Contents

Analysis

Problem Identification

While playing multiple fighting games in the past, I realised that many of them require the ability to recognise patterns, instead of allowing a player to build certain skills to progress. This was a problem while playing against AI opponents especially, as they are often built very formulaically meaning that they have a set style of fighting every single time. This simply leads to many people being able to beat them easily. Realising this problem, I decided a single player fighting game with AI that can actively adapt to different playstyles would be strong for allowing players to adapt their skills.

The skills that can be gained from games like this include quick and adaptable problem-solving skills, which cannot be achieved by games where situations you are in are repeatedly the same.

So, I will be making a 2d retro styled boxing game named "King of the Ring". I believe a game like this would gain a lot of popularity within the boxing community, without having any age restrictions as any person with an interest in boxing, and even those who aren't fans of boxing would enjoy a game like this.

Games with a similar style like 'Street Fighter' gained a lot of popularity in the 1980s and were mainly played on arcade machines and are still popular to date with new games coming out often.

Consideration of Computational Methods

Some features of this game will require me to consider specific methods of computational thinking. For example:

- I will have to use abstraction as I will not be able to add every single detail of real boxing into the game as that would not be efficient in the development as only the core fundamentals of boxing are required to make it a simple but enjoyable game. Another reason abstraction would be required is because of computational requirements. If the game contains more efficient code (due to unnecessary features being ignored), it would be accessible for people with computers that range from low to high-end systems.

- This will be an input-intensive game meaning that the program will have to maintain with lots of inputs, such as moving and punching, and processing them all at the same time. To make sure this is handled properly, I will implement concurrent thinking/ parallel processing. This means that multiple inputs will be processed simultaneously which will combat lag and missing input problems, and will overall increase the enjoyability of the game

- As this program is quite large and complex, I will use problem decomposition to break down the project into smaller self-contained tasks. This will allow me to focus on each specific aspect of the program separately which will make for more organised development. This also means that once a single task is completed, it can be tested and if it works as it is meant to, it can be recycled multiple times throughout the project if needed without having to be re-tested.

- The game will require the use of OOP as there will be many 'objects' that will be created such as the characters and different GUIs. These objects will be in their own scripts, as Unity is a script-based game engine. The

benefit of this is that each script can be tested separately to see if it works, and it should not affect how the other scripts function and work. The creation of objects is very useful as it helps with reusability (as explained above) and is a great way of organisation.

- Backtracking is a useful technique where you methodically visit each version of the project and build a final solution based on the correct versions. After a new feature is updated, I will save it and will move on to the next feature. If in the end I find that the final product doesn't work, I will retrace my steps until I get a full path of working versions which builds to a fully functioning end-solution.

Type of AI to implement:

- For the AI opponent, the mini-max algorithm is usually used for creating intelligence within games. The minimax algorithm is a decision-making algorithm that is often used in 2 player games zero sum games, where the "pay-off" for one character is the loss for the other. It explores a game tree by recursively analysing possible moves or plays and their consequences. However, this algorithm is normally for games where it has time to work out the response, such as chess or other board games. For a 2d fighting game, where the game is played in real time and is constantly updating, the algorithm may be too slow, and take longer to think of a move than to perform it. This could be an issue causing the AI to be very unresponsive and not adapting to play style of the opposite character as I would want it too.

- Another way of implementing an AI opponent is through a simple neural network. Neural networks resemble human brains, in which the neurons (nodes) are sorted in layers and they receive input, process that input and produce an output. With a game as small as this, there are only so many different types of inputs and outputted responses that the AI could make, meaning that it may be possible to create neural network for the AI which adapts to the situations that occur during the game. However, they are often trained with large amounts of data due to them usually having lots of different types of inputs and many functions and are quite difficult to create.

- A very common type of AI system in this context is a Function State Machine (FSM) which transition between a finite number of states depending on the changing external conditions, which in the case of my game would be how the player is using their attacks, or their health etc. FSMs are perfect for games which are constantly changing in real time as they can be made to be extremely responsive and adaptable.

Stakeholders

My friends, Daeniel and Jack, have an interest in boxing as well as gaming. This makes them perfect for the role of stakeholders and means I can always get tailored end-user feedback during the development stages to make sure I stay on track and produce the best product I can. Why I believe they are both well fitted stakeholders for my project:

Daeniel - A frequent viewer of boxing, used to play lots of fighting games as a child.

Jack - Has real life boxing experience and has been fighting games for the past 5 years. His favourite is 'Street fighter 5'.

Question 1 – What specific features would you like to see being implemented in the game?

Daeniel – He believed that from experience of playing fighting games, he finds that all the different combinations of attacks can be hard to remember and learn. Due to this he suggests that I make a feature where you will be able to see tips in the corner of the screen between the rounds. Building upon this I think having a screen where you can see all the different combinations of attacks would be necessary.

Jack – Likes the idea of having different characters with their own distinct abilities and fighting styles because this will give each character their own personalities.

Question 2 – Is having a variety of playable characters with unique abilities important to you?

Both agreed that having a roster of characters to choose from was a better idea than having a single character that you stay with throughout all fights as they believe they would stay more engaged for longer as once they get bored of one character, they can learn how to play a new one.

Question 3 – How do you feel about the game being more retro arcade styled instead of a realistic boxing experience?

Daeniel – Loves this idea as he believed it would be fun for situations where you are playing with friends.

Jack – Likes the idea as retro styled games are very fun to play. He also believes it would be a throwback for a certain demographic of people who would have been playing these games in arcades when they were first introduced.

Question 4 – How important is it to have customisable playable characters?

Daeniel – He likes the idea of customisable characters because it allows players to feel a connection with their character as they have had influence on how they look.

Jack – He doesn't care much about the appearance of characters and believes whether a fighting game is good or not, entirely depends on the gameplay.

Question 5 – Would you prefer a single-player game with a storyline or a multiplayer focused game?

Daeniel – Would prefer a multiplayer focused game as he would like to play the game for short periods of time and still get a good gaming experience. He said that for a game to be single player with a storyline, it would have to be a 3d game with very high-quality graphics and a deep plot for it to be enjoyable which does not fit with the idea I am trying to create.

Jack – From experience, he believes that a good 1v1 fighting experience outweighs a good story-based game and that having an AI opponent to practise against is good so that people can build up on their skills.

Question 6 – What other suggestions do you think would enhance the game experience?

Daeniel – Having good sound effects for different actions would be simple but effective features to implement.

Jack – He wants me to focus mainly on making the mechanics of the gameplay. As examples he likes having counterattack mechanics, good blocking mechanics and good combinations which deal higher damage to the opponent.

Review:

- Daeniel suggests a screen where you can see all the controls and combinations as it can get quite confusing
- Jack is emphasizing the importance of unique character abilities them having distinct fighting styles

- Both prefer having a variety of playable characters over following through with a story with one fighter
- Both are supportive of a retro arcade style for its fun and nostalgic appeal
- Daeniel believes that character customisation is a good idea, whereas jack is all about solid mechanics that have good functionality
- Both like the idea of having a 1v1, where you can play against a friend, as well having the AI to play against and practise

As there are some restrictions, such as graphical designing ability, sound engineering and time constraints are apparent, it may be the case that some of the suggested features will not be able to be designed.

To make sure their requirements are kept in the product, I will be conversing with the stakeholders during the entirety of the development process. After certain features are implemented, I will ask the opinions of the stakeholders and see what their feedback is and will act accordingly making sure there are no problems/unnecessary features. This will ensure that the end-product will be up to standards of the general users.

Research/Existing Products

After researching into different routes I can take to develop the game, I have come to the conclusion that I will be using C#, with the .NET framework. The C# scripts will be integrated with graphics and using Unity. This was chosen because lots of people who have made 2d fighting games believe this is the most efficient. As I will not have unlimited time to develop the product, I had to compromise fairly with time to develop, and the quality.

An important part of the research is to look at existing products which have correlations with the project I am planning and designing. There were a few games that stood out during the research for existing products, for various reasons.

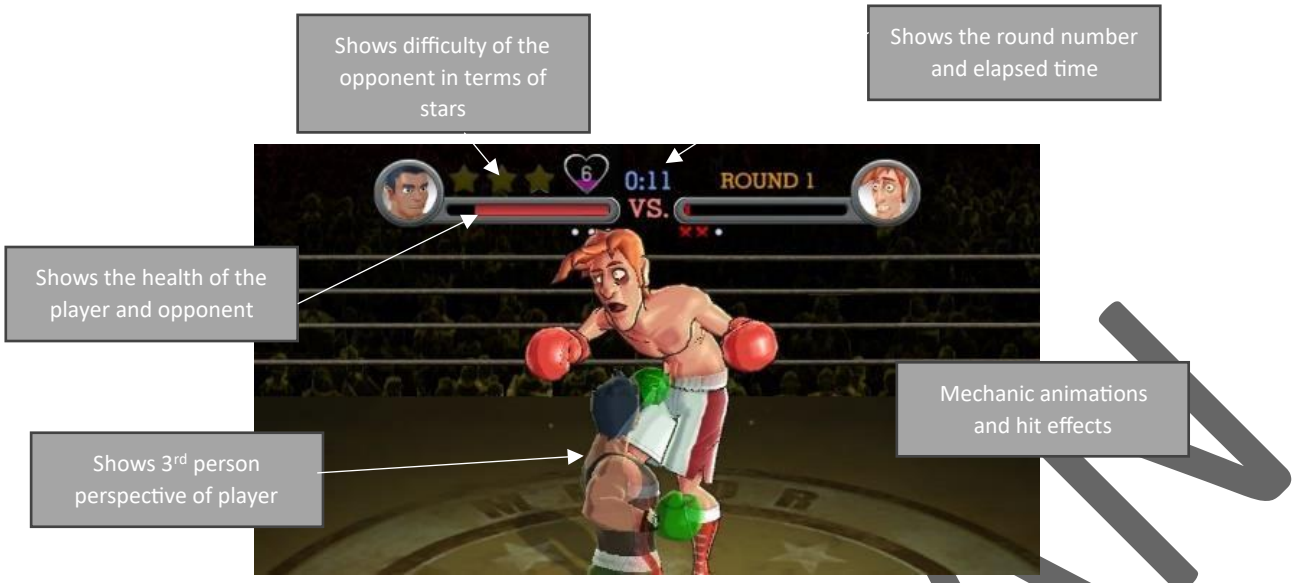
Punch-Out!!

About

Punch-Out is a single player 2d boxing game that was published by Nintendo in 2009. The game is known for the colourful characters, exaggerated animations, and strategic gameplay (Nintendo of Europe GmbH, 2009).

Nintendo decided to not publicly disclose, as they wanted to keep their development processes and technologies confidential. However, it is safe to assume that a variant of C or C++ was used to develop this game as a lot of the Wii *Giant bomb (2009)* games around this time were developed with these programming languages as they worked well with Nintendo's libraries and tools.

The game is played from a perspective from behind their character, which allows the user to anticipate and dodge incoming punches coming from the opponent.



Advantages

- Like a lot of 2d fighting games, punch-out emphasises timing and pattern recognition. This means players had to make sure to analyse how the opponent fights to exploit weaknesses and to ultimately defeat them.
- A feature I really like from this game is the progressive difficulty. As you progress and fight more Ais, the difficulty level of the opponent increases, meaning that players are required to continuously improve their strategies and skills.
- Punch-out!! Provides accessible gameplay making it easy to pick up and play, no matter how experienced you are in the gaming field. The simple controls ensured it was not too hard to learn and progress in the game. Disadvantages

- One of the limitations this game ran in to was that it did not include a multiplayer mode. This meant the game was restricted as only a single player game and did not allow players to connect a second controller and do a 1v1 mode.
- Another Limitation was the lack of customisation. Players were not able to change the appearances of their characters in any way.

Stakeholder analysis

Jacks experience - He liked the retro aesthetic of the game and felt it was enjoyable and easily accessible to any person, any age. The controls were simple but sophisticated enough so that it wasn't too easy and was still a bit of a fun challenge.

However, he did not like the fact that there were no multiplayer choices, and there was very little input from the user on how the story develops. You don't get to chose what paths you take or who you fight, it is very much hardcoded into the game. Although, He can sympathize with the fact that it is a remake of a retro arcade game and that it is technically a story-based game.

Features to Utilise

- Implementing progressive difficulty into my project would be effective as it introduces a challenging difficulty curve. Providing a challenge and increasing the difficulty after they have beaten the previous opponent means that players will feel a sense of accomplishment once they have beaten the newer opponent, and motivation to increase their skills even further, and ultimately make sure that they are consistently engaged.

- Working from one of the limitations, adding customer characterisation such as allowing players to change appearance and attire would give users a sense of ownership over their in-game avatar which in turn enhances engagement.

Fight Night Champion

About

Fight Night Champion is a boxing game released by EA Sports in 2011 which has story and multiplayer mode. The story is about a character named Andre Bishop who must overcome lots of challenges to rise through the boxing ranks to be the champion. During the game you fight lots of AI opponents with increasing difficulty as you go along (Electronic Arts, 2011).

Compared to the style I am going for, a retro 2d game, Fight night champion is a 3d realistic boxing experience.

Advantages

- A thing many people liked about the game was the visual and audio presentation. The game emphasised on the realistic graphics and the sound effects.
- The game had fighting mechanics which were extremely ahead of its time. It was the first fighting game which had total punch control which allowed players to throw punches with precise joystick movements instead of having to memorise complex combinations. This added depth to the game as it allowed players to feel the impact of throwing different types of punches.
- One of the best technical features of this game is the AI. In the story mode, every single opponent was a modified version of a generalised AI that was made. These different variations of the AI provided a different experience during the fights.

Disadvantages

- The training mode where you practise the controls and combinations used during fights was 'too simplistic' as it only focused on repetitive exercises and did not provide in-depth strategic advice. However, the game came with options where you can change the difficulty of the fights you play if you find yourself struggling.
- Although there was an online multiplayer feature, there was no 1v1 mode where you could connect a second controller and play against a friend. This created a bit of backlash as all other sport games at the time had a mode like this and people thought it would be a good idea.

Stakeholder Analysis

Daeniel – Being a fan of boxing, he found the visual graphics and the sound effects very similar to real life boxing. This encouraged him to carry on playing the game, even though it was a bit too easy.

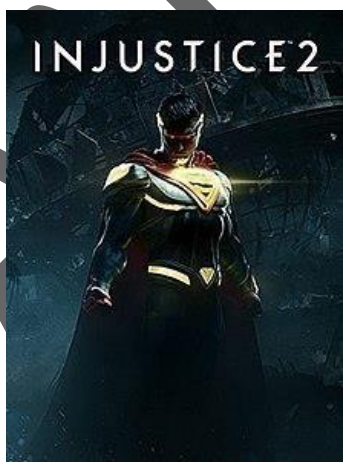
He believed that the game was too easy up until the very last fight, where he found it too difficult. Due to this he found the game boring and played without any sort of challenge until the end. The developers of the game made the progression curve too steep which was a major problem.

Features to Utilise

- I will be trying to add AI into my project for the different opponents you can face.
- Building upon the AI, I will add a training bot where you can practise in more depth and learn combinations to fight against real opponents. This will mean that when people with no experience of fighting games will be able to learn and improve their skills so that they can enjoy the game.

Injustice 2 : Gods among us

About



Injustice is a fighting game that was released in 2013. The storyline of the game takes place in a DC Comics universe, where you can play the different characters which feature in the comics/movies (Wikipedia Contributors, Warner Brothers, 2019).

I have personal experience playing this game and found it to be one of the most fun fighting games I have played, mainly due to the crossover between fighting games and the DC universe which is its unique selling point.

This game is a 2.5d game. This means that the game is portrayed in a 3d world, while incorporating a 2d gameplay.

Advantages

- The extensive character customisation options means that players can personalise the heroes or villains that they see all the time in movies and comics. The customization adds depth and a sense of personalization to the game and allows players to create their own versions of iconic characters.
- The game has a competitive online feature which allows players to fight other people online. This feature means that people could enjoy testing their skills against others and can even take part in tournaments. This created a global community of players who enjoyed the game and extended the longevity of the game.

Disadvantages

- Many players found that the learning curve was too hard of a task to overcome for beginners. They believed that the various moves, combos and character-specific abilities were too complex which effected accessibility and enjoyment for some.
- Character balance was an issue in the game. Some players felt that some characters in “injustice 2” were more powerful or had better, more effective attacks compared to others. This could have impacted the competitive mode and made it unfair for some players.

Stakeholder Analysis

Jacks experience – He enjoyed the actual gameplay and thought it was very fluid. He also enjoyed the fact that all the characters had their own special abilities and believed the storyline of the game was perfect.

In contrast to what people generally liked, he believed that there was too much customization of characters. He explained that you had to level up each character you played individually and that levelling them up was a complicated process that required too much effort. He believed this ruined the core aspect of fighting games which is the “pick up and play” feature (being able to play the game the first time and learning it quite simply).

Features to Utilise

- I am going to add a roster of characters who all have different abilities and fighting styles. Having a variety of characters to play makes the game more exciting as you always have the opportunity to learn someone new and will never face the same opponent too many times. This will in turn increase the overall engagement of the user.

Essential Features

So far, I have been given useful features and advice from stakeholders and have taken inspiration from existing products. I would like to implement all of the useful features I have talked about so far but it may not be possible due to time constraints, so I need to filter out what is essential (needs) and what is optional (wants).

Needed Features of end-product:

- Contain a boxing themed GUI with start page including "Start", "Controls" and "Settings" and boxing ring for the fights. In the boxing ring I must have health bars at the top for keeping track of the game, and also a screen especially for the winner.
- The certain mechanics that are required are left and right movement, punching, blocking and crouching. The blocking and crouching will need to be differentiated so that there are pros and cons to each one for players. Blocking will reduce the damage taken while it is held, and it will require less energy. Crouching will totally nullify the punch as it has been "dodged", however the con to this is that it will require more energy. This means that the players have the chance to decide for themselves which move is fitting for their strategy.
- Health and energy control is needed. Health signifies which character is winning at that moment in time, and if it reaches zero, that player has lost. Energy control is helpful deterring people who will punch constantly, and always keep their block up, this will make the game fairer for both players. These will have to be synchronised with health/energy bars which are a visual representation so the players know the progress of the game.
- The game will have two player, and single player mode where they can fight an AI opponent which has different levels.

The reason these features are "needed" are because they are the main components of 2d fighting games. The GUI helps users easily navigate around the game, the fighting mechanics allow the players to actually make use of the game and having an AI that will react to different inputs will allow users to have fun and practise.

Wanted features of end-product

- Player characterisation was recommended by one of my stakeholders and was highly appraised in the game Injustice 2. It would be a great for user engagement, but in terms of creating a high-quality fighting game, it isn't required.
- Adding sound effects and high-quality animations/graphics is not required as I am going for retro 2d fighter style, which often had simple animations and backgrounds.
- A story mode will not be added due to stakeholder feedback as they both believe the game will be best fitted for a 1v1 multiplayer game.
- The game will mainly be player vs player, but I will work on a simple AI model to use for a "training mode" so that users can practise their skills without having someone to play against.

Limitations of Proposed Solution

Being a beginner developer making a game like this can come with many problems such as:

LIMITATION	JUSTIFICATION	EFFECT ON GAME
OPTIMISING PERFORMANCE (SOFTWARE)	As the game is being developed, I may encounter performance issues with frame rate or inefficient memory use. Optimising code includes making use of efficient algorithms, which may be difficult for a beginner. One reason why this may occur is because for the two player game mode, there will most likely be scripts, which do very the same things for each of the players due to them being separate game objects. Solving this problem is possible, however it requires a much deeper understanding of Unity itself, and not something that can be fixed from the code. This will take a lot of time to fix, which is tricky considering the time constraints imposed on me.	The game may take up lots of space in the RAM, causing it to be slow. It could also take up a lot of secondary storage.
VISUAL DESIGN (SOFTWARE)	Creating all the assets that are required such as animations, sprites, sound effects and background art can be time-consuming and will require skills beyond just programming. This is quite important because the game will only work with a suitable GUI which has to be designed, however things like well-designed sprites and realistic sound effects are not as important as the gameplay will still be functional.	The game may not be very visually appealing to the users. In the market, this could be a big problem as people would not want to spend money on a game which has not been designed properly.
MULTI–PLATFORM SUPPORT (SOFTWARE)	The game will not be available for users on different platforms such as IOS, Android or consoles. This requires different versions of the game to be created which are compatible for use on different platforms with other instruction sets/architecture. For example, on a mobile device, I would have to create different controls for touch screens.	Many people will not be able to play the game as they don't have the required device
NON-INTUITIVE CONTROLS (HARDWARE)	Since there is a two-player mode, it can be hard to fit all the controls for both players onto one keyboard, while making them close together so that players do not get confused. The left player will have common controls with the WASD and F keys for all their actions, but the player on the right will have IJKL and H as these are similar but on the opposite side of the keyboard. This layout is uncommon and can be quite confusing to get a grip of for players.	It may get very confusing for beginners to grasp the controls, which will lead to them keep having to look at the controls screen from the menu.

System Requirements

There will be various software requirements needed for users to run the game:

SOFTWARE	JUSTIFICATION
WINDOWS 7, 8, 10, 11 OR MAC OS	The game is not specifically OS specific as C# can be run on many different platforms, but the game is not available on phones (IOS or Android)
.NET FRAMEWORK	This is the framework for C# and is needed for the game to run
DIRECTX (WINDOWS) OR METAL(MAC OS)	These are graphics API's and they are responsible for rendering the graphics on the users screen. Unity will utilise these API's specifically to ensure optimal performance

Basic hardware requirements:

HARDWARE	JUSTIFICATION
KEYBOARD	For input to play the game
MOUSE	To navigate the game menus
MONITOR	To display the game to the user
2GB SECONDARY STORAGE	There will be lots of scripts/sprites/animations etc which require space
AROUND 2GB RAM	This is an average amount of RAM required for unity games

Success Criteria

The success criteria are important for the evaluation of the project once it has been developed, it allows me to measure the success of the project, by comparing it with pre-defined features. Each of the criterion will be coded depending on the category they are in for the development:

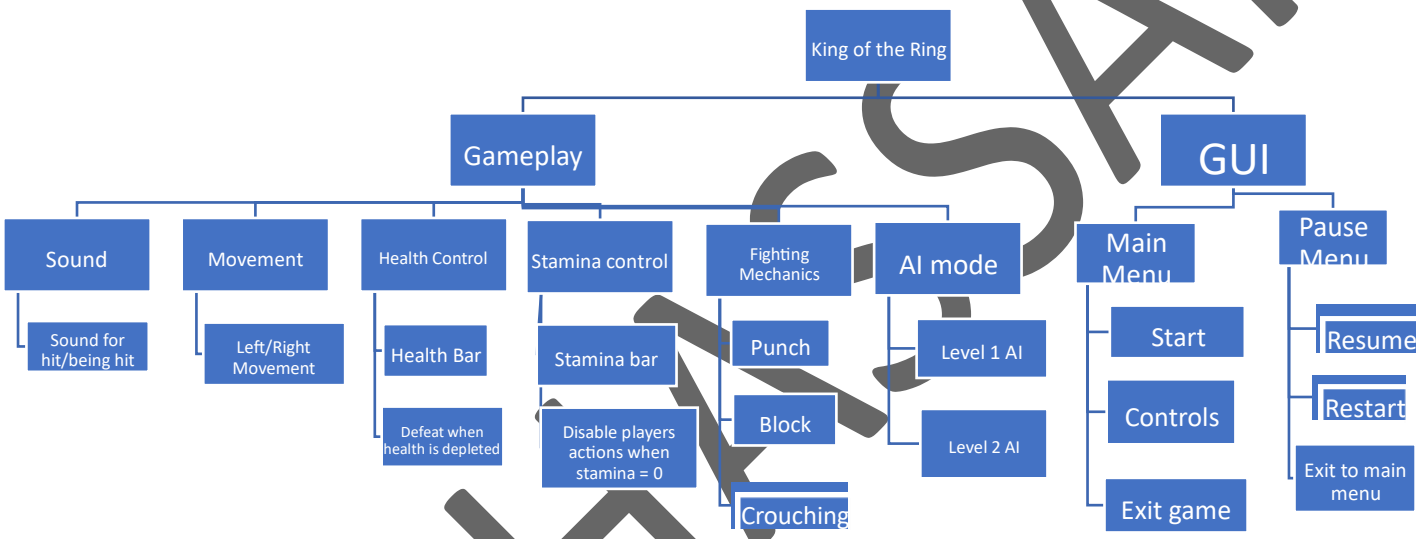
- GM = Gameplay mechanics
- GD = Graphic design
- AD = Audio/sound design
- UI = User interface

SUCCESS CRITERIA	CODE	DESCRIPTION	MEASURABLE (
ATTACKING MECHANICS	GM1	Fully functional attacking mechanics can be used in relation with hitboxes.	100%
BLOCKING MECHANICS	GM2	Having blocking mechanics where you can block incoming attacks, will negate any damage that you were meant to take.	100%
MOVEMENT MECHANICS	GM3	Allowing the player to move around the screen.	100%
HEALTH CONTROL	GM4	Having a fully functional health control system which determines the outcome of every round	100%
ENERGY CONTROL	GM5	An energy control system which will inhibit excessive punching without a structured plan to winning.	100%
SPECIAL ABILITIES	GM6	Each character will have one unique ability which they have to charge up and can use on opponents.	100%
LEVEL 1 AI OPPONENT	GM7	A mode for fighting an AI bot will be used for players trying to improve their skills. This AI should be hardcoded to do random moves	100%
LEVEL 2 AI OPPONENT	GM8	An AI which responds to the playstyle of the opponent	100%
BALANCED EXPERIENCE	GM9	Due to characters having different special abilities, some may say that certain characters may be higher powered than others due to their unique ability, so I must make sure that I make all the abilities equal in impact.	50%
SPRITE DESIGN	GD1	The different characters will have different designs pixel art designs.	100%
BACKGROUND ART	GD2	Will require art for the menu background, and the background for the fighting arena.	100%
ANIMATED ACTIONS	GD3	Simple animations for different gameplay mechanics such as punching and blocking as well as showing the impact of attacks.	100%
BACKGROUND MUSIC	AD1	Having music playing in the background which fits the theme of the game enhances the experience.	100%
SOUND EFFECTS	AD2	The sound effects when someone has been hit, or has blocked enhances the impact of attacks and creates an immersive boxing experience.	100%
START MENU	UI1	Created a boxing themed start menu with buttons for starting a game, controls and training mode.	100%
PAUSE MENU	UI2	Created a pause menu which allows users to return to game, restart and exit to main menu	100%

SUITABLE HUD	UI3	Heads up display will show the characters being used, the charge for their special attack and the health bar.	100%

Design

Problem Decomposition



The function of problem-decomposition in this style is so that each smaller sub-problem can be solved with one single task and can be entirely focused on separately to the others, meaning you can work on producing high quality functions at all stages of the development and it allows for a more organised approach, using the sub-problems as a checklist. Once each sub-problem is completed, we can also test to see if they work separately, allowing for errors to be detected straight away instead of creating the whole game and having to search for reasons why it may be producing errors at the end.

I have broken the game down into two main sections:

- Gameplay experience
- Graphical User Interface.

The reason for this was due to them being very separate parts of the final solution and having different focuses in the development stage. All of the sub-problems routing from the gameplay section are more focused on improving the experience the user feels when playing the game, and the graphical user interface section is more for the user satisfaction, and ease of use. These are both very important for my game as I would like to satisfy users experience in all ways. The graphical user interface sub-problems require more “design” work which differs to gameplay, which is more logical based coding.

As mentioned above, each sub-problem can be represented by a single task. The tasks can be explained and how they are approached below:

SUB-PROBLEM	PURPOSE	PROCESS
SOUNDS	The sounds for being hit add extra feeling of realism and you being inside the game playing as your character	When the hitbox for the attacking sides arm hits the others body, a sound file will be played which sounds like you are being hit.
MOVEMENT	A game like this would not work if the characters couldn't move left and right on the screen. In normal fighting games you can jump, but this is not normal practise in boxing so I will not add that feature, but I will add a feature for crouching to dodge.	I will create movement for the sprites using scripting in unity. To make it seem like the character is moving I will design a character animation by using two different images of them walking which will switch every time you go a few steps forward. This should make it look like the character is walking.
AI MODES	AI practise bot is for the players to have something to practise against, so they can build upon their skills and can beat their opponents. There will be 2 levels, easy and hard.	The easy AI will be picking actions based on weightings assigned to that are hardcoded where something like punch will have a higher probability than blocking, whereas the harder AI will be more adaptable. The real players actions will be recorded and read, and new weightings will be assigned to each action continuously allowing the AI to choose the best fitting action while still keeping it unpredictable.
HEALTH CONTROL	The health element allows players to know whether they are winning or losing, as the person with the least health is losing at that time. The health bar will be a visual representation of this so that the players can quickly and easily assess the progress of the game	I will create a variable for each of the players health with a set starting value and will be affected by each attack. This will be integrated with a visual cue which will be the health bar shown on screen.
STAMINA CONTROL	The stamina system will punish people who repetitively press the punch key with the hopes of winning quickly and easily. This is because when their stamina reaches zero, they are unable to do any actions until it regenerates back. There is also a visual representation of their stamina in the form of a stamina bar so they can assess the best way to win without depleting their stamina.	There will be a stamina variable which holds the value of the stamina between 0 and 100. Punching, blocking and crouching will all cost stamina and so that variable will update consistently during the game. The health bar will be attached to that variable and will update at the same time as the variable.

FIGHTING MECHANICS	Fighting mechanics (punch and block) will control the flow of the whole game. They will be made by characters having multiple sprites which would have design for each of the mechanisms. Crouching is part of this sub-procedure as well because it influences health and stamina change	These sprites will have different hitboxes which on collision do different things. For punching, the arm will have a hitbox and if this collides with the opponent's body, the health will decrement. And the block is the same however the health will decrease by a smaller amount. Crouching will completely nullify the damage taken, as nothing would have hit you.
MAIN MENU	This will have different features such as to start the game, which will switch to the arena scene. Controls will display all the controls needed to play the game. Exit the game is another feature.	Starting the game will change the scene from the menu to the main game arena. This requires me to make buttons which link scenes together.
PAUSE MENU	The pause menu is like the main menu. It will be opened using the "esc" key and will have a button for resuming the game, a restart button which does the same thing as the "start game" on main menu as well as quit to the menu which does the opposite of "start game".	This section requires the same functionality as the main menu in the way that scenes need to be linked together so that there can be smooth transition between them. These main features all need to be made as buttons.

Structure of Solution

The structure of the solution will be developed in an order that makes logical sense, so that the subroutine's which are dependent on others, will be done after the first one is done. However, it is a good idea to list the independent subroutines, which allows for choice for an order to be developed.

Independent:

- Movement (Horizontal)
- Punching and Blocking / crouching
- Both menus can be created and designed, with dysfunctional buttons, and once the full game is created with all the different scenes, the buttons can simply be linked to the other scenes. - Sprites and animations

Dependent:

- Sound effects depend on the fighting mechanics because a sound will play after a player has been hit
- Health control also depends on the fighting mechanics, so that health can be updated after every successful hit
- AI feature depends on the 1v1 to be fully functional, as it will be a copy of the 1v1 scene, however the scripts will be changed to include the AI's algorithm for one of the fighters.

The order that makes logical sense in which to build the game is:

1. The sprites and animations will be made before the start of the development of the game, so I can iteratively test whether they are working during the development process.
2. Create the movement mechanics, as this is a standalone feature and requires no combining with any other part of the game.

3. Punching/Blocking Mechanics because a lot of the other subroutines are dependent on this one being complete. Crouching will also be added to this as it s part of the fighting mechanics
4. Sound effect control as it is quite small and simple to develop.
5. Health control. This depends on the punching mechanics as well and is necessary for the game to function.
6. Stamina control. Also depends on the punching/blocking mechanics as these functions are what the stamina variable reacts to.
7. The AI opponent game mode. The gameplay features will now be finished, but the GUI needs to be completed so it is usable.

Once these are done, the game will be playable, however the GUI still needs to be created:

8. Create the health bar and stamina bar
9. Backgrounds for the gameplay
10. Main menu design
11. Pause menu design
12. Link the scenes together so that when buttons are pressed, they take you to the correct scene.

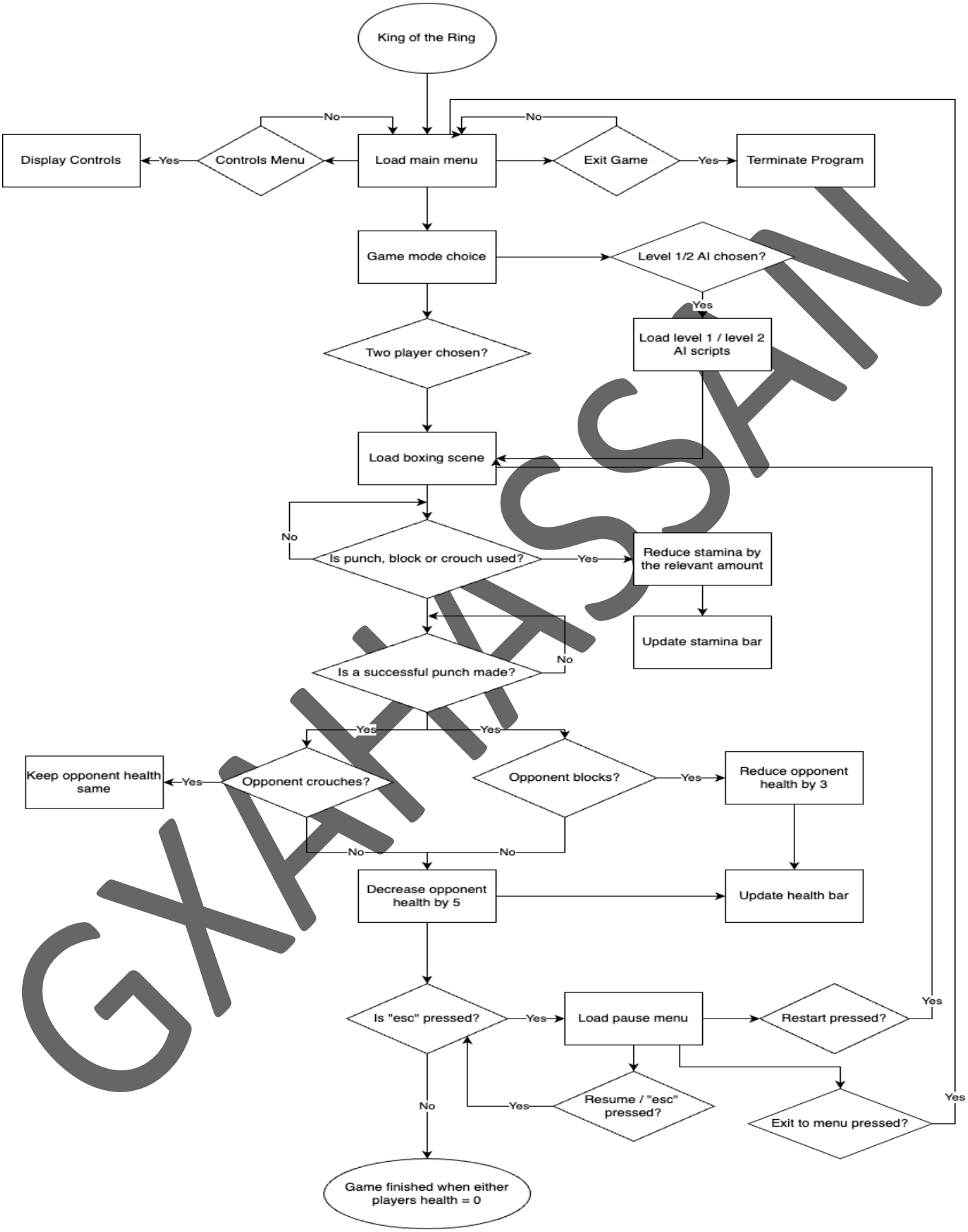
Algorithms

As I am using Unity, there is extra functionalities which come with Unity libraries which I will be using to aid me with development.

*when player1/2 is used, it means that this line will be repeated for both players, as they both have separate scripts.

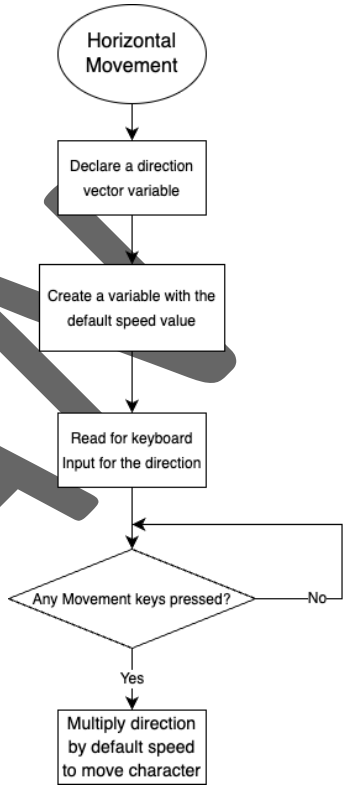
Hitboxes are areas within the game space that are used to detect collisions between two game objects, and can be varied in size of my choice

Flowchart for full game:



Horizontal Movement:

```
1. Define movement class:
2. Private float direction
3. Private float speed = default movement speed
4. Update():
5. Read direction input from player
6. FixedUpdate():
7. Set the horizontal velocity of player body to (direction *
   speed)
```



Explanation of each line:

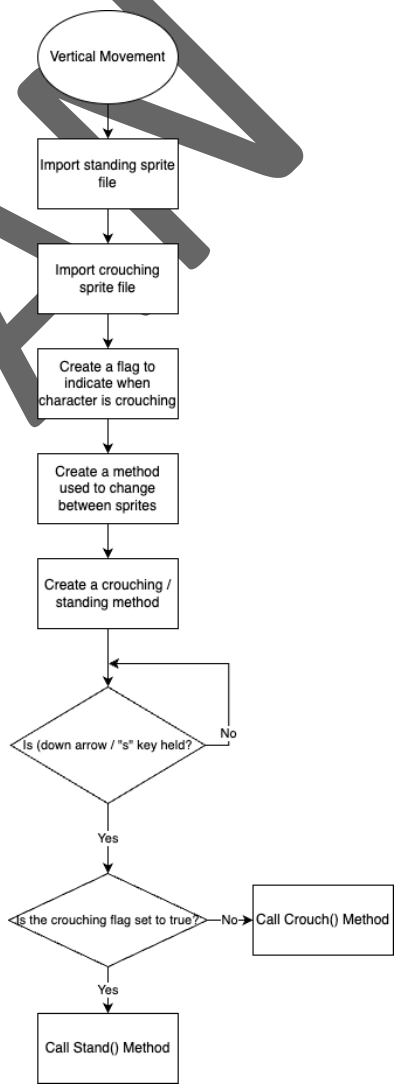
1. Creating a class to control the movement of the boxer.
2. Creates a private variable called direction.
3. Defines a variable named speed with a set value to be decided.
4. The update method is used in unity and the code within is carried out every frame.
5. Every frame, inputs from the arrow/WASD keys will be checked and should be assigned to a variable. This should produce a vector quantity from 1 to -1, where 1 will mean movement to the right, -1 means movement to the left and 0 means no movement at all (no input).
6. Fixed update is a method in which is completed every frame but will be linked with the physics engine within unity.
7. We can make the player body move in the direction of the input by multiplying the vector quantity of the direction by the speed.

Variables and Data Structures:

VARIABLE/DATA STRUCTURE NAME	TYPE	PURPOSE	JUSTIFICATION	VALIDATION
MOVEMENT CLASS	Class	This will contain all the variables and code for the movement of the character.	It allows for consistency within the movement, if another part of code isn't correct, the movement will remain working.	-
DIRECTION	Private Float	Holds a vector value from -1 to 1, where -1 indicates movement to the left, and 1 indicates movement to the right, depending on the input from user.	By doing this, we add acceleration to the player movement for a small amount of time, which adds a smoothness to the movement.	-
SPEED	Private Float	This is a default speed, which will have a set value that is multiplied by the horizontal variable to produce movement in the direction by "Horizontal"	This variable ultimately allows velocity to be worked out by multiplying it by the direction of movement.	The set value remains constant

Crouching:

The only vertical moving feature is to crouch, which will be used for dodging punches. It will be made by having switching between different sprites, which represent standing and crouching. While carrying out this mechanism, the hitbox of the character will have a different size and shape meaning that a direct punch that would have connected if the player was standing, would now not be registered as a hit.




```
1.     playerStanding = "playerStanding.file"
2.     Define crouching class
3.     Private playerCrouching = "playerCrouching.file"
4.     isCrouching = false
5.     Create characterSpriteChange method
6.     Update the sprite dependent on parameter
7.     Create Crouch method
8.     Call characterSpriteChange(crouchingSprite)
9.     Change flag isCrouching = true
10.    Create a stand method
11.    Call characterSpriteChange(standingSprite) 12.
       Change flag isCrouching = false
13. Update():
14. If "down arrow" OR "s" key is held down:
15. If isCrouching = false: 16.          Call Crouch()
17.         Else:
18.         If isCrouching = true:
19.         Call Stand()
```

Explanation of algorithm:

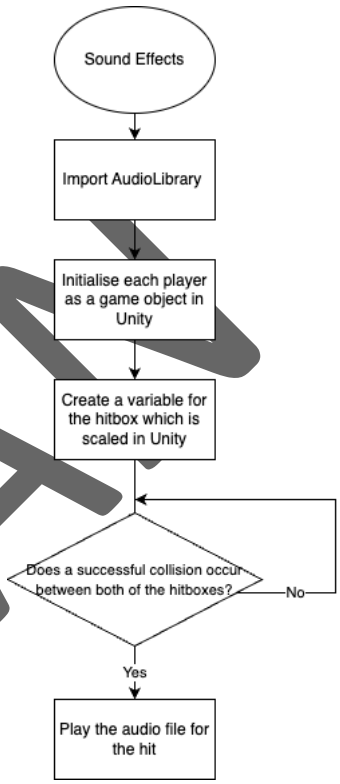
1. Create a class to contain all crouching code
2. Assign an image of the sprite standing
3. Assign an image of the sprite crouching
4. Create a flag indicating whether the player is crouching or not
5. Create a method to switch between the different sprites
6. Switch the sprites depending on which one has been stated in the parameter
7. Create a method to make character crouch
8. Change the sprite to the crouching sprite
9. Set the flag to true
10. Create a method for character to stand
11. Change sprite to standing
12. Set flag to false
13. Update section is for instructions that need to be checked every frame
14. Create an IF statement with conditions to see if any of the keys mentioned are held down

15. Have a nested IF statement to check whether the character is standing or crouching
16. If they are standing, call the crouch function to make them crouch 17. Else (If the keys are not being held)
18. And if they are crouching,
19. Make them stand by calling the stand function Key Variables and Data structures:

VARIABLE/DATA STRUCTURE NAME	TYPE	PURPOSE	JUSTIFICATION	VALIDATION
STANDING SPRITE	Sprite	Assigns a reference to a file that will be added to the game file within unity	This is the image of the sprite standing as normal	-
CROUCHING CLASS	Class	Will contain the code for the crouching function	Allows for consistency with the crouching ability.	-
CROUCHING SPRITE	Sprite	Reference to the image of the player crouching	Image of the sprite crouching, which is displayed when the crouch button is used	-
IS CROUCHING	Boolean	Flag which says whether the player is crouching or not at that point	The flag helps with the logical flow and allows the computer to understand when the player is crouched or not	Ensuring the flag represents the correct state of the player

Sound effect algorithm

From my research, I have found that having sound effects when a successful punch hits the opponent's body is a small feature that has a big effect on the overall experience. To carry this out, I will be using an AudioLibrary library.



