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

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)

CONTENTS

CHAPTER 1 - INTRODUCTION

State the problem you are trying to solve

Why is it worth tackling?

What approaches are available (briefly)?

What approaches have you chosen?

Any special knowledge you presume of the reader

Any special typography or terms

A “road map” of the report . . .

CHAPTER 2 – BACKGROUND

WHAT IS UNITY

Any information the reader requires in terms of techniques/ technology that isn’t part of the programme you have studies.

Methodology

Research philosophy

What are frames

In gaming, frame rate is the rate in which the image on the screen is refreshed. For example, movies are made up of moving images and most are projected at 24 frames per second, there are 24 still images shown, one after the other, in one second. This creates a very natural feel – most people would not be able to tell the difference between 24fps and 48fps. This value has been carefully chosen for the best

experience and space requirements since a 48fps movie file would be twice the size. 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 does not affect the size of the game but is itself affected by the hardware. A high end graphics card would be able to render more frames in one second than a low end one. 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 for example, 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

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

When using C# in Unity it is important to derive every class from MonoBehaviour because it is the base class from which every other class derives. 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 there is no need to explain them in detail. 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, this is explained in ….

The most important functions are Start and the three Update methods. Start method is called in the same frame that the script is enabled and before its Update methods. Start is great to initialize the game objects by enabling the developer to set up the object before the object is updated. 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.
http://ilkinulas.github.io/development/unity/2016/05/30/monobehaviour-constructor.html

