First-Person Shooter Video Game with Reinforcement-Learning Trained Game AI

Luke Tang

MSc Computer Science

Project Report

Department of Computer Science and Information Systems,

Birkbeck College, University of London

2021

Abstract

Video game NPCs are almost exclusively programmed with conventional techniques that tailor's them to a specific game and purpose. But this often leads to NPCs that have limited behaviours, which have to be tuned significantly for a specific purpose, especially in First-Person Shooter games, where NPCs meant to simulate human players can seem very unintelligent. This project seeks to find a reinforcement-learning-based solution to train video game NPCs. It will do this be utilising a variety of software packages to create a learning environment where an agent is able to learn to act is a satisfactory way. Rather than going through a conventional route, where AI behaviour is heavily controlled to determine its reaction to different stimuli, the agent in this project begins with no guidance, and learns through variables it can observe.

In training an agent, the project has produced models of AI behaviour that demonstrate some feasibility in using reinforcement learning to train NPCs, these agents have learned to go to a player's location, and to an extent, chase them. The process of training has led me to conclude that overall, reinforcement learning is likely not very useful to game developers for creating most NPCs.

1.1 Introduction

The very earliest First Person Shooter (FPS) video games, such as Wolfenstein 3D, included non-player characters (NPCs) who acted as enemies for the player to eliminate. Even with the comparatively underpowered hardware that was available at the time, games rendered 2D sprites to act as these enemies, while ray casting [1] techniques allowed them to project these sprites onto the mimicked 3D world. Quake (1996) was an advancement for NPCs; no longer were they 2D sprites but now they were 3D models. These early games were genre defining in many features that are undeniably core to the modern FPS video game experience. As well as health, movement systems, and guns, players also expect AI controlled NPCs as part of their gameplay experience [2].

In many ways, FPS NPCs are very similar to the player. They have possess many of the same gameplay features; in 3D games they often move about in the same way as the player, and have similar health and damage systems. This is the same in their actions, with enemies doing their best to avoid damage, while trying to damage their own enemies as much as possible. AI behaviour has seen great improvements since the early FPS games listed above. In Wolfenstein 3D, enemy behaviour was extremely limited: They fired their weapons at the direction of the player, and had set strafing motion coded to challenge the player's aim slightly [2].

Modern FPS NPCs are (mostly) more complex, which complements the greater depth of the game they appear in. These NPCs may make use of cover systems, or coordinate with other NPCs to more effectively fight the player. AI design as therefore become a careful art, to ensure it results in the most enjoyable gameplay. Most AIs have variables that can be tuned to change gameplay, and players have come to expect competent computer-controlled teammates and enemies.

An example of modern FPS NPCs are the bots in 2016's Counter-Strike Global Offensive. To many Counter-Strike players, bots are completely irrational; they wander around aimlessly, and seem to always seem to find themselves in the worst positions. Indeed, bots require a human-made navigation mesh which tells them where and where they can't walk, and what to do at certain parts of the map. Without these, they would be incapable of even climbing ramps or ducking through tunnels [3].

There are few examples of AIs in video games that have their behaviour guided by Reinforcement Learning (RL). Currently, RL in video games is limited to experimentation and research, and not to produce commercially viable NPC systems [4].

The intention of this project is to therefore create a first-person shooter game where the player will play against NPCs trained by reinforcement learning, within an arena-like environment.

2. Background

2.1 Reinforcement Learning

Reinforcement Learning is a branch of machine learning. What sets it apart from other fields within machine learning if how it interacts with its environment, and how it learns. An RL agent is constantly seeking a reward, and changes its action depending on the reward it receives [5, p.2]. Other fields of machine learning may have a target or a goal, but an RL agent starts off knowing nothing, and knows nothing about what it should seek. It learns via the observations it makes combined with its observations. Its actions may result in a reward, or a punishment. The agent will seek to take actions that maximise the reward it receives, while minimising the punishment. Successive actions shape how the agent thinks, and with enough training it learns to associate certain observations with either Reward or punishment.

2.2 Reinforcement learning concepts

There are some key terms within RL that describe components of the agents' actions and learning. The first is the reward. The reward is what the agent wants to maximise. As training continues, the agent works out the best way to achieve this.

Second is the value function. The value function works alongside the reward – it represents how much an agent could receive in the long term in a particular state. Agents will associate different states with different value functions [5].

The Environment is the space that the agent acts within and observes. When the agent acts, it observes any changes in its environment, and then adjusts its actions based on this and the reward signal [6].

Finally, the policy acts as manual, or rulebook for an agent. The policy dictates what action an agent should take in a particular state, and the policy gets updated to reflect changes in observation, and reward signal received [5, p.6].

2.3 Reinforcement learning in classic games

Many AIs have become incredibly skilful at classic games like chess. [7] These are popular testing grounds for RL agents as they represent a simplified view of the real world, with a smaller environment and fewer possible states. Many AIs have reached superhuman or master levels, and some have even completed games like checkers, that is they are impossible to beat. [7]

AlphaGo, developed by DeepMind is an example of an RL trained agent that managed to beat one of the best ever Go players in 2016. [8]

2.4 Reinforcement learning in video games

As previously mentioned, RL usage in modern video games is uncommon, however some organisations have attempted and have been successful at producing incredibly skilled agents, capable of playing and winning against the highest skilled players in two games in particular.

2.4.1 OpenAi Five

The first of these is OpenAI Five, an RL program that plays the popular MOBA (Multiplayer Online Battle Arena) game *Dota 2*. As a quick description, *Dota 2 is* a complicated player vs player game where there are two teams of people players. Each player picks a unique character out of a pool of 120, and then works with their team to destroy the enemy base[10].

OpenAI Five played the current world champions, OG Esports, in a showmatch in April 2019, where they won, demonstrating the power of the agent. [9]

OpenAI Five's success is significant – *Dota 2* is a confusing game even for veteran gamers, with many saying that at least 100 hours of gameplay is needed to just understand what to do and how the game's complex systems work [11]. Open AI had some benefits. It was able to train on incredibly powerful computation, enabling it play for a total of 40,000 years [9]. This enabled it to train on the huge number of variables about the environmental state it had access too.

2.4.2 AlphaStar

Another Successful RL agent is AlphaStar, developed by the same team that created the AlphaGo [12]. AlphaStar learned to play StarCraft II, a game notorious for demanding excellent multitasking and micromanagement skills from its players.

Like OpenAI Five, AlphaStar was put to the test in a showmatch against a professional player in 2019, where it won. AlphaStar then proceeded that to enter and play in an online StarCraft II league, where its model was adjusted to take away some advantages it had previously. This included limiting its actions over a five-second period, and how much of the map it could see at once. [12]

2.4.4 discussion on current video game RL implementation

Clearly, there is a common theme between AlphaStar and OpenAI Five. With large computational resources to train the agents, and teams of incredibly smart RL experts, it is certainly true that these two implementations took a significant amount of effort, intellectually, financially, and computationally to develop. [4, 9, 12]

This leads me to the idea that RL isn't really feasible as a tool the majority of game developers to develop AI behaviour. Even if the goal is to build sophisticated Human-mimicking AI for PvP use, it is likely that the majority of game development studios do not have the resources to afford the development cost. OpenAI is funded by Billionaire Elon Musk[13], and DeepMind is a Google subsidiary [14], and thus have access to the resources needed to further their development.

This leads to the next point. Neither of these two organisations are developing with commercial interests in mind [4]. They are not creating AI that soundly beat professionals in order to make money. Rather, these teams are using the experience of working with modern video games to further research within the field of reinforcement learning.[4, 9, 14]

2.5 The Reinforcement Learning Algorithm

One RL algorithm implemented in my project is called Qlearning. Qlearning is an RL algorithm where the "Q" stands for *quality*. Qlearning works as follows [15]:
Imagine a table, in this table the columns represent possible actions, while the rows represent possible states. Combined together, this means that each item in the table represents an action and a state [15]. This table is called the Qtable, and it is what the agent uses to decide its actions.

2.5.1 The Qlearning process

The five stages of the algorithm are, as explained by [15, 16]:

1. Initialise the Qtable.
2. Choose an action.

3. Perform the action.
4. Measure the reward.
5. Update the Qtable. *Then, repeat from step 2.*

As this shows, the Q table stores a value at each table item. After many iterations, the Qtable will begin to display accurate values for most states [16].

The agent knows nothing at when training begins, so all values within the Qtable are initialised to 0. Because it doesn't yet know what actions are good, the agent will explore, meaning it will take actions that it does not know will result in the greatest reward. The opposite of exploring is exploiting, which means it acts to maximise the reward [16]. Since all Qvalues are 0, the agent is not capable of exploiting. The *Epsilon greedy value* [15] defines this behaviour: as we know less, we will explore more and vice versa (we will switch to exploit more andexplore less) [7].

Therefore, the agent comes across new states by iteration [15]. The reward could be positive, negative, or neither. For example, If the agent is at a certain state, takes a step the right, and observes it receives a reward of +50, and taking a step left from the same spot results in a reward of 0, then it can update the Qtable at this state to increase the value of moving right.

2.5.2 The Qfunction

The Qfunction is the algorithm utilised in step 5. Of the Qlearning process to update the Qvalue, which is what decides what action to take given the current state.
It takes the current state and a possible action as inputs to return a *Qvalue*, which is the reward an agent can expect to receive if it takes that action in its current state [15]. This new Qvalue subsequently replaces the old Qvalue that was in the same field in the Qtable.
The formula is:

$$NewQ(s, a) = Q(s, a) + \alpha[R(s, a) + \gamma maxQ'(s', a') - Q(s, a)]$$

Defined in [5, p. 131, 15, 16]

Each component of the table is;

- *NewQ(s, a)* is the new Qvalue that will replace the old one in the table.
- *Q(s, a)* is the current Qvalue present in the Qtable.
- α is the learning rate.
- *R(s, a)* is the reward for taking that particular action at that state.
- γ is the discount rate.
- *maxQ'(s', a')* is the maximum expected future reward given the new state and all possible actions from there.

3. Methodology

3.1 Specification/requirements

The goal of this project is to investigate and demonstrate the feasibility of creating a self-learning reinforcement learning agent within the context of a first-person shooter video game. Emphasis is placed upon creating a working and learning agent, as opposed to fleshed-out first person shooter game mechanics, such as health systems, or impressive audio-visual effects. These features, which certainly add to the enjoyment of any game, are secondary to the AI component of the project.

Originally, I had intended for the NPC to be trained to have more advanced behaviours. These included:

- Turning to face the player, and then shooting a weapon of its own, with the intent to damage them.
- Actively trying to dodge player shots, and utilising line of sight and cover.
- Utilising verticality: the game takes place in 3D space, so the agent running up ramps and ledges to get above or below the player would have been possible.
- Training agents against each other in a multi-agent learning environment. The player would replace one of the agents during gameplay sessions.

However, as I learned more about reinforcement learning, it became apparent that getting an agent to learn all of these more complex behaviours would not only be difficult to implement, but also likely longer than the timeframe of the project would allow. This is in addition to the increased time that would be required to train an agent to act effectively with all these additional behaviours.

With this in mind, I decided to limit the scope of the project, and get the AI to learn simpler behaviour.

3.1.1 Objectives

With the overall goal of the project mentioned, we can now define a small number of specific objectives.

1. Learning from nothing. A single agent should be able to demonstrate that it has at least come some way to learning some behaviours that result in recognizable NPC behaviour.
2. If the previous goal is successful, then the agent should learn to chase the player. The AI could still be useful – It could act, say as the AI for zombies that mindlessly chase the player.

3.1.2 Users and uses

Although I intended to create a video game AI with this project, it is in no way close to commercially released video game in terms of scale, complexity, or polish. The programme consists entirely of a barebones arena, a chasing NPC, and the player character.

Users of the programme are therefore unlikely to play the game for the purposes of entertainment like commercially released video games. People likely to play this game are those interested in any of the following; game development, reinforcement learning, FPS games, or most likely, reinforcement learning within video games.

Uses of the game include acting as a proof of concept that RL is feasible in FPS games, and some may find the game mildly amusing to play for a small amount of time.

4 Tools and technologies

4.1.1 Game Engine

The overall platform that I chose to build the project with was Unreal Engine 4, developed by Epic Games [17]. Unreal engine is a sophisticated and highly versatile software kit that can be used to create many different types of 3D content. Most commonly, it is used as a platform to create video games, as it has many tools that simplify and aide in the creation of interactive 3D applications. Most notably, this includes Blueprints, which is the Unreal Engine's proprietary visual coding/scripting system, which I will describe in further detail in section 5.2.2 [18].

Other reasons I chose unreal Engine were for its ease in manipulating animation, lighting, environment building, and textures. Additionally, Unreal Engine comes with a variety of free assets which I have incorporated into the project. for instance, I have used the default Unreal Engine mannequin as the agent in-game model. The engine also has an excellent marketplace, which I have used to obtain a free set of new animations for the agent model.

