, this attribute is created for the character's subtitle, under the Audio.Play() methods for each character I gave the Subtitles text component a string value which it will be set to depending on what the character's sentence is. Within the Unity inspector I created a text component and dragged it into the script and I also created audio components which I attached mp3 files to them containing the voice clips.

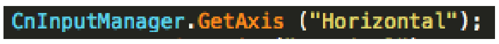
The "PleaseSelect" text will open up with the words "select a fighter', once a player has been selected (by clicking the button attached to the player's face) the character's name string will be sent to method called C1Select and this method will make the "selected" string equal to whatever name was sent to it via the public variable within the button, if the selected string already has a name then the "opponent" string will become whatever the name sent to the method was. If the select name is not empty (meaning the user selected the character they want to use, the PleaseSelect text on the bottom of the game screen will change to "Select an Opponent", after the user chooses an opponent the opponent's character's picture will also appear on the screen with the name and flag however it will appear on the right hand side rather than the left hand side where the player's character is. If the user is not satisfied with their choices they can click on the clear button which will run a method in character selected called clear, this method will reset the whole scene meaning you can choose characters again. The next step is click the next button so that you can go to the map and difficulty selection page or you can click the main menu button so that you can return to the main menu and choose a different option within that menu.


```
public void clear(){
    cleared = true;
    selected = "";
    opponent = "";
    if (cleared) {
        BackDrop.sprite = DEFAULT;
        EnemyBackDrop.sprite = DEFAULT;
        pleaseSelect.text = "Select a Fighter";
    }
    cleared = false;
    bothFightersSelected = false;
    vs.text = "";
}
```

6.3 Player Controls

Now that the character could be controlled with the keyboard, I need to make another canvas for the UI inputs because we want the game to be controlled by a mobile touch screen, not a keyboard. This is where my joystick idea comes in. I want to create a joystick, I created a joystick background picture and a stick background picture (both circles, stick circle is smaller) in adobe Photoshop and I imported them them into the Unity project. The stick image was nested into the background joystick image so that Unity knows the stick belongs to the background image.

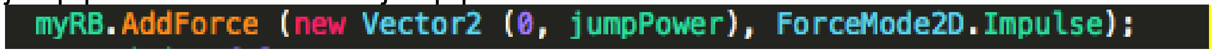
Now that the stick is in Unity I need to write a script, previously I had a script containing functions for user inputs, all I did was downloaded a few previously written Unity scripts from the Unity assets store for a joystick and I changed the keyboard input. Get functions to match the joystick inputs instead by saying CnInput manager. I wrote simple functions for moving horizontally so I had to write and then if statements to check if the value


```
CnInputManager.GetAxis ("Horizontal");
```

is lower than 0(go left) or if it's greater than 0(go right), this allowed my joystick to control the player moving left and right.

For the jump control I had to do the same thing but instead of getAxis("Horizontal") I had to obviously say Vertical. Now that the joystick was fully functioning and I was moving my player left right and jump (With the animations playing as well), I could move onto attack buttons, I added 4 Unity buttons to the canvas and 4 variables to the main player script. I attached each of them to an animation using the update function in the script, I made toggle options in the animator so that the animations know to play when a certain button is pressed.

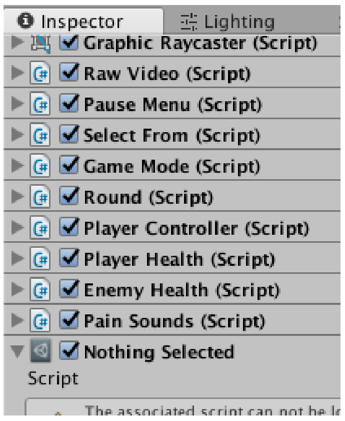
Every button had a button variable within Unity so that I could drag and drop the Unity buttons into the script section under the inspector tab (every public variable declared in the script is visible here so the user can update which is which). Every move function or property in my scripts actually work with the rigid body so for example to set the jump power I made a float jump power variable and I wrote



```
myRB.AddForce (new Vector2 (0, jumpPower), ForceMode2D.Impulse);
```

MyRB is the variable name of my rigid body, vector 2 is for X and Y axis vector, 0 is the jump power and the force mode is 2D.
y should jump up and fall back down, but what the user will see is the player jump up and fall down (with the proper animations playing).

6.4 Training Stage/Fight stage scripts



Inspector | Lighting
- Graphic Raycaster (Script)
- Raw Video (Script)
- Pause Menu (Script)
- Select From (Script)
- Game Mode (Script)
- Round (Script)
- Player Controller (Script)
- Player Health (Script)
- Enemy Health (Script)
- Pain Sounds (Script)
- Nothing Selected

Script

The associated script can not be l

This scene has the most C# scripts attached to it overall compare to the other scenes simply because this scene is where the game actually happens. As shown in the image on the right hand side, the Fight scene's inspector has nine C scripts attached to it. The raw image script is used to render and play video onto a raw image component. The pause menu script is in charge of the three different types of pause menus within the game, these pause menus include the arcade pause menu which only contains two options and those are resume and quit.

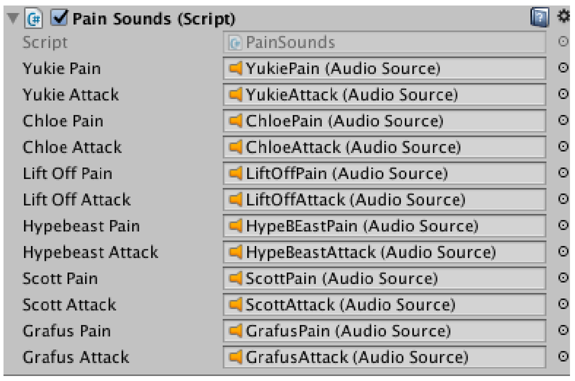
Another menu attached to this script is the exhibition/fight now pause menu which is an extended version of the arcade menu featuring the ability to view controls and change a few settings similar to the settings option in the main menu script, this pause menu also introduces the ability to rematch, meaning a user can choose to restart a match if they want by using this option. The last option this menu gives the user is the ability to return to character select and choose a different character and opponent. The last pause menu is the training menu which plays just like the fight now pause menu. In this script there is also pop up menus which appear at the end of a fight. If we are in arcade mode and we win the fight, this menu will ask us if we want to "Continue" to the next fight or if we want to "quit" and return to the main menu, however if we lose then rather than be asked to continue we will be asked if we want to restart the match.

SelectFrom.cs is used to set the settings of the fight, for example it will disable all characters and then enable the two we chose to use, it will disable all the stage backgrounds leaving the one we chose in the map difficulty scene and it will also change many settings depending on which of the three game modes we chose out of Arcade Mode, Training mode and Exhibition/Fight now mode.

The PlayerController.cs script is used to take input from the fight buttons and joystick and reflect them onto the player. So it registers which attack methods we send and it will play the correct animation, fight sound as well perform tasks such as decreasing the enemy health bar. This is the script we use in order to play every character animation as well as control the character movement. The joystick logic came from a component I downloaded in the Unity asset store called CN Controls which included its own joystick, I used this joystick to build up from and ultimately create my own joystick with a new design and Kumité logo, the joystick from CN controls was for a 3D game meaning I had to research and understand how this joystick works before proceeding to implement a similar one for a 2D game instead. Because this asset I downloaded was a free asset I was able to take some classes from it and modify them hence some of my functions and attributes needed references to the CnInputManager.

Both PlayerHealth.cs and EnemyHealth.cs are very identical, they are used to control the health and shield bars for both the player and the opponent. The PlayerController.cs script and the Opponent.cs script will communicate with these scripts by passing on information to them so for example if the user attacks the enemy, the PlayerController.cs script will tell the EnemyHealth.cs script to remove 5% from the opponent shield bar and if the shield bar is empty it will remove it from the health bar instead.

GameMode.cs and Round.cs work together exclusively, the Round.cs script is in charge of controlling the round and everything relating to the round for example the time remaining, the maximum round and the current round. The GameMode.cs will play out these settings and will allow the Round.cs know when to restart a new round, when to end one or when to pause the timer. The GameMode.cs script tells the whole project when we have found a winner of a fight and when to the fight is over.



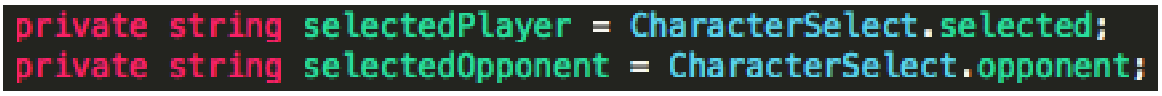
Pain Sounds (Script)

Script	PainSounds
Yukie Pain	YukiePain (Audio Source)
Yukie Attack	YukieAttack (Audio Source)
Chloe Pain	ChloePain (Audio Source)
Chloe Attack	ChloeAttack (Audio Source)
Lift Off Pain	LiftOffPain (Audio Source)
Lift Off Attack	LiftOffAttack (Audio Source)
Hypebeast Pain	HypeBEastPain (Audio Source)
Hypebeast Attack	HypeBeastAttack (Audio Source)
Scott Pain	ScottPain (Audio Source)
Scott Attack	ScottAttack (Audio Source)
Grafus Pain	GrafusPain (Audio Source)
Grafus Attack	GrafusAttack (Audio Source)

The pain sound script is used to play the attack sounds and pain sounds of each character, similar to how we implement music and catchphrases in the CharacterSelect.cs script, here we add Sound effects for each character (see section 4.6).

The fight mode was built like the training mode however, in the fight stage the user can customise their fighting experience, after the user has selected the character they want to use and the character they would like the opponent to use, they are then redirected to the map and difficulty scene which has a main canvas that holds the MapDifficulty.cs script, this script allows us to select a stage out of six possible choices.

In the training stage and fighting stage each character has their own health bar(yellow) and shield bar(blue). In order to defeat your opponent, you must eliminate their yellow health bar by landing as many hits as it takes for it to completely shrink, however in order to decrease the level of your opponent's health you must first destroy their shield bar, after the shield bar reaches zero, you can then start taking away their real health. Underneath the health bar and shield bar, there is a text label telling us the character's name, this updates every time you select a different character, the way it works is that if you choose a character in the character select scene, the c1 method passes the name variable from the CharacterSelect.cs script to the SelectFrom.sc script, an example would be this line, found in the SelectFrom.cs script.



```
private string selectedPlayer = CharacterSelect.selected;
private string selectedOpponent = CharacterSelect.opponent;
```

This means take the selected (string for the character's name) value from the CharacterSelect.cs script and make the SelectFrom variable "SelectedPlayer" equal to it. After we have the value, we then make our public text label print it onto the screen underneath the health and shield bars, I then used Unity to add font and colour to the text in order to make it pop out a little bit more for a nicer user experience.

