PROJECT REPORT

Designing a simple video game with Artificial Intelligence

MSc Computer Science

Department of Computer Science and Information Systems

Birkbeck, University of London

Sylwester Stremlau

Sstrem01

Disclaimer

I have read and understood the sections of plagiarism in the School Handbook and confirm that the work is my own, with the work of others clearly acknowledged. I give my permission to submit my submission to the plagiarism testing database that the College is using and test it using plagiarism detection software, search engines or meta-searching software.

SYLWESTER STREMLAU

Abstract

This project focuses on a creation of a very basic, arcade-like video game which includes a sophisticated and dynamic Artificial Intelligence. Unity Game Engine and the Microsoft Visual Studio tools were used to obtain the best results. The game takes inspiration from the early arcade games, with a basic and addictive game loop; 2D graphics; easy to understand mechanics, objectives and Graphical User Interface; and challenging opponents. Artificial Intelligence is used to control the opponent; this allows the user to play the game alone or with friends. Popular techniques like behaviour trees and finite state machines have been used to design the Artificial Intelligence, to create the most challenging opponents that will behave, make decisions and make mistakes like a real human. To make the project possible to finish, it only focuses on the most basic mechanics of the game and on two game modes. Possibility of expanding the project has been kept in mind during the design and implementation of the game mechanics and Artificial Intelligence. The project outcome have been evaluated by letting a small focus group play the game and answer a survey; their answers were then analyzed to find if the project meets its aims and objectives. This report is broken down into four main chapters: background information, design, implementation and the evaluation of the results. Some of those chapters are then split into two parts to ensure better understanding – first part explains the game mechanics and the second part explains the Artificial Intelligence.

Glossary

Artificial Intelligence (AI)

Artificial Intelligence is the computer simulation of human behaviour and intelligence. It includes various processes like learning, adapting and thinking to act like a real human. There are various types of AI, from weak (also known as narrow) designed to do a certain task, to more complex, like strong AI that acts and thinks more like a real human being (Rouse, 2019).

Rule Based Artificial Intelligence

Rule based AI is a way to program AI to make decisions based on rules created and modified by a human expert.

Machine Learning (ML)

Machine Learning is an implementation of Artificial Intelligence to learn and train itself using data. Machine Learning usually needs training data fed and supervised by an engineer, to look for patterns and similarities so it can make more complex and informed decisions on new data (Nagy, 2018).

Video game engine

A video game engine is a software environment that provides functionality to allow creating and developing games more easily. It provides stuff like rendering engine, physics engine, sound, scripting, animation and other essential functions needed to make a game (Ward, 2019).

Game Loop/Gameplay

Game loop is a very basic loop that controls the flow of the game. A typical game loop is: process inputs, update game and render objects on the screen (Bethke, 2003).

Game physics

Game physics are a simulation of physics that follow more or less how objects behave in real life. Games do not have to follow real life physics but they usually have their own rules that are consistent throughout the game (Bethke, 2003).

Graphical User Interface (GUI)

Graphical user interface is a way for user to interact with an application program through graphical items and indicators (Bethke, 2003).

Frame Rate

In gaming, frame rate is the rate in which the image on the screen is refreshed. In games actual objects are updated every frame and everything has to be rendered by the graphics card again and again, so the frame rate is affected by the hardware. Most computer games will be run on a variety of hardware which will render the game in different frame rates, so it is important to make sure that the physics of the game are not dependent on the frame rate (Klappenbach, 2019).

Couch co-op

Couch co-op is a mode in a game where more than two players can play together or against each other on one platform by sharing a keyboard or by playing on separate gamepads (Co-optimus.com, 2019).

Sfx

Sfx is an abbreviation of "special effects" and refers to visual enhancements or illusions in movie or TV and to sound effects.

Melee

In video games, melee explains all forms of close combat between a player and an opponent. This includes hitting an enemy with a weapon at a close range like a sword or bat.

Range combat

Range combat is the opposite of a melee combat, this means attacking and striking the opponent or an enemy from a distance. This includes using a gun, bow or any other ranged weapon.

Health Points (HP)

In game, the points the player or the opponent/enemy has at their disposal before their character 'dies', are called the health points, usually abbreviated to HP.

Computer Controlled Opponent

Computer controlled opponent is, usually abbreviated to CPU, is an opponent or an enemy that is programmed to use in game mechanics to response to the player.

Main Menu

Main Menu is a list options given to the user placed at the beginning. The options give user a way to choose to start the game or pick other features made available by the developer. For example, user could choose a game mode in the main menu, or choose to view help menu.

2D and 3D

2D games refer to games which happen on a 2D plane with the game objects and scenes rendered in 2D. 3D on the other hand refers to games which happen in 3D with everything rendered in 3D, while 2D has two axis, 'x' and 'y', 3D has one additional axis 'z' (Kotaku.com, 2019).

FPS

First Person Shooter refers to a type of games that are played from the point of view of the in game character. These games usually have a big focus on the gun combat and shooting mechanics (Technopedia.com, 2019).

RPG

Role-playing game refers to a type of game where the user takes a role and responsibilities of the in game character in a fictional setting.

Game Engine

Game engine is software that provides tools and features for development and design of games. It provides things like the audio engine, physics engine, level creation tools, programming tools and many more (Unity.com, 2019).

HUD

Head-up display in video games is a way of showing data like health of the player, stamina etc. without having the player to look away from the game.

CONTENTS

CHAPTER 1 - INTRODUCTION

1.1 - OVERVIEW

Video games are a huge part of a modern life and grew to be the biggest entertainment industry, worth almost 90 billion U.S. dollars (wepc.com, 2019). Video games come in a number of genres so it is very easy to find something that one may enjoy. Being a part of this industry can be a very enjoyable and fulfilling experience because the main aim of each game project is to create something that others will be able to have fun with.

This project focused on creating an enjoyable game experience that anyone should be able to pick up and enjoy, either alone or with friends. The aim of this project was to create a game with good overall experience that will challenge the player, with each match being distinct and exciting. The main objectives of the project were to create a working game prototype, create a stable and addictive gameplay, create intuitive and easy to use Graphical User Interface and create a dynamic and challenging Artificial Intelligence.

The project was worth tackling because creating games can be a very complex process and can teach how to code effectively and to learn how to plan and design a scalable project. It gave me an opportunity to try out some of the topics I studied during the year and gave me experience of working on a real project that has no clear end. Artificial Intelligence is a very big and important aspect in computer industry so, this project allowed me to dig deeper and understand its workings. One part of the project focused on the game mechanics and the other part on the Artificial Intelligence. This allowed creating a satisfying overall user experience without getting hanged up on small details. The game was kept very basic – inspired by Rocket League, Grand Theft Auto 2 and FIFA that provide interesting, challenging and arcade-like experience to users.

1.2 - GAME

There are two teams on a playing field, one ball and two goals just like in a classic football. Each team has to work together to score a goal by using various weapons like melee or guns to hit the ball into the opponents' goal. Each player has their own health points and the game is timed, the team with the highest score by the end of the game or the team that reaches the maximum score first, wins. Users are able to change the maximum score and game time in the game menu.

This type of game has been chosen because it is very basic, easy to implement and it is open ended so there was a lot of room to scale the game up or down if needed during the duration of the project. It was planned to have more players on each team, but the game was scaled down to include only one player on each, which made it possible for the AI to be finished in the time given for this project.

1.3 - TOOLS

The main tool that this game needed was a game engine that has all of the tools needed to create 2D games and has a programming language that is fairly easy to work with. The most common ones, mentioned in the project proposal, were GameMaker Studio, Unreal Engine and Unity. Each of these game engines provides excellent tools to create and develop games.

1.3.1 - GAMEMAKER

GameMaker is one of the most popular engines for 2D games developed by YoYo games. It supports number of platforms like macOS, Microsoft Windows or PlayStation. It uses a visual programming language and its own scripting language, Game Maker Language. It provides all important features like real-time animation editing, level editor, debugger audio mixer and many more (Steam, 2019; YoYo Games, 2019).

1.3.2 - UNREALENGINE

Unreal engine developed by Epic Games was also an excellent choice – it provides the necessary tools to develop games, like physics and audio engines, advanced AI systems, marketplace and animation toolsets. However, Unreal has been initially made for FPS games and while it is possible to create 2D games, it is less suited and research shows that there are much more Tutorials for 2D games for Unity than Unreal (UnrealEngine.com, 2019).

1.3.3 - UNITY

Unity is a free game engine developed by Unity Technologies initially released in 2005. It provides tools and frameworks for users to develop games in 2D and 3D graphics, virtual reality games and augmented reality games. It offers a scripting API in C# programming language and supports a wide range of platforms like Microsoft Windows, Nintendo Switch, PlayStation, Xbox and many more (Nintendo, 2019; PlayStation, 2019; Xbox, 2019). Unity was the main tool used to create this game, along with Visual Studio, development environment developed by Microsoft (Unity.com).

Unity has been chosen for this project because it provides all of the necessary tools that are needed to create games like Physics Engine, Graphics User Interface frameworks, Audio Engine, and has support for Artificial Intelligence development. Unity also provides free, in depth tutorials that explain every part of the game engine, and has a big community that create their own tutorials about creating games from ground up, fixing bugs and other related topics. This made it very easy to get into Unity and start creating the game without extensive experience in game development.

1.4 - STRUCTURE OF THE REPORT

In Chapter 2 I explain the background information about the main tool used in the project and the techniques used to evaluate the project.

In Chapter 3 I explain the analysis, design and the requirements of the project. This chapter shows each component and its design.

In Chapter 4 I explain the implementation of the project, this chapter breaks down into two parts, first part explains the game mechanics and the second part explains the implementation of the Artificial Intelligence.

In Chapter 5 I evaluate the project, and explain how it meets the aims and the objectives by going through the results of a survey.

In Chapter 6 I conclude the project report by going through the lessons learnt, reflections and the improvements.

CHAPTER 2 – BACKGROUND

This chapter will go over the Unity and the main components that have been used throughout this project.

2.1 - UNITY

Figure 1 show the basic Unity window, at the top of the window is a play button which allows running the game very easily. Hierarchy window, on the left, shows all game objects inside the current scene. Inspector on the right shows currently selected object and its components and at the bottom left there is projects' folder which includes every component in the game. The Scene window shows the game objects in a game scene, and the Game window shows how the game looks like.

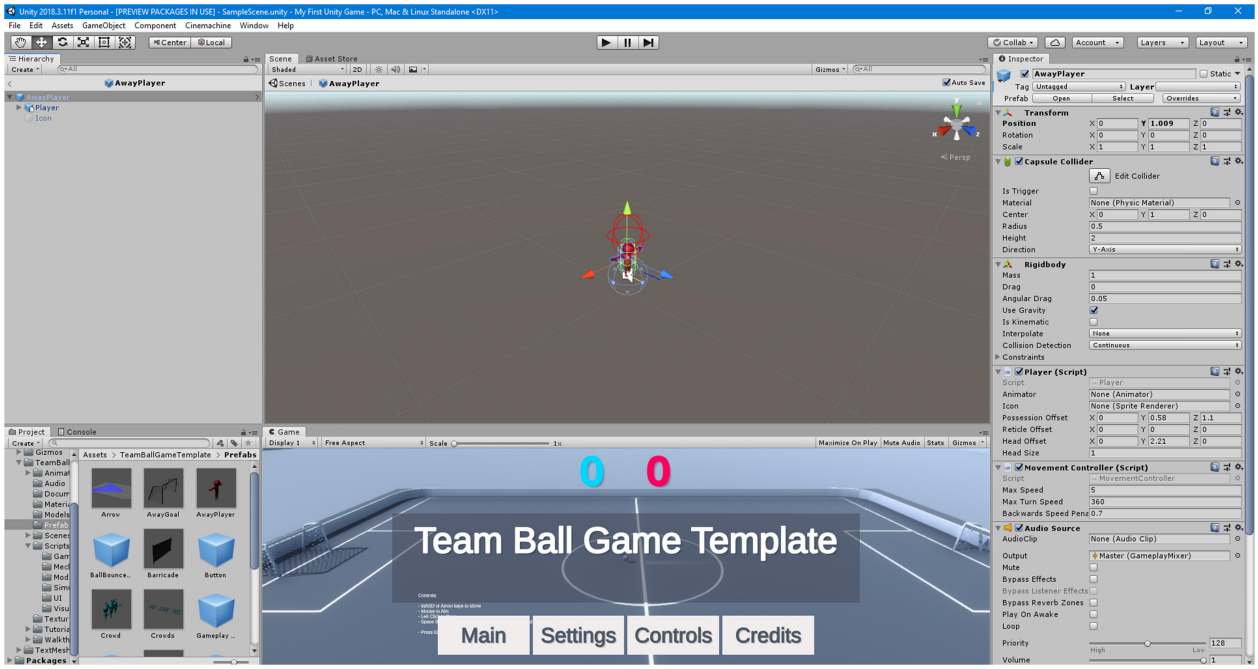


Fig. 2.1 - Unity window

2.1.1 - SCENE AND GAME OBJECTS

In Unity, each scene is like a level with its own objects, environment and UI to easily separate the game into small pieces. For example, almost every object that will be used in the Main Game is not needed in a Menu therefore there is no need to have it in the game scene.

Game object is a base class which all objects like the characters, props and scenery in the scene implement. It provides all the necessary functionality that allows the object to be affected and act like containers for other components. For example, attaching a lightning component to an object makes it a light object. All objects come with a Transform component which describes it position in a game scene (Unity.com, 2019).

A component is saved as a Script which is a C# file which acts on an object it is attached to. Components can have a very basic task like update the position of the given object every few frames, or a more complex task like simulate physics.

2.1.2 - PHYSICS

Unity provides a built-in physics engines for 3D and 2D games that handle the physical simulation. It provides Collider2D and Rigidbody2D. Rigidbody2D allows the game objects to be affected by various forces like the gravity. Collider2D allows objects to collide with each other and if paired with Rigidbody, allows objects to affect each other (Unity.com, 2019).

Collider2D is implemented in a UnityEngine.Physics2DModule and offers a lot of functionality to work with 2D collision. A Collider has to be attached to a game object and only collides with other objects that have a collider attached to it. There are various types of colliders, Box Collider2D, Circle Collider2D, Capsule Collider2D, Composite Collider2D, Polygon Collider2D and many more that work in 2D or 3D, like Sphere Collider. Colliders have few basic properties like the gameObject, tag, transform, hideFlags and name.

2.1.3 - MONOBEHAVIOUR

To act as an object component, each class should derive a `MonoBehaviour` base class. `MonoBehaviour` has all the important methods that are used by the Unity to make it work as a component. The methods are `Start()`, `Update()`, `FixedUpdate()` and `LateUpdate()` as well as `OnGUI`, `OnDisable` and `OnEnable` which are omitted in this report as they were not used. The update methods are called only if the `MonoBehabviour()` is enabled.