The Update methods are very similar to each other with slight variations in how they are called. The Update method is called every frame and is used, as the name suggests, to update the game object it is attached to. 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);
    }
}
```

LateUpdate is just like the Update method – it is called every frame but it is called only once all Update methods, inside the object and inside other objects, have been called. An example where this would be used is when following the player with a camera, updating the camera position should be done only once other objects' position has been updated to avoid issues and unexpected behavior.

The last method, FixedUpdate, is called every fixed time not dependent on the frame rate. It is mostly used to calculate physics and to avoid issues various issues with time dependent methods. For example if the player has to move right at a certain speed, its position should be updated using the FixedUpdate method. 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

In Unity scenes contain environments, objects and the menu for the game. Each scene is like a level with its own objects, 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

How User Interface is implemented in Unity

The user interface in Unity is easily implemented by creating a Canvas. All user interface elements should be inside the Canvas area. As explained in … 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. Event system has a very basic functionality exposed only because it is designed as a manager and coordinator of messages between the modules. 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 to make the user interface work properly automatically. Default settings were used for this project therefore there was no need to set everything up manually.

In game UI

Game Setup

CHAPTER 3 – ANALYSIS, REQUIREMENTS AND DESIGN

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 rules of the game are a lot different than in a classic football game, first, the players are allowed to attack the opponents and even get points when they manage to kill him. Players get weapons like a sword, gun and a shield to do so. The teams are also much smaller and should first be limited to only one player on a team to make it of manageable size to finish, especially since a lot of focus in this project is on the Artificial Intelligence aspect. 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

The 'camera' is pointed from above like in games like Grand Theft Auto, Hotline Miami, Overcooked or Undertale. This point of view and 2D graphics have been chosen because it is much easier and faster to work with and even though it may seem dated, the gaming community and the market for these type of

games is still very big. For example, Undertale, released in September 2015 was one of the best-selling games on the digital game market Steam, with over 530 thousand copies sold by the end of 2015 (Galyonkin, 2019). Undertale uses top down view and retro, 2D graphics in its gameplay and it does not stop people from buying and enjoying it.

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 he needs to achieve in order to success in it. 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 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 – arrows to move, A to jump and B to sprint or shoot. The goal of the game is to collect coins, jump on platform 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 this basic premise and is still an inspiration to many games.

Controls

PICTURE

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, not only left, right, up and down that the keyboard (arrows) allows. (Although it will still be possible to use keyboard and mouse to control the game). A full 360 movement can only be achieved with analogue sticks on a modern 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.

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 a lot 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 games because they have to manually keep looking at the ball – there is a little arrow pointing at the ball to make it easier to locate it. The second type is much easier to controls because the players only have to focus on the movement and on trying to score a goal which is the main point of the game.

The play area

Before explaining the game loop and the goal of the game, it makes most sense to first explain the game area. The game area is a classic football pitch that is a rectangle with a line in the middle separating it into two; each side 'belongs' to each team. It 'belongs' to a team in a sense that the specific team will only spawn on its half of the pitch - this part is just like in the football. There are also 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 spawns 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 the travel much faster or use guns to hit (push) it from a distance. The players can score points either by killing the opponent team – for each kill they get one point. Players also get points for kicking the ball into the goal as mentioned above.

Size of the team

There is only one player on each team but the game could be expanded to more players, even up to 10 football although that would put a heavy load on the computer. This size of the team has been chosen because of the complexity of Artificial Intelligence on which this project is focusing on. 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 also have to make sure it does not get in the way of its own team mates. More team mates would also mean that in a single player mode (which is when one actual human plays the game) new AI would have to be created which would be in the team with the human player. 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.

HP, SP, SS and Ammo

Players have few 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 (which is also the maximum they can have), if it goes below that, they can pick up boxes which replenish their Health Points by a certain number. The amount of HP that each box replenishes is a random

number between 10 and 50. 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.

Weapons

To kill the opponents, players have few weapons to use – a sword, a pistol or a machine gun. A sword drains 4 to 6 points of stamina and deals 10 to 15 damage. Stamina points also drains 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 sword. Stamina replenishes around 1 point per frame (multiplied by the time.deltatime). Going back to the weapons, players also have guns and each gun has a different speed, ammo, reload time and strength. Pistol for example is much slower and has a smaller magazine size than the Machine Gun but it deals more damage and pushes the ball harder. Machine guns on the other hand have big magazine sizes, that is, they can shoot more bullets before they need to be reloaded, and they are faster and have a shorter reload time. Players can pick up ammo from the random boxes that spawn every 10 to 40 seconds around the map.

Shield

The last thing that players have at their disposal is the shield which is a half circle that spawns in front of the character for a few seconds which gets smaller and smaller until it disappears completely. The shield lasts for around 2 seconds and can be used as much as possible. It acts just like a wall that is, whenever a ball hits it, it gets bounced back so it is very useful when defending a goal. Its size allows 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 gets no damage when the bullet hits the shield. The downside to the shield is that the player cannot use a gun or a sword during the time that the shield is up.