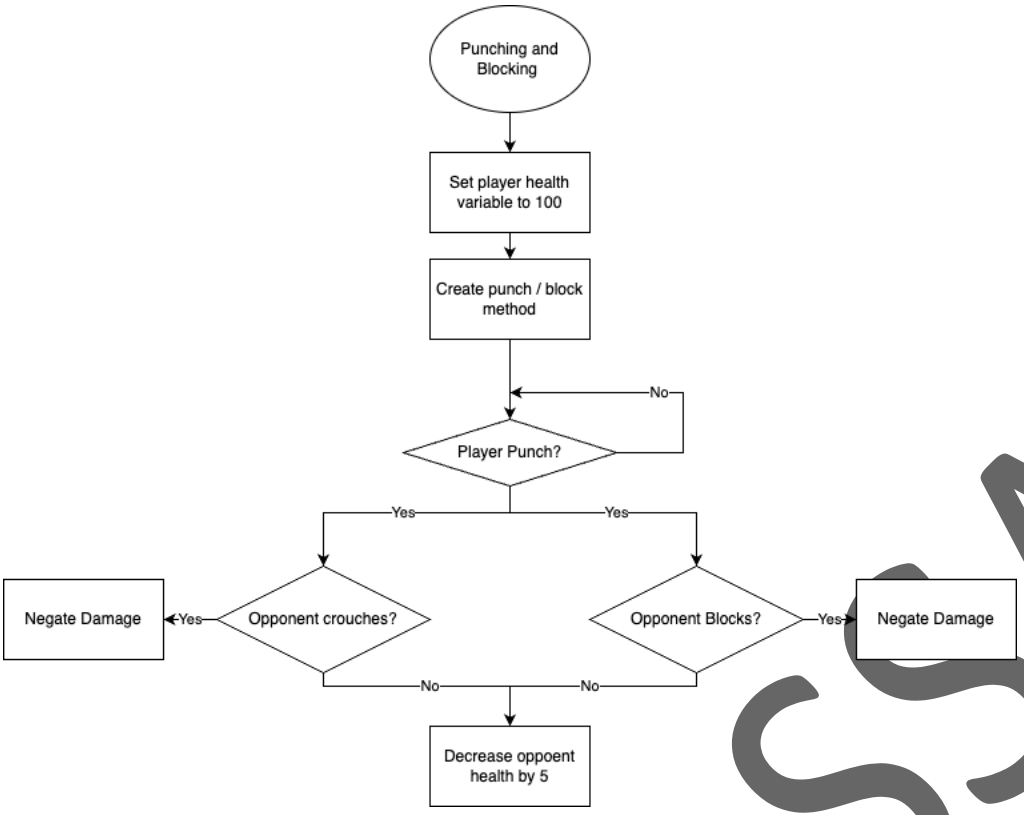

```
1. Import AudioLibrary
2. Player1/2 = GameObject("Player1/2") 3. Hitbox1 =
   Player1/2.CreateHitbox()
4.     Update():
5.     If Hitbox1.CollidesWith(Hitbox2):
6.     AudioLibrary.PlaySound(punchsound.file)
```

Explanation:

1. This is a necessary library so that sound files can be played
2. Initialises which player is which game object within unity
3. Same for the opponent player
4. Creates a hitbox which can be edited to fit the player sprite within Unity
5. Same for opponent player
6. Checks for any updates every frame
7. IF the hitbox for the first players arm (when they punch), hits the other players body,
8. Then play the sound file Key Variables:

VARIABLE/DATA STRUCTURE NAME	TYPE	PURPOSE	JUSTIFICATION	VALIDATION
PLAYER (1 AND 2)	GameObject	This assigns a name to the game object which is created within unity	It allows for a hitbox to be created for each of the players	-
PLAYER (1 AND 2) HITBOX	Hitbox	This assigns a name to the hitbox that is created and shaped within unity	This hitbox will indicate when there are collisions between the two sprites	-

Punching and Blocking Mechanics:



Each player will start with 100 health, and now, the health will decrease by 5 for each successful punch. There will also be a transition between two sprites to produce an animation for each punch, with the punching sprite having a different hitbox so that health can be updated accordingly.

However, if the player receiving the punch is using the blocking feature, they will not receive any damage.

```
1.     Player1/2 Sprite = “playersprite1/2.file”
2.     Player1/2Punch Sprite = “player1/2punch.file”
3.     Player1/2Block Sprite = “player1/2block.file”
4.     Create characterSpriteChange method
5.     Update the sprite depending on parameter passed in
6.     Player1/2PunchHitbox = Player1punch.CreateHitbox()
7.     Player1/2BlockHitbox = Player1punch.CreateHitbox() 8.
       Player1/2IsBlocking = False
9.         Start():
10.        int Player1/2Health = 100
11.        Update():
12.        IF '..' or '..' key is pressed:
13.        characterSpriteChange(Depending on button pressed)
14.        IF Player1PunchHitbox.CollidesWith(Player2):
15.        Player2Health -= 5
16.        Else If Player2PunchHitbox.CollidesWith(Player1):
17.        Player1Health -=5
18.        IF '..' is held AND not Player1/2IsBlocking: 19.
           Player1/2IsBlocking = true
```

```
20.     Else:
21.     Player1/2IsBlocking = false
22.     IF Player1/2IsBlocking:
23.     characterSpriteChange(Player1/2BlockingSprite) 24.     Player1/2Health
        remains unchanged
25.     Else if not Player1/2IsBlocking:
26.     characterSpriteChange(Player1/2Sprite)
27.     Player1/2 Health decreases by 5 as normal
```

Explanation:

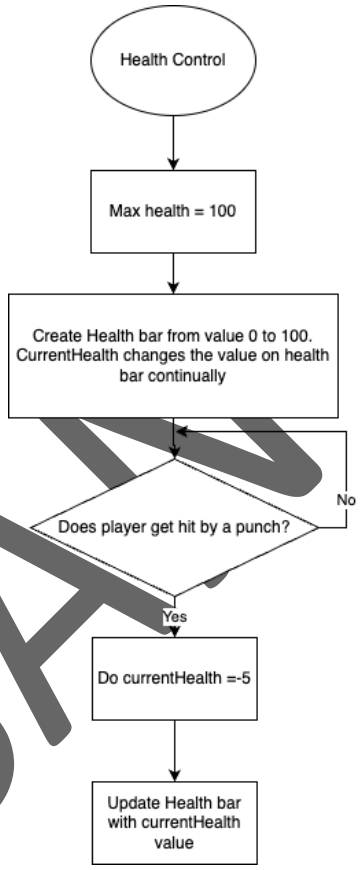
1. Define the initial sprites of both boxers
2. Load the separate sprite made for the punching motion
3. Load separate sprites made for blocking motion
4. Create a method used for switching between different sprites
5. Switch the sprites depending on button pressed and parameters
6. Create a new hitbox for the punching sprite of the character which Is edited within Unity
7. Create a new hitbox for the blocking sprite of character
8. Boolean flag used to see if player 1 or 2 is blocking
9. Start method completes code as soon as the script is run
10. It will set players health to 100
11. Checks for updates every frame
12. If one of the buttons which will be assigned for punches is pressed,
13. It will change the sprite dependent on which button is pressed to the punching sprite
14. A nested if/else if statement, checks if the punching hitbox of Player1collides with the non-punching hitbox for the Player2,
15. Then player2's health will decrease by 5
16. Else if the punching hitbox of player2 collides with the non-punching hitbox for player1,
17. Then player1's health will decrease by 5
18. If a certain button is pressed (for blocking) AND if the player is not blocking
19. Set the flag to true dependent on the player
20. If they are not pressing the button
21. Set flag to false
22. If the player is blocking
23. Change sprite dependent on the player
24. And do not change the health of the receiver of the punch (no damage is dealt)
25. Else if they are not blocking
26. Change the sprite to default sprite
27. And decrement punch receivers' health by 5

Key variables and data structures:

VARIABLE/DATA STRUCTURE NAME	TYPE	PURPOSE	JUSTIFICATION	VALIDATION

PLAYER DEFAULT SPRITES	Sprite	The default sprite of the character	This is needed so that there can be transitions between different sprites when certain functionalities are used	-
PLAYER PUNCHING SPRITE	Sprite	Different sprite for when a punch is thrown	When a punch is thrown, there needs to be visual representation of this within the game.	-
BLOCKING SPRITE	Sprite	Different sprite for blocking	There needs to be visual representation of a block.	-
PUNCHING HITBOX	Hitbox	As there is a change of sprites when a player is punching, there is also a new hitbox	Allows the program to monitor whether a punch has connected or not.	-
BLOCKING HITBOX	Hitbox	New hitbox created especially for the blocking animation	Allows the program to understand when there is a collision between a punching sprite and a blocking sprite.	-
SBLOCKING	Boolean	Flag which indicates if a player is blocking or not	Is used to see whether the damage of a punch has been negated or not.	True / false
HEALTH	Int	A numerical value of 100, used for representing health.	Every time a successful punch has been made, the health will decrease by 5.	Health updated on screen

Health control:



As stated in the fighting mechanics design, each player begins with 100 health, and the health will decrement by 5 every time a successful punch has hit from the opponent. The health variable will be integrated with a visual "health bar" which is a common UI feature on lots of 2d fighting games. It will show a visual representation of the health of each character, so that the players can change up their fighting style if needed, to ensure they win. For example if a player sees they are lower health, they may play more defensively so that and looking out for openings to attack to level out the playing field.

```
1. Int MaxHealth = 100
2. Float CurrentHealth 3. healthBar = slider
4. Start():
5. currentHealth = MaxHealth
6. healthBar.MaxValue = MaxHealth 7. healthBar.Minvalue
   = 0
8.  Update():
9.  healthBar.Value = currentHealth
10. IF player is punched:
11. TakeDamage()
12. update healthBar
```


Explanation

1. Declare max health to be 100
2. Create a float value which holds the value of health and is updated continuously
3. Set a variable called healthBar to a slider, which is a UI component in Unity which can be used to produce the visual representation of the health.
4. All code in the start function is done when game is started
5. The current health is reset to the max value
6. We will set bounds for the slider, so that it has the correct length. The full bar will be max health
7. The bar will be empty when player is at 0 health
8. Code in update function will be checked for every frame
9. Since the value of the health will be always changing, it makes sense to make the length of the health bar be updated every frame depending on the currentHealth
10. If the player is punched:
11. Use a TakeDamage function which will decrease the current health by 5
12. The changing current health from the previous function, will mean that the health bar can also be updated Key Variables and data structures:

VARIABLE/DATA STRUCTURE NAME	TYPE	PURPOSE	JUSTIFICATION	VALIDATION

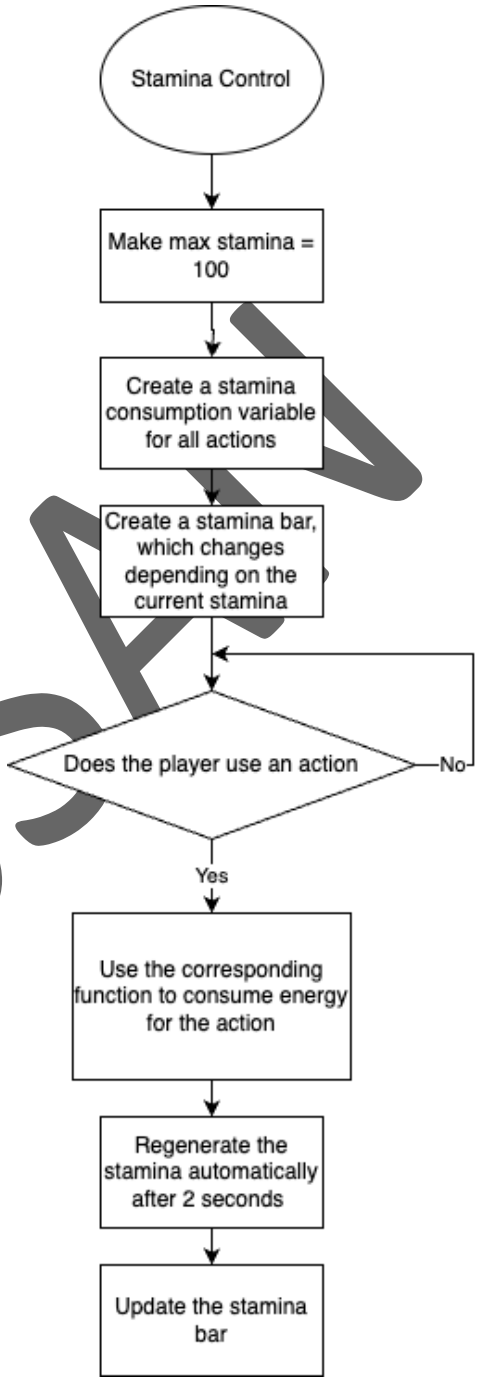
MAXHEALTH	int	Holds value 100, which is set as the initial health of each player	This is needed so that whenever the game is restarted, the initial value of health is reset for both players. It is also needed to set the max length of the slider for the health bar.	Value = 100
CURRENTHEALTH	float	Constantly changing value for the health, updated during the game	This is required to track the progress of the game, and to know who is winning or not. This is the value that is repeatedly changed from punches and will be the main piece of information on the health bar.	-
HEALTHBAR	slider	This is a user interface component to show the players how much health they have left	This is necessary so that it can easily be checked by either player briefly to get an understanding of who is winning the current fight	Fixed Min value = 0. Fixed Max value = 100. However, it has a changing health value from the currentHealth variable

Stamina (energy) Control:

Both players will have stamina, which depletes as actions are done. Punching will use a set amount of stamina, whereas blocking and crouching can be held down for any time, so the blocking and crouching will slowly decrease the stamina over time. I would like the blocking to use less stamina than crouching due to it not fully nullifying the damage delt to them, to make it so that there is a differentiable reason as to why they would choose one of the functions over the other. Just like the health, there will be a stamina bar present for the player to see on the GUI.

Both players will begin with 100 stamina. A punch will decrease the stamina by 5, blocking will decrease the stamina at a certain rate depending on how long they hold the button for blocking, and crouching will decrease the stamina at a faster rate than blocking. The functions will be accessed as part of the stamina class in the punching/blocking and crouching scripts.

```
1.      Define stamina class
2.      maxStamina = 100
3.      int currentStamina
4.      blockingStamina = 10
5.      crouchingStamina = 15
6.      punchingStamina = 5
7.      slider staminaBar
8.      Start()
9.      currentStamina = maxStamina
10.     Max staminaBar value = maxStamina 11.
        Update()
12.     staminaBar value = currentStamina
13.     regenerateStamina()
14.     Public regenerateStamina()
15.     currentStamina =+ 5 (every 2 seconds)
16.     Public drainStamina(value)
17.     currentStamina -= value * time button held
18.     Make sure energy stays between 0 and 100
19.     Public punchConsumeStamina(value)
20.     currentStamina -= value
21.     Make sure stamina is between 0 and 100
```



Explanation:

1. Create the stamina class
2. Set the max value of stamina to be 100
3. Create a currentStamina variable so that it can be changed during the progress of the game repeatedly
4. Set the rate that blocking will drain the stamina at
5. Set the rate that crouching will drain the stamina at
6. Set the value that punching will decrease the stamina by
7. Create a slider object (unity UI element)
8. Does the code in here when game starts
9. Assign the currentStamina to the value of maxStamina = 100
10. Make the stamina bar full
11. Run through this code continuously
12. Always adjust the stamina bar according to the current stamina
13. Run the regenerateStamina function
14. Create the regenerate function
15. Continuously add 5 to the currentStamina variable in time intervals of 2 seconds
16. Create the function for draining energy (used for crouching and blocking)
17. Make the currentStamina go down by the draining rate * the time the button is held down for
18. Clamp the stamina between 0 and 100 so that the stamina can never be out of this range
19. Create the function for consuming energy with a punch

20. Decrease CurrentStamina by a value when pressed
21. Same reason as above

VARIABLE/DATA STRUCTURE NAME	TYPE	PURPOSE	JUSTIFICATION	VALIDATION
STAMINA	Class	Will have the values for stamina, and public methods such as punchConsumeEnergy and drainEnergy	In the script containing the logic for punching, blocking and crouching, these methods will be used to alter the value of the stamina when they are used, so that the stamina bar can be adjusted accordingly	-
MAXSTAMINA	int	Has a value of 100, which is the max stamina	This is needed so that whenever the game is restarted, the initial value of stamina is reset for both players. It is also needed to set the max length of the slider for the health bar.	Value = 100
CURRENTHEALTH	float	Constantly changing value for the stamina, updated during the game	This is required so that the players cannot repeatedly do actions without wasting all of their energy. It requires the players to think strategically when to use certain actions. When it reaches zero players will not be able to complete moves, but this logic will be a part of the classes containing the action' logic.	-
BLOCKINGSTAMINA	float	This is the rate at which the stamina will drain by while the blocking key is pressed	Since blocking is a continuous action, that can be held down, it requires a rate at draining stamina which differs from a set value like there is for punching.	-
CROUCHINGSTAMINA	Float	Same purpose as the blockingStamina but holds a greater value as it drains more stamina	Same reasoning blockingStamina variable	

PUNCHINGSTAMINA	int	Holds the set value that stamina goes down by when a punch is thrown.	As a punch is not a continuous "hold down button" action, the value of this variable is what is subtracted from the stamina	Set value = 5
STAMINABAR	slider	This is a user interface component to show the players how much stamina they have left	This is necessary so that it can easily be checked by either player to get a rough idea of how much stamina they have left, and what actions they should use.	Fixed Min value = 0. Fixed Max value = 100. However, it has a changing health value from the currentHealth variable

Level 1 AI:

The level 1 AI will carry out random actions, which have assigned weightings. The weightings will determine which action is more likely to be carried out, by assigning a higher value to that action. However, it is still a random process because even though a certain action has a higher probability of being chosen over another, it doesn't make the probability of choosing the other action 0. This is an important feature as it imitates unpredictable behaviour that human players will often use so that users cannot predict every move that will occur, which would make the game quite boring and will not be a test of if a player is truly skilled or not. Another feature that would make the AI less predictable, is to have the actions to carry out for a random time, between two values.

```
1.      Define Level1Ai class
2.      minDuration = 1
3.      maxDuration = 3
4.      Actions = [Punch(), Block(), Crouch(), Left(), Right()]
5.      Public function ChooseNextAction():
6.      weightedActions = [5, 1, 1, 2, 2]
7.      Have the function choose a random action based on the weightings
8.      currentAction = (chosen action)
9.      Choose random value for time in between min/max Duration
10.     Public function PerfomAction(action):
11.     Have a switch-case statement, which chooses the correct function
12.     Update():
13.     ChooseNextAction()
14.     PerformAction(currentAction)
```

Explanation:

1. Create the class for the level1 AI
2. Create a variable storing the minimum time that an action can be played for

3. Maximum time that an action can be played for
4. Create an array which holds all the actions as functions, which will be accessed from the relevant classes
5. Create a function used to decide on which action to be played next
6. Assign the weightings to each function, where the index of the weight matches the index of the functions
7. Allow the code to make a random choice on which action to use based on the weightings
8. Assign a function the currentAction variable
9. Allow code to choose a random value between the min and the max duration for the duration that the action is performed
10. Create a function used to perform the certain action depending on the currentAction variable that is chosen in the choose next action function
11. Get the name of the function passed in as the parameter, and perform the relevant function in a switchcase statement
12. This code is repetitively executed every frame while the game is being run
13. Always choose the next action when the last action is over
14. Perform the new action

Key variables and data structures

VARIABLE/DATA STRUCTURE NAME	TYPE	PURPOSE	JUSTIFICATION	VALIDATION
LEVEL 1 AI	Class	This class will contain many methods which will be used in the level 2 AI	It will increase efficiency as I will not have to re-write functions	-
MIN DURATION	integer	Contains minimum value an action can be played for	It is required so that when the random duration of time is chosen, it is not too short	Value = 1
MAX DURATION	integer	Contains maximum value an action can be played for	Used so that the randomised duration of time is not too long	Value = 3
ACTIONS	Enumerator	Contains all the actions that can be used	Allows weightings to be attached to each index, and for one action to be chosen randomly	-
ACTION WEIGHTINGS	Array	Contains the weightings for each action	Allows the AI to randomly chose an action using the weightings	-

Level 2 AI:

This is a development of the first level AI, which will consider many other features of the game progress, such as:

- The user's stamina/health
- The AI's stamina and health
- What actions the other player has used

These are the main conditions that will be used to determine the actions that the AI uses. However, instead of using a new algorithm, or using hardcoded responses based on what conditions are met, I will make use of the weighting system that was used in the previous AI. Instead, these conditions will alter the weightings assigned to each action. The weightings will be assessed and processed using statistical formulas to create new weightings.

Bayes' Theorem:

After some research into which statistical formulas to use, I have found that Bayes' theorem would be a suitable fit, this is because it considers multiple factors which fit the aspects the desired AI. Bayes' theorem states:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Where:

- P(A) = Prior probability of action A (the weighting of that action)
- P(B) = Evidence of B happening is the probability that a certain condition is true (e.g. probability that the opponent player is blocking etc). These probabilities can be sampled and hardcoded in, as it will be difficult to decide based on the span of a single game as they are quite short.
- P(A|B) = The probability of doing an action given that the observed event B is occurring (e.g. the probability of throwing a punch, given that the other player's health is below 50 etc)
- P(B|A) = The likelihood of an external condition, given that the AI has done an action (e.g. the likelihood that the user is blocking, given that the AI has punched). This is once again difficult to work out from the small sample of data that can be collected from a single game, so it could be tested, and a hardcoded value could be calculated, and used for that specific action.

The probability found can then be multiplied by the previous total weighting to find the new weight for that action.

Just like before, the weightings will create a sense of realism, as there is not a 100% chance that the action used, is the highest weighted one (which is the most effective in most circumstances). This replicates natural human error and unpredictability.

This is a step up from the previous AI as it is now adaptable and will change its actions based on the user's playstyle.

Since this AI is an adapted version of the last one, I will be using methods/attributes from that AI script so that I do not have to recreate certain functions.

```
1.  Inherit behaviours from the Level1Ai class
2.  Get user Health
3.  Get user Stamina
4.  Get user lastMoves (stack which will be created in user players script)
5.  Function calculateProbability(priorProbability, likelihood, evidence):
6.  updatedProbability = (likelihood * priorProbability) / evidence
7.  updatedWeighting = updatedProbability * totalWeighting 8. Return
    updatedWeighting
9.  Update():
10. If one of the many conditions is met
11. calculateProbability(relevant parameters)
```

```
12.     chooseNextAction(updatedWeighting)
13.     PerformAction(currentAction)
```

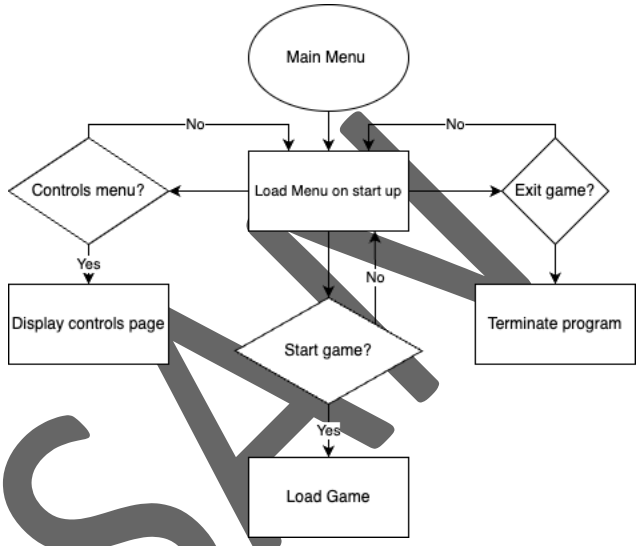
Explanation:

1. Insert a reference to the level1Ai script so that certain functions and values can be accessed
2. Retrieve current health from player script
3. Retrieve current stamina from player script
4. Retrieve recent moves from player script
5. Make a function which takes in all the necessary parameters to use Bayes' theorem
6. Calculate the probability of doing that action given the certain condition, making sure that it is normalised
7. Calculate the new weighting of that action
8. Return the value of the new weighting
9. Do this code consistently
10. Check for the conditions which require response such as other players health, stamina, last move or the AI's health, stamina etc.
11. Work out the new probability for a certain action
12. Choose the next action based on the new weighting that is passed into the function
13. Perform the chosen action Key variables and data structures

VARIABLE/DATA STRUCTURE NAME	TYPE	PURPOSE	JUSTIFICATION	VALIDATION
UPDATED PROBABILITY	float	Holds the new probability of choosing that action	Used to determine the new weighting for that action given that a certain condition is met	Value between 0 and 1 (as it is a probability)
UPDATED WEIGHTING	Int	Holds the new weighting for the action	Worked out by multiplying the updated probability by the total weighting. This is then used in the choose next action function	-
PRIOR PROBABILITY	float	Used to calculate updated probability using Bayes' theorem	It is calculated by the weighting of the action / total weighting, and passed into the function.	Value between 0 and 1.
LIKELIHOOD	float	Used to calculate updated probability using Bayes' theorem	It will be a hardcoded value for each action, that will be chosen after testing and sampling behaviours from players.	Value = chosen value after testing
EVIDENCE	float	Used to calculate updated probability using Bayes' theorem	Once again will be a hardcoded value for each action, after testing/sampling has been done	Value = chosen value after testing

Graphical User Interface:

There is a main menu and a pause menu which provides the player a straight-forward experience when navigating different sections of the game.




```
Main Menu:
```

1. Define Main menu class
2. Update():
3. DisplayMainMenu()
4. If "Start" button is clicked:
5. Switch to boxing ring scene 6. If "Controls" button is clicked: 7. Switch to controls page
8. If "Exit Game" is clicked:
9. Shut down program

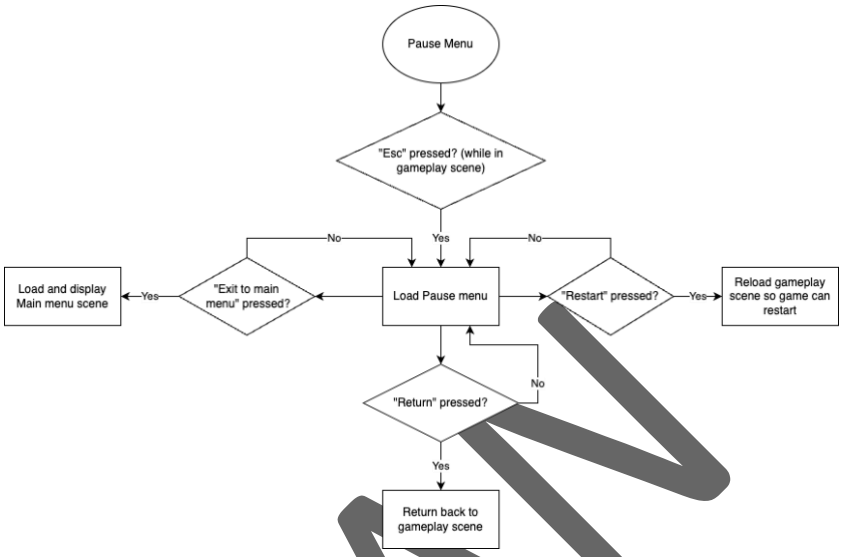
Explanation:

1. This will start up the game by loading all libraries and game data
2. Check for user input every frame
3. Function used to display the main menu on start up
4. If the start button is pressed,
5. Switch to the boxing arena scene
6. If "Controls" is clicked
7. Switch the controls page which contains instructions on how to play the game
8. If "Exit game" is clicked
9. Terminate the program

```
Pause Menu:
```

1. GameIsPaused = false
2. Define pause menu class
3. If "esc" key is pressed:
4. If GameIsPaused:

```
5.      Resume the game 6.
        Else:
7.          Display pause menu 8.
If GameIsPaused:
9.      Display Pause menu 10.
If "Resume" Clicked:
11.         Return to boxing ring
12.         If "restart" is
            clicked:
13.         Restart the current
            game in progress
14.         If "Exit to main menu"
            is clicked:
15.         Return to the main
            menu
```



Explanation

1. Flag used to indicate whether the game is paused or not
2. When escape key is pressed,
3. And if the game is already paused,
4. Then it will return to the boxing arena
5. If the flag is set to false, meaning the game is not paused
6. The pause menu will be loaded
7. If the game is paused,
8. We will display the pause menu,
9. And if Resume is clicked,
10. Return to the boxing arena
11. If restart is clicked,
12. Re-Initialise the game and load up the boxing arena,
13. If the exit button is clicked, 14. Display main menu.

Key Variables and Data Structures for both menus:

VARIABLE/DATA STRUCTURE NAME	TYPE	PURPOSE	JUSTIFICATION	VALIDATION
MAIN MENU	Class	Will contain methods for switching between different scenes within the game such as StartGame() and ControlsMenu()	Having a class helps keep all the code for the menu organised.	-
PAUSE MENU	Class	This will also contain methods for each button that is present on the menu.	The class will allow for the methods within it to be similar and increases reusability	-
START	Image	A button to transition to the boxing arena	Required for the game to being	Takes into game scene

CONTROLS	Image	Button to transition to the instructions page	Provides a usability feature	Displays all the controls
EXIT GAME	Image	Button that terminates the program	Allows for ease of use	Terminates program
RESUME	Image	Button which allows you to return from the pause menu back into the gameplay	Allows for ease of use	Takes user back into the game
RESTART	Image	Button which allows users to restart the current game in progress	Allows for ease of use	Restarts the current match
EXIT TO MAIN MENU	Image	Button which allows users to return to the main menu, where they can either look at the controls again, or exit the game.	Allows for ease of use	Takes user to main menu

Usability Features

When designing a game, certain choices need to be made into how the game can look presentable, while also being easy to use. This balance must be kept so that users can have an enjoyable experience. Within this, multiple features of the

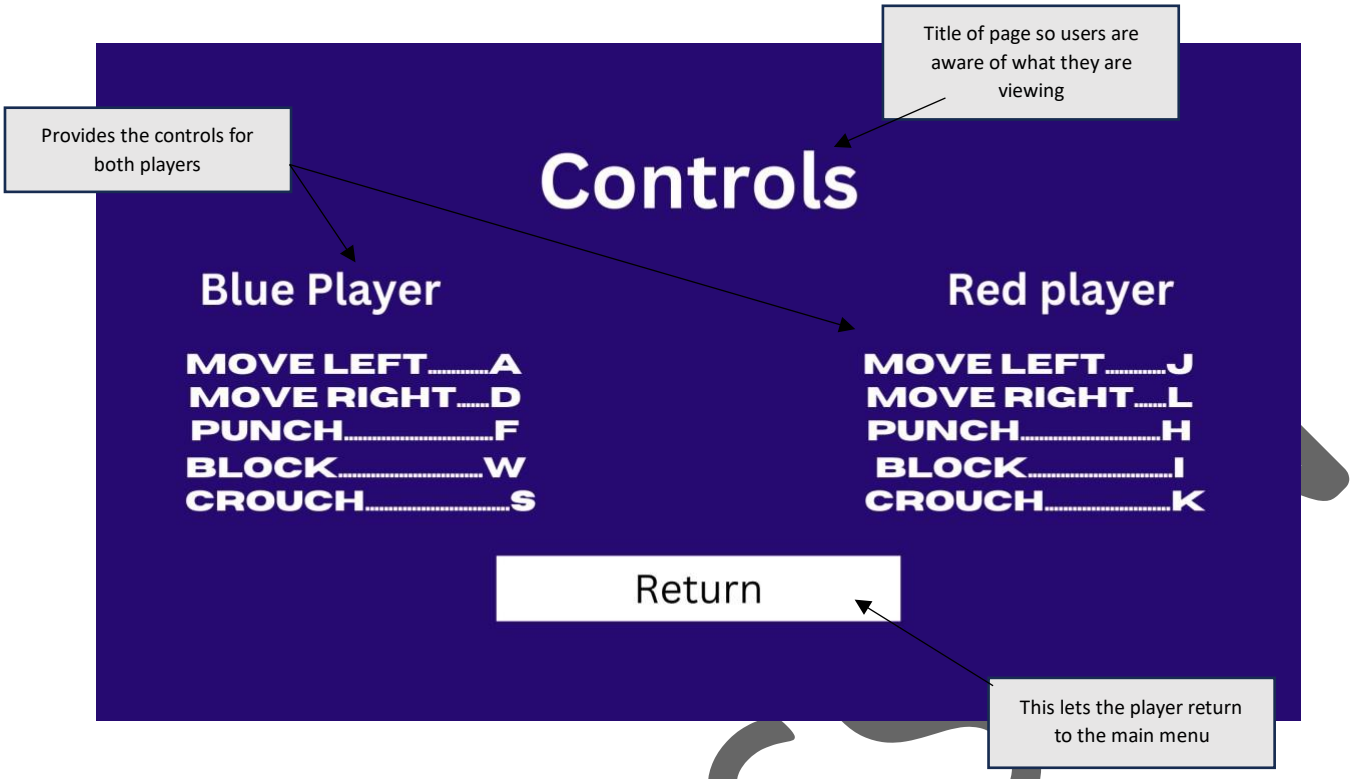
game need to be considered, from the navigating to different scenes, to the gameplay itself. The following features have been added to make the game as user friendly as possible:

USABILITY FEATURE	JUSTIFICATION
SIMPLE UI	The main menu and pause menu will both be very simple with limited functionalities to ensure that it contains all that it needs to while being easy to navigate.
CONTROLS AND CONTROLS PAGE IN MAIN MENU	This page is necessary to allow users to understand more about how to play the game. It will have all controls for both players and a brief explanation of how some of the functions work (that aren't self-explanatory).
RESTART BUTTON IN PAUSE MENU	A restart button in the pause menu (which is opened by pressing "esc") is a great way for players to start their match again, without having to exit to the main menu and pressing play again. This feature allows for overall ease of use
HEALTH AND STAMINA BARS AT THE TOP OF THE SCREEN	These will both allow the user to understand the progress of the game. The health bar will be an indication of who is winning, and the stamina bar will make people decide on the best way to play so that they do not lose health, and also do not run out of stamina.

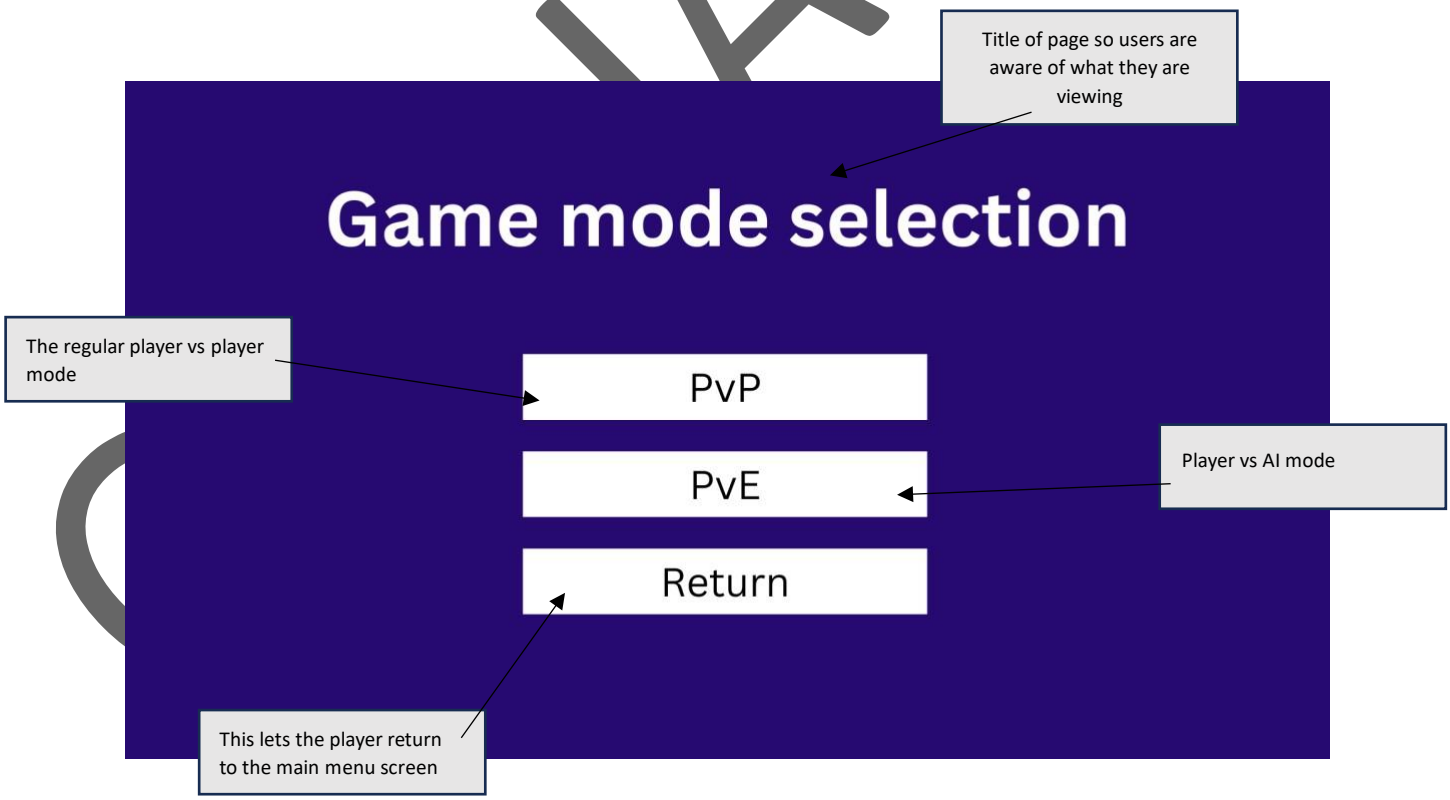
I have made initial designs of what I want the GUI to look like, with all the features that will make the game easy to navigate and understand.

Main menu:

Controls screen:



Game mode selection screen:



Difficulty of AI screen:

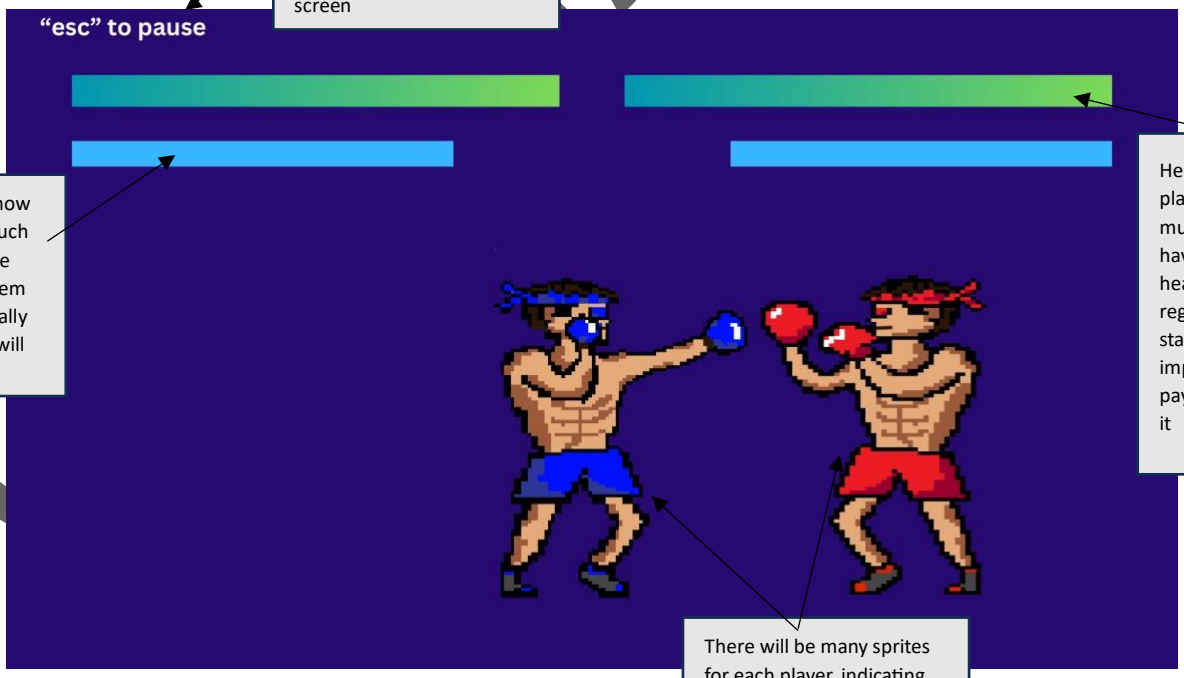
Title of page so users are aware of what they are viewing

Level 1 Ai is the easier model, which does picks a random move based on static weightings, and does not react to the moves of the user

Level 2 Ai is the harder of the models, because it reacts to the other users moves, health, stamina and its own health and stamina which means it can play more cautiously or offensively. It calculates the new weightings for each action using probabilities

This lets the player return to the main menu screen

Gameplay screen:



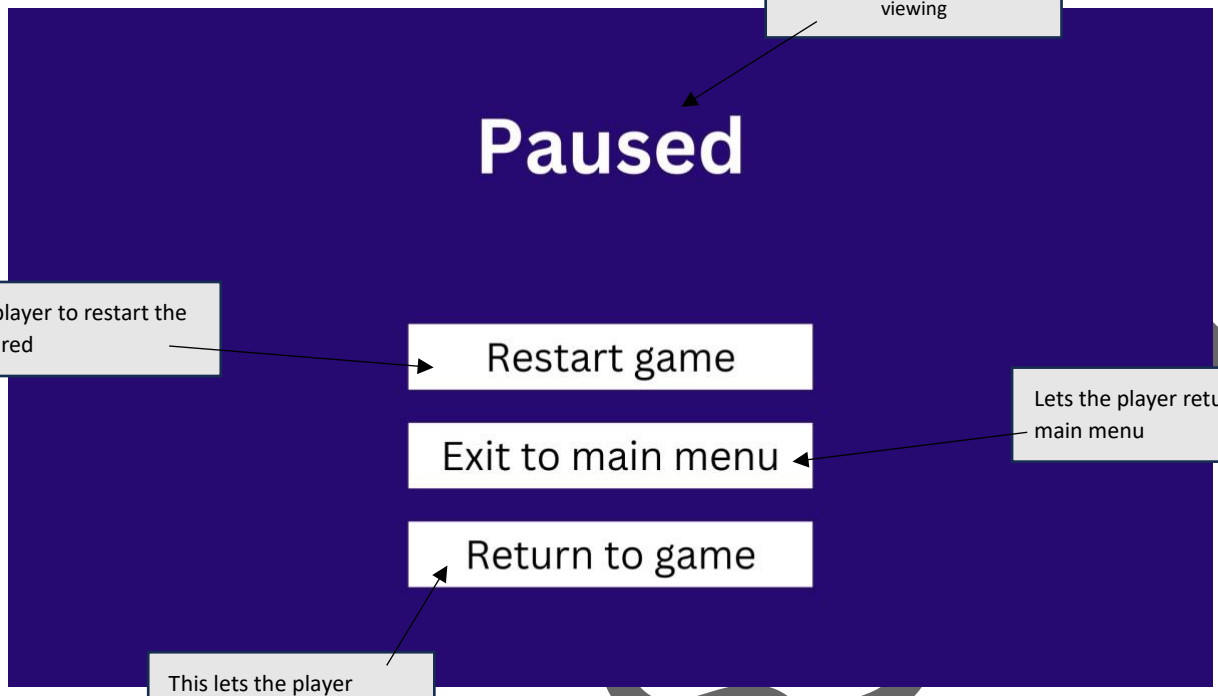
Tells the player how to pause the game if required, and will take them to pause screen

Stamina bar to show the users how much stamina they have left, and allow them to think strategically about how they will play

Health bar allows the players to see how much health they have remaining. The health does not get regenerated like the stamina, so it is important for users to pay close attention to it

There will be many sprites for each player, indicating what action they are doing at that moment in the form of an animation

Pause screen:



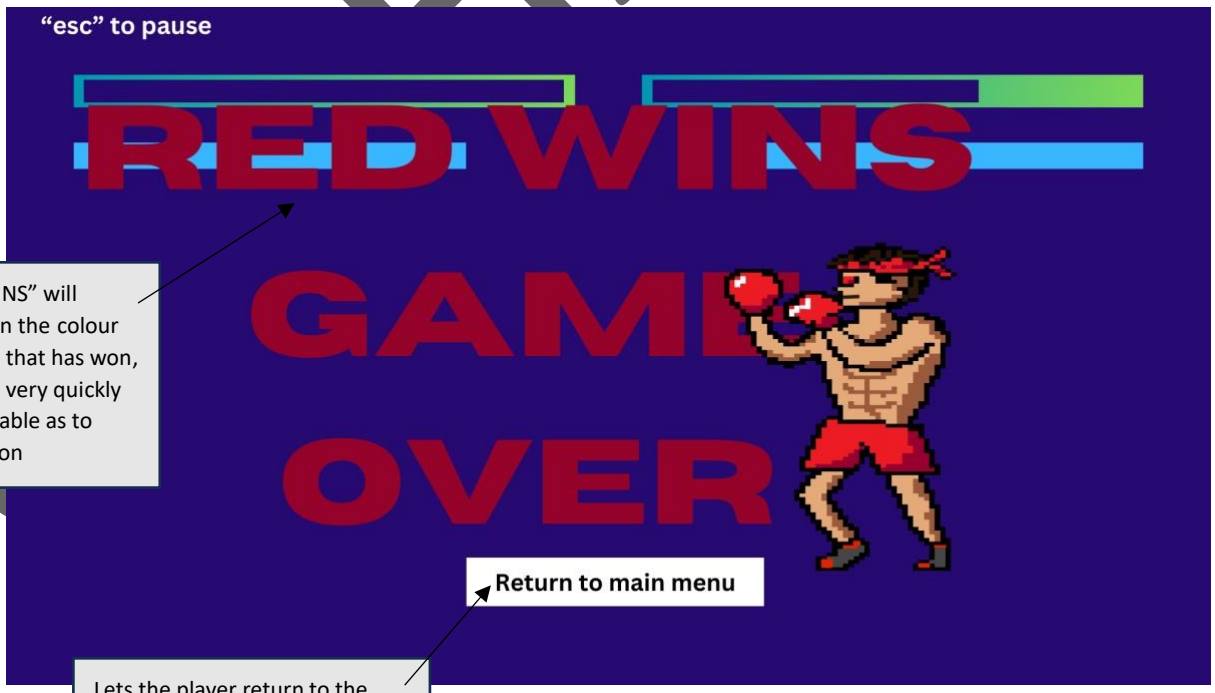
Title of page so users are aware of what they are viewing

Allows the player to restart the game if desired

Lets the player return to the main menu

This lets the player return to the game play

Game over screen:



The "…. WINS" will always be in the colour of the user that has won, so that it is very quickly distinguishable as to who has won

Lets the player return to the main menu so they can play again

Iterative Test Data

These are things that can be tested on its own during development.

