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

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. https://www.lifewire.com/optimizing-video-game-frame-rates-811784

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.

https://www.co-optimus.com/games.php?type=couch

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’.

https://kotaku.com/our-definitions-for-2d-and-3d-are-broken-please-fix-5514956

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.

https://www.techopedia.com/definition/241/first-person-shooter-fps

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.

CONTENTS

8

Chapter 1 - Introduction 1

Tools 2

GameMaker 2

UnrealEngine 2

Unity 2

Evaluation 3

Road map 3

Chapter 2 – Background 4

Unity 4

Scene 4

Physics 5

MonoBehaviour 5

User Interface 7

Colliders 7

Chapter 3 – Analysis, Requirements and Design 8

Overview 8

The view 8

Controls 9

The play area 10

The ball 10

Teams 11

Stats 11

Weapons 11

Shield 12

Pickups 12

Game Objects 12

HUD 14

Main Menu 15

GAME MODES....16
Couch co-op....17
1vs1....17
Artificial Intelligence – computer controlled opponents....17
Movement....17
Behaviours....18
Logic....20
The four states....21
Four positions....22
Choice....25
Chapter 4 – Implementation....29
Part one....29
Players....29
Weapons....35
Playing Field....41
User Interface....44
The Game Setup....46
Teams....48
Part Two – Artificial Intelligence....48
Behaviours....48
The logic....58
Four states....59
Four positions....59
Distance....61
Amount of behaviours....62
Chapter 5 – Experimentation and Evaluation....63
Evaluation....63
Overall experience....63
Graphic User Interface....63
AI (difficulty and how it plays)....64
What participants enjoyed the most and the least about the game....64

Chapter 6 – Conclusions.....65
Appendices.....65
References.....69

CHAPTER 1 - INTRODUCTION

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. The 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 enjoy and have fun with.

https://www.wepc.com/news/video-game-statistics/

This project will focus on creating an enjoyable game experience that anyone will be able to pick up and enjoy, either alone or with friends. The aim of this project is create a game with good overall experience that will challenge the player, with each match feeling distinct and exciting. The main objectives of the project are 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 Artificial Intelligence. The objectives have slightly changed compared to the ones in the Project Proposal – after doing the initial planning, this project will not focus on Machine Learning.

The project is 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 complex project that can be scaled up or down. It will give me an opportunity to try out some of the topics that I have studied during the year and give 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 will also allow me to dig deeper and understand its workings.

The project will not focus so much on refining and polishing the game mechanics because as said in project proposal, these can take a full team to develop. The project will instead focus on creating a basic video game, and designing and creating Artificial Intelligence that will behave and challenge the players. One part of the project will focus on the game mechanics and the other part on the Artificial Intelligence. This will allow creating a satisfying overall user experience without getting hanged up on small details. The game will be kept very basic – inspired by games like Rocket League, Grand Theft Auto 2 and FIFA that provide interesting, challenging and arcade-like experience to users.

There will be two teams on a play field, one ball and two goals just like in a classic football. Each team will have to work together to score a goal by using various weapons like melee, guns etc. to hit the ball into the opponents' goal. Each player will have their own health points and the game will be timed, the team with the highest score by the end of the game or the team that reaches the maximum score first, wins. Users will be 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 is a lot of room to scale the game up or down if needed. For example to make the game mechanics more basic to save time, fewer weapons can be implemented, or to make the game AI more complex, more game modes can be added where the opponents have to work in teams.

TOOLS

The main tool that this game will need is 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, are GameMaker Studio, Unreal Engine and Unity. Each of these game engines provides excellent tools to create and develop games.

GAMEMAKER

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

https://store.steampowered.com/app/585410/GameMaker_Studio_2_Desktop/

UNREALENGINE

Unreal engine developed by Epic Games is 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 originally made for FPS games and while it is possible to create 2D games, it is less suited and a basic search shows that there are much more Tutorials for 2D games for Unity than Unreal.

https://www.unrealengine.com/en-US/what-is-unreal-engine-4

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. Unity will be the main tool used to create the game, along with Visual Studio, development environment developed by Microsoft, for coding.

https://en.wikipedia.org/wiki/Unity_(game_engine)

Unity has already been explained in project proposal so this section gives a basic overview of Unity, why it has been chosen and how it looks.

Unity has been chosen for this project because it provides all of the necessary tools that will be needed to create the game 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 makes it very easy to get into Unity and start creating games without much experience in game development.

EVALUATION

Deciding whether the game is successful can be very hard because gaming is an art and art can be a very subjective experience. Therefore, 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 and the questions should be written with an answer in mind. As said in REFERENCE, it is important to avoid 'Yes' or 'No' questions as this does not give any depth to the reasoning behind 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 in REFERENCE, understanding the questions is the first step in answering it – if the question is not understood it cannot be answered correctly, or a different question will be answered.

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. 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. 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. REFERENCE

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. REFERENCE

https://smallbusiness.chron.com/evaluate-survey-results-61615.html

https://psr.iq.harvard.edu/files/psr/files/CognitiveTesting_0.pdf

https://www.surveymonkey.com/mp/how-to-analyze-survey-data/

https://books.google.co.uk/books?id=rPJqFmQe0sYC&pg=PA527&dq=evaluating+survey+results&hl=en&sa=X&ved=0ahUKEwiOlbvH7LvkAhWiVBUIHWGrA9UQ6AEIKjAA#v=onepage&q=evaluating%20survey%20results&f=false

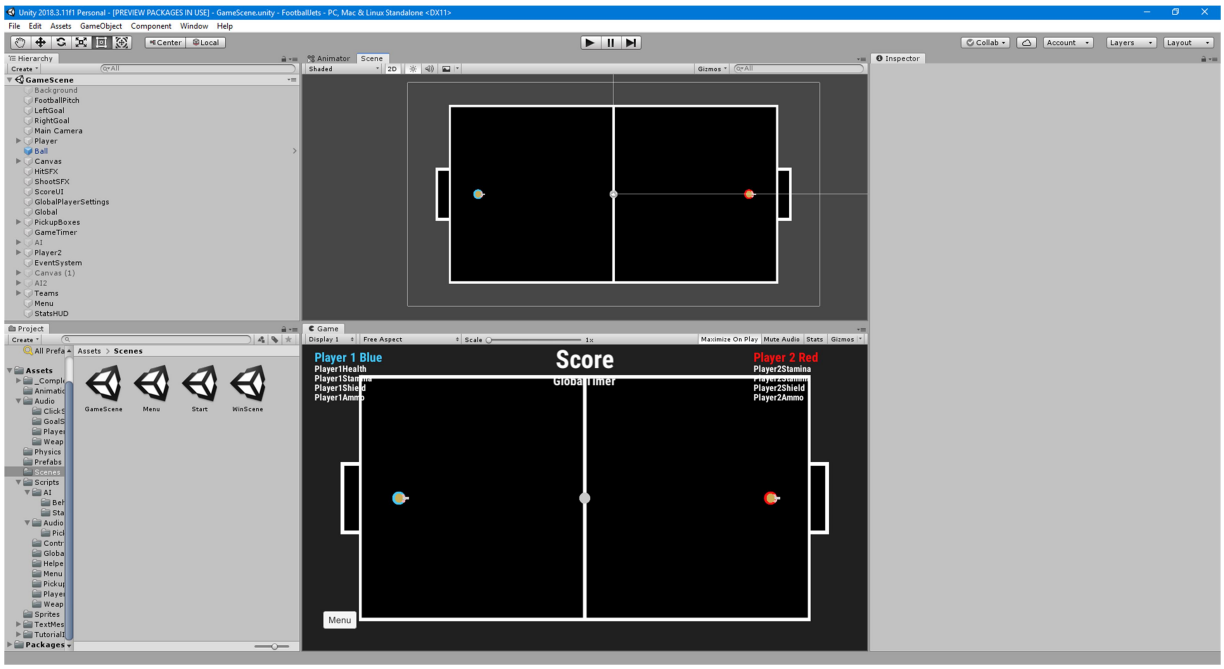
ROAD MAP

CHAPTER 2 – BACKGROUND

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

UNITY

Figbelow. shows the basic Unity window, at the top of the window is a play button which allows running the game very easily inside the game engine without first building it. 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.



SCENE

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

https://docs.unity3d.com/Manual/CreatingScenes.html

In Unity 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 controls, Rigidbody and a Collider to an object makes it a playable character. All object come with a Transform component which describes it position in a game scene.

https://docs.unity3d.com/ScriptReference/GameObject.html

https://docs.unity3d.com/560/Documentation/Manual/class-GameObject.html

PHYSICS

Game physics engine is a software that introduces and simulates laws of physics into a game. These laws can either be based on real-world physics or can be entirely new. It simulates the collision detection, rigid or soft body dynamics and fluid dynamics. Unity provides a built-in physics engines for 3D and 2D games that handle the physical simulation. The main components it provides are the Rigidbody2D and the Collider. These are explained in … but in essence, Rigidbody2D allows the game objects to be affected by various forces like the gravity. Collider allows various objects to collide with each other and if paired with Rigidbody, allows objects to affect each other.

https://docs.unity3d.com/Manual/PhysicsSection.html

MONOBEHAVIOUR

MonoBehaviour is a base class from which every other class should derive. MonoBehaviour has all the important functions that are used by the Unity in order to work properly. The methods are Start, Update, FixedUpdate, LateUpdate, OnGUI, OnDisable, OnEnable.

The last three were not used in this project so their explanation can be found in appendix..

Name	**Description**	**When it is run**	**Comments**
Start()	Start is used to initialize the game objects by enabling the developer to set it up before the object is updated.	called 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 in REFERENCE
Update()	Update the game object it is attached to	Called every frame	
LateUpdate()		Called every frame but it is called only once all	Updating the camera position should be

		Update methods, inside the object and inside other objects, have been called	done only once other objects' position has been updated to avoid issues and unexpected behavior.
FixedUpdate()	Mostly used to calculate physics and to avoid issues with time dependent methods	Called every fixed time not dependent on the frame rate.	For example if the player has to move right at a certain speed, its position should be updated using the FixedUpdate method

http://ilkinulas.github.io/development/unity/2016/05/30/monobehaviour-constructor.html

The update is most commonly used method along with the Start, because it allows 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 changed by adding a number to it in the update method. The code in figure … would make the player move by one pixel to the right every frame. The update methods are called only if the MonoBehabviour is enabled.

```
using UnityEngine;
using System.Collections;

public class PlayerController : MonoBehaviour
{
    void Update()
    {
        /...SetPosition(GameObject.pos.x + 1, GameObject.pos.y);
    }
}
```

In figure … the speed in which the object would move would be dependent on the frame rate, if the game was running at 25 frames per second, the value would be updated 25 times in one second but if the game was running at 100fps, it would be updated 100 times. 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.

https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

USER INTERFACE

The user interface in Unity is easily implemented by creating a Canvas. 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 create 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.

COLLIDERS

In Unity there is a very useful physics component to detect collision between objects called Collider2D. It is implemented in a UnityEngine.Physics2DModule and offers a lot of functionality to work with a 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.

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, then it goes to explain the game objects and their design. Lastly, it goes through the Artificial Intelligence.

OVERVIEW

The game is a mix of various game types including shooters, fighting and sports games, with a top down camera view. It was 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 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

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 fig (Galyonkin, 2019)...

The aim will be to make the game look and feel very intuitive so anyone can start it and after just few minutes, be able to understand the main goal of the game and what needs to be achieved in order to success. An example of a very intuitive and straightforward game that was used as an inspiration for this project is the Super Mario Bros platform game published by Nintendo and released in 1985. This game is a perfect example of a game that is easy to get into and very difficult to master. It has very basic controls and a very basic goal – collect coins, jump on platforms and try not to die by falling into an empty pit or getting killed by enemies. This can seem very basic but the game manages to achieve a lot with its premise and is still an inspiration to many modern games.

CONTROLS

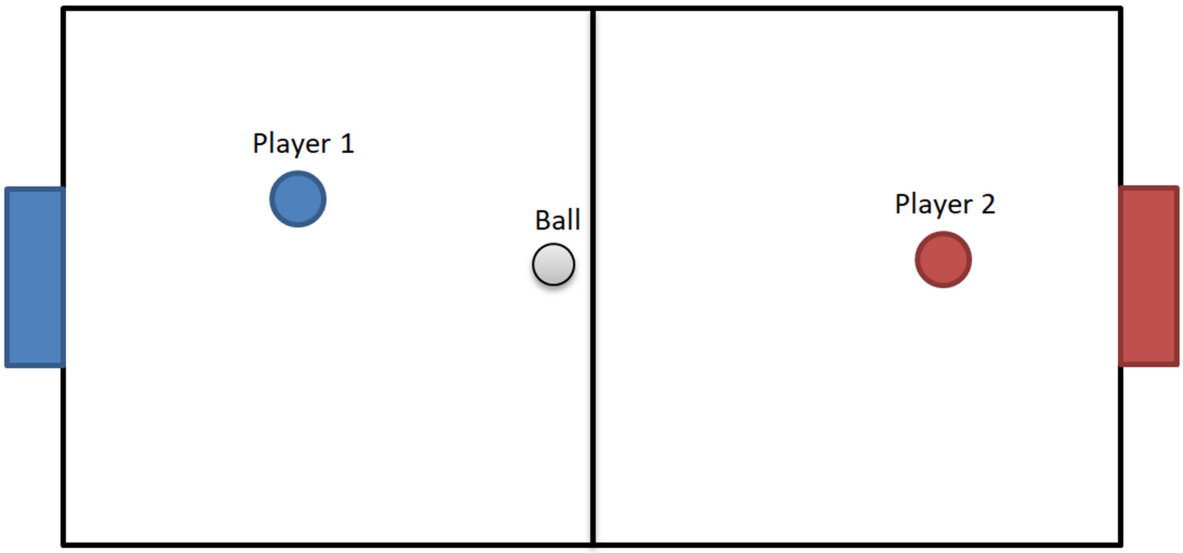
The game controls will be based on a gamepad to create a very best experience since the player should be able to move in every direction. A full 360 movement can only be achieved with analogue sticks on a gamepad. User will be able to rotate the player object using the right analogue stick, and use the weapons using the other buttons. 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 type of controls are taken from a game like Call Of Duty and Enter the Gungeon REFERENCE.