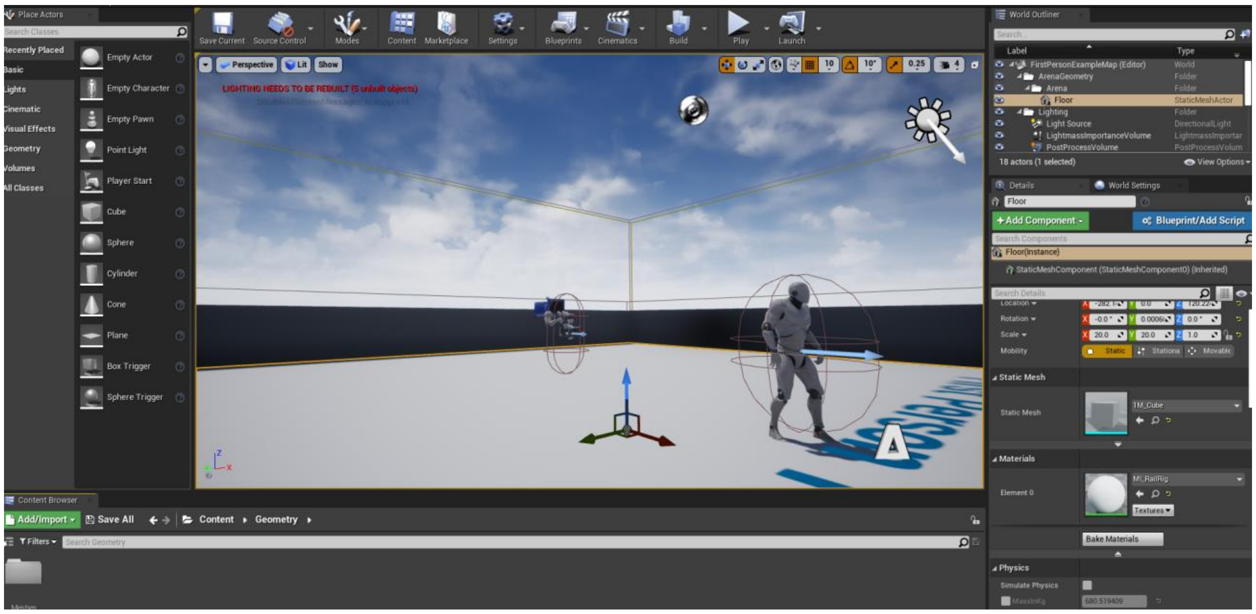


Figure 1: The Unreal Engine user interface.

I had considered other game engines, such as Unity, however I settled upon Unreal Engine because of the blueprint function which I found made building the projects incredibly simple and intuitive.

4.1.2 Programming Languages

Unreal Engine applications are coded in C++, so I had expected to learn the language in order to fulfil the project, alongside the Blueprint visual scripting system. However, I found that the Blueprint system was incredibly intuitive and simple to learn, and that all coding could be accomplished through Blueprints; all functions and methods are implemented in Blueprints in a simple, sensible manner. Thus, to increase the coherence of the project, I decided to complete all coding and scripting through blueprints.

Different blocks of code are represented by physical nodes, and the links between them are represented by wires [18].

The following demonstrates how the blueprints work:

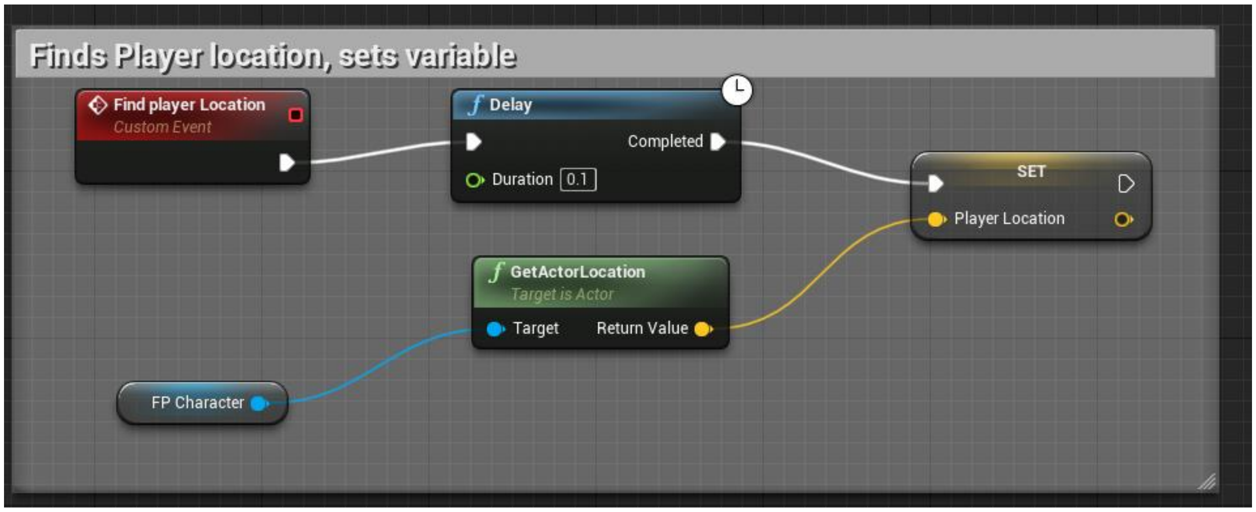


Figure 2: A line of execution within the agent model's blueprint

In the above screenshot, the following happens:

-

When this event is fired:

Wait 0.1 seconds

Get the location of Actor: FP Character (the player) -> returns vector

Set the variable “Player Location” to this variable.

-

After having familiarised myself with the Unreal Engine documentation, I found blueprints to be an incredibly useful and simple abstraction for C++ coding.

4.2 Reinforcement Learning Implementation

I implemented Reinforcement Learning into Unreal Engine through the use of MindMaker [19], which is a plugin and deep reinforcement learning package developed by Aaron Krumins, specifically for use with Unreal Engine. The Plugin is free, while the deep reinforcement learning package (DRL) costs $4.99 on the Unreal Marketplace.

MindMaker acts as a platform to increase the ease of which a developer can implement RL into their Unreal project. It provides the frameworks for a variety of important RL functions, and was essential for getting this project to work, as it allows for efficient integration of RL tools within Unreal Engine.

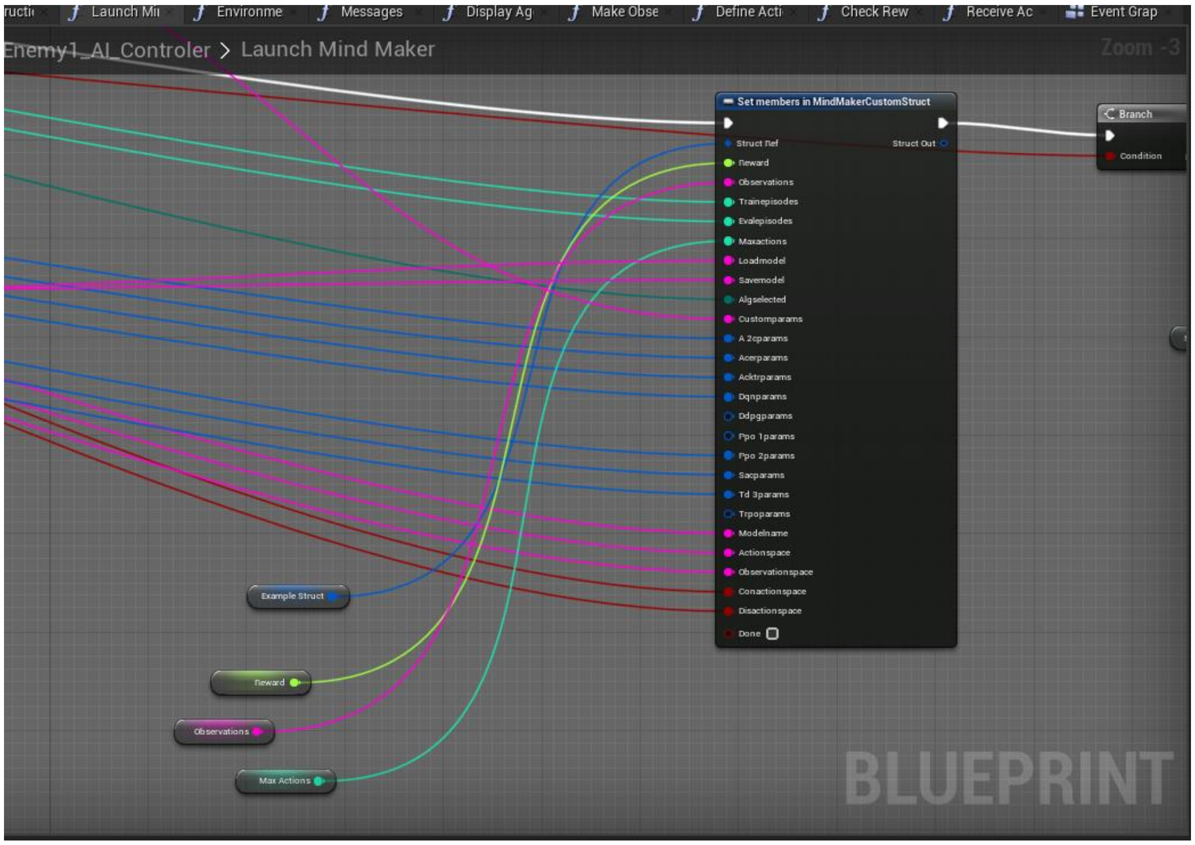


Figure 3: One of the background MindMaker functions shown; this function sets the members of the struct that stores variables used in the package.

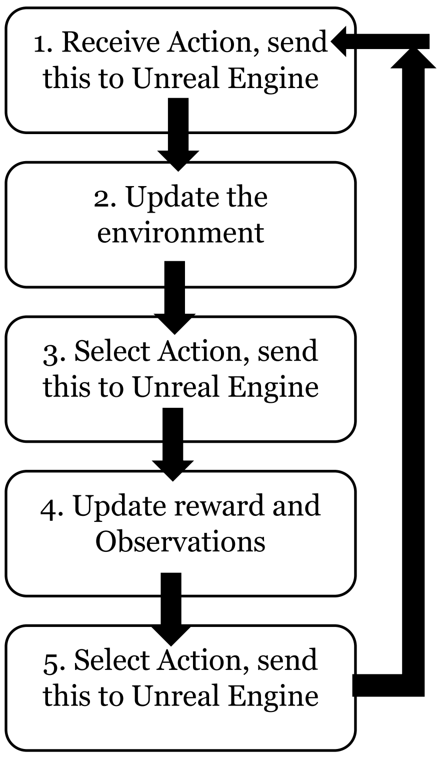
MindMaker has been designed to work within Blueprints, making the package even easier to become familiar with. It also has documentation available, written by Aaron, that allows for people new to RL to start working with the package quickly [20].

Finally, a feature of Mindmaker is that it allows users to select different RL algorithms to suit their needs, this is useful for the project as I found that testing different algorithms helped test and produce an agent [20].

5. Implementation

To increase clarity in this section, I will include screenshots of the relevant blueprints, and follow them with plain English text that describes precisely what is achieved in the blueprint.

A flow chart of how MindMaker selects actions is as follows, according to documentation provide by Krumins [20] This follows a standard learning process:


1. Receive Action, send this to Unreal Engine
2. Update the environment
3. Select Action, send this to Unreal Engine
4. Update reward and Observations
5. Select Action, send this to Unreal Engine

I therefore decided to implement each step of functionality in this order. I will first describe how MindMaker works to facilitate learning work through how I set up the environment, and then describe the steps taken to implement each step of the RL learning algorithm described in 3.1.1

5.1 How MindMaker Functions

At the start of play, MindMaker is launched, and it binds some functions contained within Unreal Engine to the MindMaker application that works outside of Unreal Engine, such as the Send Observations function (figure 6). It then starts looping through the flow of execution in the flow chart above [20].

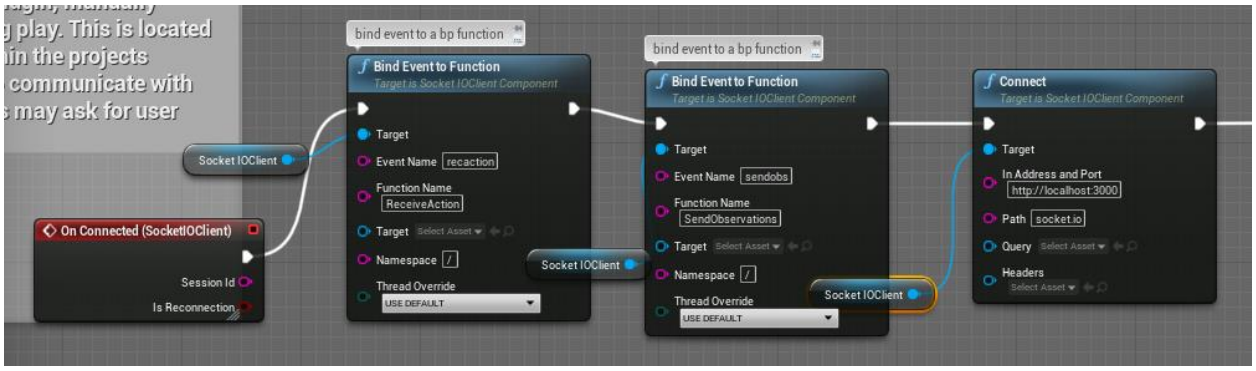

bind event to a bp function
Bind Event to Function
Target is Socket IOClient Component
Target
Event Name recaction
Function Name ReceiveAction
Namespace /
Thread Override USE DEFAULT
Socket IOClient
On Connected (SocketIOClient)
Session Id
Is Reconnection
Event Name sendobs
Function Name SendObservations
Connect
In Address and Port http://localhost:3000
Path socket.io
Query
Headers

Figure 4: Binding Unreal engine Functions

Receive Action fires next – this function receives the action from MindMaker and displays it in the environment. It was also contains the logic for checking if training is over, and also calls the rest of the functions along the flowchart [20].