Designing the UI - HUD

The User Interface is 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 RPG elements to therefore giving the player all these information is very important. Role-Playing Games have a much more focus on the game mechanics and the game is driven by the player. That is, the player makes all of the decisions, chooses what to do, the player character skills and usually makes in game decisions which steer the story in a unique direction.

On the other end there are games that are more like the movies, which means that in a way they play themselves. 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 in the middle, there will be enough information for the player to understand what is going on but not too much where the player is distracted. 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 as possible to avoid spending too much time on implementation since only few days will be given for that. The plan is to have a bar for health, stamina and shield in one of the top corners of the game and the weapon and ammo in one of the bottom corners.

The Menu will be broken down into four scenes; Title Screen, Game Setup, Main and End Game.

Title 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, starting stats (health, stamina etc.), type of opponent (another player or CPU) and other, more trivial settings like shield size or ball size. 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 of the 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.

Additional

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.

https://guides.gamepressure.com/horizon_zero_dawn/guide.asp?ID=38878

TABLE FOR POINTS

TABLE FOR GUNS

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 has been the most successful so now the developers are mainly focusing on updating it.

This game will be designed around two games 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 human player against another human player, also known as couch co-op, and the other will be human player against the Artificial Intelligence.

The first mode should be relatively easy to implement once one player character is working since the player game object would just have to be duplicated. Then the control buttons will have to be changed to fit another gamepad/keyboard, 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. This is explained further in the User Interface chapter.

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 HUD is also explained in the User Interface chapter.

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.

TEAM

To make the game independent of the amount of player teams had to be made. Since there are only two goals, there can only be two teams – Red and Blue. Each player will be given it Team component which will tell what who are his teammates, opponents, which goal he should focus on, starting position of each player and the score of the game.

This will allow to avoid having a score inside the player game object so adding new players will be much easier. The Team component could have a method like add score, so when a players health reaches 0, opponents abstract Team component will have to be called with the add score method. The team component could either be Blue Team's or Red Team's, this allows to have objects independent of each other.

1 vs 1

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.

Walls

The goals

Ball

Players

-actions etc.

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

Designing the artificial intelligence for a game like this creates a number of problems and questions that need to be taken care of. First issue to address is to find out how the AI will behave in a 1vs1 game, most of this project will focus on that part because this will be a foundation for other types of AI in game. Second issue is how the AI will behave in a team in a 2+vs2+ game; how it will recognize who are its teammates and how it will take advantage of playing in a team. The third issue is similar, but this time its teammate will be the user, so it will be important to solve how the AI will react to an unpredictable player. This section will address these issues and present the designs for each type of AI.

Movement

Point of focus

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.

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 seen in figure …. So the second step would 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. For example, when the AI has chosen to score a goal but the variables have changed before it is finished with it and it is now not the most effective task to do, it would be able to easily abandon it but keep the information gathered in the previous task and do something else without having to start at the first step.

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

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

PICTURE OF THE GOAL

How it chooses what to do

Break it down into 4 parts based on the position of the ball

Break it down even further into another four task based on the position of the players and the ball

Choose the best task

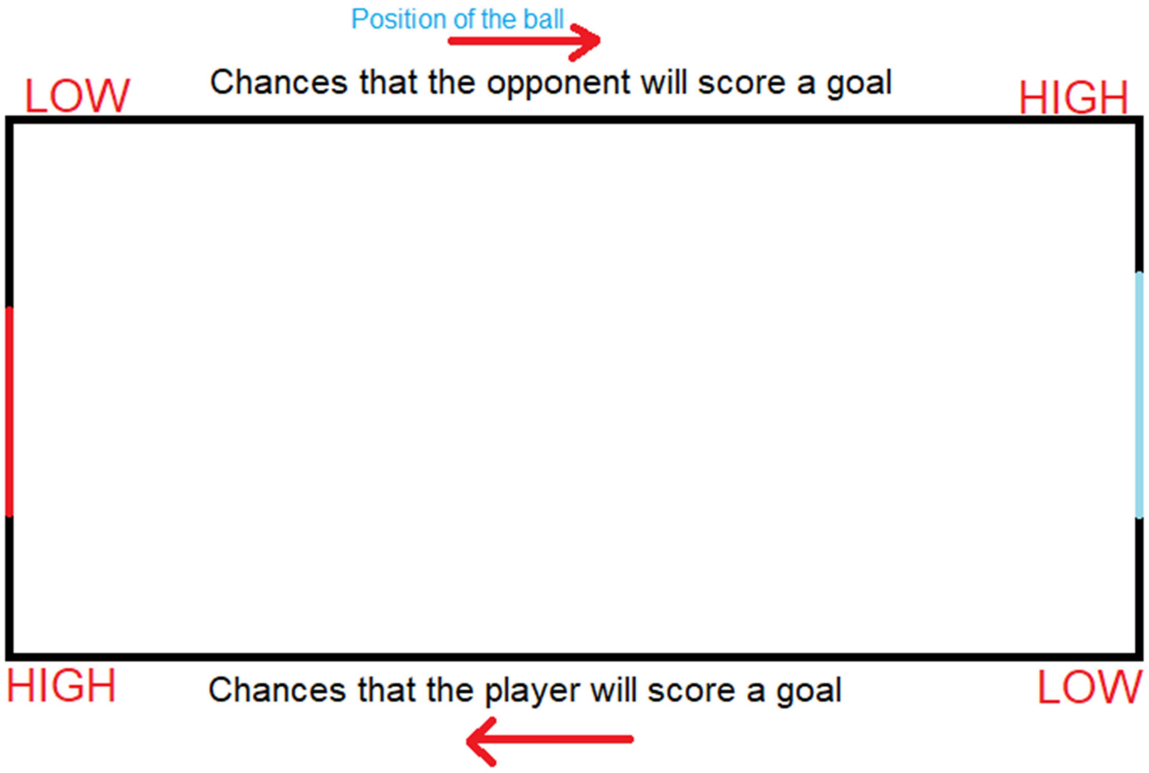
PICTURES

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. 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.

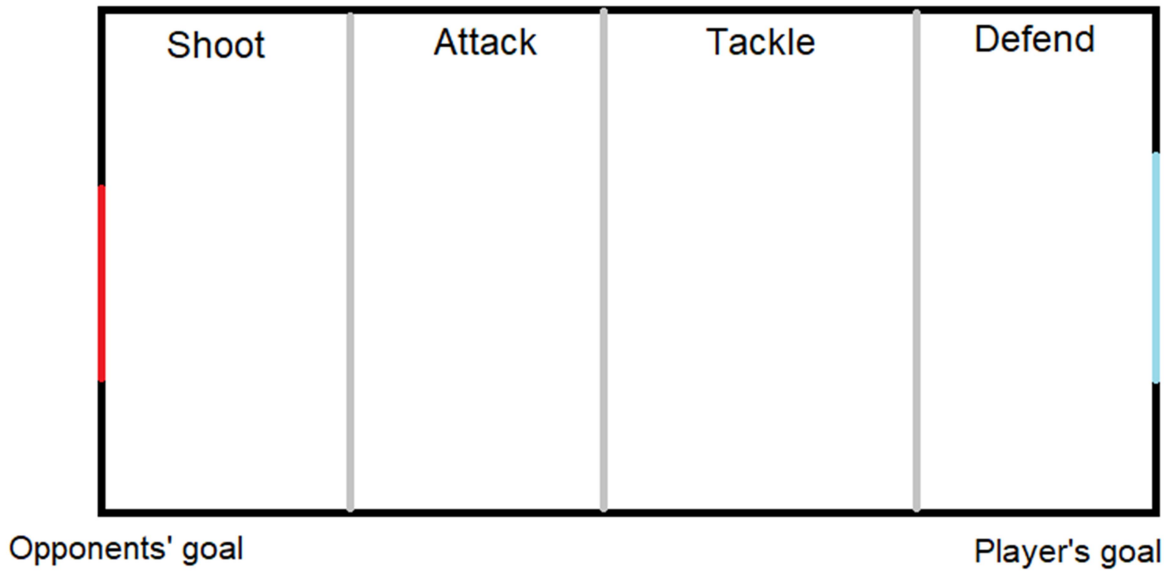
The first step is to look at the behaviour more generally, as seen in fig … a basic behaviour is broken down into three or four smaller task, to keep things simple, a similar approach will be used. The general goals of the player are to:

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

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



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.

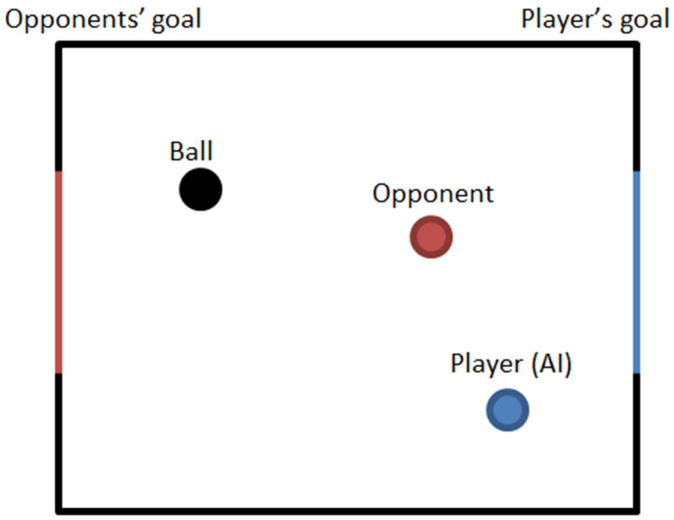
Next step would be to find a way how to break down the tasks even further. Here the position of the player and the opponents could come in helpful. Since it makes no real difference where the players and the ball are position on the y axis it will be ignored for now. It may make some difference in a game where there is more than one player in the team but for now the focus will be on a 1vs1 game.

Therefore the AI player should behave differently depending on its x position relative to the ball, opponent and the opponents' goal. For example, if the ball is in the Defend state, and the ball is between the opponent and the player, and the player's goal like in fig… Using a shield in that position may not be most effective because the player may accidentally push the ball in his own goal.

Or, if the ball is in the attack state and the player and the opponent are to the right of the ball and the ball is between opponents' goal and them. The player could quickly grab a pick up box without risking that the opponent will score a goal because the opponent has no way of scoring a goal from that position.

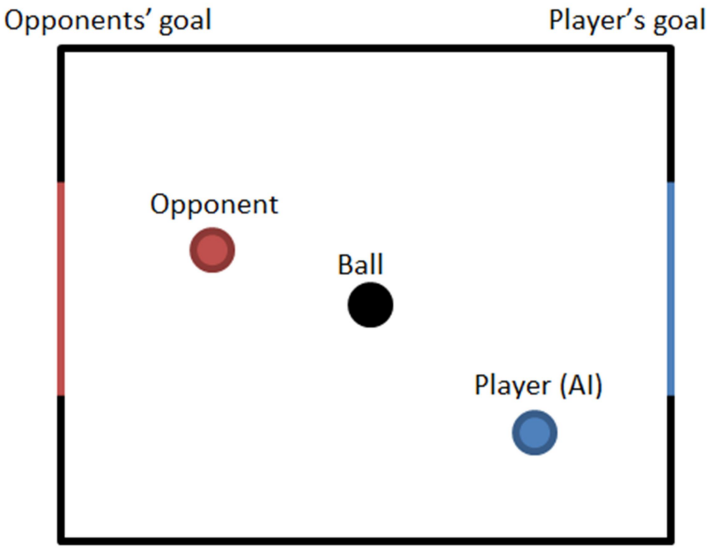
These positions are named Zero Position, One Position, Two Position and Three Position. Each of these states will prioritise a different behaviour just like the first four states did.

Position Zero



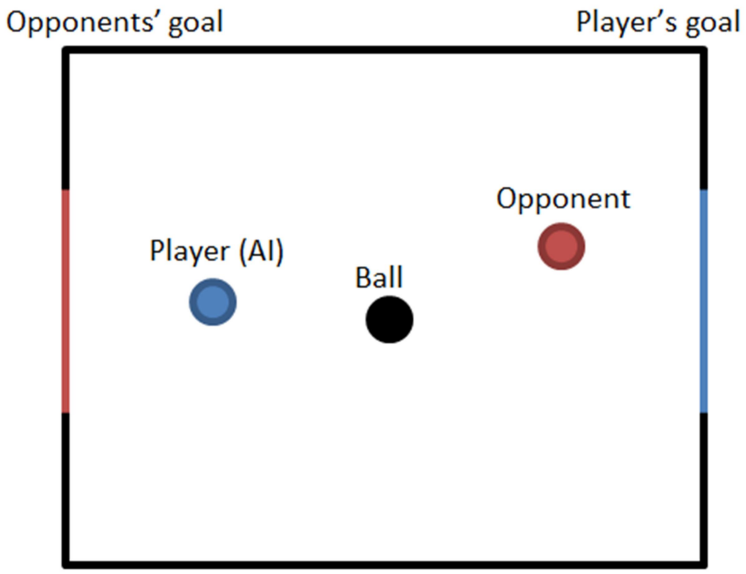
This position is advantageous for the player because there is no obstacle between the ball and the ball which makes it easy for him to score a goal. Depending on the state, the player in this position could focus on scoring a goal or, if the ball is in the defend position; he could be able to quickly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

