A picture containing text

Description automatically generated

Using Reinforcement Learning for navigation in a complex 3-dimensional environment

Jack Patterson – T00217640

B.Sc. (Hons) Computing with Games Development

Supervisor: Robert Sheehy

Final Year Project

Table of Contents

[**Table of Contents** 2](#_Toc165021145)

[Abstract 4](#_Toc165021146)

[Glossary 5](#_Toc165021147)

[1 Introduction 5](#_Toc165021148)

[2 Navigation 6](#_Toc165021149)

[2.1 What is Navigation? 6](#_Toc165021150)

[2.2 Navigation Methodologies in 3D Environments 6](#_Toc165021151)

[2.2.1 Waypoint Method 6](#_Toc165021152)

[2.2.2 Navigation Mesh Method 6](#_Toc165021153)

[2.3 Pathfinding Algorithms 7](#_Toc165021154)

[2.3.1 Undirected Algorithms 7](#_Toc165021155)

[2.2.2 Directed Algorithms 8](#_Toc165021156)

[3 Machine Learning 9](#_Toc165021157)

[3.1 What is Machine Learning? 9](#_Toc165021158)

[3.2 Reinforcement Learning 9](#_Toc165021159)

[3.2.1 Exploration 9](#_Toc165021160)

[3.2.2 Policy 9](#_Toc165021161)

[3.2.3 Rewards and Punishments 10](#_Toc165021162)

[3.3 Imitation Learning 10](#_Toc165021163)

[3.3.1 Usage with Reinforcement Learning 11](#_Toc165021164)

[4 Methodology 12](#_Toc165021165)

[4.1 Technologies Used 12](#_Toc165021166)

[4.2 Functional Specifications 12](#_Toc165021167)

[4.3 Implementation Plan 13](#_Toc165021168)

[4.4 Class Diagram 14](#_Toc165021169)

[4.5 Prototype 14](#_Toc165021170)

[4.5.1 Tasks 14](#_Toc165021171)

[4.5.2 Process 15](#_Toc165021172)

[5 Implementation 18](#_Toc165021173)

[5.1 Sprint 1 18](#_Toc165021174)

[5.1.1 Tasks 18](#_Toc165021175)

[5.1.2 Task 1 18](#_Toc165021176)

[5.1.2 Task 2  18](#_Toc165021177)

[5.1.3 Task 3 19](#_Toc165021178)

[5.1.4 Task 4 19](#_Toc165021179)

[5.2 Sprint 2 21](#_Toc165021180)

[5.2.1 Tasks 21](#_Toc165021181)

[5.2.2 Task 1 21](#_Toc165021182)

[5.2.3 Task 2 21](#_Toc165021183)

[5.3 Sprint 3 22](#_Toc165021184)

[5.3.1 Tasks 22](#_Toc165021185)

[5.3.2 Task 1 22](#_Toc165021186)

[5.3.3 Task 2 22](#_Toc165021187)

[5.4 Sprint 4 23](#_Toc165021188)

[5.4.1 Tasks 23](#_Toc165021189)

[5.4.2 Task 1 23](#_Toc165021190)

[5.4.3 Task 2 24](#_Toc165021191)

[5.4.4 Task 3 24](#_Toc165021192)

[5.5 Sprint 5 24](#_Toc165021193)

[5.5.1 Tasks 24](#_Toc165021194)

[5.5.2 Task 1 24](#_Toc165021195)

[5.6 Sprint 6 26](#_Toc165021196)

[5.6.1 Tasks 26](#_Toc165021197)

[5.6.2 Task 1 27](#_Toc165021198)

[6 Results 28](#_Toc165021199)

[7 Conclusion 29](#_Toc165021200)

[8 References 30](#_Toc165021201)

# Abstract

The rapid evolution of video game technologies requires navigational systems and technologies to evolve in tandem with it. Increasingly, there is a focus on non-player characters (NPCs) that operate autonomously of players. This paper examines the use of traditional navigation techniques with new machine learning techniques that aim to make NPCs more responsive and adaptable within three-dimensional environments. For example, the Unity ML-Agents package can be utilised to train an AI and get it to learn and adapt to complex three-dimensional environments in the Unity game engine, which can mimic and simulate the dynamic and ever-changing unpredictability of the real world.

Traditional navigational methods, which have long dominated the field, typically involve predefined paths that do not account for dynamic changes within the environment. Waypoints, the simplest form of navigation, provide a series of coordinates that NPCs follow, often resulting in predictable movements. Navigation meshes offer a more sophisticated approach by dividing the playable area into a network of interconnected nodes, allowing for more fluid and realistic movement patterns. However, both methods have limitations, particularly in handling unexpected obstacles and adapting to rapidly changing surroundings. To add to this, it is often computationally expensive to account for these changing factors.