Between both health and shield bars I have added a timer which will count down from 60 seconds so that if the timer exceeds that limit then whoever has the least amount of health left loses. I created a Timer.cs class which contains two public text variables, it also contains an update function which runs every frame allowing us to know how much time we have left, I used the `Time.deltaTime;` library which is a library within C# used for working out remaining time, I then made a public variable called remaining so that I could use the Unity inspector to set the value for it. In the Unity inspector I set the value to 60 meaning as soon as the game starts, the timer will count down from 60.

Underneath the timer, I have another text variable which will let us know which round we are currently on so for example if we were on round three, the Round.cs script will make this text label component show "Round 3" on the screen just underneath the timer, each time the round number increases, the text updates to let the user know. On both sides of the round script I have added blue circles to indicate how many rounds each player has one, so for example if the user has one three rounds out of seven and the opponent has one two out of seven then next to the user's character's name you will see three blue circles lined up next to each other and next to the opponent's name you will also see three circles.

The circles are all image components that I have declared within the Round.cs class which has control of how many of these circle images to activate and when to activate them. In Unity I imported one image of a blue circle and then I inserted 8 of them onto the scene and lined them up in the perfect spot, I then dragged and dropped them into the round.cs component I attached to the scene.

In order for the Round.cs script to have access to the images I used the GetComponent function on 8 image variables I created, now that the script had access to the variables, it can use my previously declared functions to show and hide the circles when necessary.

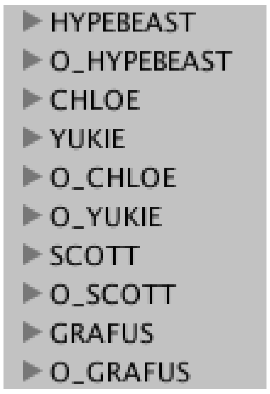
On the bottom of this scene lies a pause menu which can be used to pause the game via `Time.timeScale = 0;` which freezes the game. When the game is paused a menu pops up asking you if you want to return to main, go to settings, return to character select, resume the game or check the controls.

Main menu and character select options will start courintines and load their respective scenes, resume will resume the game and settings will open up another popup which will ask you if you want to turn the music off, or check the player controls. All the functions for these options are located within the PauseMenu.cs script.

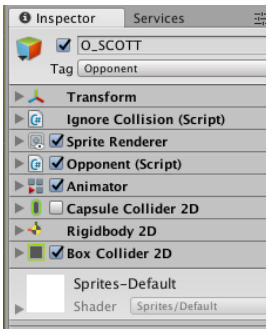
<u>6.5 Opponent AI states and Implementation</u>

6.5.1 Opponent inspector

As this game is a fighting game, this means the AI characters and user characters are all the same, meaning the AI can use the same characters that the user can use, in order to differentiate between the user characters and the opponent characters, all the character's within the fight scene are named the same way the user character's are named however the first two chars in their names are "O_" such as "O_SCOTT" or "O_YUKIE". The image on the right shows how the characters appear within the Fight scene's hierarchy, as you can see it is easy to differentiate which component is an opponent and which is the player.

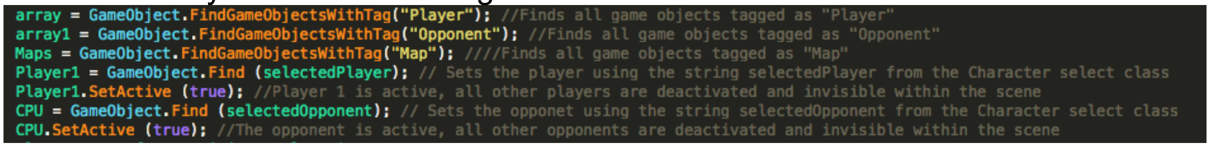


```
▶ HYPEBEAST
▶ O_HYPEBEAST
▶ CHLOE
▶ YUKIE
▶ O_CHLOE
▶ O_YUKIE
▶ SCOTT
▶ O_SCOTT
▶ GRAFUS
▶ O_GRAFUS
```



```
Inspector   Services
☑ O_SCOTT
Tag  Opponent
▶ Transform
▶ Ignore Collision (Script)
▶ ☑ Sprite Renderer
▶ ☑ Opponent (Script)
▶ ☑ Animator
▶ ☐ Capsule Collider 2D
▶ Rigidbody 2D
▶ ☑ Box Collider 2D
Sprites-Default
Shader  Sprites/Default
```

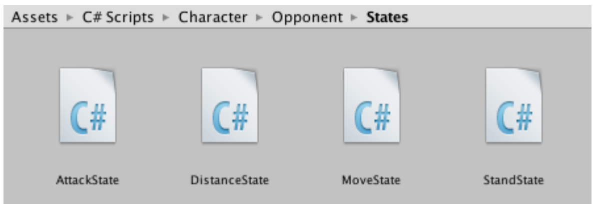
When you click on an opponent within the Unity scene hierarchy, the image on the left hand side shows how the Unity inspector shows the component. As you can see the opponent is tagged using a tag I created called "Opponent", tags are used in Unity to help you mark out certain types of objects so that within the code/script you can access them all at once, in my case I use this tag to create a set an array for opponents, the SelectFrom.cs script will use this array to find and enable the opponent chosen within the CharacterSelect.cs script and will disable all other objects within that array. The snippet blow is from the SelectFrom.cs script. As you can see we use the FindGameObectsWithTag(String) function to find all game objects within our Unity scene with this tag attached to it.



```
array = GameObject.FindGameObjectsWithTag("Player"); //Finds all game objects tagged as "Player"
array1 = GameObject.FindGameObjectsWithTag("Opponent"); //Finds all game objects tagged as "Opponent"
Maps = GameObject.FindGameObjectsWithTag("Map"); ////Finds all game objects tagged as "Map"
Player1 = GameObject.Find (selectedPlayer); // Sets the player using the string selectedPlayer from the Character select class
Player1.SetActive (true); //Player 1 is active, all other players are deactivated and invisible within the scene
CPU = GameObject.Find (selectedOpponent); // Sets the opponet using the string selectedOpponent from the Character select class
CPU.SetActive (true); //The opponent is active, all other opponents are deactivated and invisible within the scene
```

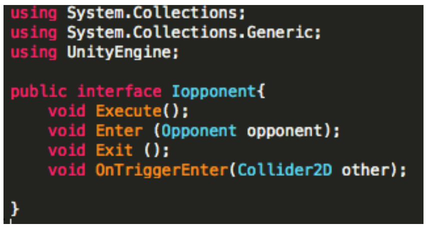
6.5.2 Opponent States

Similarly to the player, the opponent uses a transform component which allows the code to know where the opponent's rigid body is located, the sprite renderer will switch between animation sprites during each frame, the animator controller component is also similar to the player, the Opponent.cs script will be used to control and change opponent states(C# scripts shown in the image) and the box collider is used so that



Assets ▸ C# Scripts ▸ Character ▸ Opponent ▸ States

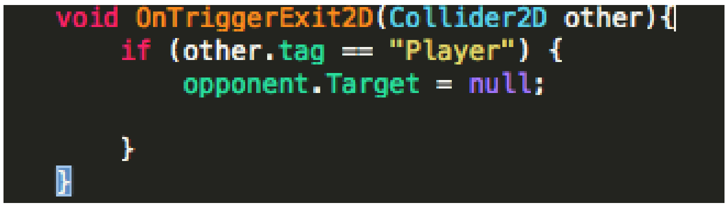
AttackState   DistanceState   MoveState   StandState the opponent can collide into the player, the edges and so that the opponent also does not fall through the ground.



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

public interface Iopponent{
    void Execute();
    void Enter (Opponent opponent);
    void Exit ();
    void OnTriggerEnter(Collider2D other);

}
```

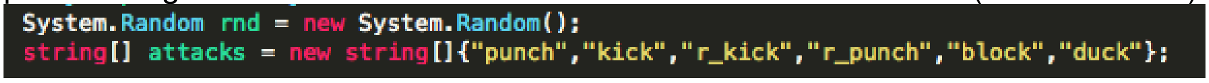
There are many different logics when it comes to figuring out how an AI works, how it makes decisions and how it reacts on it's own. In this project we will be using four scripts for four different states. Each of these four states implement the Iopponent.cs interface (snippet shown on the left hand side) which only consists of the following 12 lines of code. Each state will implement the four void methods, the execute () method is used for executing tasks as well as changing states, the enter () method is similar to a start method as explained previously this is where everything used will be initialised, exit is what happens after this state is no longer running and OnTriggerEnter is what happens when the collider we attached prioviously in the Unity inspector is triggered.

Every player within the scene has the tag "Player" attached to similar to the opponent's "Opponent" tag as explained previously, these tags are used to identify the objects within the scripts. The first thing an opponent must do is to spot the player and approach the player. I created a new script called OpponentSight.cs which contains the following snippet code. As you can see below, this script will identify a player using the player tag via collider collision. A new game object called "Sight" was attached to every opponent and it has box colliders attached to it, these colliders will then detect players using the player tags and once they collide and the opponent will enter the move state(MoveState.cs) and will approach the player.



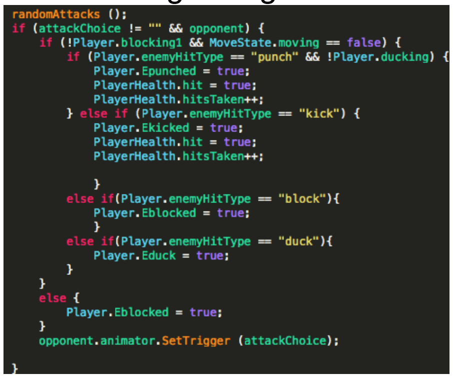
```
void OnTriggerExit2D(Collider2D other){
    if (other.tag == "Player") {
        opponent.Target = null;

    }
}
```

Once the opponent reaches a good distance(DistanceState.cs) and is close enough to fight the player, it will transition into an idle/stand state(StandState.cs) and will proceed to go back and fourth from the stand state to the attack state(AttackState.cs).



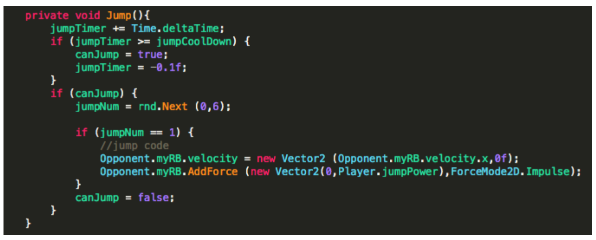
```
System.Random rnd = new System.Random();
string[] attacks = new string[]{"punch","kick","r_kick","r_punch","block","duck"};
```

The opponent now must attack the player, in order for the attacks to be random, the AttackState.cs script has an array of string objects containing strings for different attacks. I created a new System. Random object which is used to randomly generate a number, in this case we want it to randomly select a number which is within the attacks [] array index range therefore we use rnd.next (0,6) which means select a number between 0 and 5, once the opponent selects a number, for example 2, we will now tell the randomAttacks () method to set the attackChoice as a "kick".