Position One



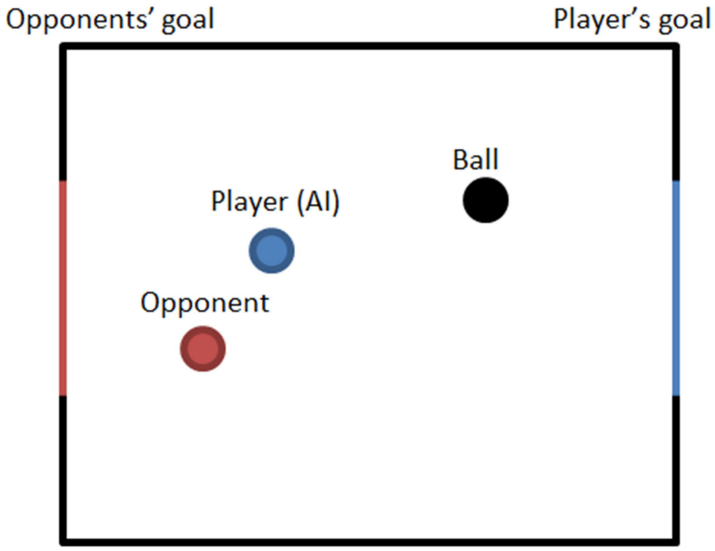
This position is neither good nor bad for the opponent and the player because the opponent and the player cannot score a goal. Usually, the most effective task to do would just be to try and turn around.

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

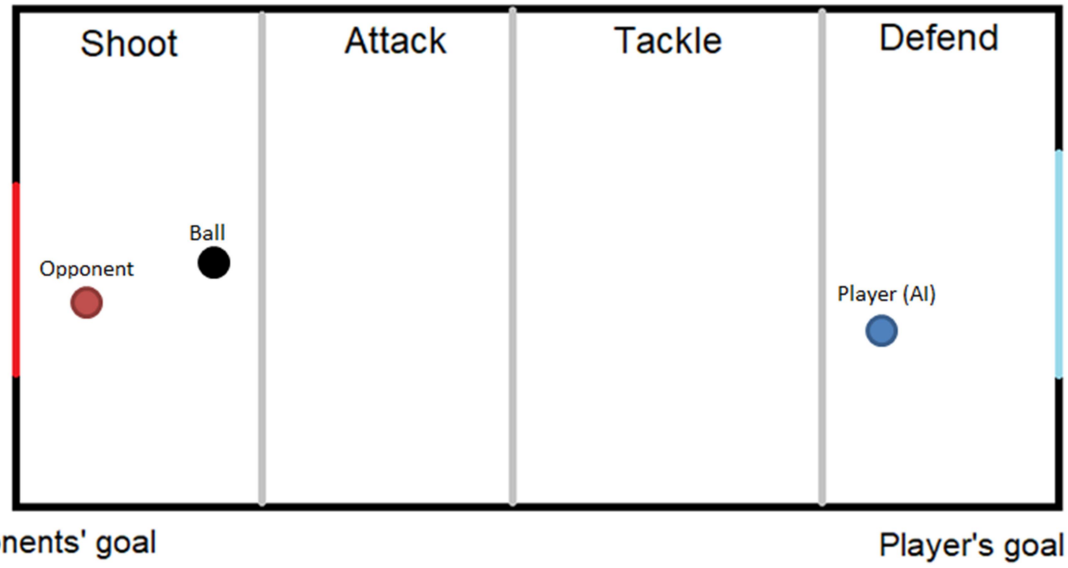


The last position is the most advantageous for the opponent because there are no obstacles between the player's goal and the ball so the opponent can easily kick it, even from a far, and score a goal. The

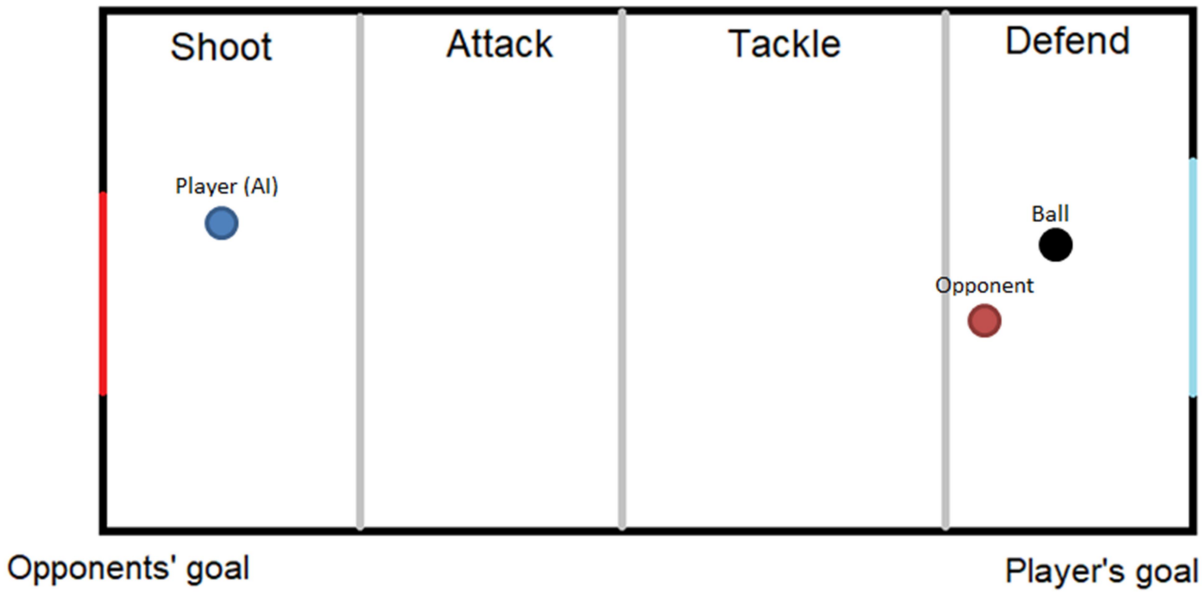
player in that position should almost always try to get close to the ball and get between it and its own goal, i.e. get into the Two Position.

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

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.



CHAPTER 4 – IMPLEMENTATION

How it works

What is prefab

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.

Walls

To 'build' the walls in the main game area, a number of Box Colliders have been attached to the Football Pitch game object and placed around the pitch. PICTURE. A default setting for a collider is that when another object collides with it, the other object will bounce back if it has a physics component called Rigidbody 2D attached to it which gives the game object control of the physics engine. This means that the game object will be affected by forces controlled from scripts and by gravity. Since this is a top down game, objects should not be affected by the gravity since there is no real up or down but they should be affected by forces. Going back to the colliders, another setting for them is to act like a trigger. This means that whenever another object collides with it, it will 'pass' through it but a message will be sent in a Collision2D format that includes all the information about the colliding object. PICTURE. The second type of collider has been added to a bullet prefab and a basic script has been added which destroys the prefab whenever it hits another object. These walls are very important because they also stop the player and the ball from moving further than the allowed game area. Circle colliders had to be added in the corners of the walls because there was a bug where a ball would sometimes stop just next to the wall which would prevent it from bouncing off it to the side, so it would get stuck and only be able to move alongside the wall. Circle colliders helped to overcome this bug in a way that whenever the ball reaches the corner it would collide with the circle which would bounce the ball away to the side. PICTURE.

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 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 is a more complex object than the wall because there is a bit more information and functionality needed to make it work. First of all, there is a basic Circle Collider2D set up just like in the walls, and the Rigidbody 2D. The most important features here are the material of the object, mass and

linear drag. Gravity can be ignored since as mentioned above, the gravity should not affect any of the objects since there is no 'real' up or down. The material of the object defines its friction and the bounciness. Since it is a ball, its friction should be very small and bounciness should be high, in this case friction is set to 0.1 (or 10%) and bounciness 1 which is a default setting. These setting allow it to behave just like a real ball. Linear drag has been set up with the mass, the higher the mass the harder it is to push the ball therefore the mass had to be kept relatively small. However the side effect of that was that the ball would travel too fast so a linear drag had to be set up, this required few hours of testing and playing around and the final outcome was 0.4 (40%) linear drag. The numbers here are very ambiguous but the most important parts are that friction, bounciness, mass and linear drag had all been used to fine-tune the way the ball behaves.

Another important part of the ball is the 'ball controller' script which keeps track of the goals and updates

Players

Players object are the most complex because there are much more information needed to be passed through it and it has more functionality that is needed to make it work. Just like the walls and the ball it has all the components that take care of the collision and movement. Since the sprites are circular, Circle Collision2D components have been used to make give the best performance possible. Complex collision detection may sometimes be processor heavy which can lead to slow performance and bugs. Rigidbody 2D has also been used but this time it was much more straightforward to set up because player has to move only when the button is pressed or when the player tilts the joystick stick in a certain direction. The player does not stop immediately when a player stops pressing a button but it still stops relatively fast compared to the ball which bounces around. If a slow slow stop time and a high mass were set up it would make the player's character feel like it is sliding on ice which is not the desired effect. The game is fast paced therefore it makes most sense to make the controls feel very responsive and easy to control. Therefore it took a lot of tests to come up with the best settings for the player movement.

What is start, update, fixed update

Values

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'. 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. 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 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. After playing other games like Grand Theft Auto 2 and Rocket League, a new way has been added that is easier for the new players. In GTA 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.

Keyboard controls

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.

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. 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.

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.

Weapons

Pickups

Wall

Behaviour

The Shoot method in the weapon game object is exactly that, the state is first checked and if it is idle, player is allowed to shoot. Which in reality takes the AbstractGun variable “currentGun” which runs a shoot method inside it.

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 which does not need to be overridden. New guns inherit this class mainly to make the code easier to understand but it could be just as easily used as the main class for the weapons. The shoot method checks if there is enough bullets, if there are not 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.

Shooter class has been added to be able to instantiate bullets in a 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’.

A reload method is very basic, it checks if player has enough bullets and adds them to the magazine accordingly. It also runs a DelayMethod(reloadDelayTime) which delays when the player is able to shoot for a given time. The reload time adds customizability to the weapons because each can have its own reload time.

Objects in Unity

How does the game loop works in Unity

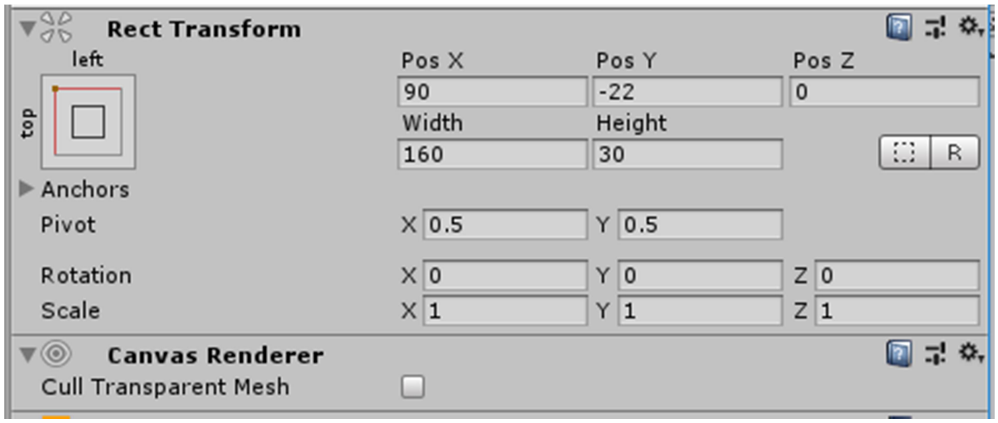
Implementing the UI

The first User Interface that has been implemented is the Head-Up Display (HUD). 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;
```