SUBROUTINE	WHAT IS BEING TESTED	JUSTIFICATION	EXPECTED RESULT
GUI	Main menu window	When the game is loaded, the program should display the menu in the correct size to fit the display.	Correct Menu is shown
	Pause menu	Once “esc” is pressed, a new screen should show up with different options	Pause menu shown after input (“esc”)
	Functioning buttons for all the menus	Each button should be able to take input from mouse and redirect the user to the desired page.	Buttons for “Start”, “Controls”, and “Exit" take you to correct page
	Game over screen	It should display which user has won, and give the option to return back to the main menu	Shows who has won, and a return to menu button
MECHANICS	Left and right movement	This will test for horizontal, and to see if it works as it should after input has been fed into program	Players should move in direction dependent on their input
	Player boundaries	In Unity, the project can take up space outside of the bounds of the screen, commonly used for looping backgrounds In different types of games. I want to make sure that the player cannot move outside of the screen bounds.	Player is stopped if they are at, and trying to exit the bounds
	Punch mechanic	When the punch key is pressed, they should switch from one sprite to another.	Shows simple animation
	Crouch mechanics	When the designated key is held, the player should stay in a crouched animation until key is lifted	Shows crouching sprite until key is lifted
	Block mechanic	When block key is held, they should switch to blocking sprite	Shows blocking sprite until key is lifted
	Hitboxes	For every state of the player (idle, crouched, punching, blocking) there will be a new hitbox, to represent a different area of space that the player is occupying	Should be able to detect when collisions occur between hitboxes

	Animations	For the game to be playable, there must be animations for each of these actions, otherwise people will not know what actions they, or the opponent is using.	When the relevant key is pressed/held, the correct animation should show.
HEALTH CONTROL	Health (punch)	If the hitbox of one of the players fist collides with the other players body and they are not blocking/ crouching, a successful punch has landed.	Health decrement by 5
	Health (block)	If one players fist collides with the other players body while blocking, health will decrease by a smaller amount	Health decrement by 3
	Health (crouch)	If one players fist collides with the other players body while crouching, health will stay the same as they have completely evaded the punch	Health remains unchanged
	Health bar update	Health bar should show a visual representation of the variable decreasing, with the amount of health decrease being proportional to the decrease in length of the bar.	Health bar should show the health variable changing during the course of the game.
ENERGY (STAMINA) CONTROL	Regenerate Energy	For the stamina system to work as intended, it should regenerate after a given time so that the player is never left without a choice of moves for too long	The stamina will regenerate in fixed intervals by a varying amount, depending on the length of the interval
	Energy (punch)	Punching decreases the stamina by a set value which will be equal to 5	Stamina variable decreases by 5
	Energy (Block)	As blocking means that you can still take damage, it makes sense for the stamina to decrease at a lower rate	The energy should drain at a slow and constant rate
	Energy (Crouch)	Since crouching means that you take no damage at all, it will drain at a faster rate than blocking	The energy will drain at a faster rate than blocking
	Energy Bar	This feature needs to be tested because it is required for the game to be functional for users, so that they can see how much stamina they have	Will show a visual representation of the energy variable being changed

LEVEL 1 AI	Basic functionality of simple AI	The AI is built upon a randomised decision which factor in weightings for each action such as moving, punching, blocking or crouching	AI should be able to execute a sequence of predefined actions in a random fashion, since punching is weighted highest, punching should occur the most.
LEVEL 2 AI	Enhanced functionality and decision making	This AI should be able to make more sophisticated decisions based on contextual information of the games progress such as either players health etc	It should react to certain conditions optimising which move to pick the best, while making some "human-like" mistakes, as the best play should not always be picked
SOUND EFFECTS	Successful punch sound	Once a player is hit and decreases in health, a punch sound will be played.	When certain hitboxes collide, a sound will play.

Post development test data

These can only be tested once the whole program has been put together, as they rely on other sections of the program being completed.

WHAT IS BEING TESTED	JUSTIFICATION	EXPECTED RESULT
BOTH PLAYERS CAN SIMULTANEOUSLY MOVE AND DO ACTIONS	This checks to see if the program can handle multiple user's inputs. The game should be run on a computer that is capable of concurrent processing. A problem could arise that the program itself is not capable of dealing with the number of inputs.	Both characters should be able to move, punch, block and crouch simultaneously, without any drop in performance. A drop in performance could either be indicated by frame rate drop and could be due to inefficient code design, or insufficient computing power.
WHEN A PLAYER REACHES 0 HEALTH, GAME IS OVER	This depends on lots of other functionalities working such as punching.	When one of the players is defeated, the game should end.
BUTTONS ON BOTH MENUS SHOULD TAKE YOU TO CORRECT PLACE	The buttons are checked post development due to them requiring other scenes to be fully completed, so they can all be linked together	The main menu will have three buttons and the pause menu will have three buttons. Overall, the 6 buttons will link 4 scenes (Boxing arena, pause menu, Start menu and controls page).
VALID/INVALID INPUT TESTS	It is easiest to test this right at the end of development, instead of iteratively as it will require all of the actions to be made correctly, and to have an assigned input key.	When a valid input key is used, the relevant action should play. When an invalid input key is used, the game should not do anything in response to this.
TEST THE SUITABILITY OF THE AI'S	Once both have been made, I need to test not only their functionality, which will be tested in the iterative testing, but how well they do what they were designed to do.	Level 1 AI should be playing in a random style, while still not being too easy to beat. The level 2 AI should be harder to beat/ be able to last longer against a real opponent, and should react to different playstyles etc.
LOAD TESTING	Test whether certain aspects will make the game slow due to their memory/CPU requirements. If I can run it on my mid-high end system, without any issues, then I can ensure that any laptop/pc with similar or better specification will be able to run the game smoothly as well. However I can not ensure the same for lower-end systems due to lack of access to systems like that.	The most recourse intensive part of the game will probably be the level 2 AI, due to the collection of recent moves, and the amount of calculations happening all at once in a very short time.

Development

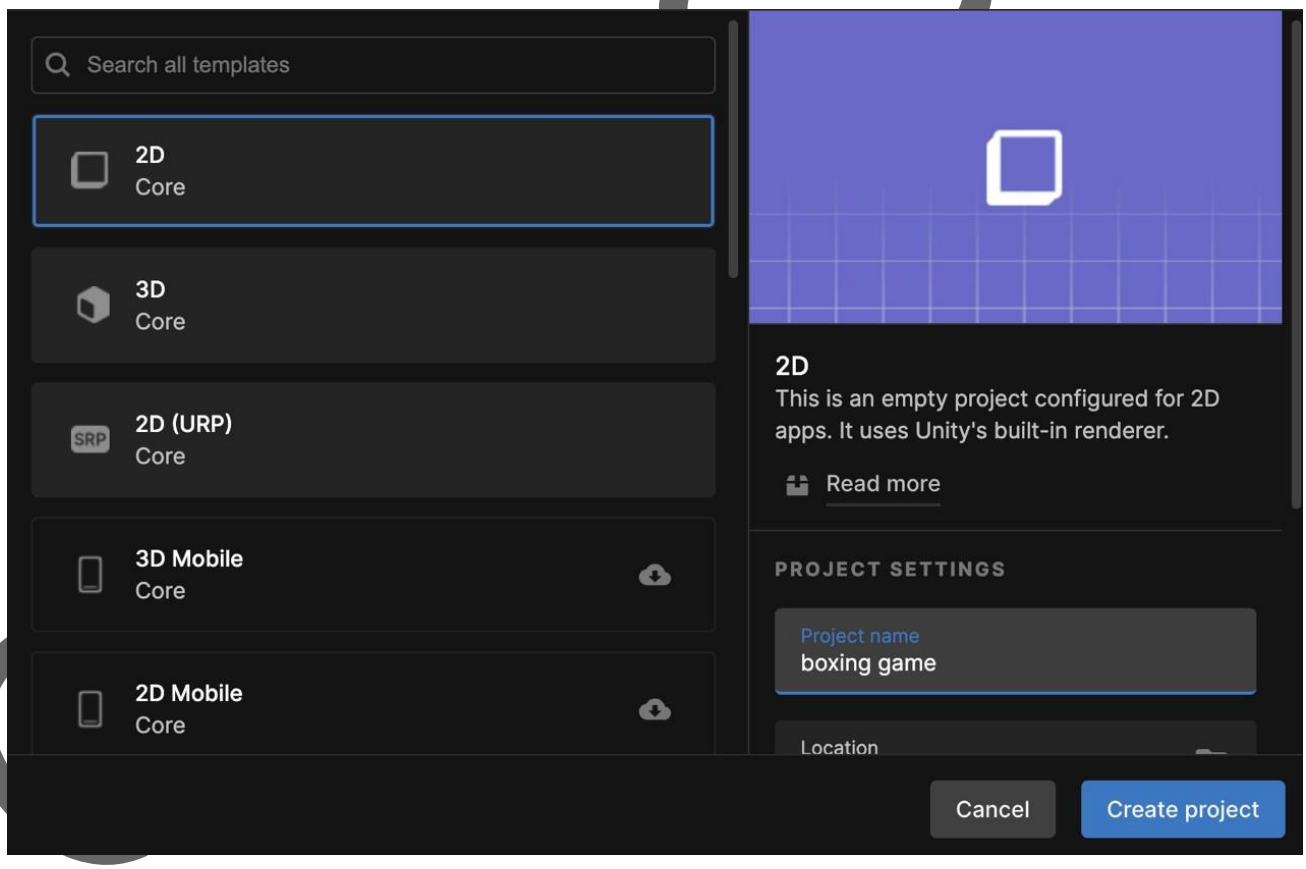
As mentioned in the limitations of the proposed solution section in the analysis, optimising the code can be a challenge for my project specifically as there will be two separate players. A few of the scripts will most likely be very similar to corresponding scripts for the other player, and after researching how to solve this, I have found that it is solvable, but requires a deeper understanding of Unity itself and not something that can be solved simply with the code. It will also take a lot of time to do which is a problem as I only have a short period of time to complete this project.

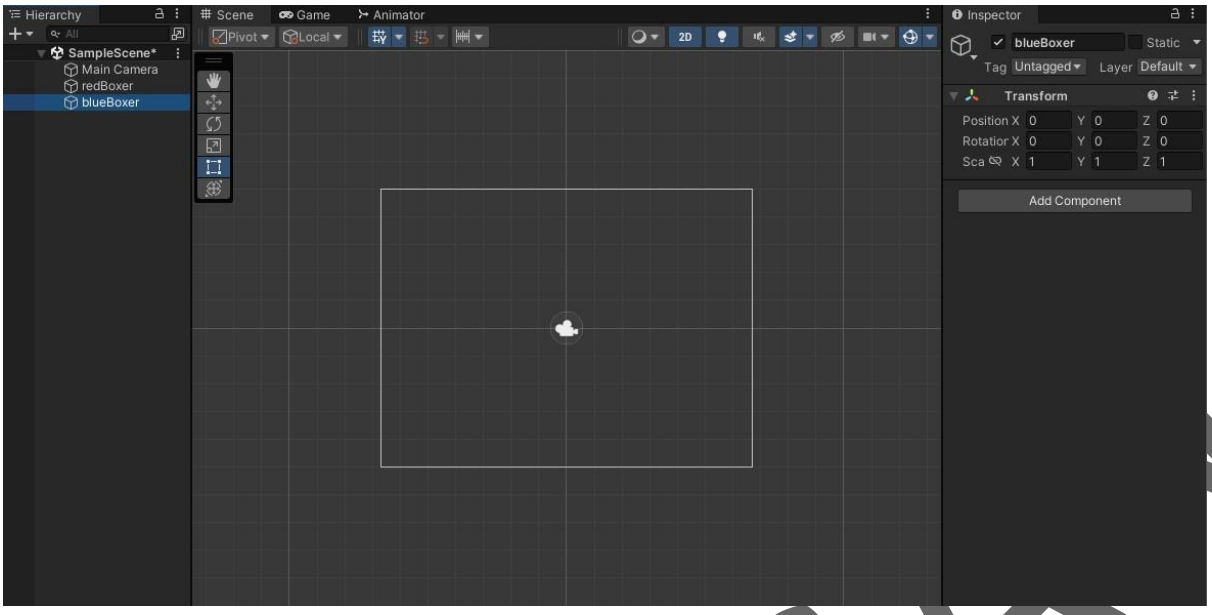
In the majority of the sections in the development stage, I will annotate the script for one of the players in depth, and will leave a screenshot of the other players script as It will be very similar with minor changes.

Setting up project in Unity

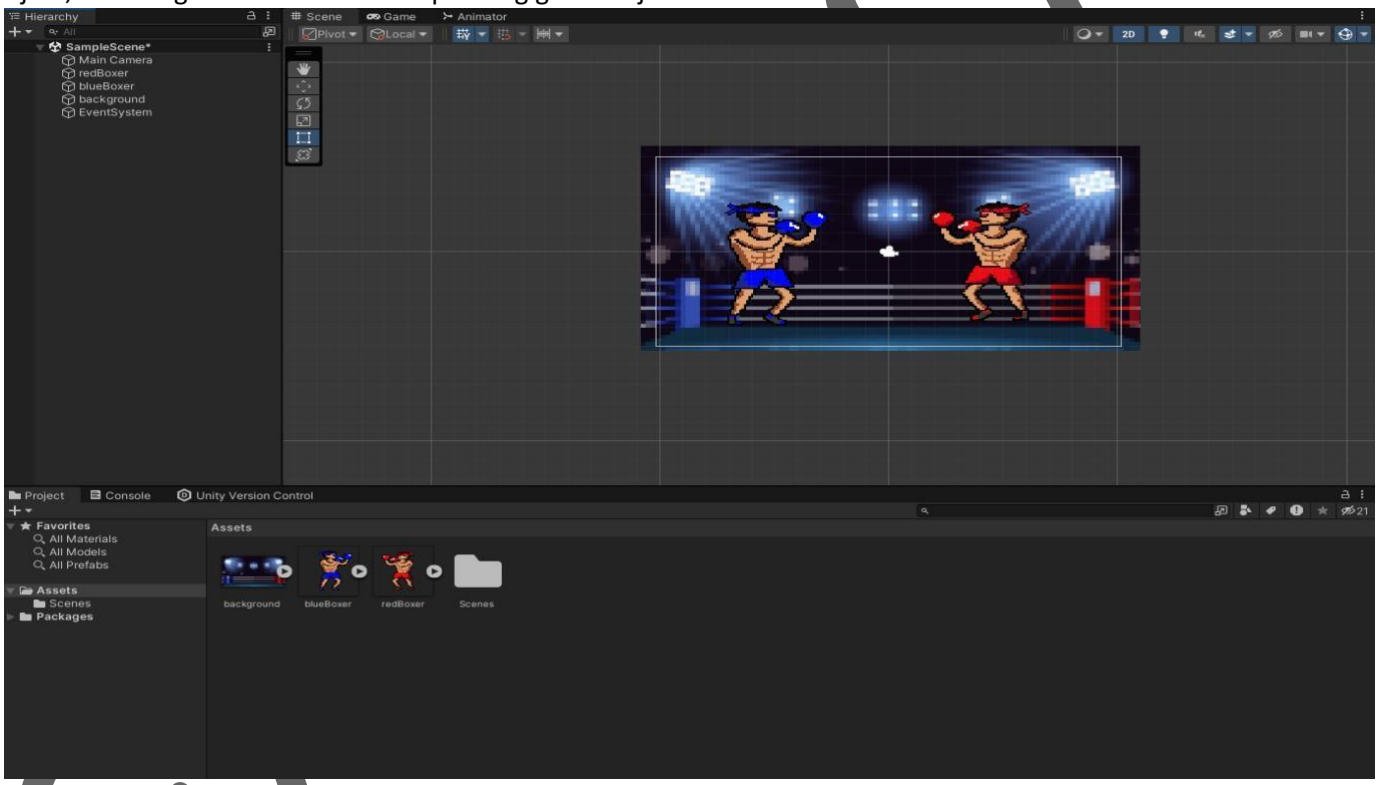
As I am working with unity, there are a lot of things to first set up in the game engine. First, I created a new 2d project which will set up the editor so that it is ready to be used for a 2d world game.

To add things to the game, I created empty game objects, in which you can add certain components to. For example I created a game object for each of the boxers, and the scripts to control their behaviour will be added when they are made. You can also add things like sprite renderers which will allow me to import an image from an assets folder, which is set up with the project, to be assigned as the sprite. This is one of the reasons why I am using a game engine, as it allows for ease while developing smaller things such as adding a sprite to the object.



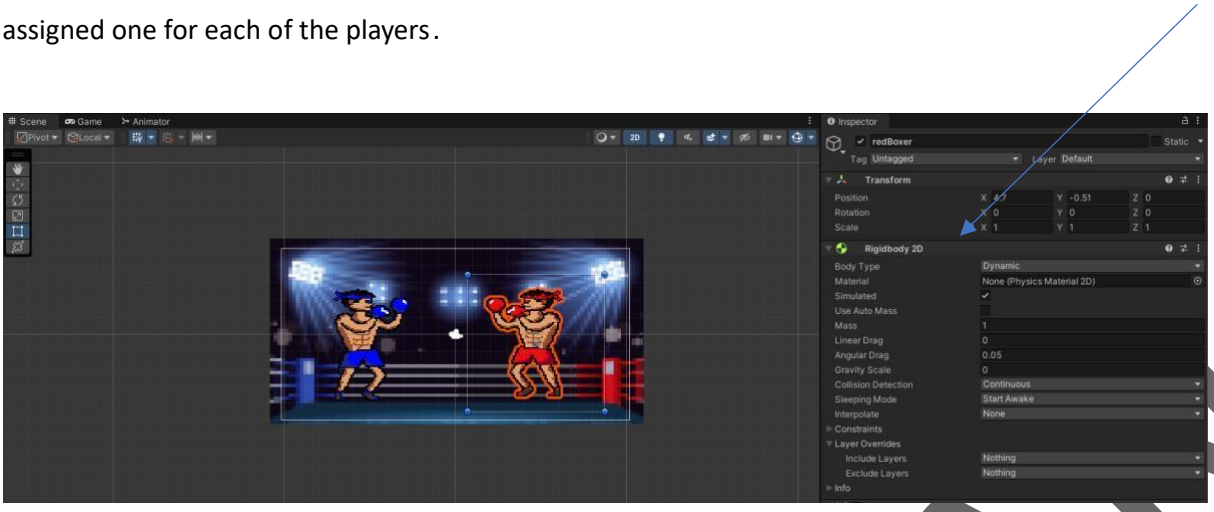


As I have already designed the sprites, and have found a suitable background for my game, I will import them into the project, and assign them to the corresponding game objects.



For an object to be able to detect collisions, and move around like a real object, it must be assigned a rigid body. So I

have assigned one for each of the players.



Now I have set up the unity project, I can start giving the game its functionality.

Player Movement:

This section will contain development for all of the left and right movement

Left and right movement:

System.collections and system.collections.generic are general namespaces for c# scripts in unity. They import classes for working with certain data structures such as lists, arrays and dictionaries etc. This will be at the start of all of my scripts

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

Using imports the unity namespace which gives core unity engine functionality for the game objects and other components. This will also be in all of the scripts as it is fundamental for Unity game development

Class created which contains all of the code for the blue boxer to move by itself. It is helpful to make it into a class so that its methods and attributes can be easily accessed from other scripts

MonoBehaviour is a namespace which contains general methods to be used in the development that are specialised to Unity

```
public class blueBoxerMovement : MonoBehaviour
{
    3 references
    [SerializeField] private Rigidbody2D rigidBody;
    7 references
    private float horizontalBlue;
    1 reference
    [SerializeField] public float speed = 5f;
```

SerializeField will allow me to change the contents of the variable straight from the unity engine. In this case it is used so that I can connect the blue boxers rigid body component to the script, so that its contents can be altered to allow the boxer to move

Horizontal Blue has been declared as a float for use in the script. It will be used as a direction vector , with -1 meaning the left, and +1 meaning right.

Speed is set a value of 5, however since it is Serialized, I will be able to change the value that it holds directly from the game engine. It is useful in this situation as I can test the speeds while the game is running to see what speed is best.

In the Start() function (which executes code before the game begins), The rigid body of the blue boxer game object will be fetched and assigned the rigidbody variable which was declared before. This allows me to access/alter the components of the rigid body such as velocity.

```
0 references
void Start()
{
    rigidBody = GetComponent<Rigidbody2D>();

}

0 references
void Update()
{

    if (Input.GetKey(KeyCode.A)){
        horizontalBlue = -1.0f;
    }
    else if (Input.GetKey(KeyCode.D)){
        horizontalBlue = 1.0f;
    }
    else{
        horizontalBlue = 0f;
    }

}

0 references
private void FixedUpdate()
{

    rigidBody.velocity = new Vector2(horizontalBlue * speed, rigidBody.velocity.y);
}
```

In the Update() function which executes every frame, input is being checked to see if the player is trying to move the character.

A – the user is trying to move the blue player to the left, which sets the direction vector to -1.

D – The user is moving the blue boxer to the right, setting the direction vector to +1

If they are not giving any relevant input, then the players direction vector will be 0, meaning they are idle.

In the FixedUpdate() function, which executes the code in fixed time intervals which are often synchronised with the physics system in Unity, the velocity of the game object is being altered. This is happening because the horizontalBlue variable changes constantly with input from the user.

The velocity component of the blue boxers rigid body is assigned a new vector variable which contains the x (horizontalBlue * speed) component and the y (rigidBody.velocity.y) component. In this case, the y component does not change, as we only care about the left and right movement. The Direction vector, horizontalBlue is multiplied by the magnitude scalar, speed which creates the new x component of the velocity, effectively moving the blue boxer left and right on the screen.

<u>Red player movement script:</u>

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

0 references
public class redBoxerMovement : MonoBehaviour
{
    3 references
    [SerializeField] private Rigidbody2D rigidBody;
    4 references
    private float horizontalRed;
    1 reference
    [SerializeField] public float speed = 5f;



    0 references
    void Start()
    {
        rigidBody = GetComponent<Rigidbody2D>();

    }

    0 references
    void Update()
    {

        if (Input.GetKey(KeyCode.J)){
            horizontalRed = -1.0f;
        }
        else if (Input.GetKey(KeyCode.L)){
            horizontalRed = 1.0f;
        }
        else{
            horizontalRed = 0.0f;
        }

    }

    0 references
    private void FixedUpdate()
    {

        rigidBody.velocity = new Vector2(horizontalRed * speed, rigidBody.velocity.y);
    }
}
```

Explanation of code

This code consistently checks for the relevant input from the user (A and D for the blue player, J and L for the red player) and changes the horizontal direction vector accordingly. The velocity is also always being calculated, however will only produce a change when the input is collected for the horizontal vector. This is a simple but effective way for creating the left and right movement because the velocity considers magnitude and direction, so the player will always move in the intended direction.

As stated in the design section, this is an essential standalone feature which is not dependent on any other features working. This means that now that I have coded and tested it, I will not ever have to revisit, until maybe the creation of the AI where its behaviours will change. This is the reason why I made feature first.

Players moving left and right to different parts of the screen

Key Variables

VARIABLES	TYPE	ROLE
RIGIDBODY	RigidBody2d	Allows the properties of the game object to be altered in the code, such as the velocity of the object.
HORIZONTAL BLUE	float	Provides a direction vector value which indicates the direction of input
HORIZONTAL RED	float	Provides a direction vector value which indicates the direction of input
SPEED	float	Default speed value

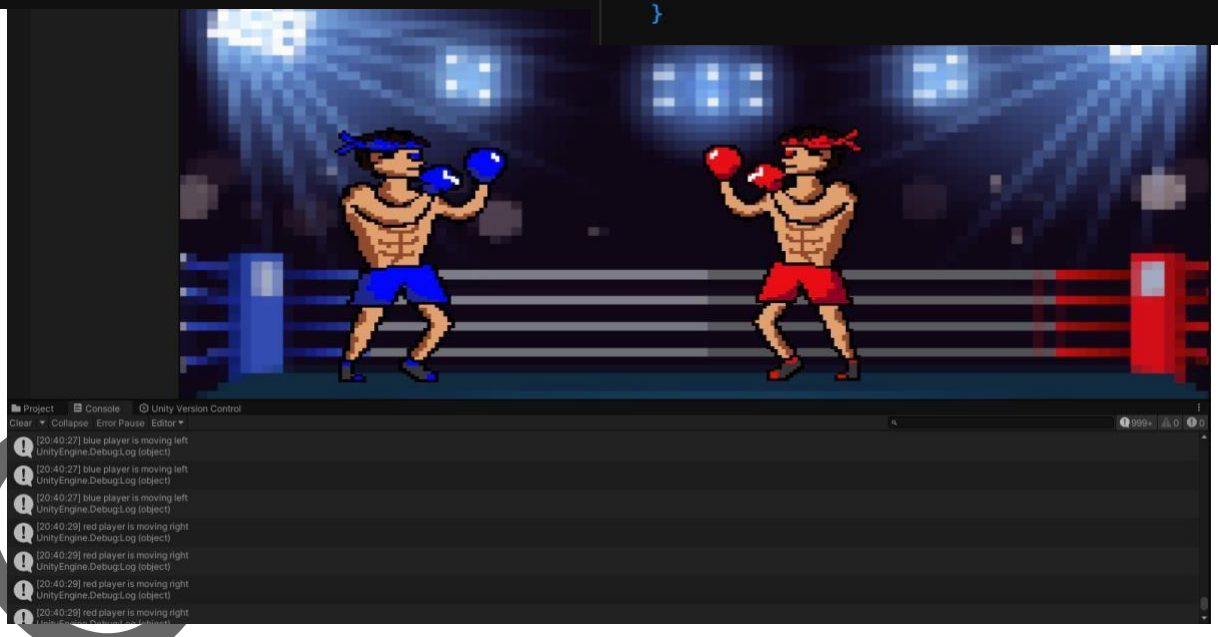
Testing

SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL
MECHANICS	Left and right movement	Players should move in direction dependent on their input	Each player independently moves in the direction intended	PASS

For testing purposes, and proving the validation of this action, I have added simple debug statements which will print the player that is moving and the direction they are moving in and see if this matches what the player is doing.

```
void Update()
{

    if (Input.GetKey(KeyCode.A)){
        horizontalBlue = -1.0f;
        Debug.Log("blue player is moving left.");
    }
    else if (Input.GetKey(KeyCode.D)){
        horizontalBlue = 1.0f;
        Debug.Log("blue player is moving right.");
    }
    else{
        horizontalBlue = 0f;
    }
```

```
void Update()
{

    if (Input.GetKey(KeyCode.J)){
        horizontalRed = -1.0f;
        Debug.Log("red player is moving left.");
    }
    else if (Input.GetKey(KeyCode.L)){
        horizontalRed = 1.0f;
        Debug.Log("red player is moving right.");
    }
    else{
        horizontalRed = 0.0f;
    }
```



As we can see, the program is recognising when each of the players is moving in any direction (Blue player is moving left, Red player is moving right), and because it is a continuous function where the key can be held down, the statement was logged multiple times

Review

Currently the horizontal movement of the players works perfectly by itself. For the next prototype, I need to add boundaries so that the boxers cannot move beyond the scope of the screen, as this is a current problem.

Player Borders:

This code has been added to the player movement script as it requires some of the variables from there.

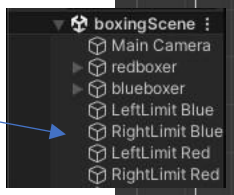
```
public class blueBoxerMovement : MonoBehaviour
{
    3 references
    [SerializeField] private Rigidbody2D rigidBody;
    7 references
    private float horizontalBlue;
    1 reference
    [SerializeField] public float speed = 5f;

    1 reference
    public GameObject rightLimitGameObject;
    1 reference
    public GameObject leftLimitGameObject;

    2 references
    private Vector3 rightLimit;
    2 references
    private Vector3 leftLimit;
```

These variables have been declared and will be assigned a game object within the Unity game engine. The game objects will be physical borders which will not allow the user to go past them.

To stop players being able to move past those game objects, the most efficient way to do this is to get the position of the game objects, and to not allow the player to move past them when they are at those coordinates.

These two variables will hold the exact positions of the game objects which are stored in the above variables.

The game objects have been created in Unity, called the limits.



The little blue dots, connected by a solid line show the region covered by the blue player. These are the bounds of where the blue player will be allowed to move.

The right limit is a lot further from the side than the left limit, just so that if both players are in the red corner, they aren't appearing as if they are overlapping each other, but the blue player will still be able to do damage

```
void Start()
{
    rigidBody = GetComponent<Rigidbody2D>();

    rightLimit = rightLimitGameObject.transform.position;
    leftLimit = leftLimitGameObject.transform.position;

}
```

In this code, I am accessing the position of the game objects, and assigning them to the variable declared before. Because the "rightLimit" is declared as a Vector3, it is assigned three dimensions, so it holds the x, y and z coordinates of the right limit game object. This is the same for the leftLimit.

```
private void FixedUpdate()
{
    if((transform.position.x <= leftLimit.x) || (transform.position.x >= rightLimit.x))
    {
        horizontalBlue = 0;
    }

    rigidBody.velocity = new Vector2(horizontalBlue * speed, rigidBody.velocity.y);
}
```

This if statement states that if the position of the blue players x coordinate(transform.position.x) is less than the x coordinate of the left limit (letLimit.x) OR if their x coordinate is greater the right limit, horizontalBlue will be assigned 0.

This means that if the player is in contact with one of the borders, their velocity will update to be 0, meaning they will not be able to move past the limit.

This code was used to create "regions" within the camera space, that were edited and scaled to correct size within the unity engine (shown as the bars underneath the characters in the screenshot). The top image shows the boundaries for the blue character.

When the player reaches the boundary, their velocity is set to 0 so they cannot move past it any further. This is required because in the Unity engine, the project can take up space bigger than just the camera view.

Key Variables

VARIABLES	TYPE	ROLE
RIGHTLIMITGAMEOBJECT	GameObject	This assigns the region that we sized in unity engine to the variable. This means I can use different properties of the limit and implement it in code.
LEFTLIMITGAMEOBJECT	GameObject	This assigns the region that we sized in unity engine to the variable. This means I can use different properties of the limit and implement it in code.
RIGHTLIMIT	The x position of the right limit	The code uses the x coordinates of the right limit so when the x position of the player reaches it, they stop
LEFTLIMIT	X position of the left limit	The code uses the x coordinates of the left limit so when the x position of the player reaches it, they stop

Testing

SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL	ACTION TAKEN
MECHANICS	Player boundaries	Player is stopped if they are at, and trying to exit the bounds	The player is permanently stopped upon reaching the boundary	FAIL	Stated below

After testing, I have found that when the player reaches the boundary, their velocity is set to 0 permanently and they are immobile for the rest of the game. This is because when their velocity is set to 0, they cannot move away from the boundary, so they are always at the boundary, and the condition is always being satisfied. This means that the velocity is always set to zero.

Solution:

The way that I fixed this was to alter the conditional statement:

```
private void FixedUpdate()
{
    if((transform.position.x <= leftLimit.x && horizontalBlue == -1.0) || (transform.position.x >= rightLimit.x && horizontalBlue == 1.0))
    {
        horizontalBlue = 0;
    }

    rigidBody.velocity = new Vector2(horizontalBlue * speed, rigidBody.velocity.y);
}
```

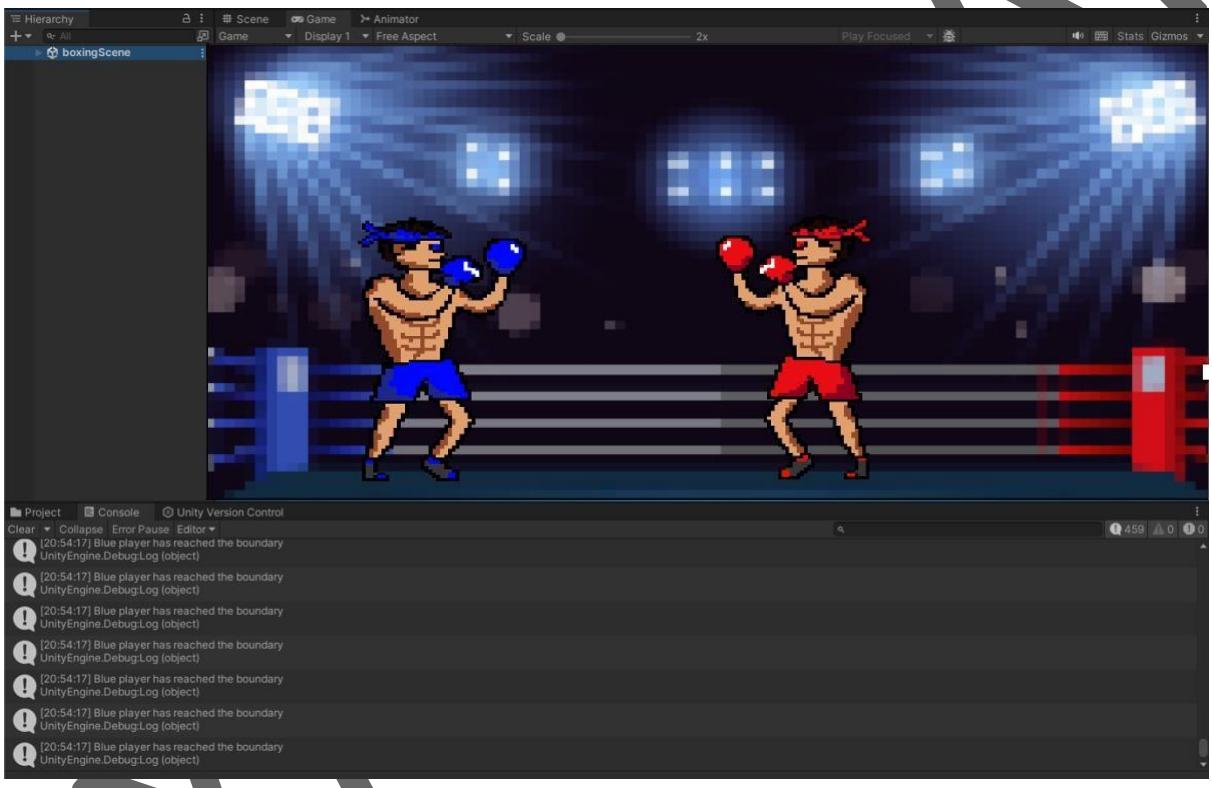
Now when the player reaches the boundary, AND they are trying to surpass the boundary, their velocity will be set to zero. If they are at the boundary, and trying to move away from it, their velocity will not be set to zero, so the condition is not satisfied (It is False) so they are allowed to move away from it. This satisfies the expected result.

SECOND TEST:

SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL
MECHANICS	Player boundaries	Player is stopped if they are at the boundaries and trying to exit	Player is stopped if they are at the boundaries and trying to exit	PASS

To help prove validation of this action, I have added a debug statement in the code to see when the program realises that the player has reached the boundary.

```
private void FixedUpdate()
{
    if((transform.position.x <= leftLimit.x && horizontalBlue == -1.0) || (transform.position.x >= rightLimit.x && horizontalBlue == 1.0))
    {
        horizontalBlue = 0;
        Debug.Log("Blue player has reached the boundary");
    }
```



Review

The players now have free movement left and right and cannot move past the limits that are defined in the code. Success criteria reference GM3 has now been met.

Fighting Mechanic Animations

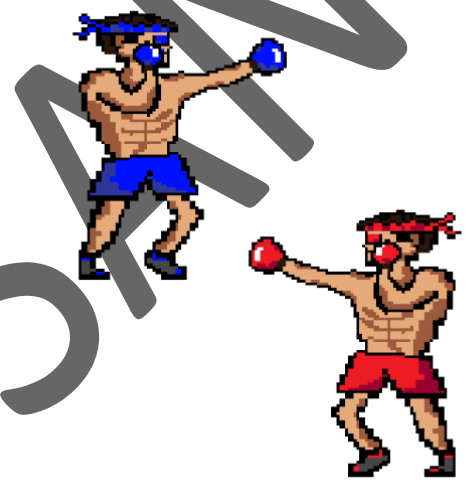
To begin thinking about the fighting mechanics, they all require animations to be made. Since I have created the sprites for all the actions (punching, blocking and crouching), I can now make animations in Unity. To make an animation in

Unity you must create an animator, which is a component that can be added to a game object, to control and manage the animations made for that object.

I will be creating all the animations, then testing and reviewing them all together as they all require the same work with slight differences.

For a lot of these animations, they require the use of hitboxes. Hitboxes (aka box/circle colliders) are components that can be added to a game object which can either be used as triggers, or as physical barriers. Each player will have three main hitboxes, one for detecting a punch landing on them (detection hitbox), one for preventing the players from moving through each other (barrier hitbox).

Punching:

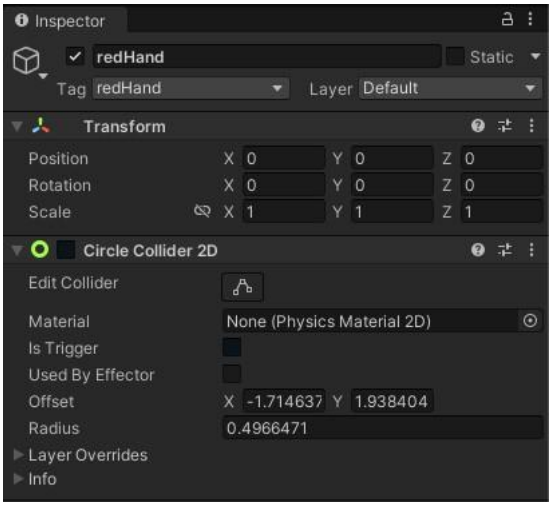


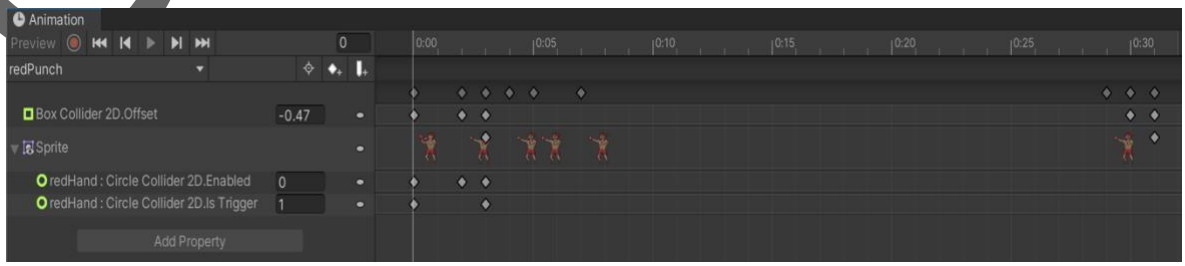
The punching sprites has been made for both players, as shown to the side. The key bind for punching will be “F” for the blue player, and “H” red player.

For the punching animation specifically, a circle hitbox needs to be added to the boxing glove, so that when it collides with the other players detection hitbox, the damage can be done. For this I have created child game objects for each of the boxers which holds the hand of their player.

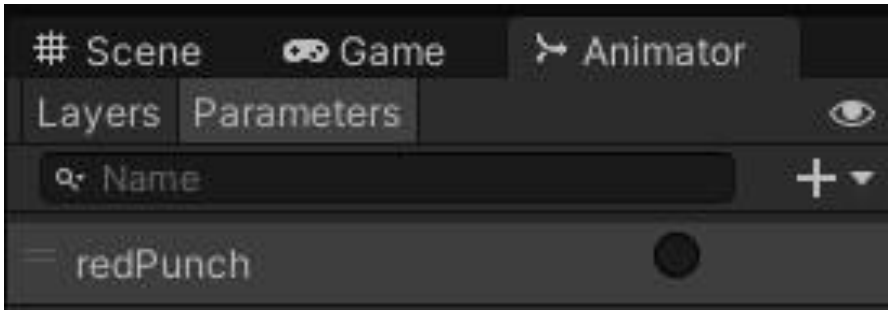
Each of the hands hold a component called a circle collider 2D which will be used as the hitbox for the hand.

I have also given each of the hands, a tag which holds the name of the game object. This will be useful later when creating the health aspect of the punch, as it will allow the game to detect which hitbox is colliding with which, so that health can be decreased only when two specific hitboxes collide.





As we can see above, the sprite changes from the normal stance to the punching stance which is held for a while so that the animation is not too quick that it can be seen. There are also lots of key frames throughout where the values of the components on the left change. In the keyframe where the sprite is changed, the position of the box collider is moved, and the circle collider is enabled for a single frame, and after this it is disabled, as well as it not being a trigger anymore. This will prevent a single punch from doing the damage of multiple punches.



In the red boxer animator window, I have created a parameter called redPunch, which is a trigger. This means that when the trigger is activated, the punch animation will be carried out.

With this trigger, I can now create a script which will control the players animations and I will be able to set the trigger to "on" when a certain condition is true.

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

0 references
public class RedPlayerController : MonoBehaviour
{
    7 references
    [SerializeField]public Animator animator;
```

As this is a new script, I need to import all of the necessary namespace's again, as well as creating a class for the animations.