```
randomAttacks ();
if (attackChoice != "" && opponent) {
    if (!Player.blocking1 && MoveState.moving == false) {
        if (Player.enemyHitType == "punch" && !Player.ducking) {
            Player.Epunched = true;
            PlayerHealth.hit = true;
            PlayerHealth.hitsTaken++;
        } else if (Player.enemyHitType == "kick") {
            Player.Ekicked = true;
            PlayerHealth.hit = true;
            PlayerHealth.hitsTaken++;

            }
        else if(Player.enemyHitType == "block"){
            Player.Eblocked = true;
            }
        else if(Player.enemyHitType == "duck"){
            Player.Educk = true;
        }
    }
    else {
        Player.Eblocked = true;
    }
    opponent.animator.SetTrigger (attackChoice);

}
```

The image to the left is a snippet of the void Combat () function, this function runs the randomAttacks () method which will select a random string out the attacks string, it will check if the opponent is within the attack range and that the payer is not blocking, if the conditions are met, the opponent will hit the player and it will take some health off of the player's health bar. The next step is to play the attack's animation in the game which is what the last line is set to do, if the attack was "r_kick", within the opponent's animator controller it will find this trigger and trigger it so that the animation plays.

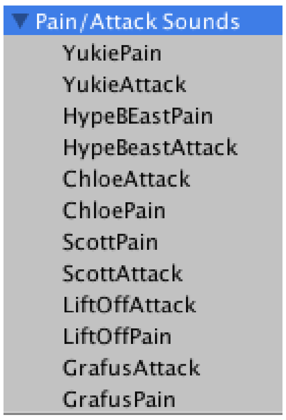


```
private void Jump(){
    jumpTimer += Time.deltaTime;
    if (jumpTimer >= jumpCoolDown) {
        canJump = true;
        jumpTimer = -0.1f;
    }
    if (canJump) {
        jumpNum = rnd.Next (0,6);

        if (jumpNum == 1) {
            //jump code
            Opponent.myRB.velocity = new Vector2 (Opponent.myRB.velocity.x,0f);
            Opponent.myRB.AddForce (new Vector2(0,Player.jumpPower),ForceMode2D.Impulse);
        }
        canJump = false;
    }
}
```

The combat function is the first function called every time the opponent enters the attack state. As you can see above the opponent can also duck and block. Similar to the player, the opponent can also jump using similar logic to the player. The player jumps via our control whereas the opponent jumps via the state transitions. Since we don't want the opponent to jump very often, we must use a JumpCoolDown timer which means the opponent must wait a certain amount of time until it is eligible to jump again.

The jump function also includes a randomizer which will choose a random number, if the number is equal to 1 then the opponent can jump, this is another logic I came up with to limit how often the opponent jumps so that the opponent will not jump around too often or else it'll be difficult to fight.

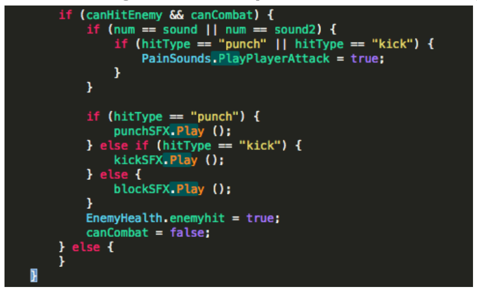
6.6 Sound Effects



6.6.1 Sound Effects implementation in the inspector

Sound effects was a key part of this fighting game project, the user would like more feedback other than what they see, I download a few special effects sounds from a website called freesoundeffects.com which provided me with punching sounds as well as kicking and blocking sounds. Sound effects sounds was a good idea however after further research long with having people test the game, most people said they would rather the characters speak and react using their own voices along with the sound effects. I decided to give the characters their own voices within the fight scene to go along with the character select voices.

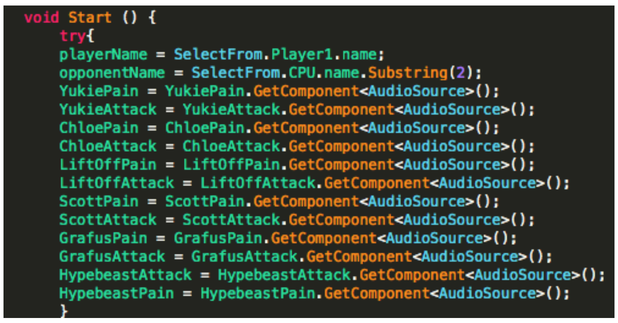
6.6.2 Player SFX (Sound Effects)



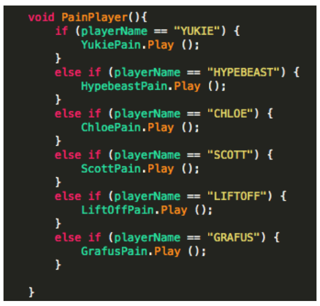
```
    if (canHitEnemy && canCombat) {
        if (num == sound || num == sound2) {
            if (hitType == "punch" || hitType == "kick") {
                PainSounds.PlayPlayerAttack = true;
            }
        }

        if (hitType == "punch") {
            punchSFX.Play ();
        } else if (hitType == "kick") {
            kickSFX.Play ();
        } else {
            blockSFX.Play ();
        }
        EnemyHealth.enemyhit = true;
        canCombat = false;
    } else {
    }
}
```

Within the Player.cs script I created three SFX (Sound Effect) audio components for block, punch and kick.
When a player presses a kick or punch or block button, the hitType string is set to "kick", "punch" or "block" depending on what the user chose, once this information is passed, within the attackEnemy () method (snippet on the left), we can see that if one of the conditions is met, an audio source for that string will be played using the .Play() method.

The next step is to take away some health from the enemy. The pain and attack sounds are each unique to each individual character, the snippet on the right shows the start function of the PainSound.cs script. The GetComponent function will get the component from the Unity inspector as explained previously.



```
void Start () {
    try{
    playerName = SelectFrom.Player1.name;
    opponentName = SelectFrom.CPU.name.Substring(2);
    YukiePain = YukiePain.GetComponent<AudioSource>();
    YukieAttack = YukieAttack.GetComponent<AudioSource>();
    ChloePain = ChloePain.GetComponent<AudioSource>();
    ChloeAttack = ChloeAttack.GetComponent<AudioSource>();
    LiftOffPain = LiftOffPain.GetComponent<AudioSource>();
    LiftOffAttack = LiftOffAttack.GetComponent<AudioSource>();
    ScottPain = ScottPain.GetComponent<AudioSource>();
    ScottAttack = ScottAttack.GetComponent<AudioSource>();
    GrafusPain = GrafusPain.GetComponent<AudioSource>();
    GrafusAttack = GrafusAttack.GetComponent<AudioSource>();
    HypebeastAttack = HypebeastAttack.GetComponent<AudioSource>();
    HypebeastPain = HypebeastPain.GetComponent<AudioSource>();
    }
```



```
void PainPlayer(){
    if (playerName == "YUKIE") {
        YukiePain.Play ();
    }
    else if (playerName == "HYPEBEAST") {
        HypebeastPain.Play ();
    }
    else if (playerName == "CHLOE") {
        ChloePain.Play ();
    }
    else if (playerName == "SCOTT") {
        ScottPain.Play ();
    }
    else if (playerName == "LIFTOFF") {
        LiftOffPain.Play ();
    }
    else if (playerName == "GRAFUS") {
        GrafusPain.Play ();
    }

}
```

There are four key methods in the painSound.cs script and those are PainPlayer (), PainOpponent (), AttackPlayer () and AttackOponnent (). All four of these methods look and function very similarly, for example here is a snippet of the PainPlayer () method. As you can see it checks which player we are using (the start menu snippet initialises the playerName string which is equal to the name of player 1 from the SelectFrom.cs script) and then it plays the audio source (from the inspector) for that particular player.

The PainOpponent method does the same thing only instead of player we use Opponent and both AttackOpponent and AttackPlayer work the same way only instead of pain, we use attack. Because these are private void methods, in order for the player and opponent to access these methods, we use public static Boolean true and false variables, if the player attacks the enemy, the player attack variable is true therefore the PainSound.cs script will play the AttackOpponent method and then reset the Boolean variable to false.

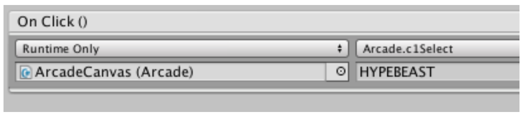
6.6.3 Opponent SFX (Sound Effects)

The Opponent uses the same sound effects and pain/attack sound audio sources as the player however, within the script these AudioSource.Play methods are located in different areas and play via different conditions. The attack sounds are played only when the enemy's AttackState.cs Combat () function is running.

6.7 Arcade mode

6.7.1 Selecting a character in arcade mode

The Arcade scene uses the Arcade.cs script in order to define most of it's components such as buttons as well as character selection. Similarly, to the CharacterSelect.cs script, the Arcade.cs script contains its own C1Select (String s) function which is used to select players by taking the player's name string as an argument and making the SelectFrom.cs selected variable equal to it. Above is an image from the Unity inspector of how the select button for the character called "HYPEBEAST" works.

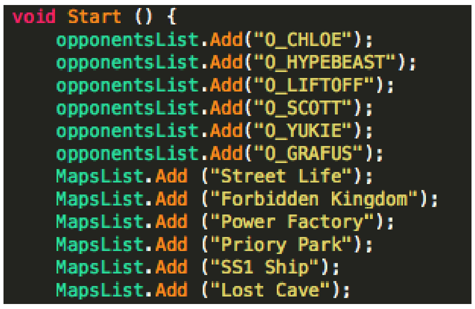


On Click ()
Runtime Only | Arcade.c1Select
ArcadeCanvas (Arcade) | HYPEBEAST

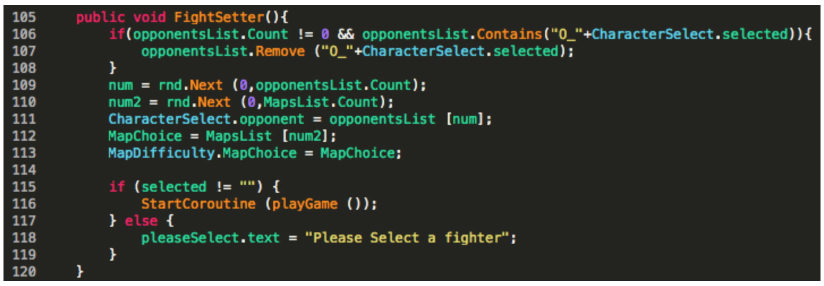
When the button is clicked, it will run the Arcade.c1Select (String s) function with the argument "HYPEBEAST". This C1Select function works how the C1Select function from the CharacterSelect.cs script, however in arcade mode the C1Select function only runs for the player and not the opponent, the reason for this is simply because in arcade mode, opponents are selected at random.

6.7.2 Selecting an opponent and a stage at random.

In the Arcade.cs script I have created two public static lists of strings called “opponentsList” and “MapsList”, these lists will be used to hold the names of all characters and all stages. In the start method I have added the names of all the opponents and all the stages (maps). In order to select an opponent at random, we will use this list to randomly choose an index from it. I declared and created two integers called “num” and “num2” and used the rnd.next function to generate a random number just like I did with random enemy attacks as previously explained.