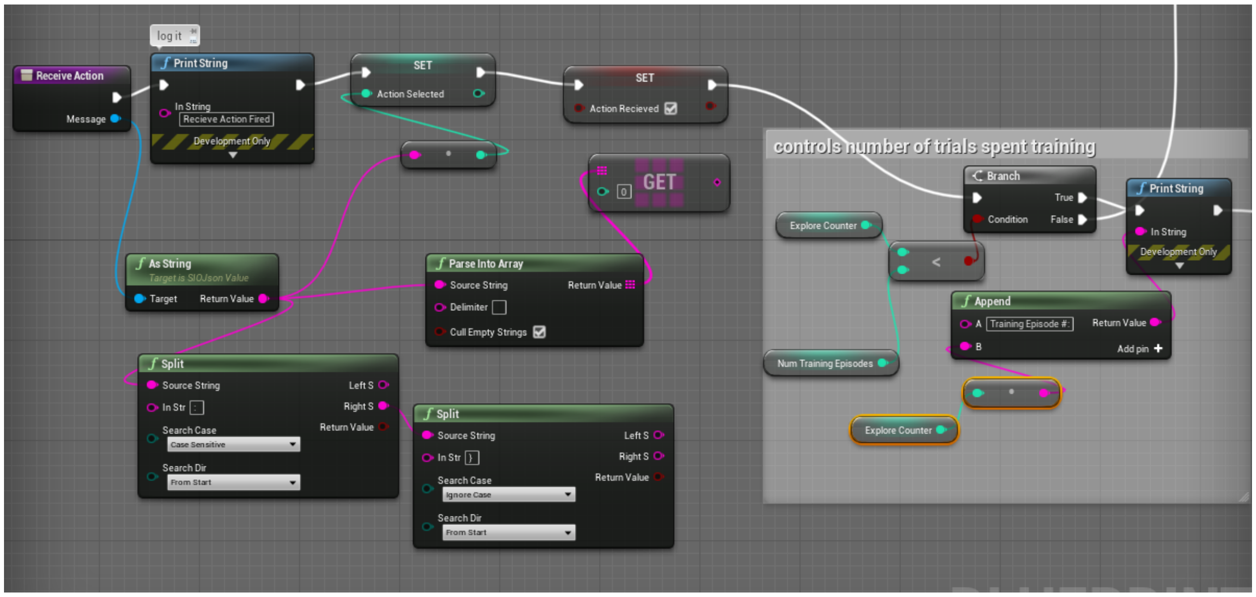


Figure 4: receiving the action from MindMaker, and then checking what training episode it is.

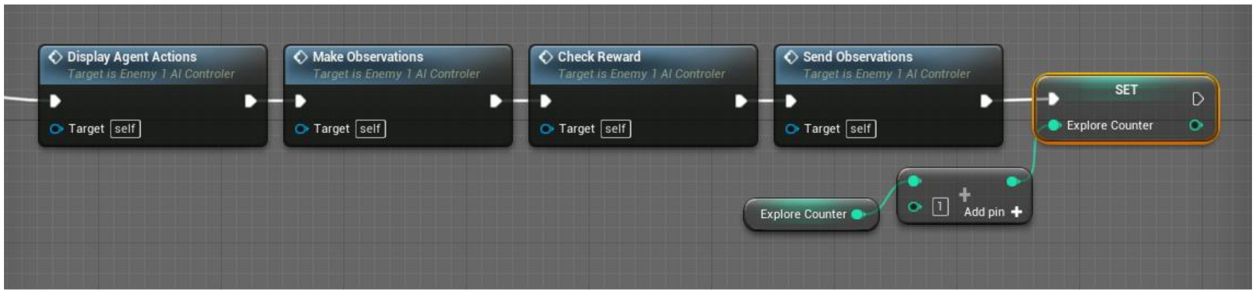


Figure 5: Receive action calling the rest of the flowchart

Display Agent Actions, Make Obervations, and Check Reward will all be examined in depth in the rest of this section. The 'Send Observations' function communicates with the MindMaker application, which takes the observations and adjusts the neural network according to the values it receives.

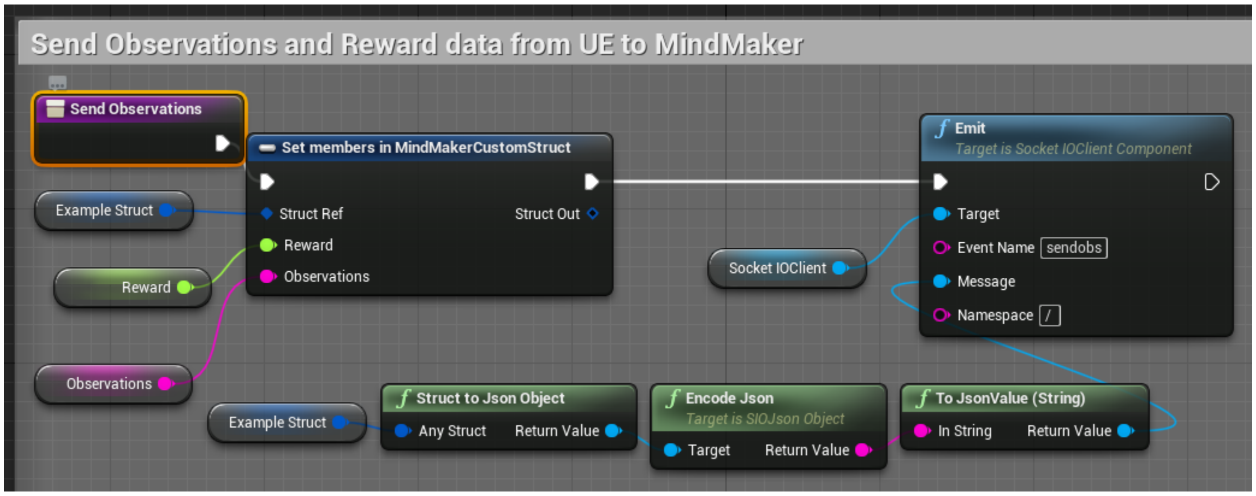


Figure 6: The Send Observations function

This function notes the reward and observations, and then sends them to the application. When MindMaker Application has adjusted its neural network, it sends another action to Unreal engine and the Receive Action function is called again, starting another learning iteration [20].

5.2 The environment

The first area of implementation is setting up the environment. The environment is the setting in which the agent learns behaviours. All possible states that the agent and any other entities could be in are contained within the environment [6]. For instance, the current state from section 3.1.3 can be thought of as entirely contained within the environment.

The environment that I created for this project was intentionally very simple. For the physical arena, I used unreal engine's FirstPersonShooter template [21]. This was then adapted significantly. I decreased the size of the arena in both the X&Y directions. Additionally, I removed cubes and blocks that serve to act as obstacles for agents and players. With these actions, training the agent Should be faster.

The arena is approximately 1900 x 1900 units (one unreal unit is equivalent to 1 centimetre).

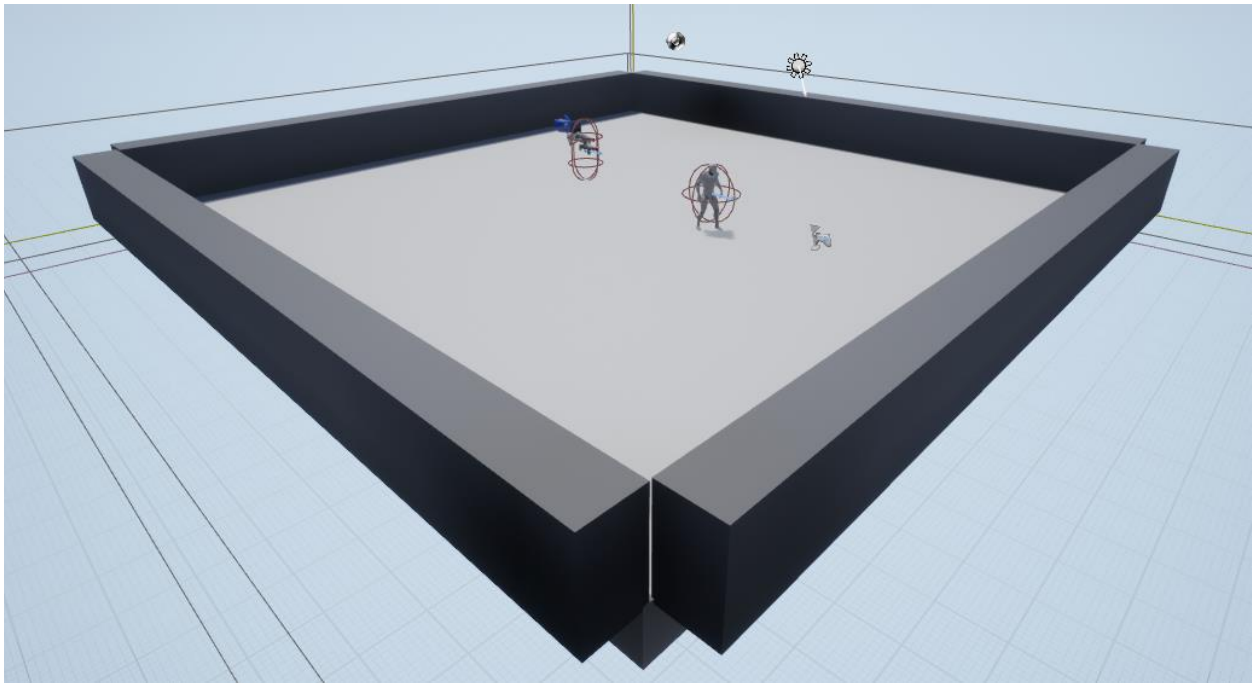


Figure 7: The training environment and arena

In the picture above, the agent is the humanoid figure near the centre of the image. On the image left, and behind the figure, is the player character. They are represented by a pair of arms and a camera component. this is where the player character starts and is facing when training/play begins.

Setup of the player character was done by Unreal Engine as part of the FirstPersonShooter Template, and I did not alter these – it comes with camera mouse movement and keyboard WASD key binds setup [21]. Since I was primarily concerned with programming the AI, I was satisfied with the basic controls supplied by the game engine. The Blueprints for these are shown in the appendices.

Unless mentioned, all functions shown are located in the 'Enemy1_AI_Controler' blueprint, which is an extension of a blueprint provided by MindMaker.

5.3 Action space

The action space refers to the possible actions that the agent can take. Initially, I attempted to take an approach where if you action space would be a value between 0 and 359. This would correspond to its rotation, and when the agent selected in action it would rotate to face that angle. After experimentation and helpful email correspondence with the creator of mind maker, Aaron Krumins, I decided that this approach was much too complex to train an agent in the amount of time I had.

I decided to drastically decrease the number of possible actions, going from the 360 previously, to just 4. These corresponded to the agent facing north and then moving forwards a short amount, facing east and moving forwards, facing west and moving forwards, or facing South and moving forwards.

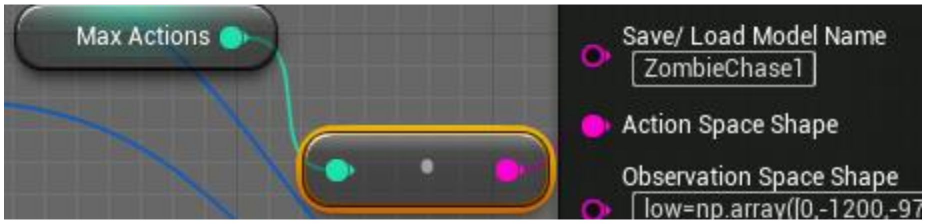
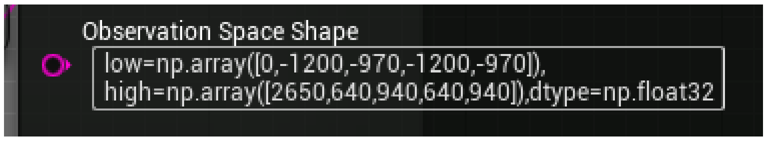


Figure 8: Setting action space. The node between 'Max Actions' and 'Action Space shape' converts the integer to a string.

The image above shows the MindMaker setting of the action space. In this case, it was set to whatever variable 'Max Actions' was, which was 4.

5.4 Observation space.

The observation space entailed setting up all the information that the agent has access to.



Observation Space Shape

```
low=np.array([0,-1200,-970,-1200,-970]),
high=np.array([2650,640,940,640,940]),dtype=np.float32
```

Figure 9: The Observation space shape in MindMaker.