I have declared an animator, which is a component I will be able to access in the script. It is serialised so that I can add the animator directly into the script from the unity engine.

```
void Update()
{
    //for punching
    if (Input.GetKeyDown(KeyCode.H))
    {
        animator.SetTrigger("redPunch");
    }
```

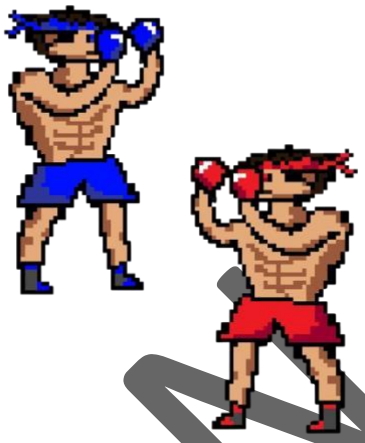
In the update function, I have created an if statement that checks if the "H" key is pressed, and if it is, it will set the trigger to on, meaning the animation will play.

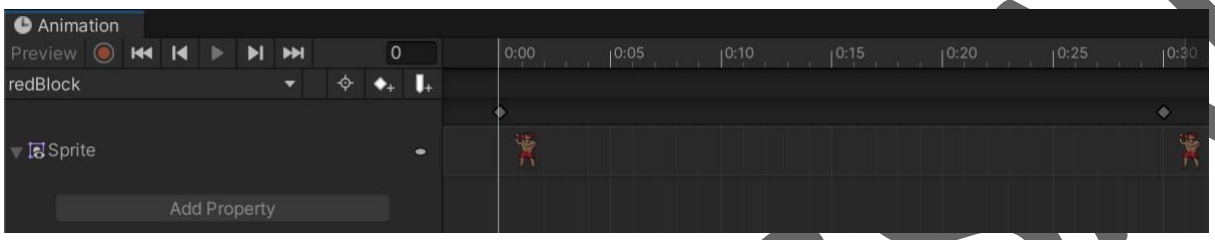
Blocking:

The blocking sprites are displayed to the right. The key that will be used for blocking for the blue player will be "W" and for the red player it will be "I".

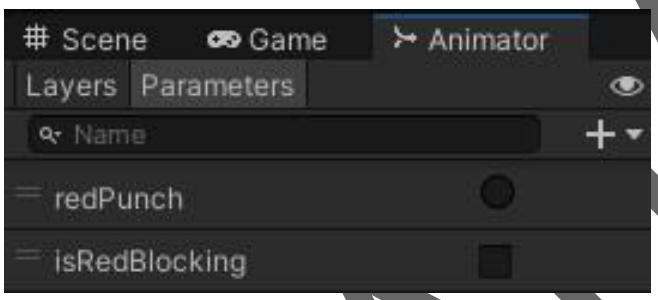
This feature will be different to punching as it will not require any change of hitbox of any sort, it will simply just be a change of sprite, and all of the behaviours of this feature will be controlled later, when the health and stamina scripts are made.

However this feature is not an animation that will be triggered then turned off, I want it to be one where the player can hold it down for as long as they want. The stamina feature will then limit how long they can hold it for.





This shows the animation is just a change in sprite over a certain period. In the animator for that object, I have created another parameter which is a Boolean called “isRedBlocking”.



This bool has been used so that when it is true, the animation is played. I can now change when the bool is true in the script.

```
// for blocking
if (Input.GetKeyDown(KeyCode.I) || Input.GetKey(KeyCode.I))
{
    animator.SetBool("isRedBlocking", true);
}
if (Input.GetKeyUp(KeyCode.I))
{
    animator.SetBool("isRedBlocking", false);
}
```

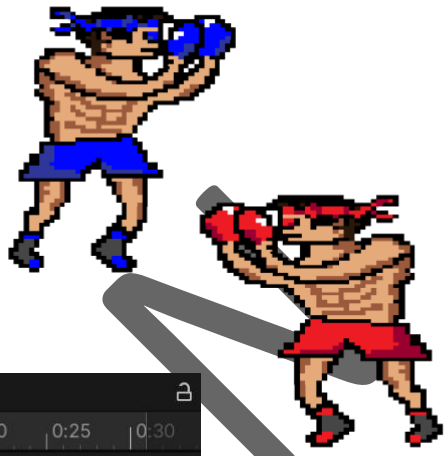
If the “I” key is pressed OR held, then set isRedBlocking to true and play the animation

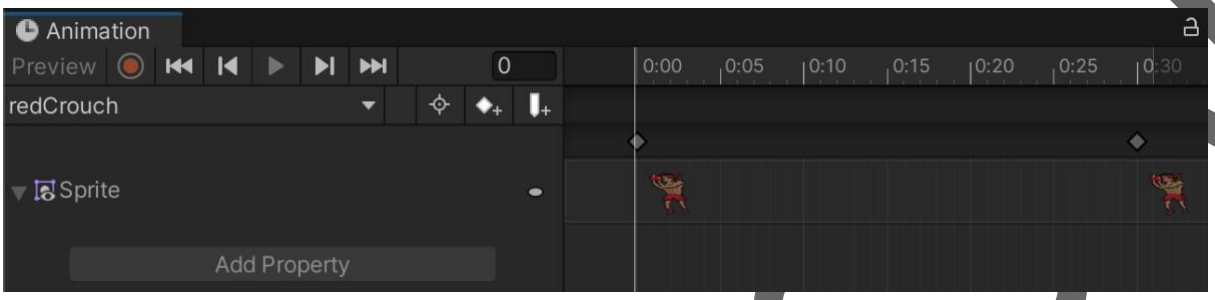
When the “I” key is released, set isRedBlocking to false, therefore stop the animation from playing

Crouching:

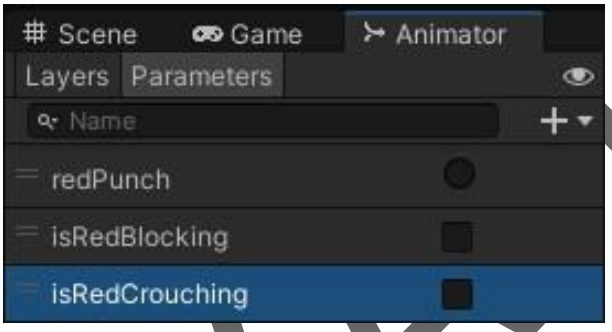
This feature is very similar to the blocking feature in terms of the animation, the differences will come later when the health and stamina scripts are made. The differences being that crouching will nullify all damage taken but will use up more stamina per second.

The key bind for crouching will be “S” for the blue player, and “K” for the red player.





This animation just has a change of sprite and will be controlled by the isRedCrouching Boolean parameter.



When the isRedCrouching parameter is set to true, the animation will play. Now I can set this up in the script in the same way as the blocking feature.

```
// for crouching
if (Input.GetKeyDown(KeyCode.K) || Input.GetKey(KeyCode.K))
{
    animator.SetBool("isRedCrouching", true);
}
if (Input.GetKeyUp(KeyCode.K))
{
    animator.SetBool("isRedCrouching", false);
}
```

If the “K” key is pressed OR held, then set isRedCrouching to true and play the animation

When the “K” key is released, set isRedBlocking to false, therefore stop the animation from playing

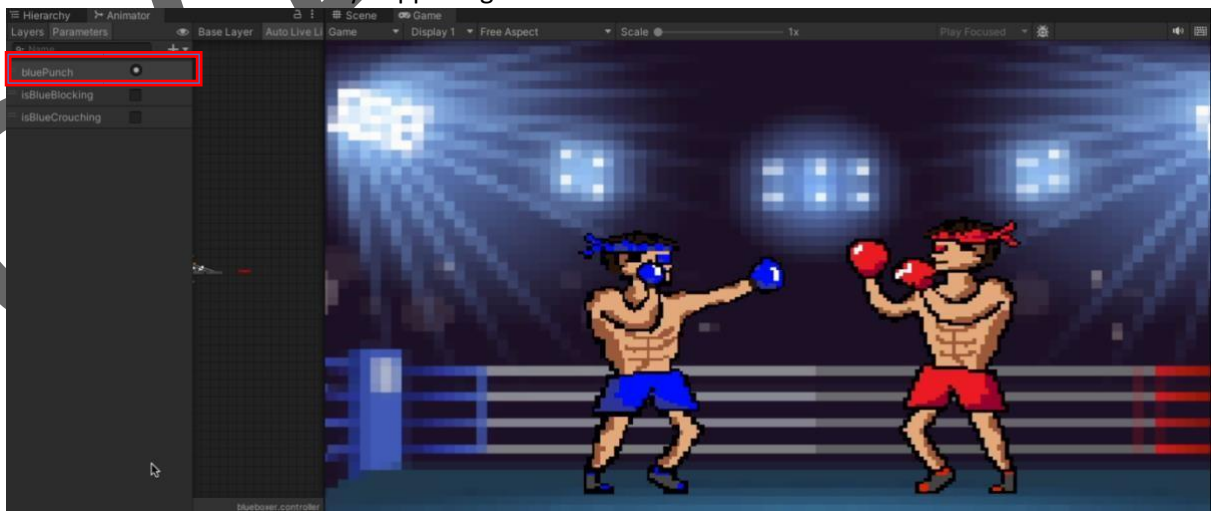
Key Variables

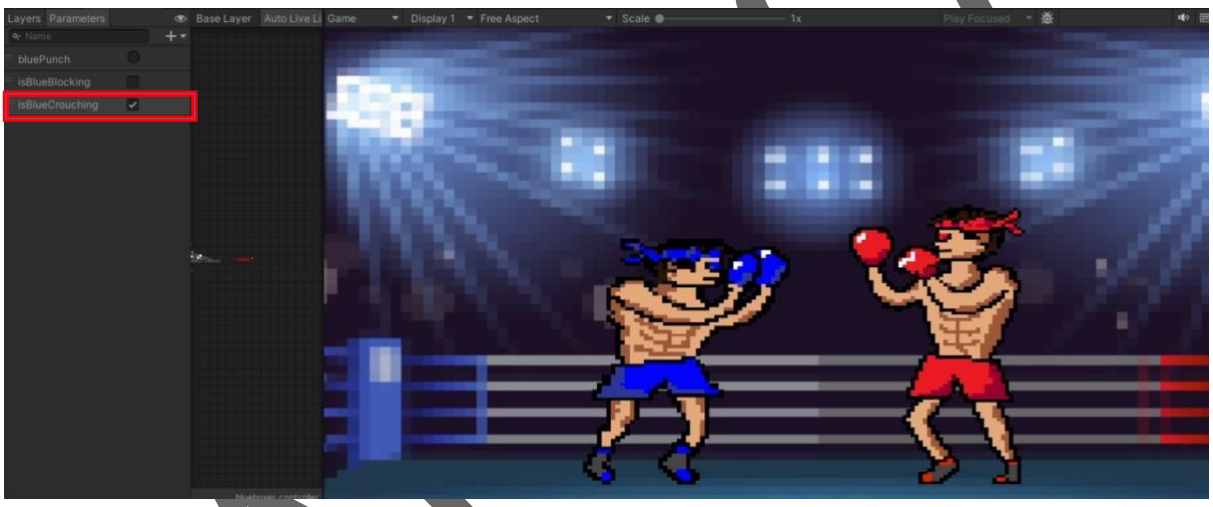
VARIABLES	TYPE	ROLE
ANIMATOR	Animator component	This holds the animator for that object, so I can access and control certain attributes of the animations (such as the parameters)
REDPUNCH	Trigger	This trigger tells is used to tell the program when to play the punch animations. When the trigger is on, the animation is played.
BLUEPUNCH		
IS RED BLOCKING	Boolean	This tells the program whether the be playing the blocking animation or not. When true, the animation will play When false, the animation will stop
IS BLUE BLOCKING		
IS RED CROUCHING	Boolean	This tells the program whether the be playing the crouching animation or not. When true, the crouching animation will play When false, the crouching animation will stop
IS RED CROUCHING		

Testing

SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL
MECHANICS	Animations	When the relevant key is pressed/held, the correct animation should show.	When the relevant key is pressed/held, the correct animation is shown.	PASS

To prove validation of this, I have look at when the parameters are activated in the unity engine and seen if they match up with the animation that is currently happening.





These screenshots show that when I have pressed the correct key, the correct parameter is in the state it should be in meaning that the program understands when what animations it should be doing with certain input.

Review

The animations are a very important part of the game as they are a great visual representation of what action you are doing and allows both players to understand what is happening. Now that they are added and are working as they should, I can add the functionality to all of the actions by adding the health handling script. This will deal with all the actions that cause damage, and those that dodge damage.

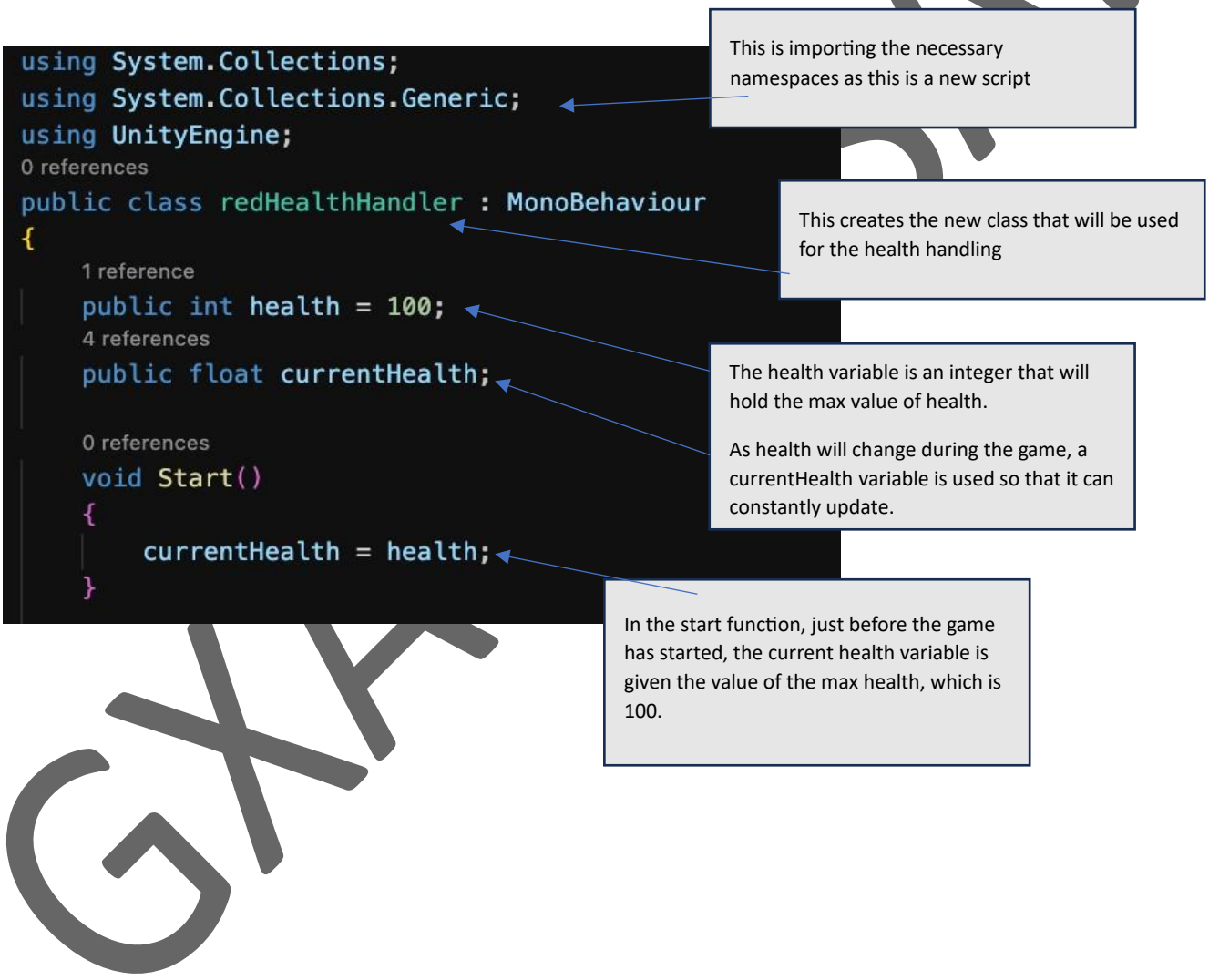
Health Control

The health system will take care of all of the collisions and will detect whether a hit will cause damage or not. Punching, blocking, and crouching will all have separate behaviours when it comes to health which will be explained.

Punching:

When the hand of one player collides with the other players hitbox, they will do 5 damage. The health will not be regenerated, so that the game can be quick yet still fun.

Before in the animation controller, I had set up the punch animation so that the circle collider (collision detector) of the hand is enabled during as soon as the sprite changes. This means that when the punching animation is played, any collisions that are made with the hand can be detected, and therefore health can be changed accordingly.



A KO (knock-out) function is made for when a player is defeated. At this moment it is mainly used for testing cases, so that the player game object is destroyed when health is at 0, but in the future, it will be given added functionality to add a game over screen etc.

```
public void KO()
{
    Destroy(gameObject);
    Debug.Log(" RED KO ");
}
1 reference
public void TakeDamage(int damage)
{
    currentHealth -= damage;
    Debug.Log("Red Health: " + currentHealth);
}
```

In the KO function, the game object which the script is joined to, will be destroyed and disappears on the screen.

A debug statement is also added so that we can monitor the state of the player at all times.

Taking damage is made into a function as it will be used a few times, for punching and for blocking. It will take in an integer representing the value which is what will be taken away from the current health

This will take the damage away from the players health, and will log what the health is as it is taken away

This is a function within the UnityEngine namespace, which allows collisions to be used as a trigger to events. This function defines another collider component, and I have named it other (for other player).

```
void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("blueHand"))
    {
        TakeDamage(5);
    }
    if (currentHealth <= 0)
    {
        KO();
    }
}
```

If the other collider's tag has the name "blueHand" (meaning the blue player is punching the red player), then call the TakeDamage function, with a damage of 5.

This is the reason why I mentioned in the punch development, that I will be giving each hand a tag, so that it can be identified when detecting collisions

If the health gets to 0, then the KO function is called and the game is over.

Explanation

In this prototype, I added functionality for a punch which does 5 damage to the opponent character and when their health reaches 0, they disappear. The screenshot shows the health decreasing of the blue boxer, and when they reach 0 health, the game object disappears.

These scripts have been added to the relevant players. Now when the player uses the punch key (“F” or ”H”), the hand of that player becomes a hitbox, and detects collisions with the other players body, and if it collides properly, health will be decreased by 5 for the opponent.

Key Variables

VARIABLES	TYPE	ROLE
HEALTH	int	Has the value 100, which is the initial value for health and is set to the currentHealth on start up.
CURRENTHEALTH	float	This is the temporary health variable that is repeatedly changing during the duration of the gameplay.
DAMAGE	float	This is a parameter for a the “TakeDamage()” function and the value inside the function will be subtracted from the current health.
OTHER	Collider 2D component	This represents the other players hitboxes, and is used for detecting collisions between the two players. In my case, we are checking for the “other” players tag, and if it is tagged as the other players hand, then damage will be taken.

Testing

SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL
HEALTH CONTROL	Health (punch)	Health of damaged player decrements by 5	Health of damaged player decrements by 5	PASS

EVIDENCE

The validation for this is that the currentHealth variable decreases as it is meant to. To show evidence of this, I added a debug statement in the script to show the value of the current health variable as it changes.

To the side is a screenshot of the red boxer getting hit, and their health decreasing at the same time (as well as past debug statements showing exchange of punches).

Review

The punching feature works perfectly now, it takes damage when it is meant to and it recognises when one of the player has lost all of their health.

Now I need to work on the health handling of the other actions which will nullify the damage taken. All of that code will probably build upon the code I have annotated in this section.

Blocking / Crouching:

The blocking feature will decrease the damage being taken. Punches will do 3 damage to someone who is blocking now. The reason why it is a small change as it still needs to do enough damage so that it will make a difference, but this will be balanced by the fact that it will take up a lot less energy.

Crouching will be different as it will completely nullify the punch, no damage will be done to the opponent player. However, this will take up a lot of energy compared to blocking.

```
public class BlueHealthHandler : MonoBehaviour
{
    1 reference
    public int health = 100;
    5 references
    private float currentHealth;

    4 references
    [SerializeField] public Animator animator;
```

Instead of altering the hitboxes with the blocking and crouching animations, I will import the animator so the blocking/crouching parameters can be checked. This will allow me to check whether the player is blocking or crouching at that time.

It is serialized so that I can assign the animator in the Unity game engine

```
void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("redHand") && (animator.GetBool("isBlueBlocking") == false && animator.GetBool("isBlueCrouching") == false))
    {
        TakeDamage(5);
    }
    else if (other.CompareTag("redHand") && animator.GetBool("isBlueBlocking") == true)
    {
        TakeDamage(3);
        Debug.Log("Blue BLOCKED red punch");
    }
    else if (other.CompareTag("redHand") && animator.GetBool("isBlueCrouching") == true)
    {
        Debug.Log("Blue CROUCHED under red punch");
    }
    if (currentHealth <= 0)
    {
        KO();
    }
}
```

If the opponents hand collides with the players hitbox, and they are NOT blocking AND NOT crouching, then it counts as a normal punch, and 5 damage is be dealt.

If the opponents hand collides with the players hitbox, and they ARE blocking, then take 3 damage instead of 5, and for testing reasons I have added a debug statement so that I know exactly when this condition is met.

If the opponents hand collides with the players hitbox, and they ARE crouching, then do not deal any damage, and a debug statement is also added for this feature too.

Explanation

This code is an addition from the previous, It now states extra conditions for when damage can be taken, which accounts for the blocking and crouching actions. As stated before blocking will reduce damage taken to 3, whereas crouching will completely nullify damage taken.

This uses the animator components of the game object that the script is connected to, to find the state of the player when the punching collisions occur. When a punch successfully lands on the players hitbox, the script will check the state of the player to see if they are in the blocking or crouching animation and will take damage accordingly.

Key Variables

*Same as the health handler (punch)

Testing:

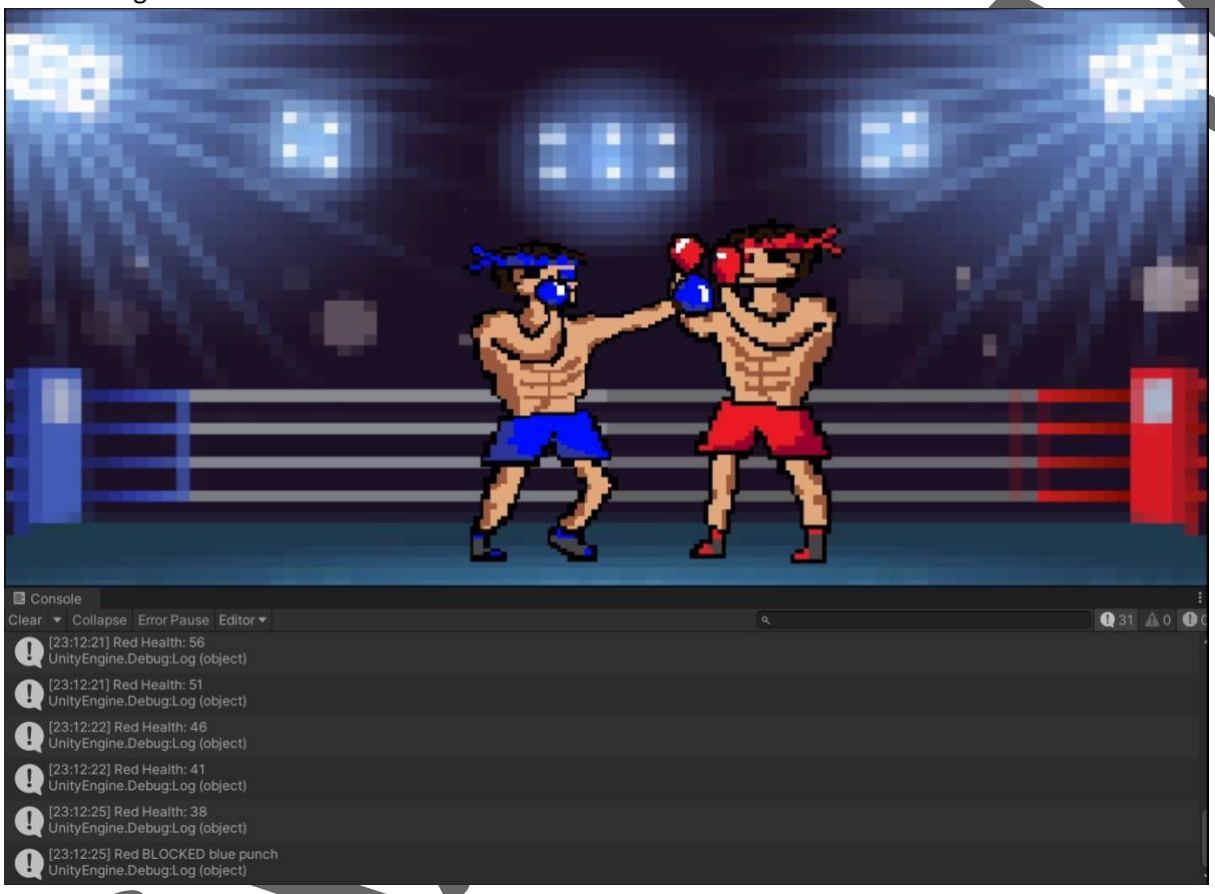
SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL
HEALTH CONTROL	Health (block)	Health decreases by 3 when punch is blocked	Health decreases by 3 when punch is blocked	PASS

	Health (crouch) when	Health remains unchanged when player crouches crouches under punch punch	Health remains PASS under player	unchanged

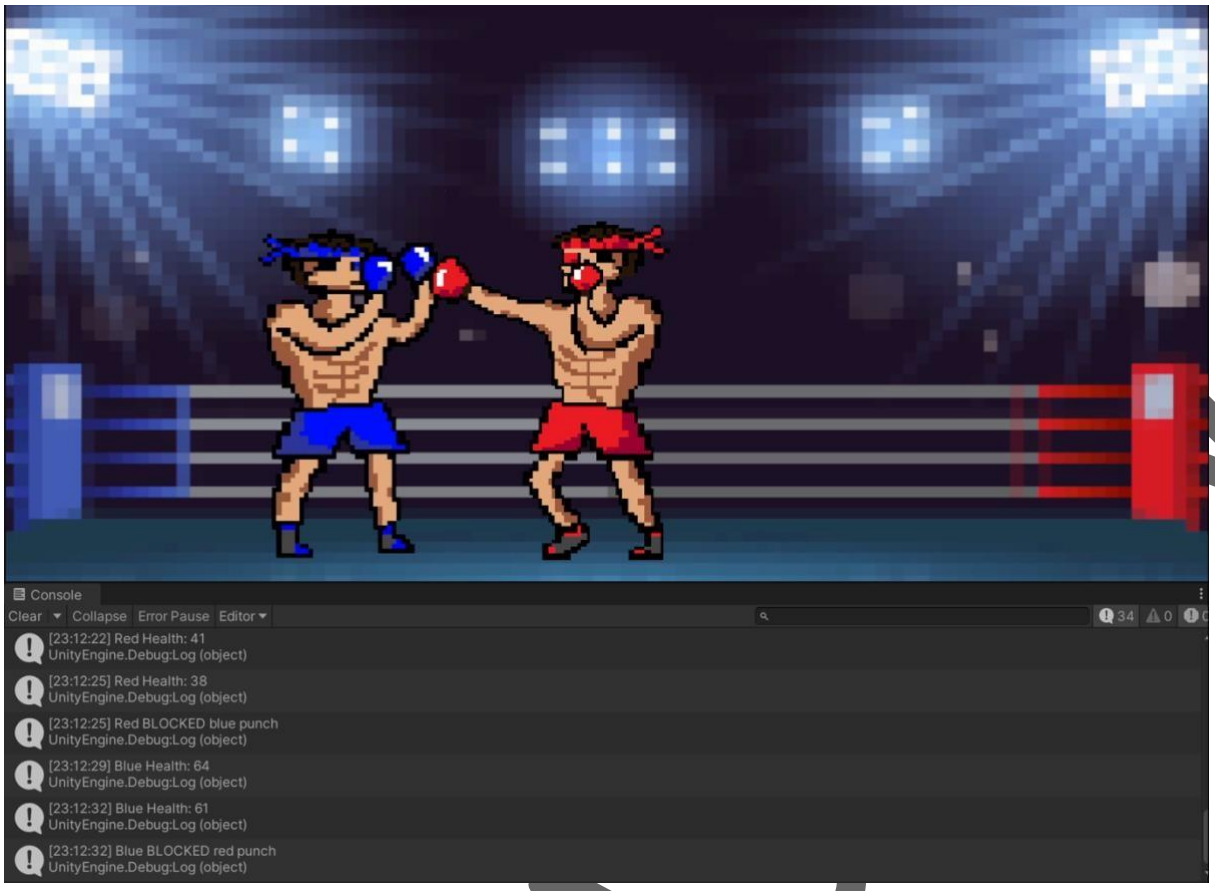
EVIDENCE

The validation of this system will be making sure that the health variable is the amount it should be when reacting to changes.

For blocking:

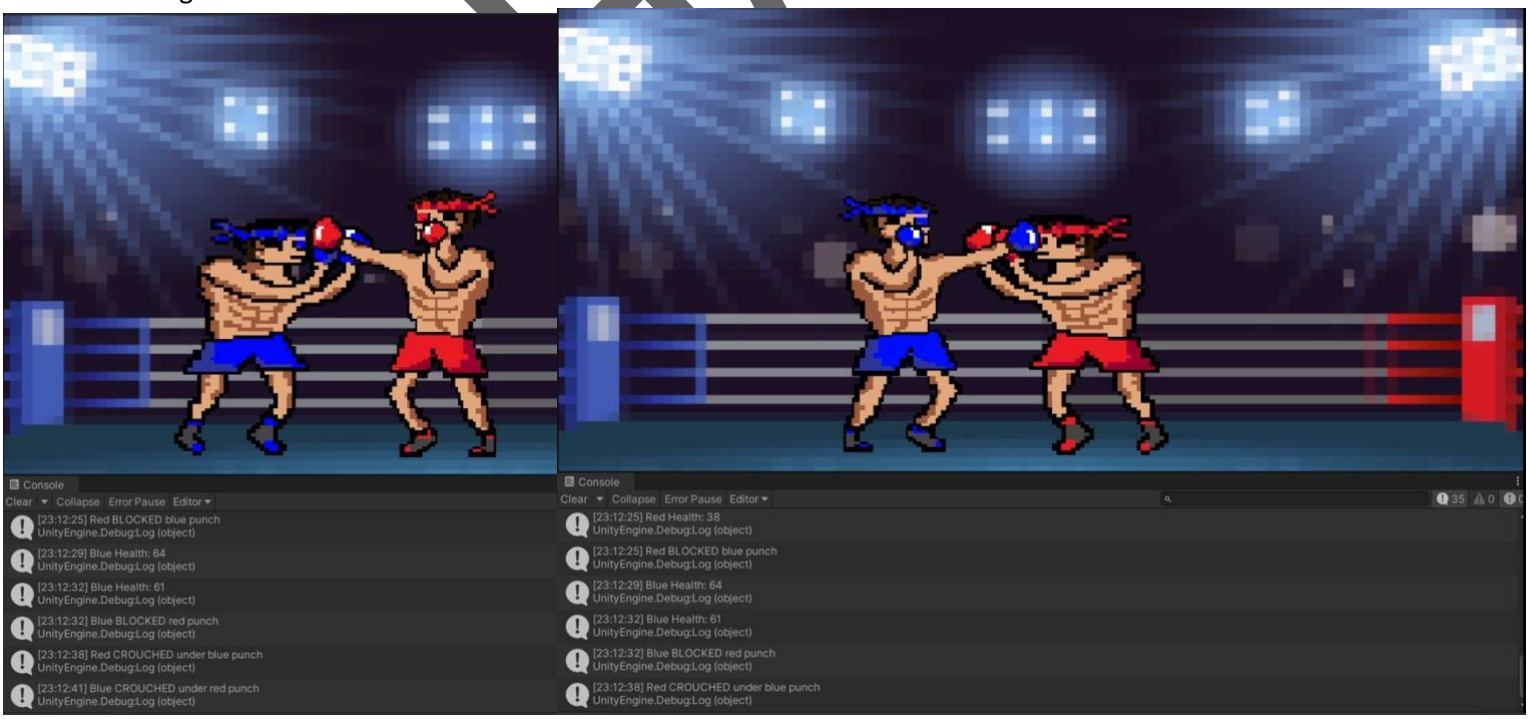


As we can see, the health was initially at 41 before the punch was thrown. The red player used the block action while the blue player punched them, and the health went from 41 -> 38 (-3 as it should do), and it was debugged that the punch was BLOCKED, meaning there is no confusion between the crouch and block action



As we can see here, the blue player's health goes from 64 -> 61 (-3) and it is clearly debugged that a block was detected and nulled the damage of the red players punch.

For crouching:



As seen in both of the screenshots, the program has recognised that the opposite player is crouching, so health is not changed at all, and it is debugged clearly that the player has crouched under the opposite players punch. This means that the correct collision is detected and that this action is working how it should be.

Review

Now I have a fully functional health system, which decides how much health will be taken from the player depending on collisions between hitboxes, and what actions they are currently doing. The only thing left to do is to create a health bar, which is the visual representation of the health variable on the screen for the players to see.

Next I will create the energy/stamina system so that players have to measure the costs of each action they do and how effective it will be.

Stamina Control

Stamina control is a very important part of the game, as it will make sure that players are not constantly abusing the power of an action. For example, it prevents someone from crouching the whole game, or just pressing the punch key as fast as they can. It also gives an extra challenge as it makes players think about what actions they can do, and what they may try to tire out their opponent, so that they can go for the win. Punching, blocking and crouching all have a cost for stamina, where punching will be a certain amount for each punch, but blocking and crouching will have a rate of energy consumption. The rate for blocking will be less than crouching, as mentioned before, because it still causes the player to take damage. One reason this is very different from the health control script is that stamina regenerates, unlike health meaning I will have to make use of Unity's time scripting features.

For this feature, I was able to find a way to create a script which could be applied to both of the players, meaning there was no two different versions of the code

Regeneration of stamina

For the first part of implementing this feature, I wanted to sort a way to regenerate stamina at a certain rate. I will make it so that every few seconds (will test to find best suited value) the stamina will regenerate.

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

0 references
public class Stamina : MonoBehaviour
{
    4 references
    private int maxEnergy = 100;
    1 reference
    public float regenerationRate = 3f;
    10 references
    public int currentEnergy;
    4 references
    private float lastRegenerationTime;
```

I created a new script for the stamina so I made a new class as well. I also had to make the include the necessary namespaces.

This is a variable that holds the value of the max stamina that the player can have.

This holds the rate that the energy will regenerate by and will be used in the regeneration function.

This is a variable for the energy that will change constantly during the game.

This holds the time since the last regeneration,

```
void Start()
{
    currentEnergy = maxEnergy;
    lastRegenerationTime = Time.time;
}
```

Before the first frame is ran, the current energy will be assigned a value from the max energy = 100.

The "Time.time" syntax grabs the time since the beginning at the time of being called, since it is in the Start() function, the time assigned will be 0:00.

```
1 reference
private void RegenerateEnergy()
{
    float timeSinceLastRegen = Time.time - lastRegenerationTime;
    float energyToAdd = regenerationRate * timeSinceLastRegen;

    currentEnergy += Mathf.RoundToInt(energyToAdd);

    currentEnergy = Mathf.Clamp(currentEnergy, 0, maxEnergy);

    lastRegenerationTime = Time.time;
}
```

A new variable is created in the function called timeSinceLastRegen. This works out the time since the last regeneration, by taking away the last regeneration time from the current time.

The last regen time is then used to work out how much energy should be added to the players current energy so that by multiplying it by the regeneration rate.

The energyToAdd is then added on to the current energy, but since the calculation for energyToAdd may result in a floating-point number, I have made the value round to the closest integer.

To ensure the max energy doesn't exceed 100, the current energy variable is clamped to values from 0-100.

The lastRegenerationTime is then given a new value of time, so that the next regeneration can happen at the correct time

```
{
    //regenerate energy constantly after 1.75 seconds from last regen
    if (Time.time - lastRegenerationTime >= 1.75f)
    {
        RegenerateEnergy();
    }
}
```

Now in the update function, which is run every frame, when the time between the last regeneration is >= 1.75 seconds (I found this was a good time as it was not too quick, but was perfect), the regenerate energy function has been ran.

Explanation

The code above successfully works with the timing system in unity to regenerate the energy of the player in fixed intervals. I did this by making the program recognise when 1.75 seconds had passed from the last regeneration, and to add a certain calculated amount of energy to their current energy. This will continue to happen until energy has reached 100, as I have clamped the current energy variable between the values 0-100.

Key variables:

VARIABLES	TYPE	ROLE
MAX ENERGY	int	Has the value 100, which is the initial value for energy and is set to the currentEnergy on start up.
CURRENTENERGY	int	This is the temporary energy variable that is repeatedly changing during the duration of the gameplay.
REGENERATION RATE	float	This is the rate which regeneration of energy will take.
LAST REGEN TIME	float	This holds the time of when the last regeneration occurred.
TIME SINCE LAST REGEN	float	This is the time since the last regen occurred and will be used in the calculations for the energy to add variable.
ENERGY TO ADD	float	This takes the time since last regen, and the regeneration rate, finds the product and produces a value for the energy to be added to the current energy. This is the main feature of the regeneration function.

Testing:

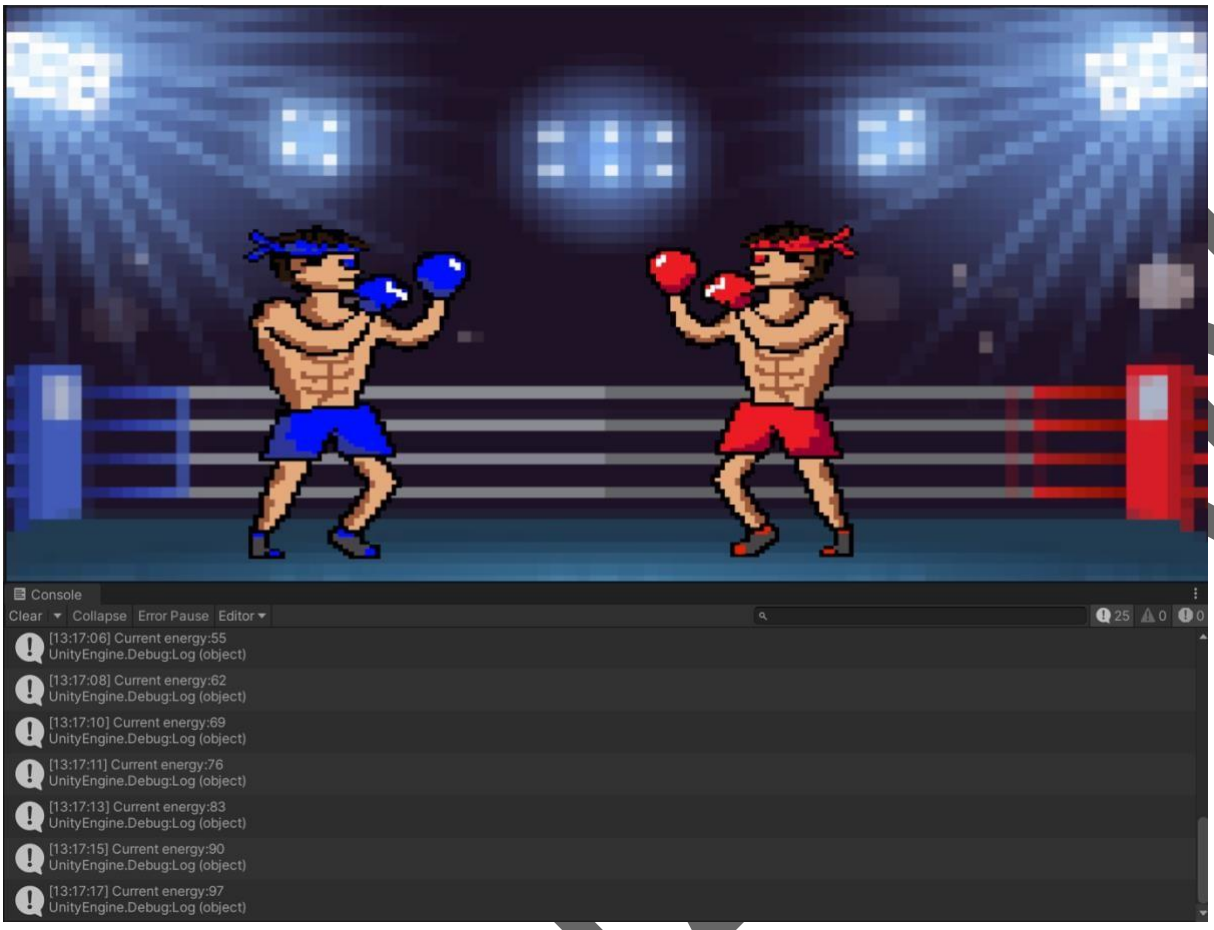
SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL
STAMINA CONTROL	Regenerate Energy	The stamina will regenerate in fixed intervals by a varying amount, depending on the length of the interval	The stamina regenerates at a stable and balanced rate as it should	PASS

EVIDENCE / Validation

To show validation of the energy regenerating at a stable rate, I've added a debug statement to the "Update()" function where the regenerate energy function is called, so that the current energy can be displayed to the user in the console, so that we can see the rate at which the energy goes up while regenerating.