```
void Start () {
    opponentsList.Add("O_CHLOE");
    opponentsList.Add("O_HYPEBEAST");
    opponentsList.Add("O_LIFTOFF");
    opponentsList.Add("O_SCOTT");
    opponentsList.Add("O_YUKIE");
    opponentsList.Add("O_GRAFUS");
    MapsList.Add ("Street Life");
    MapsList.Add ("Forbidden Kingdom");
    MapsList.Add ("Power Factory");
    MapsList.Add ("Priory Park");
    MapsList.Add ("SS1 Ship");
    MapsList.Add ("Lost Cave");
```

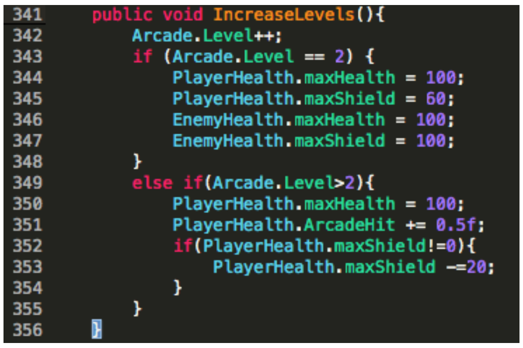


```
105     public void FightSetter(){
106         if(opponentsList.Count != 0 && opponentsList.Contains("O_"+CharacterSelect.selected)){
107             opponentsList.Remove ("O_"+CharacterSelect.selected);
108         }
109         num = rnd.Next (0,opponentsList.Count);
110         num2 = rnd.Next (0,MapsList.Count);
111         CharacterSelect.opponent = opponentsList [num];
112         MapChoice = MapsList [num2];
113         MapDifficulty.MapChoice = MapChoice;
114 
115         if (selected != "") {
116             StartCoroutine (playGame ());
117         } else {
118             pleaseSelect.text = "Please Select a fighter";
119         }
120     }
```

As shown in the FightSetter () method, line 109 will generate a random number between 0 and the Opponent’s list size and line 110 will do the same but for the MapsList.

Now we can set the CharacterSelect.opponent string to the index of that random number as shown in line 111, the same will be done for MapDifficulty.MapChoice (line 113). After every level, the arcade will continue to run this FightSetter () method to randomize stages and opponents. Line 106 and 107 are used so that the opponent does not use the same character as the player.

6.7.3 Arcade difficulty and levels



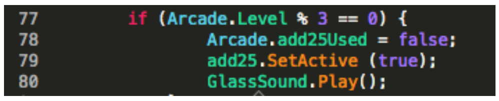
```
public void IncreaseLevels(){
    Arcade.Level++;
    if (Arcade.Level == 2) {
        PlayerHealth.maxHealth = 100;
        PlayerHealth.maxShield = 60;
        EnemyHealth.maxHealth = 100;
        EnemyHealth.maxShield = 100;
    }
    else if(Arcade.Level>2){
        PlayerHealth.maxHealth = 100;
        PlayerHealth.ArcadeHit += 0.5f;
        if(PlayerHealth.maxShield!=0){
            PlayerHealth.maxShield -=20;
        }
    }
}
```

The arcade difficulty starts off as the normal “easy” difficulty defined in the MapDifficulty.cs script, this is the default level 1 difficulty. After the first fight is over, the level increases to level 2 and here is where the difficulty is altered. In GameMode.cs there is a method called IncreaseLevels (), this function runs every time a player wins a fight in arcade mode and proceeds to the next fight.

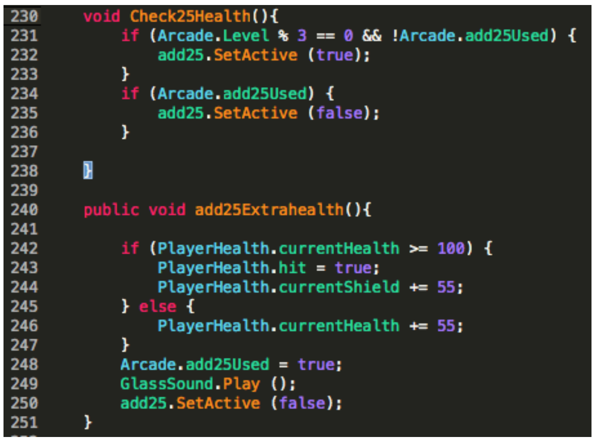
The Arcade.level integer is incremented by 1 and if the level is equal to 2 then the player’s shield is decreased from 100 to 60 at the beginning of the fight, however if the level is greater than 2 then the enemy’s attacks will take away an additional 5% from the player’s health meaning the opponent is stronger by 5%. Every time the player goes up a level (wins a fight) the next opponent’s hits are 5% stronger than they were before and the player’s shield decreases until it gets to zero. The higher the level, the more powerful the opponent is and the less shield the player has.

6.7.4 Arcade score, shield breaker and 50% addition health

By default, the arcade high score is 4900, in order to ensure that users do not cheat and try to beat this score by adding more rounds to each fight, this score will alter to 4900 multiplied by the number of rounds so for example if we choose 3 rounds then the high score is 4900 multiplied by three which is 14,700xp. The default high score is declared in the MenuScript.cs script as a public static integer. In the GameMode.cs we check to see when we can activate and deactivate both the shield breaker and 50+ health scripts. Every 3 levels we unlock a new 50+ health booster as shown in the image above, line 77 simply means, if the current level is a multiple of three then we set the 50+ health button to active (visible and useable), once it is pressed and we add 50+ heath then the button is disabled again until the next level we reach which is a multiple of 6.



```
77          if (Arcade.Level % 3 == 0) {
78                  Arcade.add25Used = false;
79                  add25.SetActive (true);
80                  GlassSound.Play();
```



```
    void Check25Health(){
        if (Arcade.Level % 3 == 0 && !Arcade.add25Used) {
            add25.SetActive (true);
        }
        if (Arcade.add25Used) {
            add25.SetActive (false);
        }

    }

    public void add25Extrahealth(){

        if (PlayerHealth.currentHealth >= 100) {
            PlayerHealth.hit = true;
            PlayerHealth.currentShield += 55;
        } else {
            PlayerHealth.currentHealth += 55;
        }
        Arcade.add25Used = true;
        GlassSound.Play ();
        add25.SetActive (false);
    }
```

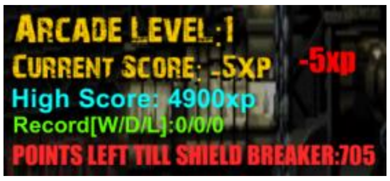
On the left we see two more methods for the 50+ health button, the Check25Health () method is used to check if we can activate the 50+ health button and make it visible, this method will be running in the update function (a default C# method which runs every frame). The add25Extra health button is the method which the button runs, this method is attached to the button in the Unity inspector.

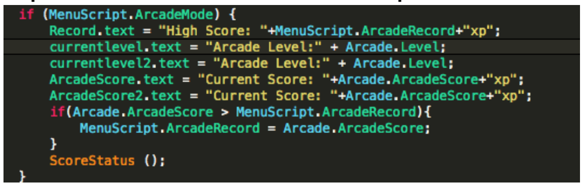
The shield breaker is unlocked every time the user earns 700xp points which are earned in two ways, winning a fight/level earns the user 100 points and every attack landed earns them 10xp, this is done by incrementing the score by 100 whenever the user clicks continue after they have won a fight, and whenever the opponent's health decreases. The arcade score can also decrease, when an opponent lands a strike the user loses 5xp points and when the user loses and chooses to retry the level they lose 300xp.

In the fight scene the shieldbreaker has a text component as explained previously, this text is updated every frame in case a new shield breaker is added or removed. The GameMode.cs script has two functions which run in its update function, these functions are CheckShieldBreaker () and check25Health (), both functions update the text components for their corresponding features.

6.7.5 Arcade Scoreboard

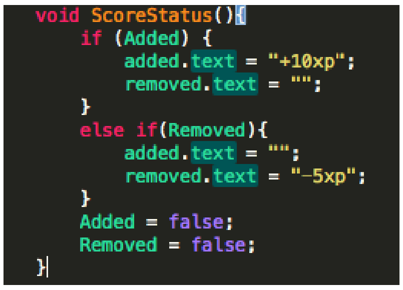
The arcade has a scoreboard feature which is only visible in arcade mode, it is essentially a Unity game object component which holds 1 background image and 6 text components, the first text component is the current level, this text component is updated in the GameMode.cs update function and so are the others. There are two text components which show the user how





```
if (MenuScript.ArcadeMode) {
    Record.text = "High Score: "+MenuScript.ArcadeRecord+"xp";
    currentlevel.text = "Arcade Level:" + Arcade.Level;
    currentlevel2.text = "Arcade Level:" + Arcade.Level;
    ArcadeScore.text = "Current Score: "+Arcade.ArcadeScore+"xp";
    ArcadeScore2.text = "Current Score: "+Arcade.ArcadeScore+"xp";
    if(Arcade.ArcadeScore > MenuScript.ArcadeRecord){
        MenuScript.ArcadeRecord = Arcade.ArcadeScore;
    }
    ScoreStatus ();
}
```

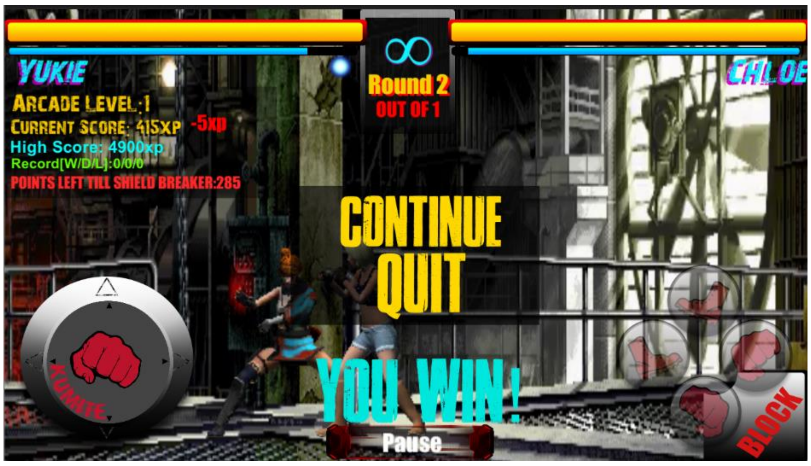
many new points have been added and how many are removed, in the image above we can see that the player had 5xp points removed because the player took a hit from the opponent. This is the "removed" text component, the other is the "added" text component which appears in green rather than red, here is a script of how both work in the ScoreStatus () method. If we have attacked an opponent the added is true and removed is false, if we have been attacked the added is false and remove is false.