Defining the observation space shape tells the agent what values to expect when they make observations [6]. Each index of the low and high array corresponds to an index in the observations array that is sent to MindMaker:

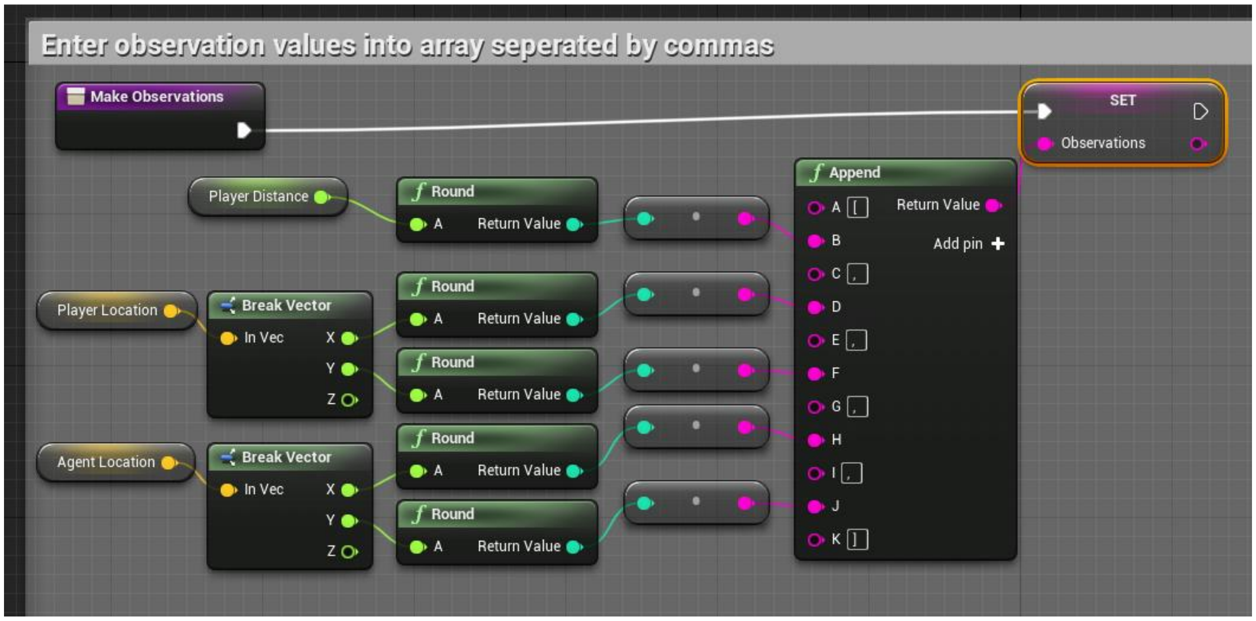


Figure 10. From the 'Make Observations' Function.

The observation space consists of 3 variables which make up the five elements of the observation array (which is sent to MindMaker as a string). The first variable is player distance, which is a float value rounded to the nearest integer, converted to a string. it has a low value of 0 and a high value of 2650. This distance is the maximum possible distance that the player can be from the agent (i.e. the player and agent in opposite corners of the environment). The next two values represent the X&Y values of the players location. Low values of -1200 and -970 represent the location of the bottom left corner of the arena. High values of 640 and 940 represent the x and y values at the top right corner of the arena. These x and y values are also rounded to integers.

As the arena is in 3D space, location values are represented by a vector variable, which holds information on x, y, and z values. A node is therefore used to break the vector into its X&Y components before it is appended into the string.

These are the observations that are sent to MindMaker, at step 4 of the algorithm described in section 2.5.1

5.5 Selecting action

At step 1. of the process, the MindMaker selects an action, defined within the Action Space. Since the project has a discrete action space, of value '4', the agent selects an action between 0 and 3. These four possible actions correspond to those explained in section 6.2

It's important to note these functions take place in the Agent's AI Controller, which is a separate blueprint designed to handle the logic of any pawn (i.e. NPC) it is assigned to. This is done for both increased decoupling (so that an AI controller could be set to any number of NPC pawns), and because AI controllers have access to greater AI behaviours in UE4 that are not used in this project.

The blueprint code for the movement actions is as follows:

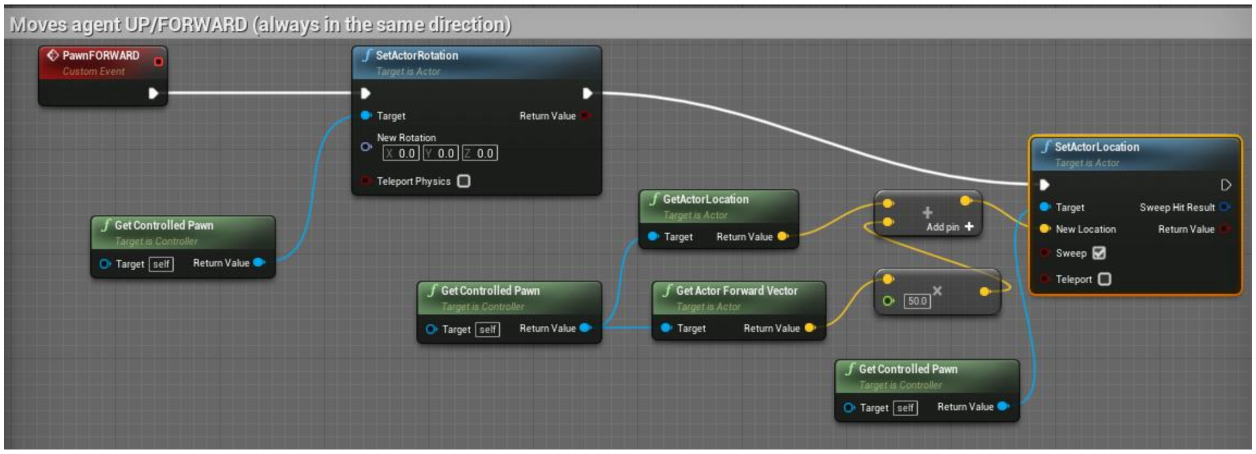


Figure 11: Code for the function 'Pawn FORWARD'

In plain English, when called, the event gets the controlled pawn, and sets its absolute rotation to 0,0,0, which ensures it always faces forwards. It then calls a Set Actor Location function, which takes a target, which in this case is the controlled pawn, and sets its destination to whatever is input. The destination is calculated by getting the actors current location, and the actor's forward vector, multiplying the forward vector by 50 (this is the distance we want the actor to move) and then adding that to the actor location. This effectively tells the agent to teleport forward in whatever direction it was set to face. Details on why the agent doesn't move naturally like a person are explained in Testing section 6.1.2.

The other actions, such as 'Pawn RIGHT' and 'Pawn Back', are coded identically, except that their rotations are set to different values. E.g. 'Pawn BACK' sets z = 180°. These can be viewed at Appendices.

5.6 Update the environment

Step 2. of the process involves updating the environment to reflect the action that was selected. This means simply moving the agent in 3D space. This Logic is contained within a function called 'Display Agent Actions'. in blueprints it looks like the following:

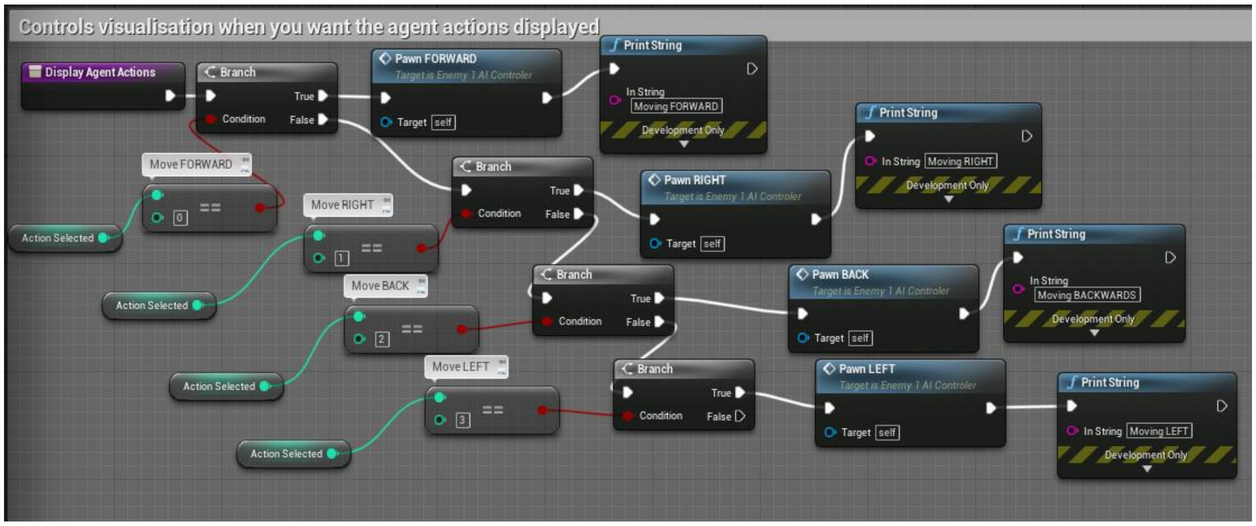


Figure 12: 'The Display Agent Actions' blueprint.

This function consists of nested if statements that check the value of the 'Action Selected Variable': if action selected equals 0, then call the 'Pawn FORWARD' function. this is repeated for every value of 'action selected' up to 3. The cold action then moves for agent in the correct direction. A string is also printed onto the screen which shows the user which action agent has taken. This is for debugging purposes.

5.7 Make observations and check reward

After the agent has update its environment, it makes observations and checks the reward it has received for the actions just taken. the observations that the agent in this project has access to are those shown in figure x. These are: 'Player Distance', 'Player Location' and 'Agent Location'.

At every tick, i.e. every frame, which is set to 30 frames per second, these events are called. This updates each of them to current values:

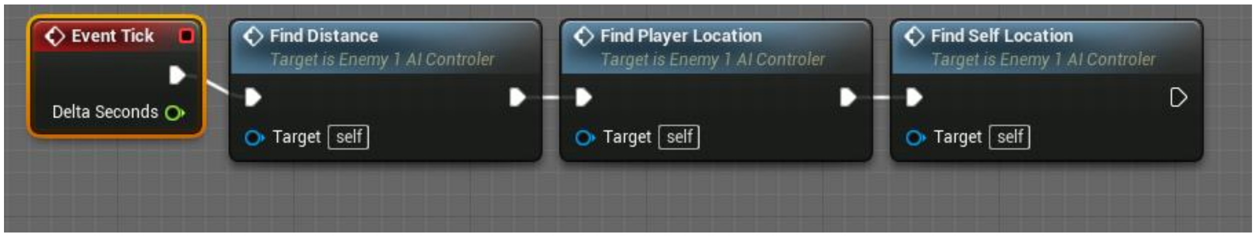


Figure 13: all three events are called every tick.

This ensures that the values for each variable are accurate when the agent makes observations and checks the reward signal. All three of these functions are contained within the AI controller Blueprint.

5.7.1 Player distance

Player Distance holds a float value of the distance in Unreal units between the agent and the player. The function that calculates it is called 'Find distance':

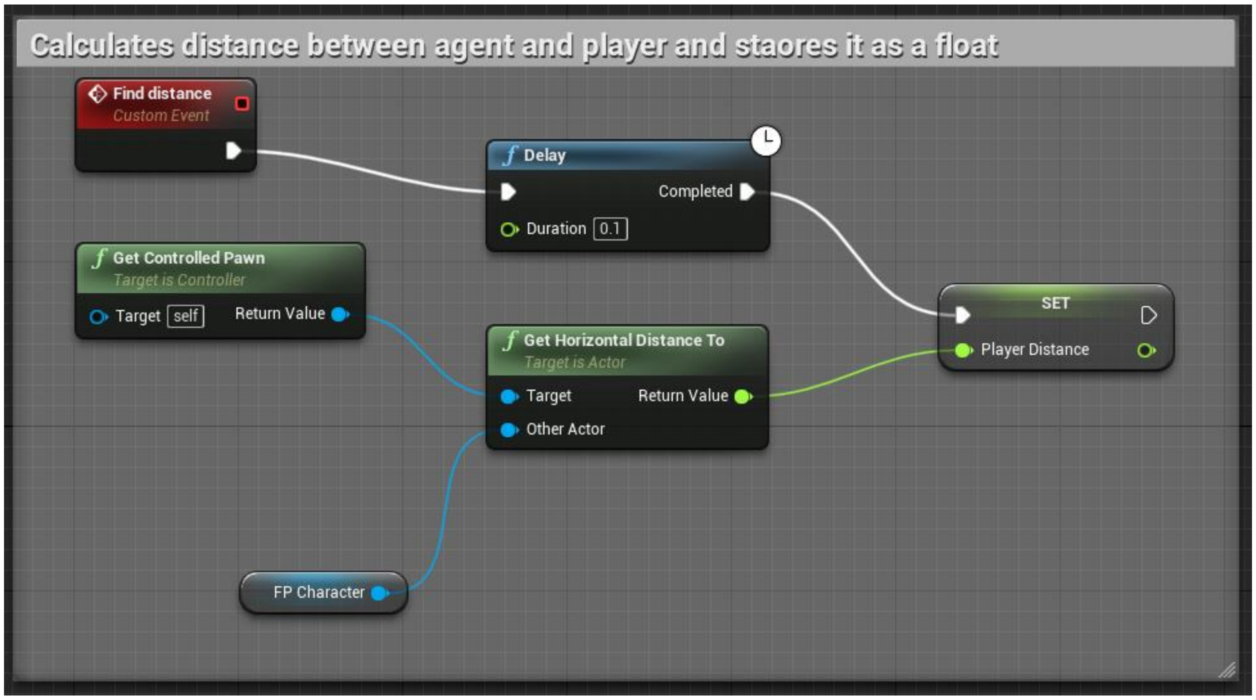


Figure 14: Calculating and storing Player Distance variable.

Description of execution:

***

Event called

Get the controlled pawn (the agent) and get a reference to the player (FP Character)

These are placed in the fields of the 'Get Horizontal distance to' function, which returns a float value.

Set Player Distance to this value.