Some less experienced players may have difficulty with controlling the movement and rotation separately therefore an option to allow the player to move and rotate with one stick will be added. This should make the game easier to control and would leave their right hand for other tasks like using the weapons. More experienced players may find this type of controls not be flexible enough because it restricts their movement therefore they would be able to switch between the two. 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.

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 that is a rectangle with a line in the middle separating it into two; each side belonging to each team as shown in fig.. There will be goals on each side to which the players have to kick the ball into to score a goal. A team gets one point only when the ball hits the back of the goal, so when the ball is only halfway inside the goal, the team will not get the point. [PICTURE]



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. The players can score points either by killing the opponent.

TEAMS

There will only be one player on each team but the game could be expanded to more players, even up to 10 – this would however require larger playing field. This size of the team has been chosen because of the complexity of Artificial Intelligence. Bigger size of the teams would make the AI very complex because it would have to dynamically adapt to a lot more situations. For example, in a one-on-one game the AI only needs to follow the ball, block its own goal and try to score the goal. In a two-on-two game the AI would have to do all that, plus it would have to make sure it does not get in the way of its own team mates. Bigger team would also mean that in a single player mode (which is when one use plays the game) new AI would have to be created which would be in the team with the user. This means that the AI would have to behave and react to the players' behaviour differently since a real human is much less predictable. Therefore, focus as of now is on a one-to-one game.

To make the game independent of the amount of player teams has to be made. Since there are only two goals, there can only be two teams – Red and Blue. Each player will be given its Team component which will hold information about teammates, opponents, goals, starting position of each player and the score of the game.

STATS

Users will have a number of things that 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 that is bound to the game character, 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).

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.. 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

SHIELD

The last thing that players will have at their disposal is the shield. It will spawn in front of the character for a few seconds and slowly disappears. 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 since player will get no damage when the bullet hits the shield. Player cannot use a gun or a sword when the shield is up.

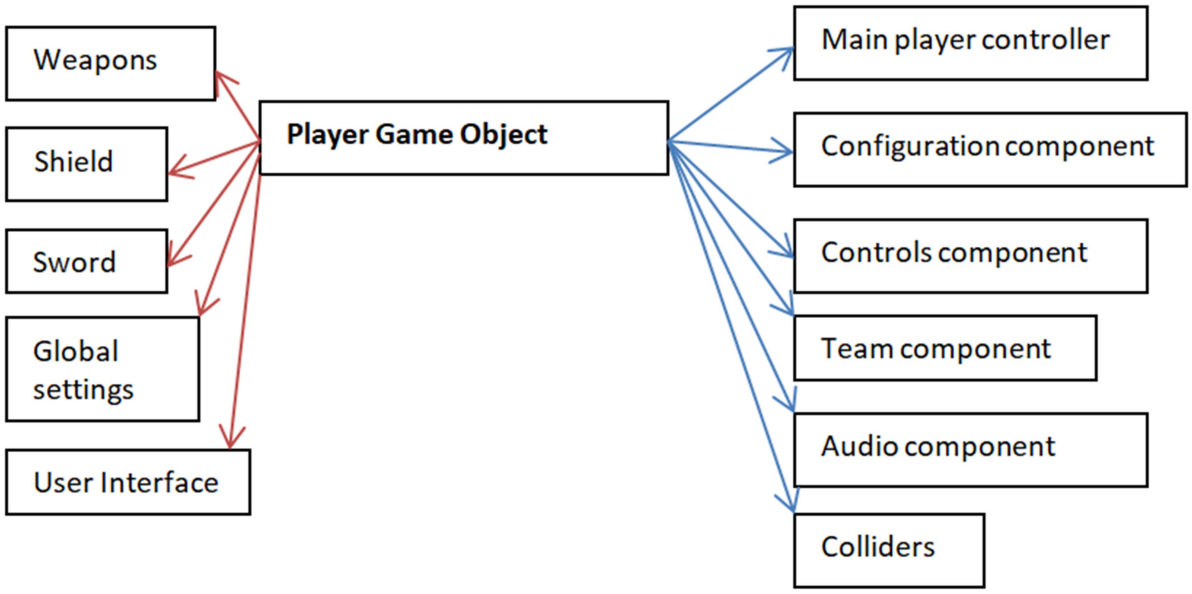
PICKUPS

Players will be able to keep up boxes which will either replenish their health, stamina or ammo. Table. Shows the effect of each box.

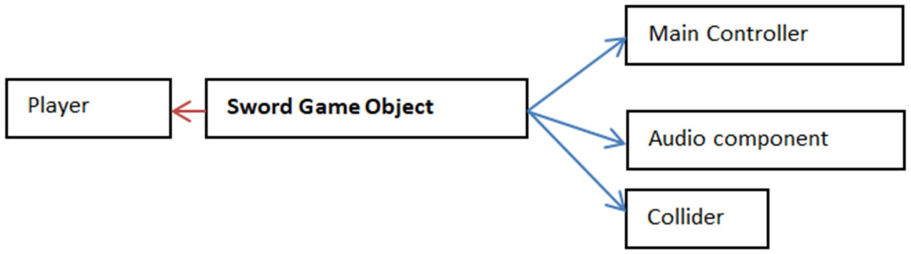
Name	Effect
Replenish Health	10 to 50
Replenish Stamina	20 to 100
Replenish Ammo	50 to 200, random weapon

GAME OBJECTS

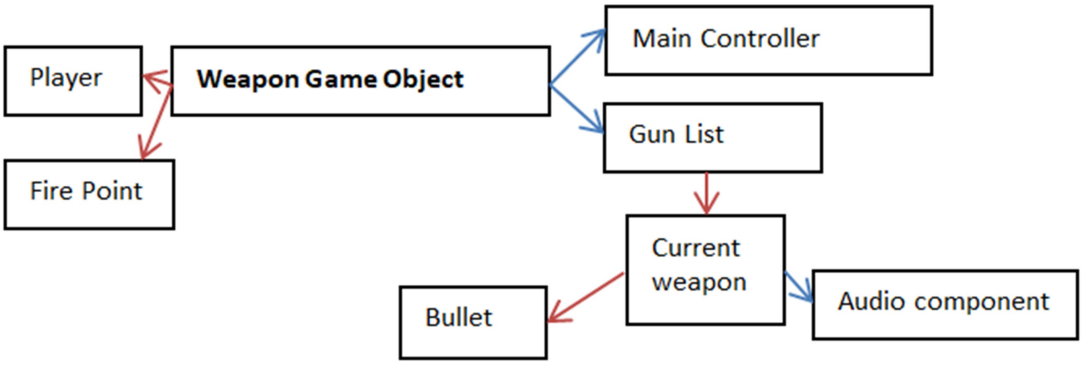
Fig. shows the player object, red arrows point to other game objects, and blue arrows point to components within the game object. In reality, there will most likely be cross referencing between the objects, for example, controls component would point to sword, shield and weapons. Main Player Controller may reference Global Settings, and components would reference the Main Player Controller. 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 game object from one point. This will most likely be the most complex game object in the game because it has the most functionality.



Most objects like the shield or a ball will have the same components as the sword game object because it has all the necessary components to have a very basic functionality. The object shown in fig. would also reference global settings, but to make it easier, it should do so through the player because it would most likely be its child.

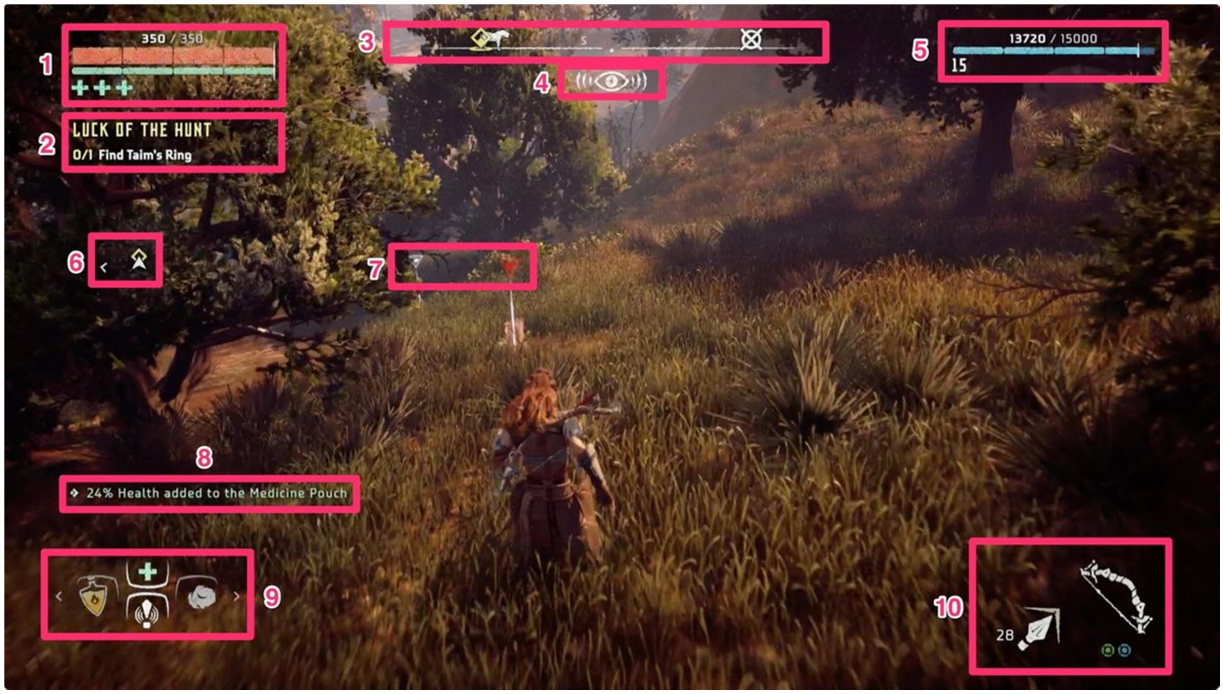


Weapon game object will be a bit more complicated than the sword game object. Fire point will be the position relative to the player from which the bullet should spawn. This will have to be passed down through the Gun List to the actual gun object. Guns will be kept as separate game objects and will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



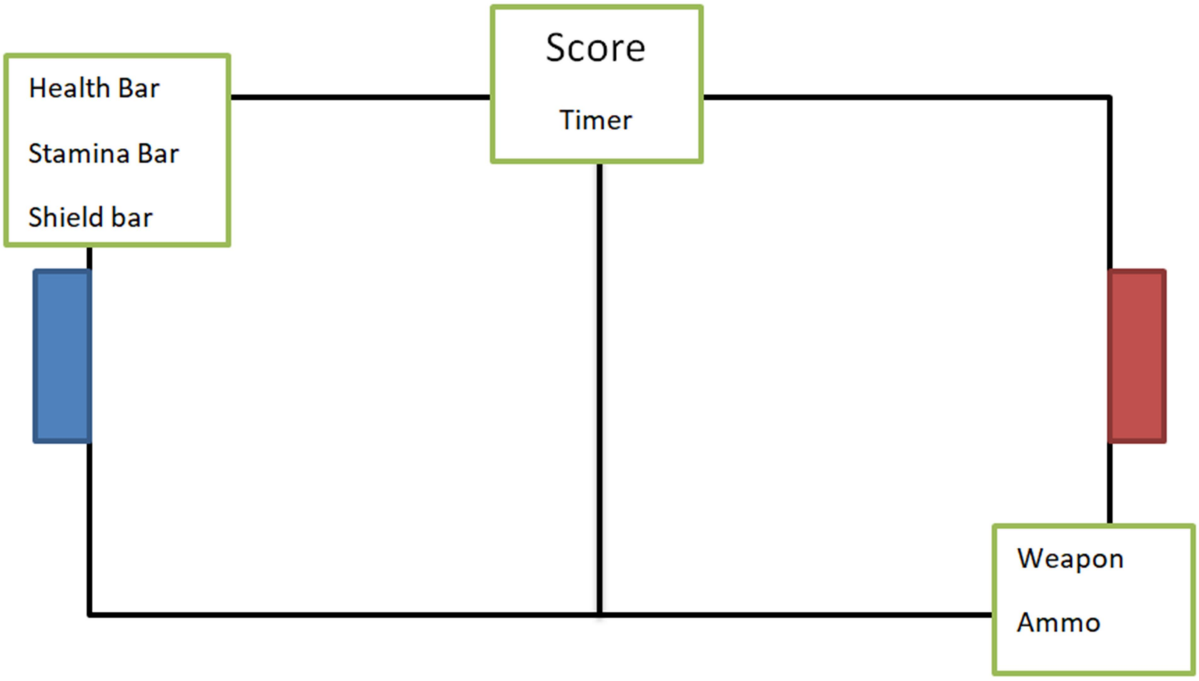
HUD

The User Interface should be easy to use, intuitive and convey lots of information in a very easy to understand way. Modern games range from excessive information on the screen to as little as possible. An example of a game where there is an excessive amount of information on the screen is the Horizon Zero Dawn action role-playing game developed by Guerrilla Games and released in 2017. Figure… shows the UI and as it can be seen, the player is given all the important information on the screen like health points, stamina, weapons, map and quick access items. 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.