```
void ScoreStatus(){
    if (Added) {
        added.text = "+10xp";
        removed.text = "";
    }
    else if(Removed){
        added.text = "";
        removed.text = "-5xp";
    }
    Added = false;
    Removed = false;
}
```

6.7.6 Arcade Fight over

At the end of the game, the character who has won will play their win animation, in the character called Yukie's case, she will do a quick dance and taunt her opponent. This animation is done similarly to the other animations; a win trigger is set to the fighter's animator. After this animation is complete, the GameMode.cs script will run the Time.Timescale = 0; which will freeze the screen and a new window will open up.

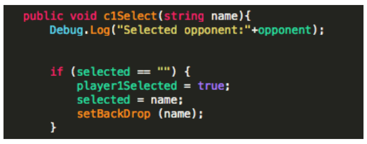


When an arcade fight is over there are two possible outcomes, if we win (as shown in the image), we'll be shown the win screen, where the player will be asked if they want to quit (return to main menu) or continue onto the next level. If the player loses they'll be shown the lose screen which is similar to the win screen only instead of the continue option they'll be asked to retry. The continue button awards 100xp when pressed whereas the retry option removes 300xp points.

6.8 Testing the project

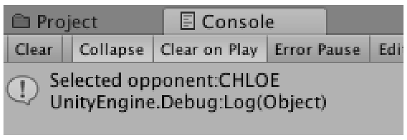
6.8.1 Manual testing

Throughout the project, every error and exception I received I was using the Unity C# built in Debug class to locate and fix the errors. The main method of locating errors after they occurred was to use the C# Debug.Log() method which prints out values to the Unity console similarly to a print statement, one example was when I was trying to select an opponent but I kept receiving errors because the system was saying an opponent with that name was not found. I decided to add a Debug.Log statement to check what the name I was sending to the system was and here was the result.



```
public void c1Select(string name){
    Debug.Log("Selected opponent:"+opponent);


    if (selected == "") {
        player1Selected = true;
        selected = name;
        setBackDrop (name);
    }
```

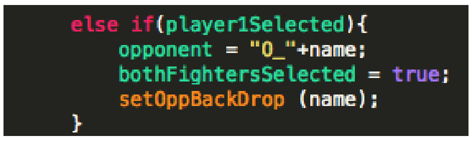
CharacterSelect.cs



```
Selected opponent:CHLOE
UnityEngine.Debug:Log(Object)
```

Unity Console

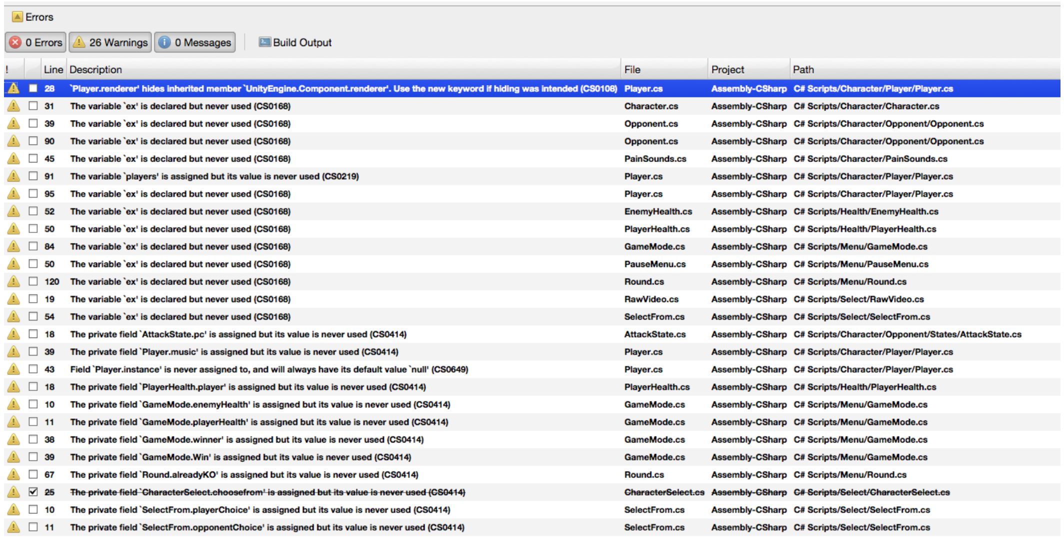
The reason the system could not find the opponent was simply because I named all my opponents the same name with “O_” as the first two characters so in this case the C1Select () function should have returned “O_CHLOE” and not “CHLOE” as shown in the console above.



```
else if(player1Selected){
    opponent = "O_"+name;
    bothFightersSelected = true;
    setOppBackDrop (name);
}
```

Now that I know what the problem is, how do I fix it? Very simple, if the player has been selected, the next string to be set (opponent string) must have “O_” concatenated to the beginning of it. Now the problem is fixed and we can choose opponents. Throughout my project I have been using the Debug class to assert a few conditions, log results and check if a few references equal each other. Check the bibliography section for more detains on the debug class.

MonoDevelop also allows me to build all the scripts at once and check if there are any errors, it will send me updates and warning as shown in the screenshot below on all my scripts as shown in the image below. Not everything labelled is an error but this feature allows me to check what could go wrong before it actually goes wrong. This feature has helped me save time by eliminating errors before they had even occurred, after I received errors which there were no explanations on, I used this log to check what warnings might have appeared first.

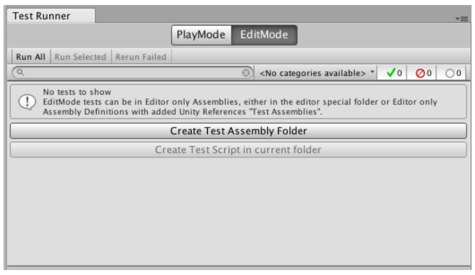

Errors

0 Errors | 26 Warnings | 0 Messages | Build Output

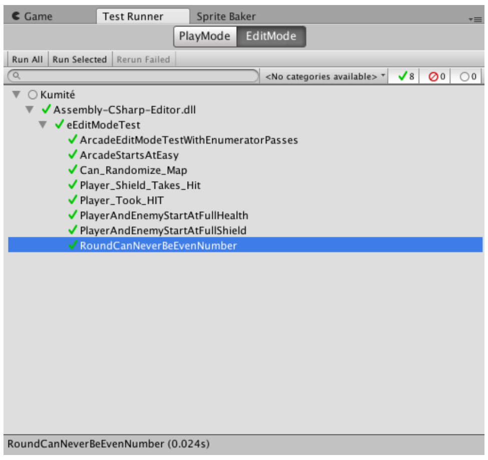
!		Line	Description	File	Project	Path
⚠	☐	28	`Player.renderer' hides inherited member `UnityEngine.Component.renderer'. Use the new keyword if hiding was intended (CS0108)	Player.cs	Assembly-CSharp	C# Scripts/Character/Player/Player.cs
⚠	☐	31	The variable `ex' is declared but never used (CS0168)	Character.cs	Assembly-CSharp	C# Scripts/Character/Character.cs
⚠	☐	39	The variable `ex' is declared but never used (CS0168)	Opponent.cs	Assembly-CSharp	C# Scripts/Character/Opponent/Opponent.cs
⚠	☐	90	The variable `ex' is declared but never used (CS0168)	Opponent.cs	Assembly-CSharp	C# Scripts/Character/Opponent/Opponent.cs
⚠	☐	45	The variable `ex' is declared but never used (CS0168)	PainSounds.cs	Assembly-CSharp	C# Scripts/Character/PainSounds.cs
⚠	☐	91	The variable `players' is assigned but its value is never used (CS0219)	Player.cs	Assembly-CSharp	C# Scripts/Character/Player/Player.cs
⚠	☐	95	The variable `ex' is declared but never used (CS0168)	Player.cs	Assembly-CSharp	C# Scripts/Character/Player/Player.cs
⚠	☐	52	The variable `ex' is declared but never used (CS0168)	EnemyHealth.cs	Assembly-CSharp	C# Scripts/Health/EnemyHealth.cs
⚠	☐	50	The variable `ex' is declared but never used (CS0168)	PlayerHealth.cs	Assembly-CSharp	C# Scripts/Health/PlayerHealth.cs
⚠	☐	84	The variable `ex' is declared but never used (CS0168)	GameMode.cs	Assembly-CSharp	C# Scripts/Menu/GameMode.cs
⚠	☐	50	The variable `ex' is declared but never used (CS0168)	PauseMenu.cs	Assembly-CSharp	C# Scripts/Menu/PauseMenu.cs
⚠	☐	120	The variable `ex' is declared but never used (CS0168)	Round.cs	Assembly-CSharp	C# Scripts/Menu/Round.cs
⚠	☐	19	The variable `ex' is declared but never used (CS0168)	RawVideo.cs	Assembly-CSharp	C# Scripts/Select/RawVideo.cs
⚠	☐	54	The variable `ex' is declared but never used (CS0168)	SelectFrom.cs	Assembly-CSharp	C# Scripts/Select/SelectFrom.cs
⚠	☐	18	The private field `AttackState.pc' is assigned but its value is never used (CS0414)	AttackState.cs	Assembly-CSharp	C# Scripts/Character/Opponent/States/AttackState.cs
⚠	☐	39	The private field `Player.music' is assigned but its value is never used (CS0414)	Player.cs	Assembly-CSharp	C# Scripts/Character/Player/Player.cs
⚠	☐	43	Field `Player.instance' is never assigned to, and will always have its default value `null' (CS0649)	Player.cs	Assembly-CSharp	C# Scripts/Character/Player/Player.cs
⚠	☐	18	The private field `PlayerHealth.player' is assigned but its value is never used (CS0414)	PlayerHealth.cs	Assembly-CSharp	C# Scripts/Health/PlayerHealth.cs
⚠	☐	10	The private field `GameMode.enemyHealth' is assigned but its value is never used (CS0414)	GameMode.cs	Assembly-CSharp	C# Scripts/Menu/GameMode.cs
⚠	☐	11	The private field `GameMode.playerHealth' is assigned but its value is never used (CS0414)	GameMode.cs	Assembly-CSharp	C# Scripts/Menu/GameMode.cs
⚠	☐	38	The private field `GameMode.winner' is assigned but its value is never used (CS0414)	GameMode.cs	Assembly-CSharp	C# Scripts/Menu/GameMode.cs
⚠	☐	39	The private field `GameMode.Win' is assigned but its value is never used (CS0414)	GameMode.cs	Assembly-CSharp	C# Scripts/Menu/GameMode.cs
⚠	☐	67	The private field `Round.alreadyKO' is assigned but its value is never used (CS0414)	Round.cs	Assembly-CSharp	C# Scripts/Menu/Round.cs
⚠	☑	25	~~The private field `CharacterSelect.choosefrom' is assigned but its value is never used (CS0414)~~	~~CharacterSelect.cs~~	~~Assembly-CSharp~~	~~C# Scripts/Select/CharacterSelect.cs~~
⚠	☐	10	The private field `SelectFrom.playerChoice' is assigned but its value is never used (CS0414)	SelectFrom.cs	Assembly-CSharp	C# Scripts/Select/SelectFrom.cs
⚠	☐	11	The private field `SelectFrom.opponentChoice' is assigned but its value is never used (CS0414)	SelectFrom.cs	Assembly-CSharp	C# Scripts/Select/SelectFrom.cs

6.8.2 Unity Automated testing

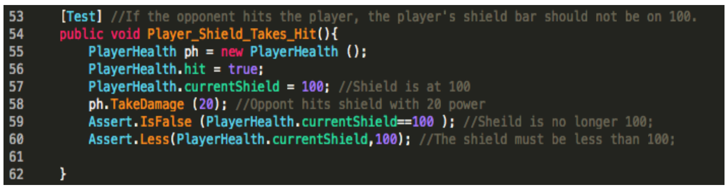
The Unity Test runner is an automated testing tool which is used to test code in both edit mode and play mode, it can also be used to target certain platforms (in my case iOS and Android). It uses an integration of the NUnit library which is an open source testing library. The manual testing worked well for me however, I decided to figure out a way to do automation testing which is where I discovered the Unity test runner.

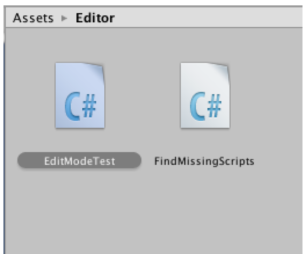

Test Runner
PlayMode
EditMode
Run All
Run Selected
Rerun Failed
<No categories available>
Create Test Assembly Folder
Create Test Script in current folder
No tests to show
EditMode tests can be in Editor only Assemblies, either in the editor special folder or Editor only Assembly Definitions with added Unity References "Test Assemblies".

The EditModeTest.cs script I created specifically for automation testing using the test runner is located in the Editor folder which is located in the assets folder. This script contains 8 automated tests which I wrote out whilst testing the game. As you can see in the screenshot attached, all tests passed successfully (green tick) after I began writing and running them individually. A test example which I wrote is the test I wrote in order to test the player's shield (see Player_Shield_Takes_Hit test below).