***

5.7.2 Find Player location

Find Player Location finds the location of the player (the user), and returns a vector of x, y, and z values:

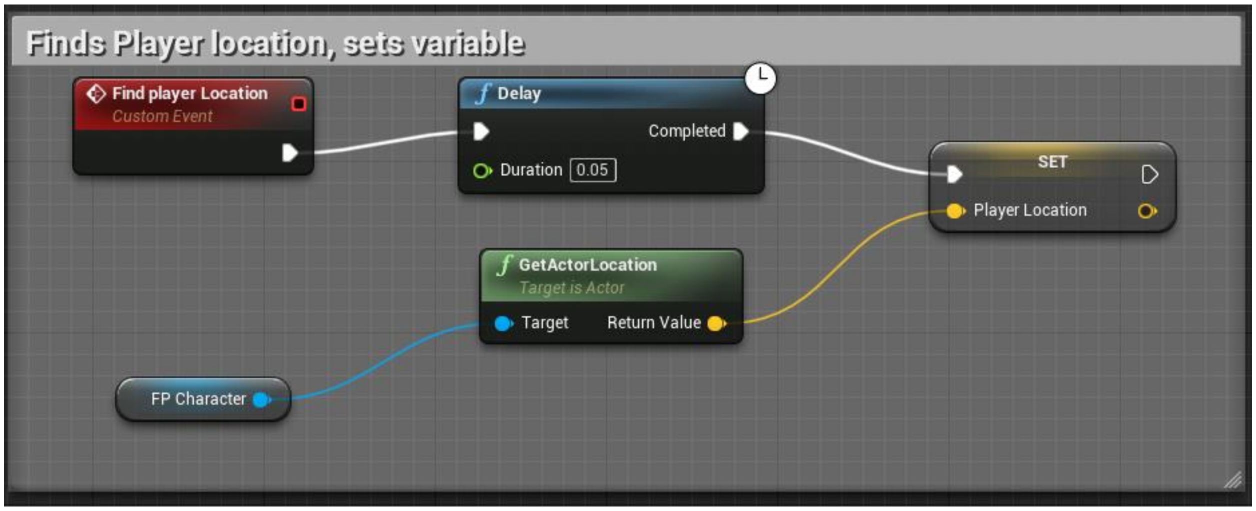


Figure 15: calculates and stores player location.

Description of execution:

***

Event called

Wait .05 seconds

Get the location of actor FP character – This is the player character, which returns a vector value

Set Player Location to this vector value

***

5.7.3 Find Self Location

Find Self Location is similar to the last function, but calculates and stores the location of the agent itself.

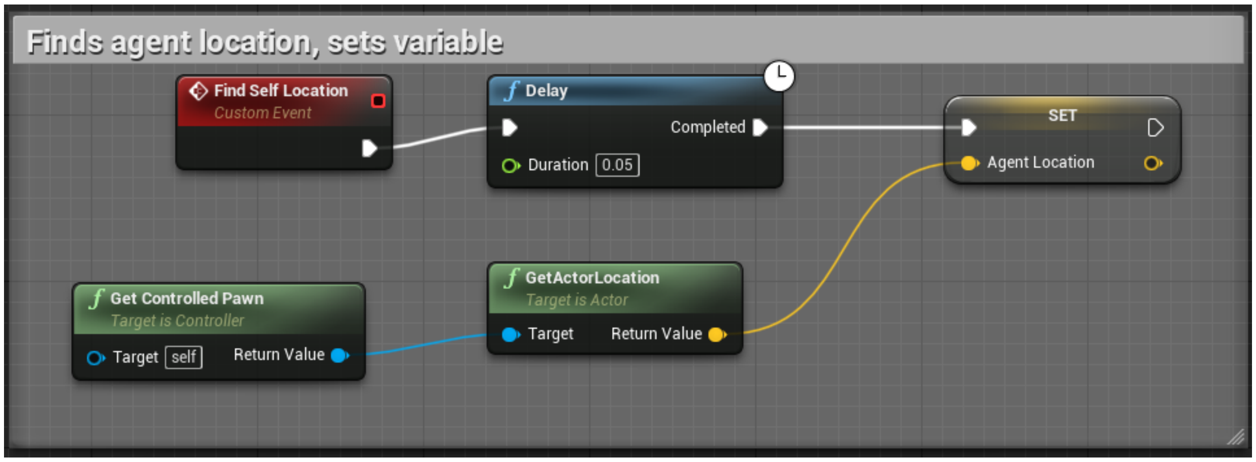


Figure 16: find Self Location

Description of execution:

***

Event called

Wait 0.05 seconds

Set the location of the controlled pawn (The agent) -> vector value

Set Agent Location to this return value.

***

These three variables are fed into the observations. Thus, the agent has access to: the distance between itself and the player, the player's location, and its own location. These are the variables it learns to associate with different reward signals.

5.8 Check Reward

The second part of this part for flow chart is checking the reward signal that the agent receives. My approach to calculating reward was based on the distance to the player. If the agent receives a greater reward the closer it is to the player, than an inverse function with the Player Distance should calculate reward.

Initially I found that I should check the value of Player Distance, then give a reward that decreases as that value grew in blocks. It looked like the following, in blueprint form:

***

```
If Player Distance < 150 then:
        Set reward = 100
If Player Distance < 400 then:
        Set reward = 50
```

```
If Player Distance < 650 then:
    Set reward = 20
Else
    Set Reward = -10
```

***

I found that this was not precise enough. The agent had to be in fairly close proximity to gain the smallest reward, and the punishment signal was the same if the agent was just outside of the reward distance, or if the agent was the furthest distance possible.

I then reworked the function to be more precise and nuanced. As a result, the reward signal was now continuous. The overall blueprint function for calculating the reward is shown in figure x.

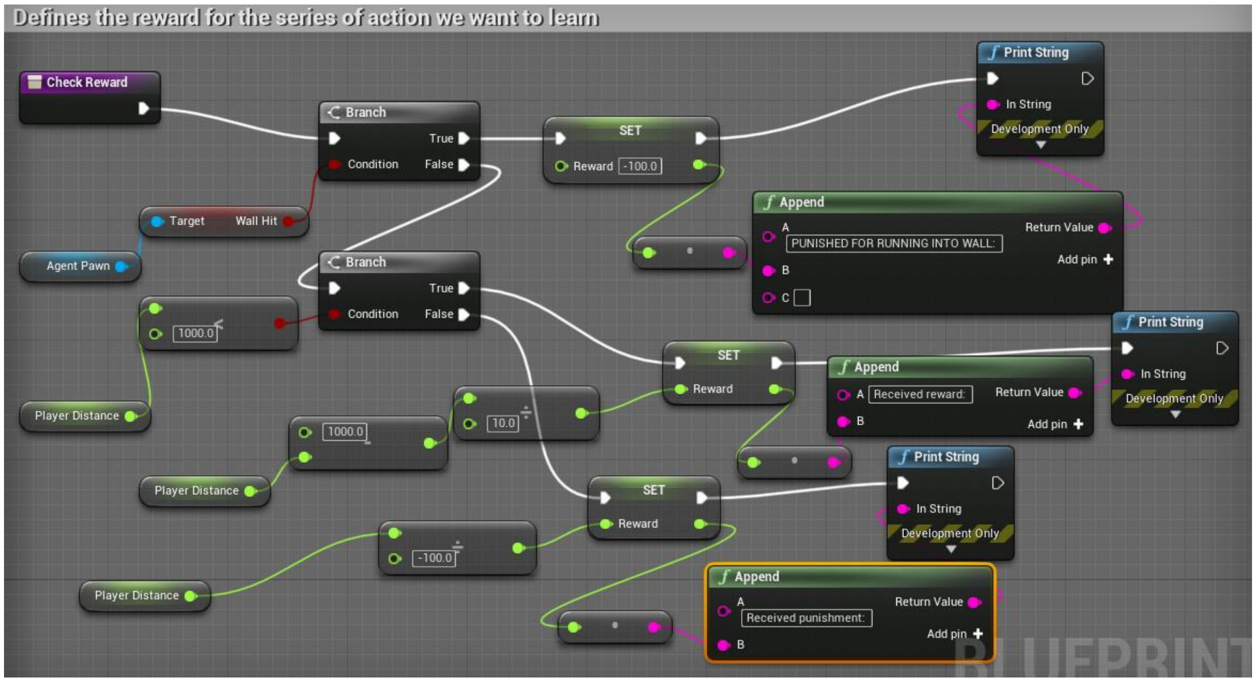


Figure 17: the blueprint function that checks the reward signal.

I will first describe the flow of execution, then explain its workings.

***

```
Check reward called
If WallHit = true:
    Set reward = -100
    Print (“Punished for running into wall: ”, reward)
    Break
```

```
Else if Player Distance < 1000:
        Set reward = (1000 - Player Distance) / 10
        Print (“received reward: “ reward)
Else:
        Set reward = (Player distance / -100)
        Print (“received punishment: ”reward)
```

***

The two branch nodes in the function check 1) if the WallHit variable is true, and 2) if the value of the play distance variable is less than 1000. If WallHit is set to true, the agent receives a large punishment. WallHit is true if the agent has collided with the boundaries or walls of the arena. Wall hit was a necessary addition to this Check Reward function because without it, the agent would not learn the boundaries of its environment. WallHit is combined with another function, contained within the agent model's blueprint that moves the agent to the centre of the arena if it detects a collision. This is explained in more detail in Testing, section 6.1.1.

If Player Distance < 1000, Then the agent gets a reward, calculated by (1000 – Player distance) / 10. This results in a linearly scaling reward that gets larger as Player distance decreases; at 900 units the reward is 9, and at 100 it is 90. Punishment is calculated similarly. If Player Distance > 1000, then reward is set to distance / -100. Notice the change in divisor from 10 to – 100; Negative ensures that reward will also be negative, while dividing by a larger (absolute) number ensures a generally smaller punishment for being over 1000 distance away. As the maximum distance an agent and player can be from each other is roughly 2500 units, I felt that maximum distances for punishment was too great, as dividing by minus 10 at this distance would still result in a punishment of -150 (2500 – 1000 /-10). The agent therefore receives a small to moderate reward for being close, and small punishment for being far away.

6. Testing

Overall, testing the programme focussed firstly on fixing bugs and obstacles that stopped that agent from training, and secondly on optimising the process so that the agent learned faster.

6.1 Bug fixing

6.1.1 Running into walls

As I began training, I first found that the agent would constantly get stuck on walls – when it collided with a wall it could not move, even when attempting to perform an action to move the other way. In order to overcome this, I implemented some logic that would solve this, and train the agent not to travel near the walls.

Within the agent model's blueprint (MindMakerCharacterBP), I implemented the following logic that would teleport the agent to its starting location if it touched the walls.

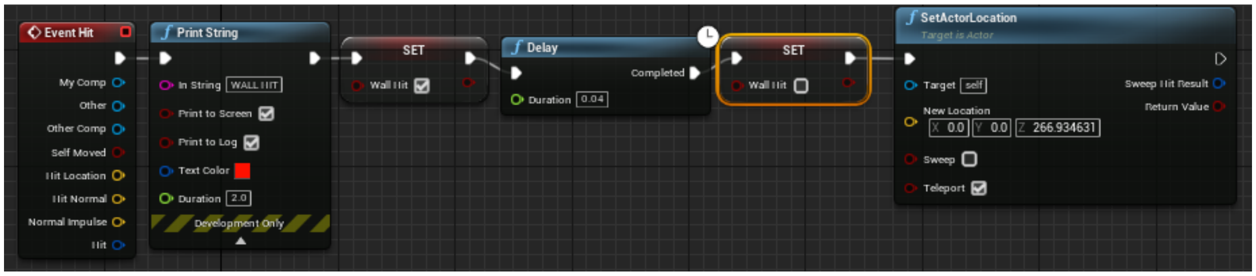


Figure 18: Function that teleports agent to starting location

Flow of execution:

***

Event called

Print string ("WALL HIT") – For debugging

Set Boolean variable 'WallHit' to true

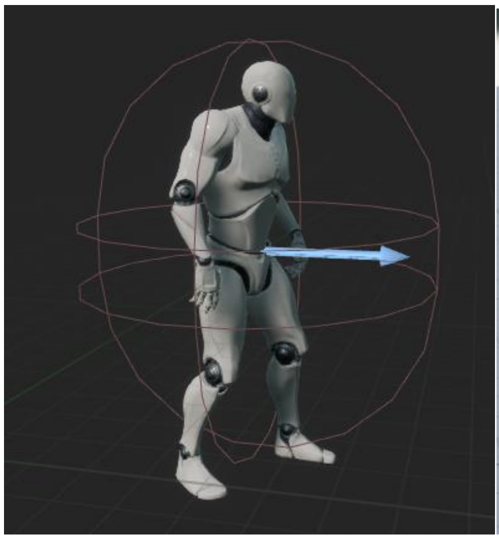
Wait 0.04 seconds

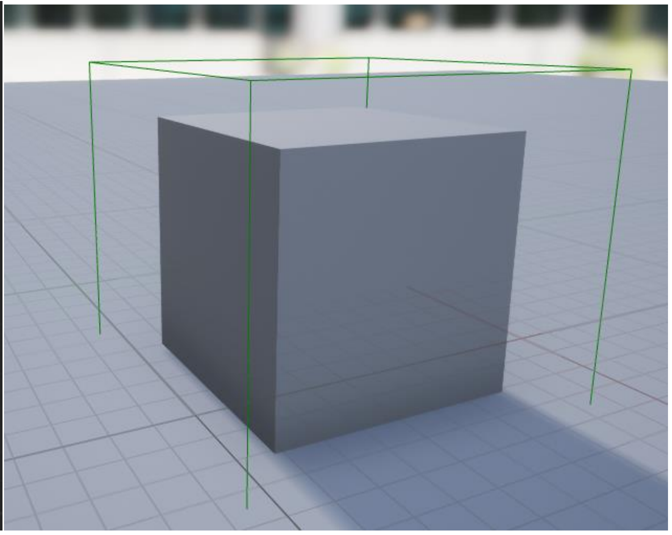
Set Boolean variable 'WallHit' to false

Set the location of self to the same location that the agent starts gameplay

***

In the last section I mentioned how WallHit affected the 'Check Reward' function. Since I wanted to punish the agent for touching a wall, WallHit is briefly set to true when this happens. The 0.04 second delay before setting it to false corresponds roughly the duration of a single frame (or 'tick') of a 30 frames-per-second game. This gives the 'Check Reward' function time to see that WallHit is True and then set the punishment to -100 for that frame's training episode.