`Update()` and `Start()` are most commonly used methods because they allow adding a behavior to an object. For example, if an object has to move from one side to the other, its x or y position could be incremented inside the `Update()`. The Fragment 2.1 would make the player move by one pixel to the right, every frame.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class PlayerController : MonoBehaviour
6  {
7      void Update()
8      {
9          /...SetPosition(GameObject.pos.x + 1, GameObject.pos.y);
10     }
11 }
```

Code Fragment 2.1 - Example of a position method (pseudocode)

The speed in which the object would move would be dependent on the frame rate. Therefore the higher the frame rate, the faster the object would move. `FixedUpdate()` gets rid of this issue by updating the object in a fixed time. This time value can be accessed by using the `Time.fixedDeltaTime` which shows the in game time in seconds (Unity, 2019; Balkanay, 2016).

Name	Description	Called	Comments
`Start()`	Used to initialize the game objects by enabling the developer to set it up before the object is updated.	In the same frame that the script is enabled and before its Update methods. It is ran only once in the lifetime of its script.	Start is used similarly to a constructor and should be used instead of it when deriving from `MonoBehaviour` to avoid issues or unexpected results as explained by (Balkanay, 2016).
`Update()`	Update the game object it is attached to	Every frame	
`LateUpdate()`		Every frame but it is called only once all Update methods, inside the object and inside other objects, have been called.	Updating the camera position should be done only once other objects' position has been updated to avoid issues and unexpected behavior (Balkanay, 2016).
`FixedUpdate()`	Mostly used to calculate physics and to avoid issues with time dependent methods.	Every fixed time not dependent on the frame rate.	If the player has to move right at a certain speed, its position should be updated using the FixedUpdate method.

Table 2.1 – Main methods in MonoBehaviour

2.1.4 - USER INTERFACE

The user interface in Unity is easily implemented by creating a Canvas object. All user interface elements should be inside the Canvas area. Canvas is a basic Game Object with a Canvas component attached to it. Adding a UI element to Unity automatically creates a Canvas and sets it as a parent to that element. EventSystem object has to be added to use as a messaging system otherwise it would not be possible to interact with the objects.

Event system is used to send event messages to various objects by using a mouse, keyboard or other input systems. Its main objectives are to manage the selected game objects, manage and update the input modules and other like managing Raycasting. Unity sets up everything that is needed automatically to make the user interface work. Default settings were used for this project.

2.2 - EVALUATION

The main aim of evaluating this project will be to try and be as objective as possible. It will be evaluated by giving a focus group a number of questions and evaluating their answers. The questions will be written in a way that the answers can be easily cross referenced. As said by Ashe-Edmunds (2019), it is important to avoid 'Yes' or 'No' questions as these do not give any depth to the

answer. A question like “Did you like the game?” will be turned to “What did you like about the game the most?” The questions should also be easy to understand, since as explained by H. Harrison (2015), understanding the questions is the first step in answering it – if the question is not understood it cannot be answered correctly.

The group will be around 10 to 15 people and it will be made sure that everyone has played few matches in each game mode. This many participants are quite likely to provide enough information to reach saturation (Charmaz, 2006). Having a small group of people will ensure that the answers can be individually evaluated and that each person had a similar experience with the game (surveymonkey.com, 2019). It would also be very hard to find a group of people willing to give at least one hour of their time to play the game and answer the survey.

The answers will be evaluated by going through each answer, cross referencing them, looking for patterns and similarities and lastly, by finding how they compare with the aims and objectives of this project. It will be important to be objective and avoid faulty generalization (J.Wiley, 2004; surveymonkey.com, 2019).

CHAPTER 3 – ANALYSIS, REQUIREMENTS AND DESIGN

This chapter goes through the analysis, requirements and the design of the game. The first part goes through the overall rules of the game, how it will be played, and then it goes on to explain the game objects and their design. Lastly, it goes through the Artificial Intelligence.

3.1 - OVERVIEW

The game is a mix of various game types including shooters, fighting and sports games, with a top down camera view. It is heavily influenced by games like Rocket League, FIFA and Hotline Miami. On a basic level it is very similar to a classic football game in which there are two teams, one ball and two goals. Each team tries to kick the ball into the opponents' goal to score a point, the team with the highest points at the end of the match wins.

However, the teams are much smaller and should first be limited to only one player on a team to make the project a manageable size to finish, especially since a lot of focus in this project is on the computer controlled opponents. Having bigger team size would introduce new ways that the AI should behave and new situation to which it would have to adapt, this would complicate the process and may make the game impossible to finish in a short span of time.

The rules:

- No offside – there will be 'walls' placed around the play area which the ball will bounce off and which the players will not be able to move beyond
- Team gets two points per goal - only if the ball is all the way in the opponents goal and touches the back wall of the goal
- Players can use shield, swords and guns to defend their goals
- The position of the players resets to their default position after each goal and the ball goes back to the middle
- Players can also use the weapons to attack other players to drain their health
- Players start with 100HP, 100Stamina and 999 Ammo for each weapon
- Players can pick up boxes that spawn in one of the six fixed positions on the map
- One box spawns every 10 to 30 seconds
- Maximum of three boxes can be present at the same time
- Once the players health reaches zero, they are reset to their default position and their health is set back to 100 and the opponents team gets 1 points
- Players can only use one weapon at the same time

3.1.1 - THE VIEW

As explained in the project proposal, the 'camera' will be pointed from above like in Space Invaders or Hotline Miami. This setup fits the game the most because it fits the arcade style of the game and will be much easier to implement than a 3D game. This type of retro feel in a game is also very popular in modern games, for example, one of the best-selling games on the digital game market Steam in 2005, with over 530,000 copies sold was Undertale which has very basic, 2D graphics as seen in Figure 3.1 (Galyonkin, 2019). As said in the project proposal, the aim will be to make the

game look and feel very intuitive so the goal of the game is clear and easy to understand. Similar to Super Mario Bros platform game published by Nintendo and released in 1985 which has a very basic game loop but manages to engage and challenge the player.

Fig. 3.1 - Undertale game window

3.1.2 - CONTROLS

The game controls will be based on a gamepad to create a very best experience since the player should be able to move in every direction a full 360 movement can only be achieved with it. User will be able to rotate the player object using the right analogue stick, and use the weapons using the other buttons as seen in Figure 3.2. Since the user will be using his left hand to move, it would also make sense to use the left hand to trigger sprint, this should be achieved by using the L1 button. These types of controls are taken from a game like CallOfDuty and Enter the Gungeon (PwrDown, 2019; Game Guides, 2019).

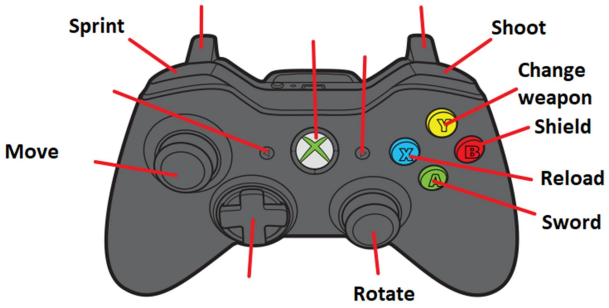


Fig. 3.2 – Initial controller layout

Some less experienced players may have difficulty with controlling the movement and rotation separately therefore alternative controls will be provided as seen in Figure 3.3. This should make the game easier to control and would leave their right hand for other tasks like using the weapons or a shield. A similar approach has been used in a game called Rocket League where players can either have their camera set to be behind-the-player at all times or to have it focused on the main point of the game which is the ball. The first type of control allows the players to be able to look around the game area and be aware of what is going on around them but it also makes it harder to control the game because they have to manually keep looking at the ball.

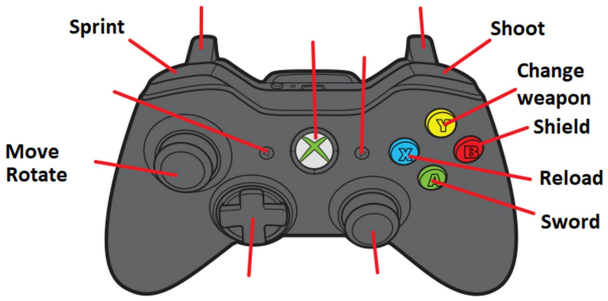


Fig. 3.3 – Alternative controls

3.1.3 - THE PLAY AREA

Before explaining the game loop and the goal of the game, it makes sense to first explain the game area. The game area will be a classic football field with each side belonging to each team as shown in Figure 3.4. There will be goals on each side to which the players have to kick the ball into to score a point. A team gets one point only when the ball hits the back of the goal. The playing field should be a basic game object with colliders around. The goals should be a separate game object to allow collision detection between the ball and the goal.

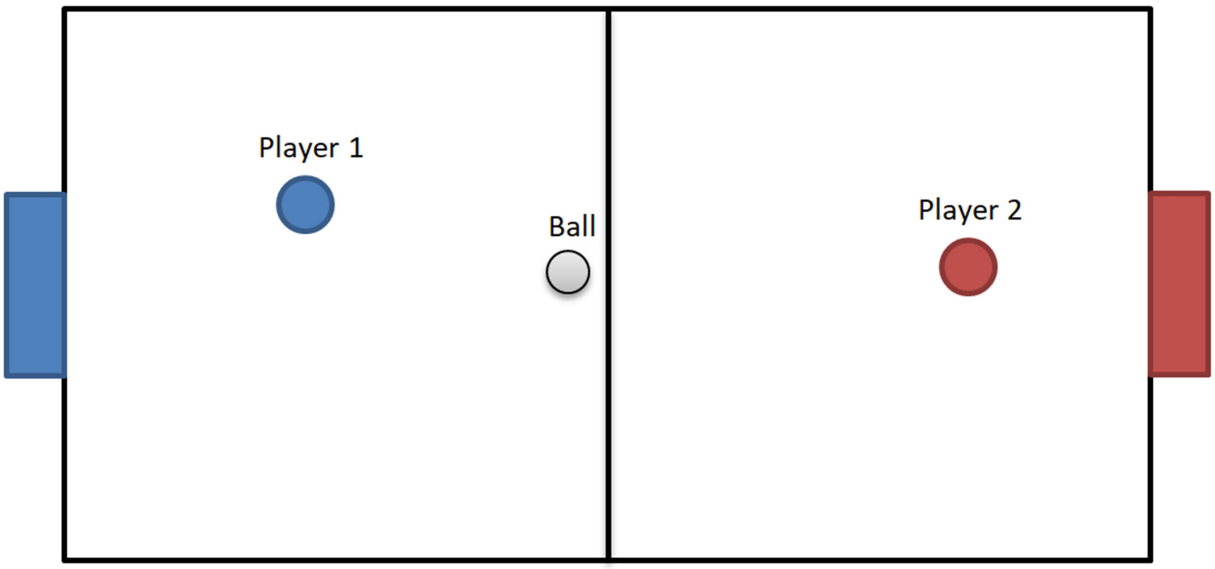


Fig. 3.4 - Playing field design

3.1.4 - THE BALL

At the beginning of the match the ball will spawn exactly in the middle of the pitch and whenever it gets hit into the goal, it resets there. This gives each team a fair position from which they can start the game. Player can push the ball with their characters, use swords to hit the ball or use guns to hit it from a distance. A ball should be a basic game object with a Collider2D and Rigidbody2D to simulate a real ball. A component should be created that checks for the collision between the ball and the goal and assigns points to the correct team.

3.1.5 - TEAMS

There should be a two team controller objects or components that hold the information about each team, like the teammates, opponents, goals, starting position of each player and the score of the game. Although the project will only focus on one player on each team, the team objects should be extendable to more players. As it can be seen in Figure 3.5, there should be Red and Blue team objects or components that would belong to a one Team Controller.

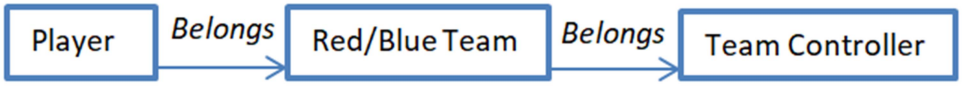


Fig. 3.5 – Team hierarchy

3.1.6 - STATISTICS

Users will have a number of statistics they need to keep track of, Health Points (HP), Stamina Points (SP), Shield Size (SS) and Ammo. The Health Points work very similar as in other games, the player starts with 100HP, if it goes below that, they can pick up boxes which replenish their Health Points by a certain number. When the player HP goes to zero, his position is reset to its default position and one point is given to the opposite team.

Stamina points will drain when a player sprints and just like with health points, player starts with 100 and once it goes to zero, player cannot sprint or use a sword. Stamina replenishes around 1 point per frame (multiplied by the `time.deltatime`).

3.1.7 - WEAPONS

A sword drains 4 to 6 points of stamina and will deal 10 to 15 damage. Players will have guns and each gun will have a different speed, ammo, reload time and strength. For now, only two are designed but more could be added. Table 3.1 shows two guns, Pistol and the Machine Gun, each gun has its own stats, strengths and weaknesses.

Name	**Speed**	**Magazine**	**Damage per bullet**	**Reload time**
Pistol	Slow	Small	High	Fast
Machine Gun	Fast	Large	Low	Medium

Table 3.1 – Weapons

3.1.8 - SHIELD

The last thing that players will have at their disposal is the shield. It will spawn in front of the character whenever the user presses “Shield” button (Figure 3.2) for a few seconds and slowly disappear. The shield will last for around 2 seconds and the player will be able to use it as much as possible. It will act just like a wall. That is, whenever a ball hits it, it will bounce back. It will be very useful when defending a goal. Its size will allow the player to push the ball away and with little bit of practice, can be used to score points. Shield can also be used to block bullets and player cannot use a gun or a sword when the shield is up.

3.1.9 - PICKUPS

Players will be able to keep up boxes which will either replenish their health, stamina or ammo.

Name	Effect
Replenish Health	10 to 50
Replenish Stamina	20 to 100
Replenish Ammo	50 to 200, random weapon

Table 3.2 - Pickups and their effects

3.2 - GAME OBJECTS

Figure 3.6 shows the player object, red arrows point to other game objects, and blue arrows point to components within the player. In reality, there will most likely be cross referencing between the objects, for example, controls component would point to sword, shield and weapons. Every component should reference the Main Player Controller for the information about its children objects, configuration or other components within. This would make it easier to control the player object from one point. This will most likely be the most complex game object in the game because it has the most functionality. Configuration component should keep all the information about the gamepad controls that a component like Controls would refer to. Controls should have all the necessary methods to use weapons, movement etc.

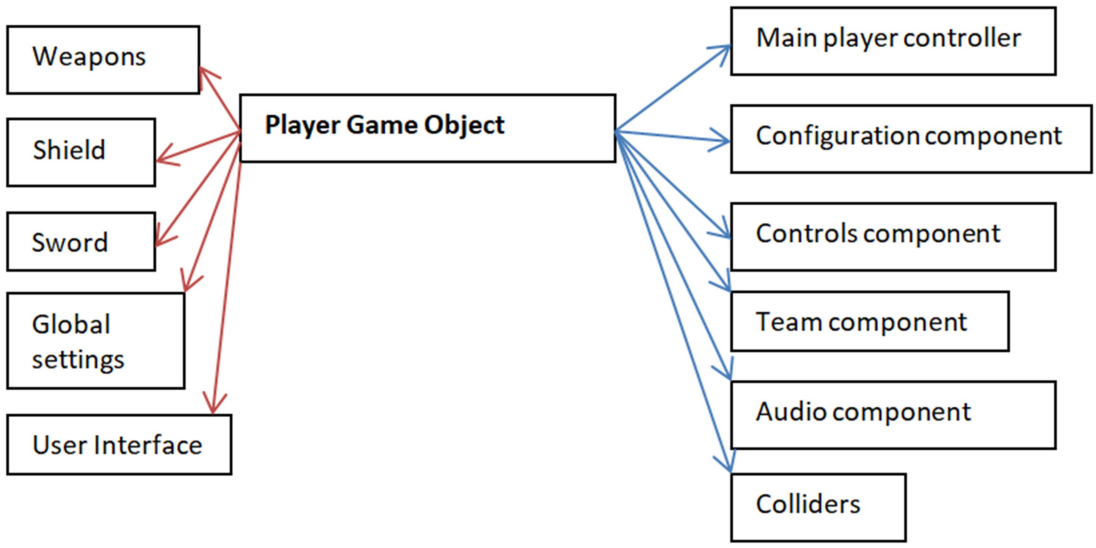


Fig. 3.6 - Player game object

Most objects like the shield or a ball will have the similar components as the sword object because they have similar functionality. For example, ball has the same components as the one shown in Figure 3.7 plus a Rigidbody2D to simulate a real ball.

The object shown in Figure 3.7 should also reference global settings for the information about its statistics.

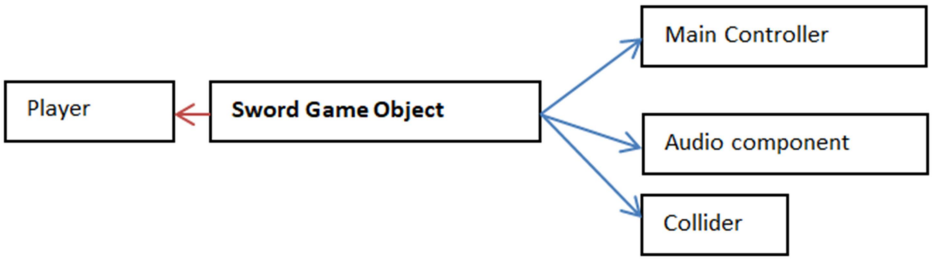


Fig. 3.7 - Sword game object

Weapon game object will be a bit more complicated than the sword game object. Fire point will be the position relative to the player from which the bullet should spawn. This will have to be passed down through the Gun List to the actual gun object. Guns will be kept as separate game objects and will have their own Main Controller components. This component should be a base class which has all the basic functionality of a gun like shoot, reload etc.

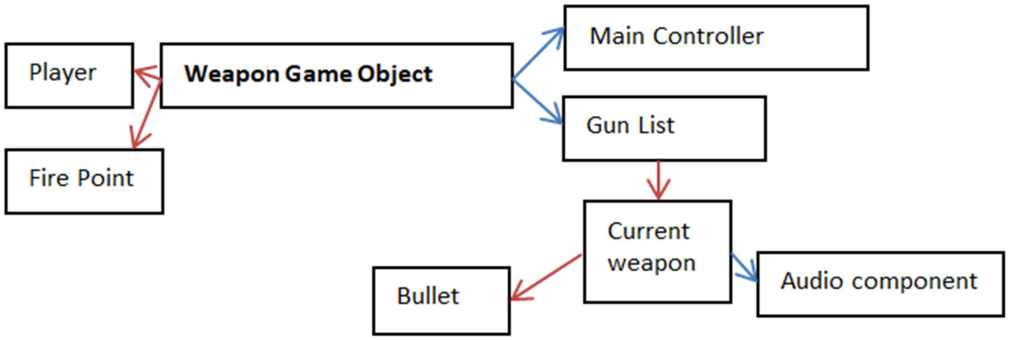


Fig. 3.8 - Weapon game object

3.3 – USER INTERFACE

3.3.1 - HUD

The User Interface should be easy to use, intuitive and convey lots of information in easy to understand way. Modern games range from excessive information on the screen to as little as possible. An example of a game with excessive amount of information is the Horizon Zero Dawn action RPG developed by Guerrilla Games and released in 2017. As it can be seen in Figure 3.9, the player is given all the important information on the screen like health points, stamina, weapons etc. The game has role playing elements therefore giving the player all these information is very important. As it can be seen, there are ten UI elements.

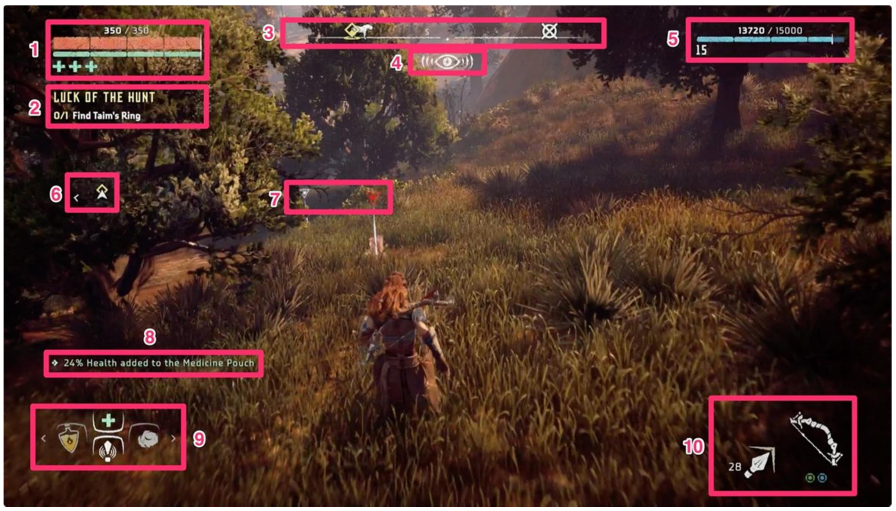


Fig. 3.9 - Horizon Zero Dawn in-game screenshot

On the other end there are games that are more like the movies. Uncharted 4, an action adventure game developed by Naughty Dog and released in 2017, is an example of a game where the player does not have too much impact on the game world and almost every event is scripted. This allows the game designers to have as little information on the screen as possible. In fact, most of the time the HUD is hidden except the times when the player is using weapons.

The game in this project will be somewhere in the middle, only necessary information will be shown. This is especially important because the game is fast paced and competitive so the player will only be able to take a brief look at the HUD. It takes an inspiration from older arcade games like Donkey Kong or Space Invaders. It will also be kept basic to avoid spending too much time on implementation as time is very limited. The plan is to have a bar for health, stamina and shield in one of the top corners of the game, as shown in Figure 3.10 and the weapon and ammo in one of the bottom corners. Score and timer will be kept in the middle top as in games like Donkey Kong Arcade.

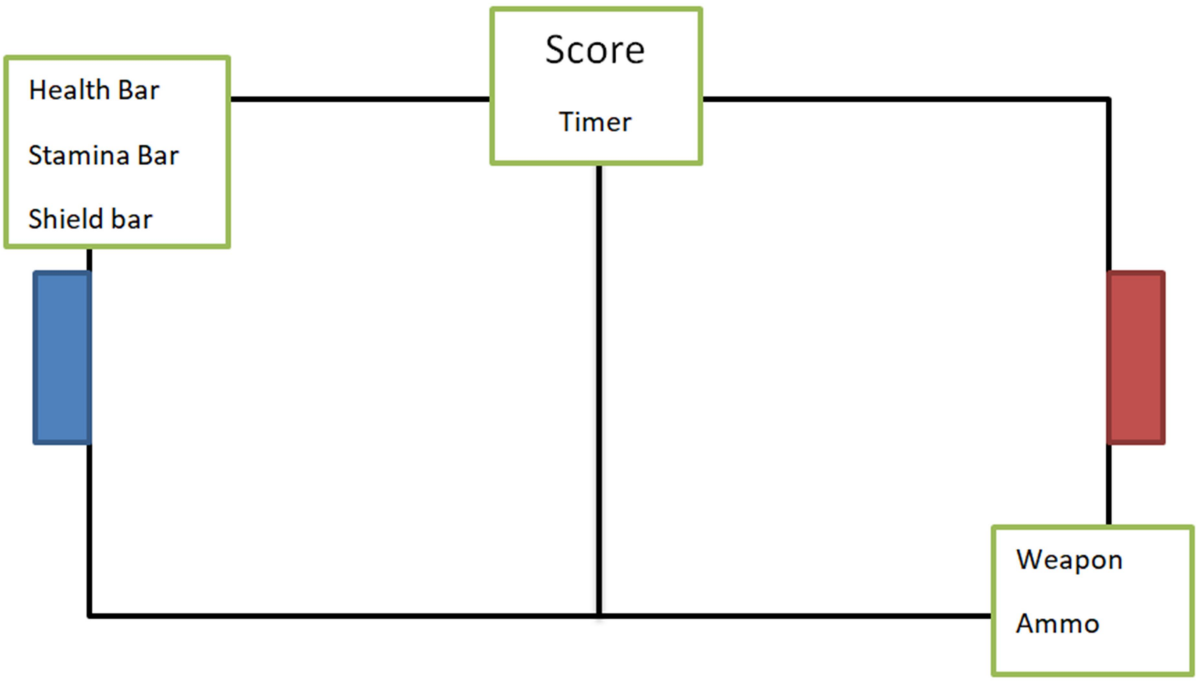


Fig. 3.10 - HUD design

3.3.2 - MENU

The Menu will be broken down into four scenes; Title Screen, Game Setup, Main and End Game. The user will be able to transition between the game scenes by using the available buttons. Figure 3.11 shows in detail which scenes the user will be access from each scene. One thing worth pointing out is that the user should not be able to access the End Game screen from other scene than the Main game. End Game screen shows the end of the game scores which are unknown until the end of the game.

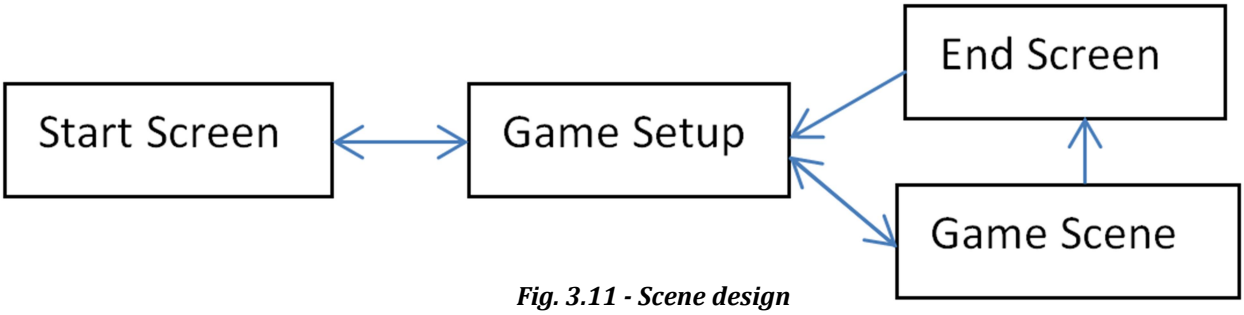


Fig. 3.11 - Scene design