To address these challenges, this paper delves into the application of machine learning techniques, specifically reinforcement learning and imitation learning, to develop more adaptive navigational systems. Reinforcement learning enhances NPC autonomy by using algorithms that learn from the environment through trial and error, improving their decision-making processes over time. Imitation learning, on the other hand, involves training NPCs to mimic human-like decisions based on demonstrated behaviours, thereby facilitating more natural and intuitive NPC movement.

The aim is to get the agent to be capable of moving with a complex range of movements through a complex environment that will contain dynamically changing obstacles and generally requiring a wider range of adaptation abilities compared to more traditional models. The model is rewarded and punished based on its ability to adapt and reach a goal as well as the speed and efficiency of reaching their objectives.

While this is applied to a video game environment, the concepts of this can have real world implications for autonomous navigation such as in drones, robots, and autonomous vehicles to say a few. It also serves as an example of how more traditional AI and ML systems can be integrated in new and novel ways and the potential of game environments to assist in solving real-world problems.

# Glossary

*Unity* – Unity is a game engine.

*Unity ML-Agents* – Solution released by Unity for using Machine Learning in the engine. Uses a mix of C# and Python. Uses PyTorch underneath.

*PyTorch* – Python machine learning framework.

# 1 Introduction

Systems of navigation in video games are vital, as they influence how a character or entity traverses the often-diverse environments of modern video games. This paper delves into how navigation has been developed within three-dimensional environments, focusing on navigation of non-player characters (NPC) who are devoid of direct human intervention. This paper explores the application of Machine Learning (ML) and, more specifically, Reinforcement Learning (RL) in navigating the NPC in these complex environments. The application of these technologies, especially in three-dimensional game environments, can be used to emulate real-life applications and have potential applications to areas such as autonomous robots, drones, and vehicles.

Traditionally, video game navigation had primarily relied upon techniques such as navigation meshes and waypoints. However, due to technological advancements, machine learning and the general explosion of artificial intelligence technologies, has opened the door for new, more diverse, and dynamic navigation methodologies. These can enhance the adaptability and intelligence of in-game systems so that they can move in ways that required a lot more human intervention before. This paper aims to discuss and explore these traditional navigation methods while evaluating the potential of artificial intelligence to revolutionise these techniques.

In the following sections, various navigation techniques will be explored followed by a discussion on traditional pathfinding algorithms. We will also delve into how machine learning, particularly through reinforcement and imitation learning, can be integrated into these systems to facilitate more complex and realistic navigational behaviours. Then, an implementation will be carries out using the Unity game engine, their Unity ML-Agents package and PyTorch. By comparing these methodologies, the study seeks to outline potential advancements and the evolving landscape of navigational technologies in digital environments.

# 2 Navigation

## 2.1 What is Navigation?

In a video game, navigation is deciding how to move from one place to another. (Ubisoft Forge. 2021) In the context of this study, this applies to moving a character from a game without influence from a player. However, it is also applicable to adjacent fields such as robotics and applied to scenarios such as autonomous drone or vehicle movement. Traditionally, in games, this is done through methods such as the navigation mesh or waypoints, and applying machine learning is a relatively new and emerging approach.

## 2.2 Navigation Methodologies in 3D Environments

### 2.2.1 Waypoint Method

The first, and most basic, method of navigation is using waypoints. It is still used in some games to this day for characters which do not need a more complex range of movement. It consists of having a point in space, and the position of the character is changed to go towards it, moving directly to the point. There are navigation markers that direct the algorithm to move in a particular direction of shortest path. (Lawande et al., 2022) The most obvious downside of this is that it doesn’t account for any hazards or objects which are in the way. This can be accounted for and corrected but they require more intervention which are not included in the navigation. Overall, this approach is mostly unused as all characters follow a set, identical path. They may however be potentially useful in future machine learning tasks as waypoints may be useful as markers for the AI to more roughly move towards.

### 2.2.2 Navigation Mesh Method

The second, and most employed, method is the navigation mesh. This converts the surface of objects, known as a mesh, into a set of convex regions, which can be trivially navigated within. (Ubisoft Forge. 2021) This is known as the navigation mesh. A pathfinding algorithm is then used to find the best path through the mesh from the characters current location to the target location. Optionally, extra post-processing techniques may be applied such as smoothing of the path.

This method is more preferred to the waypoint system as modern navigation meshes have many extra functions that waypoints do not. Due to the nature of meshes being the surface, it will automatically account for things such as slopes within the path. It can exclude polygons which have hazards within them which makes the character avoid them. It can also define a cost to certain paths which will encourage the character to travel along the path of least cost. For example, a character is moving from point A to point B. They can choose to use the road which has a cost of 1 per polygon, or they can walk across a field which has a cost of 2 per polygon. The character will choose the path with a lower cost. They will generally use the road but if the field is short, it may be more beneficial to move across that.