```
//regenerate energy constantly after 1.75 seconds from last regen
if (Time.time - lastRegenerationTime >= 1.75f)
{
    RegenerateEnergy();
    Debug.Log("Current energy:" + currentEnergy);
}
```

So every time the energy is regenerated (every 1.75 seconds), the program will debug the current energy as it regenerates.



In the above screenshot, I set the value of the current energy to 55 in the unity engine, so that I could see if the regenerate energy function was working as it should. As we can see the energy increases at a constant rate every 2 seconds (this value is rounded in the debug log timer.

This means that it has passed the test and shown evidence that this section is working as it should.

Review

Now that the regeneration of energy code is functional, I can focus on making the actions have energy costs. This will make sure that the energy system works as it is intended to. After that then I will need to add an energy bar so that the energy can be displayed to the users during the game. This is another step in solving the problem that was decomposed in the design, leading to a final product.

Consume energy while punching

As punching will take a fixed amount of energy every time, the function for consuming energy with a punch will be quite straight forward. The function should be classed as public so it can be used as a method in other scripts. I will use it as a method in the animation controller script, so that when the punch key is pressed, the animation is played and the stamina is reduced.

```
public void PunchConsumeEnergy(int amount)
{
    currentEnergy -= amount;
    currentEnergy = Mathf.Clamp(currentEnergy, 0, maxEnergy);
}
```

I created a new method in the stamina class that will take a certain amount of energy every time it is called. This takes in the value of how much stamina is to be reduced as a parameter, and takes it away from the current energy.

Once again the program knows to only give current energy a value between 0-100 as I have clamped it between these bounds.

```
public class RedPlayerController : MonoBehaviour
{
    7 references
    [SerializeField]public Animator animator;
    11 references
    [SerializeField]public Stamina stamina;

    0 references
    void Update()
    {
        //for punching
        if (Input.GetKeyDown(KeyCode.H) && stamina.currentEnergy > 0)
        {
            animator.SetTrigger("redPunch");
            stamina.PunchConsumeEnergy(5);
        } else if (Input.GetKeyDown(KeyCode.H) && stamina.currentEnergy == 0)
        {
            Debug.Log("NO STAMINA!");
        }
```

This is in the animation controller script for the red player . I have created a reference to the stamina class script, so that its attributes and methods can be used

When the punch key is pressed, and the stamina is greater than 0, the animation is played and the punchConsumeEnergy function is called with the “amount” as 5.

If the player tries to punch, without any stamina, then simply debug “no stamina” and no action is done.

Explanation

The above code will introduce a stamina cost for the punching action. This will mean that when the when the player uses the punch action, their stamina will be reduced by 5, and when they have no stamina, they cannot do the action at all.

There was many possible ways to implement this feature, but it made sense to add the stamina cost process in the script with the animation controls, as it had already been tested and we knew it worked as it should, This means that the only thing to now test, is to see if the stamina goes down by 5 when the player uses punch.

Key Variables

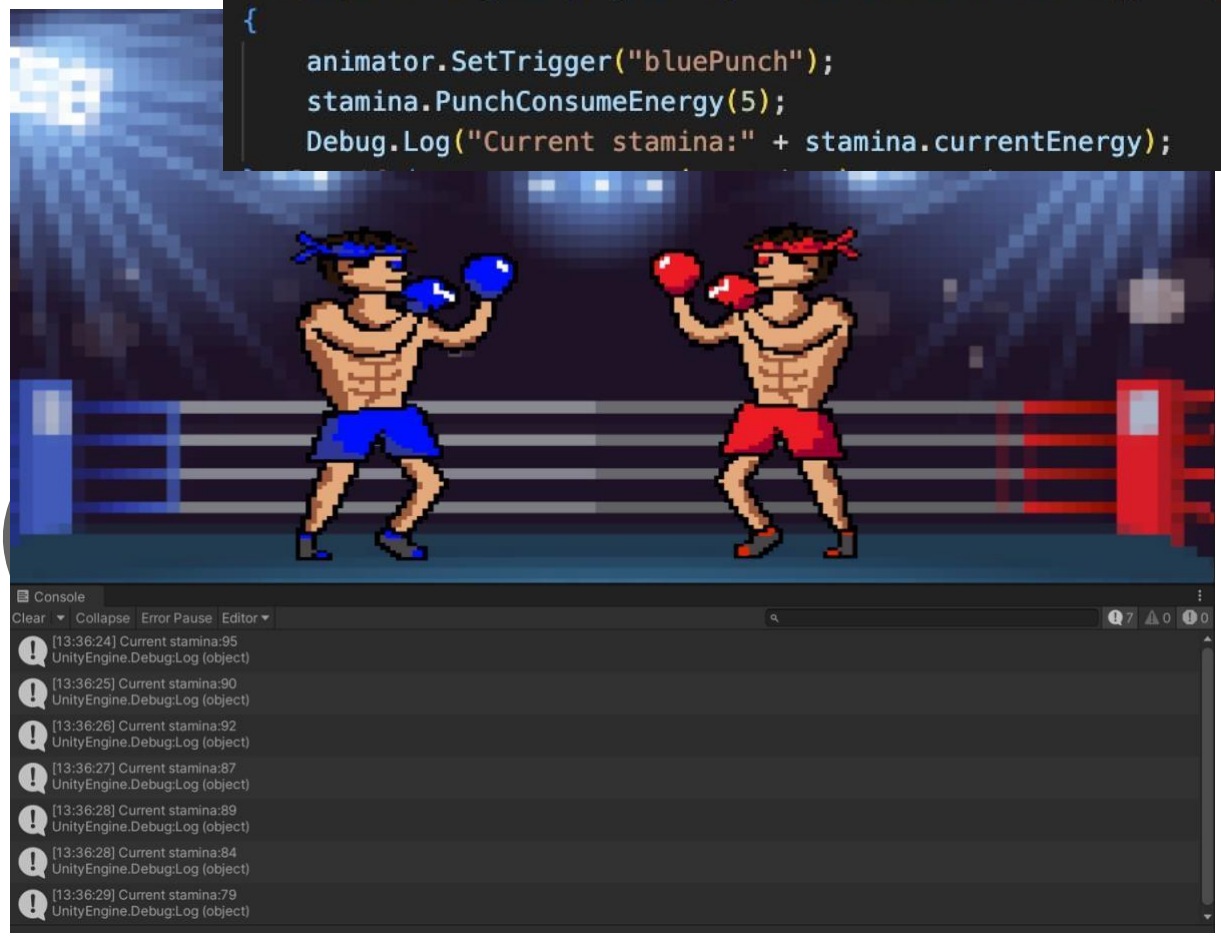
VARIABLES	TYPE	ROLE
AMOUNT	int	This is the parameter for the punching energy consumption script, and represents the amount of stamina the current stamina will decrease by.

Testing:

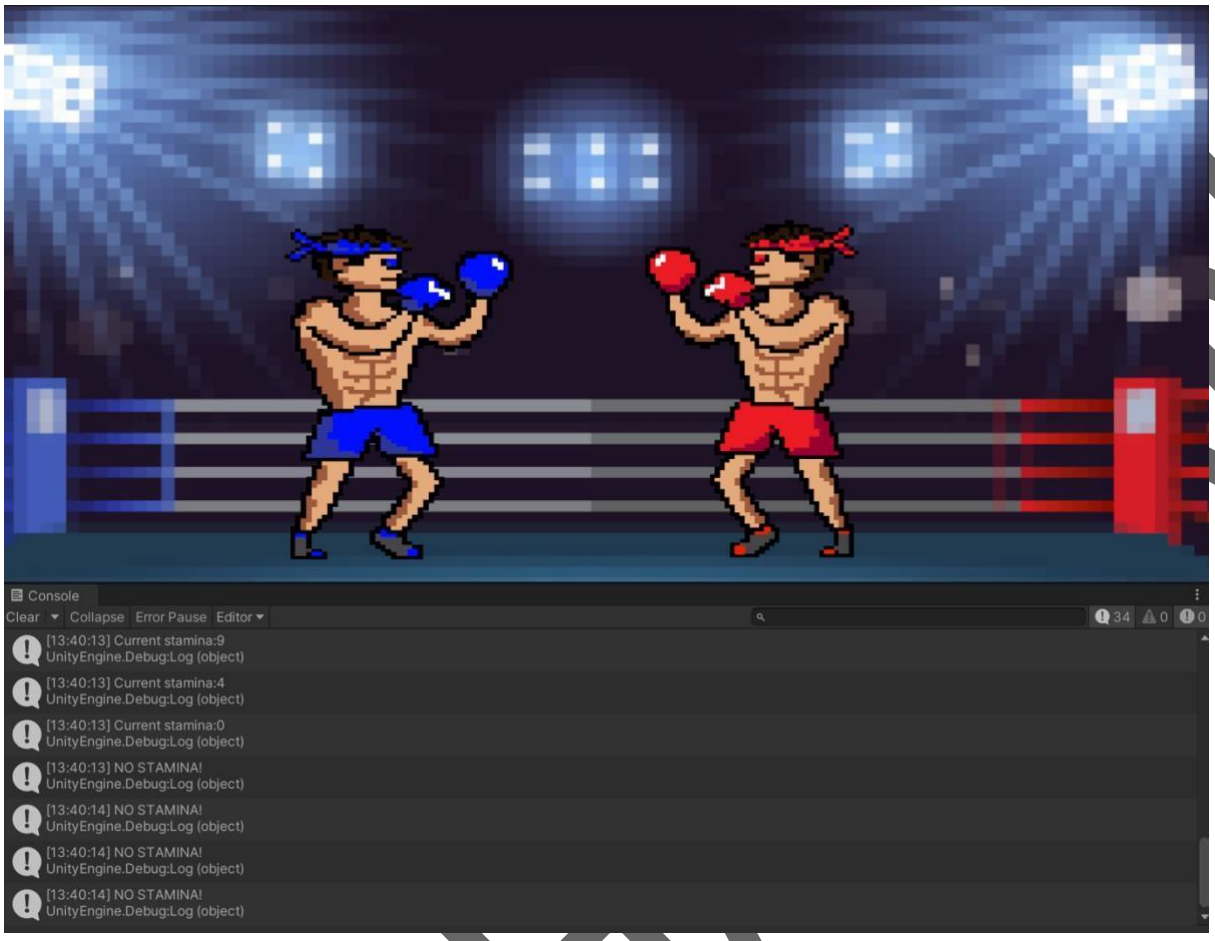
SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL
STAMINA CONTROL	Energy (punch)	Stamina variable decreases by 5 when a punch is thrown	Stamina variable decreases by 5 when a punch is thrown	PASS

Evidence

```
// for punching
if (Input.GetKeyDown(KeyCode.F) && stamina.currentEnergy > 0)
{
    animator.SetTrigger("bluePunch");
    stamina.PunchConsumeEnergy(5);
    Debug.Log("Current stamina:" + stamina.currentEnergy);
```



In the screenshot of the code, I have added a debug statement, so that when the punch action is carried out, the energy is printed so that I can see how it is changing. In the screenshot of the game, I can see that when the punch is thrown, the current stamina variable goes down, and at the same time, the energy is regenerating, so that it goes up and down at the same time.



Here we can see that the energy is going down as it should when a punch is thrown, until the energy reaches 0, where the program realises this and doesn't allow the animation to be played, and the fact that no stamina is left is debugged. The reason "no stamina" will not need to be printed for the user to see, is because there will be a stamina bar being made, which will clearly show when there is no stamina left.

Review

Now the punching action has a cost assigned to it which has been set to 5. Whenever either of the players presses the punch key, the animation plays, the damage is dealt if the punch is successful, and stamina will decrease by 5. If the player has no stamina left, then they will not do any animation, therefore meaning the damage will not be dealt as that depends on the players animation parameters.

Now I can work on giving the punching and crouching features the stamina cost as well. This is in a different section as the cost is continuous and is taken away as the blocking or crouching feature is held. This differs as punching is a single cost to do the action.

Consume energy while blocking and crouching

Blocking and crouching will drain the energy from the player as the action can be held down for as long as the player would like. This means that I will once again have to use the time syntax within Unity as well as a rate of energy drain so that the energy can be consumed as expected.

```
public class Stamina : MonoBehaviou
{
    5 references
    private int maxEnergy = 100;
    1 reference
    public float regenerationRate =
    11 references
    public int currentEnergy;
    4 references
    private float lastRegenerationT
    0 references
    public float blockingDrainRate
    0 references
    public float crouchingDrainRate
```

I have added two new floats, which represent the rate of energy drain for the actions. I set these to variables as I wanted to test different rates, to see how fast or slow the energy will drain

```
public void drainEnergy(float amount)
{
    float drainAmount = amount * (10 * Time.deltaTime);
    currentEnergy -= Mathf.RoundToInt(drainAmount);
    currentEnergy = Mathf.Clamp(currentEnergy, 0, maxEnergy);
}
```

This is the public function I have created for energy draining, which will be used for blocking and crouching. It will take in a parameter of either of the variables that I stated above, and work out how much energy will be taken away.

Once again the rounded drainAmount is taken away from current energy as current energy is an integer., and it is clamped between values 0 -100.

The drainAmount float is worked out by taking in the rate of energy drain that was passed in by the reference , named amount, and multiplying it by a scalar value (in this case I am trying 10) by the Time.deltaTime feature, which returns the time it took for the last frame to complete.

The reason I have used Time.deltaTime is because we want the energy to drain at a steady rate which is not too big. Time.deltaTime will produce a very small number for the majority of PC's, which is why I have used a scalar multiplier so that the drainAmount will always be just about big enough. For different computers, the time for a frame to complete will take different values, however it will be a consistent way for calculating the draining amount for both players on that pc, which is better than using a hardcoded value and trying to make it work for all cases.

Since the stamina script has been referenced in the animation controller script, I can simply use the public attributes and methods in the animation script, which will allow me to assign an energy cost to the blocking and crouching features.

```
// for blocking
if (Input.GetKeyDown(KeyCode.I) || Input.GetKey(KeyCode.I) && stamina.currentEnergy > 0)
{
    animator.SetBool("isRedBlocking", true);
    stamina.drainEnergy(stamina.blockingDrainRate);
}
else if (Input.GetKeyDown(KeyCode.I) || Input.GetKey(KeyCode.I) && stamina.currentEnergy == 0)
{
    animator.SetBool("isRedBlocking", false);
    Debug.Log("NO STAMINA!");
}
if (Input.GetKeyUp(KeyCode.I))
{
    animator.SetBool("isRedBlocking", false);
}
```

I have added extra conditions to the Boolean expressions for these if statements, which include consideration of the stamina. The first if statement will do the action when the stamina is greater than 0, and when the "I" Key is being held down, it will consistently call the drainEnergy function from the stamina script, leading to a reduce in energy over a period of time.

However when they press or hold the "I" key and they do not have any stamina left, the animation is not set to true, meaning the action is not done, and for testing purposes, It will debug "NO STAMINA!" so that I can monitor the behaviours of the game object over time.

The last if statement is identical to how it was written before, just meaning when the key is lifted, the blocking animation stops.

Explanation:

This section of the code will drain the energy of the player over time as they hold down the blocking or crouching feature, which was needed due to blocking and crouching being continuous features that can be held down for a long period of time, contrasting to the punching function where the action is done one time, every time the relevant key is pressed.

Key variables

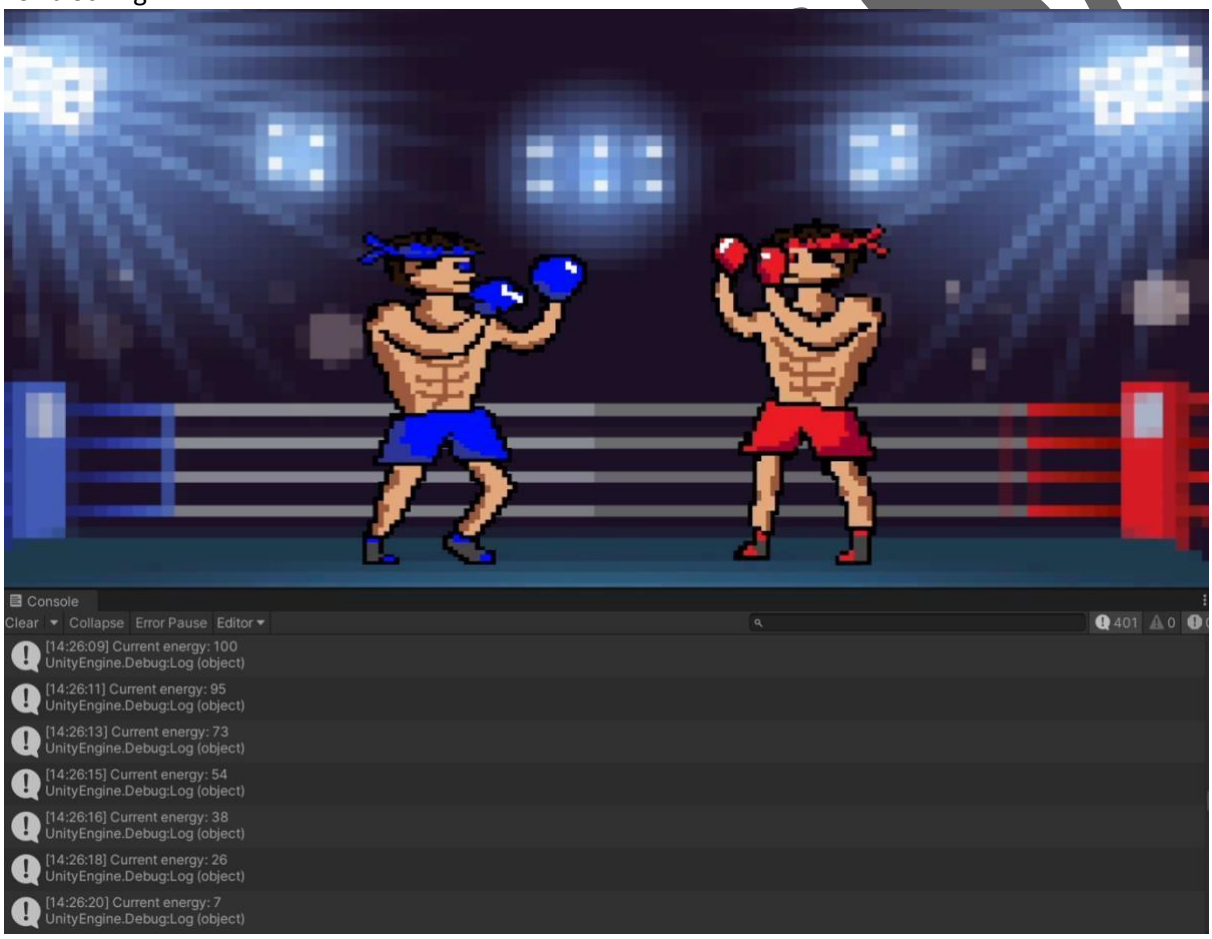
VARIABLES	TYPE	ROLE
BLOCKING DRAIN RATE	float	This is the rate at which the energy will drain from the character when holding the blocking feature, and is used to work out how much energy should be taken away from the character when they use the blocking action.
CROUCHING DRAIN RATE	float	This is the rate at which energy will drain from the character when the crouching feature is used. This is larger than the blocking feature as when used, it nullifies all damage taken, therefore it needs to have a higher energy cost to balance this out.
DRAIN AMOUNT	float	This is the energy to be taken away from the characters current energy, and is calculated using the drain rate of that action, and the time functions in the unity editor.

Testing

SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL
STAMINA CONTROL	Energy (block)	The energy should drain at a slow and constant rate	The energy decreases at a steady rate when the blocking action is used	PASS
	Energy (crouch)	The energy will drain at a faster rate than blocking	The energy decreases at a steady rate when the crouching action is used, and it is sufficiently faster than blocking	PASS

Evidence

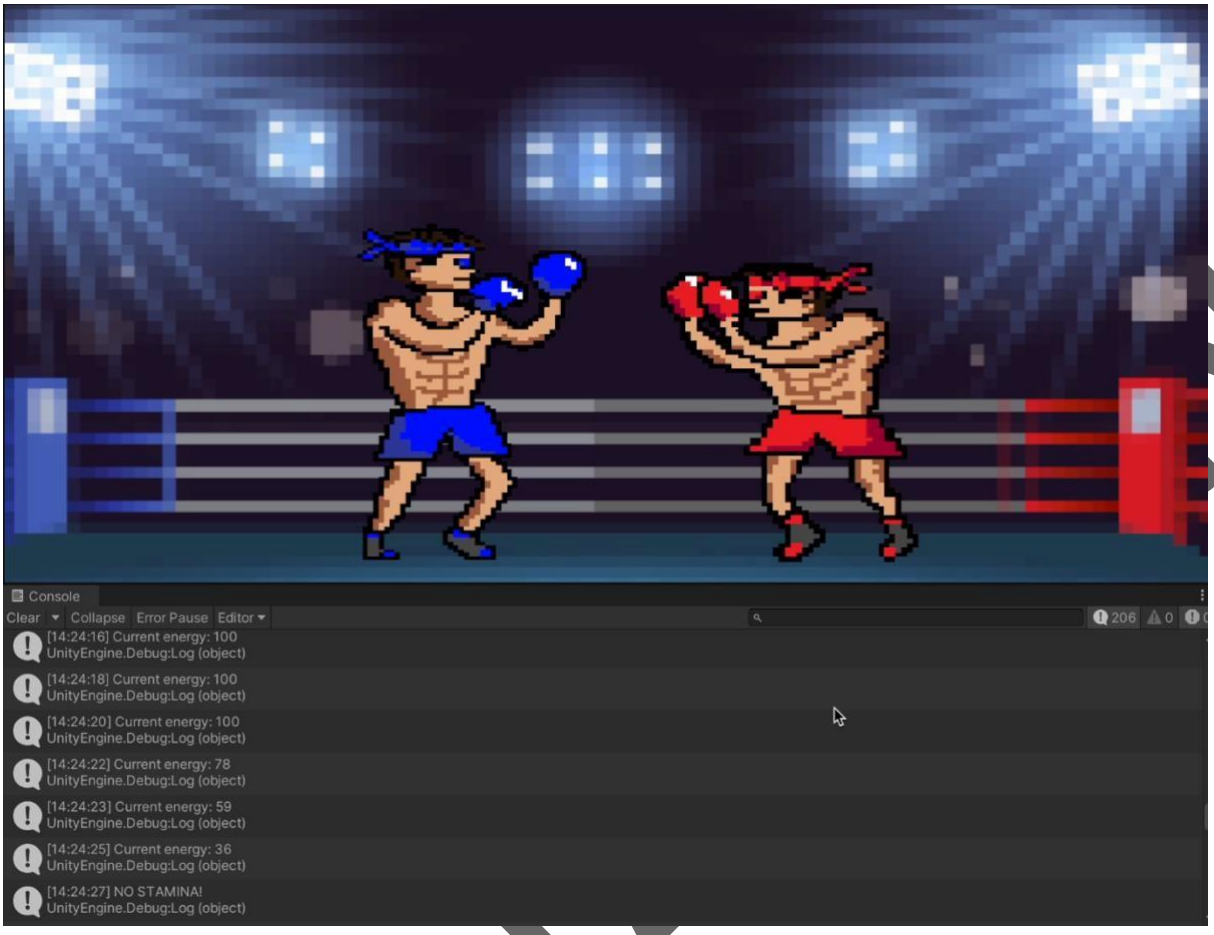
For blocking:



This proves that the energy is drained as it should over time using this action

As you can see above, the energy is drained from 100-0 in about 11-12 seconds through holding down this action. This is sufficient time, considering that the player should not be using this action for this long anyways, otherwise they will most likely lose.

For crouching:



This shows that the stamina is drained over time while the crouch action is used.

As the crouching action fully nullifies damage to that player, it is the most balanced option for it to cost more stamina. So constant use of the crouching action will take the user from 100->0 stamina in about 7 seconds. This action should necessarily not be held down for that long anyways, and should be in reaction to another players action.

Review

Now that the blocking and crouching actions have an energy cost assigned to them, the stamina control functionality has all been added and is working as expected. I can now add the UI feature, which is the energy bar.

Since the health and energy both need a graphical representation in the form of a health or stamina bar, I can now make these together.

Health / Stamina bar

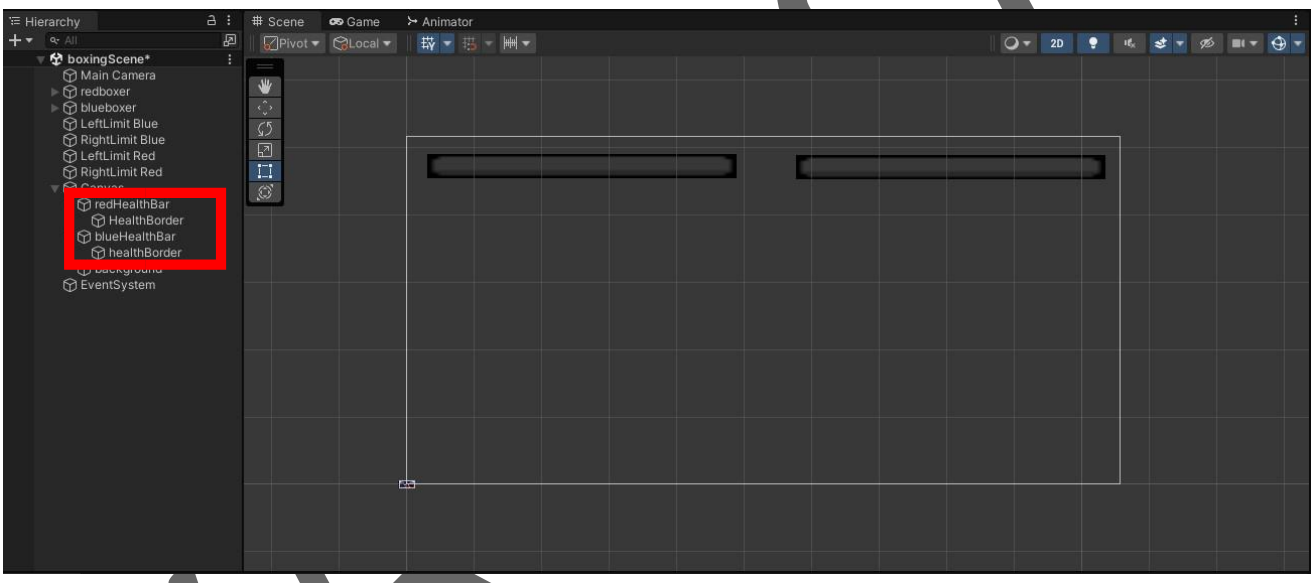
Now that the functionality of both of these features has been added, and they are working as intended, I can now add graphical representation of these features, so that the players can easily see the progress of the game while it is happening. This includes a health bar and stamina bar for each of the players.

The health bars will work by created a UI design of them in Unity, and controlling their features using the currentHealth or currentEnergy variables, and making the bar fill to the same percentage as the current variable values are to their max value, which in both cases is 100.

Health bar

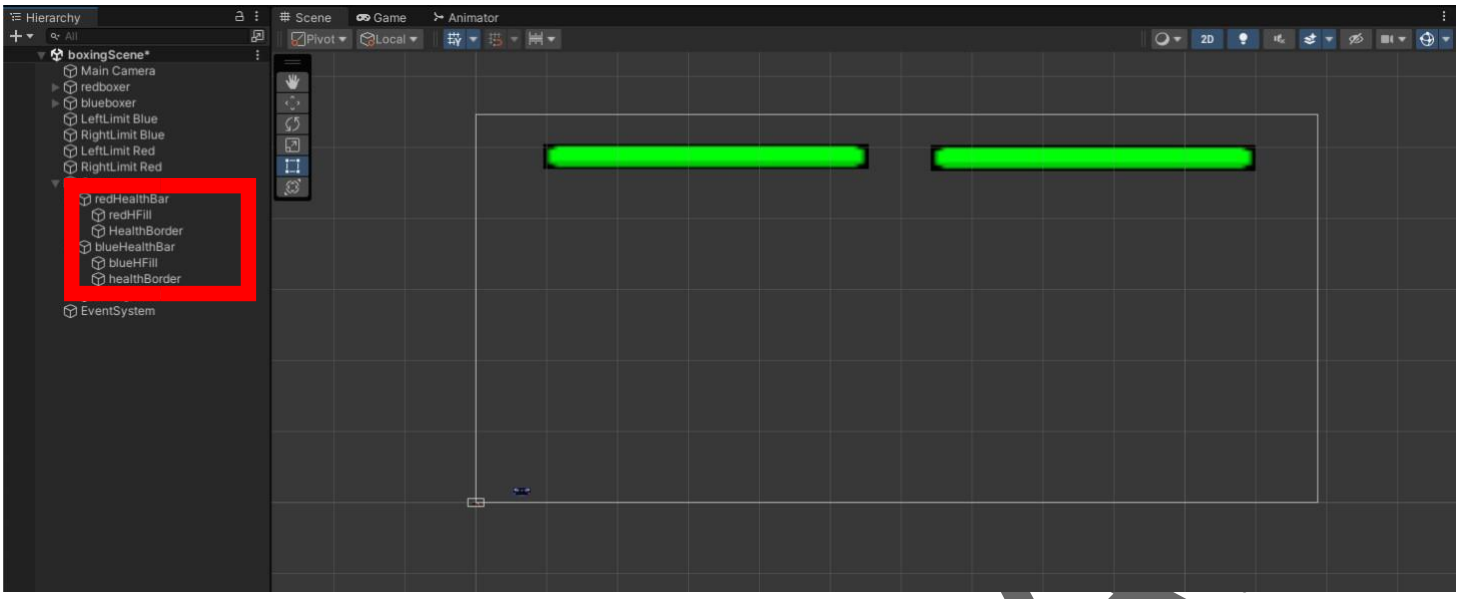
I will first create and design the health bar in unity.

The health bars will have two main aspects, the border, and the fill. The border is simply the design for the edges of the health bar, which is not too important, but the fill is the part which will be changing in relation to the currentHealth variable. I will do this by creating a slider game object in unity, which will take a value between 0, and a max value of my choice (100), and will change length depending on the value given to it.
This is useful as I can match it pretty easily to the health variable, so that it takes the value of length from that.



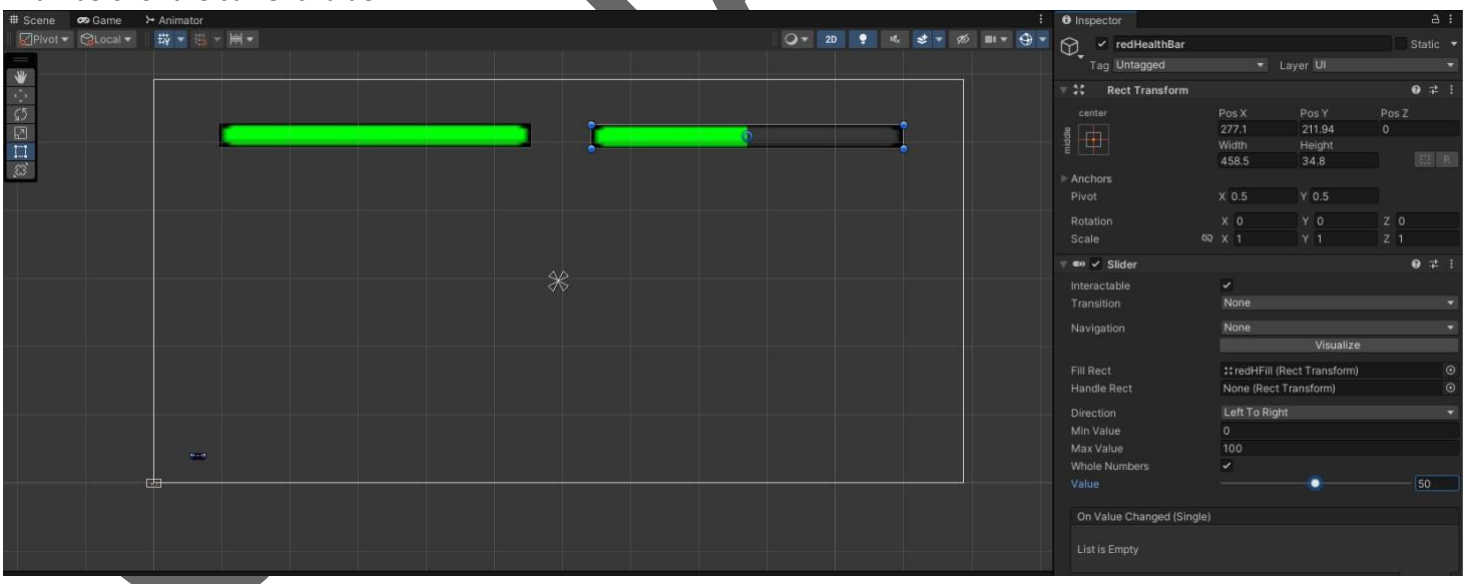
Here we can see that I have created two game objects, redHealthBar and blueHealthBar, and created child game objects which are called the health borders. The health borders have been assigned the image that I have found and will be using for the health bars.

Now I will create two new images for each of the health bars called fill, which will be the image changing with the slider. I have given these a green colour as this is what is usually used for the colour of health bars in games. However later I might think about adding a feature which changes the colour of the health bar as the health decreases.



I have added the fill, and transformed it so that it matches the size of the border, and so that it sits underneath it perfectly, and have grouped them together so that they will now move together.

To the redHealthBar / blueHealthBar game objects, I have added a slider component. In the slider component I can control what image is being used as the slider, which I have set to the fill of each health bar, and I can set the min and max value. The max value has been set to 100 and the min value has been set to 0, so that it matches perfectly with the current health variable, and it will always be in a perfectly representative position so that it is accurate enough for the player to understand it. Since current health is also an integer, I have set it so that the slider can only take whole numbers for the current value.



As can be seen in the screenshot, I have tested the slider with the value of 50, which is shown to be directly in the middle of the health bar as it should. Now that the health bar is full set up, I need to link its value to the current health variable so that it updates as the health variable changes.

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

In the health handler script, I have declared another namespace to use, “UnityEngine.UI” which allows me to access certain UI components such as a slider.

```
public class BlueHealthHandler : MonoBehaviour
{
    2 references
    public int health = 100;
    5 references
    private float currentHealth;

    4 references
    [SerializeField] public Animator animator;
    3 references
    public Slider slider;

    0 references
    void Start()
    {
        currentHealth = health;
        slider.maxValue = health;
    }
```

I have declared a reference to the slider, so that I can access certain components of the slider in this script

In the start function, I have set the value of the slider to the max value for health, just like the current health variable.

```
0 references
void Update()
{
    slider.value = currentHealth;
}
```

In the update function, the value of the currentHealth is constantly being changed with the value of the currentHealth variable, meaning it will always be representing the health as it should

Explanation

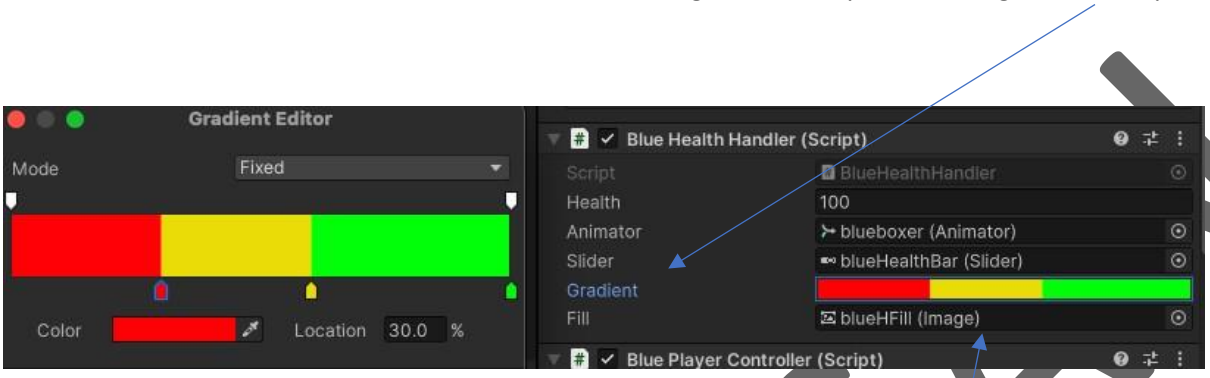
The value of the slider is being manipulated in the health controller script, so that it is always set to the value that the current health variable has. This means that the slider is always representing the health variable, therefore meaning it is an accurate representation for the player. Linking back to the problem decomposition, this was a necessary UI feature, as it will mean that both players will always know the progress of the game, and who is winning at that moment in time. In reaction to the health bar, they may change their playstyle to ensure that they win the game.

Adding gradient fill

To make it even more representable for the health, I have decided it will be easier for players to understand what value their health is at if the colour of the bar changes when at certain values. For example, when the health =

60, the health bar will change colour from green to yellow, and when it is below 30, it will change to red. This will make it even easier for the player to know the progress of the game, without even having to look at the health bar, as their peripheral vision will pick up the sudden colour change, and allow them to focus on the game even better.

To do this, I have added two new references to components in the game from the health handler script, which is the fill of the health bar, which holds the colour of the health bar, and a gradient component. The gradient component allows



The gradient I have made will be green from 100%-60%, yellow from 60%-30%, and red from 30%-0%. In the gradient component, the colour at that point is represented by the normalised value from 0-1. This means that for example, the colour at the point where the value is 0.60, is yellow, and at 0.10, the colour is red.

```
public class BlueHealthHandler : MonoBehaviour
{
    2 references
    public int health = 100;
    5 references
    private float currentHealth;

    4 references
    [SerializeField] public Animator animator;
    4 references
    public Slider slider;
    2 references
    public Gradient gradient;
    2 references
    public Image fill;


    0 references
    void Start()
    {
        currentHealth = health;
        slider.maxValue = health;
        fill.color = gradient.Evaluate(1f);

    }
```

I have added the reference to the gradient and the fill, which I had assigned game objects to up here.

In the start function, I have assigned the colour of the health bar fill, to the colour representing 1 on the gradient, which is green.

```
0 references
void Update()
{
    slider.value = currentHealth;
    fill.color = gradient.Evaluate(slider.normalizedValue);
}
```

In the update function, the code will always set the colour of the health bar fill to the relevant value of the gradient scale. Since. The slider can take values from 0-100, it has to be normalised so that it will take a value between 0 and 1, but still represent the same position. So the fill will be the colour of the gradient at that specific place on the slider

me to make a gradient colour scheme, in which the fill can follow off.

IMPORTANT – at first I was confused on how I could make this feature. Since the health bar is more of an optional UI feature (I could have simply displayed the health as text on the screen), I thought reinventing a new way to make this would be an inefficient use of time, and that there is only so many ways that you can make a health, so I was able to find a tutorial with how to do it on Youtube, and can confirm that I mainly used the video for this feature:

https://www.youtube.com/watch?v=BLfNP4Sc_iA&t=773s&ab_channel=Brackeys

Explanation of code

The gradient component assigns a colour to a value between 0 and 1. The slider, which represents the health between 0 and 100, will be normalised, so that it contains a value between 0 and 1 (but still represents the same position), and will be constantly changing, so the "gradient.Evaluate(slider.normalizedValue)" will be changing colour when it reaches 0.6 (green to yellow) and 0.3 (yellow to red). This means the colour of the health bar fill will also change when the slider reaches 0.6, or 0.3 of the full length.

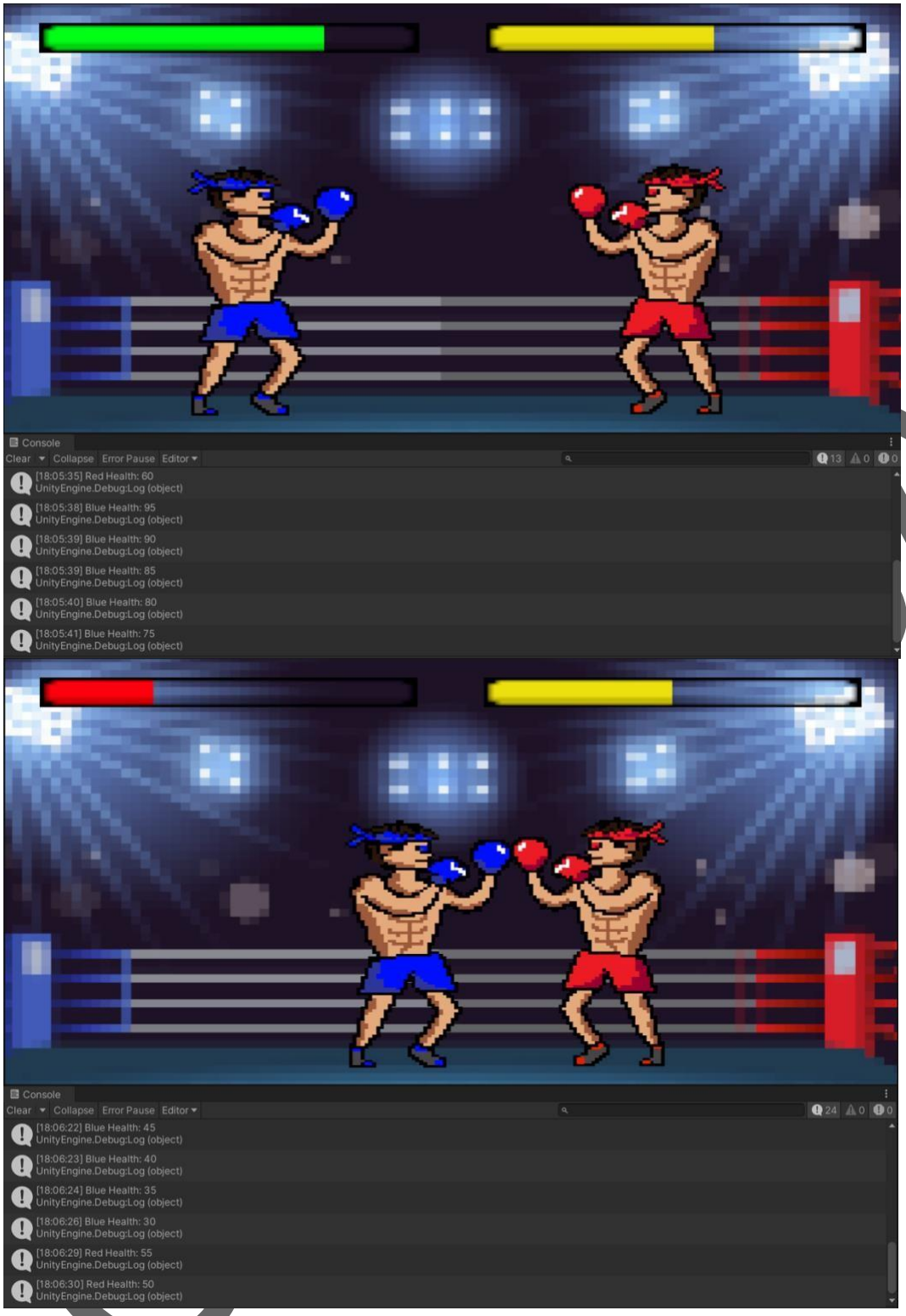
Now the health bar will change colour depending on the current health of the player, which is good as it will add extra ease when trying to understand progress of the game while playing it. Key Variables

VARIABLES	TYPE	ROLE
SLIDER	Slider (UI feature)	This is the component I am using to control the length of the fill of the health bar, and I am using this to make it equal to the value in the health bar.
FILL	Image (UI feature)	I am changing the colour of the fill of the health bar in the script, so I have to reference it in the code.
GRADIENT	Gradient (UI feature)	The colour of the health bar will be assigned from this colour.

Testing

SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL
HEALTH CONTROL	Health bar	Health bar should show the health variable changing during the course of the game.	Health bar should show the health variable changing during the course of the game.	PASS

Evidence



As we can see in both of the screenshots above, the health is debugged, and the health bar is an accurate representation of the health variable. When the health reaches 60, the bar turns yellow, and when the health is 30, the bar turns red.

EDIT: The video of the game shows the health bar working as it should.

Review

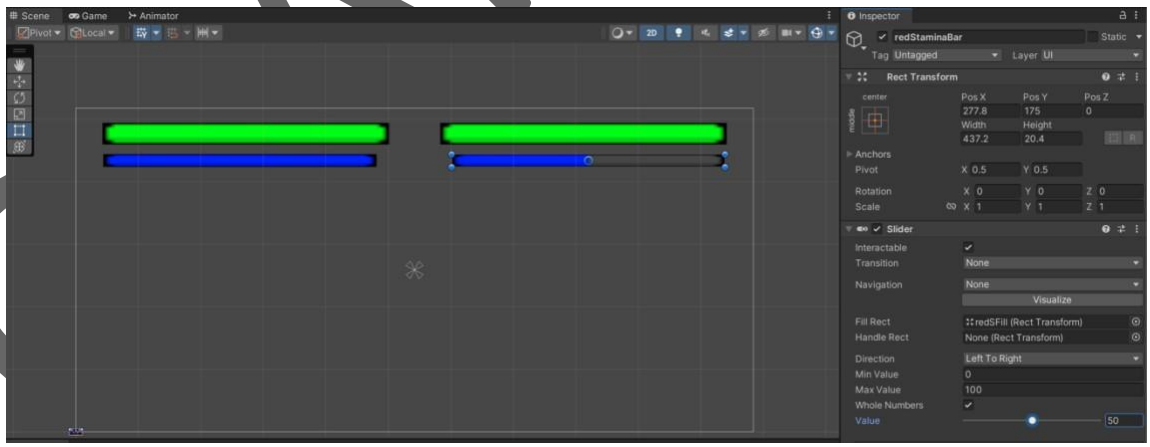
Now the health bar has full functionality as it constantly updates its position while the health changes. It also changes colour when the health reaches certain values. This means that the players can easily assess the progress of the game at all times.

Now this is done, I can move on to creating the energy bar, which is very similar.

Stamina bar

The stamina bar will be set up in the exact same way as the health bar, but it will be blue, and will not have a gradient for different stages of stamina, as the stamina recharges meaning it will be constantly changing colour and could distract the user.

Below shows a screenshot of the stamina bar below the health bar, with the value being at 50 (the slider value component on the bottom right) and the stamina bar showing a good visual representation of that.



The code will be pretty similar as well, but instead it will be following the value of the currentEnergy variable instead.

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

0 references
public class Stamina : MonoBehaviour
{
    5 references
    private int maxEnergy = 100;
    1 reference
    public float regenerationRate = 3f;
    11 references
    public int currentEnergy;
    4 references
    private float lastRegenerationTime;
    0 references
    public float blockingDrainRate = 11f;
    0 references
    public float crouchingDrainRate = 13.5f;
    2 references
    public Slider slider;
```

I have used the unity engine UI name space so that I can access the slider component of the game object

I have created a reference to the slider which I set up to be used as the stamina bar

```
0 references
void Start()
{
    currentEnergy = maxEnergy;
    lastRegenerationTime = Time.time;
    slider.maxValue = maxEnergy;
}

0 references
void Update()
{
    //regenerate energy constantly after 1.75 seconds from last regen
    if (Time.time - lastRegenerationTime >= 1.75f)
    {
        RegenerateEnergy();
    }
    slider.value = currentEnergy;
}
```

At the start, the stamina bar slider will take the value of max energy, which = 100. This is also the max value of the slider as shown above

In the update function, I want the value on the slider to constantly update to the value of currentEnergy as this will be changing a lot during the game

Explanation

As mentioned before, this is very similar to the health bar, but it follows the currentEnergy variable. It will constantly update its value with that variable. This means that users can now see how much stamina they have left, so that they can plan what style of fighting they would like to adopt, and allows them to improve their critical thinking skills.