Start screen will be the first thing the players see when they open the game. It will be kept very basic, with a name and a logo of the game, creator name and start button. It is also planned to have a theme song playing in the background just like in the games mentioned previously since this makes the game much more memorable and keeps with the theme of arcade games.

Start button in the Title screen would send the user to the game setup screen; here the user will be able to setup the settings of the game to his likings. User will be able to change settings like the maximum game score, time of the game or starting stats. Since the user will see this screen after every match, the total game score for each team will also be shown here.

Game setup has been taken from more modern games like CallOfDuty where players are given an option to create custom games that fit their play style. In arcade games there were usually fixed settings (CallOfDuty, 2019). Game settings will be preset to default settings which will be tested and which the game will be designed around. It is also planned to give user an option to save the settings of the game and the game score to a file although this will not be a priority since time will be very limited.

The End Game screen will be kept very simple; it will have the name of the team that won the game or “Draw!” if there is a draw. It will also have the score of the current game and the Continue button which will send the user back to the Game Setup. This will be like a transition screen between the game and setup screen so there is no need to add too much information.

3.4 - GAME MODES

The game modes are the way the user can play a game. Most games come with various game modes to find the one that will be most popular with users on which then the developers focus on the most. An example is Fortnite developed by Epic Games and released in 2017, it started with numerous game modes but the Battle Royale mode has been the most successful so now the developers are mainly focusing on updating it.

This game will be designed around two game modes to avoid making it overcomplicated, especially because any extra thing added to a game will impact the creation of the Artificial Intelligence and it is important to keep the scope of the game on a reasonable level. The two modes will be couch co-op, and the other will be User against the computer controlled opponent (AI). In both modes, there will only be one player on the team to avoid making the AI too complex.

3.4.1 - COUCH CO-OP (1VS1)

The first mode should be relatively easy to implement once one of the player object is implemented. The player game object would just have to be duplicated and the components should be setup accordingly. Then the control buttons will have to be changed to fit another gamepad/keyboard, and then it will have to be assigned to a different team. Important aspect that will have to be considered is to make sure that each user is able to see its character's status like health or stamina points.

3.4.1 - PLAYER VS. AI

The second mode is almost the same like the first but here the player will play against the Artificial Intelligence. This is the main mode of the game and the project focuses on it the most. Here, the user will not have to see its opponents' status so there will be much more space to spread out the HUD.

3.5 - ARTIFICIAL INTELLIGENCE – COMPUTER CONTROLLED OPPONENTS

Designing AI for a game like this creates a number of problems and questions that need to be taken care of. Main issue to address is to find out how the AI should behave in a 1vs1 game; this will be the basis for other game modes. It is very unlikely that there will be enough time to make the AI work in different game modes, as shown in project proposal Chapter 5; there should be just enough time to make the AI work in one game mode. Therefore this project will not focus on other game modes but the AI will be designed in such a way that it can be easily extended.

3.5.1 - MOVEMENT

The first issue will be to find out what the AI should focus on – the general direction that the player should look in. The obvious answer would be to look at the ball at all times and move towards it, this may seem reasonable but it would disable the AI to use weapons to attack the opponent, especially in a game with more than one player on each team, because it first needs to aim its gun in a direction that it wants to shoot. So a method will need to be created to allow it to look in one direction and move in another.

Using weapons and the shield is very straightforward since these methods should be already created as mentioned in a previous section. A way to call that method will only be needed – this could be done with a static method that takes the player game object and runs, for example, a `UseSword` method inside it that will be stored in Controls component.

Helper methods should be created that calculate the distance from the player object to another object, finds the rotation to a wanted object or finds the opponents' position. These should make it easier to create more sophisticated behaviours.

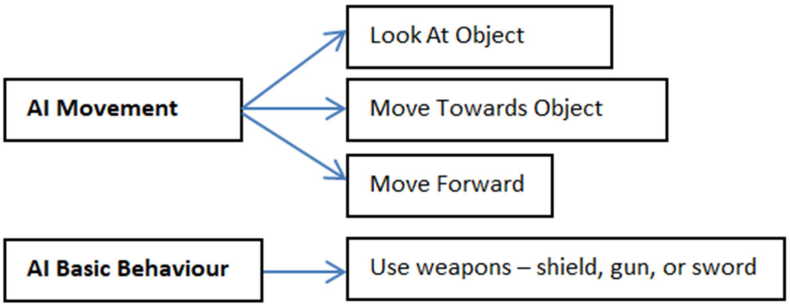


Fig. 3.12 - AI Behaviours

3.5.2 - BEHAVIOURS

The first, and most likely the most important behaviour, is to score a goal. This behaviour will have to be broken down into separate steps. The default weapon to use to shoot a goal is the sword, and to use a sword the player has to be close to the object that it wants to affect. Therefore, the first step would be to get close to the ball, this can be easily achieved by finding out the balls directions and moving towards it. Once the player is close to the ball, it should not use a sword yet because it could be positioned in a way that if he uses it, the ball will move in a different direction than desired. As shown in Figure 3.13, the blue circle is the AI player, and if it hits the ball, the red arrow shows where the ball will go and the green arrow shows where the ball should go – towards the opponents' goal. The second step should be to position itself around the ball so that the ball faces the goal. The last step would be to actually use the sword.

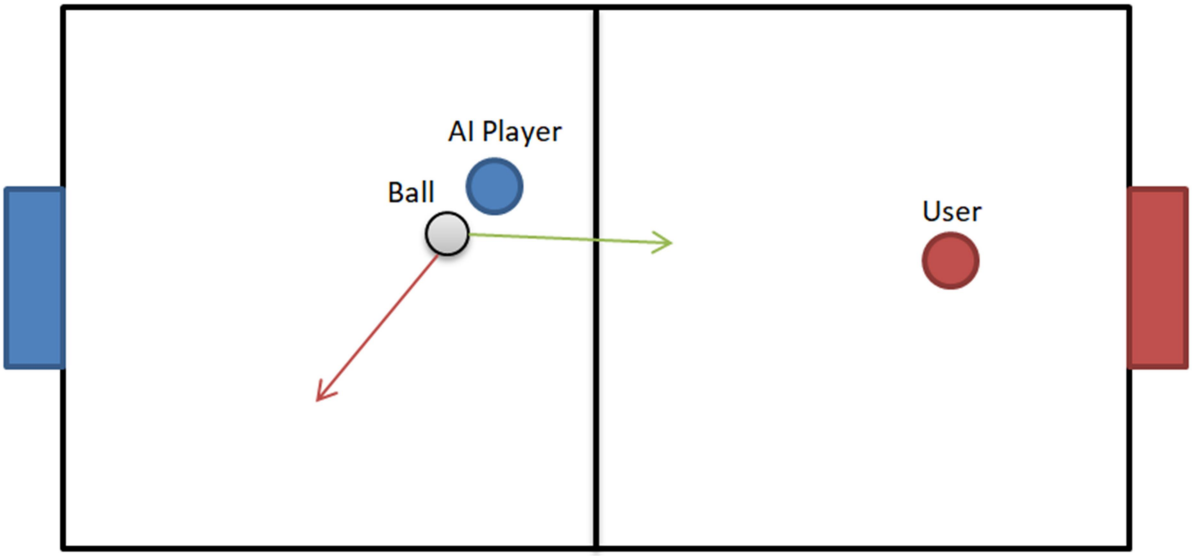


Fig. 3.13 - Green arrow- where the ball should go, Red arrow – where the ball would go

As it can be noted, there are a lot of tasks and calculations that need to be done before the task is completed. This philosophy of breaking the task into smaller pieces inspired by the micro productivity, as explained by Cummings (2018), will be used throughout this project. Breaking the tasks can make them a lot easier to achieve and makes them much more flexible.

There will be lots of behaviours that will need to be implemented which the AI should use to its advantage. Table 3.3 shows the behaviours that should be implemented.

Behaviour	How it will be achieved (steps)	Steps needed	Comments
Shoot the ball at the goal	Get close to the ball, position itself, use a sword	3	
Attack the opponent with a sword	Find the opponents position, rotate to look at the opponent, move towards the opponent, use a sword	4	
Attack the opponent with a gun	Find the opponents position, look at the opponent, use a gun. Additional steps may be needed if the player has no ammo or needs to reload – these should however be done automatically in the `UseGun()` method – or if the player has to switch weapons.	3 to 6	
Defend the goal using shield	Find the balls position, look at it, use a shield.	3	The shield should not be used when the player is facings its' team goal since it can risk an own goal.
Defend the goal not using the shield	Find the position of the ball, move towards it, and position itself so the ball is between opponents' goal and the player. Player could additionally use a sword to kick the ball away.	3	
Get a pickup box	Check if there are any pickup boxes around, check if any of them are needed (for example if the player has 100HP there is no need to pick up a health box), check if it is close enough, move to its position	3	It is important to make sure that the pickup box is not too far, otherwise the player is risking that the opponents team will score a goal

Table 3.3 - Behaviours that should be implemented

3.5.3 - LOGIC

The next step after creating the basic behaviours is to find a way in which the AI will choose the most effective behaviour in a given moment. This can also be done by using behaviour trees which are a way to structure the switching between various tasks easily. This means that each decision will have to be broken down into smaller parts.

The first step is to look at the behaviour more generally, as seen in Table 3.3 a basic behaviour is broken down into three or four smaller tasks, to keep things simple, similar approach will be used. The general goals of the player are to score a goal, defend it, make it harder for the opponent to score and keep HP above 0.

The AI player cannot realistically try to do all of those tasks at the same time so it should do them based on the position of the ball. For example, it should try to score a point when the ball is close to the opponents' goal. The chances that the opponent will score a goal decrease the further away the ball is from the player's goal and the chances that the player will score a goal increase the closer the ball is to the opponents' goal. All of these states and positions are relative to the player that the

Artificial Intelligence is controlling and it is assumed that the player starts on the right side (Collendachise and Ogren, 2018).

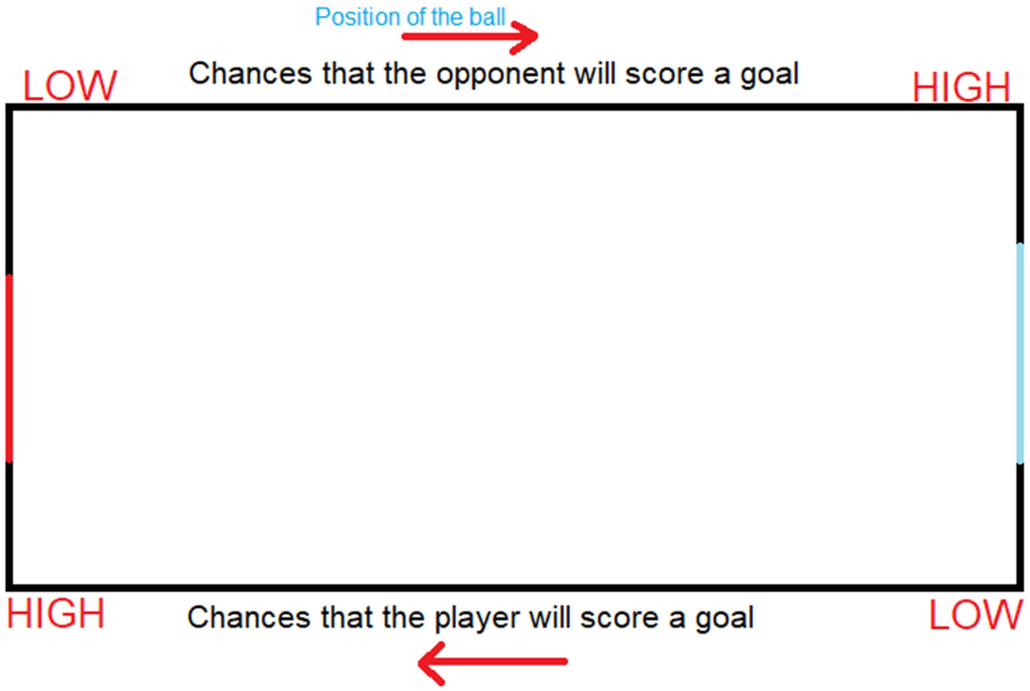


Fig. 3.14 - How the position of the ball affects the chances of scoring

This means that the position of the ball can be used to help decide which one of the four tasks should be prioritized. So the play area has to be broken down into four states, these states are Shoot, Attack, Tackle and Defend. These will be the first step in the behaviour tree.

3.5.3.1 - THE FOUR STATES

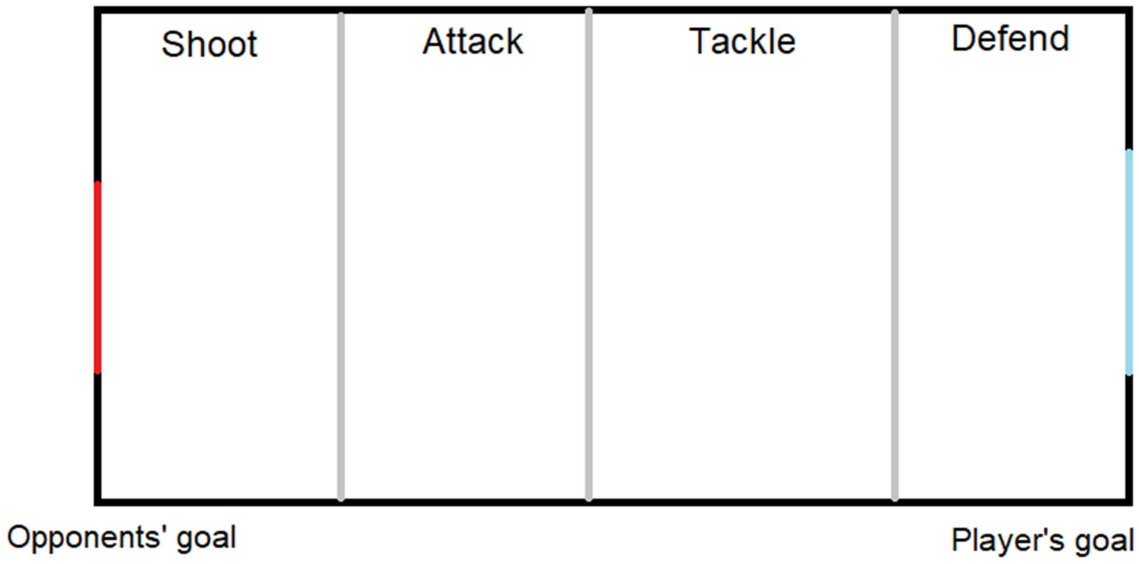


Fig. 3.15 - The four states

3.5.3.1.1 SHOOT

In the shoot state the player should focus on scoring the goal by using any means necessary. The main point of focus would be to position itself in a position that will give him the most chances of scoring a goal. It should not focus on getting pickup boxes unless its health is really low and he is really close to it.

3.5.3.1.1 - ATTACK

In this state, the most important task will be to take the ball away from the player, attack the player to drain his health points and get a pick up box if needed. This state is more flexible than the shoot state because there are low chances that the opponent will score a goal.

3.5.3.1.2 - TACKLE

Similarly to the attack state, player has much more freedom although it should be more careful because the chances that the opponent will score a goal are higher. In a game with more than one player on one team, one of the team mates should start to position itself and get ready to defend the goal while the other player should try to kick the ball away towards the opponents' goal.

3.5.3.1.3 - DEFEND

This state along with the shoot state are the most crucial for the outcome of the game because they decide which of the players will score the most points. In this state the main task should be to defend the goal and make it as hard as possible for the opponent to score a goal. Players' main task should be to use the shield to block the goal and sword to kick the ball away from the goal. Unless the player has really low health points, he should not try and get pick up boxes.

3.5.3.2 - FOUR POSITIONS

Next step would be to find a way to break down the tasks even further. Here the position of the player and the opponents could come in helpful. Since it makes no real difference where the players and the ball are positioned on the y axis, it will be ignored. Therefore the AI player should behave differently depending on its x position relative to the ball, opponent and the opponents' goal.

These states are named Position Zero, Position One, Position Two and Position Three. Each of these states will prioritize a different behaviour just like the first four states did.

3.5.3.2.1 - POSITION ZERO

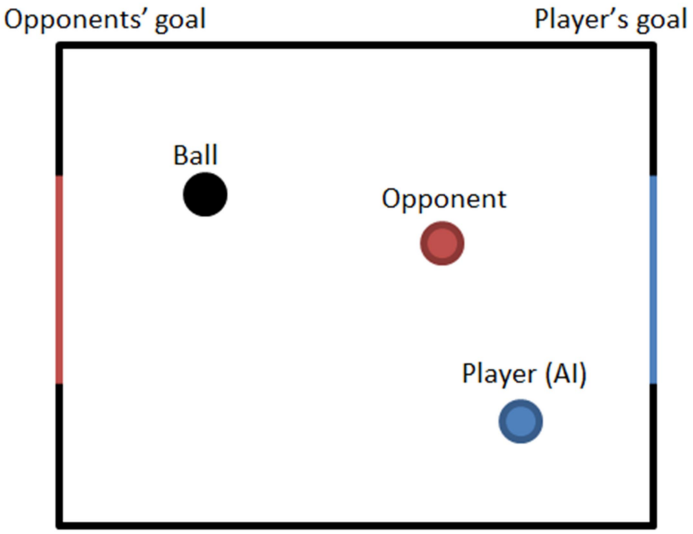


Fig. 3.16 - Position zero

This position is advantageous for the player because there is no obstacle between the ball and the opponent's goal which makes it easy for it to score a goal. Depending on the state, the player in this position could focus on scoring a goal or, if the ball is in the defend position; could be able to quickly grab a pick up box or attack the opponent with no risk.

3.5.3.2.2 - POSITION ONE

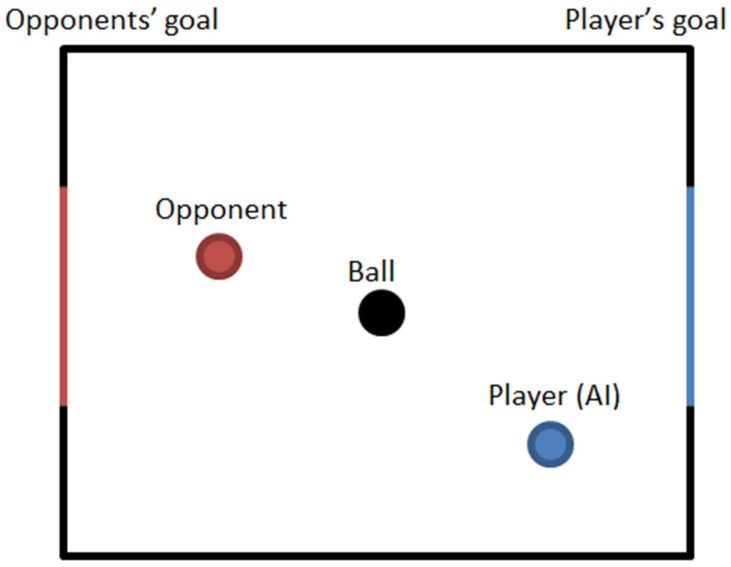


Fig. 3.17- Position one

This position is neutral for the opponent and the player because the opponent and the player cannot score a goal. Usually, the most effective task to do would be to try turn around.

3.5.3.2.3 - POSITION TWO

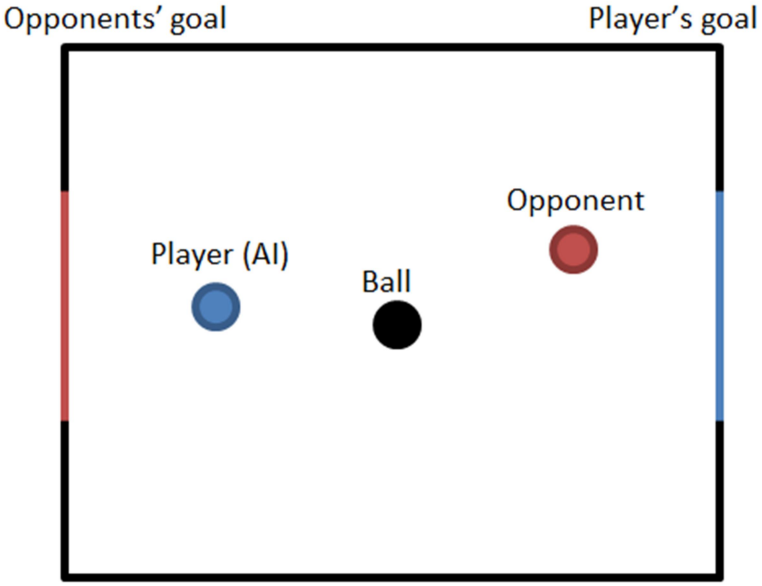


Fig. 3.18 - Position two

This position is similar to the Position One but this time both the player and the opponent are at risk. This position is the most neutral and what the player does is heavily dependent on the state that the ball is at. For example, in a Defend State the player should try to use the shield to defend its own goal but in the Attack State it should try to score a goal.

3.5.3.2.4 - POSITION THREE

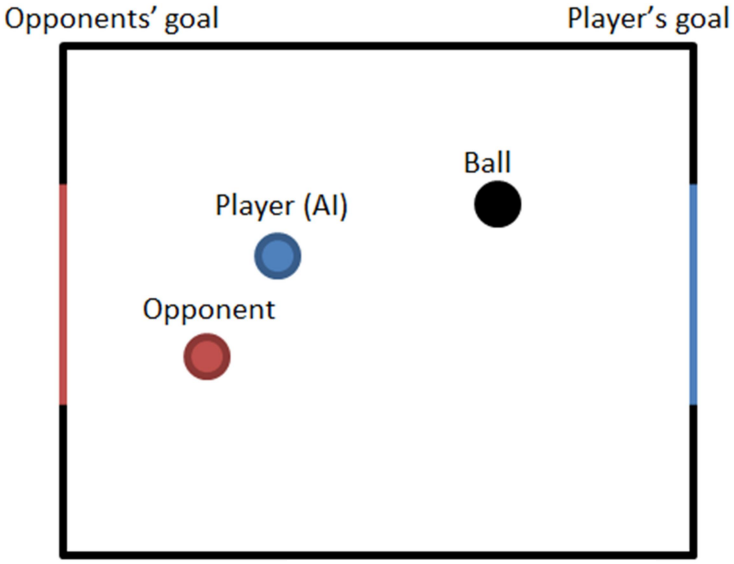


Fig. 3.19 - Position three

The last position is the most advantageous for the opponent because there are no obstacles between the player's goal and the ball so the opponent can easily kick it, even from a far, and score a goal. The player in that position should almost always try to get close to the ball and get between it and its own goal, i.e. get into the Two Position.

3.5.3.3 - CHOICE

To cover almost every possible situation the player may find itself in, there is one more issue to consider. That is the distance from the player to the ball; its behaviour should be slightly different if the player is far from ball. For example, if the ball is in the shoot area, and the player is in the defend area like shown in Figure 3.20 instead of just sprinting to the ball and risking that the opponent will kick the ball away, the player may use a gun to try to shoot the ball inside the goal from a distance.

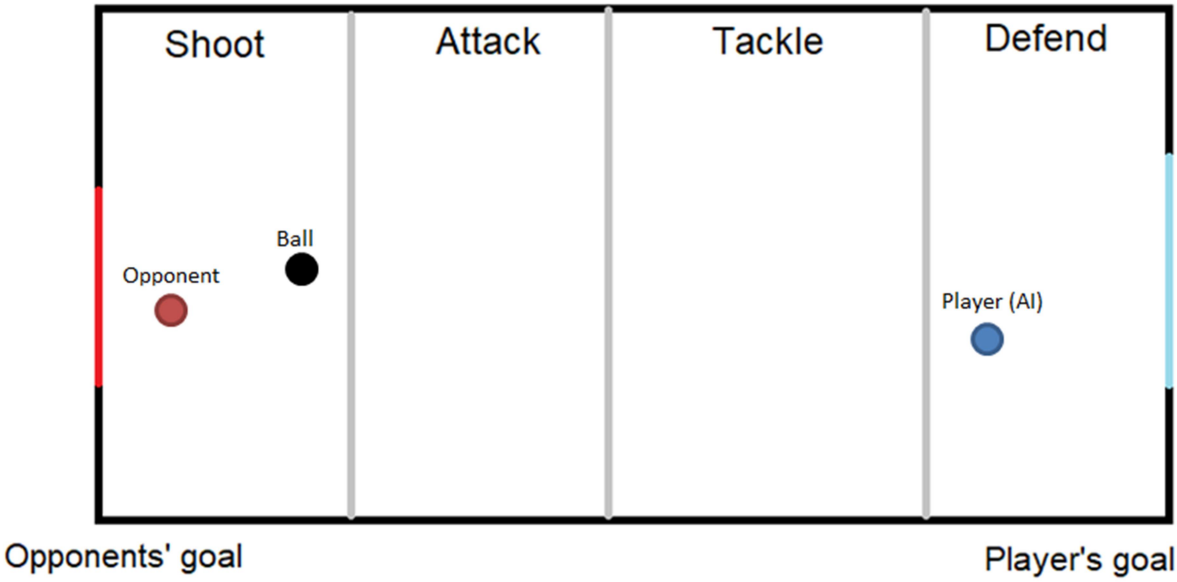


Fig. 3.20 - Player in the position one, far from the ball

Similarly, if the ball is in the defend area, the player is in the shoot area and the opponent is close to the ball, the player may attack the opponent with a gun to try and drain his health to stop him from scoring a goal.

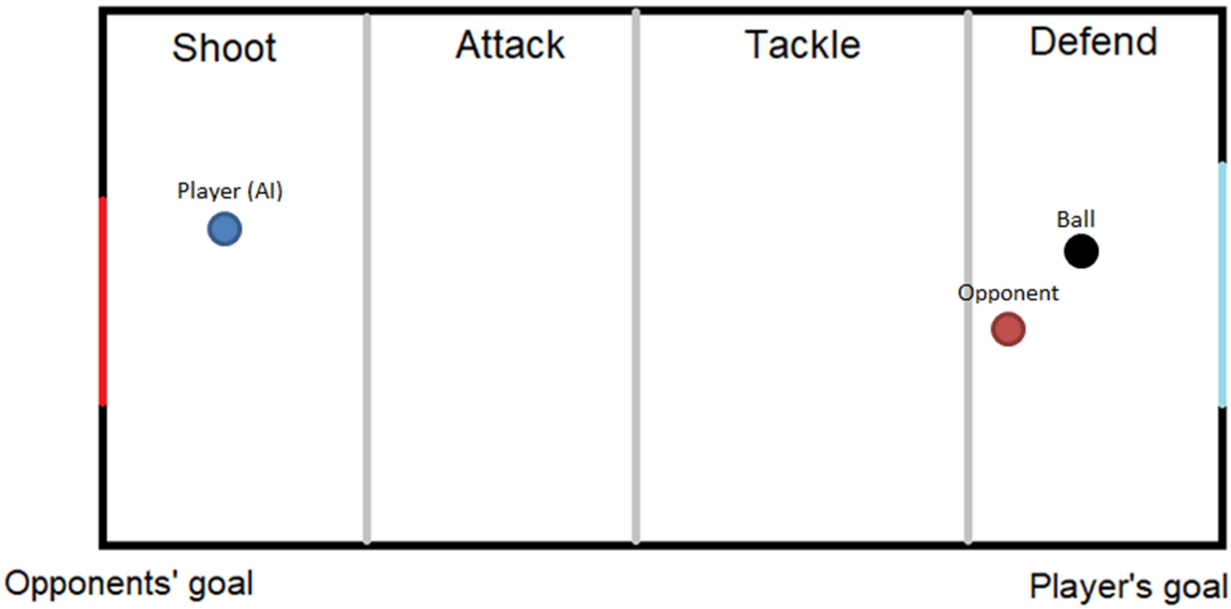


Fig. 3.21- Player in the position three, far from ball

The full process of choosing behaviour is shown in Figure 3.22 and as it can be noted it is kept basic to make sure the AI can quickly choose its behaviour.

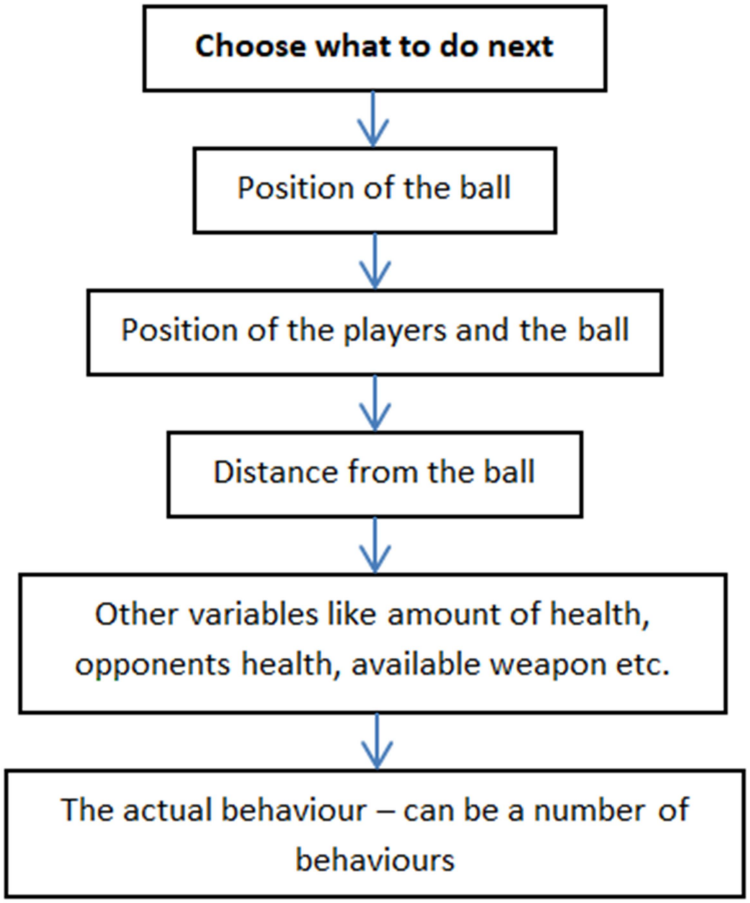


Fig. 3.22 - Decision tree for the AI's behaviour

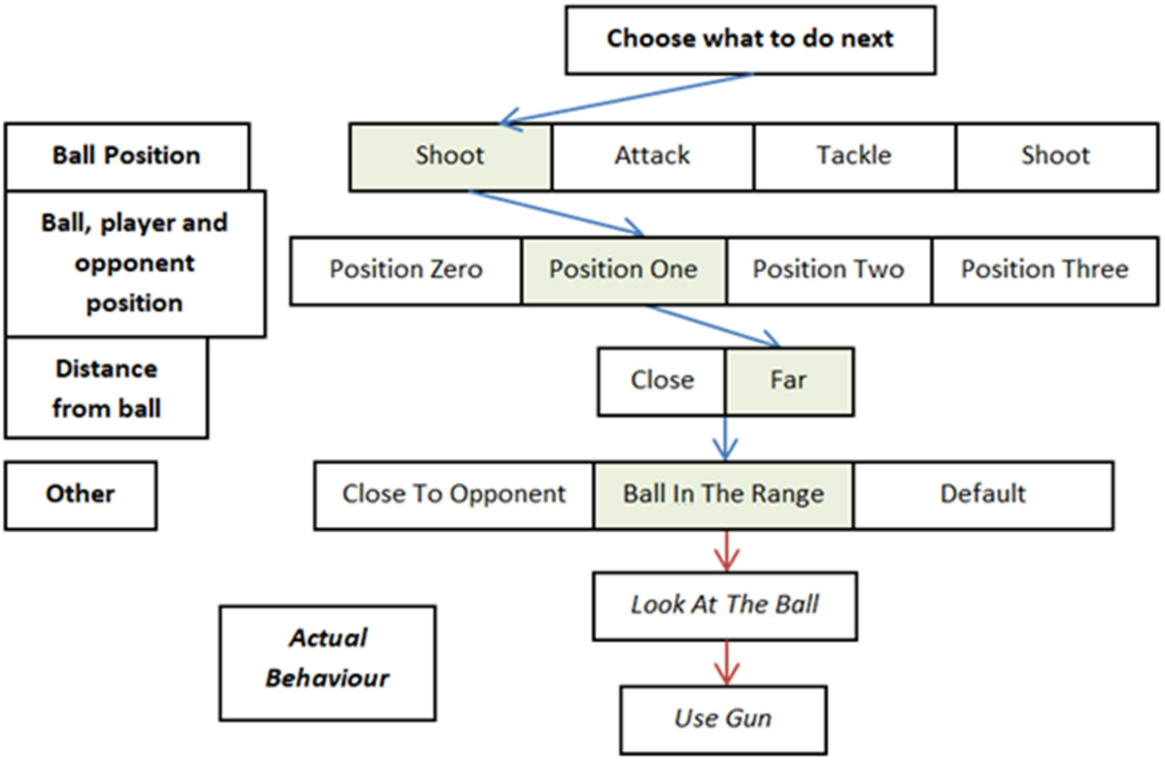


Fig. 3.23- How the behaviour would be chosen for Figure 3.20.

CHAPTER 4 – IMPLEMENTATION

This chapter is broken down into two parts. The first part explains the implementation of the players, game mechanics, physics and the user interface. The second part explains the implementation of the Artificial Intelligence – the behaviour and the logic.

4.1 - PART ONE

The implementation process started first by creating the player, playing field, goals, ball, and then lastly weapons, team systems, pickups and user interface. Figure 4.1 shows the final look of the game.

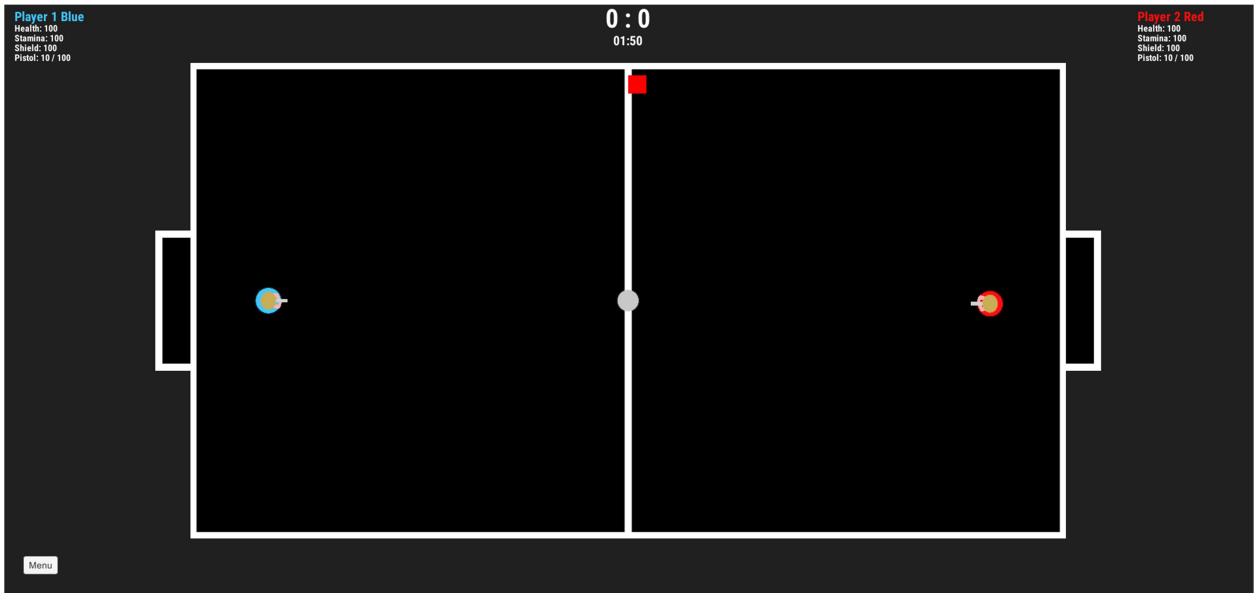


Fig. 4.1 - Game window

4.1.1 - PLAYER

Player object is the most complex because there is much more information needed to be passed through it than in any other object. It also has more functionality that is needed to make it work as shown in the Design Chapter. Fig. shows the components inside the player game object but the components break down into few main areas – controls, collision, weapon behaviour, game behaviour and lastly, the audio on which this project will not focus.

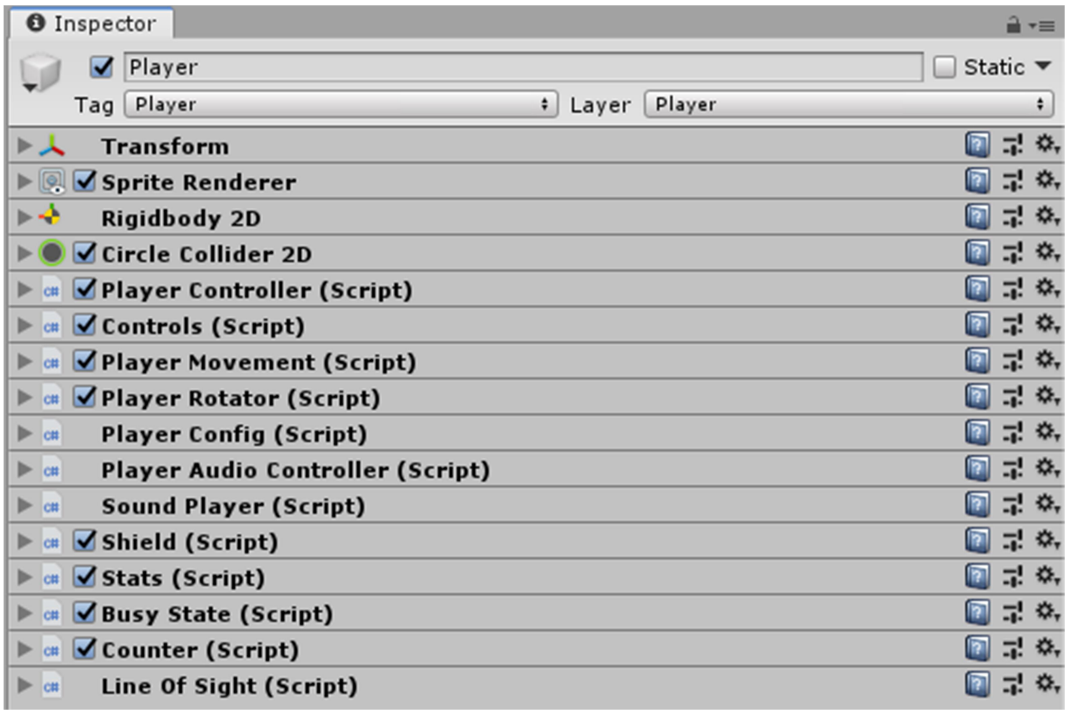


Fig. 4.2 - Player object in Unity

4.1.1.2 - COLLISION

As seen in Figure 4.1 the player sprite is circular therefore Circle Collider2D component have been used. It takes care of full collision between the player and other objects. Along with Rigidbody2D it gives a very convincing impression that the object has weight and bounciness. Lots of testing has been done to find the best settings of mass and drag to give the best feeling possible. The player has to move fast but not too fast that the game is impossible to control, and it has to feel like it has weight when turning so it does not feel unnatural to move the player around.

4.1.1.3 - PLAYER CONTROLLER

This is the main component that has some basic methods used by other components but its main task is to act like a point from which other components can reference other game objects or components that relate to the player object.

4.1.1.4 - CONTROLS, PLAYER MOVEMENT AND PLAYER ROTATOR

4.1.1.4.1 - CONTROLS

A player object can be controlled in two main ways; by Artificial Intelligence or by a joystick/keyboard. This is controlled by a Boolean value `aiControlled` inside the Controls. If that value is true, the `Update()` method will not be processed and the controls will be automatically picked up by the AI game object.