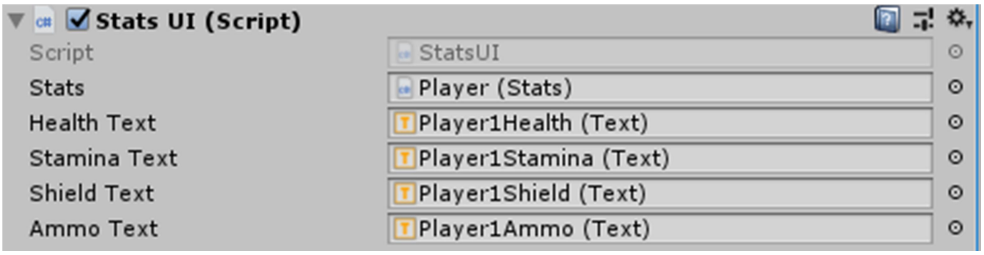
- extracts the scores

```
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 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 which 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 Scene

The game settings 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

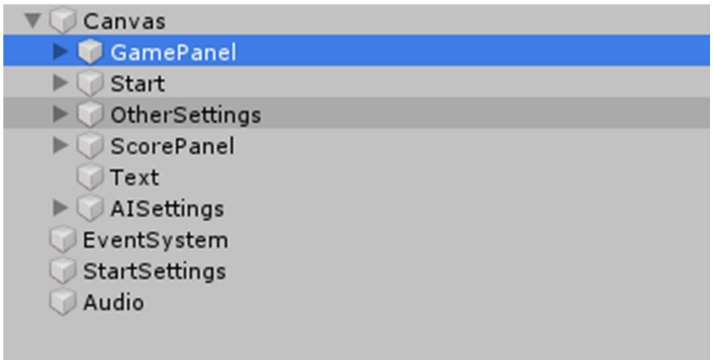
button, Unity has a very straightforward way to pass a value to the method it is calling. AddHealth method is called with 1 passed to it, in the Add button and the same method is called to subtract from that variable but -1 is passed to it. Fig. shows the AddHealth method that is called when the Add button is pressed.

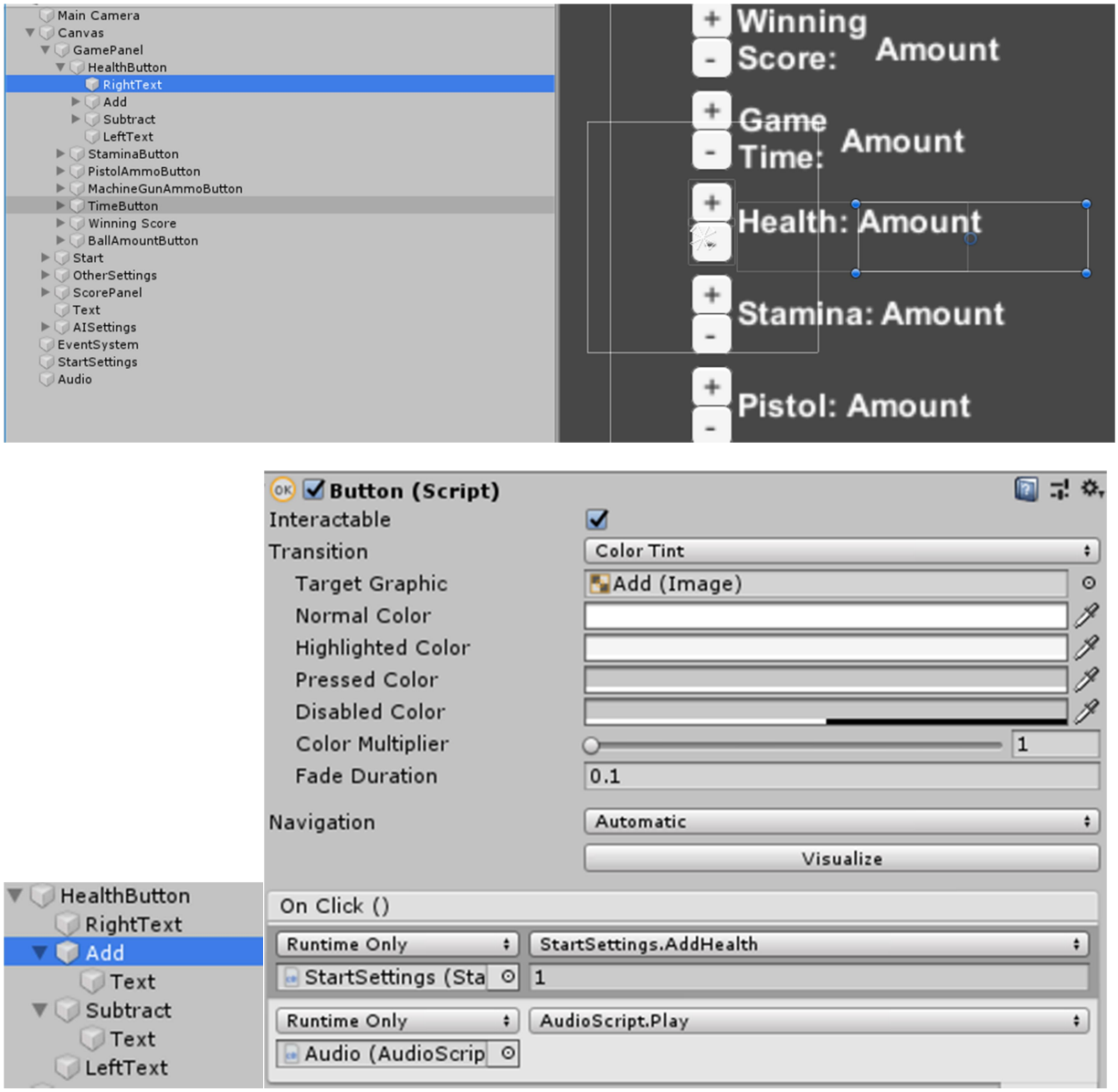
```
0 references
56  public void AddHealth(int health)
57  {
58      GameStartSettings.health += health;
59      if (GameStartSettings.health < 10) {GameStartSettings.health = 10;}
60  }
```

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

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.

Implementation

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.

Artificial Intelligence Implementation

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 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 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
public static void MoveTowards(GameObject player, GameObject destination)
{
    float x = destination.transform.position.x;
    float y = destination.transform.position.y;
    MoveTowards(player, x, y);
}
```