Navigation meshes are not perfect however and suffer from some big downsides. Some of these can be mitigated with some extra intervention.

The most glaring downfall of navigation meshes are that they are computationally expensive and must be “baked” prior to the game running. While real-time baking of the navigation mesh is possible it is expensive and is something that is generally avoided.

Another downside is it doesn’t react well to dynamic scenarios, such as a character using a jet pack to get to a ledge that it usually cannot or evading a falling boulder. A method called navigation mesh links were developed but they need to be placed and the mesh baked, again making it an undesirable thing to do at runtime. (Ubisoft Forge. 2021) This may be applied to the first example. This leaves playing the game feeling very static as it can only move within that link spot.

In the case of the falling boulder, it can easily calculate an evasion course for the boulder at that moment but unless some form of prediction is used it may evade the current position of the boulder without regard for where it will be in the future. It could check the position of the boulder every frame and then calculate a new path, but this is expensive again and the character wouldn’t have time to make much progress. Overall, when it is possible for the agent to use mobility action, such as jumping, the high number of places it can reach from each location dramatically increases the connectivity of the navigation mesh. Therefore, it increases its memory cost as well as the runtime cost of the pathfinding algorithms, to the point where using a navigation mesh ends up being too costly. (Ubisoft Forge. 2021)

## 2.3 Pathfinding Algorithms

Traditional pathfinding algorithms can be split into two main categories: undirected and directed algorithms. Undirected algorithms are akin to moving wildly. The algorithm does not plan how to get to its destination, only that it knows that it will eventually get there if it tries for long enough. In the inverse, directed algorithms use some form of method to assess where exactly they are in relation to the target point. Generally, this is done by assessing their current distance to the target, which will in turn find the shortest path to a target. This section will only cover some of those most common algorithms as there are countless algorithms, and many have extensions which enhance their usage. (Rafiq, Asmawaty Abdul Kadir and Normaziah Ihsan, 2020)

### 2.3.1 Undirected Algorithms

Undirected algorithms consist of two main algorithms. These are breadth-first search and depth-first search.

Breadth-first search searches every node connected to the node it is at. It then searches all the nodes connected to those, and so on until a solution is reached. (Lawande et al., 2022) This is evidently expensive as it is searching in every direction continuously. It will however do what is needed and will find the shortest path.

Depth-first search takes the opposite approach and searches starting from the furthest away nodes. If it does not find the target in these nodes it will come down a level to the next set of nodes closest to the origin and so on until it finds the target. (Lawande et al., 2022) While it will generally find the target in a much smaller target area, this has similar downsides to breadth-first search being that it searches a much larger area than needed as it is not focused on a direction.

### 2.2.2 Directed Algorithms

Directed algorithms in the context of games development largely consist of two main algorithms. These are the Dijkstra and A\* pathfinding algorithms. The thing that makes them different from undirected algorithms is that they have some way to assess their progress to the target. This is known as assessing the cost to the next node. Each node in this case would be something such as a way point, or a polygon on a navigation mesh.

Dijkstra algorithm is used by always choosing the next node that gives the least cost. (Lawande et al., 2022) This will always find a solution, but it may not be the optimal or shortest path. It is also more inefficient. Dijkstra is not used as much anymore but it was the popular solution prior to A\* being adopted as a standard.

A\* uses the approach of Dijkstra, but it also makes use of estimation to estimate the cost of a path. This means that it will combine the information of each and will attempt to determine the best path. (Candra, Budiman and Pohan, 2021) It will also backtrack if it feels a different path may work out to be less expensive. This works much quicker and gives a much shorter path. A\* is often extended into more specific algorithms for a specific game’s need such as HA\* and D\*. (Rafiq, Asmawaty Abdul Kadir and Normaziah Ihsan, 2020) It is also applied with other algorithms to perform a similar purpose. For example, it can be combined with an algorithm such as DPA to have larger control of obstacles. (Kapi, 2020)

# 3 Machine Learning

## 3.1 What is Machine Learning?

Machine Learning is a set of algorithms which can ingest data and can output information, or tasks based on these inputs. (Alonso et al., 2021)This can be used to greatly extend existing functionality in video games. In the context of navigation in video games, it can create a much more dynamic and responsive navigation system. This can allow for extended functionalities not applicable to most traditional pathfinding algorithms without extension such as variable pathfinding. This means for example that it can go through a door and take a shorter path if it possesses a key, or else will take a longer route if not. It will also allow for better adaptation of dynamic situations which is a major downfall of traditional pathfinding.