```
28      void FixedUpdate()
29      {
30          if (!aiControlled)
31          {
32
33              float sprint = Input.GetAxis(playerConfig.sprintKey);
34              float moveHorizontal = Input.GetAxis(playerConfig.horizontalL);
35              float moveVertical = Input.GetAxis(playerConfig.verticalL);
36              // Movement
37              playerMovement.MovePlayer(moveHorizontal, moveVertical);
```

Code Fragment 4.1 – Getting input from the gamepad

As explained in a previous chapter, the controls have been split in two. User can control the player using a left stick and rotate with right, or do both with the left stick. Boolean variable `aiControlled` has been added which omits the controls if the AI is enabled for this player object. Fragment 4.1 shows the movement part and the 4.2 shows how the object is rotated. For both, `Input.GetAxis` is used, the button from which the axis is taken is saved, as it can be seen in lines 32-35, inside the playerConfig (Figure 4.3).

```
44              float lookHorizontal;
45              float lookVertical;
46              if (useLeftToRotate)
47              {
48                  lookHorizontal = moveHorizontal;
49                  lookVertical = moveVertical;
50              }
51              else
52              {
53                  lookHorizontal = Input.GetAxisRaw(playerConfig.horizontalR);
54                  lookVertical = Input.GetAxisRaw(playerConfig.verticalR);
55              }
56              playerRotator.Rotate(lookHorizontal, lookVertical);
57
58              // Sprint
59              playerMovement.Sprint(sprint);
```

Code Fragment 4.2 – Getting rotation from the user

4.1.1.4.2 - ROTATION

Rotation works in a fairly basic way, an angle is calculated by first finding the arctangent of the coordinates, converting it to degrees using the `Mathf.Rad2Deg()` method and then by converting it to a Quaternion rotation using the `AngleAxis()` method and lastly by using the `Quaternion.Slerp()` method which by the definition in Unity manual, spherically interpolates between two points by using a time parameter. `Quaternion.Slerp()` can be really useful because the speed in which the player rotates can be adjusted so the game looks and feels much smoother than if the player object was 'snapped' into the wanted position – `rotateSpeed` variable in Fragment 4.3, line 31.