Key variables

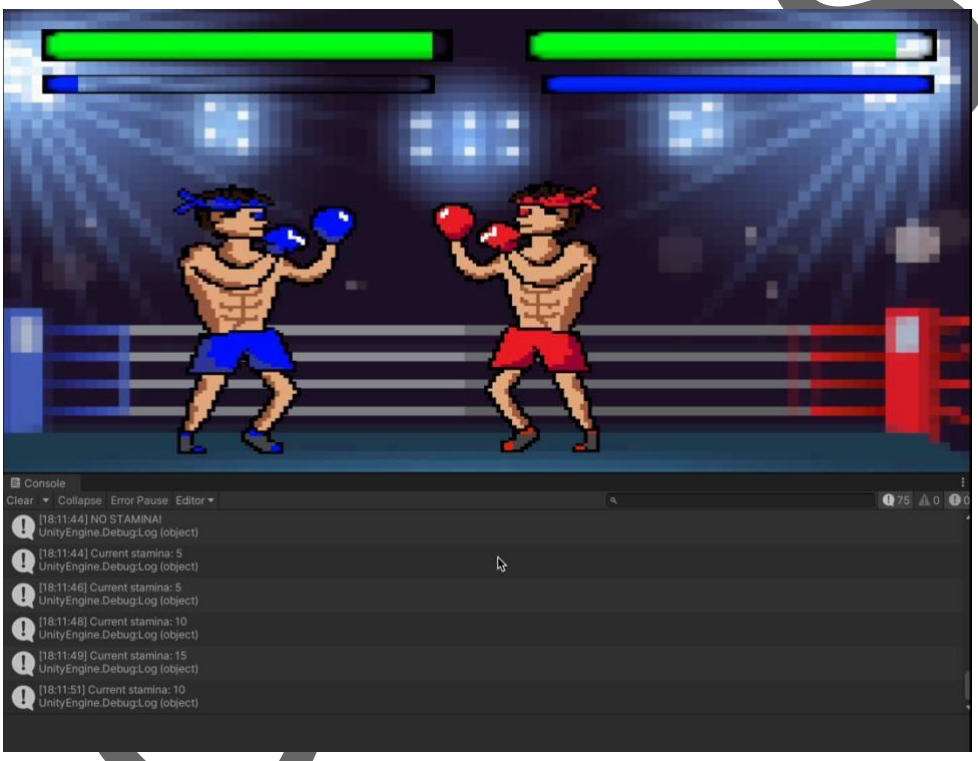
VARIABLES	TYPE	ROLE

SLIDER	Slider (UI feature)	This component is being used to control the length of the stamina bar by assigning it a value of the current energy so that its length its length compared with the max length of the bar, is representative with the current energy compared to the max energy the player can have.

Testing

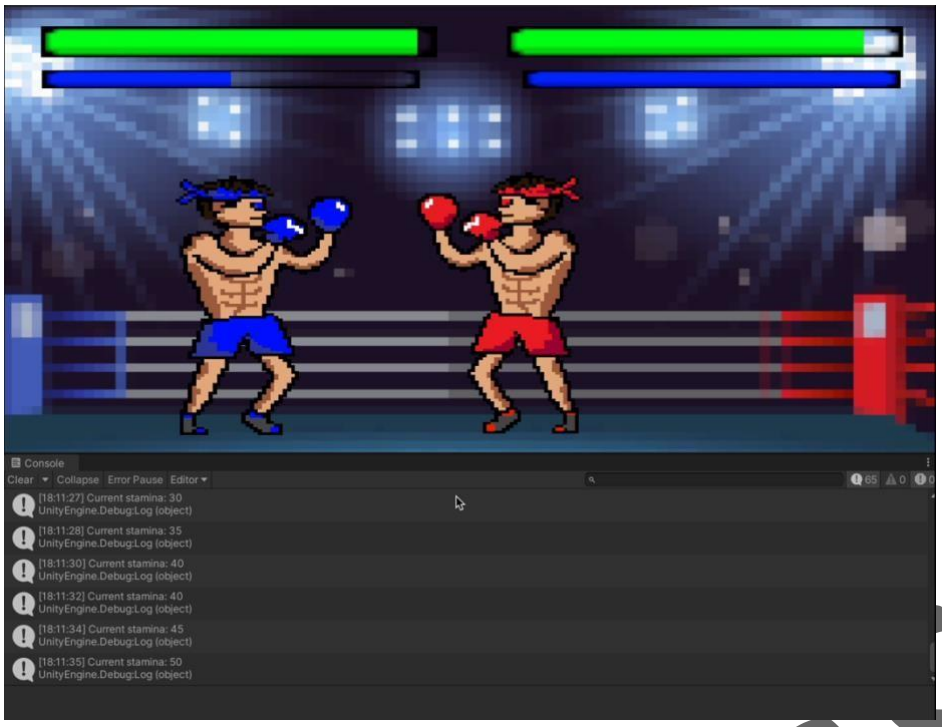
SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL
STAMINA CONTROL	Stamina bar	Will show a visual representation of the energy variable being changed	The length of the bar changes, with the current energy variable, when actions that cost energy are used	PASS

Evidence



These are both screenshots of the stamina bar working as it should. The value “currentEnergy” (I mistakenly used the wrong name for the variable while making the debug statement) variable is shown to be changing, and the stamina bar is following the visual representation of the value being held in the variable.

This shows the stamina being low, and the stamina representing this



This shows the stamina variable being at 50, and the stamina bar being at the middle of the full bar

Review

This feature, along with the health bar is vital to the game, otherwise players will never know the progress of the game, and will be blindly punching/blocking/crouching without any knowledge of the stamina costs associated with them.

Now that the stamina bar has been created, the 1v1 (player vs player) game mode is practically functional, and the gameplay works as it should. Now I will move onto making the 2 levels of the AI's. Then after this I can work on adding the other UI features such as the menus and the game over screen.

Level 1 AI

To make the AI game mode, I have duplicated my 1v1 project into a new scene and named it AI boxing scene. I also altered the code for the red boxer scripts (renaming them to AIboxer….) so that the functions like moving left and right, punching, blocking and crouching do not take input, but rather just have a function which carries out that action. This is needed so that when the AI Decides what action to do, the function can be called easily.

The criteria I wanted to meet for the first level AI was to:

1. Have actions each with an assigned weighting so that some actions are more likely to happen over others, those actions being decided by me. These will be decided based on what I think will make a human like player
2. Have the actions duration be a random time between limits that I choose
3. Don't allow blocking/crouching or moving in either direction happen after they have just done that action. For example, the AI will be able to crouch after blocking, but not pick blocking straight after blocking. However they can punch after just having done the punch action

First for the AI to work in the functions that I have created, I had to make the actions into functions themselves, that can be called for a random amount of time. This was a case of adapting the scripts for movement and the actions so that they would make the AI do those actions First I took and adapted the movement script:

```
0 references
public void MoveLeft()
{
    moveDirection = -1.0f;
}

0 references
public void MoveRight()
{
    moveDirection = 1.0f;
}
```

In the new "AIMovement" script, I changed the "horizontal ...." Variable to move direction. When the moveleft function is called, the direction vector will be turned to -1 so the AI moves left, and the same but opposite goes for the right movement

The AI player controller (controls the actions) script was also an adaptation off the normal player controller script:

Coroutines were created so that code could be done line after line for the blocking and crouching actions

Since punching is just a single action that happens, and doesn't require the button to be held down, it doesn't need a coroutine and can just be set as a function

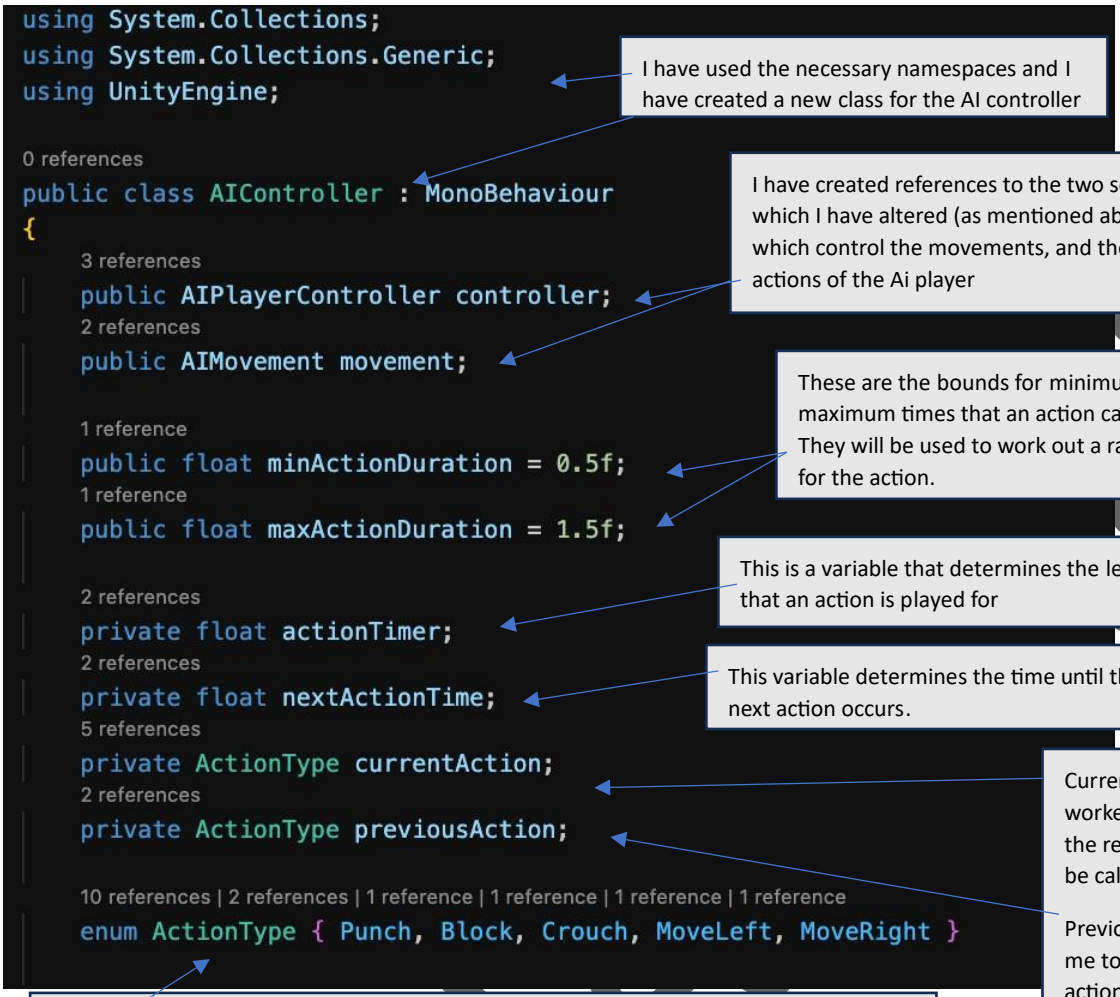
The block function uses the coroutine if it has been called and the stamina is greater than 0. If not then "no stamina" is debugged and the same is done for the crouch function. If stamina is greater than 0, then the crouch coroutine is called, and if it is 0, then "no stamina" is debugged and the coroutine is not called

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

0 references
public class AIPlayerController : MonoBehaviour
{
    7 references
    [SerializeField] private Animator animator;
    8 references
    [SerializeField] private Stamina stamina;
    0 references
    private Coroutine crouchCoroutine;
    0 references
    private Coroutine blockCoroutine;


    0 references
    public void Punch()
    {
        if (stamina.currentEnergy > 0)
        {
            animator.SetTrigger("redPunch");
            stamina.PunchConsumeEnergy(5);
        }
        else
        {
            Debug.Log("NO STAMINA FOR PUNCH!");
        }
    }


    0 references
    public void Block()
    {
        if (stamina.currentEnergy > 0)
        {
            StartCoroutine(BlockCoroutine());
        }
        else
        {
            animator.SetBool("isRedBlocking", false);
            Debug.Log("NO STAMINA FOR BLOCKING!");
        }
    }


    0 references
    public void Crouch()
    {
        if (stamina.currentEnergy > 0)
        {
            StartCoroutine(CrouchCoroutine());
        }
        else
        {
            animator.SetBool("isRedCrouching", false);
            Debug.Log("NO STAMINA FOR CROUCHING!");
        }
    }
```

```
1 reference
private IEnumerator CrouchCoroutine()
{
    animator.SetBool("isRedCrouching", true);
    stamina.drainEnergy(stamina.crouchingDrainRate);

    float duration = Random.Range(1.5f, 3f);
    yield return new WaitForSeconds(duration);

    animator.SetBool("isRedCrouching", false);

}

1 reference
private IEnumerator BlockCoroutine()
{
    animator.SetBool("isRedBlocking", true);
    stamina.drainEnergy(stamina.blockingDrainRate);

    float duration = Random.Range(2f, 4f);
    yield return new WaitForSeconds(duration);

    animator.SetBool("isRedBlocking", false);
}
```

The coroutine for crouching will set the animator to true, and will drain the energy while it is being used. The coroutine will also play for a random amount of time, keeping the AI actions even more random, then after the random duration has passed, the crouching animation is set to false

The coroutine for blocking will make the blocking animation play, and drain the energy. It will wait for a random duration between 2 and 4 seconds, and after the time is up, it will deactivate the animation.



I have used the necessary namespaces and I have created a new class for the AI controller

I have created references to the two scripts which I have altered (as mentioned above) which control the movements, and the actions of the Ai player

These are the bounds for minimum and maximum times that an action can be used. They will be used to work out a random time for the action.

This is a variable that determines the length that an action is played for

This variable determines the time until the next action occurs.

Current action will be worked out by the AI and the relevant function will be called

Previous action will allow me to prevent certain actions being repeated one after the other

This is an enumerator which holds names of certain constants, which in this case are the action, not the actual functions (these will be assigned in a switch case statement). Having the actions in this structure will also be useful for when I am adding the weightings to each action.

Now that all the references and variables had been added, I could now create a the main function which is to choose the next action.

```
2 references
void ChooseNextAction()
{
    actionTimer = 0f;

    // define weights for each action
    float[] actionWeights = { 5f, 1f, 1f, 1.75f, 1.75f};

    // choose a random action that is not the same as th
    do
    {
        // choose a random action based on weights
        float totalWeight = 0f;
        foreach (float weight in actionWeights)
        {
            totalWeight += weight;
        }

        float randomValue = Random.value * totalWeight;

        for (int i = 0; i < actionWeights.Length; i++)
        {
            if (randomValue < actionWeights[i])
            {
                currentAction = (ActionType)i;
                break;
            }
            randomValue -= actionWeights[i];
        }

    } while (currentAction == previousAction && currentAction != ActionType.Punch);

    previousAction = currentAction;

    //calculate the next action time
    nextActionTime = Time.time + Random.Range(minActionDuration, maxActionDuration);
}
```

The actionTimer variable is reset every time the function is called

This is an array that has been defined as to only take values of floats, in which I have assigned weights to each of the actions. The weightings are ordered in the way they are ordered in the ActionType enum, so that the index's are the same

A totalWeight variable is declared with the value of 0, as this will be incremented in the for-each statement.

In this block of code, the program will iterate through every weight in "actionWeights" and will increment the weight to be the total weight. The main reason I implemented this feature is because I was constantly changing the weights, depending on what I thought was suitable. If I had just assigned totalWeight a value, I would've had to calculate the total weight every time while I was testing the feature.

I have then created a new variable for a randomValue, this is worked out by multiplying the total weight by the Random.value in built function, which produces a random value between 0 and 1. This will effectively come up with a random value between 0 and total weight.

In this for loop, It will be looping through every weight in "actionWeight" cumulatively, checking whether the random value generated was less than the current cumulative weight. If it is, then that means that that certain action has been chosen, and currentAction will be assigned the relevant index in the ActionType enumerator. It must break when the condition is true, otherwise the if statement will always be true after the first time it has been found to be less than the certain weight. It is made cumulative by the last line here.

This works out the time for the action to play until a next one is chosen

I have used a do-while statement which will repeat the code inside while the current action that has been chosen is the same as the previous one, AND while is not the punching action. If the current action chosen is the same, however if the action chosen was to punch, it means that the condition has been met, as the punching action can happen multiple times one after the other.

The previous action is given the contents of the currentAction variable so that when the next action is being chosen, it can be compared so that the same two actions don't happen again unless it is a punch

```
1 reference
void PerformAction(ActionType action)
{
    switch (action)
    {
        case ActionType.Punch:
            controller.Punch();
            break;
        case ActionType.Block:
            controller.Block();
            break;
        case ActionType.Crouch:
            controller.Crouch();
            break;
        case ActionType.MoveLeft:
            movement.MoveLeft();
            break;
        case ActionType.MoveRight:
            movement.MoveRight();
            break;
    }
}
```

A function is used for performing the action that was chosen in the ChooseAction function

A switch- case statement is used which takes in the action that is passed into the function, and does the relevant action by checking the cases

For the punch, block and crouching feature, these functions are all in the AIController script which I referenced and assigned the name controller. In the case where those actions have been chosen, the relevant function has been called

Both of the movement actions are in the AIMovement script which I also referenced before in the code, and gave the name movement. The functions are called from this script.

```
0 references
void Start()
{
    ChooseNextAction();
}

0 references
void Update()
{
    actionTimer += Time.deltaTime;

    if (Time.time >= nextActionTime)
    {
        //perform the current action
        PerformAction(currentAction);

        //choose the next action
        ChooseNextAction();
    }
}
```

Before the game is started, the program is told to choose the first action, this starts the actionTimer meaning that the function will automatically be called over and over again until the game is over.

In the update function, the actionTimer is constantly incremented every frame, which represents the time that the action has been played for.

When the time has come for the next action, which would've been calculated in the ChooseNextAction function, the PerformAction function is called, passing in the parameter of the action that was chosen beforehand, and the next action is chosen

Explanation

This is the first model of the AI, and it has been created in a way that It imitates calculated random behaviour that can be a human-like feature. The way this has been done is by assigning weights to the different actions, so that they can be chosen in a probabilistic fashion. I have also made it so that the actions will be done with different durations, so that it is not predictable when the Ai is going to do the next move, which would make it quite easy to strategize against.

Key variables

VARIABLES	TYPE	ROLE
MINACTIONDURATION	Float	Holds the value of the smallest duration an action can be done for
MAXACTIONDURATION	Float	Holds the value of the largest duration an action can be done for
ACTIONTIMER	Float	Holds the time elapsed from when the current action started
NEXTACTIONTIME	Float	This is used to determine when the AI should do the next action
CURRENTACTION	Action Type	This holds the current action that is being done, and is worked out in the ChooseNextAction() function
PREVIOUS ACTION	Action Type	This holds the previous action that was done, and Is used to prevent all actions (except from punching) from being repeated twice in a row
ACTIONTYPE	Enumerator	This holds the names of the actions, and is used in the PerformAction() function so that the relevant function can be called to do said action
ACTIONWEIGHTS	Float Array	This is an array consisting of floats of the weightings that each action holds
TOTALWEIGHT	Float	This is the sum of all of the probabilities, and is used when getting a random value which decides the action to be done.
RANDOMVALUE	Float	The random value is a random value in the range from 0 to totalWeights and is used for deciding which action should be chosen
ACTION	Action Type	This is a parameter that is passed into the PerformAction() function, and takes its value from the CurrentAction variable. It allows the switch-case statement to call the correct function for the action.

Testing

SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL
LEVEL 1 AI	Basic functionality of simple AI	AI should be able to execute a sequence of predefined actions in a random fashion, since punching is weighted highest, punching should occur the most.	The AI chooses an action, based on weightings at random from the set list of actions. It cannot do any action twice in a row, except for punching, and punching as the highest weighting so that action occurs the most often	PASS

Evidence

There is no specific validation for any of the parts in this section so I have decided to prove it together with the other AI at the end, however the evidence that it passes the test is in the video in the "Video of Game" section below at the end of the development.

The time stamp for the Level 1 AI section is **0:36**

Review

The level 1 AI has been a success, and it does everything I intended it to do. Moving on, the level 2 AI will be an adaptation of this AI, so the fact that I have made it work perfectly now allows me to start the development of the second AI. The adaptation will mainly be in the ChooseNextAction() function, where it will be changing the values of the weightings.

Once this is done, then all of the game play and game mode features are done, and I will be on the finishing steps with only the GUI to work on.

Level 2 AI

This is the level of the AI that will take input from the user's character, and adapt the weightings of the different actions constantly. This will be done using Bayes' theorem as I mentioned in the design section.

To get the recent moves from the player, I will have to modify the BluePlayerController (controls the input for the actions) and the BluePlayerMovement (controls the movement) scripts for the level 2 AI scene so that when the animations are played, the type of action is also pushed onto a "recentMove" stack. A problem arises with the blocking/crouching/movement actions due to them being continuous functions meaning that I will need to modify it so that the move is only pushed onto the stack one time, and not multiple times for when the relevant button is pressed

Getting recent actions

I have decided to use a stack to get the actions, as it is a Last in first out (LIFO) data structure. This is important because I will be wanting to see the most recent moves, while still wanting to keep count of how many of a certain action they have done overall. To do this, I have made a new script to set up the stack.

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

0 references
public class BlueRecentMoves : MonoBehaviour
{
    1 reference
    public Stack<ActionType> recentMoves = new Stack<ActionType>();

    3 references | 0 references | 0 references | 0 references | 0 references | 0 references
    public enum ActionType { Punch, Block, Crouch, MoveLeft, MoveRight }


    0 references
    public void DebugRecentMoves()
    {
        string debugMessage = "Recent Moves: ";

        foreach (ActionType action in recentMoves)
        {
            debugMessage += action.ToString() + ", ";
        }

        debugMessage = debugMessage.TrimEnd(' ', ',');

        Debug.Log(debugMessage);
    }
```

All necessary namespaces are used, and a new class Is made to control the use of the stack

I have declared and created the stack so that it only contains data of the ActionType enum that was below.

Debugging the contents of a stack is very different to a normal variable. It would normally just put all of the contents together, which can be hard to read . I created this quick algorithm to debug the contents of the stack in a way that is easily readable, and will help for the testing to see if it works.

To add the fighting actions, I have added to the BluePlayerController script:

```
0 references
public class BluePlayerController : MonoBehaviour
{
    7 references
    [SerializeField]public Animator animator;
    11 references
    [SerializeField]public Stamina stamina;

    6 references
    [SerializeField]private BlueRecentMoves moveManager;
```

I have added a reference to the BlueRecentMoves class which I have made above so that I can access the stack and push items onto it

```
void Update()
{
    // for punching
    if (Input.GetKeyDown(KeyCode.F) && stamina.currentEnergy > 0)
    {
        animator.SetTrigger("bluePunch");
        stamina.PunchConsumeEnergy(5);
        moveManager.recentMoves.Push(BlueRecentMoves.ActionType.Punch);
        moveManager.DebugRecentMoves();
```

When the player uses the punch ability, It will be pushing the "Punch" action onto the stack, and will debug the current state of the stack

```
// for blocking
if (Input.GetKeyDown(KeyCode.W) && stamina.currentEnergy > 0)
{
    animator.SetBool("isBlueBlocking", true);
    stamina.drainEnergy(stamina.blockingDrainRate);
    moveManager.recentMoves.Push(BlueRecentMoves.ActionType.Block);
    moveManager.DebugRecentMoves();
}
```

When the player uses the blocking ability, the blocking action will be pushed onto the top of the stack, and the current state of the stack is debugged

```
// for blocking
if (Input.GetKeyDown(KeyCode.W) && stamina.currentEnergy > 0)
{
    animator.SetBool("isBlueBlocking", true);
    stamina.drainEnergy(stamina.blockingDrainRate);
    moveManager.recentMoves.Push(BlueRecentMoves.ActionType.Block);
    moveManager.DebugRecentMoves();
}
```

When the player uses the crouching ability, the crouching action will be pushed onto the top of the stack, and the current state of the stack is debugged

For the movement, since there is no animation and it is a continuous function, I had to add some extra if statements in the BlueBoxerMovement script to make sure that whenever a movement is done, left or right, the relevant movement action is only pushed onto the stack once, rather than It being pushed onto the stack continuously as the movement button is held.

```
public class blueBoxerMovement : MonoBehaviour
{
    3 references
    [SerializeField] private Rigidbody2D rigidBody;
    7 references
    private float horizontalBlue;
    1 reference
    [SerializeField] public float speed = 5f;

    1 reference
    public GameObject rightLimitGameObject;
    1 reference
    public GameObject leftLimitGameObject;

    2 references
    private Vector3 rightLimit;
    2 references
    private Vector3 leftLimit;
    4 references
    [SerializeField]private BlueRecentMoves moveManager;
```

I have made a reference to the BlueRecentMoves class in the blueBoxerMovement so that the stack can be modified.

```
void updateStack()
{
    if (Input.GetKeyDown(KeyCode.A))
    {
        moveManager.recentMoves.Push(BlueRecentMoves.ActionType.MoveLeft);
        moveManager.DebugRecentMoves();
    }
    else if (Input.GetKeyDown(KeyCode.D))
    {
        moveManager.recentMoves.Push(BlueRecentMoves.ActionType.MoveRight);
        moveManager.DebugRecentMoves();
    }
}
```

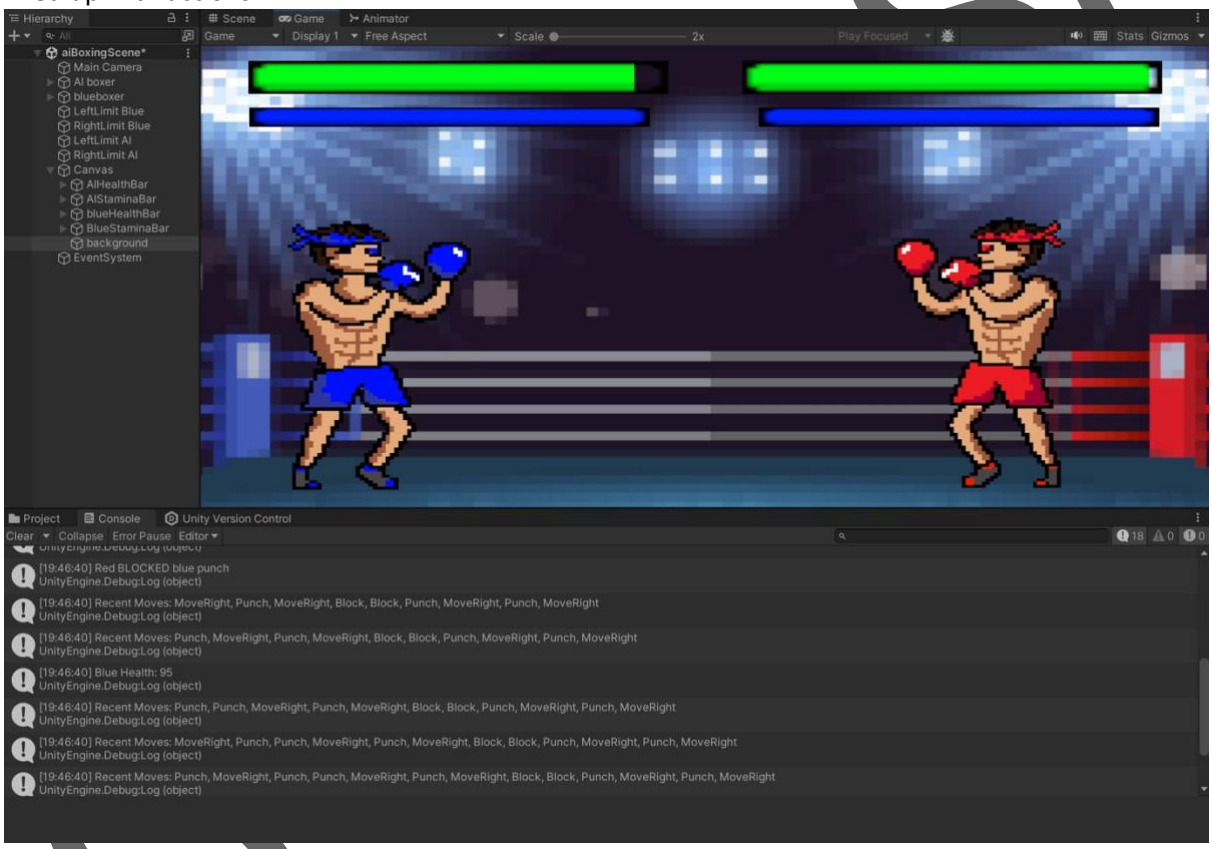
This is a function I have made so that the when the movement button is pressed, the movement in the relevant direction action is pushed onto the stack. I can do it in this way as I have already tested the movement logic at the start, and know it works so I don't need to worry about an issue arising where the action is pushed onto the stack, but the action is not actually completed.

Key variables

This section will be added to the end of the Level 2 AI creation, when it is fully finished.

Testing

I have not explicitly included this as a separate part of the iterative testing, but it is part of the full level 2 AI section as it is a needed feature for this section to work. As shown above, I added a debug algorithm to show the status of the stack after every time something is pushed onto the stack. Below is evidence of the stack being filled up with actions.



As seen above, the program debugs the contents in the stack containing all of the actions that the player has just done, with the newest actions being added to the front. This is a screenshot of the blue player against the level 1 AI, which is why the health had also decreased.

Review

Now that the recent moves are being recorded and put into a stack, I can now work on the AI accessing the data in the stack, and see if it meets certain conditions, in which a new probability for the weightings of each action will be calculated. This will be the change that the level 2 AI will bring.

AI adaptations

All of the sections here, have been added to the first level of the AI script. However since I want the first level AI and the second level AI to be part of separate I first started by making a function which works out the new weight for each action when a certain condition is met.

```
public float CalculateNewWeight(float priorWeight, float likelihood, float evidence)
{
    priorProbability = priorWeight / 10.5f;
    // calculate probability using Bayes' theorem
    float updatedProbability = (priorProbability * likelihood) / evidence;
    // calculate new weight by multiplying probability by total weight
    float updatedWeighting = updatedProbability * 10.5f;
    return updatedWeighting;
}
```

This will take the prior weighting, convert it into a probability, then use the Bayes' theorem on it. This will calculate the desirable weighting for that action.

Next, I will be making multiple conditions, such as reacting to health, stamina, recent moves and even the distance between the two players. These conditions will trigger the use of the CalculateNewWeight() function which will be used for each action, with suitable likelihood and evidence values so that the new weight of the action are matching to the situation.

CONDITION	JUSTIFICATION	LIKELIHOOD	EVIDENCE

PLAYER HEALTH < 25	When the players health is this low, the AI should become way more aggressive, and should try to finish the fight off earlier.	Punch – 0.6 Block – 0.25 Crouch – 0.08 Move left – 0.2 Move right – 0.13	The evidence is the probability of any action happening without any other conditions. I did some sampling of a couple games I played (in the 1v1 mode), and I decided upon some generalised evidence values for each action occurring : Punch = 0.4 Block = 0.2 Crouch = 0.1 Move left = 0.15 Move right = 0.15 These values all add up to one as they are the only actions that can actually occur, meaning the combined probability of all of them occurring is 1 These values will be consistent throughout all of the calculations for that specific action.
AI HEALTH < 25	When the AI's health has gotten this low, they should be playing more defensively, but not too defensively where they cannot fight back at all. This means that the calculations should lead to a higher weighting of blocking and crouching, without reducing punching and moving left (into the player) weighting by too much	Punch – 0.4 (unchanged) Block – 0.7 Crouch – 0.4 Move left – 0.25 Move right -0.25	
PLAYER STAMINA < 30	When the players stamina is in the red section (on the stamina bar), their stamina is below 30. I want the AI to be able to go in and induce actions which will cause the players stamina to reduce more. This means going in and blocking which will induce more punches and punching them so that they have to block more often.	Punch – 0.6 Block – 0.9 Crouch – 0.2 Move left – 0.55 Move right – 0.15	
PLAYER STAMINA = 0	We want the AI to go on full attack mode, as the player will have used up all stamina and will have to wait for it to recharge, meaning they are unable to protect themselves. This will add lots of challenge to make sure the player thinks strategically about	Punch – 1.0 Block – 0.25 Crouch – 0.15 Move left – 0.75 Move right – 0.075	
PLAYER HEALTH LOW / DISTANCE IS TOO HIGH.	When the players health is low and they are really far away, it is probably because they are trying to play more defensively and to hide away from the attack while the. So I want the AI to go in and put more pressure by moving left, punching and blocking/crouching.	Punch – 0.6 Block – 0.3 Crouch – 0.15 Move left – 0.5 Move right – 0.25	
REPEATED MOVES SUGGESTING AGGRESSIVE PLAY STYLE	When they are punching a lot, we need to get a bit more defensive and wait until stamina is low condition has been met.	Punch – 0.4 Block – 0.6 Crouch – 0.3 Move left – 0.15 Move right – 0.25	
REPEATED MOVES SUGGESTING DEFENSIVE PLAYSTYLE	When they are blocking/crouching a lot, we need to look for the	Punch – 0.6 Block – 0.35 Crouch – 0.15 Move left – 0.45 Move right – 0.15	

Player health / AI health < 25

```
0 references
public void conditions()
{
    if (bluehealth.currentHealth < 25f){
        punchWeight = CalculateNewWeight(punchWeight, 0.6f, punchEvidence);
        blockWeight = CalculateNewWeight(blockWeight, 0.25f, blockEvidence);
        crouchWeight = CalculateNewWeight(crouchWeight, 0.08f, crouchEvidence);
        moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.2f, moveLeftEvidence);
        moveRightWeight = CalculateNewWeight(moveRightWeight, 0.13f, moveRightEvidence);

    }
    if (AiHealth.currentHealth < 25f){
        punchWeight = CalculateNewWeight(punchWeight, 0.4f, punchEvidence);
        blockWeight = CalculateNewWeight(blockWeight, 0.7f, blockEvidence);
        crouchWeight = CalculateNewWeight(crouchWeight, 0.4f, crouchEvidence);
        moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.25f, moveLeftEvidence);
        moveRightWeight = CalculateNewWeight(moveRightWeight, 0.25f, moveRightEvidence);
    }
```

These are conditions that have been set up.

When the players health is low, the AI changes the weightings of each action so that it is more offensive. When the AI's health is low, it changes the weightings so that it is more defensive. This is done by assigning a suitable evidence and likelihood that I explained for each condition above.

Player stamina < 30 and = 0

```
if (bluecontroller.stamina.currentEnergy < 30f){
    punchWeight = CalculateNewWeight(punchWeight, 0.6f, punchEvidence);
    blockWeight = CalculateNewWeight(blockWeight, 0.9f, blockEvidence);
    crouchWeight = CalculateNewWeight(crouchWeight, 0.2f, crouchEvidence);
    moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.55f, moveLeftEvidence);
    moveRightWeight = CalculateNewWeight(moveRightWeight, 0.15f, moveRightEvidence);
}
if (bluecontroller.stamina.currentEnergy <= 0f){
    punchWeight = CalculateNewWeight(punchWeight, 0.7f, punchEvidence);
    blockWeight = CalculateNewWeight(blockWeight, 0.25f, blockEvidence);
    crouchWeight = CalculateNewWeight(crouchWeight, 0.15f, crouchEvidence);
    moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.75f, moveLeftEvidence);
    moveRightWeight = CalculateNewWeight(moveRightWeight, 0.075f, moveRightEvidence);
}
```

When the players stamina is low, the weightings should change so the AI will become slightly more offensive, which will force the player to use more actions, and there for make them run out of stamina all together.

When the stamina of the player has ran out, then then the AI should go into full attack mode. Using the observed evidence and likelihoods, Bayes' theorem should increase the weightings of the actions such as punching and moving left.

Distance conditions

```
1 reference
public Transform AI;
1 reference
public Transform BluePlayer;
3 references
public float distance;
```

```
0 references
void Update()
{
    actionTimer += Time.deltaTime;

    if (Time.time >= nextActionTime)
    {
        PerformAction(currentAction);
        ChooseNextAction();
    }
    distance = Mathf.Abs(AI.position.x - BluePlayer.position.x);
}
```

To create conditions which include the distance between the characters, I need to first calculate the distance between the players. To do this, I have imported the transform component of each of the players (which allows me to access their x coordinates).

In the update function, the distance is found out by using the absolute value of the subtraction of the two x values. This means that the correct distance will be found out every time.

```
//if distance is too great and enemy health is low
if (bluehealth.currentHealth <= 20f && distance >= 11f){
    punchWeight = CalculateNewWeight(punchWeight, 0.6f, punchEvidence);
    blockWeight = CalculateNewWeight(blockWeight, 0.3f, blockEvidence);
    crouchWeight = CalculateNewWeight(crouchWeight, 0.15f, crouchEvidence);
    moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.5f, moveLeftEvidence);
    moveRightWeight = CalculateNewWeight(moveRightWeight, 0.25f, moveRightEvidence);
}
//if distance is too low and AI health is low
if (AiHealth.currentHealth <= 20f && distance <= 5f){
    punchWeight = CalculateNewWeight(punchWeight, 0.3f, punchEvidence);
    blockWeight = CalculateNewWeight(blockWeight, 0.9f, blockEvidence);
    crouchWeight = CalculateNewWeight(crouchWeight, 0.5f, crouchEvidence);
    moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.15f, moveLeftEvidence);
    moveRightWeight = CalculateNewWeight(moveRightWeight, 0.55f, moveRightEvidence);
}
```

Now that the distance is being worked out, I can use it in two new conditions:

"If the distance between the characters is too big, and the enemies health is low" It is mostly likely that the enemy is trying to create space so that they can decide how to play the rest of the game, so it would be best for the AI to go in an put pressure on them.

"If the distance is low, and the AI's health is low" The AI should be backing off, and play more defensively, but not too defensive, otherwise there would be no chance of winning.