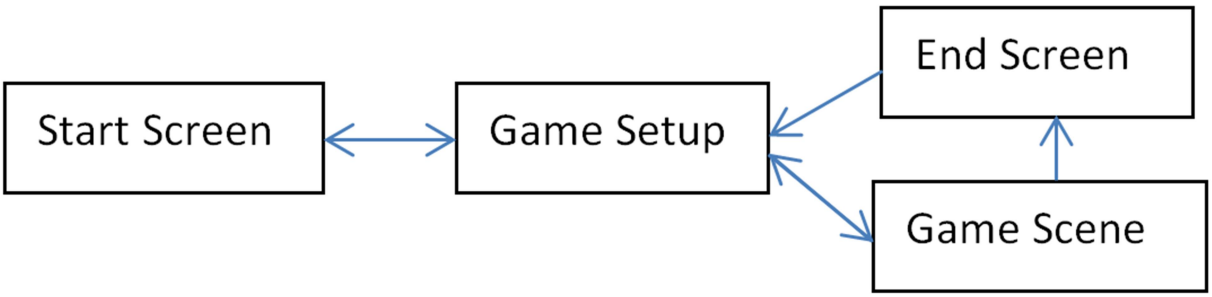
On the other end there are games that are more like the movies. Uncharted 4, an action adventure game developed by Naughty Dog and released in 2017, is an example of a game where the player does not have too much impact on the game world and almost every event is scripted. This allows the game designers to have as little information on the screen at one time 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 very fast paced and competitive so the player will only be able to take a brief look at the HUD every some time. It takes an inspiration from older arcade games like Donkey Kong or Space Invaders. It will also be kept as 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…, 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.



Main 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 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.



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 above 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 Call Of Duty where players are given an option to create custom games that fit their play style. In arcade games there were usually fixed settings. 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.

GAME MODES

The game modes are the way the user can play a game. Most games come with various games 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).

Couch co-op

The first mode should be relatively easy to implement once one of the player characters is implemented. The player game object would just have to be duplicated. 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.

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 this aspect in a big part. Here, the user will not have to see its opponents' status so there will be much more space to spread out the HUD.

The game modes will be implemented in a way that they will be able to be expanded to bigger teams but for now, to make sure the project will be possible to finish in the given time, the teams will be limited to one player only on each team. However, if possible, this can be expanded to more players if there will be enough time to expand the Artificial Intelligence's behavior to work in a team.

1vs1

This game mode will create a foundation for other game modes because this will be the default behaviour of AI. That means it will not have to worry about getting in the way of its teammates and figuring out what its teammates are doing or their positions. Its only objective will be to score a goal. Before understanding how everything will work together, it is important to understand the basic behaviours that the AI will have at its disposal.

https://docs.unity3d.com/ScriptReference/Rigidbody2D.html

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.

Movement

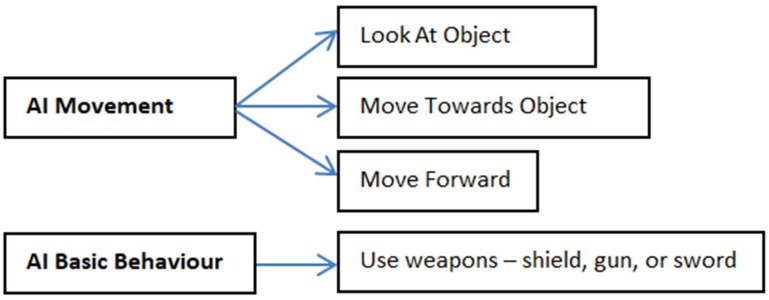
The first issue will be to find out what the AI should focus on – the general direction that the player should look in. The obvious answer would be to look at the ball at all times and move towards it, this

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 any direction.

This creates another issue – the player should not always move in the direction that it is looking at. Especially since users also have the ability to aim and move separately as mentioned in the Player Controls section. So a separate method for movement should be created.

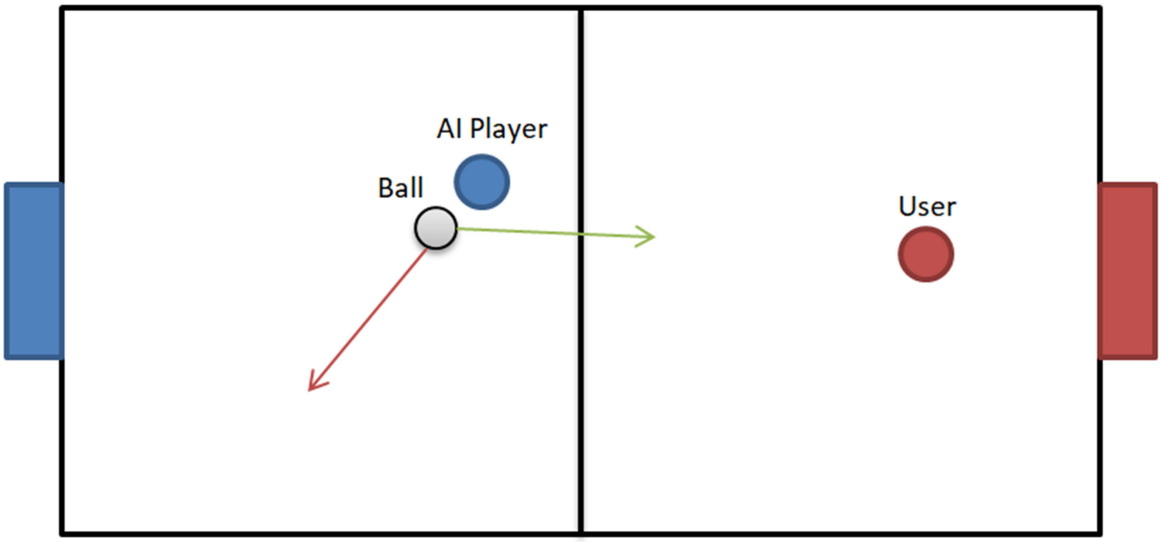
Using weapons and the shield is very straightforward since these methods should already be created as mentioned in a previous section. A way to call that method will only be needed – this could be done with a static method that takes the player game object and runs, for example, a UseSword method inside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



Behaviours

Now that the movement, rotation and weapons can be controlled, more complex behaviours can be created. 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 with this sword. 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 fig, 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.



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 in…. will be used throughout this project. Breaking the tasks can make them a lot easier to achieve and makes them much more flexible.

https://www.microsoft.com/en-us/research/project/microtasks-and-microproductivity/?irgwc=1&OCID=AID2000142_aff_7806_1246483&tduid=(ir__6n2pevwuwgkfrlspkk0sohz30n2xjqpnf02owvev00)(7806)(1246483)(%28d2de1c3646d9ad0bace4b864b91163dc%29%2881561%29%28686431%29%28at106140_a107739_m12_p12460_cGB%29%28%29)(d2de1c3646d9ad0bace4b864b91163dc)&irclickid=_6n2pevwuwgkfrlspkk0sohz30n2xjqpnf02owvev00#!publications

There will be lots of behaviours that will need to be implemented which the AI should use to its advantage. Table. Shows the rest of 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	3 to 6	

	automatically in the UseGun method – or if the player has to switch weapons.		
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

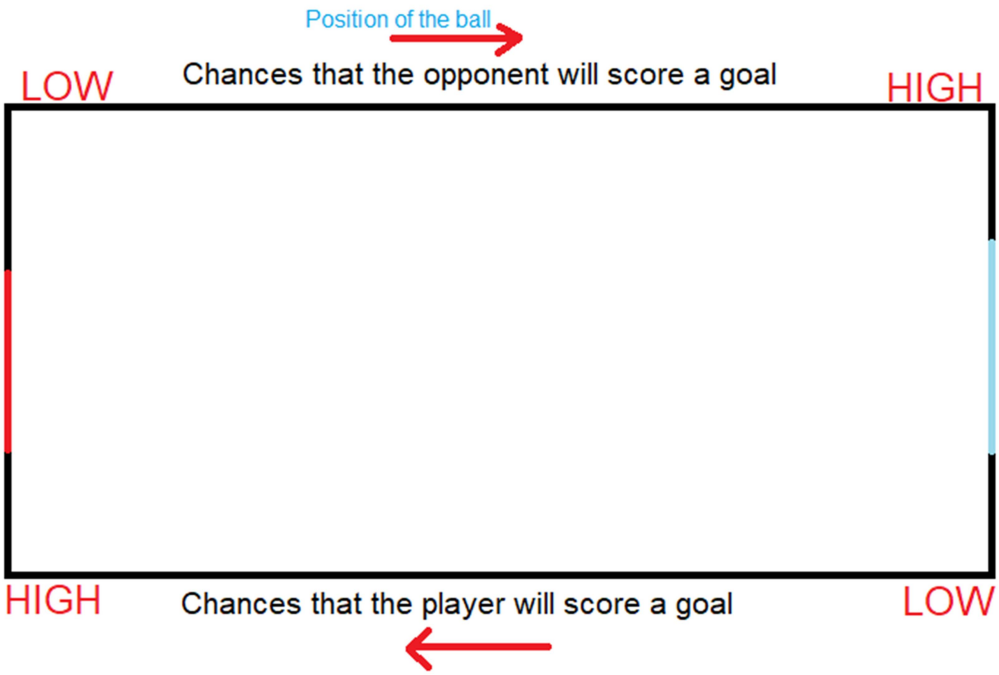
Logic

Each one of these behaviours creates a very basic behaviour tree that will be used in larger structures to create complex behaviour. The next step after creating the basic behaviours is to find a way in which the artificial intelligence 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.. a basic behaviour is broken down into three or four smaller tasks, to keep things simple, a similar approach will be used. The general goals of the player are to:

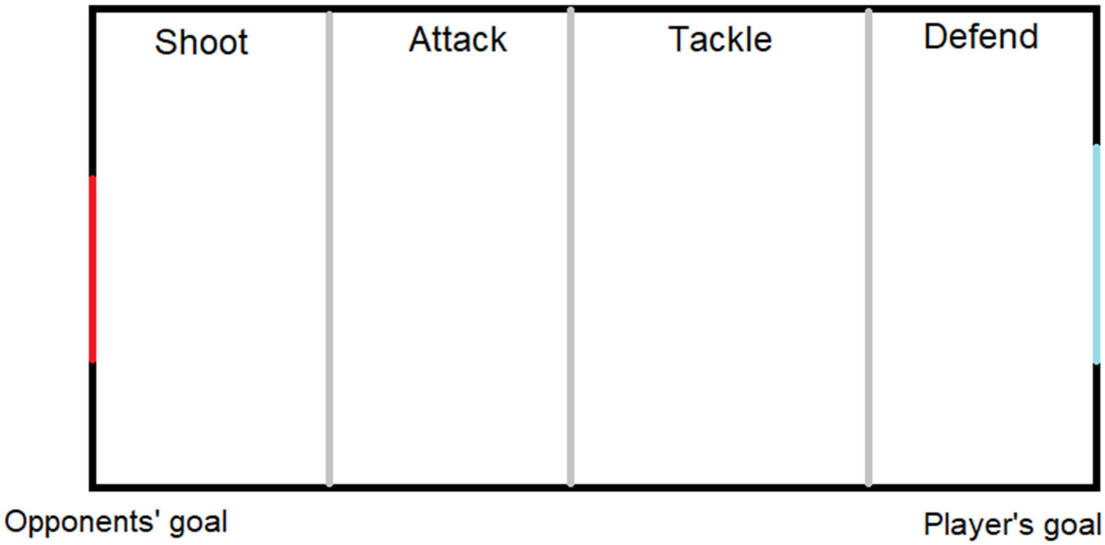
- Score a point
- Defend the goal
- Make it hard for the opponent to score a goal
- Do not die

The AI player cannot realistically try to do all of those tasks at the same time so he 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.



This means that the position of the ball can be used to help decide which one of the four tasks should be prioritised. 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.

THE FOUR STATES



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 getting pickup boxes unless its health is really low and he is really close to it.

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.

Tackle

Similarly as 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.

Defend

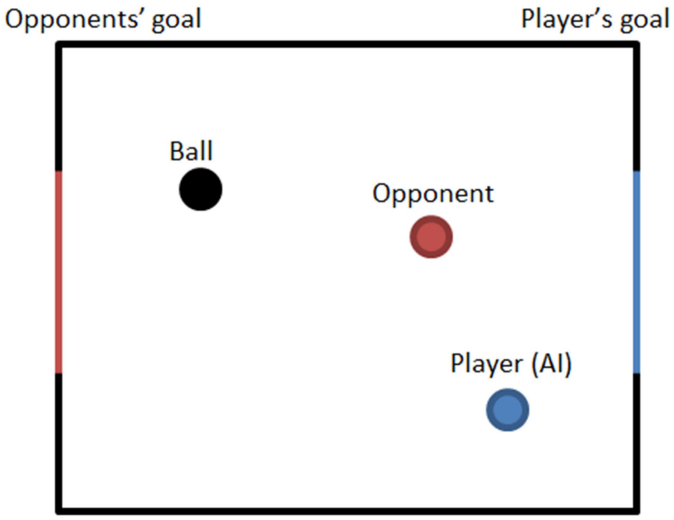
This state along with the shoot state are the most crucial for the outcome of the game because they decide which of the players will score the most points. In this state the main task should be to defend the goal and make it as hard as possible for the opponent to score a goal. Players' main task should 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.

Four positions

Next step would be to find a way how 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 for now. Therefore the AI player should behave differently depending on its x position relative to the ball, opponent and the opponents' goal.