```
28
29          float angle = Mathf.Atan2(v, h) * Mathf.Rad2Deg;
30          Quaternion rotation = Quaternion.AngleAxis(angle, Vector3.forward);
31          transform.rotation = Quaternion.Slerp(transform.rotation, rotation, playerController.globalSettings.rotateSpeed * Time.deltaTime);
32
33      }
```

Code Fragment 4.3 – Calculating player rotation

4.1.1.4.3 - MOVEMENT

Movement, compared to rotation is very straightforward; RigidBody2D objects have a method called `AddForce()` which is used to add force to an object continuously. The first part of the movement method checks if the movement is in the `deadzone`, if the joystick is only tilted very lightly there is no need to move the player object because this can mean that the joysticks rest position is not exactly 0, 0 but can be something like 0.013, 0.09.

```
61      public void MovePlayer(float moveHorizontal, float moveVertical)
62      {
63          /* moveHorizontal and moveVertical would be a vector line in the wanted
64           * direction. Joystick X and Y axis are used to get the vectors. Keyboard
65           * arrows could also be used to get that information.
66           * Deadzone is used to avoid moving the character with a very
67           * small movement of the joystick.
68           */
69
70          float h = moveHorizontal;
71          float v = moveVertical;
72          if (moveHorizontal < deadZone && moveHorizontal > -deadZone)
73          {
74              h = 0;
75              movingHorizontal = false;
76      }
77          else { movingHorizontal = true; }
```

Code Fragment 4.4 – Setting up the dead zone

The `deadzone` value (Line 72, Fragment 4.4) can be adjusted for the best feel but it should not be high because this could make the player movement feel disjointed.

```
89          Vector2 movement = new Vector2(h, v);
90          rb2d.AddForce(movement * speed * Time.deltaTime);
```

Code Fragment 4.5 – Moving the player object

4.1.1.4.4 - SPRINT

`Sprint()` is a very basic method which checks the player's stamina status and if the player has enough stamina, it changes the speed value by adding the sprint power which is stored in the Global Settings. It also drains the player's stamina level (See Appendix for the full method).

```
101         if (stats.GetStaminaStatus() > 0 && !recharging)
102         {
103             SetSprintingStatus(true);
104             speed = defaultSpeed + (defaultSpeed * (globalSettings.sprintPower * (int)sprint) * -1);
105         }
106         else
107         {
108             speed = defaultSpeed;
```

Code Fragment 4.6 – Player sprint method

4.1.1.5 - PLAYER CONFIG

The next two important components are Player Config and Stats. Player Config holds the information about the keys that are used for all of the movement like rotation, using swords etc. It does not hold the actual key but the name of the variable that Unity stores in Project Settings -> Input, Figure 4.3 shows how the movement keys are translated. Each key can be then easily set up inside the Unity.

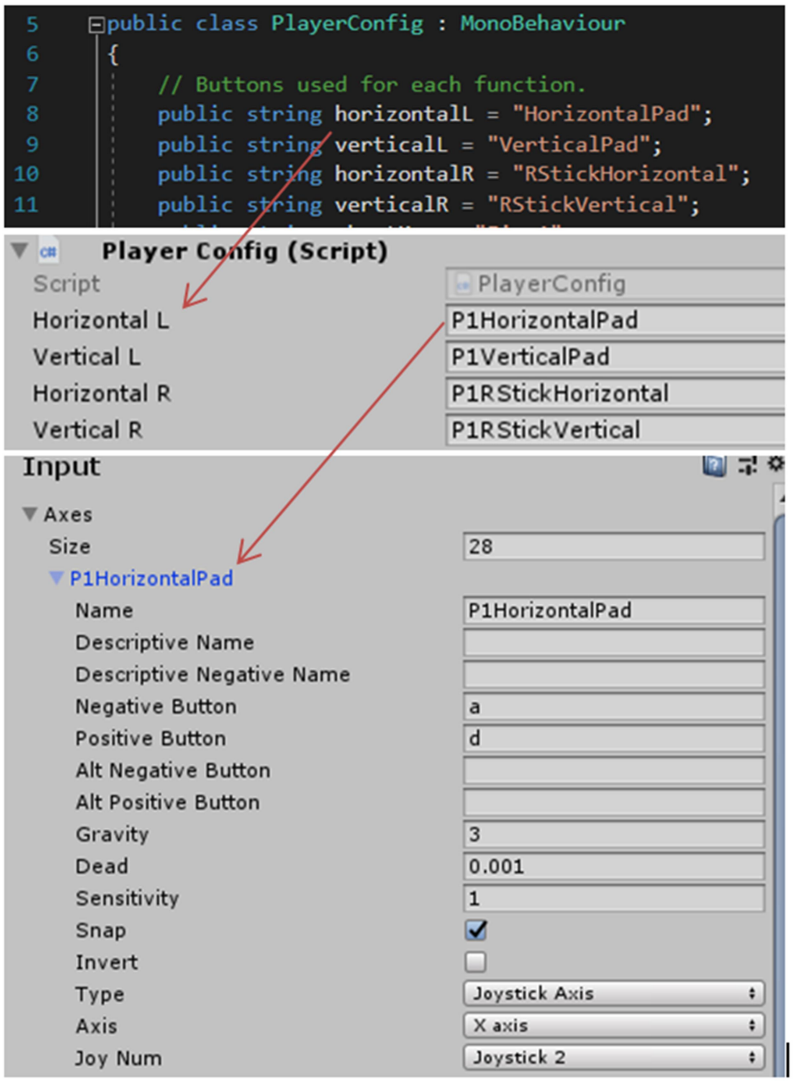


Fig. 4.3 - How the code relates to Unity

4.1.1.6 - BUSY STATE

Busy state and Counter are a helper components used by other objects. Busy state is used to block weapons to be used when the shield is up by going into `busyState` for its duration. Counter is used by AI explained in the next part, it has a value that goes from 0 to 1 in 0.01 steps then it restarts.

4.1.1.7 - SHIELD

The shield component takes care of activating and deactivating the Shield and its basic functionality, for this reason, its main controller has been kept inside the player game object. Its functionality is very basic, if a `Shield` variable is more than 4, it is set active, -4 is added to it every frame, and the objects' size is changed based on that value. When it reaches 0, the shield game object is deactivated.

4.1.1.8 - STATS

The last component is Stats, which has references to the weapons and holds the health, stamina and shield value. It is also responsible for initializing weapons and replenishing health.

The project does not focus on audio and visuals therefore the Audio Controller and Sound Player were ignored.

4.1.2 - WEAPONS

4.1.2.1 - ABSTRACT GUN

Weapons game object is a child of the player game object. It has three components – weapon controller, weapon list and the shooter. Weapon list component is a list with all of the gun game objects and it makes it easy to add new guns. Guns are held as children of the weapons game object.

```
10      private List<AbstractGun> guns;
11      public AbstractGun pistol;
12      public AbstractGun machineGun;
13
        2 references
14      public void Start()...
        1 reference
22      public AbstractGun GetNextGun(AbstractGun previousGun)...
        1 reference
33      public AbstractGun SetGun(string gunName)...
        2 references
44      public AbstractGun GetDefaultWeapon()...
        0 references
52      public List<AbstractGun> GetGuns()...
        2 references
56      public void AddAmmo(string gunName, int amount)...
```

Code Fragment 4.7– Weapon List

Weapon controller component is a proxy of the actual gun game object and the `WeaponList`. If a player changes its gun, a `NextGun()` method is run inside the `WeaponController` (line 29, Fragment 4.8), which runs a `GetNextGun()` inside the `WeaponList` (line 22, Fragment 4.7) which returns a gun. Weapon controller holds the equipped gun inside the `currentGun AbstractGun` variable which points to the gun game object.

```
22      public void Shoot()...
        1 reference
29      public void NextGun()...
        1 reference
34      public void SetGun(string gunName)...
        2 references
39      public AbstractGun GetCurrentGun()...
        4 references
43      public void ShowCurrentWeapon(bool b)...
        1 reference
47      public void Reload()...
        1 reference
51      public int GetCurrentGunAmmo()...
```

Code Fragment 4.8– Weapon Controller

To add a new gun, a new object has to be created that is a child of the Weapon game object as seen in Figure 4.4 with the `AbstractGun` component inside. Once new gun has been added to a `WeaponList`, it will be available for the player to use.

`AbstractGun` is a gun base class that holds all of the default functionality of a basic gun. Each gun has to have `AbstractGun` component to make it a gun. Its basic functionality is `Shoot()` and `Reload()` and is made in such a way that each new gun has to be set up inside the Unity as seen in Figure 4.4.

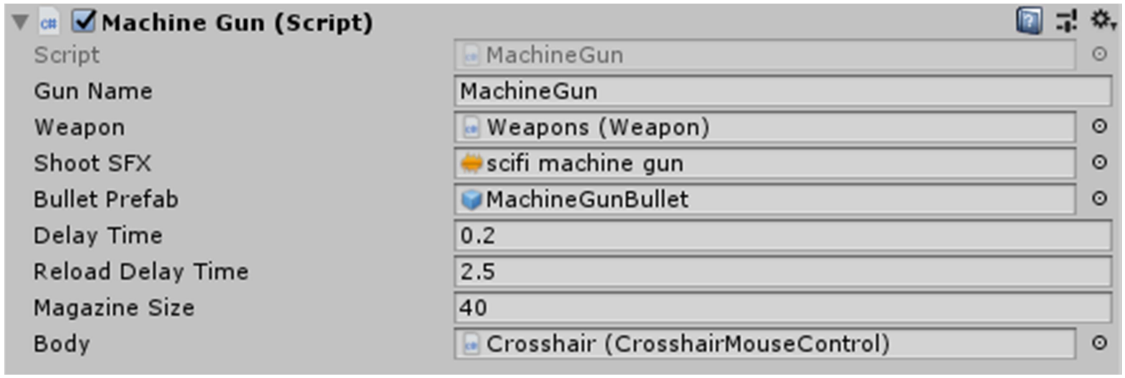


Fig. 4.4 - Machine gun object – inherits AbstractGun

```
47    public void Reload()...
72
      3 references
73    public virtual void Shoot()...
      2 references
96    public void Delay()...
      1 reference
110   public void DelayFor(float time)...
      0 references
116   public bool CheckIfReloading()...
      0 references
120   public bool CheckIfReady()...
      2 references
124   public void AddAmmo(int amount)...
      2 references
128   public int GetAmmo()...
      3 references
132   public string GetGunName()...
      0 references
136   public string GetAmmoString()...
      2 references
140   public override string ToString()...
      1 reference
151   public void InitializeGun()...
```

Code Fragment 4.9– Abstract gun component

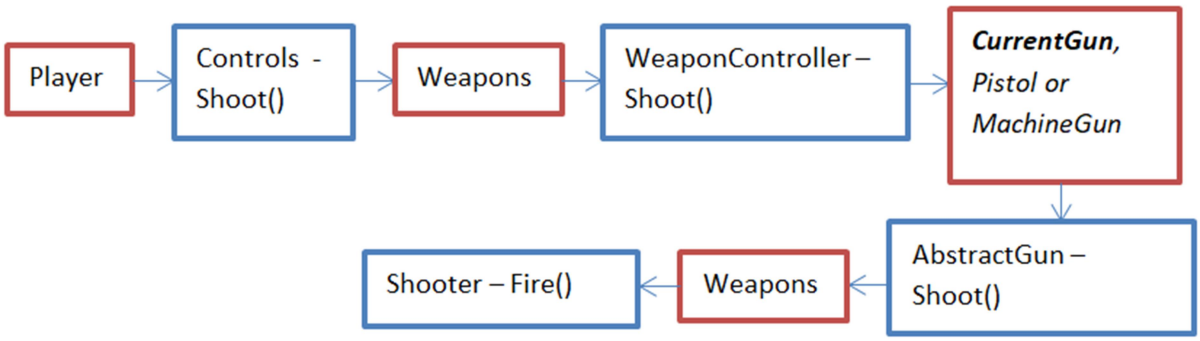


Fig. 4.5 - What happens when the player uses a Shoot method

First, the Controls reference the Weapons' method and run a `Shoot()` method inside the `WeaponController`, it then runs a `Shoot()` method on currently equipped by the player gun. Next the `Shoot()` method is run inside the `AbstractGun`, this method has the most complex behaviour. If there are not enough bullets inside the available ammo, it will call the Reload method and the Shoot method again, and if it has enough bullets it will call a `Fire()` method in the shooter class which takes care of Instantiating bullets as seen in Fragment 4.10 line 84.

```
73      public virtual void Shoot()
74      {
75          /*
76           * If there is enough bullets in the magazine, shoot.
77           * Otherwise reload which takes care of the issue when there is
78           * no more bullets left.
79           */
80          if (availableAmmo > 0 && readyToShoot)
81          {
82              body.Pullback(0.5f);
83              weapon.soundPlayer.PlaySound(weapon.soundSource, shootSFX);
84              weapon.shooter.Fire(bulletPrefab, weapon.firePoint);
85              availableAmmo -= 1;
86              readyToShoot = false;
87              countdown = delayTime;
88          }
89          else if(availableAmmo == 0 && ammo > 0)
90          {
91              // Automatic reload.
92              Reload();
93              Shoot();
```

Code Fragment 4.10– Shoot method

The bullets are instantiated based on the position of the `FirePoint` object which is a child of the player object, so its position and rotation changes based on the position of the player as seen in Figure 4.6.

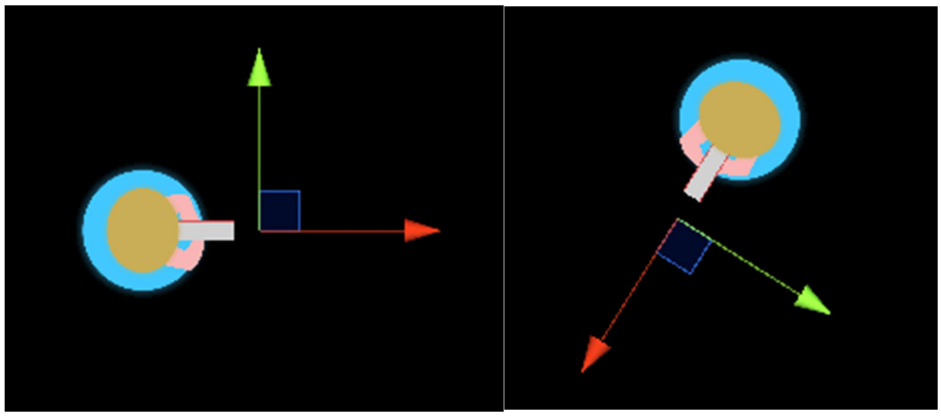


Fig. 4.6 - Fire point position when the player rotates

4.1.2.2 - BULLET PREFAB

Bullets are Prefabs – a reusable asset stored inside the projects Prefab directory, complete with all of its components and values. This makes it easy to duplicate them and customize each bullet for each gun. Just like other objects, they have Rigidbody2D and CircleCollider2D. Each bullet has a speed and a strength variable. Strength variable determines how hard it pushes another object, can be seen in line 40-41, fig, hence the manual Bounce method.

```
16      private void OnTriggerEnter2D(Collider2D collision)
17      {
18          if(collision.gameObject.CompareTag("Ball"))
19          {
20              Bounce(collision);
21              Destroy(gameObject);
22          }
```

Code Fragment 4.11– Bullet prefab colliding with a ball

```
37      void Bounce(Collider2D collision)
38      {
39          // Push the given object
40          float x = rb.velocity.x * (strength / 100);
41          float y = rb.velocity.y * (strength / 100);
42          Vector2 force = new Vector2(x, y);
43          collision.attachedRigidbody.AddForce(force);
44      }
```

Code Fragment 4.12– Bounce method used on the ball in Fragment 4.11 (l.20)

`Reload()` method checks if player has enough bullets and adds them to the magazine accordingly, (l.54-64, Fragment 4.13). It also runs a `DelayFor(reloadDelayTime)` method, which delays when the player is able to shoot again. The reload time adds customizability to the weapons. The available ammo and the magazine are held inside the AbstractGun component.

```
54          if (availableAmmo != magazineSize)
55          {
56              DelayFor(reloadDelayTime);
57              reloading = true;
58              if (ammo > magazineSize)
59              {
60                  int aAmmo = availableAmmo; // Ammo already in the magazine
61                  availableAmmo += magazineSize - aAmmo;
62                  ammo -= magazineSize - aAmmo;
63              }
64              else if (ammo > 0 && ammo < magazineSize)
65              {
66                  int aAmmo = availableAmmo;
67                  availableAmmo += ammo - aAmmo;
68                  ammo = aAmmo;
69              }
70          }
71      }
```

Code Fragment 4.13– Reload method

4.1.2.3 - SWORD

A sword is a game object with a Circlecollider2D and Sword Controller component. The `UseSword()` method inside the Sword Controller has a very basic task – check if the player has enough stamina (as it has been explained in the previous chapter, using the sword drains stamina) lines 33-35, check if player is not busy, then if not, hide the current weapon for the duration. Line 36, activates the sword sprite and creates an invisible circle collider that will push the ball in line 40.

```
30      public void UseSword()
31      {
32          float stamina = playerController.stats.GetStaminaStatus();
33          if (!playerController.busyState.GetState() && stamina > 5)
34          {
35              playerController.stats.AddStamina(-5);
36              weapon.ShowCurrentWeapon(false);
37              swordSprite.gameObject.SetActive(true);
38              playerController.body.Pullback(0.4f);
39              playerController.busyState.SetState(true);
40              Push();
```

Code Fragment 4.14– UseSword method

`Push()` method is a little more complicated, the collider is kept inside the Sword object, the sword is a circle game object with a circle collider, its position is moved forward and it is scaled up which creates a pushing effect when it collides with other objects. For this reason sword sprite is kept inside a separate object. Figure 4.7 shows what happens to the collider (green circle) when the sword is used.

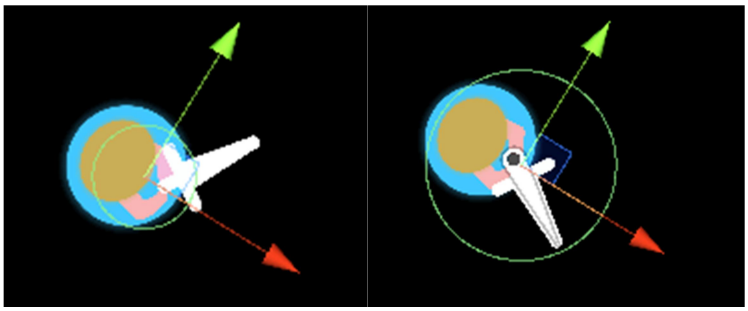


Fig. 4.7 - Fire point position when the player rotates.

```
45      private void Push()
46      {
47          /*
48           * This function creates an invisible collider and pushes it in front of the player,
49           * it is needed to create velocity so when it collides with the ball
50           * it will push it back. The force (bounciness) with which it pushes the ball can be configured in the
51           * 'Physics' folder, in the file 'Sword'.
52           * To make sure the ball is only pushed when it is infront of the player,
53           * or when it is in the hit area.
54           */
55          float x = this.transform.localPosition.x;
56          float y = this.transform.localPosition.y;
57          // Move it at a given rate
58          this.transform.localPosition = new Vector2(x + 0.6f, y);
59          // Now scale it
60          x = this.transform.localScale.x;
61          this.transform.localScale = new Vector2(x+2f, 0);
62          if (this.transform.localPosition.x >= 3)
63          {
64              // Reset the position and the scale
65              this.transform.localPosition = new Vector2(0, 0);
66              this.transform.localScale = new Vector2(0, 0);
67              playerController.busyState.SetState(false);
68              weapon.ShowCurrentWeapon(true);
69              // The sword sprite is not needed anymore
70              swordSprite.gameObject.SetActive(false);
```

Code Fragment 4.15– `Push()` method

As seen in Fragment 4.15, line 37, the sword sprite is activated. The sword sprite is a basic game object, animated using the Unity's Animator to rotate around the hilt.

4.1.3 - PLAYING FIELD

To 'build' the walls in the main game area, a number of Box Colliders have been attached to the Football Pitch object. After testing the game a bug has been found that the ball would sometimes get 'stuck' next to the wall because it would stop too close to it and there was no way to bounce it back – it would end up 'sliding' on the wall. Therefore, Circle Colliders have been added at the corners to get rid of this issue.

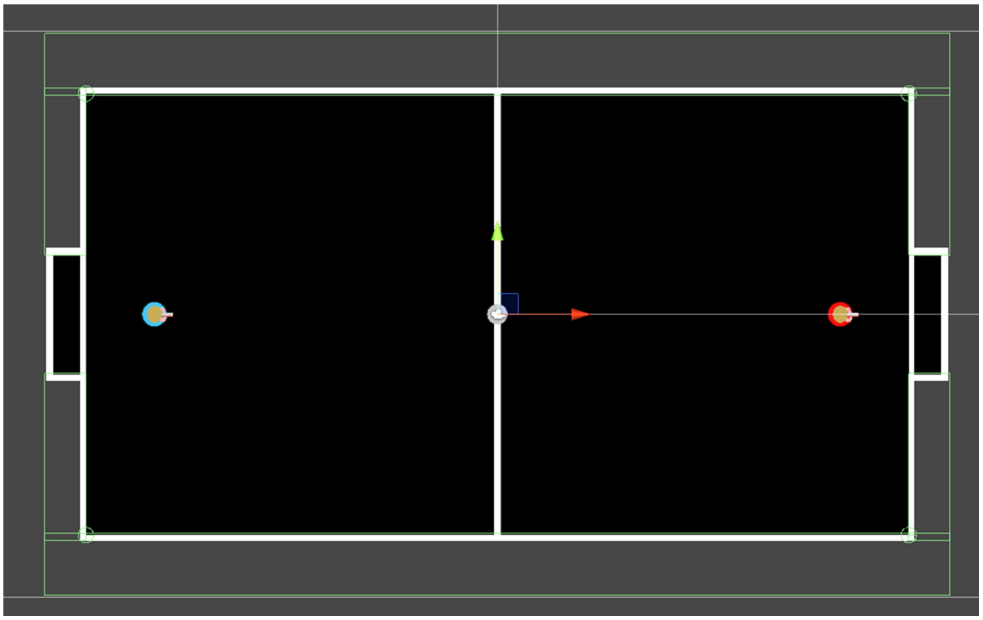


Fig. 4.8 - Colliders (Green) on the playing field.

4.1.3.1 - THE GOALS

The goals are very similar to the playing field but they are made as separate objects, they needed to have different tags (in Unity tags are one of the variables that describe an object, just like the Name) so it would be easier to write a method that recognizes which goal the ball has hit and which team should get a point. The collider in Figure 4.9 is used to check if the ball is all the way inside the goal, seen in Fragment 4.16.

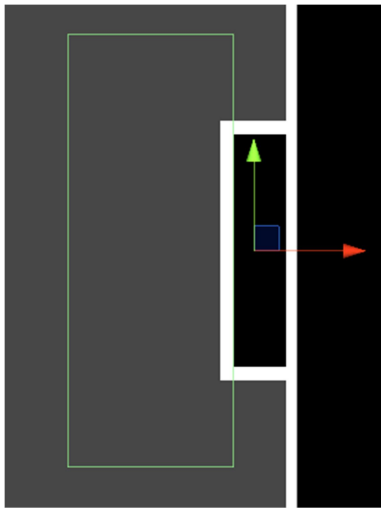


Fig. 4.9 - Goal Collider

4.1.3.2 - THE BALL

The ball has a circle collider and Rigidbody2D just like the player object. The Rigidbody2D takes care of simulating the physics of the ball – the ball had to have a much smaller mass than the player object so the player can push it. It also has a BallBounce Material which sets its friction and the bounciness, so the object behaves like a ball.