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.

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

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  }
```

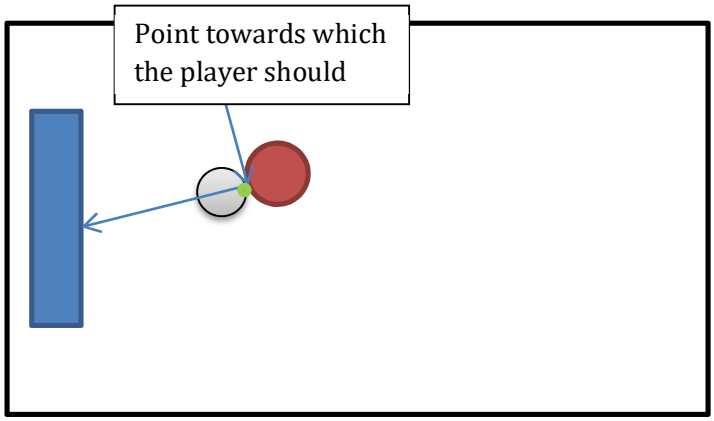
Figure. Shows the GetDirectionVector method which converts a radian to a directional vector. A more basic version of this method is used inside the MoveTowards method, since a radian is first found by using the GetRadian method, then the GetDirectionVector is found separately. Fig. shows that method.