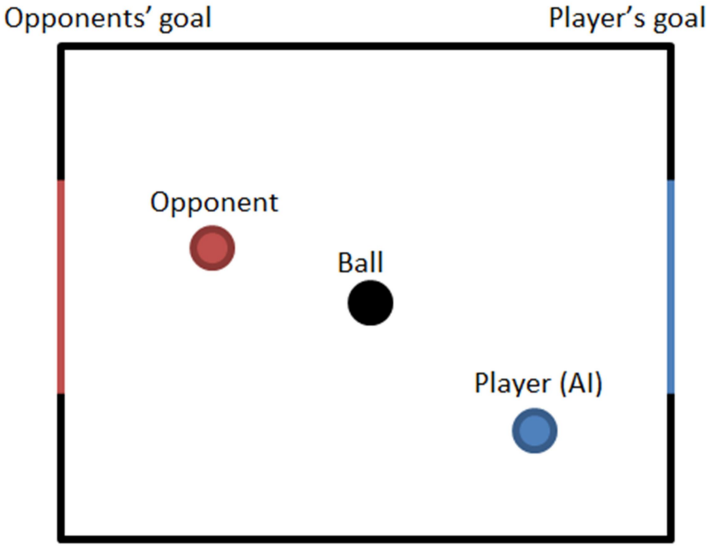
These positions are named Zero Position, One Position, Two Position and Three Position. Each of these states will prioritise a different behaviour just like the first four states did.

Position Zero



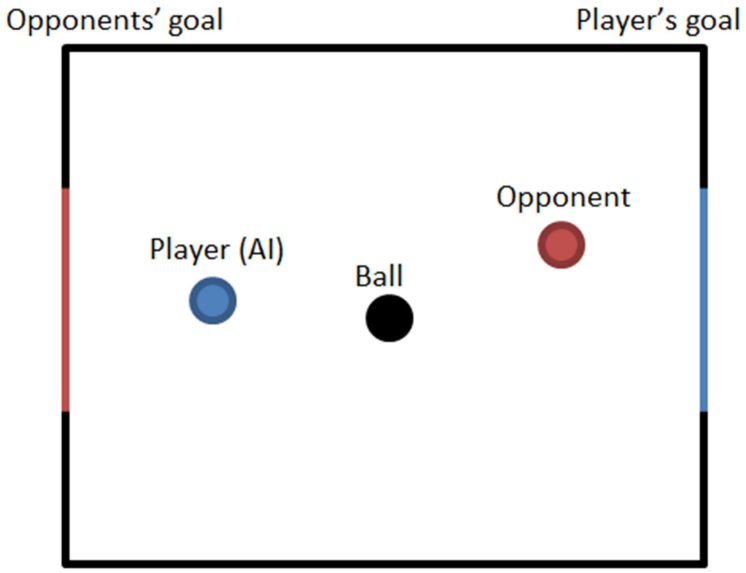
This position is advantageous for the player because there is no obstacle between the ball and the ball which makes it easy for him to score a goal. Depending on the state, the player in this position could focus on scoring a goal or, if the ball is in the defend position; he could be able to quickly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

Position One



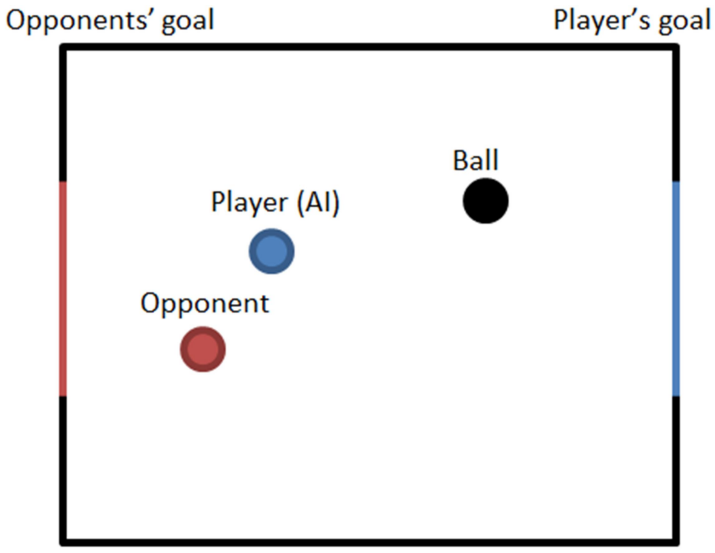
This position is neither good nor bad for the opponent and the player because the opponent and the player cannot score a goal. Usually, the most effective task to do would just be to try and turn around.

Position Two



This position is similar to the One Position 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.

Position Three



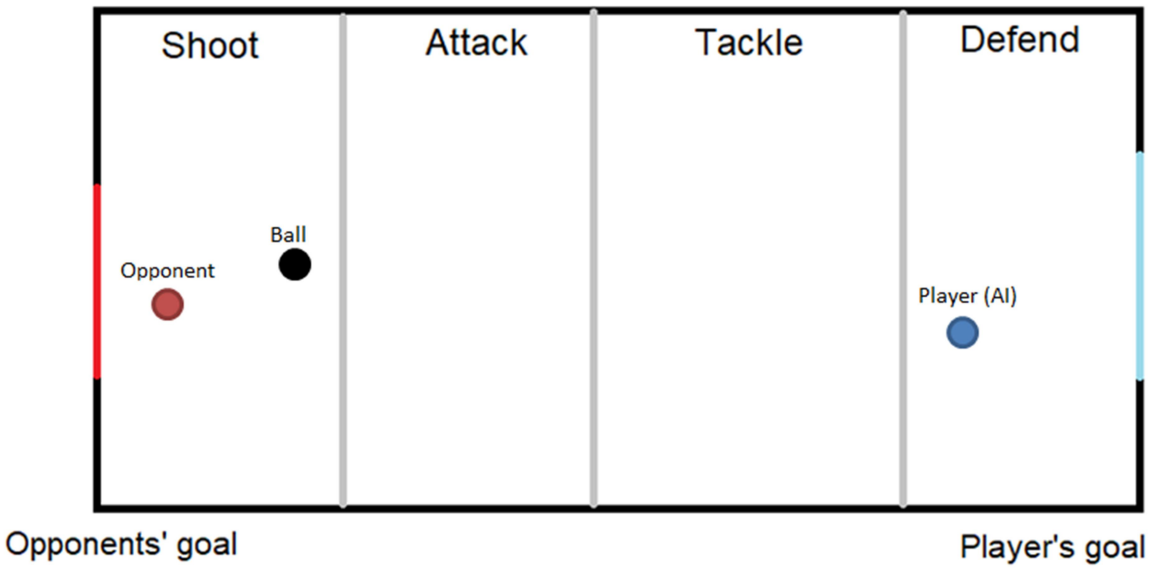
The last position is the most advantageous for the opponent because there are no obstacles between the player's goal and the ball so the opponent can easily kick it, even from a far, and score a goal. The

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.

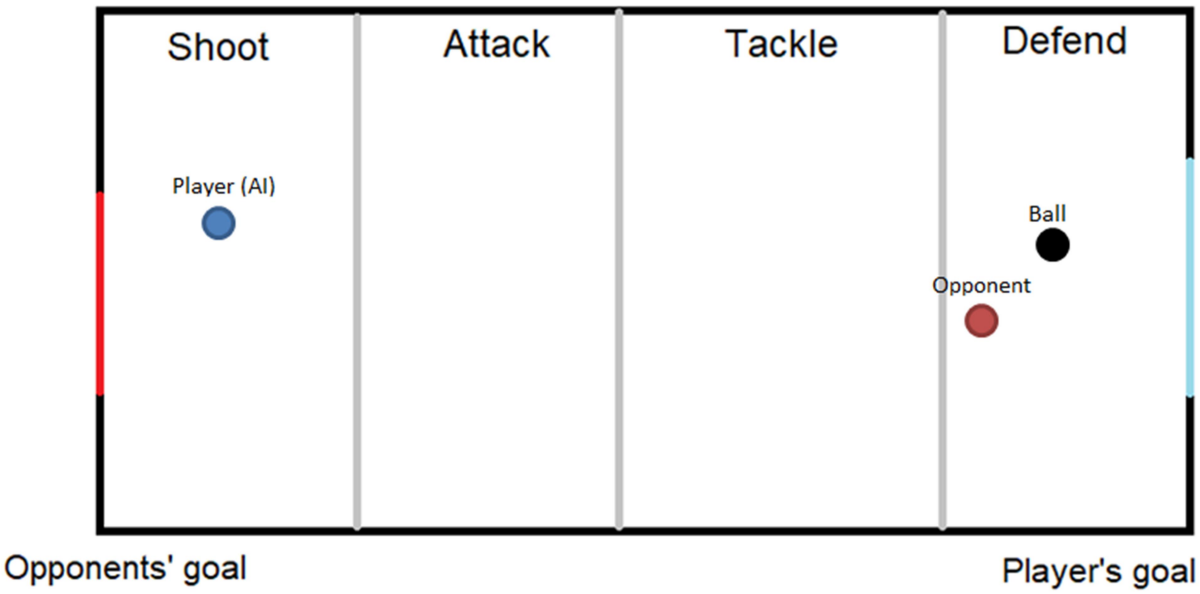
https://arxiv.org/pdf/1709.00084.pdf

CHOICE

To cover almost every possible situation that 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 the figure … 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. PICTURE



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.



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

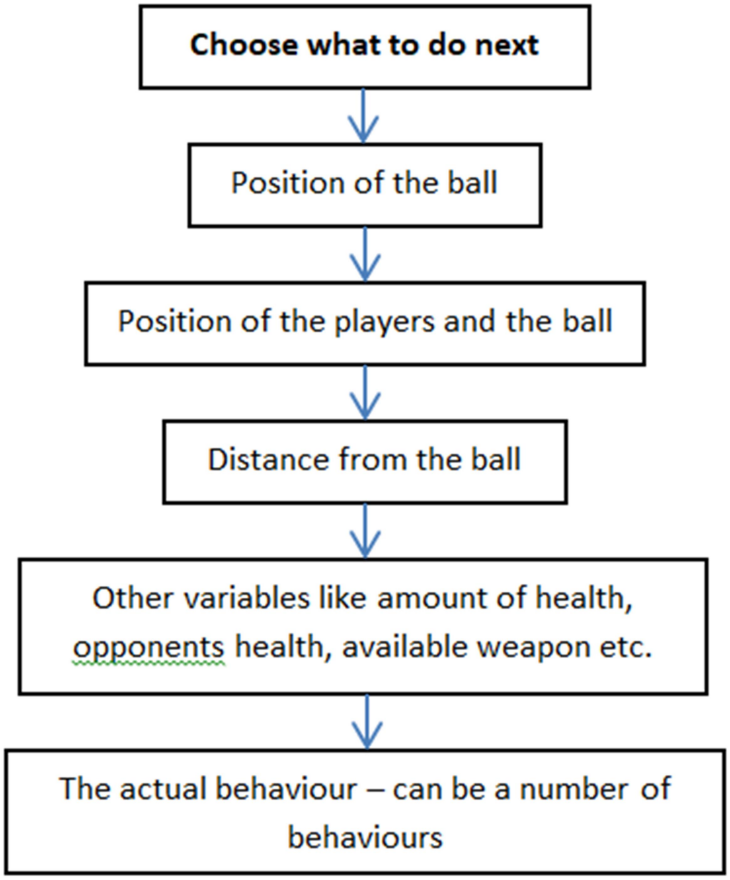
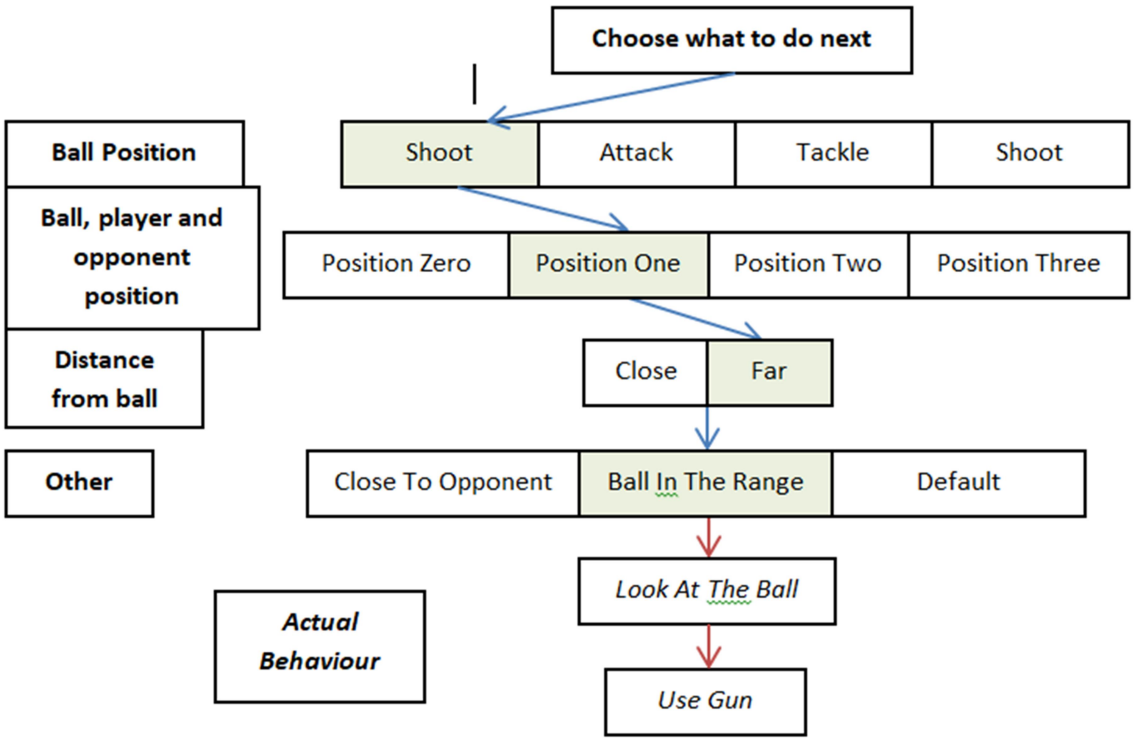


Fig. shows how the behaviour would be chosen for fig.