```
36      private void OnTriggerEnter2D(Collider2D other)
37      {
38          if (other.gameObject.CompareTag("LeftGoal"))
39          {
40              redTeam.GetComponent<TeamController>().AddPoint(1);
41              ResetPlayers();
42          }
```

Code Fragment 4.16– When the ball collides with the left goal

4.1.4 - TEAMS

Teams have been implemented just like planned – Team game object has been created with two game objects, one for each team, that hold the `TeamController` component. `TeamController` takes care of the players inside the team, the score, opponents and the goals. It has some additional methods like `ResetPlayers()` which changes all of its players to their default positions which are held inside the `GlobalGameSettings`. It also holds information about the State Boundaries which are the places where the AI changes its state.

4.1.5 - USER INTERFACE

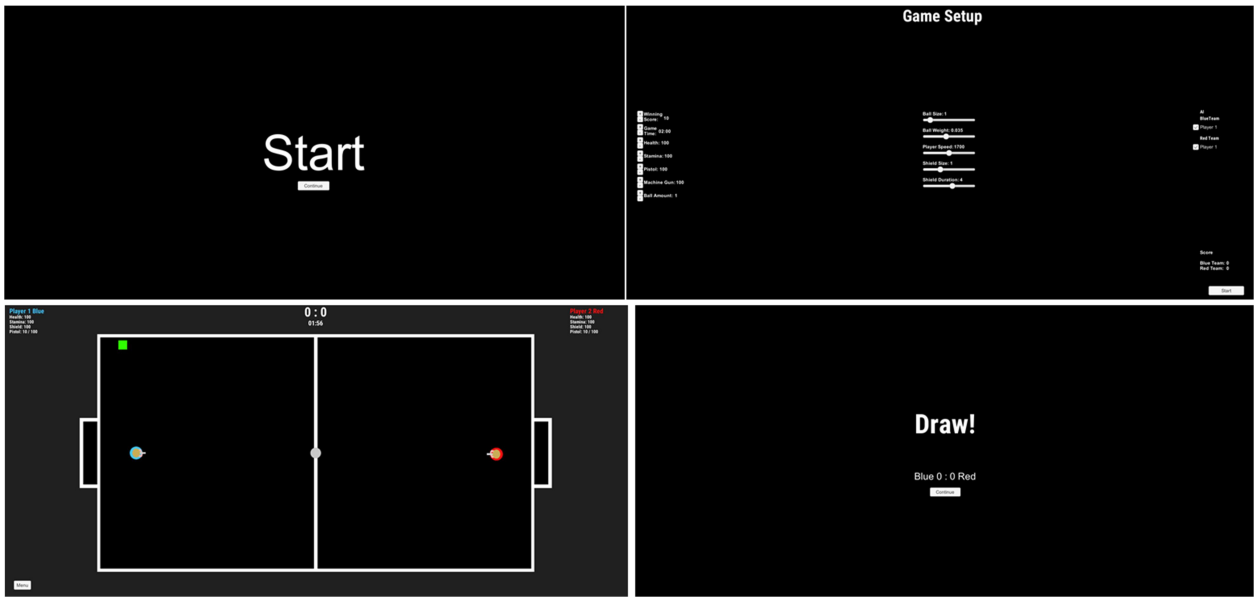


Fig. 4.10 - Four game scenes – Start, Game Setup, Main Game and Win Screen

4.1.5.1 - HUD

The first User Interface that has been implemented is the Head-Up Display (HUD) seen in the top left and right corner, of the third game scene in Figure 4.10. The information that was meant to be shown to the user as mentioned in the Design chapter has been implemented but had to be simplified. Instead of a rectangle or a bar, a numerical value is shown for the health, stamina, shield and ammo. Unfortunately there was not enough time to have two separate HUDs for both game modes. Instead, one of the sides is for the first player and the other, right side is for the second player/CPU. It was useful to do it that way when implementing the game because it allowed seeing what the AI's status is.

The HUD has been implemented in a very basic way – a Canvas and Event System object has been created and default Text objects have been added as a child of the Canvas object. The Text objects allow positioning the text with the Rect Transform component and to add text by using the Text script component. The settings have been kept default with only the font and font size changed.

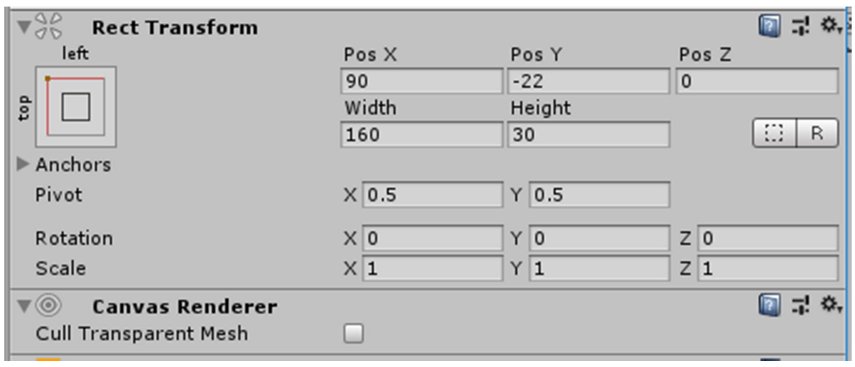


Fig. 4.11 - Rect Transform component used to position the object

To show the score, a new game object has been created with a Score UI Controller component inside. It takes both `TeamController` components, extracts the score into a string and passes them to a Text component.

4.1.5.2 - STATS

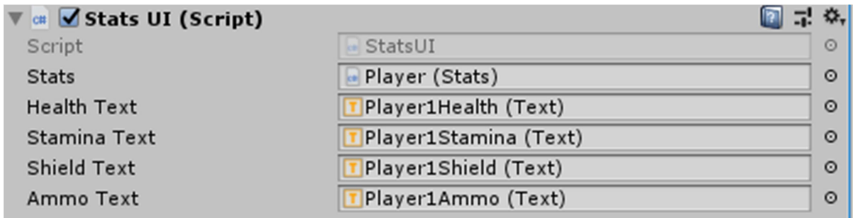


Fig. 4.12 - Stats UI object.

Stats have been done similarly, a `StatsHUD` object with two StatsUI components for each player have been created. StatsUI is very similar to Score UI Controller but it takes `StatsUI` object, which gives all of the player's statistics like health or stamina, and turns them into a string and passes to Text objects accordingly. Figure 4.12 shows a Stats UI object with everything set up; as it can be seen it is very general and can be easily duplicated in case the teams are expanded.

```
14      void Update()
15      {
16          healthText.text = "Health: " + stats.GetHealthStatus();
17          staminaText.text = "Stamina: " + stats.GetStaminaStatus();
18          float shield = 100 - stats.GetShieldStatus();
19          shieldText.text = "Shield: " + shield;
20          ammoText.text = stats.weapon.GetCurrentGun().ToString();
21      }
```

Code Fragment 4.17– Converting stats into a string

4.1.6 - THE GAME SETUP

The game settings seen in panel two in Figure 4.10, have been set up very similarly to above, update method creates a string from the status info that it takes from the `GameStartSettings` component. Each button is made up of various objects and components. As seen in Figure 4.14 a button that changes the start health has four game objects under it – two text objects and two buttons. First Text object is used to show the name of the variable that is being changed, in this case the health, and the second Text is used to show the current value of that variable.

`GameStartSettings` component is a static class that holds the information of the game as static variables. This was implemented that way because information will have to be passed between the game scenes and while Unity gives an option to pass information between the scenes, this was the fastest and easiest way to implement it. It plays a role of a Configure file.

Each text is going through the Start Settings component inside the `StartSettings` game object. This component takes care of building strings out of the values inside the `GameStartSettings` game object and it takes care of all of the methods used by the buttons.

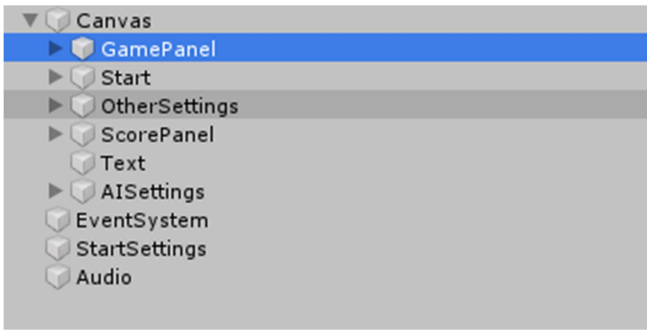


Fig. 4.13 - Game setup objects

The methods are very basic because their only role is to either add or subtract from the value inside `GameStartSettings`. Fragment 4.18 is an example of that method, as it can be seen, a value is passed to the method which is then added to the given variable. Fig 34 shows how that method is used inside the button, Unity has a very straightforward way to pass a value to the method it is calling. `AddHealth()` method is called with 1 passed to it, in the Add button and the same method is called to subtract from that variable but -1 is passed to it.

```
56      public void AddHealth(int health)
57      {
58          GameStartSettings.health += health;
59          if (GameStartSettings.health < 10) {GameStartSettings.health = 10;}
```

Code Fragment 4.18– Example of a method used inside a button

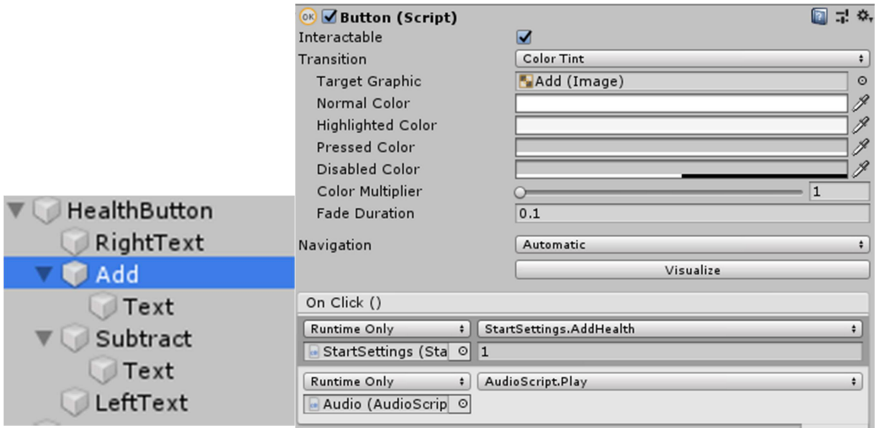


Fig. 4.14 - Add button component

Some objects have a slider instead of two buttons to fit the type of a variable. Slider allows users to select a numeric value by pressing on a button and dragging it left or right. The slider component allows calling a method when the value changes, change the minimum and maximum value or set the starting value. Line 104 what is run when the slider is changed.

```
        0 references
102     public void IncreaseBallWeight ()
103     {
104         GameStartSettings.ballWeight = GameObject.Find("BallWeightSlider").GetComponent<Slider>().value;
105     }
```

Code Fragment 4.19– Example of a method used inside a slider

4.2 - PART TWO – ARTIFICIAL INTELLIGENCE

4.2.1 - BEHAVIOURS

Artificial Intelligence has been implemented in the same way as planned – there are a number of behaviours, the play area is divided into four parts and the precise behavior is based on the position of the players. The behaviours have been broken down into groups, and each behavior is implemented as a static method inside a Static Class. Implementing behaviours as static methods seemed like a best way because it allowed separating the part that takes care of the logic with the part that does an actual method. This part will first explain the behaviours the AI can choose from and then will go on to explain the logic behind it.

The behaviours are separated as follows; `BasicBehaviour`, `DirectionBehaviour`, `GlobalBehaviour`, `MovementBehaviour` and `TeamBehaviour`. There are also helper methods; `Calculate`, `General` and `Check`.

4.2.1.1 - BASIC

`BasicBehaviour` Class has a set of methods that are no more than two lines. Their tasks are just the basic controls like using a sword or a shield, swapping guns or sprinting. Code fragment 4.20 shows the first method, `GetController()` which takes out the Controls component from the player `GameObject`.

```
7   public static Controls GetController(GameObject player)
8   {
9       return player.GetComponent<Controls>();
10  }
```

Code Fragment 4.20– A method that takes the Control component from the player game object

Fragment 4.21 shows one of the methods inside the Basic Behaviour class. While this is very basic, it makes the game code easier to read since these methods will be used often. It changes `player.controls.UseShield()` into `AIBasicBehaviour.UseShield(player)`

```
15  public static void UseShield(GameObject player)
16  {
17      GetController(player).UseShield();
18  }
```

Code Fragment 4.21– An example of a Basic method

4.2.1.2 - MOVEMENT

The player AI movement should be broken down into a number of steps. The first step is to either rotate the player character towards the wanted object or to find an angle or a direction vector towards the wanted object. The next step would be to move towards that object.

`MovementBehavior` Class takes care of all things that have to do with player movement and player rotation. These methods are slightly more complex than the `BasicBehaviour` but their tasks are simple – rotate to 'look' in a given direction or at an object and move towards an object, a given direction, or up or down. Most of the methods are already written inside the player objects, like a method to rotate, which is inside the `PlayerRotator` component in a player game object

or a method to move like a method to move the player towards x and y direction, inside the Player Controller component.

```
7   public static void LookAt(GameObject player, GameObject o)
8   {
9       /* This method makes the player look at the object.
10       * This is usually used to make the player look at the ball.
11       * First, the position of the object we want to look at, in regards of the player
12       * is found and then the player is rotated.
13       */
14      float x = o.transform.position.x - player.transform.position.x;
15      float y = o.transform.position.y - player.transform.position.y;
16      AIBasicBehaviour.GetController(player).playerRotator.Rotate(x, y);
```

Code Fragment 4.22– Finding coordinates of an object in relation to the player

The move methods are very simple, the player can move forward in the same direction it is facing or direction vectors can be given to move in a wanted direction.

```
33      Vector2 dir = AIDirectionBehaviour.GetRotationVector(player);
34      AIBasicBehaviour.GetController(player).playerController.MovePlayer(dir.x, dir.y);
```

Code Fragment 4.23– Moving the player forward by using its rotation value

```
104     double radians = player.transform.eulerAngles.z * Mathf.PI / 180;
105     float x = Mathf.Cos((float)radians);
106     float y = Mathf.Sin((float)radians);
107     return new Vector2(x, y);
```

Code Fragment 4.24– Finding the rotation vector

A more complex method inside this class is the `MoveTowards()`, which is used to move the player towards raw xy coordinates (that can be found by using `GameObject.transform.position`) or towards an object.

```
71  public static void MoveTowards(GameObject player, GameObject destination)
72  {
73      float x = destination.transform.position.x;
74      float y = destination.transform.position.y;
75      MoveTowards(player, x, y);
76  }
```

Code Fragment 4.25– `MoveTowards(object)` method

The first method however, is much more complicated because the x and y coordinates have to be converted to a direction vector. Fragment 4.26 shows the `MoveTowards()` method, as it can be seen there are three stages, first the radian towards the given x, y position is found, then that radian is converted to direction vector (l.68) and lastly, the player object is moved forward by using the `MoveForward()` mentioned in Fragment 4.25 (l.69).

```
61      public static void MoveTowards(GameObject player, float x, float y)
62      {
63          /* First the angle from the player to the given position needs to be found
64           * since the player does not necessarily has to look in the direction
65           * that it is moving.
66           */
67          double a = AIDirectionBehaviour.GetRadian(player, x, y);
68          Vector2 d = AIDirectionBehaviour.GetDirectionVector(a);
69          MoveForward(player, d.y, d.x);
```

Code Fragment 4.26– MoveTowards(coordinates) method

4.2.1.3 - DIRECTION – GUIDING THE PLAYER

4.2.1.3.1 - FINDING WHERE TO MOVE

Lines 67-68 introduce a new set of behaviors, stored in the `DirectionBehaviour` Class. This class takes care of finding direction and rotation vectors, radians and the position of a direction vector. There are four main methods inside this class, `GetRadian()`, `GetDirectionVector()`, `GetRotationVector()` and `FindPositionOf()`. These are used to guide the player towards a ball and to make sure it positions itself in a way that it can shoot at the goal easily.

The first step as explained in the Design chapter (c. 3.5.1) is to create the player movement. It is the most important step because without the AI movement, the player will not be able to score goals. Movement has been broken down into a number of steps. The first step is to move towards a ball, this is achieved by using the `MoveTowards()` which takes a player game object and a ball game object. Next, a new point has to be found where the player should position so that it faces an opponent's goal. Figure 4.15 shows how that position should look like and where the player should move towards – the green dot. The arrow should point towards the goal at all times.

This position is found by using the three methods explained above; Fragment 4.27 shows the first version of this code which returns it as a `Vector2`.

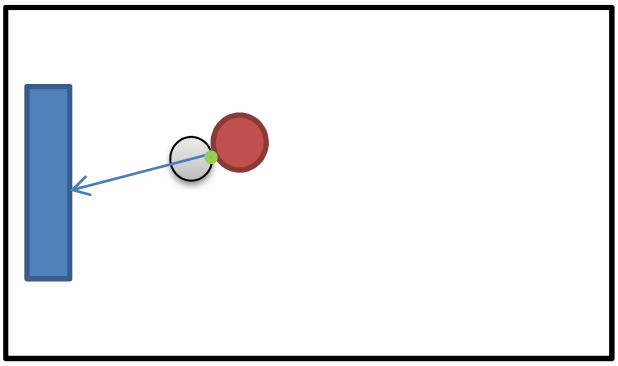


Fig. 4.15 - Green – point towards which the player should move

```
92      public static Vector2 GoRoundTheBall(GameObject ball, GameObject goal)
93      {
94          double goalAngle = AIDirectionBehaviour.GetRadian(ball, goal, 1);
95          Vector2 des = AIDirectionBehaviour.GetDirectionVector(goalAngle);
96          return AIDirectionBehaviour.FindPositionOf(ball, des);
97      }
98  }
```

Code Fragment 4.27– Finding the point shown in Figure 4.15

The first part to find that position is to find the Radian using the `GetRadian()` shown in Code Fragment 4.28. First the position of the destination in regard of the source is found (lines 16-17), then it is converted to an angle by using the `Mathf`, Unity's mathematics library. First part uses the inverse tangent equation and `Rad2Deg` converts it into degrees (l.18). Then, degrees are converted to a quaternion (l.19) from which the radian is then extracted (l.21).