Counting moves in "recentMoves" stack

```
0 references
public int GetMoveCount(ActionType action)
{
    int count = 0;

    foreach (ActionType move in recentMoves)
    {
        if (move == action)
        {
            count++;
        }
    }

    return count;
}
```

This iterates through each data item in the recentMoves stack, and checks whether the move being checked is the same as the move we are trying to count, if it is, then increase the count, and return the count value as an integer

This is the method that I have added to the "BlueRecentMoves" class so that the moves in the stack (that I specify) can be counted as the game progresses. This will allow me to create new conditions including the count of certain moves that the blue player has made.

```
public class BlueRecentMoves : MonoBehaviour
{
    2 references
    public Stack<ActionType> recentMoves = new Stack<ActionType>();

    10 references | 1 reference | 1 reference | 1 reference | 1 reference | 1 reference
    public enum ActionType { Punch, Block, Crouch, MoveLeft, MoveRight }
    1 reference
    public int punchCount = 0;
    1 reference
    public int blockCount = 0;
    1 reference
    public int crouchCount = 0;
    1 reference
    public int moveLeftCount = 0;
    1 reference
    public int moveRightCount = 0;


    0 references
    void Update()
    {
        punchCount = GetMoveCount(ActionType.Punch);
        blockCount = GetMoveCount(ActionType.Block);
        crouchCount = GetMoveCount(ActionType.Crouch);
        moveLeftCount = GetMoveCount(ActionType.MoveLeft);
        moveRightCount = GetMoveCount(ActionType.MoveRight);
    }
```

A variable to hold the count of each object is then declared with an initial value of 0

The function is then used for each of the action counts in the update function so that it is constantly updated as the game goes along.

```
//recent moves suggesting an agressive playstyle
if (recentMoves.punchCount >= 20 && recentMoves.moveRightCount >= 50){
    punchWeight = CalculateNewWeight(punchWeight, 0.4f, punchEvidence);
    blockWeight = CalculateNewWeight(blockWeight, 0.6f, blockEvidence);
    crouchWeight = CalculateNewWeight(crouchWeight, 0.3f, crouchEvidence);
    moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.15f, moveLeftEvidence);
    moveRightWeight = CalculateNewWeight(moveRightWeight, 0.25f, moveRightEvidence);
}
//recent moves suggesting a defensive playstyle
if((recentMoves.blockCount >= 30 || recentMoves.crouchCount >= 20) && recentMoves.moveLeftCount >= 50){
    punchWeight = CalculateNewWeight(punchWeight, 0.6f, punchEvidence);
    blockWeight = CalculateNewWeight(blockWeight, 0.35f, blockEvidence);
    crouchWeight = CalculateNewWeight(crouchWeight, 0.15f, crouchEvidence);
    moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.45f, moveLeftEvidence);
    moveRightWeight = CalculateNewWeight(moveRightWeight, 0.15f, moveRightEvidence);
}
```

If the player is using punch, and move right (into the AI player) a lot, then the AI should play a bit more defensive so that it can preserve health. The new weightings will be calculated with suitable likelihoods

If the player seems to block or crouch a lot, and having moved away from the AI player a lot, then the AI should take advantage of this and go in and put more pressure on them.

Explanation

Now the Level 2 AI contains constantly changing weightings for each of the actions, which are calculated using Bayes' theorem. Suitable evidence and likelihood values were used for each time the new weightings were calculated. I have made some specific conditions which will alter the weightings of each of the actions, if the condition is met. This is better than making the AI do something specific once a condition is met, as this would end up being predictable. Instead using the weighting system alongside Bayes' theorem, it maintains a random nature, with certain biases.

Key variables

VARIABLES	TYPE	ROLE
DISTANCE	float	This is the horizontal distance between the two players. It is used in some of the conditions for the AI to respond to.
PUNCH WEIGHT	float	Holds the value of the weight of the action. It will change a lot during the progress of the game in this level of the AI
BLOCK WEIGHT	float	
CROUCH WEIGHT	float	
MOVE LEFT WEIGHT	float	
MOVE RIGHT WEIGHT	float	
PUNCH EVIDENCE	float	

BLOCK EVIDENCE	float	Holds the value of the evidence for that action. It is used as part of the calculations of Bayes' theorem. The evidence values are stated above in the "conditions" table.
CROUCH EVIDENCE	float	
MOVE LEFT EVIDENCE	float	
MOVE RIGHT EVIDENCE	float	
PUNCH COUNT	int	Holds the count of that action. It is incremented as the game goes along and is found by iterating through the stack which holds the actions
BLOCK COUNT	int	
CROUCH COUNT	int	
MOVE LEFT COUNT	int	
MOVE RIGHT COUNT	int	

Testing

SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL	SOLUTION
LEVEL 2 AI	Enhanced functionality and decision making	It should react to certain conditions optimising which move to pick the best, while making some "human-like" mistakes, as the best play should not always be picked	Since the "Conditions()" function is called in the update() section, the conditions are being checked every single frame, meaning that if a condition is met, the weighting will increase every frame, until it is infinitely large.	FAIL	The weight increasing infinitely while the condition is true is because it is called in the update function. However I still want it to be called often so that the weightings can continue to change during the game, so I will introduce a delay between calling the function.

Solution

To solve the issue, I have created a time delay between each of the calls of “conditions()”in the update function

```
3 references
private float nextConditionCheckTime;

0 references
void Start()
{
    ChooseNextAction();
    punchWeight = 5f;
    blockWeight = 1f;
    crouchWeight = 1f;
    moveLeftWeight = 1.75f;
    moveRightWeight = 1.75f;

    nextConditionCheckTime = Time.time + 5f;
}

0 references
void Update()
{
    actionTimer += Time.deltaTime;

    if (Time.time >= nextActionTime)
    {
        PerformAction(currentAction);
        ChooseNextAction();
    }
    distance = Mathf.Abs(AI.position.x - BluePlayer.position.x);

    if (Time.time >= nextConditionCheckTime)
    {
        conditions();
        nextConditionCheckTime = Time.time + 5f;
    }
}
```

I have declared the nextConditionCheckTime as a float variable so that it can be used in the class

In the start function, I have assigned it an initial value of 5 seconds after the start. This is because Time.time gets the value of time at the time it is called.

In the update function, instead of having conditions() called by itself, If the time has reached the next condition check time, then conditions is called, and the new next condition check time is calculated. This works in a loop basically iterating every 5 seconds.

Test 2

SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL	SOLUTION
LEVEL 2 AI	Enhanced functionality and decision making	It should react to certain conditions optimising which move to pick the best, while making some “human-like” mistakes, as the best play should not always be picked	Some of the likelihoods for the actions in the conditions are too large, meaning they will only do that while the condition is true. For example, when the AI’s health was below 25, the only action it was doing was the block. The problem was mainly occurring with the “health” conditions	FAIL	I need to decide new likelihood values for the actions which change the weighting quite dramatically, so that the changes are less dramatic

```
if (bluehealth.currentHealth < 25f){
    punchWeight = CalculateNewWeight(punchWeight, 0.5f, punchEvidence);
    blockWeight = CalculateNewWeight(blockWeight, 0.25f, blockEvidence);
    crouchWeight = CalculateNewWeight(crouchWeight, 0.08f, crouchEvidence);
    moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.2f, moveLeftEvidence);
    moveRightWeight = CalculateNewWeight(moveRightWeight, 0.13f, moveRightEvidence);

}
if (AiHealth.currentHealth < 25f){
    punchWeight = CalculateNewWeight(punchWeight, 0.4f, punchEvidence);
    blockWeight = CalculateNewWeight(blockWeight, 0.3f, blockEvidence);
    crouchWeight = CalculateNewWeight(crouchWeight, 0.3f, crouchEvidence);
    moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.25f, moveLeftEvidence);
    moveRightWeight = CalculateNewWeight(moveRightWeight, 0.4f, moveRightEvidence);
}
```

I changed the values of some of the likelihoods so that they are less dramatic moves. This will ensure that the weighting does not change too drastically, and will not only just do one action when the condition is true.

Test 3

SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL
LEVEL 2 AI	Enhanced functionality and decision making	It should react to certain conditions optimising which move to pick the best, while making some "human-like" mistakes, as the best play should not always be picked	The AI reacts to the certain conditions, and acts accordingly, while still keeping its random nature via the weighting system. Some conditions require drastic change in playstyle, where some are more subtle, and both changes are noticeable.	Pass

Evidence

Just like the level 1 AI, there is not any necessary validation for this section, but there is testing to be done to make sure that the level 2 AI is working as it should.

The evidence for the testing of the level 2 AI is in the video under the "Video of Game" section at the end of the development.

The timestamp in the video is: **1:24**

Review

I have now made an AI system, which is a development of the first level AI. It takes in certain conditions, works out new probabilities for each action, makes a new weighting, checks for the conditions in fixed intervals and acts accordingly to the conditions. This makes the system similar to a finite state machine model, due to the AI having certain states in which it could be in: not reacting to any conditions, reacting to

conditions resulting in a more offensive playstyle and reacting to conditions resulting in a more defensive playstyle. However this model could have been made in a more complex way by introducing a neural network, and the model I have made could be the building steps towards that type of model. The conditions could be a means of gathering information and large amounts of data of how the AI should act. The reason I couldn't make this is because of that fact exactly, it takes lots of training with data, which I don't have enough time to gather.

Now that the whole game is finished, I just need to add a few Graphical user interface features such as the main menu (+menu pages), pause menu and the game over screen. Once that is done the game is fully developed and is playable.

Graphical User Interface

This section will include the UI features, such as the main menu, pause menu and the game over screen. These are necessary components of the game as they allow the player to navigate the game as they wish. The main criterion for this section is that they the menus are easy to use, and are not confusing. One optional criteria would be to make it so that they are matching to the theme of the game, however this would require lots of graphical design which I don't think I could easily find on the internet for use, and I would have to design these, so for that reason I am deciding not to implement these.

There are 4 scenes within the game; main menu, boxing scene (1v1), ai boxing scene (level 1) and ai boxing scene (level 2). In unity I can use the scene manager module in my scripts so that I can switch between them when certain conditions are met.

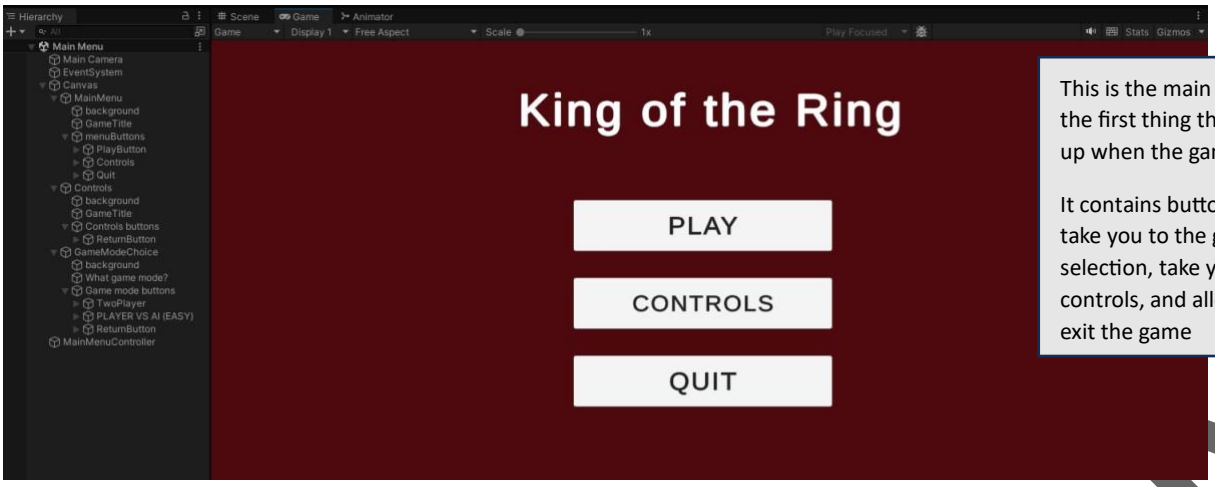
Since all of the rest of the game is made, and the testing/validation for the features in this section is quite easy to do, I have decided to add the tests and proof of evidence of the buttons etc working in the video under the section "Video of Game". The relevant timestamps will be mentioned in each section.

The Main Menu

The main menu will have its own scene and will have multiple pages which will all have their own purpose. In the Usability features of the project (As part of the design section), I had designed all of the different pages already so that they are easily navigable , and now I just need to recreate them In Unity, adding functionality to all of the buttons.

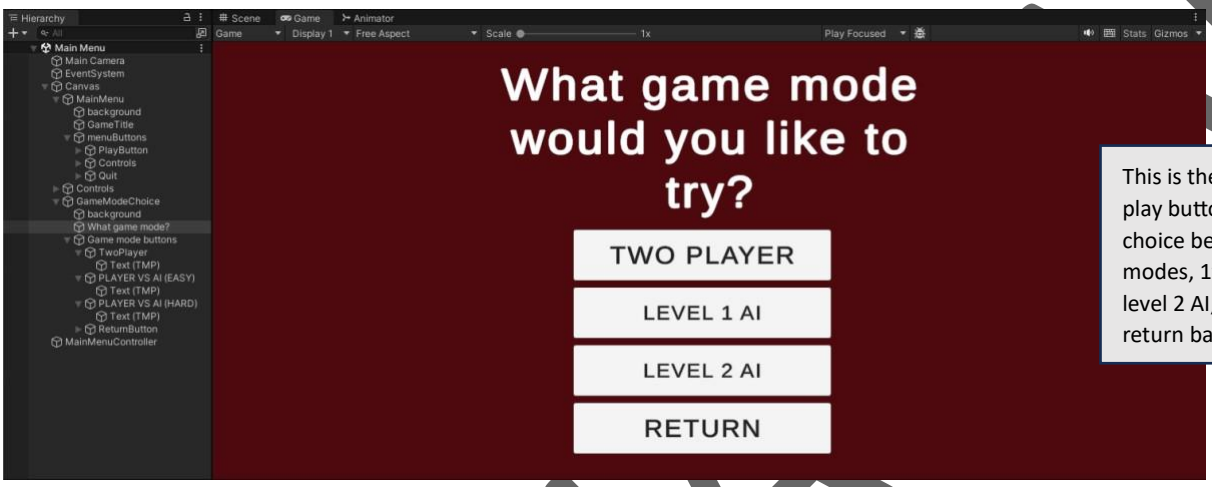
For the buttons, I have used TextMeshPro buttons which are pre-installed features in unity which allow me to create a button easily, and allow me to attach a function to the button in the script.

As I mentioned above, there are 4 scenes, meaning all of the different pages in the menu are all in one scene. This has been done to speed up navigation through the menu as loading a new scene can take time. Instead for each of the buttons I have created their own function which will make the opacity of all of the contents on that page 100%. This will mean that this page will show up and block out the others.

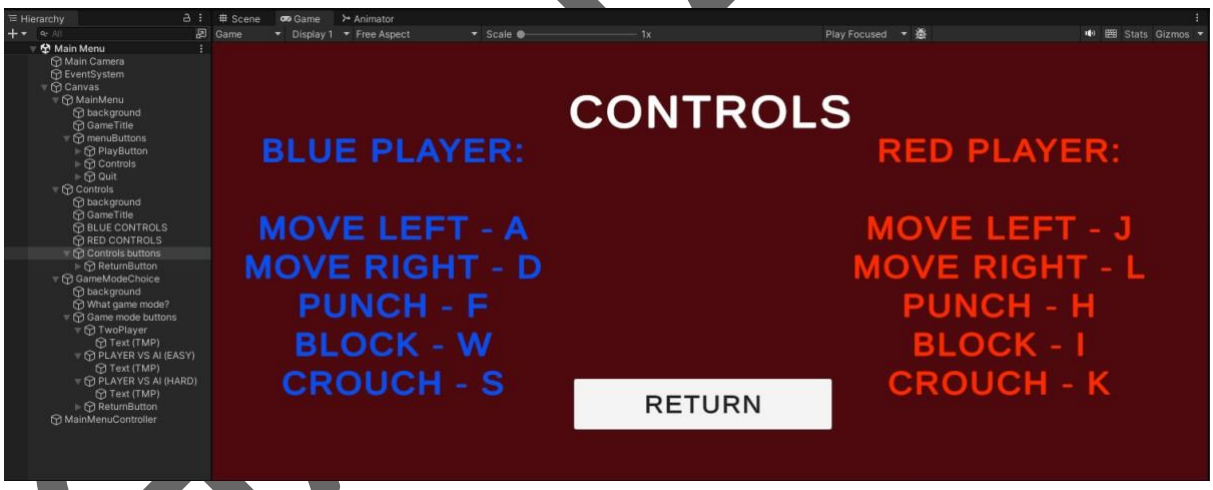


This is the main menu screen, the first thing that will show up when the game begins.

It contains buttons which take you to the game mode selection, take you to see the controls, and allow you to exit the game



This is the follow through from the play button, and gives the users a choice between the three game modes, 1v1, the level 1 AI, the level 2 AI, as well as a choice to return back to the main menu.



This is the control page, and tells the user how to use the game properly. The only button is the return button, which will take you back to the main menu page.

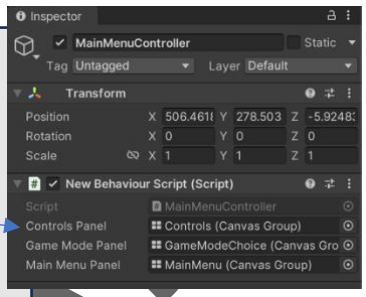
As you can see on the side, main menu has been set up fully with all of the necessary game objects. For all of the buttons, and transitions to work perfectly I needed to create a script, to handle all of this.

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

0 references
public class MainMenuController : MonoBehaviour
{
    6 references
    public CanvasGroup controlsPanel;
    6 references
    public CanvasGroup gameModePanel;
    2 references
    public CanvasGroup mainMenuPanel;
```

All of the necessary namespaces are used, as well as the "UnityEngine.SceneManagement" namespace which will allow me to control the scenes which are in the game file, which are the ones that I mentioned above.

The CanvasGroup component comes from the "UnityEngine" namespace and allows me to refer to a certain group of game objects within a section in the canvas

I have made a refence to each of the different pages in the main menu that I have created, and I filled the references in Unity as seen to the right



The "alpha" component of each of the canvas groups refers to its opacity , 0 being invisible and 1 being fully visible.

```
public void Start()
{
    mainMenuPanel.alpha = 1;
    mainMenuPanel.blocksRaycasts = true;
    controlsPanel.alpha = 0;
    controlsPanel.blocksRaycasts = false;
    gameModePanel.alpha = 0;
    gameModePanel.blocksRaycasts = false;
}
```

In the start function, I have set the opacity of each of the scenes except from the main menu panel to 0, meaning only the main menu page, which was shown above, is displayed when the game is started

"blocksRaycasts" is a component which will make the game objects on that canvas group interactable. blocksRaycasts = true means that the buttons will react to clicks, but when set to false, the buttons will not react to any input. This is needed so that only the buttons which are shown on the screen at that time are interactable.

```
public void gameModeChoice()// load game mode choice menu
{
    gameModePanel.alpha = 1;
    gameModePanel.blocksRaycasts = true;
}
0 references
public void gameModeReturn() // Return to the main menu from game mode choie
{
    gameModePanel.alpha = 0;
    gameModePanel.blocksRaycasts = false;
}
0 references
public void Controls()// Load the controls menu
{
    controlsPanel.alpha = 1;
    controlsPanel.blocksRaycasts = true;
}
0 references
public void controlsReturn() // return to main menu from the controls screen
{
    controlsPanel.alpha = 0;
    controlsPanel.blocksRaycasts = false;
}
```

These are all of the functions to be used for the buttons that change the page within the main menu.

gameModeChoice will make it so that the game mode choice screen is visible and allows interaction. gameModeReturn makes it so that the game mode choice screen disappears and the main menu panel shows again.

And the control screen functions are very similar to the game mode, with a function to go to the controls screen, and one to return back to the main menu

```
public void pvpGame()// load the
{
    SceneManager.LoadScene("boxin
}
0 references
public void pveGame()// load the
{
    SceneManager.LoadScene("aiBox
}
0 references
public void pveGame2()// load lev
{
    SceneManager.LoadScene("aiBox
}
0 references
public void QuitGame()//  Quit th
{
    Application.Quit();
}
```

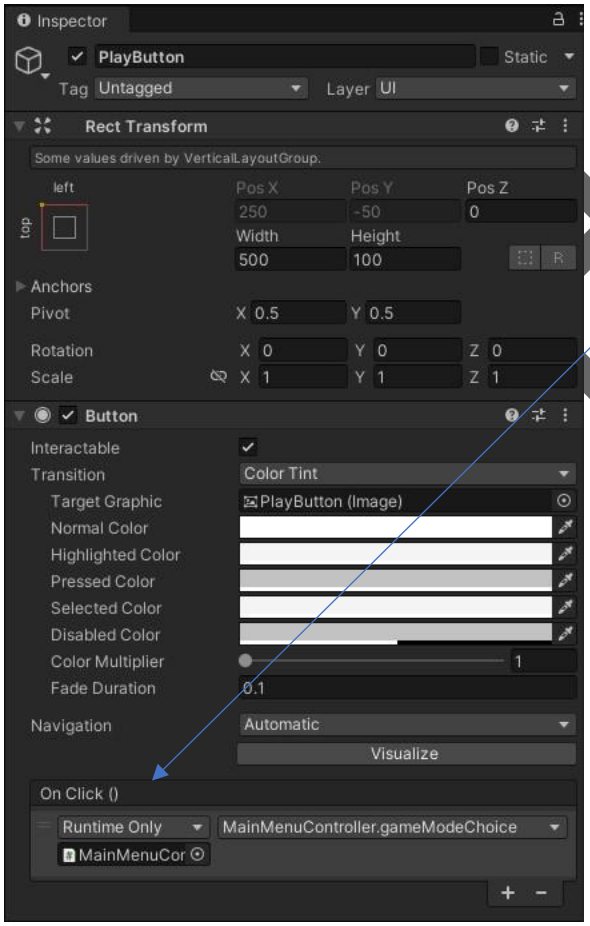
These are the functions that will change the actual scene of the game.

pvpGame will take the user to the two player (1v1) screen.

pveGame will take the user to the level 1 AI scene
pveGame2 will take the user to the level 2 AI scene

This is a function that will terminate the game, and is going to be used for the quit game button on the main menu panel.

Now that I have created all of the functions, I will assign them to the relevant buttons in the Unity editor. For example the play button on the main menu screen:



In the play button game object, it has a button component, which has an OnClick() function, in which I have set the action to the "MainMenuController.gameModeChoice" method which will take the player to the game mode selection screen as shown above in the code.

I have done this for all of the buttons on all of the pages, so that they do the function they are supposed to do.

Explanation

Now the main menu looks, and works as intended. All of the buttons take you to the correct page and they work pretty fast. It made sense to create the menu at the end of the development, as it depends on most of the scenes being made already, so that I can make buttons which link to those scenes. I have also made it so that there is minimal loading during the transitions between the pages in the main menu, by creating a system which actually doesn't change scene, but instead works with the certain components (alpha and blocksRaycasts) so that the pages area always there, but not always visible. This is faster than if I were to create a new scene for each of the pages, and also saves space as each scene takes up a lot of storage.

Key variables

There is no variables being used in this script.

Testing

SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL
GUI	Main menu window	When the game is loaded, the program should display the menu in the correct size to fit the display.	Correct Menu is shown on start up	PASS
	Functioning buttons for all the menus	Buttons for "Start", "Controls", and "Exit" take you to correct page	All of the buttons within the main menu do what they state they will do	PARTIAL PASS (requires pause menu functionality)

Evidence

The evidence of the validation is in the "Video of Game" section. The timestamp for the main menu is **0:00** or **2:30**

Review

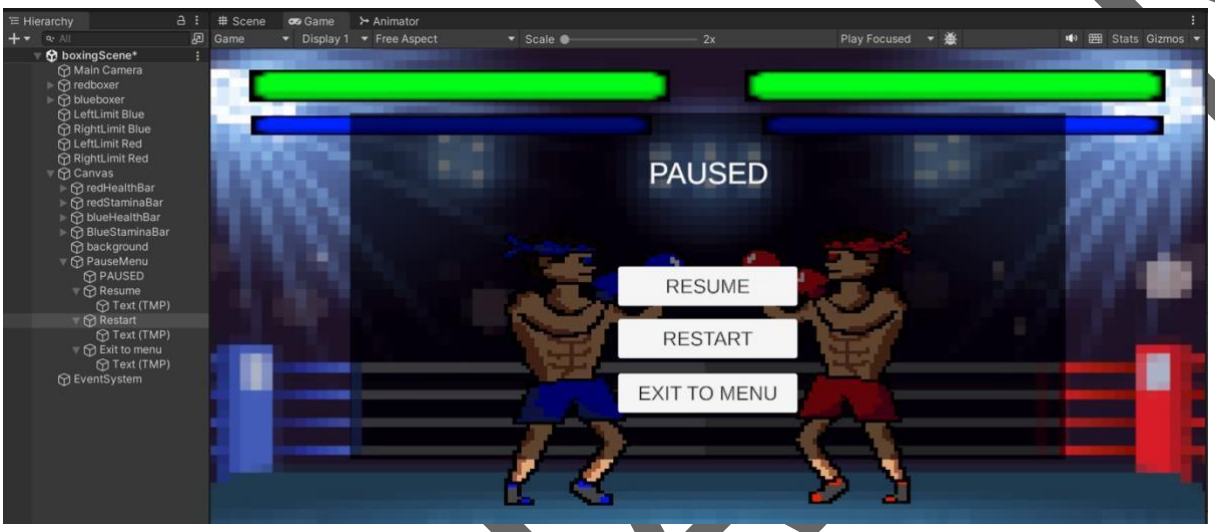
Now the main menu works as it should, I can work on the pause menu, which will be very similar to this stage but it will be a part of the three boxing scenes instead, and will not require different pages like the main menu. After that I will only need to make the game over screen, which will once again be quite similar, and I will be finished with the development of the project, and can move on to evaluation.

Pause menu

I was struggling to find a way to create a pause menu, as the way I had created the main menu wouldn't work as that was in a separate script, meaning that if it switched to a different scene, the progress of that specific game would end. I did research into ways that other developers had solved this, and found using the "Time.timeScale" syntax would allow me to completely pause the game and all features. With this I could stop the game from progressing while the pause menu is shown, using a Boolean value for "isPaused".

Since I have three game modes, I made a script which would work for all three modes without repetition of code.

The way the pause menu was created and designed was very similar to the main menu, however the way that it is navigated is much simpler, as each function will have a direct action, and will not require other "pages" like the main menu did. To create the pause menu, I created a panel UI game object in unity which is grouped under the canvas section in the hierarchy, in which I added a header to, along with three buttons, "Resume" "Restart" and "Exit to main menu".



Now I will add functionality so that the pause menu is opened when the "esc" key is pressed and so that the buttons lead to the expected place. For the AI game modes especially, the AI should stop doing any actions when game is paused.

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

0 references
public class PauseMenuController : MonoBehaviour
{
    3 references
    public GameObject PauseMenu;
    3 references
    public bool isPaused;
    2 references
    private string currentSceneName;
```

Using all of the necessary namespaces, including the Scene management module so that I can control transitions between different scenes.

I created a new class for the contents of this script.

I declared three variables. "PauseMenu" is to hold the game object assigned to the pause menu panel.

"isPaused" is a bool which is used as a flag to tell if the game is paused at that time.

"currentSceneName" is declared so that the scene that is currently being played.

```
0 references
void Start()
{
    PauseMenu.SetActive(false);
    currentSceneName = SceneManager.GetActiveScene().name;
}
1 reference
public void PauseGame()
{
    PauseMenu.SetActive(true);
    Time.timeScale = 0;
    isPaused = true;
}
1 reference
public void ResumeGame()
{
    PauseMenu.SetActive(false);
    Time.timeScale = 1;
    isPaused = false;
}
0 references
public void RestartGame()
{
    Time.timeScale = 1;
    SceneManager.LoadScene(currentSceneName);
}
0 references
public void ExitToMenu()
{
    SceneManager.LoadScene("Main Menu");
}
0 references
void Update()
{
    // if game is not paused, pause. If game is paused, resume.
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        if (isPaused)
        {
            ResumeGame();
        }
        else
        {
            PauseGame();
        }
    }
}
```

In the start function, I don't want the pause menu panel to be activated, so I set it to false.

I also want to retrieve the name of the current scene and assign it to currentSceneName so It can be used for the "restart" button

In the PauseGame function, the pause menu panel will be activated and will show on the screen, isPaused is set to true. Time.timeScale will also be set to 0, meaning that the game is stopped, and no time is passing in the project space (effectively pausing the game).

The resumeGame function does the opposite of the PauseGame function, setting the bool to false, deactivating the pause menu panel and setting the time to run at normal speed.

The restart game feature will cause a transition to the current scene, and in doing so the progress of the game will be reset as the scene will start again. At the same time it must set timescale back to 1, so that when the game starts again, it can be playable.

The exit to menu function will take the player to the main menu scene

In the update function, when the escape key is pressed and the game is not paused, the game will pause. If the game is already paused, then the game will resume.

IMPORTANT – As the pause menu requires a pause in the clock, I didn't know how to implement this as I am quite new with unity, and do not know all of the special syntax and features you can use. So I looked on YouTube for help within a tutorial and was able to find a video explaining the Time.timeScale feature. Although the restart button was fabricated myself, there is only so many ways to create the pause menu, so I used some of the code from the video in the development of this section:
https://www.youtube.com/watch?v=9dYDBomQpBQ&ab_channel=BMo

Explanation

The pause menu is a panel game object, which can be activated and deactivated at different times. This allows me to control when the pause menu shows up. The "esc" button on the keyboard will be used to bring about the pause menu, where the unity's in-built clock is stopped meaning the AI is stopped, and any code written in the "Update()" function throughout the scripts will not be reacted to.

Key variables

VARIABLES	TYPE	ROLE
PAUSE MENU	GameObject	This allows me to control the components of the pause menu UI object, like “setActive” etc
ISPAUSED	bool	This is the bool so that the program knows when the game is paused. This is useful in the if statement including the press of “esc”
CURRENT SCENE NAME	string	This allows me to retrieve the name of the current scene. It is used for the restart button and allows the script to be used multiple times for the three different game mode scenes.

Testing

SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL
GUI	Pause menu	Once “esc” is pressed, a new screen should show up with different options. The game should also pause not allowing any movement, actions etc to happen.	When “esc” is pressed in game, the pause menu shows with the three intended options, “Resume” “Restart” and “Exit to main menu”. While in the pause menu “esc” can also be used to return. The unity time manager pauses, not allowing any inputs/actions to occur	PASS
	Functioning buttons for all the menus	All the buttons on both menus should work as intended	All of the buttons within the pause menu do what they state they will do	PASS (main menu and pause menu functions)

Evidence

Evidence of the validation and testing of this section is in the video under the “Video of Game” section. The timestamp is **0:15** – **0:35**

Review

Now I have a functioning pause menu, the game is even more usable, as the users can navigate the game easier. The only thing left to do Is add a game over screen, which will be made in a very similar way to the pause menu. However, It will be designed in a different way and will only have a single button to go back to the main menu.

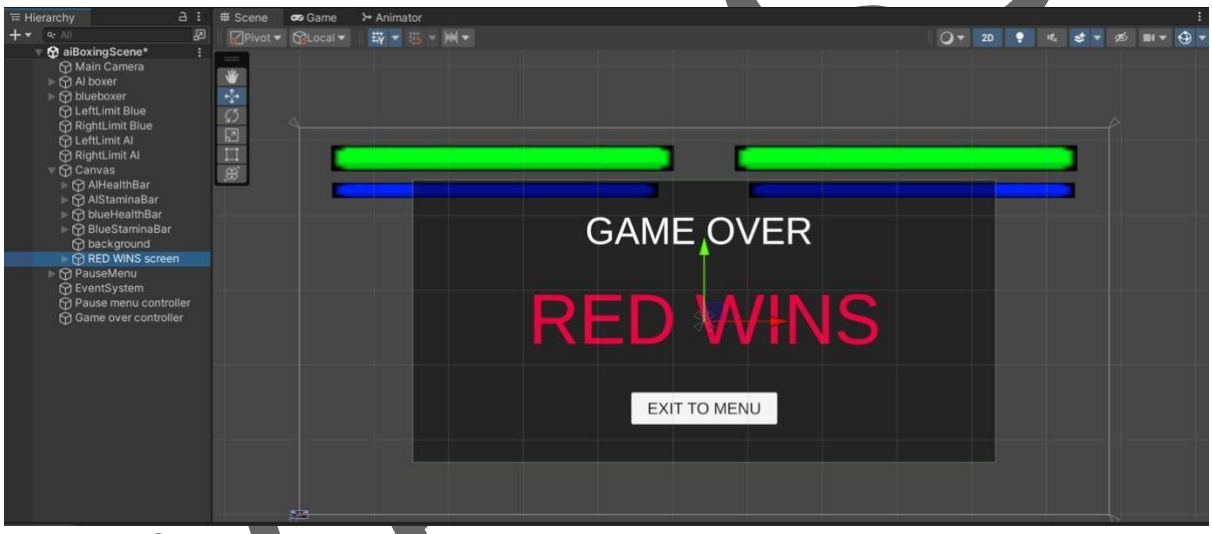
Game over screen

This was set up in a similar way to the pause menu, where a panel was made and designed to clearly state the winner, as well as having an exit to main menu button so that the user(s) can decide whether they want to play again or quit the game. As there is two possibilities for who wins, I will have to make two panels which states

the correct boxer who won. The activation of this page will be in the “KO()” function that was created when making the health handling feature earlier in development.

One thing that can be reused, is the ExitToMainMenu function from the pause menu section, as it will do the same thing for the game over screens, and there would be no reason to repeat the code again. I will assign this same function for the button.

These are the simple designs I made:



The script to manage the screen:

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

0 references
public class GameOverScreen : MonoBehaviour
{
    2 references
    public GameObject blueGameOverScreen;
    2 references
    public GameObject redGameOverScreen;

    0 references
    void Start()
    {
        blueGameOverScreen.SetActive(false);
        redGameOverScreen.SetActive(false);
    }

    0 references
    public void ShowBlueGameOverScreen()
    {
        blueGameOverScreen.SetActive(true);
    }
    0 references
    public void ShowRedGameOverScreen()
    {
        redGameOverScreen.SetActive(true);
    }
}
```

I made a new class for the behaviours of the game over screen. I started by importing the game objects which hold the panels which I created above.

When the game begins, I want the panels to be inactive as the end of the game has not occurred yet.

This is a method which will be used to activate the game over screen (when the blue player wins). It will be used in the health handler for the red player/AI player (for the relevant game mode)

This is a method which will be used to activate the game over screen (when the red player wins). It will be used in the health handler for the blue player

In the health handler scripts, when the player has reached 0 health, the KO function is done, which destroys the game object, signifying that the player has lost all its health and has lost. In this function I have added the methods that I have made above, so that the game over screen will show up.

To do this I made reference to the GameOverScreen class, in the relevant scripts, so that I could access its public methods

```
1 reference
public void KO()
{
    Destroy(gameObject);
    Debug.Log(" BLUE KO ");
    gameOverScreen.ShowRedGameOverScreen();
}
```

In the Blue health handler script, in the KO function, I have added the ShowRedGameOverScreen script, which indicates that the Red player has won. This is because this function will play when the blue player has no more health left

```
1 reference
public void KO()
{
    Destroy(gameObject);
    Debug.Log(" RED KO ");
    gameOverScreen.ShowBlueGameOverScreen();
}
```

In the red health handler script, in the KO function, I have added the ShowBlueGameOverScreen script, which indicates that the blue player has won. This is because this function will play when the blue player has no more health left

Explanation

Above were the steps taken to create a game over screen. It works very similar to the pause menu in the way it is set up, but its behaviours are different. When one of the players health reaches 0, the "KO()" function is ran (located in the health handler script) , the relevant game over screen is played. Key variables

VARIABLES	TYPE	ROLE
RED GAME OVER SCREEN	GameObject	Importing the game object holding the game over screen panel will allow me to control when it is viewed by setting it active and inactive
BLUE GAME OVER SCREEN	GameObject	

Testing

SUBROUTINE	WHAT IS BEING TESTED	EXPECTED RESULT	ACTUAL RESULT	PASS / FAIL
GUI	Game over screen	Should show who has won at a suitable time, and a return to menu button should be functional	When one of the players has been defeated, the game over screen for the opposite player shows up (showing they have won). There is also a functioning return to main menu button	PASS

Evidence

Evidence of the validation and testing of this section is in the video under the "Video of Game" section. The timestamp for the game over screen is **1:17**

Review

The game over screen is working as intended as shown in the testing above. Now, all of the sections that were possible to complete are done, therefore the development is finished. Now I need to evaluate the solution that I have made and compare it with the initial problem and specifications

Video of game

If it doesn't work:

https://youtu.be/2YsT-03uyyg

In the description of the video, it contains a few timestamps for the important parts of the game, but majority of the references to the video, will contain also contain a time stampe

Evaluation

Post development testing

The justification for why I am doing each of these tests is at the end of the design section.

WHAT IS BEING TESTED	EXPECTED RESULT	PASS/FAIL	EVIDENCE/COMMENTS
BOTH PLAYERS CAN SIMULTANEOUSLY MOVE AND DO ACTIONS	Both characters should be able to move, punch, block and crouch simultaneously, without any drop in performance. A drop in performance could either be indicated by frame rate drop and could be due to inefficient code design, or insufficient computing power.	PASS	0:08 Both players can move and do actions simultaneously meaning that the program can easily handle multiple inputs.
WHEN A PLAYER REACHES 0 HEALTH, GAME IS OVER	When one of the players is defeated, the game should end.	PASS	1:15 As seen from the health bar, of the red player, the health goes down until it is zero, and when it is zero, the game finishes. The game over screen is then displayed where it displays the winner, and gives the users option to go back to main menu.
BUTTONS ON BOTH MENUS SHOULD TAKE YOU TO CORRECT PLACE	The main menu will have three buttons and the pause menu will have three buttons. Overall, the 6 buttons will link 4 scenes (Boxing arena, pause menu, Start menu and controls page).	PASS	0:00 and 2:41 (main menu) / 0:15 pause menu All the buttons in all menus have the correct functions assigned with them so they do exactly what they say they do.
VALID/INVALID INPUT TESTS	When a valid input key is used, the relevant action should play. When an invalid input key is used, the game should not do anything in response to this.	PASS	Whole video The valid input tests were successfully shown throughout the iterative testing phase for all of the sections where input was required. The testing for the menus input was shown as every time I clicked, an effect was shown on the recording. However it is hard to prove invalid test input as they invalid inputs don't do anything, like expected
TEST THE SUITABILITY OF THE AI'S	Level 1 AI should be playing in a random style, while still not being too easy to beat. The level 2 AI should be harder to beat/ be able to last longer	PASS	1:53 The first level AI is not too hard to beat, as it should have been, this can be seen by the fact that I beat it relatively fast with a

	against a real opponent, and should react to different playstyles etc.		sufficient amount of health left. At the timestamp stated above, I have compared the level 2 AI to the first one, and I noticed that It was holding up a lot better, and it was staying on a similar health throughout. This shows the suitability of the developments on the level 1 AI.
LOAD TESTING	The most recourse intensive part of the game will probably be the level 2 AI, due to the collection of recent moves, and the amount of calculations happening all at once in a very short time.	PASS	Whole video As we can see throughout the whole video, there was not any time where the game had any freezing issues or crash, even during the level 2 AI gameplay. This means that the memory requirements of the game did not exceed a large amount.

Evaluation of solution

Evaluation of success criteria

CODE	SUCCESS CRITERIA	MEASURABLE	MET	EVIDENCE AND COMMENTS (VIDEO REFENRENCE)	FUTURE IMPROVEMENTS
GM1	Attacking mechanics	100%	FULLY MET	0:11 The attacking mechanic that is used in this game is the punch feature. As shown in the video, this feature clearly works as it is meant to. When the punching button is pressed, the correct animation is shown, the health goes down the right amount if it connects successfully and the stamina goes down accordingly	N/A
GM2	Blocking mechanics	100%	FULLY MET	1:25 (shown throughout the level 2 AI fight) The blocking/crouching features both nullify the damage that is taken and is shown in the iterative testing phase, and in the video during the fights.	N/A
GM3	Movement mechanics	100%	FULLY MET	0:11 Both the characters have right and left movement as they should, they cannot move through each other and they cannot move out of the bounds of the screen.	N/A
GM4	Health control	100%	FULLY MET	1:25 (shown throughout the level 2 AI fight) As punches successfully hit the opponent, their health decreases by the relevant amount. If the opponent is blocking or crouching, they will either lose less health, or will stay the same.	N/A
GM5	Energy control	100%	FULLY MET	1:25 (shown throughout the level 2 AI fight) As an action is used, the stamina of the player goes down by a set value. When a player blocks or crouches, their stamina decreases in a “draining” type, unlike the punch action. The stamina also	N/A

				regenerates at a steady rate, whereas the health does not.	
GM6	Special abilities	100%	NOT MET	N/A This feature was not implemented as it required a roster of different characters, all with their own special designs, animations etc. This would have added a lot of time to the completion of the project which I did not have.	I could make a roster of characters for the next version, each with their own designs, animations and special abilities. They could also have their own ultimate ability which could deal a lot of damage if it connects.
GM7	Level 1 Ai opponent	100%	FULLY MET	0:36 This AI does random moves in a weighted fashion as shown in the video. This means that the actions that are thrown aren't completely random, and are more biased towards certain actions which is required for a game like this.	I think a good adaptation of this would be getting the weightings to be perfect. The way to do this would be do to lots of testing and data collection to see what the distribution of action usage is for multiple different human users, and use that data to decide the weightings, instead of me deciding them on the spot.

GM8	Level 2 AI opponent	100%	PARTIALLY MET	1:25 This AI is a big step up from the level 1, as it uses a statistical model to work out the conditional probabilities of each action working, however the fact that it only really reacts to a certain set of conditions makes it less adaptable itself.	An improvement for a future model would be to make more conditions, and gather data based around the actions that are chosen using Bayes' theorem. With this data I could then try to build a neural network, as there is only 5 inputs for each character (each action), so it would not be extremely confusing to make. With the data gathered, I would be able to make this model way more adaptable and intelligent itself so it could make its own decisions.
GM9	Balanced experience	50%	FULLY MET	N/A The balanced experience would mainly be for the two player mode, and since it the two player mode gives both the characters equal opportunities to win, it therefore must be a balanced experience. It would only be unbalanced if there was ways to level up your characters through objectives or quests etc, which is not a feature so it is not a worry	I would possibly create an online mode, so that having a balanced experience could be measured better. I would still not add a "levelling" system where you can upgrade your attributes etc, as this would create a unfair game, unless I made a matchmaking system.
GD1	Sprite design	100%	FULLY MET	Throughout the video The sprites have designs for each action, and the sprites were made into animations that shows when an action was done	I would have more sprites for different characters, with unique designs instead of just a single boxer with different colours for their gloves
GD2	Background art	100%	PARTIALLY MET	Throughout the video There was background art for the main gameplay arena, however I did not have time to create/find anything for the menu background.	I could add some background design for the main menu, as this would contribute to the theme of the game better

GD3	Animated actions	100%	PARTIALLY MET	1:25 (shown throughout the level 2 AI fight) All of the actions (except from the left and right movement) had their own animations, which were played when the relevant key was pressed for that action. However due to the animations for moving left and right have not been created, I don't think I can pass this success criteria as fully passed	I would add some animations for the left and right movement which would act in the exact same way as the blocking and crouching mechanic, where the animation would be played for as long as the button is held down.
AD1	Background music	100%	NOT MET	N/A I could not find suitable music to use for the background, and did not find this to be a particularly necessary feature of the game as it has minimal effect on the game play itself	I could find some suitable music to play in the different scenes.
AD2	Sound effects	100%	NOT MET	N/A Once again, I found it hard to find certain sound effects that were suited for my game. Even though this was a bit more important the background music, I still believe it has a pretty small effect on the overall experience that you receive while playing the game.	I could find some sound effects for players getting hit and maybe some movement sounds.
UI1	Start menu	100%	FULLY MET	0:00 All of the different pages within the main menu works, and the buttons take you to the correct places. It is also very user friendly and simple, as it is easy to navigate	N/A
UI2	Pause menu	100%	FULLY MET	0:15 The pause menu is minimal as it should be, showing only the important features that it should have. The way to activate the pause menu (pressing "esc" key) works, and all of the relevant buttons on the pause menu take the user to the intended place.	I think the menu is fine as it is at the moment, but with some background art it would bring a better overall experience to playing the game.

UI3	Suitable HUD	100%	FULLY MET	Throughout video I believe that the HUD in game is suitable, as it shows the important information which is the health bar and stamina bar. The health bar takes the value of the current health variable assigned to both characters, and the stamina bar does the same for the current energy variable for each character. This helps to show the players accurate representation of both of the values, meaning they can understand the progress of the game while playing.	If I was to add special abilities and ultimate abilities, I would have to have a bar to show the ultimate ability charging, as they would not be able to use it continually as that would be too overpowered.

Evaluation of Usability features

USABILITY FEATURE	MET	EVIDENCE AND COMMENTS (VIDEO REFERENCE	FUTURE IMPROVEMENTS
SIMPLE UI	FULLY MET	0:00 – 0:10 / 0:15 This is to show the design of the main menu and of the pause menu. I have made them both so that they are very easy to navigate. This is because I have limited the actual functionality of both menus for the sake that they are usable for anyone.	N/A

CONTROLS PAGE IN MAIN MENU	FULLY MET	0:02 This is the controls page, and effectively tells the user how to play the game. Since I had to fit the controls for both players onto a single keyboard, I found it hard to use the "common" controls for games like this for both of the players. The blue player has the common "WASD" movement and action key binds, whereas the red player uses "IJKL" format which is the same structure of the other players controls, but it is just on the other side of the mirror.	If I could improve upon the controls, I would try and make the game compatible with games controllers, such as Playstation or Xbox controllers, as both players would have the same controls. It would also be useful for making certain combinations with the joystick etc.
RESTART BUTTON IN PAUSE MENU	FULLY MET	0:24 The restart button works as it should, by working out what the current scene is called in the script, and recalling the name of the scene so that the function related to the button can start that scene again. This effectively restarts the game, and it will take you to the correct game mode every time.	N/A
HEALTH AND STAMINA BARS AT THE TOP OF THE SCREEN	FULLY MET	1:25 (shown throughout the level 2 AI fight) Both the health and the stamina bar work as they should, representing the relevant variables in the correct way, and constantly updating as the changes are made.	A change I could have done was to design them in specific way so that it is very obvious which one is which, with symbols, or certain animations at different health/stamina.

Stakeholders review on usability features

USABILITY FEATURE	DAENIELS COMMENTS	JACKS COMMENTS	REFLECTION

SIMPLE UI	"I think that the main menu is simple, which is good because it means you can go to the right places easily, but I think with the simplicity came an unappealing design. I think the engagement with the game would be longer lasting if both menus had a better design."	"The pause menu seems fine, as you don't want that menu to be too confusing and you want it to be simple. However the main menu is too plain. It is the first thing that you see when entering the game and having a well designed menu that connects with the theme would be a good addition"	The main reflection from both of the stakeholders is that the main menu Is too simple. In a way this is good as it means that people wont be confused, but with being too simple, it is quite boring as well, which could have an effect on players opinions on the game.
CONTROLS PAGE IN MAIN MENU	"The controls themselves aren't too hard to memorise, as the blue players controls are quite standard, and the red players controls are very similar. The controls page itself is very straightforward and easy to understand."	"I understand that the controls had to be fitted onto a single keyboard, but I would rather use a controller, as it would be easier to use. Another thing is, I think that the controls page would have been better if you had a design of a keyboard, and you highlighted the keys and explained what the relevant action is. This is just a feature I think would be good but its not necessary, and what you have made is quite clear."	The opinions on the controls themselves was not too bad, but Jack said that using a controller would have been better. This is something I have noticed and mentioned in the evaluation of the usability features above.
RESTART BUTTON IN PAUSE MENU	"It is a useful feature for if you wanted to restart the game in any way. Accessing the pause menu is also quite straightforward and is a common way of getting to a pause menu"	"The pause menu itself is a good idea, as it allows players to take a quick break, or for them to restart or quit. The restart feature itself is quite good and it works as it should"	Both of the players had a positive feeling about the restart feature on the pause menu, as well as the pause menu itself. Having good feedback on a
HEALTH AND STAMINA BARS AT THE TOP OF THE SCREEN	"The health bar is perfect, it shows the health of each character accurately, and even changes colours when the health is at different stages, this is a very useful feature for when you are focusing. The stamina bar is the same. However at first, it was hard to tell exactly which one is which."	"The health and stamina bars are both fine, and clearly work as they should. I think if you maybe added an icon indicating which bar is for which stat, it would be better for people playing for the first time"	The general feedback is that the health and stamina bars were well made and worked fine. The problem with them was that they both were not sure which one was which, but after the game started and they were playing, they found it easy to tell which was which.

Limitations of the solution

LIMITATION	EFFECT ON GAME	HOW TO REDUCE EFFECT IN FUTURE DEVELOPMENTS
OPTIMISING PERFORMANCE	Contrary to what I initially thought, the code was not inefficient, so the memory and storage requirements were not that recourse intensive. This meant that the game would run smoothly and did not crash while in use.	All though this is not an issue in the current stage that it is at, it could be an issue if the game was further developed with more features that were more complex. It could then be made more efficient by peer review and testing. I could see what other developers think of the algorithms written and gather information on how to improve it (as I am still a beginner).
VISUAL DESIGN	The design of the characters, background, animations and the menus, were all limited as I don't have the best artistic skills, and it can be hard to find good designs for a lot of these things so that they match the theme perfectly.	I would most likely find someone to make a few designs and animations for each thing to make the game more visually appealing. It would have a big effect on engagement with the game as it is a feature that many heavily consider while picking games to play.
MULTI–PLATFORM SUPPORT	Having the game only available to PC users limits the potential market for the game quite a lot, as a large group of people do not have access to a suitable PC to play games like this.	I would have to learn how to make build version of the games that would work on different platforms such as games consoles and phones.
ONLINE PLAYER MODE	Since there is no online feature, it is quite hard for a lot of people to access it. I believe the two player mode would be the most popular game mode as it can be really fun playing with friends, but if you can only play with friends on a single system, a lot of people may be discouraged to play the game as a whole.	I would do some more research into how to create online games, and try to implement this as a feature. To make the game even more engaging for some, I would try to make a competitive mode with a ranking system, so that people will play against people with similar skills, so that there is more challenge.
NON-INTUITIVE CONTROLS	Like I have mentioned before, the controls are not the most intuitive, however for trying to fit them all onto one single keyboard, I think I have made the best decision. One problem about this is new players will most likely take a while to get used to the new controls, as they are not necessarily similar to others, making this game harder to simply "pick up and play".	I would try and solve this by introducing compatibility with a games controller of some type, and I would use controls that are similar to other popular fighting games. This would make it easier for people to pick up and the controls would make more sense.

Maintenance of solution

Maintenance features are those that will help the upkeep of the game. This will be especially important if the game was published to the public, and if it had an online mode. The maintenance features also include features that would increase engagement with the game, and ones that will bring more popularity to the game.

MAINTENANCE	JUSITIFCATION
BUG REPORTS	This is a necessary feature I would need to implement if the game were to have an online mode. If people had bugs or they crash, I would need to create a system that reports the information of the problem to my system so that the issue can be investigated and see if it is a common issue that can be solved.
BIGGER CHARACTER ROSTER	A bigger character roster would make the game more reusable, as it would mean players will get less bored after each time playing. This is good as it will increase overall engagement of a wide variety of players.
FEEDBACK SYSTEM	It would be helpful to gain insight into what suggestions people would have to improve the game, so I would add a feedback form so that people can voice their thoughts. This is the best way that good, user friendly updates can be made as you are directly satisfying the user wants and needs.
3D GAME UPDATE	As I made the game in unity, there is great support for 3d games. I could make my game into a 3d modelled game, which has better visual effects with the lighting and camera angles etc. Before doing this I would have to do more surveys to see if people rather a 3d game with better visual effects over the nostalgic feel of a retro style 2d game.
GAMES CONTROLLER SUPPORT	From stakeholder feedback, it seemed to be a good idea to add support for games controllers. This would require me to add certain libraries and modules in the scripts so that I could actually collect input from the controllers.
IMPROVEMENTS ON THE AI	An improvement can be made to the current AI with enough data. What I could do is gather data from current games against the AI, and see what kind of playstyles were most effective in doing most damage, and winning against the users. With this data I could implement a neural network which decides the best moves for the AI. A further development from this could be to create an account system, where you can play online and save your recent games. With this, could be a separate AI mode which adapts to your playstyle, and means that as you improve, the AI also improves so there will always (up to some limit) be a good training bot there for you to practise on.

Code Listing

Blue boxer movement:

```
0 references
public class blueBoxerMovement : MonoBehaviour
{
    3 references
    [SerializeField] private Rigidbody2D rigidBody;
    7 references
    private float horizontalBlue;
    1 reference
    [SerializeField] public float speed = 5f;

    1 reference
    public GameObject rightLimitGameObject;
    1 reference
    public GameObject leftLimitGameObject;

    2 references
    private Vector3 rightLimit;
    2 references
    private Vector3 leftLimit;


    0 references
    void Start()
    {
        rigidBody = GetComponent<Rigidbody2D>();

        rightLimit = rightLimitGameObject.transform.position;
        leftLimit = leftLimitGameObject.transform.position;

    }

    0 references
    void Update()
    {

        if (Input.GetKey(KeyCode.A)){
            horizontalBlue = -1.0f;
        }
        else if (Input.GetKey(KeyCode.D)){
            horizontalBlue = 1.0f;
        }
        else{
            horizontalBlue = 0f;
        }

    }

    0 references
    private void FixedUpdate()
    {
        if((transform.position.x <= leftLimit.x && horizontalBlue == -1.0) || (transform.position.x >= rightLimit.x && horizontalBlue == 1.0))
        {
            horizontalBlue = 0;
        }

        rigidBody.velocity = new Vector2(horizontalBlue * speed, rigidBody.velocity.y);
    }
}
```

Red boxer movement:

```
public class redBoxerMovement : MonoBehaviour
{
    3 references
    [SerializeField] private Rigidbody2D rigidBody;
    7 references
    private float horizontalRed;
    1 reference
    [SerializeField] public float speed = 5f;

    1 reference
    public GameObject rightLimitGameObject;
    1 reference
    public GameObject leftLimitGameObject;

    2 references
    private Vector3 rightLimit;
    2 references
    private Vector3 leftLimit;


    0 references
    void Start()
    {
        rigidBody = GetComponent<Rigidbody2D>();

        rightLimit = rightLimitGameObject.transform.position;
        leftLimit = leftLimitGameObject.transform.position;

    }

    0 references
    void Update()
    {

        if (Input.GetKey(KeyCode.J)){
            horizontalRed = -1.0f;
        }
        else if (Input.GetKey(KeyCode.L)){
            horizontalRed = 1.0f;
        }
        else{
            horizontalRed = 0.0f;
        }

    }

    0 references
    private void FixedUpdate()
    {
        if ((transform.position.x >= rightLimit.x && horizontalRed == 1.0) || (transform.position.x <= leftLimit.x && horizontalRed == -1.0))
        {
            horizontalRed = 0;
        }

        rigidBody.velocity = new Vector2(horizontalRed * speed, rigidBody.velocity.y);
    }
}
```

Blue player controller:

```
0 references
public class BluePlayerController : MonoBehaviour
{
    7 references
    [SerializeField]public Animator animator;
    11 references
    [SerializeField]public Stamina stamina;

    0 references
    void Update()
    {
        // for punching
        if (Input.GetKeyDown(KeyCode.F) && stamina.currentEnergy > 0)
        {
            animator.SetTrigger("bluePunch");
            stamina.PunchConsumeEnergy(5);
        } else if (Input.GetKeyDown(KeyCode.F) && stamina.currentEnergy == 0)
        {
            Debug.Log("NO STAMINA!");
        }
        // for blocking
        if (Input.GetKeyDown(KeyCode.W) || Input.GetKey(KeyCode.W) && stamina.currentEnergy > 0)
        {
            animator.SetBool("isBlueBlocking", true);
            stamina.drainEnergy(stamina.blockingDrainRate);
        }
        else if (Input.GetKeyDown(KeyCode.W) || Input.GetKey(KeyCode.W) && stamina.currentEnergy == 0)
        {
            animator.SetBool("isBlueBlocking", false);
            Debug.Log("NO STAMINA!");
        }
        if (Input.GetKeyUp(KeyCode.W))
        {
            animator.SetBool("isBlueBlocking", false);
        }
        // for crouching
        if (Input.GetKeyDown(KeyCode.S) || Input.GetKey(KeyCode.S) && stamina.currentEnergy > 0)
        {
            animator.SetBool("isBlueCrouching", true);
            stamina.drainEnergy(stamina.crouchingDrainRate);
        }
        else if (Input.GetKeyDown(KeyCode.S) || Input.GetKey(KeyCode.S) && stamina.currentEnergy == 0)
        {
            animator.SetBool("isBlueCrouching", false);
            Debug.Log("NO STAMINA!");
        }
        if (Input.GetKeyUp(KeyCode.S))
        {
            animator.SetBool("isBlueCrouching", false);
        }
    }

}
```

Red player controller:

```
public class RedPlayerController : MonoBehaviour
{
    7 references
    [SerializeField]public Animator animator;
    11 references
    [SerializeField]public Stamina stamina;

    0 references
    void Update()
    {
        //for punching
        if (Input.GetKeyDown(KeyCode.H) && stamina.currentEnergy > 0)
        {
            animator.SetTrigger("redPunch");
            stamina.PunchConsumeEnergy(5);
        } else if (Input.GetKeyDown(KeyCode.H) && stamina.currentEnergy == 0)
        {
            Debug.Log("NO STAMINA!");
        }
        // for blocking
        if (Input.GetKeyDown(KeyCode.I) || Input.GetKey(KeyCode.I) && stamina.currentEnergy > 0)
        {
            animator.SetBool("isRedBlocking", true);
            stamina.drainEnergy(stamina.blockingDrainRate);
        }
        else if (Input.GetKeyDown(KeyCode.I) || Input.GetKey(KeyCode.I) && stamina.currentEnergy == 0)
        {
            animator.SetBool("isRedBlocking", false);
            Debug.Log("NO STAMINA!");
        }
        if (Input.GetKeyUp(KeyCode.I))
        {
            animator.SetBool("isRedBlocking", false);
        }
        // for crouching
        if (Input.GetKeyDown(KeyCode.K) || Input.GetKey(KeyCode.K) && stamina.currentEnergy > 0)
        {
            animator.SetBool("isRedCrouching", true);
            stamina.drainEnergy(stamina.crouchingDrainRate);
        }
        else if (Input.GetKeyDown(KeyCode.K) || Input.GetKey(KeyCode.K) && stamina.currentEnergy == 0)
        {
            animator.SetBool("isRedCrouching", false);
            Debug.Log("NO STAMINA!");
        }
        if (Input.GetKeyUp(KeyCode.K))
        {
            animator.SetBool("isRedCrouching", false);
        }

    }

}
```

Blue health handler:

```
public class BlueHealthHandler : MonoBehaviour
{
    1 reference
    public GameOverScreen gameOverScreen;
    2 references
    public int health = 100;
    4 references
    public float currentHealth;

    4 references
    [SerializeField] public Animator animator;
    4 references
    public Slider slider;
    2 references
    public Gradient gradient;
    2 references
    public Image fill;


    0 references
    void Start()
    {
        currentHealth = health;
        slider.maxValue = health;
        fill.color = gradient.Evaluate(1f);

    }

    0 references
    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("redHand") && (animator.GetBool("isBlueBlocking") == false && animator.GetBool("isBlueCrouching") == false))
        {
            TakeDamage(5);
        }
        else if (other.CompareTag("redHand") && animator.GetBool("isBlueBlocking") == true)
        {
            TakeDamage(3);
            Debug.Log("Blue BLOCKED red punch");
        }
        else if (other.CompareTag("redHand") && animator.GetBool("isBlueCrouching") == true)
        {
            TakeDamage(0);
            Debug.Log("Blue CROUCHED under red punch");
        }
        if (currentHealth <= 0)
        {
            slider.value = 0;
            KO();
        }
    }

    1 reference
    public void KO()
    {
        Destroy(gameObject);
        Debug.Log(" BLUE KO ");
        gameOverScreen.ShowRedGameOverScreen();
    }
    3 references
    public void TakeDamage(int damage)
    {
        if (damage == 0)
        {
            return;
        }
        currentHealth -= damage;
    }

    0 references
    void Update()
    {
        slider.value = currentHealth;
        fill.color = gradient.Evaluate(slider.normalizedValue);
    }
}
```

Red health handler:

```
public class redHealthHandler : MonoBehaviour
{
    1 reference
    public GameOverScreen gameOverScreen;
    2 references
    public int health = 100;
    4 references
    public float currentHealth;
    4 references
    [SerializeField] public Animator animator;
    4 references
    public Slider slider;
    2 references
    public Gradient gradient;
    2 references
    public Image fill;

    0 references
    void Start()
    {
        currentHealth = health;
        slider.maxValue = health;
        fill.color = gradient.Evaluate(1f);
    }

    0 references
    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("blueHand") && animator.GetBool("isRedBlocking") == false && animator.GetBool("isRedCrouching") == false)
        {
            TakeDamage(5);
        }
        else if (other.CompareTag("blueHand") && animator.GetBool("isRedBlocking") == true)
        {
            TakeDamage(3);
            Debug.Log("Red BLOCKED blue punch");
        }
        else if (other.CompareTag("blueHand") && animator.GetBool("isRedCrouching") == true)
        {
            TakeDamage(0);
            Debug.Log("Red CROUCHED under blue punch");
        }
        if (currentHealth <= 0)
        {
            slider.value = 0;
            KO();
        }
    }

    1 reference
    public void KO()
    {
        Destroy(gameObject);
        Debug.Log(" RED KO ");
        gameOverScreen.ShowBlueGameOverScreen();
    }
    3 references
    public void TakeDamage(int damage)
    {
        if (damage == 0)
        {
            return;
        }
        currentHealth -= damage;
    }

    0 references
    void Update()
    {
        slider.value = currentHealth;
        fill.color = gradient.Evaluate(slider.normalizedValue);
    }
}
```

Stamina controller:

```
public class Stamina : MonoBehaviour
{
    5 references
    private int maxEnergy = 100;
    1 reference
    public float regenerationRate = 3f;
    12 references
    public int currentEnergy;
    4 references
    private float lastRegenerationTime;
    0 references
    [SerializeField]public float blockingDrainRate = 10f;
    0 references
    [SerializeField]public float crouchingDrainRate = 11f;
    2 references
    public Slider slider;

    0 references
    void Start()
    {
        currentEnergy = maxEnergy;
        lastRegenerationTime = Time.time;
        slider.maxValue = maxEnergy;
    }

    0 references
    void Update()
    {
        //regenerate energy constantly after 1.75 seconds from last regen
        if (Time.time - lastRegenerationTime >= 1.75f)
        {
            RegenerateEnergy();
            Debug.Log("Current stamina: " + currentEnergy);
        }
        slider.value = currentEnergy;
    }

    1 reference
    private void RegenerateEnergy()
    {
        float timeSinceLastRegen = Time.time - lastRegenerationTime;
        float energyToAdd = regenerationRate * timeSinceLastRegen;

        currentEnergy += Mathf.RoundToInt(energyToAdd);

        currentEnergy = Mathf.Clamp(currentEnergy, 0, maxEnergy);

        lastRegenerationTime = Time.time;
    }
    0 references
    public void PunchConsumeEnergy(int amount)
    {
        currentEnergy -= amount;
        currentEnergy = Mathf.Clamp(currentEnergy, 0, maxEnergy);
    }

    0 references
    public void drainEnergy(float amount)
    {
        float drainAmount = amount * (10 * Time.deltaTime);
        currentEnergy -= Mathf.RoundToInt(drainAmount);
        currentEnergy = Mathf.Clamp(currentEnergy, 0, maxEnergy);
    }
}
```

AI movement controller:

```
public class AIMovement : MonoBehaviour
{
    3 references
    [SerializeField] private Rigidbody2D rigidBody;
    1 reference
    [SerializeField] private float speed = 5f;

    1 reference
    public GameObject rightLimitGameObject;
    1 reference
    public GameObject leftLimitGameObject;

    2 references
    private Vector3 rightLimit;
    2 references
    private Vector3 leftLimit;

    6 references
    private float moveDirection = 0f;

    0 references
    void Start()
    {
        rigidBody = GetComponent<Rigidbody2D>();

        rightLimit = rightLimitGameObject.transform.position;
        leftLimit = leftLimitGameObject.transform.position;
    }

    0 references
    private void FixedUpdate()
    {
        if ((transform.position.x >= rightLimit.x && moveDirection == 1.0f) ||
            (transform.position.x <= leftLimit.x && moveDirection == -1.0f))
        {
            moveDirection = 0f;
        }


        rigidBody.velocity = new Vector2(moveDirection * speed, rigidBody.velocity.y);
    }


    0 references
    public void MoveLeft()
    {
        moveDirection = -1.0f;
    }

    0 references
    public void MoveRight()
    {
        moveDirection = 1.0f;
    }
}
```

AI player controller (actions):

```
public class AIPlayerController : MonoBehaviour
{
    7 references
    [SerializeField] private Animator animator;
    8 references
    [SerializeField] private Stamina stamina;
    0 references
    private Coroutine crouchCoroutine;
    0 references
    private Coroutine blockCoroutine;



    0 references
    public void Punch()
    {
        if (stamina.currentEnergy > 0)
        {
            animator.SetTrigger("redPunch");
            stamina.PunchConsumeEnergy(5);
        }
        else
        {
            Debug.Log("NO STAMINA FOR PUNCH!");
        }
    }


    0 references
    public void Block()
    {
        if (stamina.currentEnergy > 0)
        {
            StartCoroutine(BlockCoroutine());
        }
        else
        {
            animator.SetBool("isRedBlocking", false);
            Debug.Log("NO STAMINA FOR BLOCKING!");
        }
    }


    0 references
    public void Crouch()
    {
        if (stamina.currentEnergy > 0)
        {
            StartCoroutine(CrouchCoroutine());
        }
        else
        {
            animator.SetBool("isRedCrouching", false);
            Debug.Log("NO STAMINA FOR CROUCHING!");
        }
    }
```

```
    1 reference
    private IEnumerator CrouchCoroutine()
    {
        animator.SetBool("isRedCrouching", true);
        stamina.drainEnergy(stamina.crouchingDrainRate);

        float duration = Random.Range(1.5f, 3f);
        yield return new WaitForSeconds(duration);

        animator.SetBool("isRedCrouching", false);

    }

    1 reference
    private IEnumerator BlockCoroutine()
    {
        animator.SetBool("isRedBlocking", true);
        stamina.drainEnergy(stamina.blockingDrainRate);

        float duration = Random.Range(2f, 4f);
        yield return new WaitForSeconds(duration);

        animator.SetBool("isRedBlocking", false);
    }
}
```


AI health handler:

```
public class AIHealthHandler : MonoBehaviour
{
    1 reference
    public GameOverScreen gameOverScreen;
    2 references
    public int health = 100;
    5 references
    public float currentHealth;
    4 references
    [SerializeField] public Animator animator;
    4 references
    public Slider slider;
    2 references
    public Gradient gradient;
    2 references
    public Image fill;

    0 references
    void Start()
    {
        currentHealth = health;
        slider.maxValue = health;
        fill.color = gradient.Evaluate(1f);
    }

    0 references
    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("blueHand") && animator.GetBool("isRedBlocking") == false && animator.GetBool("isRedCrouching") == false)
        {
            TakeDamage(5);
        }
        else if (other.CompareTag("blueHand") && animator.GetBool("isRedBlocking") == true)
        {
            TakeDamage(3);
            Debug.Log("Red BLOCKED blue punch");
        }
        else if (other.CompareTag("blueHand") && animator.GetBool("isRedCrouching") == true)
        {
            TakeDamage(0);
            Debug.Log("Red CROUCHED under blue punch");
        }
        if (currentHealth <= 0)
        {
            slider.value = 0;
            KO();
        }
    }

    1 reference
    public void KO()
    {
        Destroy(gameObject);
        Debug.Log(" RED KO ");
        gameOverScreen.ShowBlueGameOverScreen();
    }
    3 references
    public void TakeDamage(int damage)
    {
        if (damage == 0)
        {
            return;
        }
        currentHealth -= damage;
        Debug.Log("Red Health: " + currentHealth);
    }

    0 references
    void Update()
    {
        slider.value = currentHealth;
        fill.color = gradient.Evaluate(slider.normalizedValue);
    }
}
```

Level 1 AI controller (action choice):

```
public class AIController : MonoBehaviour
{
    3 references
    public AIPlayerController controller;
    2 references
    public AIMovement movement;

    1 reference
    public float minActionDuration = 0.5f;
    1 reference
    public float maxActionDuration = 1.5f;

    2 references
    private float actionTimer;
    2 references
    private float nextActionTime;
    5 references
    private ActionType currentAction;
    2 references
    private ActionType previousAction;

    10 references | 2 references | 1 reference | 1 reference | 1 reference | 1 reference
    enum ActionType { Punch, Block, Crouch, MoveLeft, MoveRight }

    0 references
    void Start()
    {
        ChooseNextAction();
    }

    0 references
    void Update()
    {
        // Increment the action timer
        actionTimer += Time.deltaTime;

        // Check if it's time to choose the next action
        if (Time.time >= nextActionTime)
        {
            // Perform the current action
            PerformAction(currentAction);

            // Choose the next action
            ChooseNextAction();
        }
    }
```

```
    void ChooseNextAction()
    {
        // Reset the action timer
        actionTimer = 0f;

        // Define weights for each action
        float[] actionWeights = { 5f, 1f, 1f, 1.75f, 1.75f};

        // Choose a random action that is not the same as the previous one
        do
        {
            // Choose a random action based on weights
            float totalWeight = 0f;
            foreach (float weight in actionWeights)
            {
                totalWeight += weight;
            }

            float randomValue = Random.value * totalWeight;

            for (int i = 0; i < actionWeights.Length; i++)
            {
                if (randomValue < actionWeights[i])
                {
                    currentAction = (ActionType)i;
                    break;
                }
                randomValue -= actionWeights[i];
            }

        } while (currentAction == previousAction && currentAction != ActionType.Punch);

        // Store the current action as the previous action
        previousAction = currentAction;

        // Calculate the next action time
        nextActionTime = Time.time + Random.Range(minActionDuration, maxActionDuration);
    }

    1 reference
    void PerformAction(ActionType action)
    {
        switch (action)
        {
            case ActionType.Punch:
                controller.Punch();
                break;
            case ActionType.Block:
                controller.Block();
                break;
            case ActionType.Crouch:
                controller.Crouch();
                break;
            case ActionType.MoveLeft:
                movement.MoveLeft();
                break;
            case ActionType.MoveRight:
                movement.MoveRight();
                break;
        }
    }
}
```

Blue recent moves:

```
public class BlueRecentMoves : MonoBehaviour
{
    2 references
    public Stack<ActionType> recentMoves = new Stack<ActionType>();

    10 references | 1 reference | 1 reference | 1 reference | 1 reference | 1 reference
    public enum ActionType { Punch, Block, Crouch, MoveLeft, MoveRight }
    1 reference
    public int punchCount = 0;
    1 reference
    public int blockCount = 0;
    1 reference
    public int crouchCount = 0;
    1 reference
    public int moveLeftCount = 0;
    1 reference
    public int moveRightCount = 0;


    0 references
    void Update()
    {
        punchCount = GetMoveCount(ActionType.Punch);
        blockCount = GetMoveCount(ActionType.Block);
        crouchCount = GetMoveCount(ActionType.Crouch);
        moveLeftCount = GetMoveCount(ActionType.MoveLeft);
        moveRightCount = GetMoveCount(ActionType.MoveRight);
    }

    0 references
    public void DebugRecentMoves()
    {
        string debugMessage = "Recent Moves: ";

        foreach (ActionType action in recentMoves)
        {
            debugMessage += action.ToString() + ", ";
        }

        debugMessage = debugMessage.TrimEnd(' ', ',');

        Debug.Log(debugMessage);
    }

    5 references
    public int GetMoveCount(ActionType action)
    {
        int count = 0;

        foreach (ActionType move in recentMoves)
        {
            if (move == action)
            {
                count++;
            }
        }

        return count;
    }
}
```

Blue player controller (specially uses stack to record recent moves for Level 2 AI):

```
public class BluePlayerControllerAI : MonoBehaviour
{
    7 references
    [SerializeField]public Animator animator;
    11 references
    [SerializeField]public Stamina stamina;

    6 references
    [SerializeField]private BlueRecentMoves moveManager;
    0 references
    void Update()
    {
        // for punching
        if (Input.GetKeyDown(KeyCode.F) && stamina.currentEnergy > 0)
        {
            animator.SetTrigger("bluePunch");
            stamina.PunchConsumeEnergy(5);
            moveManager.recentMoves.Push(BlueRecentMoves.ActionType.Punch);
            moveManager.DebugRecentMoves();

        } else if (Input.GetKeyDown(KeyCode.F) && stamina.currentEnergy == 0)
        {
            Debug.Log("NO STAMINA!");
        }
        // for blocking
        if (Input.GetKeyDown(KeyCode.W) && stamina.currentEnergy > 0)
        {
            animator.SetBool("isBlueBlocking", true);
            stamina.drainEnergy(stamina.blockingDrainRate);
            moveManager.recentMoves.Push(BlueRecentMoves.ActionType.Block);
            moveManager.DebugRecentMoves();
        }
        else if (Input.GetKeyDown(KeyCode.W) || Input.GetKey(KeyCode.W) && stamina.currentEnergy == 0)
        {
            animator.SetBool("isBlueBlocking", false);
            Debug.Log("NO STAMINA!");
        }
        if (Input.GetKeyUp(KeyCode.W))
        {
            animator.SetBool("isBlueBlocking", false);
        }
        // for crouching
        if (Input.GetKeyDown(KeyCode.S) && stamina.currentEnergy > 0)
        {
            animator.SetBool("isBlueCrouching", true);
            stamina.drainEnergy(stamina.crouchingDrainRate);
            moveManager.recentMoves.Push(BlueRecentMoves.ActionType.Crouch);
            moveManager.DebugRecentMoves();
        }
        else if (Input.GetKeyDown(KeyCode.S) || Input.GetKey(KeyCode.S) && stamina.currentEnergy == 0)
        {
            animator.SetBool("isBlueCrouching", false);
            Debug.Log("NO STAMINA!");
        }
        if (Input.GetKeyUp(KeyCode.S))
        {
            animator.SetBool("isBlueCrouching", false);
        }
    }

}
```

Level 2 AI controller (chooses actions):

```
5   public class level2AI : MonoBehaviour
6   {
        3 references
7       public AIPlayerController controller;
        2 references
8       public AIMovement movement;
        2 references
9       public AIHealthHandler AiHealth;
        5 references
10      public BlueRecentMoves recentMoves;
        2 references
11      public BlueHealthHandler bluehealth;
        2 references
12      public BluePlayerControllerAI bluecontroller;
13
        1 reference
14      public Transform AI;
        1 reference
15      public Transform BluePlayer;
        3 references
16      public float distance;
17
        1 reference
18      public float minActionDuration = 0.5f;
        1 reference
19      public float maxActionDuration = 1.5f;
20
        2 references
21      private float actionTimer;
        2 references
22      private float nextActionTime;
        5 references
23      private ActionType currentAction;
        2 references
24      private ActionType previousAction;
25
        10 references | 2 references | 1 reference | 1 reference | 1 reference | 1 reference
26      enum ActionType { Punch, Block, Crouch, MoveLeft, MoveRight }
27
        18 references
28      public float punchWeight;
        18 references
29      public float blockWeight;
        18 references
30      public float crouchWeight;
        18 references
31      public float moveLeftWeight;
        18 references
32      public float moveRightWeight;
33
        8 references
34      public float punchEvidence = 0.4f;
        8 references
35      public float blockEvidence = 0.2f;
        8 references
36      public float crouchEvidence = 0.1f;
        8 references
37      public float moveLeftEvidence = 0.15f;
        8 references
38      public float moveRightEvidence = 0.15f;
39
        3 references
40      private float nextConditionCheckTime;
41
```

```
42      void Start()
43      {
44          ChooseNextAction();
45          punchWeight = 5f;
46          blockWeight = 1f;
47          crouchWeight = 1f;
48          moveLeftWeight = 1.75f;
49          moveRightWeight = 1.75f;
50
51          nextConditionCheckTime = Time.time + 5f;
52      }
53
                0 references
54      void Update()
55      {
56          actionTimer += Time.deltaTime;
57
58          if (Time.time >= nextActionTime)
59          {
60              PerformAction(currentAction);
61              ChooseNextAction();
62          }
63          distance = Mathf.Abs(AI.position.x - BluePlayer.position.x);
64
65          if (Time.time >= nextConditionCheckTime)
66          {
67              conditions();
68              nextConditionCheckTime = Time.time + 5f;
69          }
70      }
71
                2 references
72      void ChooseNextAction()
73      {
74          // Reset the action timer
75          actionTimer = 0f;
76
77          // Define weights for each action
78          float[] actionWeights = { punchWeight, blockWeight, crouchWeight, moveLeftWeight, moveRightWeight};
79
80          // Choose a random action that is not the same as the previous one
81          do
82          {
83              // Choose a random action based on weights
84              float totalWeight = 0f;
85              foreach (float weight in actionWeights)
86              {
87                  totalWeight += weight;
88              }
89
90              float randomValue = Random.value * totalWeight;
91
92              for (int i = 0; i < actionWeights.Length; i++)
93              {
94                  if (randomValue < actionWeights[i])
95                  {
96                      currentAction = (ActionType)i;
97                      break;
98                  }
99                  randomValue -= actionWeights[i];
100             }
101
102         } while (currentAction == previousAction && currentAction != ActionType.Punch);
103
104         // Store the current action as the previous action
105         previousAction = currentAction;
106
107         // Calculate the next action time
108         nextActionTime = Time.time + Random.Range(minActionDuration, maxActionDuration);
109     }
110
```

```
111     void PerformAction(ActionType action)
112     {
113         switch (action)
114         {
115             case ActionType.Punch:
116                 controller.Punch();
117                 break;
118             case ActionType.Block:
119                 controller.Block();
120                 break;
121             case ActionType.Crouch:
122                 controller.Crouch();
123                 break;
124             case ActionType.MoveLeft:
125                 movement.MoveLeft();
126                 break;
127             case ActionType.MoveRight:
128                 movement.MoveRight();
129                 break;
130         }
131     }
132
133
134
135     //LEVEL 2 AI STUFF
136
            40 references
137     public float CalculateNewWeight(float priorWeight, float likelihood, float evidence)
138     {
139         float priorProbability = priorWeight / 10.5f;
140         // calculate probability using Bayes' theorem
141         float updatedProbability = (priorProbability * likelihood) / evidence;
142         // calculate new weight by multiplying probability by total weight
143         float updatedWeighting = updatedProbability * 10.5f;
144         return updatedWeighting;
145     }
146
            1 reference
147     public void conditions()
148     {
149         if (bluehealth.currentHealth < 25f){
150             punchWeight = CalculateNewWeight(punchWeight, 0.5f, punchEvidence);
151             blockWeight = CalculateNewWeight(blockWeight, 0.25f, blockEvidence);
152             crouchWeight = CalculateNewWeight(crouchWeight, 0.08f, crouchEvidence);
153             moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.2f, moveLeftEvidence);
154             moveRightWeight = CalculateNewWeight(moveRightWeight, 0.13f, moveRightEvidence);
155
156         }
157         if (AiHealth.currentHealth < 25f){
158             punchWeight = CalculateNewWeight(punchWeight, 0.4f, punchEvidence);
159             blockWeight = CalculateNewWeight(blockWeight, 0.3f, blockEvidence);
160             crouchWeight = CalculateNewWeight(crouchWeight, 0.3f, crouchEvidence);
161             moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.25f, moveLeftEvidence);
162             moveRightWeight = CalculateNewWeight(moveRightWeight, 0.4f, moveRightEvidence);
163         }
164         if (bluecontroller.stamina.currentEnergy < 30f){
165             punchWeight = CalculateNewWeight(punchWeight, 0.6f, punchEvidence);
166             blockWeight = CalculateNewWeight(blockWeight, 0.9f, blockEvidence);
167             crouchWeight = CalculateNewWeight(crouchWeight, 0.2f, crouchEvidence);
168             moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.55f, moveLeftEvidence);
169             moveRightWeight = CalculateNewWeight(moveRightWeight, 0.15f, moveRightEvidence);
170         }
171         if (bluecontroller.stamina.currentEnergy <= 0f){
172             punchWeight = CalculateNewWeight(punchWeight, 0.7f, punchEvidence);
173             blockWeight = CalculateNewWeight(blockWeight, 0.25f, blockEvidence);
174             crouchWeight = CalculateNewWeight(crouchWeight, 0.15f, crouchEvidence);
175             moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.75f, moveLeftEvidence);
176             moveRightWeight = CalculateNewWeight(moveRightWeight, 0.075f, moveRightEvidence);
177         }
```

```
        //if distance is too great and enemy health is low
        if (bluehealth.currentHealth <= 20f && distance >= 11f){
            punchWeight = CalculateNewWeight(punchWeight, 0.6f, punchEvidence);
            blockWeight = CalculateNewWeight(blockWeight, 0.3f, blockEvidence);
            crouchWeight = CalculateNewWeight(crouchWeight, 0.15f, crouchEvidence);
            moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.5f, moveLeftEvidence);
            moveRightWeight = CalculateNewWeight(moveRightWeight, 0.25f, moveRightEvidence);
        }
        //if distance is too low and AI health is low
        if (AiHealth.currentHealth <= 20f && distance <= 5f){
            punchWeight = CalculateNewWeight(punchWeight, 0.3f, punchEvidence);
            blockWeight = CalculateNewWeight(blockWeight, 0.9f, blockEvidence);
            crouchWeight = CalculateNewWeight(crouchWeight, 0.5f, crouchEvidence);
            moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.15f, moveLeftEvidence);
            moveRightWeight = CalculateNewWeight(moveRightWeight, 0.55f, moveRightEvidence);
        }
        //recent moves suggesting an agressive playstyle
        if (recentMoves.punchCount >= 20 && recentMoves.moveRightCount >= 50){
            punchWeight = CalculateNewWeight(punchWeight, 0.4f, punchEvidence);
            blockWeight = CalculateNewWeight(blockWeight, 0.6f, blockEvidence);
            crouchWeight = CalculateNewWeight(crouchWeight, 0.3f, crouchEvidence);
            moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.15f, moveLeftEvidence);
            moveRightWeight = CalculateNewWeight(moveRightWeight, 0.25f, moveRightEvidence);
        }
        //recent moves suggesting a defensive playstyle
        if((recentMoves.blockCount >= 30 || recentMoves.crouchCount >= 20) && recentMoves.moveLeftCount >= 50){
            punchWeight = CalculateNewWeight(punchWeight, 0.6f, punchEvidence);
            blockWeight = CalculateNewWeight(blockWeight, 0.35f, blockEvidence);
            crouchWeight = CalculateNewWeight(crouchWeight, 0.15f, crouchEvidence);
            moveLeftWeight = CalculateNewWeight(moveLeftWeight, 0.45f, moveLeftEvidence);
            moveRightWeight = CalculateNewWeight(moveRightWeight, 0.15f, moveRightEvidence);
        }


    }


}
```


Main menu controller:

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.SceneManagement;
5
    0 references
6   public class MainMenuController : MonoBehaviour
7   {
        6 references
8       public CanvasGroup controlsPanel;
        6 references
9       public CanvasGroup gameModePanel;
        2 references
10      public CanvasGroup mainMenuPanel;
11
12      //When game starts, only the main menu is visible
        0 references
13      public void Start()
14      {
15          mainMenuPanel.alpha = 1;
16          mainMenuPanel.blocksRaycasts = true;
17          controlsPanel.alpha = 0;
18          controlsPanel.blocksRaycasts = false;
19          gameModePanel.alpha = 0;
20          gameModePanel.blocksRaycasts = false;
21      }
        0 references
22      public void gameModeChoice()// load game mode choice menu
23      {
24          gameModePanel.alpha = 1;
25          gameModePanel.blocksRaycasts = true;
26      }
27
        0 references
28      public void pvpGame()// load the two player game
29      {
30          SceneManager.LoadScene("boxingScene");
31      }
        0 references
32      public void pveGame()// load the single player game
33      {
34          SceneManager.LoadScene("aiBoxingScene");
35      }
        0 references
36      public void pveGame2() // load level 2 ai game
37      {
38          SceneManager.LoadScene("aiBoxingScene2");
39      }
        0 references
40      public void gameModeReturn() // Return to the main menu from game mode choie
41      {
42          gameModePanel.alpha = 0;
43          gameModePanel.blocksRaycasts = false;
44      }
        0 references
45      public void Controls()// Load the controls menu
46      {
47          controlsPanel.alpha = 1;
48          controlsPanel.blocksRaycasts = true;
49      }
        0 references
50      public void controlsReturn()
51      {
52          controlsPanel.alpha = 0;
53          controlsPanel.blocksRaycasts = false;
54      }
55
        0 references
56      public void QuitGame()//  Quit the game
57      {
58          Application.Quit();
59      }
60  }
```

Pause menu controller:

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.SceneManagement;
5
    0 references
6   public class PauseMenuController : MonoBehaviour
7   {
        3 references
8       public GameObject PauseMenu;
        3 references
9       public bool isPaused;
        2 references
10      private string currentSceneName;
11
        0 references
12      void Start()
13      {
14          PauseMenu.SetActive(false);
15          currentSceneName = SceneManager.GetActiveScene().name;
16      }
        1 reference
17      public void PauseGame()
18      {
19          PauseMenu.SetActive(true);
20          Time.timeScale = 0;
21          isPaused = true;
22      }
        1 reference
23      public void ResumeGame()
24      {
25          PauseMenu.SetActive(false);
26          Time.timeScale = 1;
27          isPaused = false;
28      }
        0 references
29      public void RestartGame()
30      {
31          Time.timeScale = 1;
32          SceneManager.LoadScene(currentSceneName);
33      }
        0 references
34      public void ExitToMenu()
35      {
36          SceneManager.LoadScene("Main Menu");
37      }
        0 references
38      void Update()
39      {
40          // if game is not paused, pause. If game is paused, resume.
41          if (Input.GetKeyDown(KeyCode.Escape))
42          {
43              if (isPaused)
44              {
45                  ResumeGame();
46              }
47              else
48              {
49                  PauseGame();
50              }
51          }
52      }
53  }
54
```

Game over screen:

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
    0 references
5   public class GameOverScreen : MonoBehaviour
6   {
        2 references
7       public GameObject blueGameOverScreen;
        2 references
8       public GameObject redGameOverScreen;
9
        0 references
10      void Start()
11      {
12          blueGameOverScreen.SetActive(false);
13          redGameOverScreen.SetActive(false);
14      }
15
        0 references
16      public void ShowBlueGameOverScreen()
17      {
18          blueGameOverScreen.SetActive(true);
19      }
        0 references
20      public void ShowRedGameOverScreen()
21      {
22          redGameOverScreen.SetActive(true);
23      }
24  }
25
```