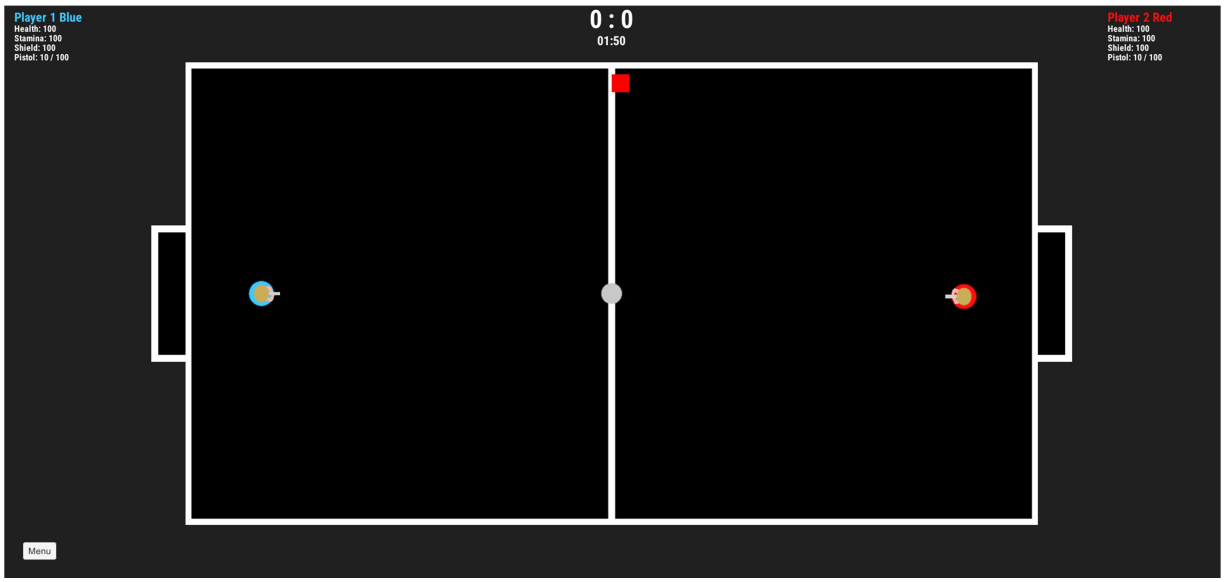

Choose what to do next
Ball Position
Shoot
Attack
Tackle
Shoot
Ball, player and opponent position
Position Zero
Position One
Position Two
Position Three
Distance from ball
Close
Far
Other
Close To Opponent
Ball In The Range
Default
Look At The Ball
Actual Behaviour
Use Gun

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.

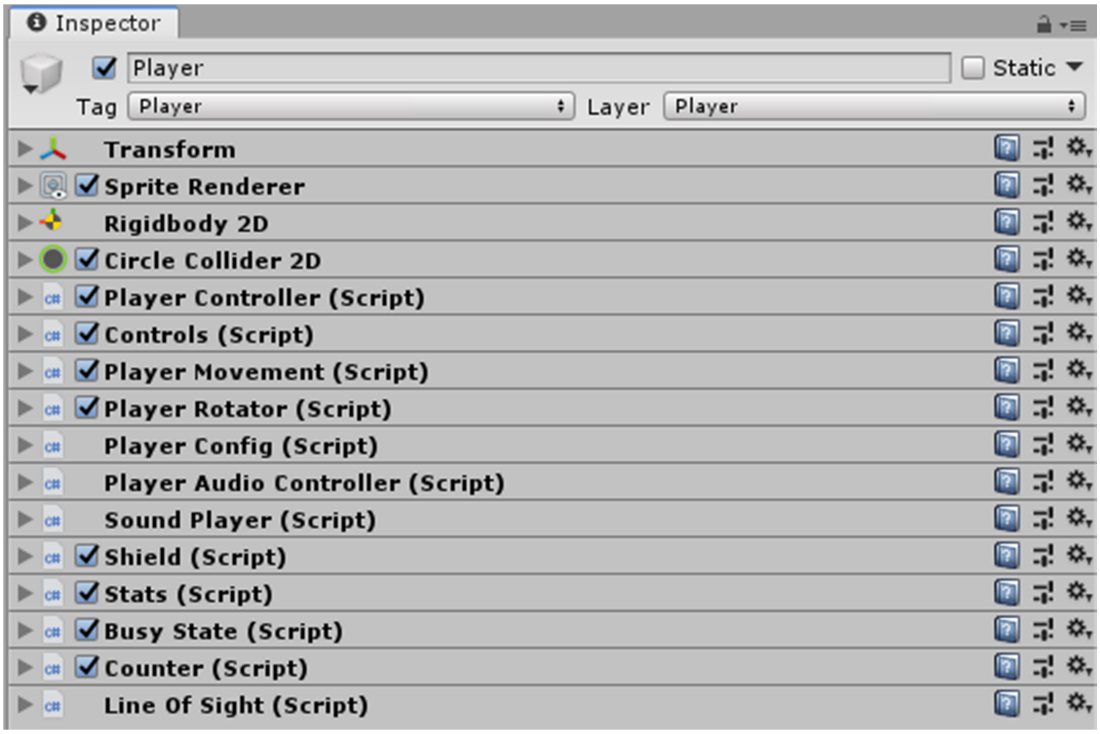
PART ONE

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



PLAYERS

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.



Collision

As seen in fig. 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.

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.

Controls, Player Movement and Player Rotator

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 called '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);
```

Joystick controls are also split into two, first is that the player controls the movement with left analogue stick and the rotation with the right, the second way is that the player controls the movement and the rotation with the left stick. The first way is how the game was meant to be controlled initially because it is similar to the way other popular shooter games control, left stick is used to move and right stick is used to point the gun which allows for very flexible controls. At first this seemed right as it has been explained in design chapter but after adding more features it turned out that right hand had to also be used for shooting, shield, attacking, changing guns and reloading. In a fast paced game this can turn out to be very difficult to manage and creates a very steep learning curve. A new way has been added that is easier for the new players. In GTA for example, players control the movement and the rotation with the left stick which leaves the right hand for other task. In a 3D game, Rocket League, players can switch between having the camera pointing at the ball all the time or having a classic Behind The Player camera movement. Having the camera pointing at the main point of focus makes the game a lot easier to control but it does not allow to easily look around the game area which can sometimes be crucial to the outcome of the game. Players can easily switch between two modes which adds flexibility and fits much more playstyles. This project uses both methods and just like in Rocket League, players can switch between the two control types so new players with rotation and movement in the left stick and once they gain more skill, they can switch to the other way for more flexible controls.

```
44              float lookHorizontal;
45              float lookVertical;
46              if (useLeftToRotate)
47              {
48                  lookHorizontal = moveHorizontal;
49                  lookVertical = moveVertical;
50              }
51              else
52              {
53                  lookHorizontal = Input.GetAxisRaw(playerController.playerConfig.horizontalR);
54                  lookVertical = Input.GetAxisRaw(playerController.playerConfig.verticalR);
55              }
56              playerRotator.Rotate(lookHorizontal, lookVertical);
57
58              // Sprint
59              playerMovement.Sprint(sprint);
60
```

Rotation

Rotation works in a fairly basic way, an angle is calculated by first finding the arctangent of the x,y coordinates then by converting it to degrees using the Mathf.Rad2Deg method and then by converting it to a Quaternion rotations 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 line 31 in fig..

```
21      public void Rotate(float lookHorizontal, float lookVertical)
22      {
23          if ((lookHorizontal > playerController.playerMovement.deadZone || lookHorizontal < -playerController.playerMovement.deadZone) || (l
24          {
25              h = lookHorizontal;
26              v = lookVertical;
27          }
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      }
```

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.

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

The deadzone value can be adjusted for a best feel but it should not be high because this could make the player movement feel disjointed. Line 23 figabove, and line 72 fig. below

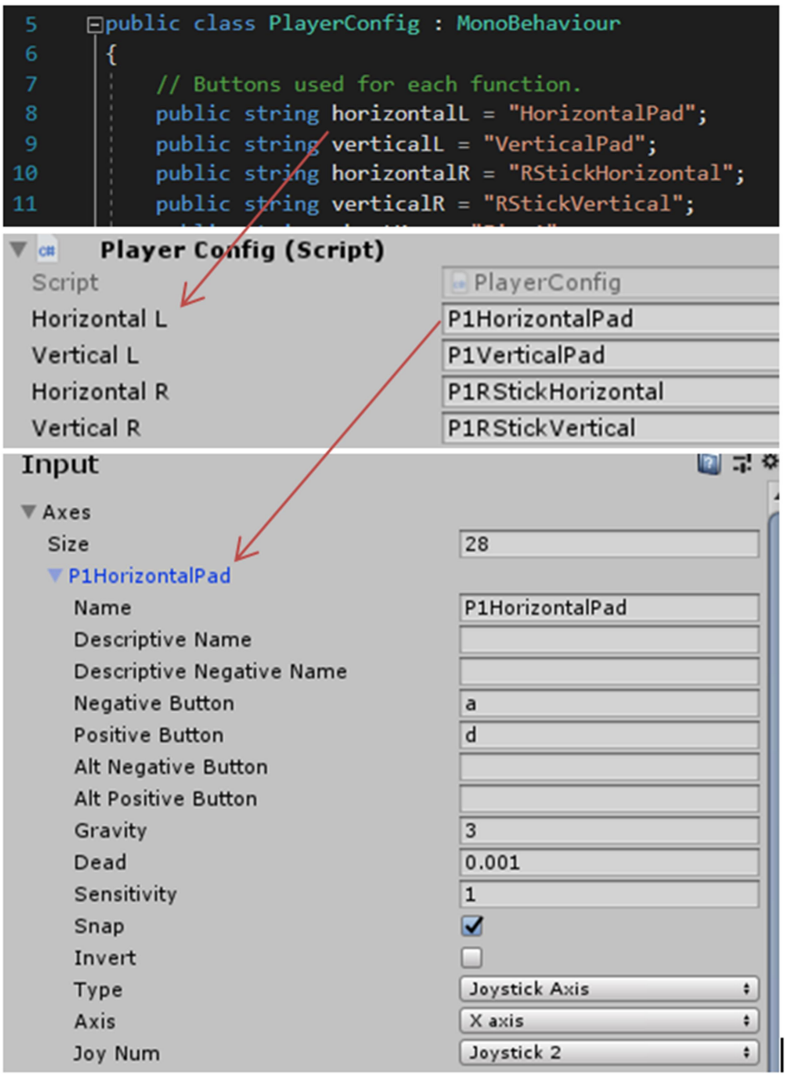
```
61      public void MovePlayer(float moveHorizontal, float moveVertical)
62      {
63          /* moveHorizontal and moveVertical would be a vector line in the wanted
64           * direction. Joystick X and Y axis are used to get the vectors. Keyboard
65           * arrows could also be used to 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; }
```

The sprint is a very basic method which checks the player's stamina status and if the player has enough stamina, it changes the speed float value by adding the sprint power which is stored in the Global Settings. It also drains the player's stamina level. The full method can be found in Appendix..

```
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;
109             }
110         }
```

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.. Fig. shows how the movement keys are translated. Each key can be then easily set up inside the Unity.



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 'busy state' 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 the cycle repeats.

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. Appendix. Shows the full class.

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 can be ignored.

Weapons

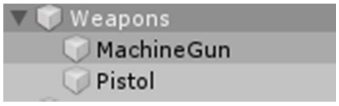
Weapons game object is a child of the player game object. It has three components – weapon controller, weapon list and the shooter. Weapon list component is a list with all of the gun game objects and it makes it easy to add new guns. Guns are held as children of the Weapons game object.

```
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)...
```

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 figbelow), which runs a GetNextGun() inside the WeaponList (line 22 figabove) 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()...
```

To add a new gun, a new object has to be created that is a child of the Weapon game object as seen in figbelow 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 an abstract class which inherits the MonoBehaviour base class and the IGun interface. It 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 figbelow.

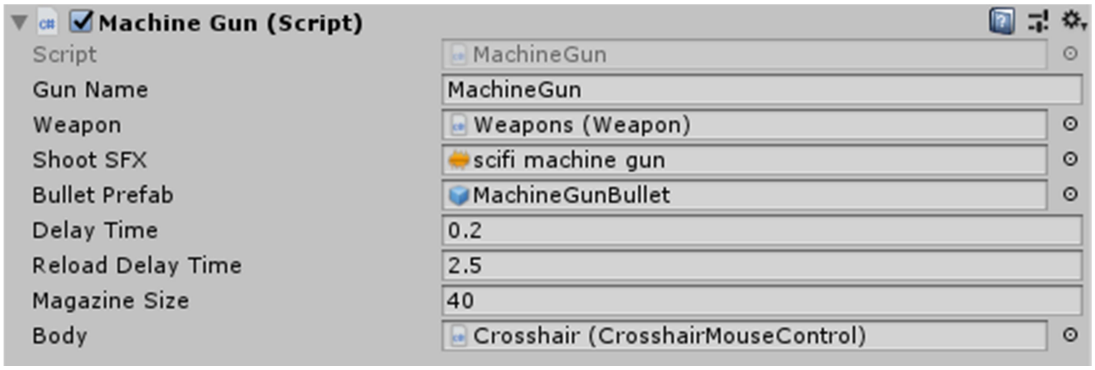
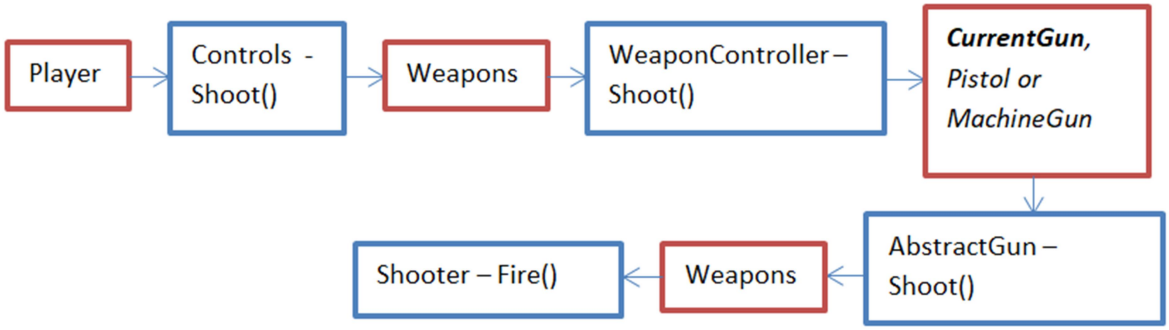


Fig above shows a machine gun, as it can be seen everything can be customised like the name, delay and reload time, magazine size, bullet and the SFX.