```
7       public static double GetRadian(GameObject source, float x, float y, float offset = 1)
8       {
9           /* It is the first part in converting an angle from source object to another
10          * object to an actual position around the source object.
11          * This method calculates the angle from the source object to the destination
12          * by first finding the position of the destination in regards of the source,
13          * then converting it to an angle. Once the angle is found, it is converted
14          * to a Quaternion from which a radian is extracted by using the euler angles.
15          * This method should only be used in GetVectors' method. */
16          float destinationx = x - source.transform.position.x;
17          float destinationy = y - source.transform.position.y;
18          float angle = Mathf.Atan2(destinationx * offset, destinationy * offset) * Mathf.Rad2Deg;
19          Quaternion rotation = Quaternion.AngleAxis(angle, Vector3.forward);
20
21          double radians2 = rotation.eulerAngles.z * Mathf.PI / 180;
22          return radians2;
```

Code Fragment 4.28– Converting position to a radian

The next step is to convert the radian to a direction vector. This is achieved by using the `Mathf.Cos()` and `Mathf.Sin()` method which return the cosine and sine of an angle.

```
41          float x2 = Mathf.Cos((float)radian) * offset;
42          float y2 = Mathf.Sin((float)radian) * offset;
43
44          return new Vector2(x2 * size, y2 * size);
```

Code Fragment 4.29– converting a radian to a direction vector

The last method, `FindPositionOf()`, converts the direction vector from the source to coordinates.

```
73          float sourcex = source.transform.position.x - directionVector.y;
74          float sourcey = source.transform.position.y - directionVector.x;
75
76          return new Vector2(sourcex, sourcey);
```

Code Fragment 4.30– converting direction vector to coordinates

4.2.1.3.2 - IMPROVEMENTS

There is one issue with the code above, if the ball is between the player and the goal, the player object can move towards it but if the player is between the ball and the goal, it would keep pushing the ball away from itself. The next step would be to find a way to go around the ball; this can be achieved by adding few more steps into the code.

First, an additional point has to be found – point that faces the player. This can be achieved in the exact same way as above but the radian has to be inversed since now it does not have to be between the player and the ball. Figure 4.16 shows two points, green and red, green shows the point facing the goal and the red point shows the point facing the player.

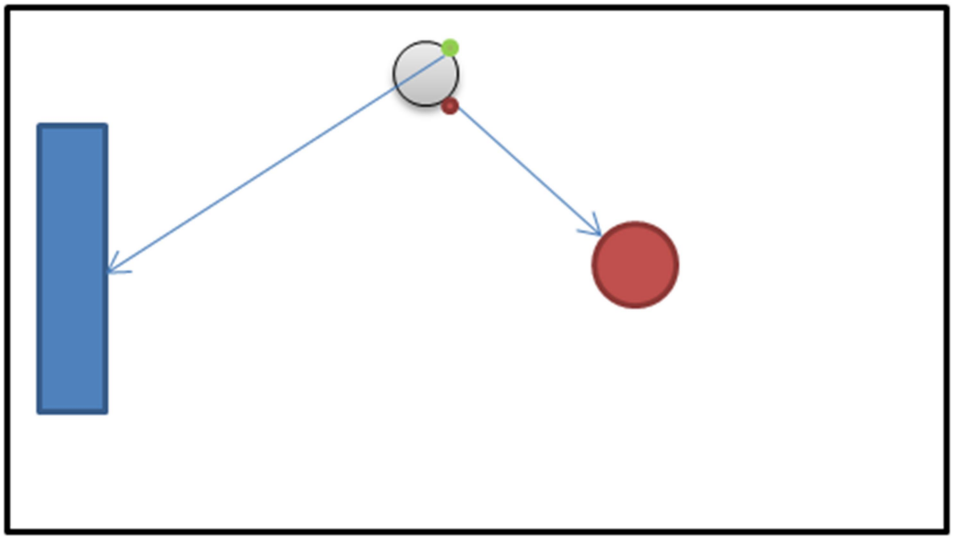


Fig. 4.16 - Green – goal position, red - where the player is positioned

```
34      public static Vector2 FindPointBetween(GameObject player, GameObject ball, GameObject goal)
35      {
36          /* First find the angle from the ball to the player (playerAngle),
37           * and from the ball to the goal (goalAngle).
38           * These points are used to guide the player around the ball thus, a counter
39           * value is needed, in this case r that goes from 0 to 1.
40           * The position of the point in which the player should move will keep circling
41           * the ball. Imagine an arrow a few pixels away from the ball, playerAngle would make it
42           * point in the direction of the player, and goalAngle would make it point
43           * to the goal (we want to move player to the other side of that point so the ball is
44           * between the player and the goal). */
45
46          double goalRadian = AIDirectionBehaviour.GetRadian(ball, goal, 1);
47          double playerRadian = AIDirectionBehaviour.GetRadian(ball, player, -1);
48          float counter = player.GetComponent<Counter>().a;
49
```

Code Fragment 4.31 - Finding a radian of the points from Fig. 4.16

Next step would be to guide the player to the green point. This can be achieved by creating a new point, which would go around the ball. To create that point, green and red points have to be combined, and multiplied by a counter method that would go from 0.1 to 1 – at 0.1 the new point would be in the same position as the red point, at 0.5 it would be between red and green and at 1 it would be at green point which is the player destination.

$$newPoint = (redPoint * (1 - counter)) + (greenPoint * counter)$$

Equation 4.1 - Finding where the player should move towards

Since only one counter is needed and the class is static, it was easier to have the counter be a component of the player game object. The class is very simple as its only job is to cycle from 0 to 1 in a steady step (See Appendix for a full Counter method).

```
56          Vector2 playerDes = AIDirectionBehaviour.GetDirectionVector(playerRadian);
57          Vector2 playerPos = AIDirectionBehaviour.FindPositionOf(ball, playerDes);
58          if (playerPos.x > player.transform.position.x && playerPos.y > player.transform.position.y)
59          {
60              playerRadian += 6.3;
61          }
62          float newRadian = (float)playerRadian * (1 - counter) + (float)goalRadian * counter;
```

Code Fragment 4.32–Combining both radians

After testing and visualizing the new point, it was found that the point would circle the ball in a wrong way because the start and end point of the ball was at the bottom. Figure 4.17 shows the old start point, if it was kept like that, the player character would have to circle almost whole ball to position itself. Figure 4.18 shows its new path.

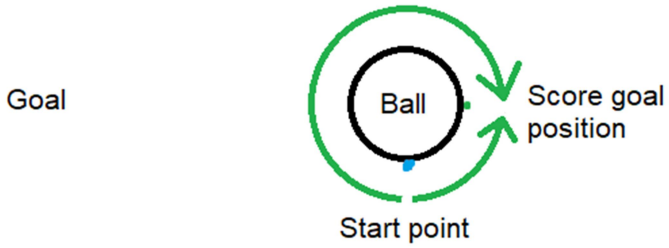


Fig. 4.17 - Old start point

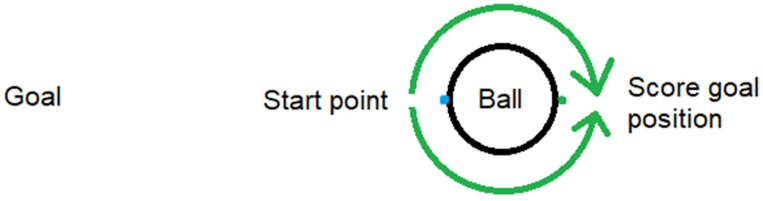


Fig. 4.18 - New start point

This method returns a Vector2 with a position of the point towards which the player should move towards. So, the next step would be to guide the player towards that vector. This is done in Code Fragment 4.32, lines 58-61.

The last step is finding the position of the point by getting the direction vector from the new radian calculated in equation 4.1, seen in line 62.

```
63          Vector2 des = AIDirectionBehaviour.GetDirectionVector(newRadian, 1.2f);
64          Vector2 pos = AIDirectionBehaviour.FindPositionOf(ball, des);
65          return new Vector2(pos.x, pos.y);
```

Code Fragment 4.33–The last step of `FindPointBetween()` method

```
22          Vector2 coordinates = AICalculate.FindPointBetween(player, ball, goal);
23          AIMovementBehaviour.MoveTowards(player, coordinates.x, coordinates.y);
```

Code Fragment 4.34– Implementation of the above method, `PositionInFrontOf()`

`PositionAndShoot()` is a slightly improved version of the above method, as it also makes the player look at the ball and use a sword to shoot it towards the goal.

```
32          AIMovementBehaviour.LookAt(player, ball);
```

Code Fragment 4.35– First part of the method

There is no need to calculate the points mentioned above until the player is close to the ball to make sure the performance is not affected, so `goalDirection` is the point shown in Figure 4.18.

```
27  public static void PositionAndShoot(GameObject player, GameObject ball, GameObject goal)
28  {
29      /* First a position of the goal is found, then player is rotated to face the ball.
30      * Next distance from the player to the point from which he would be able to shoot is found.
31      * Then the player is moved closer to it or uses a sword to shot at the goal. */
32      AIMovementBehaviour.LookAt(player, ball);
33
34      Vector2 goalDirection = AIDirectionBehaviour.FindPositionOf(ball, goal, 0.9f); // The point where player should position itself
35      if (AICalculate.CalculateLengthBetween(player, ball.transform.position.x, ball.transform.position.y) < 2) // If player is close to the ball
36      {
```

Code Fragment 4.36– The second part, checking if the player is close enough

Then once it is close enough (l.35), it is positioned by using the `PositionInFrontOf()` shown in Code Fragment 4.34, or it uses a sword.

```
37          if (AICalculate.CalculateLengthBetween(player, goalDirection.x, goalDirection.y) > 0.3)
38          {
39              PositionInFrontOf(player, ball, goal);
40          }
41          else
42          {
43              AIBasicBehaviour.UseSword(player);
44          }
```

Code Fragment 4.37– Moving closer or using a sword

The last method that needs explaining is the `CalculateLengthBetween()` used in the above methods. This method uses the distance formula which is show in equation 2. Code Fragment 4.38 shows how it has been implemented. (See appendix for a full list of methods)

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Equation 4.2 - Calculating the distance between two objects

```
8   public static float CalculateLengthBetween(GameObject o1, GameObject o2)
9   {
10      /*This method calculates the length between two objects.
11       */
12      float x1 = o1.transform.position.x;
13      float y1 = o1.transform.position.y;
14      float x2 = o2.transform.position.x;
15      float y2 = o2.transform.position.y;
16
17      return Mathf.Sqrt((Mathf.Pow((x2 - x1), 2) + Mathf.Pow((y2 - y1), 2)));
```

Code Fragment 4.38– Calculating the distance between two objects in Unity

Name	Task
`CheckIfCloseToPickup()`	It goes through each pickup inside a passed list, calculates the distance between it and returns a bool value based on the results.
`GetClosestPickup()`	This method checks if there are any active pickups in the game, checks which one is the closest and returns its position as a vector.
`CheckIfFarFromBall()`	This method calculates the length between the player and the ball, and if the distance from ball is larger than allowable distance, from Global Settings and returns a Boolean value based on the results.
`CheckIfCloseToOpponent()`	Check if the distance between first game object and second game object is 2 and returns a bool.
`CheckOpponentsHealth()`	Check if the opponents health is lower than 20.
`CheckIfBallApproaching()`	Check if the ball is close to the player by checking that the distance is between 2 and 3.

Table 4.1 – Other behaviours

4.3 - THE LOGIC

As it can be seen, each of the above behaviors creates a basic tree - each behavior builds upon other to create a more complex behavior. Similar approach will be used for the logic as explained in the Design chapter. Since time was limited, there was not enough time to refactor the code to take most of the advantages of behavior trees – a more basic approach has been taken. This part explains how the basic set of behaviors is chosen by splitting the playing field into four areas, and later goes into more detail how it chooses a smaller set of behaviors based on the position of the ball until it finally chooses a concrete behavior.

4.3.1 - FOUR STATES

As explained in the design chapter (Figure 3.15), there are four states – Shoot, Attack, Tackle and Defend. These states are of equal size, so naturally the play area is broken down first into two parts in the middle of the pitch and then, each is broken down again at around quarter of the pitch – the exact value is stored inside the `TeamController` component as an Integer value – `stateBoundary`. The logic, shown in Code Fragment 4.39 that chooses the behavior is stored inside the `AIController` component which main task is to run a correct state inside a `FixedUpdate()`. (See appendix for full class)

```
47          xPos = ball.transform.position.x;
48          if (xPos >= boundary)
49          {
50              defendState.Run();
51          }
52          else if (xPos < boundary && xPos > 0)
53          {
54              tackleState.Run();
55          }
56          else if (xPos < 0 && xPos > -boundary)
57          {
58              attackState.Run();
59          }
60          else if (xPos <= -boundary)
61          {
62              shootState.Run();
63          }
```

Code Fragment 4.39– Choosing the state

Each of these states inherits an `AIState` abstract class which has the necessary and references to objects. The most important methods are `Run()` (seen in Fragment 4.40, l.50-62), `Run [`*position number from zero to three]* `Position` and `RunDefault`. `RunDefault` is used as a safety measure in case everything else fails; it runs a `PositionAndShoot()` from the `GlobalBehaviour`.

4.3.2 - FOUR POSITIONS

The more important methods are the `Run()` and `RunPosition()`. `Run()` chooses the behavior that should be ran based on the player, opponent and the ball position as explained in the Design chapter. It uses a switch based on the position given by the `GetPositionStatus()`.

```
10      public new void Run()
11      {
12          int position = AIHelperMethods.GetPositionStatus(player, opponent, ball, goal);
13          switch (position)
14          {
15              case -1:
16                  RunDefault();
17                  break;
18              case 0:
19                  RunZeroPosition();
20                  break;
21              case 1:
22                  RunOnePosition();
23                  break;
24              case 2:
25                  RunTwoPosition();
26                  break;
27              case 3:
28                  RunThreePosition();
29                  break;
30          }
31      }
```

Code Fragment 4.40– Choosing the position

The `GetPositionStatus()` returns a number from 0 to 3, and -1 if the position has not been found. It uses logic equations to determine which number to return. For example, the zero position is when the ball is between the opponents' goal and the player. So, the propositions are:

G = Goal is on the left side of the ball,

LO = Opponent is on the left side of the ball,

LP = Player is on the left side of the ball,

LB = Ball is on the left side of the opponent and the player,

Zero Position: Goal is on the left side of the ball, Ball is on the left to player and the opponent

$$G \wedge (\neg LO \wedge \neg LP) => G \wedge LB$$

Equation 4.3 – Logic proposition for Position Zero

Fragment 4.41 shows the implementation of Zero Position. (See appendix for all implementations)

```
57          bool goalIsOnTheLeft = goalXPosition < ballXPosition;
58          bool aiIsOnTheLeftToBall = aiXPosition < ballXPosition;
59          bool playerIsOnTheLeftToBall = playerXPosition < ballXPosition;
60          bool ballIsOnTheLeftToPlayerAndAI = !playerIsOnTheLeftToBall && !aiIsOnTheLeftToBall;
61          bool ballIsOnTheRightToPlayerAndAI = playerIsOnTheLeftToBall && aiIsOnTheLeftToBall;
62          // 0
63          bool zeroPosition = (goalIsOnTheLeft & ballIsOnTheLeftToPlayerAndAI) || (!goalIsOnTheLeft & ballIsOnTheRightToPlayerAndAI);
64
65          if (zeroPosition)
66          {
67              return 0;
68          }
```

Code Fragment 4.41– Implementation of Zero Position

The other four `RunPosition()` methods that are inside each state run the behaviours explained in previous part. As was explained, there are four different positions, named Zero, One, Two and Three. If the time was not limited, these behaviours would be held inside a separate class or in an actual tree since the class clashes with the SOLID principles, like Single Responsibility Principle which says that every module should only have one responsibility. This was a faster way to implement that (LH, 2019).

4.3.3 - DISTANCE

The first and last thing that has to be checked inside each method is the distance from the ball. This is done by using the `CheckIfFarFromBall()` inside the `CheckBehaviour`. This follows the same principles as the States and `Run()` methods, this time the player should behave differently depending on the distance from the ball. For example, if the player is in a defend state and far from ball, it could use a gun to kick the ball away from its goal to make it harder for the opponent to score a goal. If it is close to the ball it could use a shield to block the opponent from scoring a goal. Code Fragment 4.42 shows a `RunZeroPosition()` inside the Defend State, as it can be noted, the behavior is basic but since the position of the ball changes very fast, it creates a dynamic behavior.