```
    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.

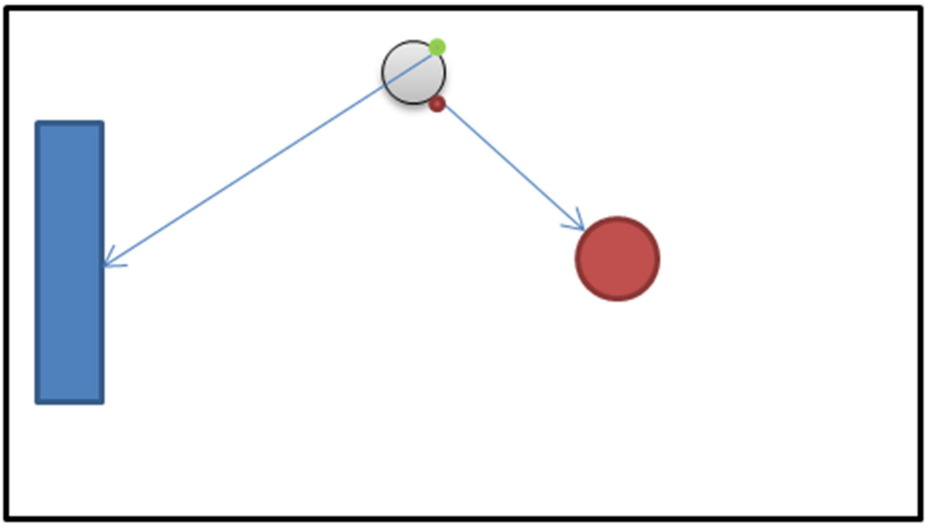
This position is found by using the three methods explained above; fig. shows the first version of this code which returns it as a Vector2.

```
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:

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

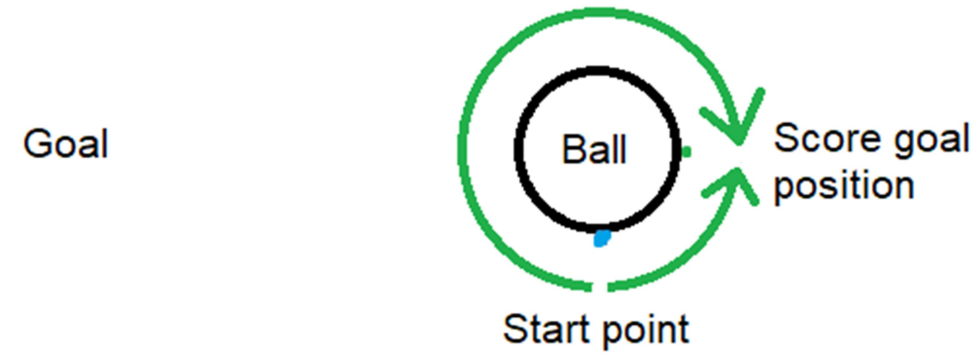
Since only one counter is needed and the class is static, it was easier to have the counter be a component of the player game object, as can be seen in fig. The class is very simple as its only job is to cycle from 0 to 1 in a steady step. Fig. shows that class. The value added in line 14 determines the speed, this has been carefully chosen to match the player speed.

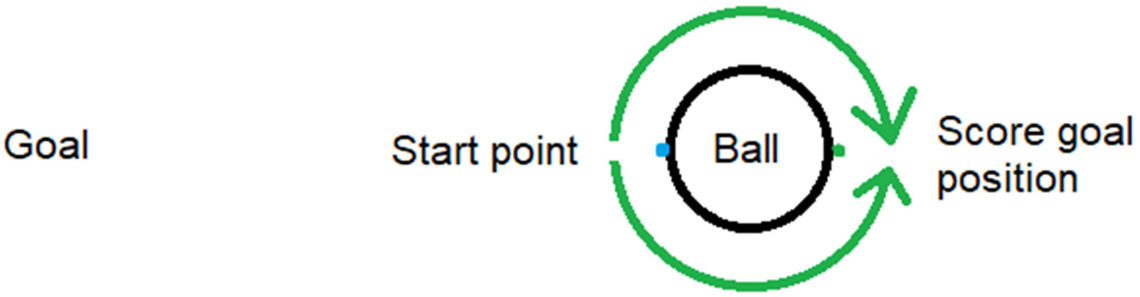
```
5   public class Counter : MonoBehaviour
6   {
7       public float a;
        0 references
8       void Start()
9       {
10          a = 0;
11      }
        0 references
12      void Update()
13      {
14          a += 0.01f;
15          if (a >= 1)
16          {
17              a = 0;
18    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