(MachineGun component is a new class that inherits the AbstractGun class)

```
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()...
```

Figbelow shows what happens when the player uses a Shoot method.

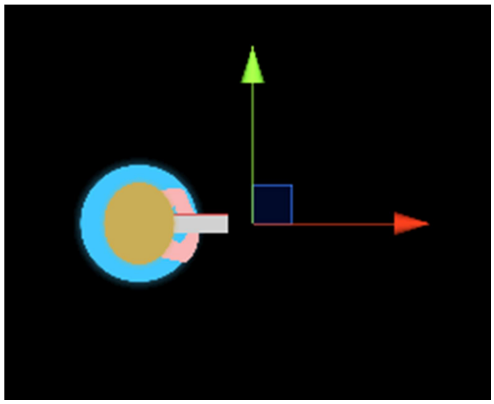


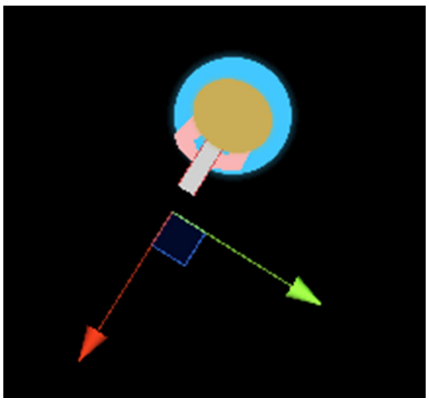
First, the Controls reference the Weapons method and run a Shoot() method inside the WeaponController, it then runs a Shoot() method on currently equipped by the player gun. Next the Shoot() method is run inside the AbstractGun, this method has the most complex behaviour. If there are not enough bullets 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 line 84 fig..

```
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();
94          }
95      }
```

Shooter component has been added to be able to instantiate bullets in random places to add more reality to the game and if there will be new variables like for example, wind that will affect the bullets, shooter class will be able to take care of that without having to add new functionality to the Shoot method. As one of the SOLID principles says, 'Methods should only have one responsibility'.

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. Fig. and fig. shows what happens to the fire point when the player rotates.





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 customise 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          }
```

```
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      }
```

A reload method checks if player has enough bullets and adds them to the magazine accordingly, line 54 and 64. It also runs a DelayFor(reloadDelayTime) method, line 56, 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      }
```

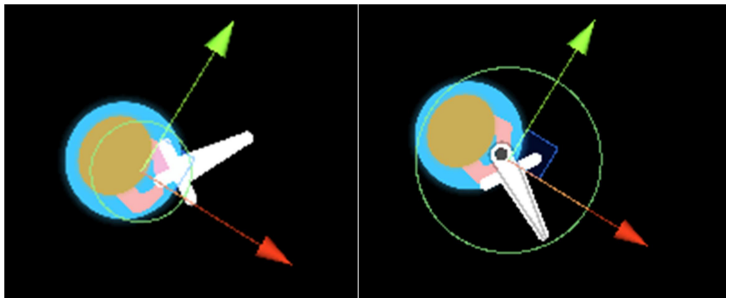
Sword

A sword is a game object with a circle collider and a sword controller component. The UseSword method inside the sword controller has a very basic task – check if the player has enough stamina (as in the Design chapter it has been explained, using the sword drains stamina) line 33 - 35, check if player is not in busy state line 33, then if not, hide the current weapon for the duration, line 36, activate the sword sprite and create an invisible circle collider that will push the ball, 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();
41
42          }
43      }
```

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. Fig. shows what happens to the collider (green circle) when the sword is used.

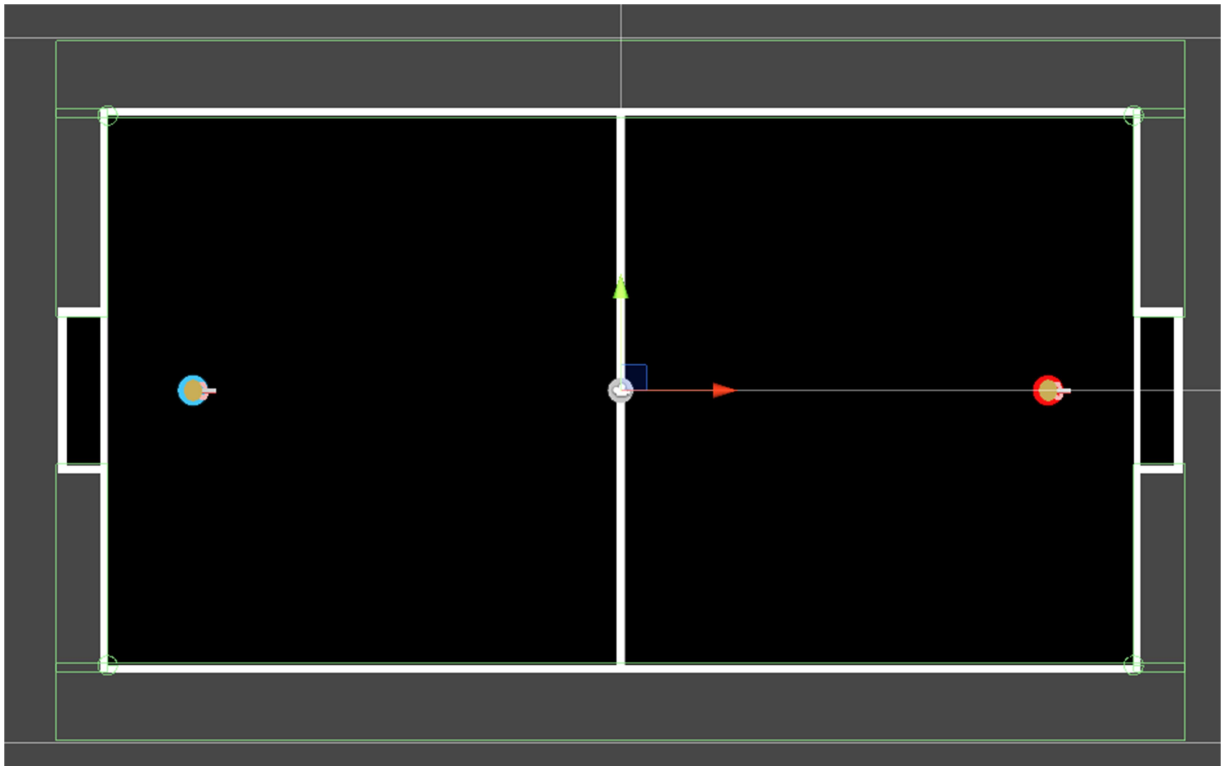


```
45      private void Push()
46      {
47          /*
48           * This function creates an invisible collider and pushes it in front of the player,
49           * it is needed to create velocity so when it collides with the ball
50           * it will push it back. The force (bounciness) with which 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);
71          }
72      }
```

As seen in fig. 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 pivot (which is the 'handle'). Since this project does not focus on VFX, the overview of the Animator can be found in Appendix.

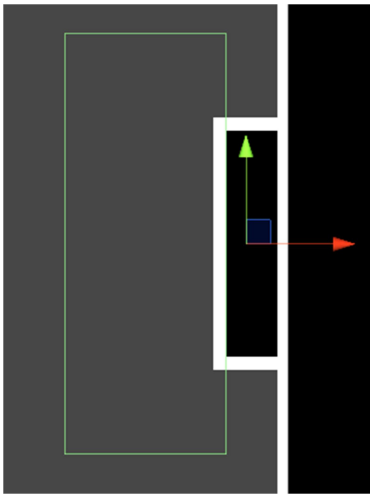
Playing Field

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.



The goals

The play area does not have any other components than the sprite which is just a rectangle with lines around it and one line in the middle. 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 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. At first it was planned that the goal would be just a straight line and whenever a ball would collide with it, it would count as a goal. However, after testing it turned out this is not the most effective and fair placement of the goals so new goals had to be designed. The new goal allow the player to get inside it similarly like in the game Rocket League, in that game a goal is only counted when the entire ball enter the goal. This allows the players to defend the goal more easily.

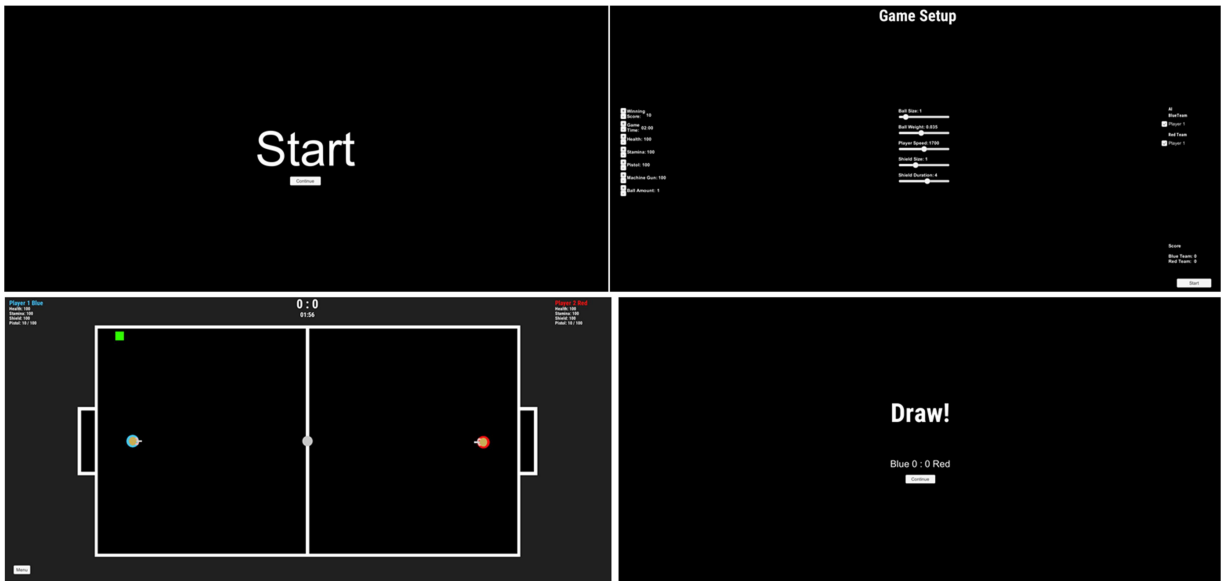


The ball

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. The other component is the Ball Controller which takes care of collision detection between it and the goal as seen in fig..

```
36      private void OnTriggerEnter2D(Collider2D other)
37      {
38          if (other.gameObject.CompareTag("LeftGoal"))
39          {
40              redTeam.GetComponent<TeamController>().AddPoint(1);
41              ResetPlayers();
42          }
43          if (other.gameObject.CompareTag("RightGoal"))
44          {
45              blueTeam.GetComponent<TeamController>().AddPoint(1);
46              ResetPlayers();
47          }
48      }
```

USER INTERFACE

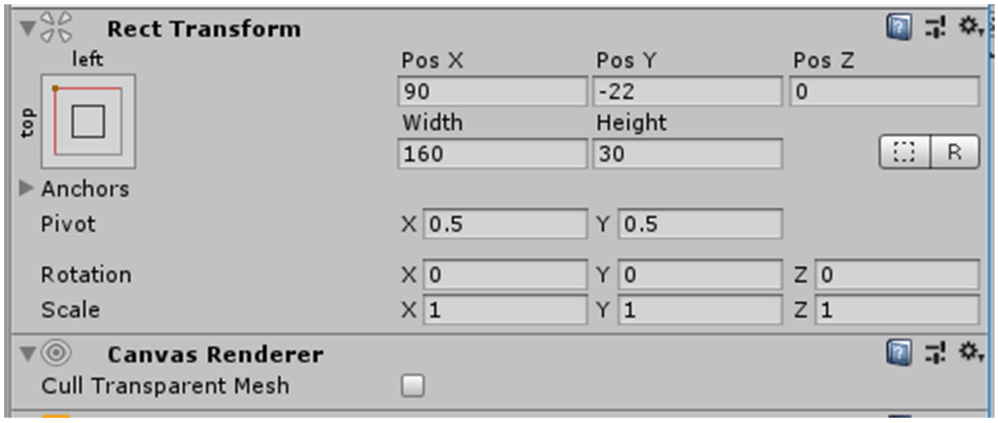


HUD

he 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 figureabove. The information that was meant to be shown to the user as mentioned in the design 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 player vs. CPU and player vs. player, game modes. So, 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.

However, this way is very limited because if there would be more than two players on each team, it would be hard to find space to show each player's status. The best solution would be to have a separate HUD for each of the game modes; one HUD would have the AI's opponent health only displayed above its character and hide everything else the other HUD would be designed for a player vs. player and have the HUD as it is but show the weapon and ammo in the bottom corners. FIGURE

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.



To show the score, a new game object has been created with a Score UI Controller component inside. It:

- takes two TeamController components

```
 9    public TeamController blueTeam;
10    public TeamController redTeam;
11    public Text scoreText;
```

- extracts the scores

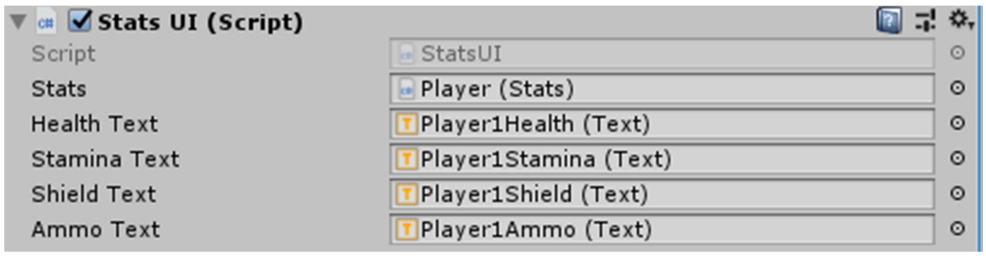
```
19        int p1Score = blueTeam.GetScore();
20        int p2Score = redTeam.GetScore();
```

- creates a string out of them
- passes them to a Text component

```
22        scoreText.text = p1Score + " : " + p2Score;
```

Stats

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 … 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.