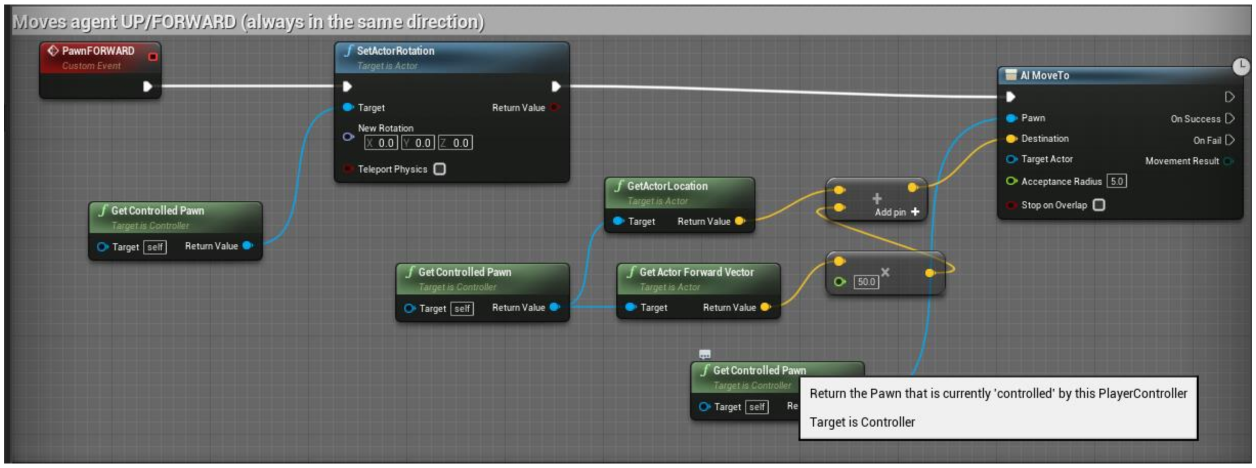
Figures 19 and 20: Collision boxes surround both the agent and the walls. If these two collide (touch), the event in figure x is called,

Although this worked well in moving the agent whenever it came into contact with the wall, the agent still did not seem to learn to not touch walls, nor even to learn to find the player. This was addressed in the next issue I solved.

6.1.2 Solving differences in agent movement and observations

MindMaker selects a new action every tick. This means that the agent should perform a new action 30 times a second, or once every 0.033 seconds. This meant that new actions were issued to the agent before it could complete the previous one. This caused dissonance between the observations that the agent made, and what they should have been. The slow acceleration and movement speed of the agent was the root cause of these issues. However, I found that despite increasing movement speed and acceleration, the agent was still failing to learn properly.

To remedy this, I changed the movement functions. Originally, they looked like this:



Return the Pawn that is currently 'controlled' by this PlayerController

Target is Controller

Figure 21: faulty movement function.

The only difference is that final line of execution is an AI MoveTo, compared to the current Set Actor Location. This function tells the pawn to walk to the specified destination, while the current solution teleports the pawn there. Changing these movement functions fixed the problem, and now the agent Movements did not lag behind its observations. This mean that the agent started learning faster after making this change.

6.2 Optimisation

I felt that the location variables Player Location and Agent Location did not have to be float values.

I decided to make some changes in the 'Make Observations' function in order to try and make the decrease the number of possible values the agent had access to. This was done by rounding the float values extracted from the player and agent location vectors.

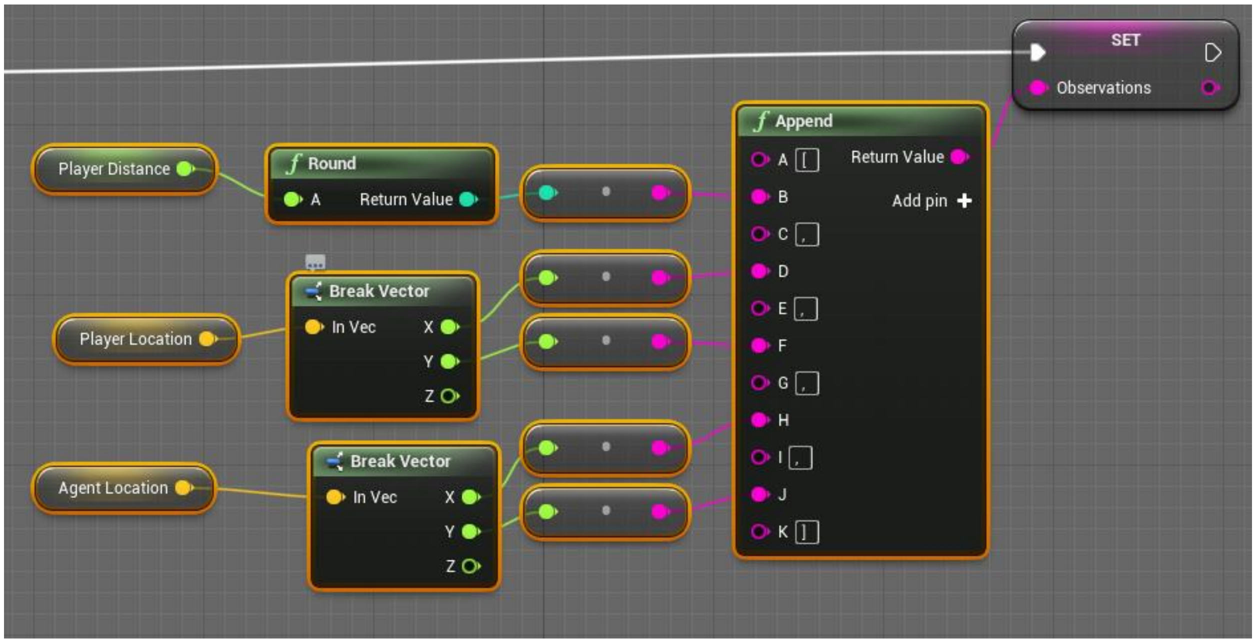
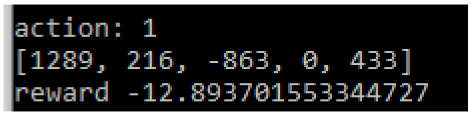


Figure 22: the original 'Make Observations' function, before rounding location variables for the player and agent.

I'm unsure if this made training any faster. However, I am in no doubt that it decreased the possible values in the training space.



```
action: 1
[1289, 216, -863, 0, 433]
reward -12.893701553344727
```

Figure 23: a screenshot of the observations produced during a training episode

The console output in figure shows, in order from top to bottom, the action the agent took, it's observations, and the reward it received. In this case, it moved eastwards/right (action = 1), the distance from the player was 1289 units, The players location was (216, -863), and the agent's location was (0,433). It received a reward of -12.89, which was punishment for being too far away.

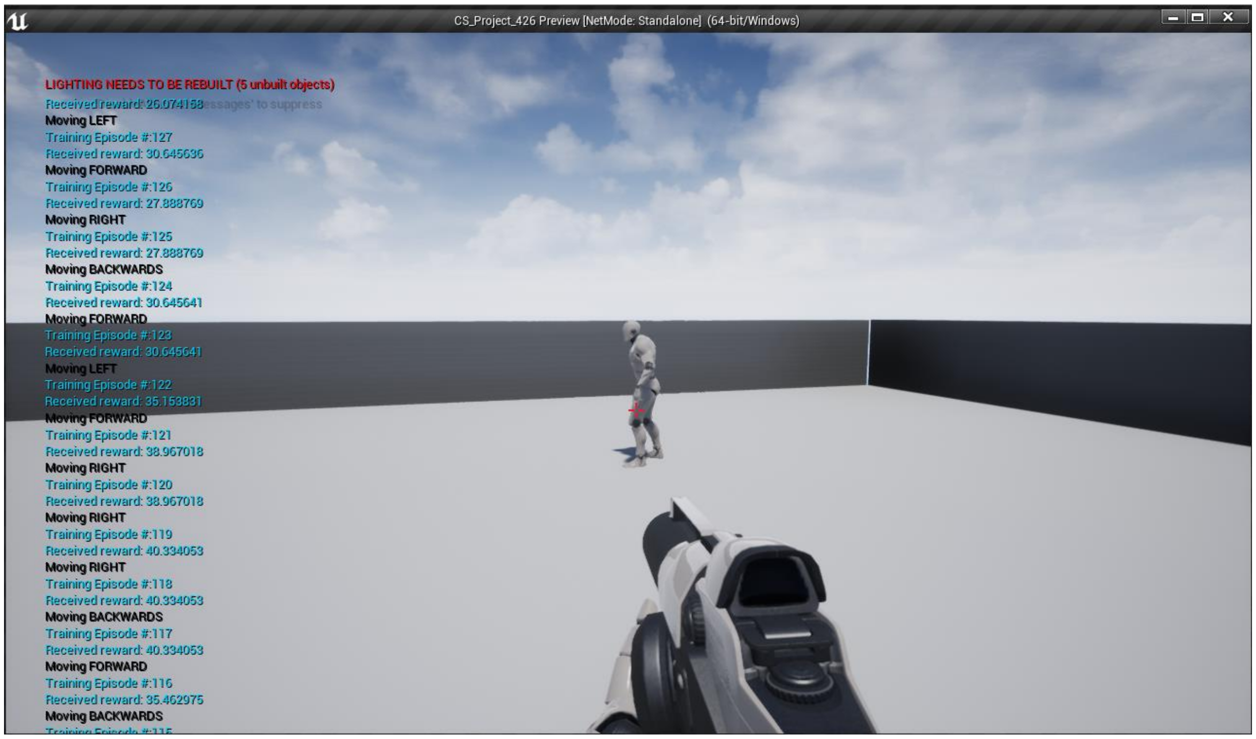
6.3 Using PPO2 and other algorithms

In my proposal, and in the background section of this report, I had suggested that I would use Qlearning as the algorithm to train the agent, as my action space was discrete. DQN was reasonably fast in training an agent to go to a stationary player, with 3000 episodes being more than enough for the agent to know to go to the spot the player was standing. PPO2 about the same, and I found it trained agents on stationary targets in about the same time.

Unfortunately, I could not other algorithms packaged with MindMaker - A2C, TD3, and SAC - to work within the project. This means I cannot comment on their potential effects.

7. Results and Analysis

7.1 Project result


CS_Project_426 Preview [NetMode: Standalone] (64-bit/Windows)

```
LIGHTING NEEDS TO BE REBUILT (5 unbuilt objects)
Received reward: 26.074158essages' to suppress
Moving LEFT
Training Episode #:127
Received reward: 30.645636
Moving FORWARD
Training Episode #:126
Received reward: 27.888769
Moving RIGHT
Training Episode #:125
Received reward: 27.888769
Moving BACKWARDS
Training Episode #:124
Received reward: 30.645641
Moving FORWARD
Training Episode #:123
Received reward: 30.645641
Moving LEFT
Training Episode #:122
Received reward: 35.153831
Moving FORWARD
Training Episode #:121
Received reward: 38.967018
Moving RIGHT
Training Episode #:120
Received reward: 38.967018
Moving RIGHT
Training Episode #:119
Received reward: 40.334053
Moving RIGHT
Training Episode #:118
Received reward: 40.334053
Moving BACKWARDS
Training Episode #:117
Received reward: 40.334053
Moving FORWARD
Training Episode #:116
Received reward: 35.462975
Moving BACKWARDS
```

Figure 24: The agent in training

The project yielded a product that demonstrated an AI agent able to be taught to chase a player character. This is successful. When starting up the project, the agent has no set behaviour, and takes random actions. However, with a significant number of training episodes, the agent begins to learn that is should move towards the player, thus enabling the chase behaviour that is intended.

This trained model can then be saved and used in the future to have set behaviours for agents. Then, as this model is applied to an NPC, the game can be exported to an executable game that has the agent trained and chasing the player from the start.

Various models trained models have been created, some simple, others more complex. In my projects accompanying GitHub repo, I have included 3 trained models that I feel demonstrate the achievements of the project. The first is Stationary_PPo2, a model where the agent has learned to come if player stands still. StationaryDQN is another model that moves to the players location, if they are stationary in the starting location. The last is MovingPPO2, an agent that has learnt to somewhat chase the player.

Unfortunately, I was not able to export a ‘shipped’ build that can be run without Unreal Engine. I unfortunately encountered a few critical errors during project packaging that I could not overcome. Despite a large number of attempts to remedy this I could not get it to export. Fortunately, the project can still be experienced, it just entails installing Unreal Engine to the computer that will run the models. Instructions on how to run these are in the README of this project's GitHub repo.

7.2 Comparing the result to initial goals – strengths and weaknesses

7.2.1 Strengths

Comparing what I have created to the goals, I can be certain of completing objective one. It is clear that the agents have learned behaviour. During the initial testing, I produced models that clearly learned to go to the players location, albeit when they were standing still.

Another strength of the project is its extensibility. Now that I have created an environment and set it up so that the agent can learn within it, I am able to increase the complexity of the task, or even change what the agent is being rewarded for altogether.