```
53      public new void RunZeroPosition()
54      {
55          /* OGoal | Ball | Player, AI | AIGoal
56           */
57          if (AICheckBehaviour.FarFromBall(player, ball))
58          {
59              AIBasicBehaviour.Sprint(player, 1);
60              AIMovementBehaviour.LookAt(player, ball);
61              AIBasicBehaviour.UseGun(player);
62              AIGlobalBehaviour.PositionAndShoot(player, ball, goal);
63
64          }
65          else
66          {
67              if (AICheckBehaviour.CheckIfBallApproaching(player, ball))
68              {
69                  AIBasicBehaviour.UseShield(player);
70              }
71              AIGlobalBehaviour.PositionAndShoot(player, ball, goal);
72          }
73      }
```

Code Fragment 4.42– `RunZeroPosition()` inside the Defend State

4.3.4 - AMOUNT OF BEHAVIOURS

There are four states, each state has four positions and each position has another two positions (not counting the behaviours inside each distance), there are a total of 32 behaviours.

$$4 * 4 * 2 = 32$$

Equation 4.4 – Total amount of behaviours

CHAPTER 5 – EXPERIMENTATION AND EVALUATION

This part goes through the analysis and the evaluation of the game.

5.1 - EVALUATION

The project is evaluated by letting a focus group play the game and answer a short survey. It was made sure that each person in the focus group had a similar experience of the game otherwise some answers may have been biased. Each of them played the game for at least fifteen minutes in two modes – couch co-op with a friend, and against the AI. The requirements to play the game had to be kept low; otherwise it would be very difficult to gather a group of people to play the game.

As the aim of this project was to create a game with a good overall experience, the first trend within the gathered data was to understand the overall experiences of those taking part in the research. Thus the first two questions asked about the overall experience of the game and how it made the user feels. The third question focused on the Graphical User Interface – how it feels and why. Fourth question asked about the difficulty of the Artificial Intelligence. The fifth question was broken into two parts, one of them was to rate the AI and the other part asked to elaborate. Those two questions referred to one of the main aims and objectives about the project which is the AI. The last two questions asked about the overall experience of the game – what people liked and disliked. Answers to each question have been evaluated one by one.

5.1.1 - OVERALL EXPERIENCE

The main aim of this project was to make the overall game experience enjoyable, fun and addictive, this means that the game should be stable and working. It should not have many bugs that would limit the game or make the game a chore to play. There seems to be an agreement between the participants that the game is exactly that – addictive and fun –, participant 1 (P1) said “I felt desperate to win and angry when I lost”, another participant said, P2 “Short games made it easy to get hooked on the game, and the 1vs1 mode got me feeling quite competitive … I could imagine myself spending some time playing it as it got me hooked on pretty fast”. One of the participants also echoed the point by saying “If you play one time, you want another round”.

About the overall experience P3 said that “I had a lot of fun playing the game. It makes you focus “, P2 said “Overall the game ran smoothly and I’ve enjoyed playing it”. Participants felt very good about the game’s overall experience and about the fun factor which was very important when designing the game. P4 showed that the game met its main aim “The game is fun, yet quite challenging which makes it enjoyable as it’s the perfect in-between”.

5.1.2 - GRAPHICAL USER INTERFACE

While the project did not focus a lot on the GUI, it was still a big aspect and one of the objectives, as it can greatly affect the game. So, one question, split into two parts has been designated to find out how participants felt about it. First part asked them to rate it based on how intuitive it was and another part asked them to elaborate. The average score was 1.8 with 1 being ‘very intuitive’ and 5 ‘very unintuitive’, which means that the project has met this objective.

Participant 1 said “The interface used a well-known, traditional layout, which all gamers are accustomed to”, P3 said “… it does seem pretty straightforward”, whereas, P4, said that “…seeing it for the first time it looks a little confusing because you don’t really know what and how the things will change in the game…”. It is important to remember that some of the participants had experience with video games and some were beginners, but it seems like there is an agreement that the UI is easy to use, however the project could be improved by slowly introducing each piece of UI instead of all at once how it was done to ensure clarity.

5.1.3 - AI (DIFFICULTY AND HOW IT PLAYS)

The most important aspect of the project was the AI, therefore two of the questions were mainly about the AI, and other four questions related to it – the participants were told to focus on the AI. First question asked them to rate how challenging was the AI with 1 being ‘very easy’ and 5 being ‘very hard’, the average score was 3.9. The difficulty may have affected how the participants felt about the game as not everyone likes difficult games. The project may be improved by giving a difficulty setting which makes the AI easier, this way everyone from various backgrounds would be able to enjoy the game fully. P5 said “… I lost 1 to 7”. However, this means that the project has met one of its aims and objectives which was to create a challenging AI.

The next question relates closely to the previous question because the first part asks the participants to rate how they felt playing against the AI with 1 being ‘felt like a real human being’ and 5 being ‘felt like a scripted component’. The average score was 2.2 which is a great score as it means that most people felt like playing against a real human being. P1 said “It’s not easy to foresee his moves”, P3 said “… games like these often feel very scripted … but this game did not, which I am very pleased with”. Another said “It did not feel like playing against a CPU … it wasn’t aiming at the goal all the time and sometimes even missed the goal which made it seem quite realistic”, these points are very important because this means that the AI is dynamic, which is one of the objectives, and that its behaviour is complex enough to convince the users that they are not playing with a scripted opponent. It also means that the AI Logic Design has been successful because the decision it makes are very natural, as P5 said “The CPU responded quicker than me, however, it felt like playing with an experienced player more than a scripted opponent.”

5.1.4 - WHAT PARTICIPANTS ENJOYED THE MOST AND THE LEAST ABOUT THE GAME

Most participants liked the originality of the game; the challenge and the addictiveness i.e. after playing few rounds they wanted another round to get better and beat the opponent. P3 said “I like that it is challenging because I am not going to get bored too soon.”, P4 even praised the customizability of the game by saying “I like the fact that you can change whatever you want in the game” although P3 said that “When playing with the menu it makes it go a little crazy” so these could be tested and improve to avoid unwanted behaviour.

The main complaints were towards the graphics and animation of the game on which this project did not focus but is something that could be improved. P2 said “Graphics and animations, as well as the occasional bugs”, P4 also mentioned the bugs in the game, “It’s quite annoying that I have to start each game as the computer does not move otherwise”. As playing the game myself, I have noticed bug as well, these should be taken care of as it impacts the game and may stop some players from enjoying it and playing it.

5.2 TESTING

5.2.1 – QA TESTING

The game has been tested using quality assurance testing techniques. The first technique is by playing the game and trying to get into as many situations as possible using every game mechanic. The main aim here is to 'break' the game to find as many bugs as possible. QA testing is usually done by a separate group because they try to interact with the game in a way that was not expected by the developers. Hence why many new games come with a patch on the day of the release – thousands of people that play the game for a first time have a higher chance of breaking the game than a small QA team (Lachance, 2016; Dassanayake, 2019; Sheridan, 2015; Makuch 2017).

Appendix 1.3 shows the results of the QA testing. As it can be noted some of the bugs have been fixed or partially fixed. However, many have unknown origin which makes it very hard to point exactly which feature or method is malfunctioning. Here unit testing would come in helpful, it would help to test each method and object easier but with the time constraints it was not possible to do so (Levy and Novak, 2010; Gamedesigning.org, 2018).

An example of a fixed bug is the bug 2. The fix has been very straightforward, however enlarging the colliders made the player get pushed whenever the shield was used next to a wall. This is unwanted behaviour because the shield should not be affected by the wall.

5.2.2 – PERFORMANCE TESTING

The next type of testing that has been done is the performance testing using the Unity's Profiler analysis window. It helps to optimize the game by giving the developer a detailed report about time spent in parts of the game like rendering or animating. The performance of computer hardware like CPU, GPU, memory and audio can be easily analyzed frame by frame (Unity, 2019).

This game has a very little number of objects and is very basic so even if the game is not optimized well, it would not put much load on computer hardware. However, if the performance came out bad, it would mean that the game is very badly optimized and there are issues that should be fixed as soon as possible. If the game kept getting larger, the issues would also grow until the game was unplayable.

The performance results came out very positive with fairly smooth with occasional hikes in the performance of the CPU. These hikes happen before loading a new feature and are usually accompanied by a stutter in a game. This is something that should be fixed in the game by investigating the issue closely and making appropriate changes to game code.

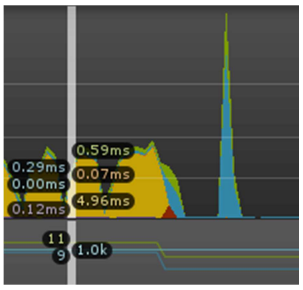


Fig. 5.1 – A hike in the CPU time – a stutter when loading components or game objects

CHAPTER 6 – CONCLUSIONS

6.1 - LESSONS LEARNT

The first and most important lesson I learnt when working on the project is how to manage one. Very early I realized that it is much better to work every day and have a clear short term and long term goals in terms of achievements, otherwise it is very easy to get stuck on one feature or part of the code. This is why it is very important to manage the project well to get the most of the time.

This brings me to the next lesson I learnt – it is impossible to write the perfect code as there is always something that could be improved. I have found myself sometimes spending large amount of time making sure that the code is readable, extendable and it performs as well as possible. However, some features or methods may not need to be extended or the implementation was good enough for the task and while rewriting or changing the code would 'improve' it, the time could be better spent working on something else like a new feature. This relates to the previous part, which is to have short and long term goals in mind when writing code.

I have also learned that while some code is impossible to refactor, it is important to make sure it is clean and at least readable with the comments. While I sometimes forgot to write comments because I thought that I understand the code at the time, when I came back to it after a while, it was sometimes hard to understand what is going on. Therefore I had learnt to try and write clean code and leave comments in any methods that do not have a trivial task.

The next thing I learnt is how important the design is and how important it is to not always follow it closely. The part that was not designed very well is the AI behaviour trees, while an outline has been written how it should be implemented; it was not clearly designed how each tree and leaf will work. This led to a number of redundant classes and game objects that introduced unnecessary complexity. It is also important to sometimes not follow the design, as initially the HUD was designed to be in top left corner and the ammo and weapon to be in the right bottom corner, however after testing it and adding couch co-op this had to be changed or otherwise the player would not be able to see their stats. Another part was to initially have a crosshair (a reference point where the gun will shoot) in front of the player. However after it was implemented, it turned out that it did not fit the game, and the weapon sprite was enough to tell the user where the gun is aiming.

The last lesson I learnt is that a project can go on indefinitely very easily as there is always something that can be improved, changed, refactored or added. This project could easily take another twelve months and I would easily have enough things to do every day for eight hours a day. In that time, there is a number of things that I would improve.

6.2 - REFLECTION AND IMPROVEMENTS

The first thing that I would add is bigger teams to play against, in couch co-op or in AI modes. This means adding more AI behaviour, as seen in Chapter 4, the Team AI Behaviour file has no methods; it would be really interesting to design and implement behaviours for larger teams. This would most likely need much more planning than just for one player as each AI player would have to be aware of other player and their opponents but it would lead to a much more possibilities. As calculated at the end of the Chapter 4, there are over 32 possible behaviours, this number would grow exponentially.

The next part that could be improved is the difficulty of the AI, as shown in the evaluation of the surveys, the difficulty scored 3.9 on a scale from 1 to 5. This number is really high because it may discourage other players from enjoying the game as first impressions are very important. The difficulty could be controlled with various settings like in other games, for example, 'easy', 'medium' or 'hard'. A more interesting approach would be to do what has planned in the project proposal, that is, have the AI react and adapt the difficulty to the player.

From more technical points, it would be very beneficial to refactor the code that deals with the AI and rewrite it to actual behaviour trees to make it easier to extend the behaviours. This should also help to fix the AI to adapt to new settings in Game Setup screen as after changing the size of the ball or its weight, makes the AI behave strangely. For example, the AI cannot position itself properly if the ball in enlarged because how it positions around it has been implemented with the default ball size in mind. It would also be important to have more quality assurance testing to get rid of most of the bugs as these affect the game negatively.

The part I would scrape off is the sword, I would rewrite it as I feel like it is not working as well I wanted it to and it has too many bugs which introduce unwanted behaviour. This would have to be planned and designed again with the opponents and the ball in mind as the swords' behaviour is to kick the ball and attack the opponent.

Lastly, I would add more game modes like 1vs5, larger playing field, more weapons, nicer sounds and music and improve the graphics. This should help to make the game feel finished and make users more willing to play it.

Overall, I think that the project was a success and I have achieved what I wanted to with it. It taught me a lot about starting, making and finishing a project and made me a better programmer. It also taught me how to compromise on various areas and stages in the project, and allowed me to have experience that could be applied in real world situations.

1.1 – GITHUB REPOSITORY

Includes diagrams, questionnaire, questionnaire answers, code, additional images, build of the game, other components.

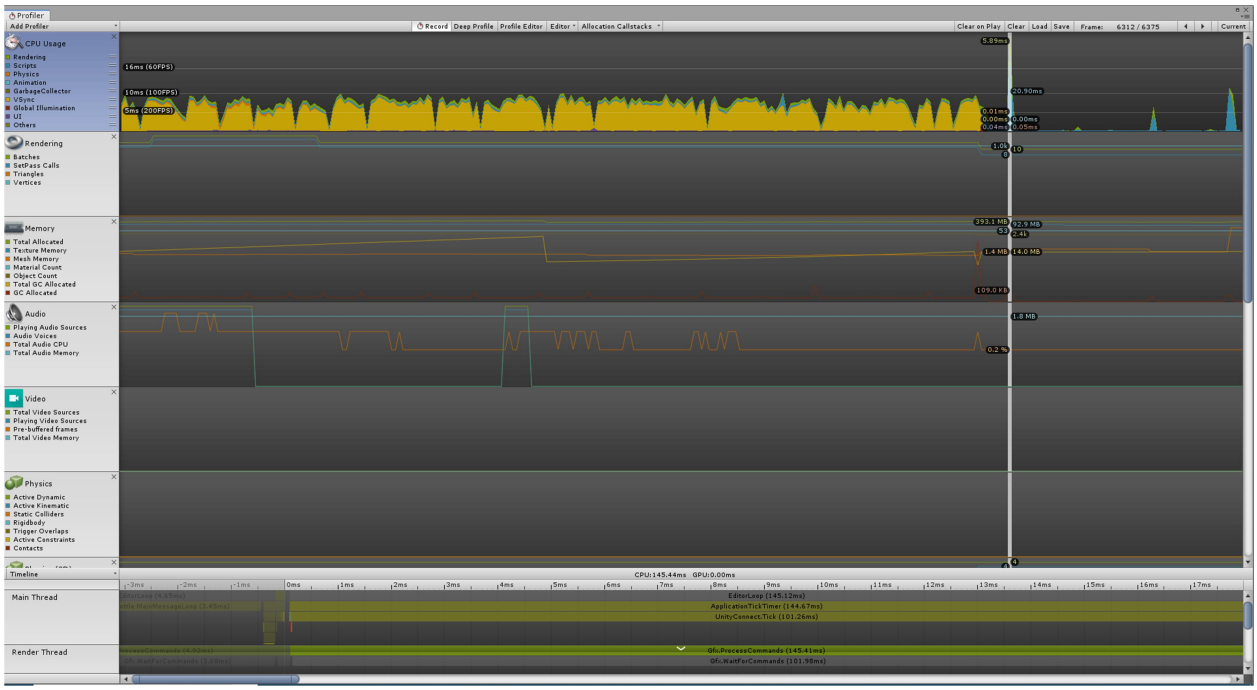
https://github.com/SylanProjects/FootballJetsProject

1.2 - OTHER EXAMPLES OF GAME UI

1.3 - QA TESTING RESULTS

No.	Test Description	Outcome	Fix
1	Forcing the AI to shoot the goal from a Shoot State	The AI either hits the ball too lightly or too hard which makes the ball fly out of the playing field.	`ShootInTheMiddle()` in AIGlobalBehaviour is not working as expected.
2	Using a shield next to the wall	Makes the player get stuck for the duration of the shield - the shield object is spawned outside the playing field.	Has been fixed by making the colliders in the playing field larger
3	Using a sword on the ball	It sometimes does not push the ball	The sword collider position and size should be checked to make sure it is spawned correctly
4	Using the buttons	The buttons send the user to a wrong scene	Has been fixed by changing the scene number
5	Forcing the AI opponent to use a pickup	It would start moving towards it but as soon the ball goes into a different state it changes its behaviour	
6	Using the gun	The gun does not reload when it is changed during the reload process	
7	Using the buttons to set which player is the AI	They do not have an effect on the game	
8	Forcing the AI to use sword on the player	Makes the player object 'teleport' few pixels forward	Has been fixed by deleting the method that pushes the player object forward
9	Getting AI to score at a goal	It is not able to position itself when it is very close to the goal so it just keeps pushing the ball from one side to another	
10	General (playing the game)	The AI sometimes moves very slowly in front of the ball until the ball reaches a different state	
11	General	The AI sometimes wiggles towards the ball	
12	Forcing the AI to use a shield	The AI sometimes keeps using a shield when the ball is moving away from it	

1.4 – PERFORMANCE TESTING RESULTS



REFERENCES

Bethke, E. (2003). Game development and production. Plano, Tex.: Wordware Pub., pp.3 - 36.

Nagy, Z. (2018). Artificial Intelligence and Machine Learning Fundamentals. Birmingham: Packt Publishing Ltd.

Rouse, M. (2019). What is AI (artificial intelligence)? - Definition from WhatIs.com. [online] SearchEnterpriseAI.
Available at: https://searchenterpriseai.techtarget.com/definition/AI-Artificial-Intelligence [Accessed 4 Apr.
2019].

Ward, J. (2019). What is a Game Engine?- GameCareerGuide.com. [online] Gamecareerguide.com. Available at:
http://www.gamecareerguide.com/features/529/what_is_a_game_.php [Accessed 4 Apr. 2019].

Ashe-Edmunds, S. (2019). *How to Evaluate Survey Results*. [online] Smallbusiness.chron.com. Available at: https://smallbusiness.chron.com/evaluate-survey-results-61615.html [Accessed 13 Sep. 2019].

Balkanay, I. (2016). *Don't Use Constructors To Initialize Monobehaviours.*. [online] Ilkinulas.github.io. Available at: http://ilkinulas.github.io/development/unity/2016/05/30/monobehaviour-constructor.html [Accessed 13 Sep. 2019].

Charmaz, K. (2006). *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis.* London: Sage Publications.

Co-optimus.com. (2019). *Co-Optimus - Your Prime Source for Co-Op Gaming- Co-Op Videogame Database*. [online] Available at: https://www.co-optimus.com/games.php?type=couch [Accessed 12 Sep. 2019].

Desktop, G. (2019). *GameMaker Studio 2 Desktop on Steam*. [online] Store.steampowered.com. Available at: https://store.steampowered.com/app/585410/GameMaker_Studio_2_Desktop/ [Accessed 13 Sep. 2019].

H. Harrison, C. (2019). [online] Psr.iq.harvard.edu. Available at: https://psr.iq.harvard.edu/files/psr/files/CognitiveTesting_0.pdf [Accessed 13 Sep. 2019].

Klappenbach, M. (2019). *How to Optimize and Improve Graphics Performance and Frame Rates*. [online] Lifewire. Available at: https://www.lifewire.com/optimizing-video-game-frame-rates-811784 [Accessed 12 Sep. 2019].

Kotaku.com. (2019). [online] Available at: https://kotaku.com/our-definitions-for-2d-and-3d-are-broken-please-fix-5514956 [Accessed 12 Sep. 2019].

Methods for Testing and Evaluating Survey Questionnaires. Wiley Series in Survey Methodology. (2004). Wiley.

SurveyMonkey. (2019). *How to Analyze Survey Data: Methods & Examples | SurveyMonkey*. [online] Available at: https://www.surveymonkey.com/mp/how-to-analyze-survey-data/ [Accessed 13 Sep. 2019].

Technologies, U. (2019). *Unity - Manual: Scenes*. [online] Docs.unity3d.com. Available at: https://docs.unity3d.com/Manual/CreatingScenes.html [Accessed 13 Sep. 2019].

Technologies, U. (2019). *Unity - Scripting API: GameObject*. [online] Docs.unity3d.com. Available at: https://docs.unity3d.com/ScriptReference/GameObject.html [Accessed 13 Sep. 2019].

Techopedia.com. (2019). *What is First Person Shooter (FPS)? - Definition from Techopedia*. [online] Available at: https://www.techopedia.com/definition/241/first-person-shooter-fps [Accessed 12 Sep. 2019].

Techopedia.com. (2019). *What is First Person Shooter (FPS)? - Definition from Techopedia*. [online] Available at: https://www.techopedia.com/definition/241/first-person-shooter-fps [Accessed 12 Sep. 2019].

Unrealengine.com. (2019). *Unreal Engine | What is Unreal Engine 4*. [online] Available at: https://www.unrealengine.com/en-US/what-is-unreal-engine-4 [Accessed 13 Sep. 2019].

WePC.com. (2019). *2018 Video Game Industry Statistics, Trends & Data - The Ultimate List*. [online] Available at: https://www.wepc.com/news/video-game-statistics/ [Accessed 12 Sep. 2019].

Colledanchise, M. and Ögren, P. (2018). *Behavior trees in robotics and AI*. Boca Raton, FL: CRC Press, Taylor & Francis Group.

Cummings, T. (2018). *Micro-productivity: Accomplishing Major Goals With Minor Effort*. [online] Lifehack. Available at: https://www.lifehack.org/articles/productivity/micro-productivity-accomplishing-major-goals-with-minor-effort.html [Accessed 13 Sep. 2019].

Game Guides. (2019). *Controls of PC, PlayStation 4 and Xbox One - Call Of Duty: WW2 Game Guide*. [online] Available at: https://guides.gamepressure.com/call_of_duty_ww2/guide.asp?ID=42390 [Accessed 13 Sep. 2019].

LH, S. (2019). *SOLID Principles: Explanation and examples*. [online] Medium. Available at: https://itnext.io/solid-principles-explanation-and-examples-715b975dcad4 [Accessed 13 Sep. 2019].

PwrDown. (2018). *Enter The Gungeon: List of Controls, Key Bindings & Settings - PwrDown*. [online] Available at: https://www.pwrdown.com/gaming/enter-the-gungeon-controls-key-bindings-settings/ [Accessed 13 Sep. 2019].

Yoyo Games. (2019). GameMaker | YoYo Games. [online] Available at: https://www.yoyogames.com/gamemaker?utm_source=google_adwords&utm_medium=text_ads&utm_campaign=Game_Making_UK&utm_term=Your_own_game&gclid=Cj0KCQjw_OzrBRDmARIsAAIdQ_IP7w13pKYc3WxHl1IJmA4qwdfgDgHpu50_W22XcgqsJIEUaEczJN4aAl7ZEALw_wcB [Accessed 13 Sep. 2019].

Call of Duty®. (2019). *Call of Duty®*. [online] Available at: https://www.callofduty.com/uk/en/ [Accessed 15 Sep. 2019].

Nintendo. (2019). *Nintendo*. [online] Available at: https://store.nintendo.co.uk/nintendo-switch.list?ds_rl=1253622&gclid=CjwKCAjwwvfrBRBIEiwA2nFiPetiz5Xbu5-k3xA7lcZ8oHrUI4-dAE9b_jUrDLM4Sn6Gskqx4FKsPhoCmBwQAvD_BwE [Accessed 15 Sep. 2019].

Playstation. (2019). *PS4™*. [online] Available at: https://www.playstation.com/en-gb/explore/ps4/ [Accessed 15 Sep. 2019].

Xbox.com. (2019). *Xbox Official Site: Consoles, Games and Community | Xbox*. [online] Available at: https://www.xbox.com/en-GB/ [Accessed 15 Sep. 2019].

Dassanayake, D. (2019). *Borderlands 3 how to get shift codes, rare loot boost, day one patch update*. [online] Express.co.uk. Available at: https://www.express.co.uk/entertainment/gaming/1177607/Borderlands-3-release-date-news-shift-codes-rare-loot-day-one-patch [Accessed 15 Sep. 2019].

Gamedesigning.org. (2019). [online] Available at: https://www.gamedesigning.org/video-game-tester/ [Accessed 15 Sep. 2019].

Lachance, M. (2016). *How much people, time and money should QA take? Part1*. [online] Gamasutra.com. Available at: https://www.gamasutra.com/blogs/MathieuLachance/20160113/263446/How_much_people_time_and_money_should_QA_take_Part1.php [Accessed 15 Sep. 2019].

Levy, L. and Novak, J. (2010). *Game development essentials*. Clifton Park, N.Y.: Delmar/Cengage Learning.

Makuch, E. (2017). *Horizon Zero Dawn Day One Patch Detailed*. [online] GameSpot. Available at: https://www.gamespot.com/articles/horizon-zero-dawn-day-one-patch-detailed/1100-6448046/ [Accessed 15 Sep. 2019].

Sheridan, C. (2015). *Here's what you'll get in The Witcher 3's day-one patch*. [online] gamesradar. Available at: https://www.gamesradar.com/uk/witcher-3-patch-notes/ [Accessed 15 Sep. 2019].

Testing, G. (2019). *Best Practices for QA Testing | Global App Testing*. [online] Globalapptesting.com. Available at: https://www.globalapptesting.com/best-practices-for-qa-testing [Accessed 15 Sep. 2019].