```
 8        public Stats stats;
 9        public Text healthText;
10        public Text staminaText;
11        public Text shieldText;
12        public Text ammoText;
```

Figure… shows the update loop, it is very basic since it only has one task – to convert the status into a string.

```
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        }
```

The Game Setup

The game settings seen in panel two in figabove, 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 few different objects and components. As seen in fig… a button that changes the amount of health each player starts with 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.

Each text is done 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.

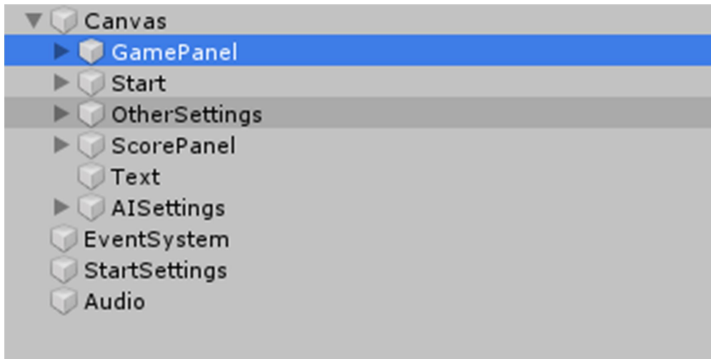
The methods are very basic because their only role is to either add or subtract from the value inside GameStartSettings. Fig… shows an example of that methods, as it can be seen, a value is passed to the method which is then added to the given variable. Fig… shows how that method is used inside the button, 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. Fig. shows the AddHealth method that is called when the Add button is pressed.

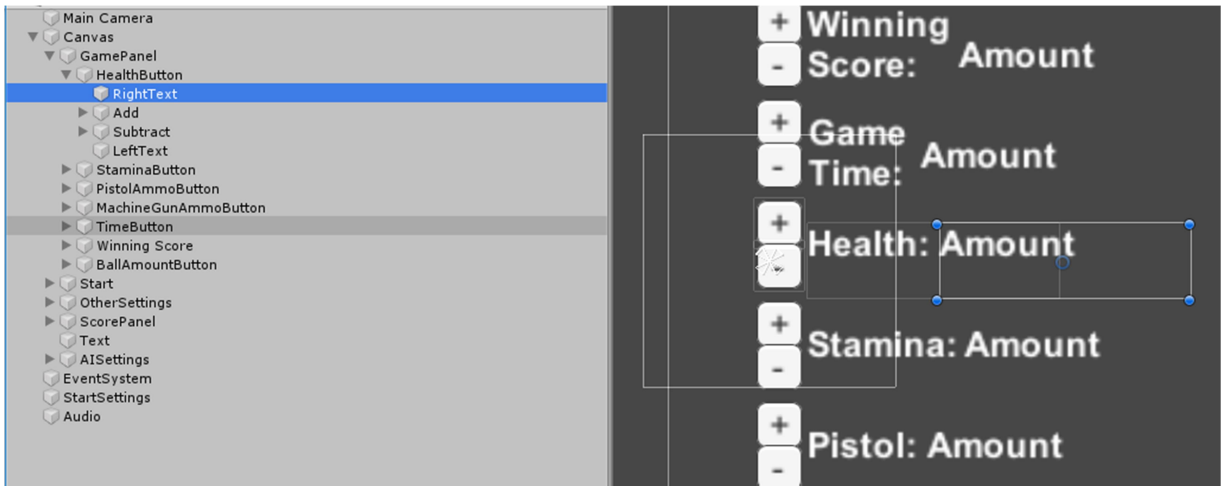
```
0 references
public void AddHealth(int health)
{
    GameStartSettings.health += health;
    if (GameStartSettings.health < 10) {GameStartSettings.health = 10;}
}
```

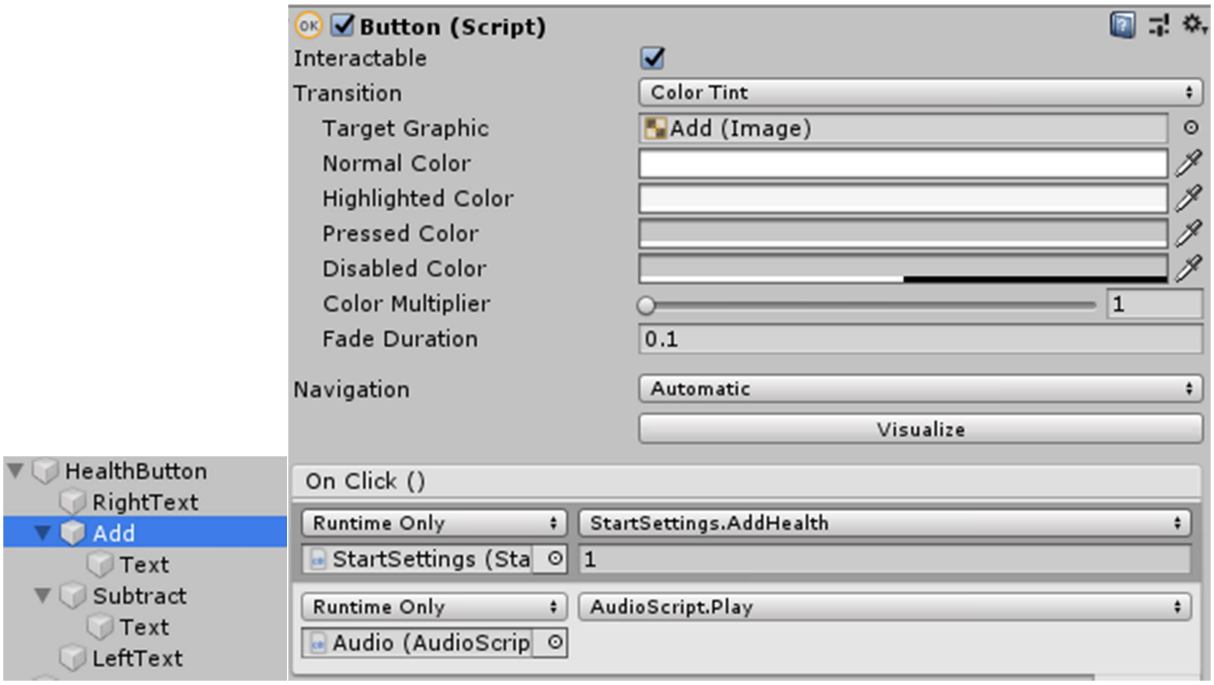
Some objects have a slider instead of two buttons to fit the variable much more. Slider allows users to select a numeric value by pressing on a button and dragging it left or right. The slider component allows calling a method when the value changes, change the minimum and maximum value or set the starting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

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

https://docs.unity3d.com/Manual/script-Slider.html







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.

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 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.

Part Two – Artificial Intelligence

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 various groups, and each behavior is implemented as a static method inside a Static Class. Implementing behaviours as static method seemed like a best way because it allowed to separate the part that takes care of the logic with the part that

does an actual method. This part will first explained the behaviours that the AI can choose from and then will go on to explain the logic behind it – that is, how it chooses the behavior.

The behaviours are separated as follows; Basic Behaviour, Direction Behaviour, Global Behaviour, Movement Behaviour and Team Behaviour. There are also helper methods that are not actual behavior, i.e. they do not do an actual thing but they are closely correlated with the methods inside the behaviours, these are; Calculate, General and Check.

Basic

Basic Behaviour 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. Figure… shows the first method, GetController which takes out the Controls component from the player GameObject. Figure … 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)

Appendix.. shows all of the methods.

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

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

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. However, this only works if the object is a pickup or another player game object, if it is a ball, more steps are needed.

Movement Behavior Class takes care of all things that have to do with player movement and player rotation. These methods are slightly more complex than in the Basic Behaviour Class 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. Fig.. shows the Look At method which finds the x and y coordinates of the object in relation to the player, and rotates player towards it.

```
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);
17  }
```

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. Fig. and fig. shows the first Move Forward method.

```
3 references
100 public static Vector2 GetRotationVector(GameObject player)
101 {
102     /* This method is used to find the direction vector of the player's rotation.
103      */
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);
108 }
```

```
26  public static void MoveForward(GameObject player)
27  {
28      /* This method moves player forward but it does not
29       * take care of the direction. To make the player run to the ball
30       * use RunToTheBall method.
31       */
32
33      Vector2 dir = AIDirectionBehaviour.GetRotationVector(player);
34      AIBasicBehaviour.GetController(player).playerController.MovePlayer(dir.x, dir.y);
35
36  }
```

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

The second method, shown in fig… finds the x and y position of the object and call the first method.

```
10 references
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  }
```

The first method however, is much more complicated because the x and y coordinated have to be converted to a direction vector. Fig. shows the Move Towards 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 and lastly, the player object is moved forward by using the Move Forward method mentioned above.

```
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);
70  }
```

Direction – Guiding the player

This introduces a new set of behaviors which is the Direction Behaviour 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.

Fig. shows the GetRadian method, along with its explanation.

```
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;
23  }
```

Figure. Shows the GetDirectionVector method which converts a radian to a directional vector. A more basic version of this method is used inside the MoveTowards method, since a radian is first found by using the GetRadian method, then the GetDirectionVector is found separately. Fig. shows that method.

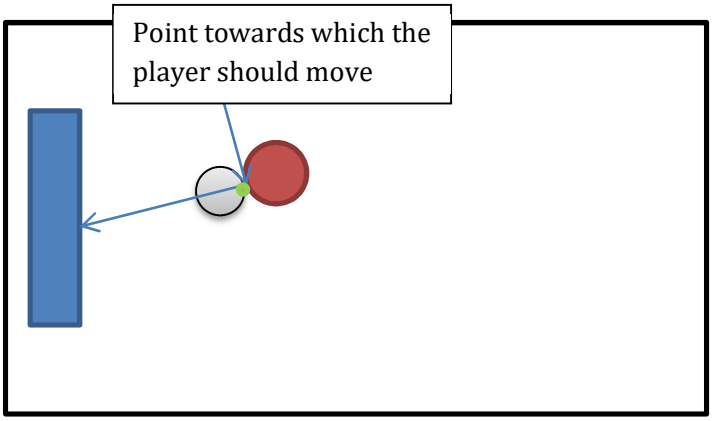
```
    1 reference
33  public static Vector2 GetDirectionVector(GameObject source, GameObject destination, float offset, float size = 1)
34  {
35      /* This method is the second part in the three part methods which
36       * converts a radian to actual position.
37       * This method returns a vector in the wanted direction. It is used with
38       * the GetRadian method to get the exact vector in which position to move.  */
39      double radian = GetRadian(source, destination, offset);
40
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);
45  }
```

The next method is the FindPositionOf which is the reversed version of the GetDirectionVector method. Fig. shows the method and its explanation.