```
Kumitė
  ✓ Assembly-CSharp-Editor.dll
    ✓ eEditModeTest
        ✓ ArcadeEditModeTestWithEnumeratorPasses
        ✓ ArcadeStartsAtEasy
        ✓ Can_Randomize_Map
        ✓ Player_Shield_Takes_Hit
        ✓ Player_Took_HIT
        ✓ PlayerAndEnemyStartAtFullHealth
        ✓ PlayerAndEnemyStartAtFullShield
        ✓ RoundCanNeverBeEvenNumber
RoundCanNeverBeEvenNumber (0.024s)
```


```
    [Test] //If the opponent hits the player, the player's shield bar should not be on 100.
    public void Player_Shield_Takes_Hit(){
        PlayerHealth ph = new PlayerHealth ();
        PlayerHealth.hit = true;
        PlayerHealth.currentShield = 100; //Shield is at 100
        ph.TakeDamage (20); //Oppont hits shield with 20 power
        Assert.IsFalse (PlayerHealth.currentShield==100 ); //Sheild is no longer 100;
        Assert.Less(PlayerHealth.currentShield,100); //The shield must be less than 100;

    }
```



7. Critical Appraisal

7.1 Summary and critical analysis of work completed

I believe all of my original requirements from the Gantt chart and the very first plan I ever submitted were met except for one requirement which was that the game must be a 3 dimensional game. After spending hours upon hours of researching how I would turn a fighting game into a 3D game I decided that this was not the best method to create a fighting game therefore I had to find either a solution or an alternative. I decided that I will make the game 2D however I will create the characters in 3D and then convert each animation into 2D sprites.

I had a few issues with the Unity engine, sometimes when there is an error the whole software crashes and restarts meaning I lost my work quite a lot due to a few simple errors I made when coding. One key part of the project that kept going wrong was the arcade feature, I tried many different logics in order to implement the arcade mode but most of them were not working, In arcade mode the gamer decides who the opponent is and which stage to use the Select and Stage scenes are usually responsible for this however, in Unity once you go back to a stage you must do the settings all over again meaning the game kept asking the user to choose these settings whereas in the arcade mode they should be done at random.

After weeks and weeks of researching different ways I could make the arcade's continue button generate a new random fight, I came across the idea of creating a new scene which will be in ran between each fight, this scene is called the Reset scene, after testing it and failing multiple times, I came with new ideas such as running the load fight scene within the Reset.cs start function, this turned out to work well and the arcade mode finally began to work, I converted the Reset scene into a Vs scene where the player, opponent and map are shown before the fight begins. The arcade mode was by far the hardest part of this project and it took over a month to complete, 3 weeks was researching as well as trial and error and 1 week was to finally implement it correctly.

Throughout this project I had many issues, the first issue was keeping up with deadlines, because I had never created a project this big or complex before I struggled in the experience department, I also had issues when it came to the C# language simply because I have never used an object oriented programming which was not java so I therefore had to take a few pluralsight courses and research the language in order to try and understand how it works and how I would start this project later on.
One of the biggest issues I had was not the code, but how the code was organised and structured, since I have never created a project which uses more than 3 different java classes and files I struggled organising five C# scripts which I had designed, another issue was that some functions and objects were created and used in the wrong files meaning it was difficult to try and locate them later on.

After struggling with code management I came up with the idea of splitting the scripts and naming them differently, this meant instead of five scripts which had 1000 lines of code each, I could split that into 12-15 scripts which are nicely organised and each have no more than 300 lines of codes. I then created separate folders within the C#

scripts folder so that it would be easy to identify which scripts are for the player, which are for the menus and which are for the opponents.
In order to have saved time, one thing I would have done was to create separate C# scripts for different subjects at the start because I wasted a whole week trying to organise them and make them work. Another thing I would do if I could go back would be to add more try/catch blocks because every time I received an exception in runtime I lost some changes I made within Unity.

The game works just as I imaged it would, I have met all the requirements although some took a lot longer than expected and some were a lot more challenging than expected but in the end all the requirements were made including the requirements I had to alter. If I were to do this project again, something I'd do differently definitely would have been structuring the code correctly by categorising each of the 32 scripts by placing them into different folders, this would have saved me so much time because whilst writing the scripts, it was getting very confusing at times.

7.2 Discussion on social, commercial and economic context

I believe that games make a huge impact in today's society especially mobile phone games due to the fact that most if not all people have smart phones and there are games out there to suit everyone's needs and interests. Games play a huge role in today society, especially for young people who are our future due to the fact that they spend a lot of time on these games meaning a lot of their development and learning can come from things that they see on these screens. These games that are being played on mobile devices can range from educational games for children and toddlers to learn subjects such as maths and English as well as games which allow us to learn words in different languages. In fact, there are even games for domestic pets such as house cats which tend to be developed for tablets.

Many people may believe that a fighting game such as Kumité could be influencing violence and aggression within young people, these games tend to have an age rating, for example if I were to rate the Kumité game I would say it's for people over the age of 7 simply because the violence is not gory and there is no blood, most of it is playful. This game can benefit young children by keeping them entertained and out of trouble considering the game is very addictive and it can help them release some stress as well as negative energy.

The methods used to create Kumité can be used commercially, in fact I have already played and reviewed a few mobile phone applications and games written in C# and created with Unity. Not only can my methods be commercially used and suitably adapted, to my understanding they already are. I have referenced a few games made with Unity (See bibliography). C# is a very powerful programming language and it is widely popular and used by many organisations such as Microsoft.

This project is expected to be relevant in the teaching environment because it can be used as an example of how one person can create such a large and complex mobile phone application. Unity Engine is a very popular game development engine in the IT world, experience in Unity will open doors to many jobs in the computer science and informatics field.

7.3 My personal development during the project

After weeks and weeks of learning everything relating to C# and Unity which I was having an issue with, I realised that C# started to become natural, I didn't need pluralsight videos or guidance or any additional help. After the prototype demonstration I finally began to understand the basics of C# and Unity, I could then go on to start doing complex tasks such as recording the player movements, creating animations and animators and attaching these components to the scenes as well as the scripts. The game began to finally look like a game, it was playable, I had buttons, joysticks, music, characters and most things you see in the game today.

This project helped me improve my programming ability not only in C# but also in Java, I realised that they are both quite similar and my confidence has grown when it comes to writing code, the game was hard to create at first but after learning all the components and software I needed it became fun to create. This project has opened many doors when it comes to my knowledge in software engineering, I now understand how to manage code, how to write automated tests as well as how to use the Unity game engine which is very popular in today's software engineering market. I can now say that I understand C#, basic objective C and Java, knowledge on these programming languages has given me the confidence and the ability to learn new programming languages now that I understand basic logic and concepts.