Overall I feel that the project is a success. I still feel that it remains useful and interesting to those I thought would be interested in the project in section 3.1.2

7.2.2 Weaknesses

One certain weakness of the project is producing models that had learned to chase/follow the player. This was more challenging than the previous task. The agents definitely learned to move in the general direction of the player, but it was hard to get them to learn to follow the player properly. Therefore, I think I can only partially claim to meet objective two. I think that the issue here is consistent training. They need more training episodes to associate a shorter player distance with the reward, since the player location constantly changes, as opposed to a stationary player, where the location variable is constant, and thus easier to associate with the reward.

Another issue with my project is the lack of gameplay elements. Although the aim of the project was purely based on the AI, the lack of gameplay elements mean it may be hard to call the project a 'game', in the sense that a game provides entertainment value. Including features like a hit counter or changing the textures and models to more resemble a simple but playable game would improve this issue.

7.3 The effect of Policies

Policies made some impact on the project, though not too great. I expected PPO2 to be moderately faster in all situations but found no discernible difference between the two when training the agent for a stationary player.

However, I did find that PPO2 was noticeably faster when training the agent on a moving player. This was reasonably expected – DQN is considered slower to train in general [22].

7.4 The effect of the environment

The size and scale of the environment has a huge impact upon the complexity of the agent. Factors like having a large arena space had large impact on training times. In the early stages of building my project, I was forced to decrease the size of the arena, and remove all obstacles and features. Although this does not affect the outcome of this project, I think that it has an impact on how useful RL agents will be in commercial game development.

My environment was effectively a 2D square, as the Z axis was not utilised. The same cannot be said for the most 3D games, which make extensive use of all three dimensions. These worlds often have slopes, ramps, and ledges, which may change dynamically, or in some cases be randomly generated [23]. To train agents here would require a significant deal of computational power, as the number of variables increases significantly. While I was able to simplify my learning environment to increase training speed, this is not feasible for developers creating modern video games, which are often incredibly feature rich [24].

7.5 Alternative approaches

There are certainly other ways to achieve a similar outcome to what I produced. The most significant, and general change I could have made would have been to use another game engine. I mentioned Unity in a previous section, which has good RL packages available, such as Unity ML [25]. However, I found that blueprints was incredibly intuitive and decided to continue using Unreal Engine.

In terms of training, there are definitely different ways I could have made the agent chase the player. Changing what the agent observes would have made a big difference. I had considered perhaps utilising Unreal Engine AI perception feature as an observation variable – but wasn't sure how to implement it. Additionally, another method I considered was having the AI agent shoot a line trace (essentially a laser that can trigger logic) directly out in front of it at all times. If this laser collided with the player, then it would be rewarded. I decided against this approach as I thought it would require significant training to implement, and certainly more than the approach I went with.

8. Evaluation and conclusion

8.1 Possible additions

With I am satisfied with my project, I believe that with additional time and agent training, I could add features to the project that would positively affect a user's experience and enjoyment of playing.

Firstly, I would increase the agent's action space. Currently the agent only has four actions, it can move north, east, south, and west. Expanding the agents action space include four additional actions, would increase training times, but be a simple addition to make. These actions would be to move NW, NE, SW, and SE, taking the total number of possible actions to eight. This would make the agent move less like a snake game, and more like a natural 3D character.

I also believe that the agent, with enough training, could have its behaviour extended, such that it could be suitable for different types of games. In particular, multi-agent environments, that is, NPCs controlled by separate agents that learn independently. I believe that the largest barrier to this would be training times, as the machine running the training would have to do twice as much computation, even more so if the agents can observe each other, which adds more variables to train for [26].

Further additions would focus mainly on non RL features. I mentioned in section 8.2 that an issue with my project is a lack of gameplay elements. These would be features that would be added next with additional time. Features like a simple counter that would count how many times the agent came within a 'hit' radius of a player would add fun gameplay elements to the programme. I would also like to add features to change gameplay elements as time goes on. Spawning multiple NPCs (all controlled by the same RL-trained model) would add to a 'zombie hoard' type of gameplay experience, these NPCs could be faster or perhaps be controlled by different models so that they move differently

Overall, I think that these features would improve my project from an AI showcase to a fleshed-out game.

8.2 Conclusion: Overall discussion of RL usage to power NPCs

RL's current strength is its ability to simulate human behaviour, and in games such as Dota 2 or StarCraft, they can do it very well. I think that this creates a dilemma for RL in video games. Most developers don't want NPCs that are smarter than people; they want NPCs that fill very specific roles [27]. Only in player-vs-player games, are bots with human-like behaviour desired. This, combined with the enormous resource cost to produce agents of suitable quality, means that there is little point in investing in producing them.

Consider the issues with the agent's environment discussed in the last section. Most video games consist of more than one level, and the different combinations of what is currently in the agent's environment or not leads to thousands upon thousands of different situations it has to be trained in. In Open AI five, the agents were limited to a pool of just 17 heroes out of a total of 120

[9], due to the large number of different hero combinations that a 5 vs 5 game include. The agents had to be trained again for every change in their environment.

Existing non-RL methods produce sufficient AI behaviour arguably better and far quicker – all with a fraction of the resources. In fact, during my time learning Unreal Engine, I found that the built-in tools that the Unreal engine had for creating AI behaviour were sophisticated and fleshed out [28]. I found that I could create non-RL NPC behaviour swiftly – getting an NPC to follow the player using behaviour trees took less than 10 minutes. In addition to this, it is much simpler to create simple AI behaviour that can be used in multiple scenarios and environments, with only minimal adaptation. For example, while learning Unreal engine, I found that an NPC character could be moved from one level to the next and still act in a function well, and all I had to do was place a new navigation mesh (a level component that tells the AI where it can walk) [28].

Therefore, I think that currently RL video games have two key problems holding back widespread implementation. The first problem is resource cost - They currently cost far too much in resources and computational power to develop a quality agent. The second problem is inflexibility - RL agents are only effective in environments that have been pre trained in and thus are not flexible in being transferred to environments with variables that they are unfamiliar with [29]. This means having to train agents for all possible situations they would find themselves in,

Thus, I believe that RL video game implementations, such as Open AI Five and AlphaStar, will remain useful for simulating Human-like behaviour in video games, but not for developing typical NPC AIs.

References

[1] Lode, "*Raycasting*", Lodev.org, 2020. [Online].
Available: https://lodev.org/cgtutor/raycasting.html. [Accessed: 14- Apr- 2021].

[2] K. Thor Jenson, "*The Complete History Of First-Person Shooters*", PCMAG, 2017. [Online]. Available: https://www.pcmag.com/news/the-complete-history-of-first-person-shooters. [Accessed: 14- Apr- 2021].

[3] Source Modding Community, "Navigation Meshes", *Developer.valvesoftware.com*, 2021. [Online]. Available: https://developer.valvesoftware.com/wiki/Navigation_Meshes. [Accessed: 14- Apr- 2021].

[4] The AlphaStar Team, "AlphaStar: Mastering the Real-Time Strategy Game StarCraft II", DeepMind, 2019. [Online]. Available: https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii. [Accessed: 14- Apr- 2021].

[5] R. S. Sutton and A. G. Barto, *Reinforcement Learning, an Introduction*. 2nd ed. Cambridge, MA: MIT Press, 2018.

[6] A. Kathuria, "Getting Started With OpenAI Gym | Paperspace Blog", Paperspace Blog, 2021. [Online]. Available: https://blog.paperspace.com/getting-started-with-openai-gym/. [Accessed: 24- Sep- 2021].

[7] D. Silver, "*Reinforcement Learning 10: Classic Games Case Study*", Youtube.com, 2018. [Online]. Available: https://www.youtube.com/watch?v=ld28AU7DDB4. [Accessed: 26- Mar- 2021].

[8] "*Artificial intelligence: Google's AlphaGo beats Go master Lee Se-dol*", BBC News, 2016. [Online]. Available: https://www.bbc.co.uk/news/technology-35785875. [Accessed: 14- Apr- 2021].

[9] "*OpenAI Five Defeats Dota 2 World Champions*", OpenAI, 2019. [Online]. Available: https://openai.com/blog/openai-five-defeats-dota-2-world-champions/. [Accessed: 28- Mar- 2021].

[10] G. Balaji, "What is Dota 2 : Everything you need to know about the one of the most played MOBA Esports Title by Valave | The SportsRush", The SportsRush, 2021. [Online]. Available: https://thesportsrush.com/what-is-dota-2-everything-you-need-to-know-about-the-one-of-the-most-played-moba-esports-title-by-valave/. [Accessed: 26- Sep- 2021].

[11] "I played 1,000 hours of Dota 2, this is my review", The Verge, 2014. [Online]. Available: https://www.theverge.com/gaming/2014/9/29/6862757/dota-2-the-1000-hour-review. [Accessed: 14- Apr- 2021].

[12] "AlphaStar: Grandmaster level in StarCraft II using multi-agent reinforcement learning", DeepMind, 2019. [Online]. Available: https://deepmind.com/blog/article/AlphaStar-Grandmaster-level-in-StarCraft-II-using-multi-agent-reinforcement-learning. [Accessed: 12- Apr- 2021].

[13] "DeepMind claims breakthrough as AI masters StarCraft II game", Ft.com, 2021. [Online]. Available: https://www.ft.com/content/d659b056-fb28-11e9-a354-36acbbb0d9b6. [Accessed: 28- Sep- 2021].

[14] "About", Deepmind, 2021. [Online]. Available: https://deepmind.com/about. [Accessed: 27- Sep- 2021].

[15] T. Simonini, "*Diving deeper into Reinforcement Learning with Q-Learning*", Medium.com, 2018. [Online]. Available: https://medium.com/free-code-camp/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe. [Accessed: 14- Apr- 2021].

[16] ADL, "*An introduction to Q-Learning: reinforcement learning*", freeCodeCamp.org, 2018. [Online]. Available: https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/. [Accessed: 14- Apr- 2021].

[17] "Unreal Engine | The most powerful real-time 3D creation platform", Unreal Engine, 2021. [Online]. Available: https://www.unrealengine.com/en-US/. [Accessed: 28- Sep- 2021].

[18] "Blueprint Visual Scripting", Docs.unrealengine.com, 2021. [Online]. Available: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/. [Accessed: 30- Sep- 2021].

[19] A. Krumins, "MindMaker AI Plugin", Unreal Engine Marketplace, 2020. [Online]. Available: https://www.unrealengine.com/marketplace/en-US/product/mindmaker-ai-plugin. [Accessed: 26- Mar- 2021].

[20] A. Krumins, 2021. [Online]. Available: https://www.autonomousduck.com/mindmaker.html. [Accessed: 27- Sep- 2021].

[21] "First Person Shooter Tutorial", Docs.unrealengine.com, 2021. [Online]. Available: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/CPPTutorials/FirstPersonShooter/. [Accessed: 30- Sep- 2021].

[22] "Algorithms — Spinning Up documentation", Spinningup.openai.com, 2021. [Online]. Available: https://spinningup.openai.com/en/latest/user/algorithms.html#the-on-policy-algorithms. [Accessed: 30- Sep- 2021].

[23] "Procedural Generation", Mit.edu, 2021. [Online]. Available: http://www.mit.edu/~jessicav/6.S198/Blog_Post/ProceduralGeneration.html#:~:text=What%20is%20Procedural%20Generation%3F,or%20non%2Dplayer%20character%20dialogue. [Accessed: 29- Sep- 2021].

[24] L. Nacke and C. Lindley, Flow and Immersion in First-Person Shooters: Measuring the player's gameplay experience. 2021.

[25] U. Technologies, "Machine Learning Agents | Unity", Unity, 2021. [Online]. Available: https://unity.com/products/machine-learning-agents. [Accessed: 30- Sep- 2021].

[26] "Multi-Agent Reinforcement Learning (MARL) and Cooperative AI", Medium, 2021. [Online]. Available: https://towardsdatascience.com/ive-been-thinking-about-multi-agent-reinforcement-learning-marl-and-you-probably-should-be-too-8f1e241606ac. [Accessed: 30- Sep- 2021].