```
    3 references
56  public static Vector2 FindPositionOf(GameObject source, GameObject destination, float offset)
57  {
58      /* This method is a reversed version of the GetDirectionVector method.
59       * It finds a direction vector towards a wanted object (destination) and
60       * then finds the x and y coordinates of that direction vector.
61       * It is used when guiding the AI to avoid the ball. */
62      Vector2 directionVector = GetDirectionVector(source, destination, offset);
63      float x2 = directionVector.x;
64      float y2 = directionVector.y;
65
66      float sourcex = source.transform.position.x - y2;
67      float sourcey = source.transform.position.y - x2;
68
69      return new Vector2(sourcex, sourcey);
70  }
```

The first step as explained in the Design chapter, is to create the player movement, this is the most important step because without the AI movement, the player will not be able to score goals. As explained, the 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 method 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. Fig. 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; fig. shows the first version of this code which returns it as a Vector2.

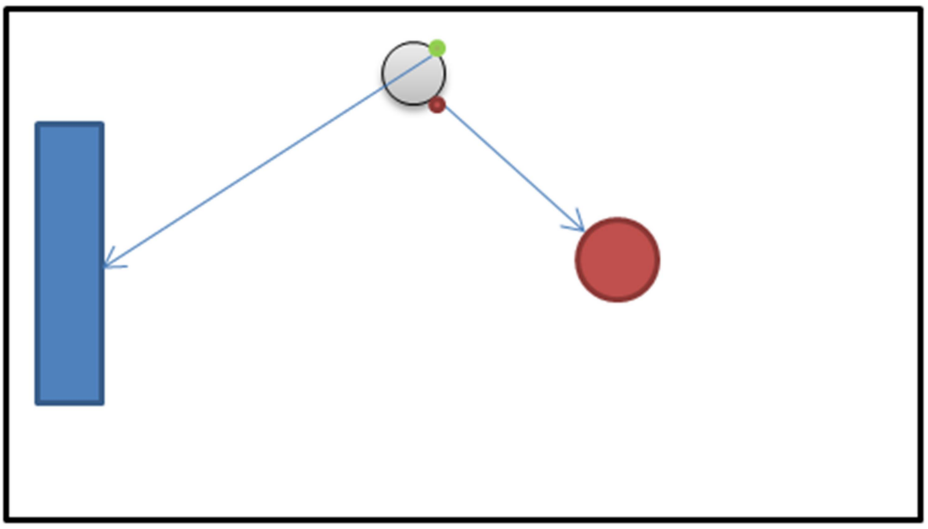



```
92      public static Vector2 GoRoundTheBall(GameObject ball, GameObject goal)
93      {
94          double goalAngle = AIDirectionBehaviour.GetRadian(ball, goal, 1);
95          Vector2 des = AIDirectionBehaviour.GetDirectionVector(goalAngle);
96          return AIDirectionBehaviour.FindPositionOf(ball,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

However, there is one more issue, if the ball is between the player and the goal, the player game 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 round 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 in fig above but the radian has to be inversed since now it does not have to be between the player and the ball. Fig. shows two points, green and red, green shows the point acing the goal and the red point shows the point facing the player. Fig. shows the first part of the code.

```
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
```



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. The equation is:

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

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, as can be seen in fig. The class is very simple as its only job is to

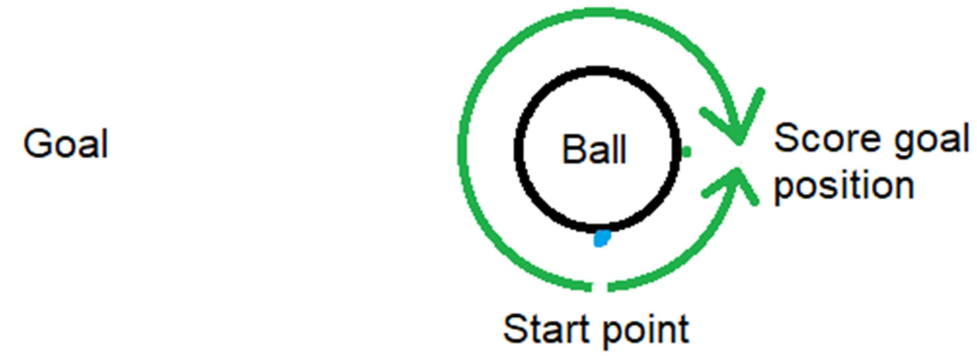
cycle from 0 to 1 in a steady step. Fig. shows that class. The value added in line 14 determines the speed, this has been carefully chosen to match the player speed.

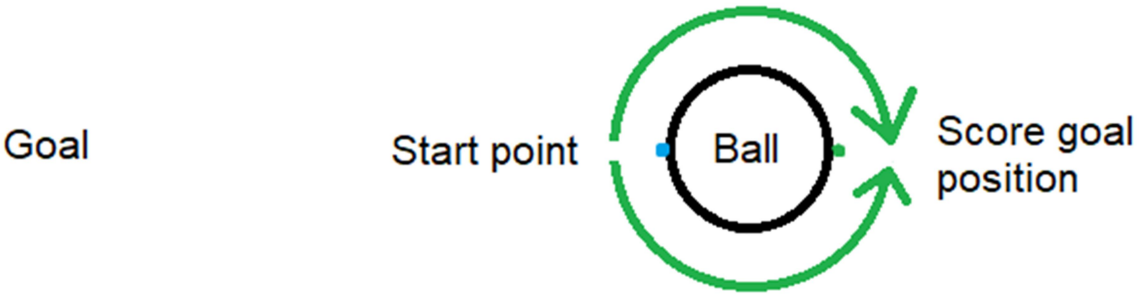
```
5   public class Counter : MonoBehaviour
6   {
7       public float a;
        0 references
8       void Start()
9       {
10          a = 0;
11      }
        0 references
12      void Update()
13      {
14          a += 0.01f;
15          if (a >= 1)
16          {
17              a = 0;
18          }
19      }
20  }
```

Fig. shows the last part of the 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;
63          Vector2 des = AIDirectionBehaviour.GetDirectionVector(newRadian, 1.2f);
64          Vector2 pos = AIDirectionBehaviour.FindPositionOf(ball, des);
65          return new Vector2(pos.x, pos.y);
66      }
```

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. Fig. shows the old start point, if it was kept like that, the player character would have to circle almost whole ball to position itself. Fig. shows its new path.





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 inside the Global Behaviour Class, this class takes care of the main movement logic like positioning in front of the ball; positioning and shooting at the goal; and shooting the ball in the middle of the pitch. Fig. shows the first method – this method shows how the method above has been implemented.

```
8   public static void PositionInFrontOf(GameObject player, GameObject ball, GameObject goal)
9   {
10      /* This method position the player in front of the given object.
11       * It is mainly used to position the player in front of the ball in such a way that the ball is
12       * between the player and the goal.
13       */
14
15      Vector2 coordinates = AICalculate.FindPointBetween(player, ball, goal);
16
17      /* Now we know what is the position of the point that the player will move towards.
18       * Next we need to move the player towards that point.
19       * MoveTowards does not require the player character to look in the
20       * certain direction so looking and moving can be done separately.
21       */
22      AIMovementBehaviour.MoveTowards(player, coordinates.x, coordinates.y);
23  }
```

The next method is PositionAndShoot which 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 towards the goal. Fig. shows the full method, as it can be noted there is no need to calculate the points mentioned in fig. so when the player is farther than 2 (around 20 pixels), it only just moves towards the ball. The third method is very similar to the one shown in fig. but there is an additional method call before the UseSword, which rotates the player to look at the middle of the pitch.

```
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      {
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          }
45      }
46      else {AIMovementBehaviour.MoveForward(player);}
47  }
```

The last method that needs explaining is the CalculateLengthBetween used in the above methods. This method uses the distance formula which is show in eq… Fig. shows how it has been implemented.

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

```
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)));
18   }
```

The last class, CheckBehaviour, has helper methods – CheckIfCloseToPickup, GetClosestPickup, CheckIfFarFromBall, CheckIfCloseToOpponent, CheckOpponentsHealth and CheckIfBallApproaching.

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.

THE LOGIC

The last part to explain is the logic, how the player chooses its behavior. As it can be seen, each of the above behaviors creates a basic tree - each behavior builds upon other behavior to create more complex behavior, fig. shows how PositionAndShoot connects with other behaviours. Similar approach will be used for the logic as explained in Design chapter. Since the 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 behavior is chosen by splitting the play field into four areas, and later goes into more detail how it then chooses a smaller set of behaviors based on the position of the ball until it finally chooses a concrete behavior.

FOUR STATES

As explained in the design chapter (fig.), there are four states – (opponents goal) Shoot, Attack, Tackle and Defend (player's goal). 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 int value, "State Boundary". The logic, shown in fig., that chooses the behavior is stored inside the AIController component which main task is to run a correct state inside a FixedUpdate method. As it can be noted it is a very basic 'if' statement that chooses the state based on the x position of the ball. Full Class can be found in appendix…

```
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          }
```

Each of the states inherits a AIState abstract class which has all necessary methods that should be implemented and references to objects that are used inside each state. The full class can be found in Appendix.. but the most important methods are Run (seen in fig. line 50, 54, 58 and 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 method from the GlobalBehaviour.

FOUR POSITIONS

The more important methods are the Run method and RunPosition methods. Run method 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 method. Fig. shows the full method.

```
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      }
```

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 ^ (¬LO^¬LP) => G ^ LB

Fig. shows how Zero Position has been implemented. Appendix.. shows the full method but the same principle is used throughout.

```
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          }
```

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 Reponsibility Principle which says that every module should only have one responsibility. Unfortunately this was a faster and safer way to implement that.

https://itnext.io/solid-principles-explanation-and-examples-715b975dcad4

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 method inside the Check Behaviour. This follows the same principles as the states and run methods, this time the player should behave differently depending on the distance it is 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. Fig. shows a RunZeroPosition method 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      }
```

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 is a total of:

$$4 \times 4 \times 2 = 32$$

AI DESIGN ADD APPENDIX WHERE EACH OF THESE METHODS EXPLAINS WHAT IT DOES

CHAPTER 5 – EXPERIMENTATION AND EVALUATION

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

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 may be 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 very low; otherwise it would be very difficult to gather a group of people to play the game. Each group of answers to each question will be evaluated one by one.

The questions were designed to gather answers which would then be developed into themes which would then refer back to the aim and objectives. 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 feel. The third question focused on the Graphical User Interface – how it feels and why; it was one of the objectives to create an intuitive, easy to use GUI. 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 projects which is the AI. The last two questions asked about the overall experience of the game – what people liked and disliked about the game.

OVERALL EXPERIENCE

One of the main aims and objectives 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”.

GRAPHIC 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.

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."

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.

CHAPTER 6 – CONCLUSIONS

What have you learnt?

Basically, "reflect" on the work you have done.

What additional features/extensions can be done to the work and/or what would you have done if you had more time.

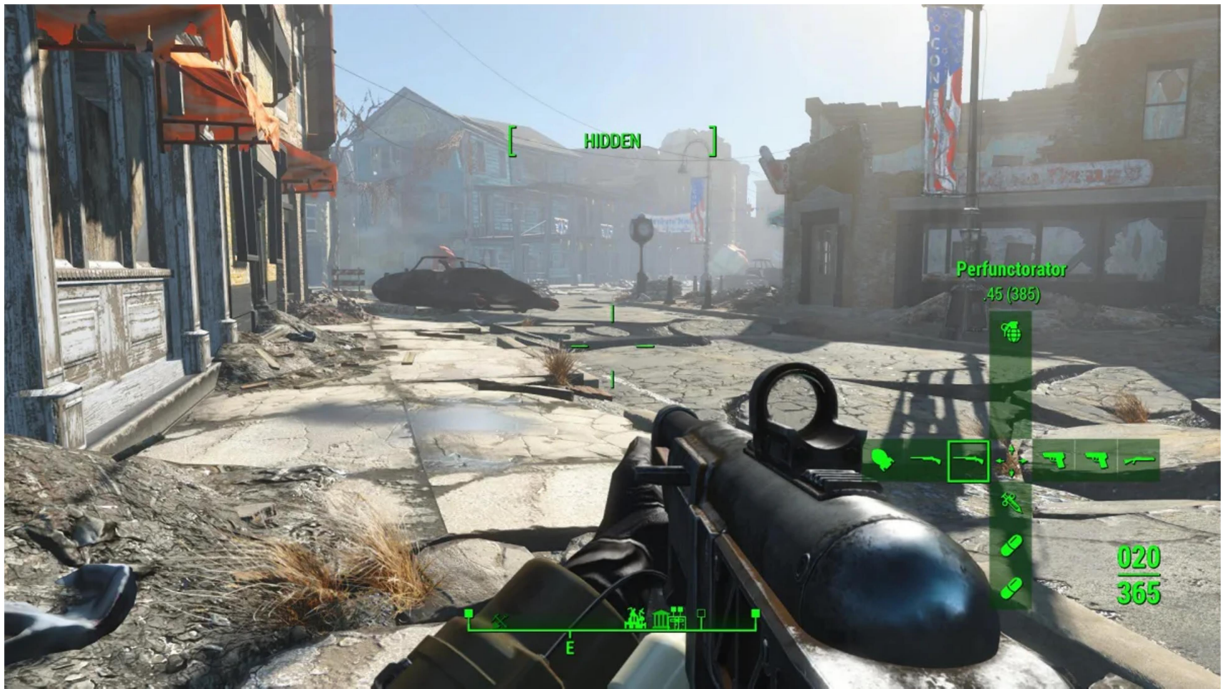
APPENDICES

Include design diagrams, data formats, etc.

Basically, don't overfill the report.

Link to the GitHub repository

Game UI


HIDDEN
Perfunctorator
.45 (385)
020
365
E


4:54 AM / CLEAR
THE FALL OF THE HOUSE OF REARDON
Kill the monsters in the manor.
Warg
5 Viziman Champion
10 Swallow
35 Blunt crossbow bolt


4:00
YOU
PACKET LOSS
INDICATOR
100
BOOST


DIVISION RIVALS
28:36
AVE
1-1
Ban
R1 CALL FOR 2ND PLAYER
R2 CHANGE KICK TAKER
L1 SWITCH RECEIVER
L3 CHANGE CAMERA
R ADJUST START POSITION
TARGET ON/OFF
7 ROBERTO FIRMINO
94 FABINHO
START

OnDisable is called when the object is disabled or destroyed and is usually used for cleanup code. OnEnable is the opposite, it is called when the object is enabled.

OnGUI as the name suggests is used for rendering Graphic User Interface Events like clicking a button on a screen, using GamePad to navigate the GUI etc. Therefore it may be called for each event – several times per frame. This was also not used in this project because the Graphic User Interface was implemented by using the built in User Interface objects.

```
1 reference
public class Shield : MonoBehaviour
{
    public PlayerController player;
    public Transform shield;

    0 references
    public void Update()
    {

        if (player.stats.GetShieldStatus() > 0)
        {
            shield.transform.gameObject.SetActive(true);
            float shieldDuration = GameStartSettings.shieldDuration - 3.5f;
            player.stats.AddShield(-4 + shieldDuration);
            float size = (player.stats.GetShieldStatus() / 100) * 4f * GameStartSettings.shieldSize;
            shield.transform.localScale = new Vector2(0.25f, size);
        }
        if (player.stats.GetShieldStatus() < 1)
        {
            player.stats.SetShield(0);
            shield.transform.gameObject.SetActive(false);
            player.busyState.SetState(false);
        }
    }
    1 reference
    public void UseShield()
    {
        if (player.stats.GetShieldStatus() == 0 && !player.busyState.GetState())
        {
            player.busyState.SetState(true);
            player.stats.SetShield(100);

        }
    }
}

```

REFERENCES