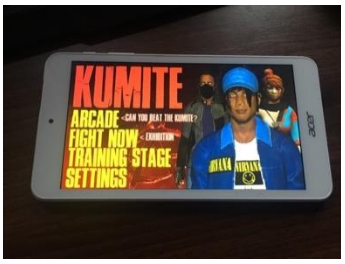
I have decided to start learning C++ which is another popular programming language. Thanks to this project, I learned a whole new programming language in the space of a few months and I can safely say I understand how game engines work as well as how mobile phone applications are created, this will be good for my future as a software engineer because most companies focus on these languages as well as a few other similar languages which I can also learn quickly.

8.Conclusion

When I was deciding what kind of project I wanted to do I was planning on using Java considering it was the only programming language I knew at the time, my knowledge with java was not great considering I was not a strong programmer and I had always struggled with java modules. I had the idea of a fighting game but I had no idea how I would create one with java, so because I was determined to create a fighting game I had no option but to learn about game engines and how they interact with programming languages. When I came across C# and Unity I had doubts, simply because I was told that C# is similar to the C and C++ programming languages which are all arguably a lot harder than Java

The first few weeks I was struggling with C# and Unity, the Unity UI was extremely complex since it is built for professionals and the C# library was a whole new programming library which I had to get used to. I previously had 2 years of Java experience and I was still struggling with it, so it felt like a huge mistake taking on a whole new programming language which I knew very little about and I only had a matter of months to learn how to code with it. After weeks and weeks of studying C# I began getting used to it, I knew basic programming concepts which helped me pick up the basics very quickly however Unity was still an issue but after watching a lot of pluralsight videos I began getting the hang of both C# and Unity. To conclude, I believe that I have not only met my requirements, but I have surpassed some of them, for example I war originally only meant to make the game for iOS devices such as iPhone and iPad however, because the title of the project is “Games with smart phone” I decided to push myself and create a version for android devices too. Most smart phone users either have an android mobile phone or an iPhone.

I wanted my game to be multiplatform and by the end of the project I achieved this. I sent the APK (Android application package) to a few of my friends with android mobile phones and they downloaded the game and tested it, they enjoyed playing it. I also decided to optimize the game and make it available for android tablets as well as iPads. This was requirement I succeeded, every other requirement was also completed and I added additional features to the game such as arcade mode and training mode which were not part of the original requirements I set myself when creating this application. I have reviewed my Gantt chart as well as my interim report and plan and I can safely say I have done everything which I set out to do and the game works perfectly on tablets and mobile phones. This was by far the longest project I have ever done; I wrote thousands of lines of code spread out over approximately 32 different C# Scripts.



Kumité running on an Acer tablet[Android]

Kumité running on an Apple iPhone 6s[iOS]

9. Bibliography and Citations

[1] Activasoft. Google Play store. 09 11 2017. 06 12 2017 <https://itunes.apple.com/us/app/shadow-fight-2/id696565994?mt=8>.

[2] Anything, Colors Do. Free Video Backgrounds | Stock Footage | Smoke | Red Smoke | Color Smoke. 03 January 2016. 11 01 2018 <https://www.youtube.com/watch?v=1erLZOQn4jk>.

[3] App store Games. 12 01 2018. Apple Inc. 12 01 2018 <https://itunes.apple.com/au/genre/ios-games/id6014?mt=8>.

[4] Audiotrimmer. Audiotrimmer.com. 09 01 2018 <https://audiotrimmer.com>.

[5] authors), Wikipedia(Various. Street Fighter . 11 10 2017 <https://en.wikipedia.org/wiki/Street_Fighter_(video_game)>.

[6] Carle, Christ. Street Fighter: The Complete History. Chronicle Books; 01 edition (1 May 2010), n.d.

[7] creators, The software. How To Download And Install UNITY 3D. 27 07 2016. 29 04 2018 <https://www.youtube.com/watch?v=jkV5X6FWvrM>.

[8] difficult, Hardly. Unity Test Runner #Unity3d C#. 30 11 2017. 21 04 2018 <https://www.youtube.com/watch?v=VqmcWnjreqg&t=195s>.

[9] Engine, Unity. Testing - Test Runner. 29 04 2018 <https://docs.Unity3d.com/Manual/testing-editortestsrunner.html>.

[10] Era, Beatz. (FREE) Kodak Black Type Beat - "Decisions" | Tunnel Vision Type Beat Instrumental I Prod.Ditty Beatz. 19 02 2017. Ditty Beats. 11 10 2017 <https://www.youtube.com/watch?v=7bitHnmcpS8>.

[11] ezgif. ezgif - Animated gifs made easy. EzGif. 03 April 2018 <ezgif.com>.

[12] FREE, Antoine Lavenant - Street Fighter (Dubstep Remix). Antoine Lavenant - Street Fighter (Dubstep Remix) . 21 11 2011. 11 04 2018 <https://www.youtube.com/watch?v=5L6RhqoaBis&list=LLXlie_VYIQEie31FW_2JZ1A&index=8&t=0s>.

[13] games, nekki. Shadow fighting 2. 16 11 2017. 06 12 2017 <https://itunes.apple.com/us/app/shadow-fight-2/id696565994?mt=8>.

[14] gifer. 02 04 2018 <https://gifer.com/en/2246>.

[15] GoodaDesign. Burning ash rise background. 07 August 2017. 11 March 2018 <https://www.youtube.com/watch?v=GYIwS3TTtyw>.

[16] Huskey, Darry. 02 06 2017. tekken review ign. 30 11 2017 <http://uk.ign.com/articles/2017/06/02/tekken-7-review>.

[17] I Am Street Fighter - 25th Anniversary Documentary. By Street Fighter/Capcom. n.d.

[18] Inc, Rhyme. freesoundeffects.com. Rhyme Inc. 12 04 2018 <https://www.freesoundeffects.com/free-sounds/fight-sounds-10034/>.

[19] Karnage, Lord. Classic Game Room HD - STREET FIGHTER 2 TURBO on SNES. <https://www.youtube.com/watch?v=NalltB-ihz8>.

[20] KillerOnTheBEat. FREE] Energic Intense Trap Beat - "Codeine" | Prod. by kILLerOnTheBeat | Free Type Beat 2017. 27 09 2017. 23 03 2018 <https://www.youtube.com/watch?v=xMl2rPiMnBg>.

[21] Maximo, Adobe. Adobe. 02 April 2018 <https://www.mixamo.com/#/>.

[22] Microsoft. Debug class in C#. 29 04 2018 <https://msdn.microsoft.com/en-us/library/system.diagnostics.debug(v=vs.110).aspx>.

[23] mp3louder. mp3louder. 11 02 2018 <http://www.mp3louder.com>.

[24] MrSoundtabel. Thunder sound effect. 14 08 2012. 09 04 2018 <https://www.youtube.com/watch?v=T-BOPr7NXME&list=LLXlie_VYIQEie31FW_2JZ1A&index=3&t=0s>.

[25] Playstation. Sony Playstation - Tekken 7. 13 09 2017 <https://www.playstation.com/en-gb/games/tekken-7-ps4/>.

[26] Playstation, Sony. Playstation Street Fighter V. 12 09 2017 <https://store.playstation.com/en-gb/product/EP0102-CUSA01222_00-SF5DELUXE2000000>.

[27] Point, Social. Monster Legends. 16 11 2017. 04 12 2017 <https://itunes.apple.com/us/app/shadow-fight-2/id696565994?mt=8>.

[28] readers, Natural. 02 02 2018 <https://www.naturalreaders.com/online/ >.

[29] RealTutsGML. Game Maker Studio vs Unity 5. 16 November 2016. 11 October 2017 <https://www.youtube.com/watch?v=HbcLuYBxbIE>.

[30] S.A, Vivid Games. Real boxing . 16 09 2017. 06 12 2017 <https://itunes.apple.com/us/app/shadow-fight-2/id696565994?mt=8>.

[31] Scott, Cavan. Tekken Volume 1. Vol. 1. Titan Comics; 01 edition (19 Dec. 2017), n.d.

[32] Sinasac, Brian. Unity 2D Fundamentals - Character Control. 07 June 2017. 15 November 2017 <https://www.pluralsight.com/courses/Unity-2d-fundamentals-character-control>.

[33] SoundEffectsArchive. Glass Mirror Break Sound Effect. 07 04 2016. 25 03 2018 <https://www.youtube.com/watch?v=ayDQ3sOhoN0>.

[34] Store, Unity Asset. Unity Asset Store. Kirill nadezhdin. 29 10 2017 <https://assetstore.Unity.com/packages/tools/input-management/cn-controls-joystick-touchpad-button-and-d-pad-15233>.

[35] Studio, Unity Asset Store/Tank & Healer. Simple Health Bar Free. 13 10 2017 <https://assetstore.Unity.com/packages/tools/gui/simple-health-bar-free-95420>.

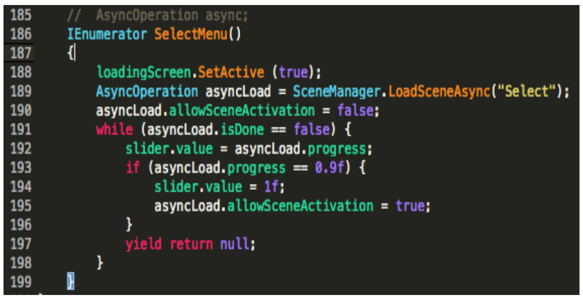
[36] TREE, PNG. PNG Tree Graphic Design. 2017. 04 02 2018 <https://pngtree.com/so/game-buttons/3>.

[37] Unity. Playing Video In Unity - Introduction and Session Goals [1/8] Live 2017/7/12. 14 07 2017. 11 04 2018 <https://www.youtube.com/watch?v=kwuHYwf7DvA&t=6s>.

[38] WhatTheFf. Dota2 female pain sounds. 14 02 2016. 23 03 2018 <https://www.youtube.com/watch?v=WFSh6vkwlb8&index=4&list=LLXlie_VYIQEie31FW_2JZ1A&t=105s>.

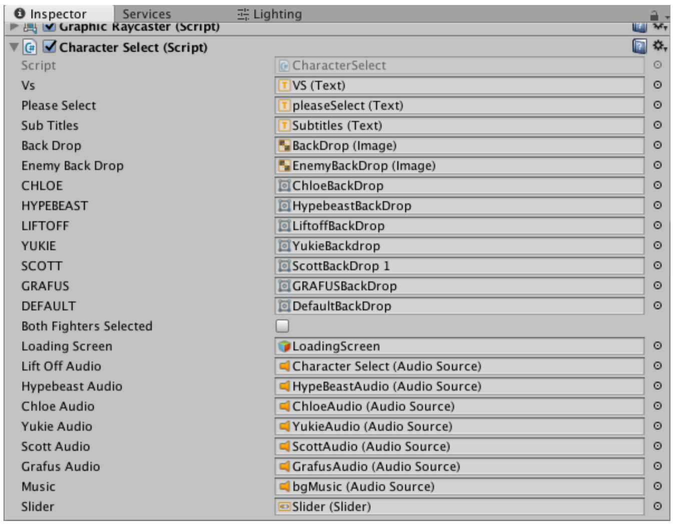
10. Appendix

1.

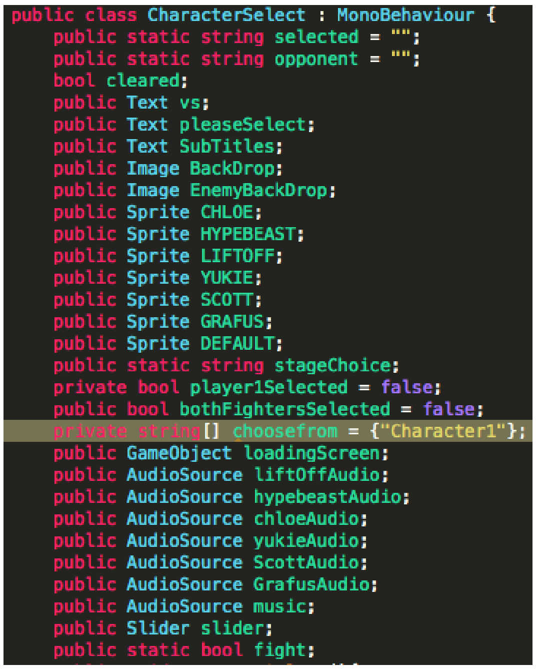