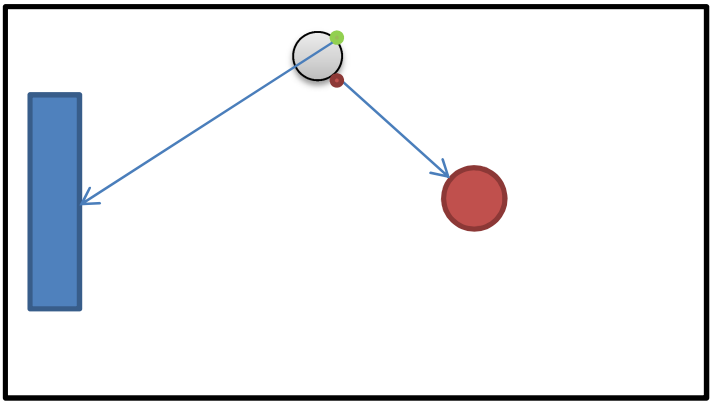
```
27  public static void PositionAndShoot(GameObject player, GameObject ball, GameObject goal)
28  {
29      /* First a position of the goal is found, then player is rotated to face the ball.
30      * Next distance from the player to the point from which he would be able to shoot is found.
31      * 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 logic



How it chooses the behaviours

AI movement design

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.

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

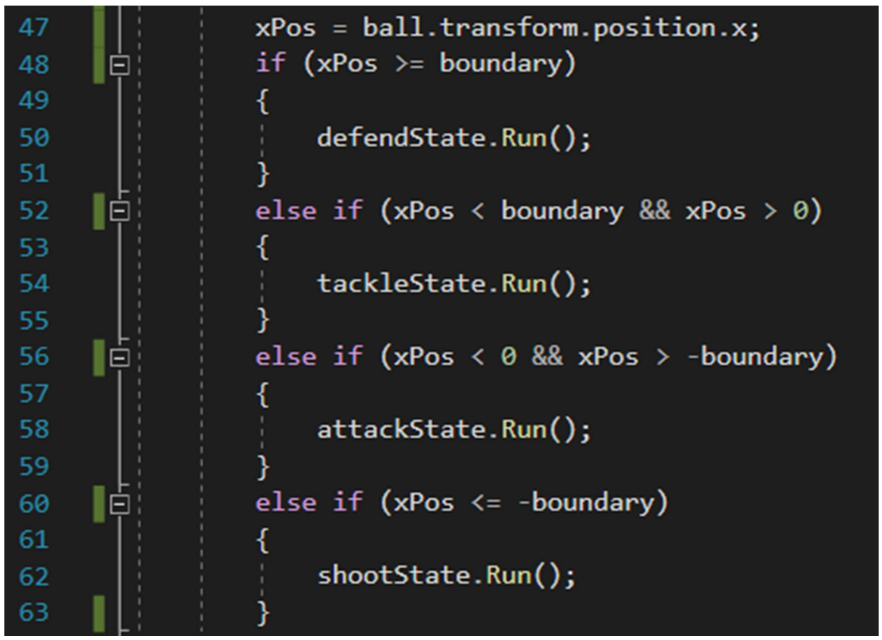
Name	Task
CheckIfCloseToPickup	It goes through each pickup inside a passed list, calculates the distance between it and returns a bool value based on the results.
GetClosestPickup	This method checks if there are any active pickups in the game, checks which one is the closest and returns its position 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.

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.

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, creatively 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

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      }
```

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

Describe how the implementation maps onto the design you have already discussed.

You should use “code snippets” to illustrate special features of your work or difficult (awkward) bits of coding. Don’t make any of these snippets longer than half a page (and include line numbers if possible). If the code fragment is longer than half a page then break it up into smaller bits.

Describe the code, both in terms of the overall architecture and in terms of the snippets. Make sure the reader understands what you have done and why!

CHAPTER 5 – EXPERIMENTATION AND EVALUATION

I have most of the answers already so here I will analyse them and evaluate if my project was successful, the questionnaire will be kept in the appendix

Questionnaire

Answer

1. In one or two sentences, how was the overall game experience?

2. In one or two sentences, how did the game make you feel?

3. i) How intuitive did you find the User Interface (menu)? Rate from 1 (very intuitive) to 5 (very confusing)

3. ii) What was it about it that made you feel this way?

4. How challenging was the game against the CPU (computer opponent)? Rate from 1 (very easy) to 5 (very hard)

5. i) How did you feel playing against the CPU? Rate from 1 (felt like a real human being) to 5 (felt like a scripted opponent)

5. ii) What was it about it that made you feel this way?

6. What did you enjoy the most about the game?

7. What did you enjoy the least about the game?

If you haven't done too much testing (for instance it is GUI based) then include a "walk-through" of the application with screenshots showing the scenarios in which the application can be used. After all, the examiners may not be near a computer to actually "run" your code.

You also need to clearly show how you have evaluated your work and show how it meets the original aims and objectives.

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.

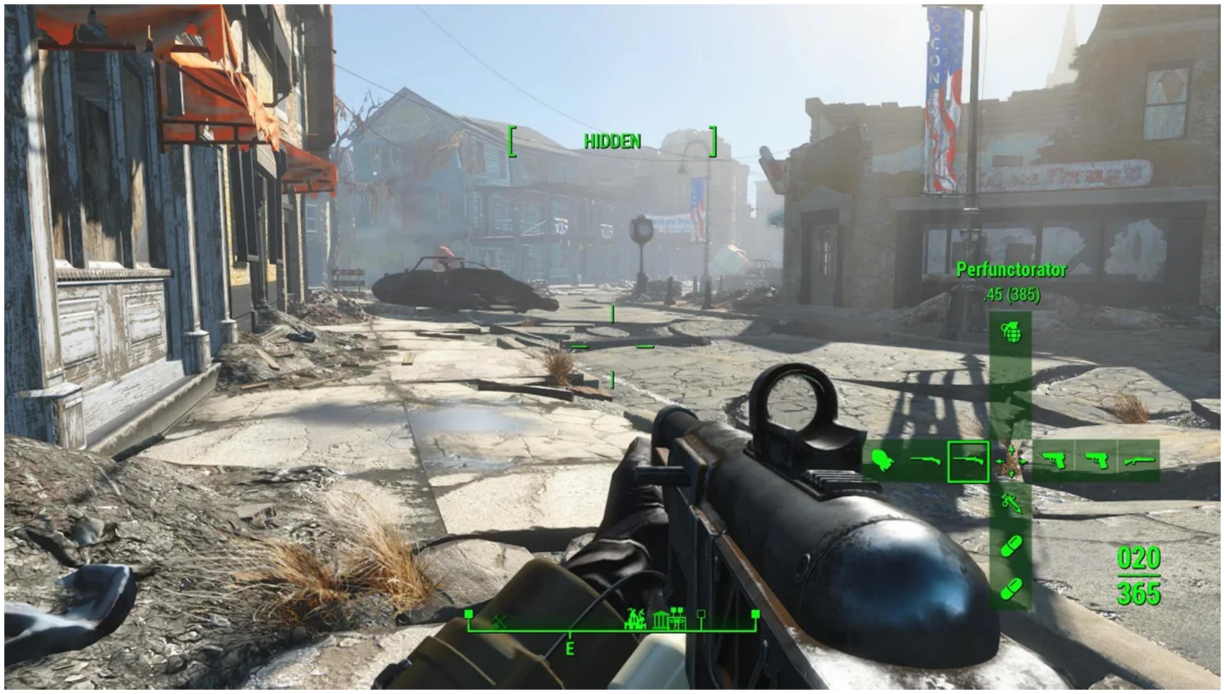
APPENDICES

Include design diagrams, data formats, etc.

Basically, don't overfill the report.

Link to the GitHub repository

Game UI




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


$00048624
16:45
193 13
POLICE

REFERENCES