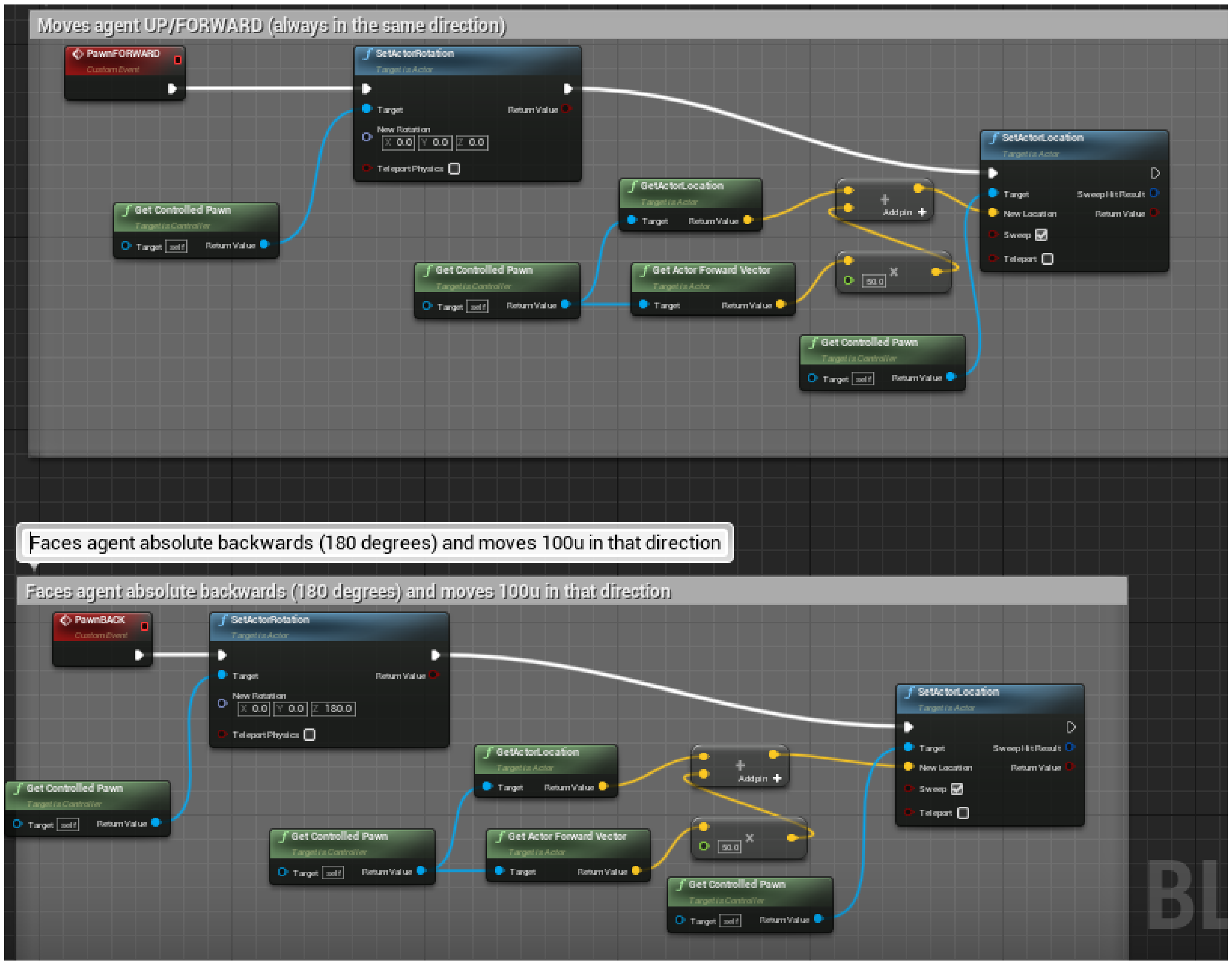
[27] M. McPartland and M. Gallagher, Creating a Multi-Purpose First Person Shooter Bot with Reinforcement Learning. IEEE, 2021.

[28] "Behavior Tree Overview", Docs.unrealengine.com, 2021. [Online]. Available: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/BehaviorTreesOverview/. [Accessed: 30- Sep- 2021].

[29] A. Krumins, "Creating Next-Gen Video Game AI With Reinforcement Learning", Medium, 2021. [Online]. Available: https://towardsdatascience.com/creating-next-gen-video-game-ai-with-reinforcement-learning-3a3ab5595d01. [Accessed: 30- Sep- 2021].

Appendixes

Agent movement functions – forward and back

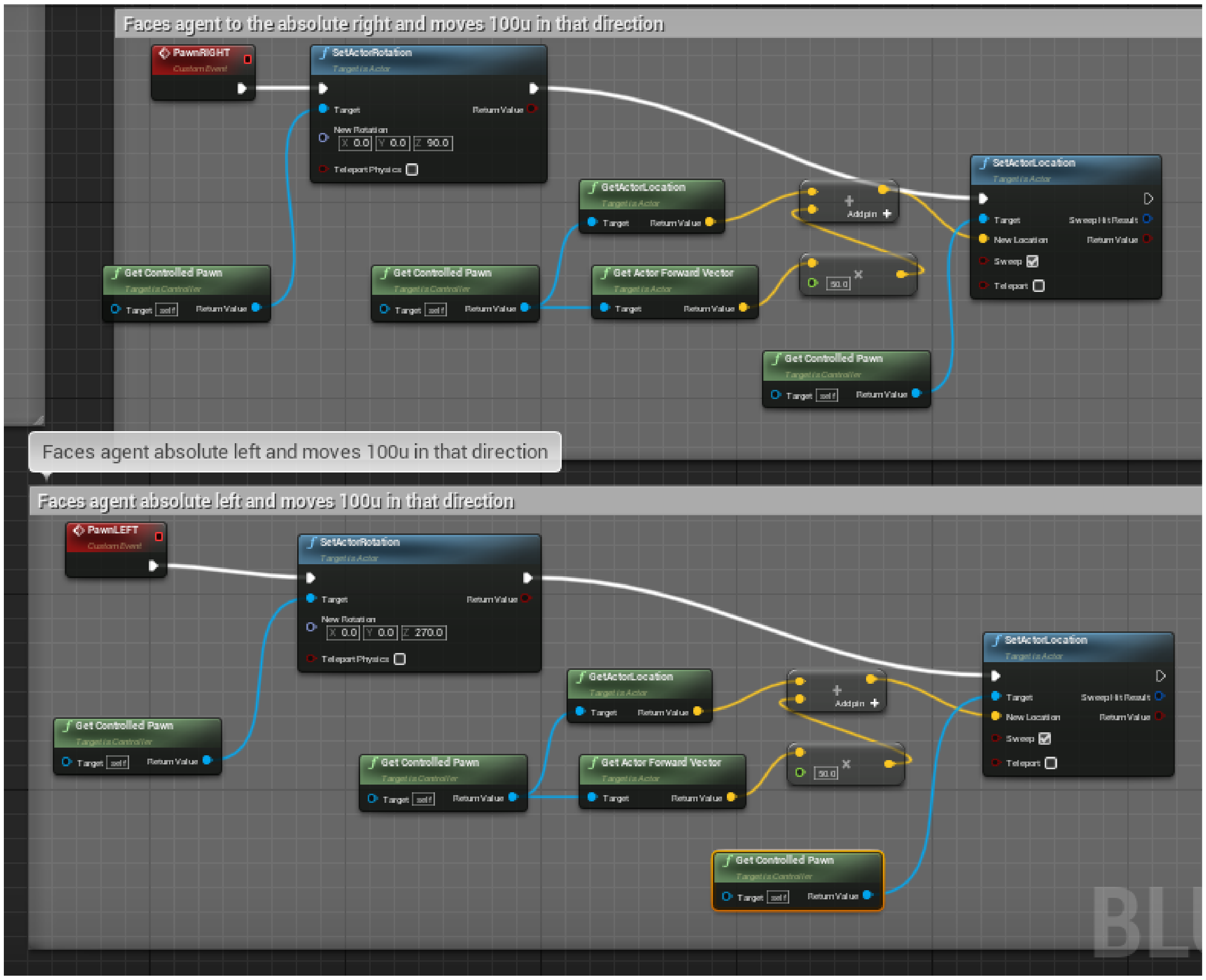


Moves agent UP/FORWARD (always in the same direction)

Faces agent absolute backwards (180 degrees) and moves 100u in that direction

Faces agent absolute backwards (180 degrees) and moves 100u in that direction

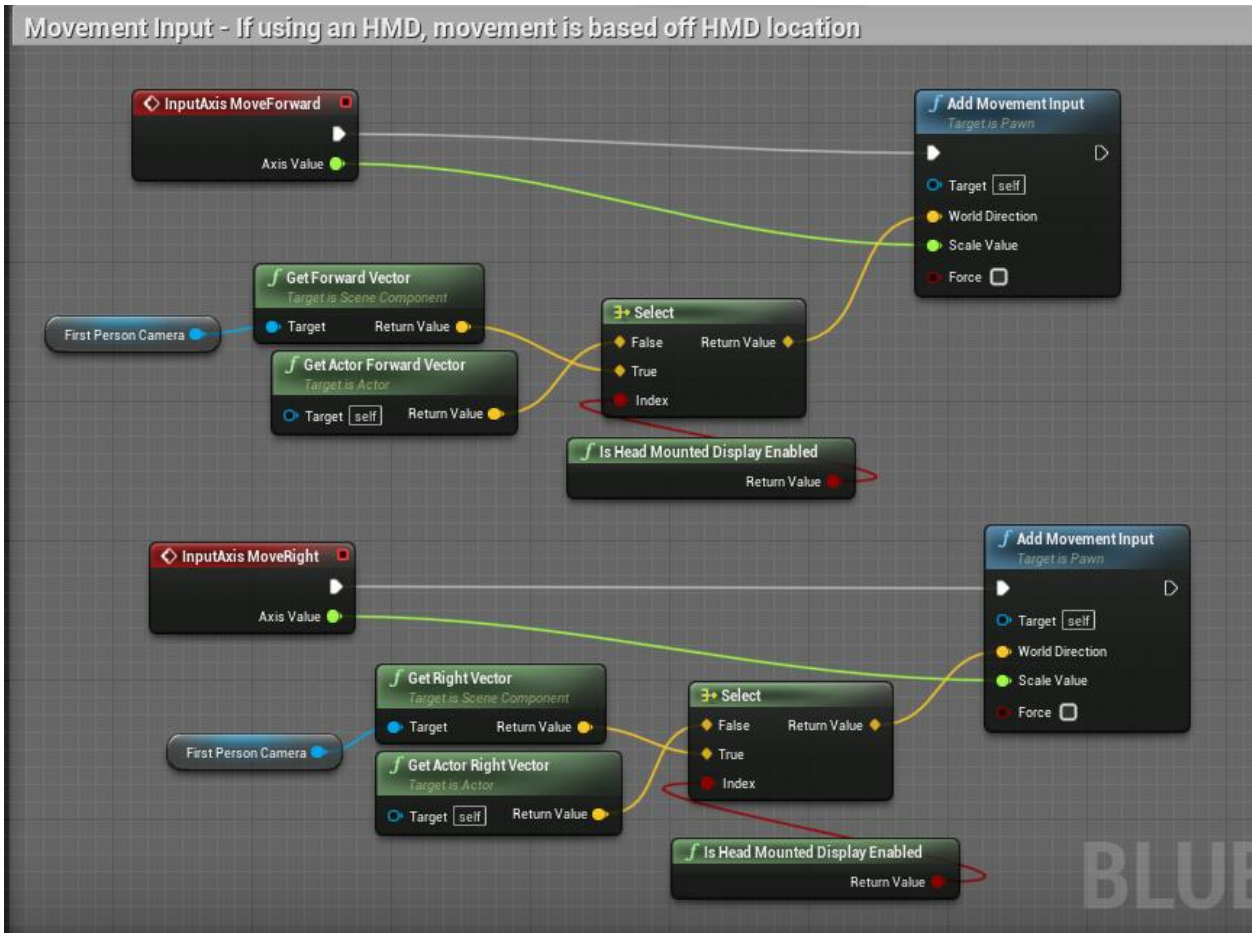
Agent movement functions– Right and Left


Faces agent to the absolute right and moves 100u in that direction

Faces agent absolute left and moves 100u in that direction

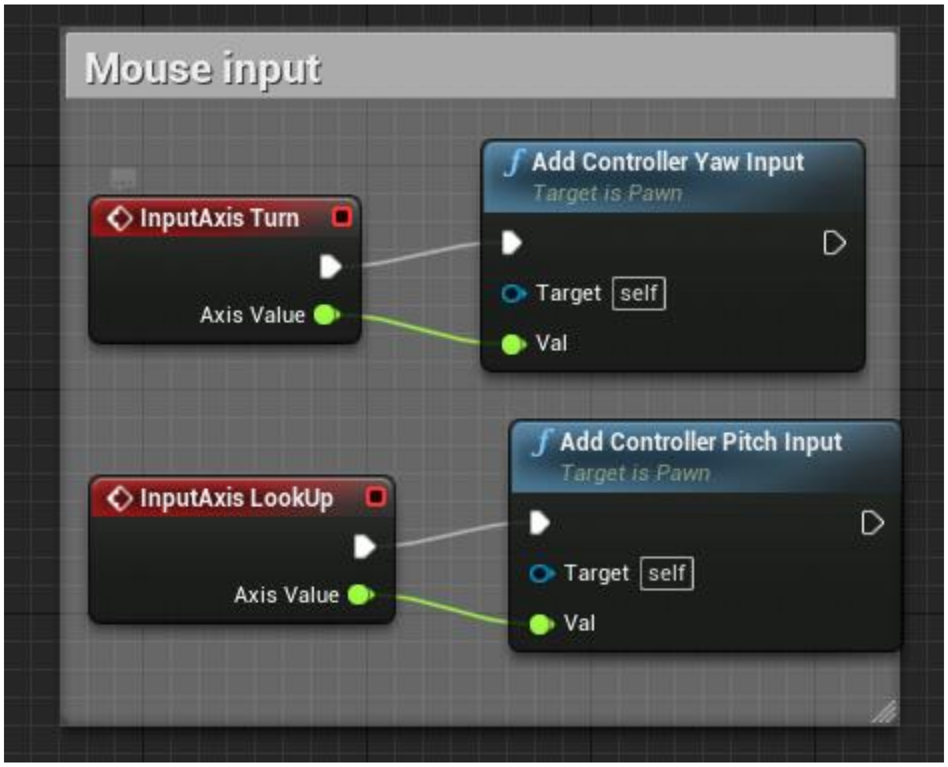
Faces agent absolute left and moves 100u in that direction

Player movement functions – keyboard movement. Note: part of the level template by Unreal Engine, not by me.



Functions f

Player movement functions – Mouselook movement. Note: part of the level template by Unreal Engine, not by me.


Mouse input
InputAxis Turn
Axis Value
Add Controller Yaw Input
Target is Pawn
Target
self
Val
InputAxis LookUp
Axis Value
Add Controller Pitch Input
Target is Pawn
Target
self
Val