```
    // AsyncOperation async;
    IEnumerator SelectMenu()
    {
        loadingScreen.SetActive (true);
        AsyncOperation asyncLoad = SceneManager.LoadSceneAsync("Select");
        asyncLoad.allowSceneActivation = false;
        while (asyncLoad.isDone == false) {
            slider.value = asyncLoad.progress;
            if (asyncLoad.progress == 0.9f) {
                slider.value = 1f;
                asyncLoad.allowSceneActivation = true;
            }
            yield return null;
        }
    }
```

2.1



Script	CharacterSelect
Vs	VS (Text)
Please Select	pleaseSelect (Text)
Sub Titles	Subtitles (Text)
Back Drop	BackDrop (Image)
Enemy Back Drop	EnemyBackDrop (Image)
CHLOE	ChloeBackDrop
HYPEBEAST	HypebeastBackDrop
LIFTOFF	LiftoffBackDrop
YUKIE	YukieBackdrop
SCOTT	ScottBackDrop 1
GRAFUS	GRAFUSBackDrop
DEFAULT	DefaultBackDrop
Both Fighters Selected	☐
Loading Screen	LoadingScreen
Lift Off Audio	Character Select (Audio Source)
Hypebeast Audio	HypeBeastAudio (Audio Source)
Chloe Audio	ChloeAudio (Audio Source)
Yukie Audio	YukieAudio (Audio Source)
Scott Audio	ScottAudio (Audio Source)
Grafus Audio	GrafusAudio (Audio Source)
Music	bgMusic (Audio Source)
Slider	Slider (Slider)

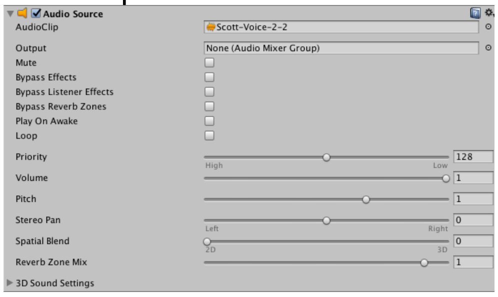


```
public class CharacterSelect : MonoBehaviour {
    public static string selected = "";
    public static string opponent = "";
    bool cleared;
    public Text vs;
    public Text pleaseSelect;
    public Text SubTitles;
    public Image BackDrop;
    public Image EnemyBackDrop;
    public Sprite CHLOE;
    public Sprite HYPEBEAST;
    public Sprite LIFTOFF;
    public Sprite YUKIE;
    public Sprite SCOTT;
    public Sprite GRAFUS;
    public Sprite DEFAULT;
    public static string stageChoice;
    private bool player1Selected = false;
    public bool bothFightersSelected = false;
    private string[] choosefrom = {"Character1"};
    public GameObject loadingScreen;
    public AudioSource liftOffAudio;
    public AudioSource hypebeastAudio;
    public AudioSource chloeAudio;
    public AudioSource yukieAudio;
    public AudioSource ScottAudio;
    public AudioSource GrafusAudio;
    public AudioSource music;
    public Slider slider;
    public static bool fight;
```

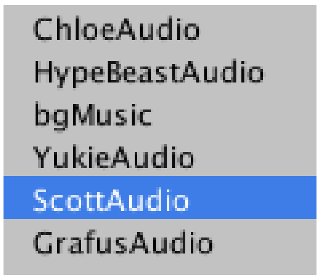
CharacterSelect.cs fields appear in the Unity inspector(left) ready for Unity components to be dragged into them so that the CharacterSelect script(right) can use them within the code.

3.1

The Audio source (left) is attached to the corresponding character's audio component(right), in this case we are looking at Scott's Audio source with the mp3 audio clip called "Scott-Voice-2-2" attached to it.

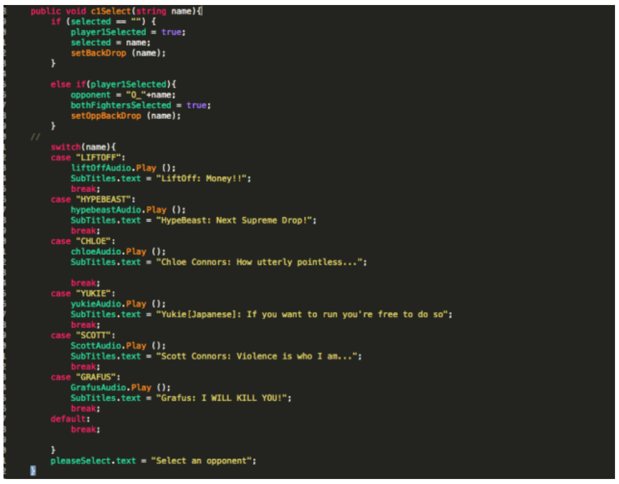


AudioClip	Scott-Voice-2-2
Output	None (Audio Mixer Group)
Mute	☐
Bypass Effects	☐
Bypass Listener Effects	☐
Bypass Reverb Zones	☐
Play On Awake	☐
Loop	☐
Priority	128
Volume	1
Pitch	1
Stereo Pan	0
Spatial Blend	0
Reverb Zone Mix	1



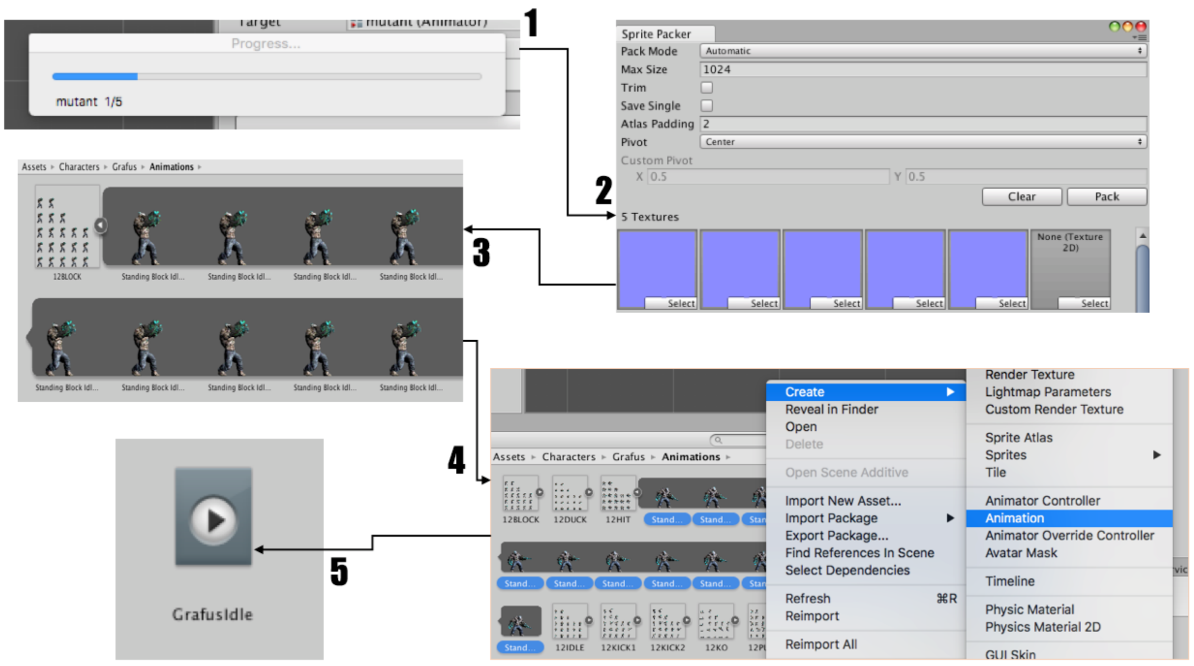
3.2

The c1Select function takes a character name, sets their subtitle text, plays their audio and tells the SelectFrom.cs script who to use in the next fight.



```
public void c1Select(string name){
    if (selected == "") {
        player1Selected = true;
        selected = name;
        setBackDrop (name);
    }

    else if(player1Selected){
        opponent = "O_"+name;
        bothFightersSelected = true;
        setOppBackDrop (name);
    }
//
    switch(name){
    case "LIFTOFF":
        liftOffAudio.Play ();
        SubTitles.text = "LiftOff: Money!!";
        break;
    case "HYPEBEAST":
        hypebeastAudio.Play ();
        SubTitles.text = "HypeBeast: Next Supreme Drop!";
        break;
    case "CHLOE":
        chloeAudio.Play ();
        SubTitles.text = "Chloe Connors: How utterly pointless...";

        break;
    case "YUKIE":
        yukieAudio.Play ();
        SubTitles.text = "Yukie[Japanese]: If you want to run you're free to do so";
        break;
    case "SCOTT":
        ScottAudio.Play ();
        SubTitles.text = "Scott Connors: Violence is who I am...";
        break;
    case "GRAFUS":
        GrafusAudio.Play ();
        SubTitles.text = "Grafus: I WILL KILL YOU!";
        break;
    default:
        break;

    }
    pleaseSelect.text = "Select an opponent";
}
```

Appendix 4.0



```
Sprite Packer
Pack Mode       Automatic
Max Size        1024
Trim            [ ]
Save Single     [ ]
Atlas Padding   2
Pivot           Center
Custom Pivot
  X 0.5          Y 0.5
                              Clear   Pack
5 Textures
```

After a sprite is baked using the Sprite baker, the sprite baker opens up the sprite packer which allows us to “pack” all the sprites together into a group of sprites (each frame of the animation has its own sprite), following steps 1-5 we create a .anim component which we will later add to a state within the animator so that this animation plays in this case Grafus’s idle animation.