## 3.2 Reinforcement Learning

Reinforcement learning is a form of machine learning that functions by training the AI to display desirable behaviour. This AI then passes down the obtained information to the next episode and will always attempt to emulate the behaviour that will give it the most desirable outcome and the biggest reward.(Alonso et al., 2021)

### 3.2.1 Exploration

The basis of the algorithm that governs many reinforcement learning algorithms works by a system known as exploration. At its core, exploration is where the AI will input random value and attempt to find out what it will achieve. It will remember the series of inputs and will remember if it yields a positive result, a negative result or will lead to the end of the episode without any rewards.(Varghese and Mahmoud, 2021)

By combing all these experiences of random inputs, it will eventually determine which of these will yield the most positive rewards most consistently, thus creating desirable behaviour. Exploration is a vital part of training as it allows the AI to try different approaches to issues to determine the most appropriate one. These appropriate approaches are then saved in the policy.

### 3.2.2 Policy

The policy is what determines what the AI will do in a scenario. This means that it will map observations, or inputs, into a dictionary of observations to actions. The observations are information obtained from inputs such as what the camera can see or number values for touch from structures such as ray casts. (Alonso et al., 2021)In the inverse, actions are what the AI has determined in the best thing to do when the observations match.

For example, an input may be its current position and rotation in three-dimensional space. These are seven floating point values, three for position and four for rotation. It may also get the position of where it needs to go which is another three floating point values. The ten floating point values will be the observations. From that, it may determine that it needs to rotate the direction it is facing by a certain amount. It may also determine that it needs to move in a specific direction to make their current position as close as possible to the target position. This determination is done through training and the policy will determine that this is the best course of action to be positively rewarded.

### 3.2.3 Rewards and Punishments

Desirable behaviour is obtained by performing desirable actions, which they are rewarded for, and for punish undesirable actions. For example, in a racing game there may be a track with various checkpoints which, if hit, will yield a reward in the form of a positive integer or floating-point value. However, if the car hits the wall on the edge of the track, it will be given a punishment in the form of a negative value. It may also be given a reward by staying as close as possible to the centre of the track. Finally, it may also be given a punishment if it stays motionless for too long. This final punishment is because the AI may attempt to stay motionless to avoid punishments. Through incentives such as these, a functioning racing AI will exist at the end of this and will consistently move to each checkpoint and remain on the track.

Rewards, both positive and negative, need to be given regularly as the AI may struggle or fail to perform specific actions if they are too sparse. In the car example, if the checkpoints are too few and far between the AI may struggle to move towards it, especially if the track is not a straight line.

Rewards also scale, but only with other given rewards. Sticking to the car example, if the positive reward given for hitting a checkpoint is 100, but the negative reward for crashing into the wall is -1, the reward is so miniscule in comparison to the reward that it may be negligible to the AI. Large rewards are appropriate for reaching the final objective, but, for a smaller objective like the checkpoint, a smaller reward such as 5 may be a better scale. The numbers in the example are arbitrary and rewards and punishments could be decimal numbers, or numbers in the thousands or millions. The only thing that matters is the values of the other rewards in comparison. (Fang et al., 2022)

## 3.3 Imitation Learning

Imitation learning is a technique utilised to teach the AI how to perform an action through demonstration. (Zheng et al., 2021) This is by performing the actions that the developer wants the AI to learn. For example, in Unity’s ML-Agents package a developer can “record” themselves doing what they wish the AI to imitate. They can define the information that they record. This can then be input to the AI which will attempt to recreate it. The policy will then be determined through it attempting to recreate the actions of the input instead of a system of rewards and punishments.

Some imitation learning algorithms learn for different purposes. The imitation learning algorithm Behavioural Cloning will attempt to exactly recreate actions which can be useful for a scenario that little to nothing will change, and it will consistently carry out the same actions. In the inverse, an algorithm such as GAIL (Generative Adversarial Imitation Learning). This approach lets the AI make its own choices and assigns a reward based on how similar it is to the input recording. This will encourage the AI to maximise the reward similarly to reinforcement learning. This is known as inverse reinforcement learning.(Zheng et al., 2021) This can be useful in more flexible situations where the AI should not carry out exact same actions each time.

### 3.3.1 Usage with Reinforcement Learning

Imitation learning can be utilised along with reinforcement learning and, in certain cases, can produce a more appropriate result than using either technique alone. Imitation learning could be used initially to train the model to skip the beginning steps of reinforcement learning. It could be taught the basic actions and then once they have learned those through imitation learning, reinforcement learning could be used to train the existing policy to carry out tasks in a more fluid way as it would no longer be dependent on a recording. (Varghese and Mahmoud, 2021)However, this could be easily abused or overtrained in imitation learning as well.

# 4 Methodology

Reinforcement learning will be utilised in the approach to create a model which is capable of navigation. This was determined due to the nature of wanting the model not being tied to a specific environment and thus imitation learning is inappropriate for this use case. This involves:

* Creating various environments for the model to train in.
* Defining inputs to the model.
* Training and rewarding the model in this environment.

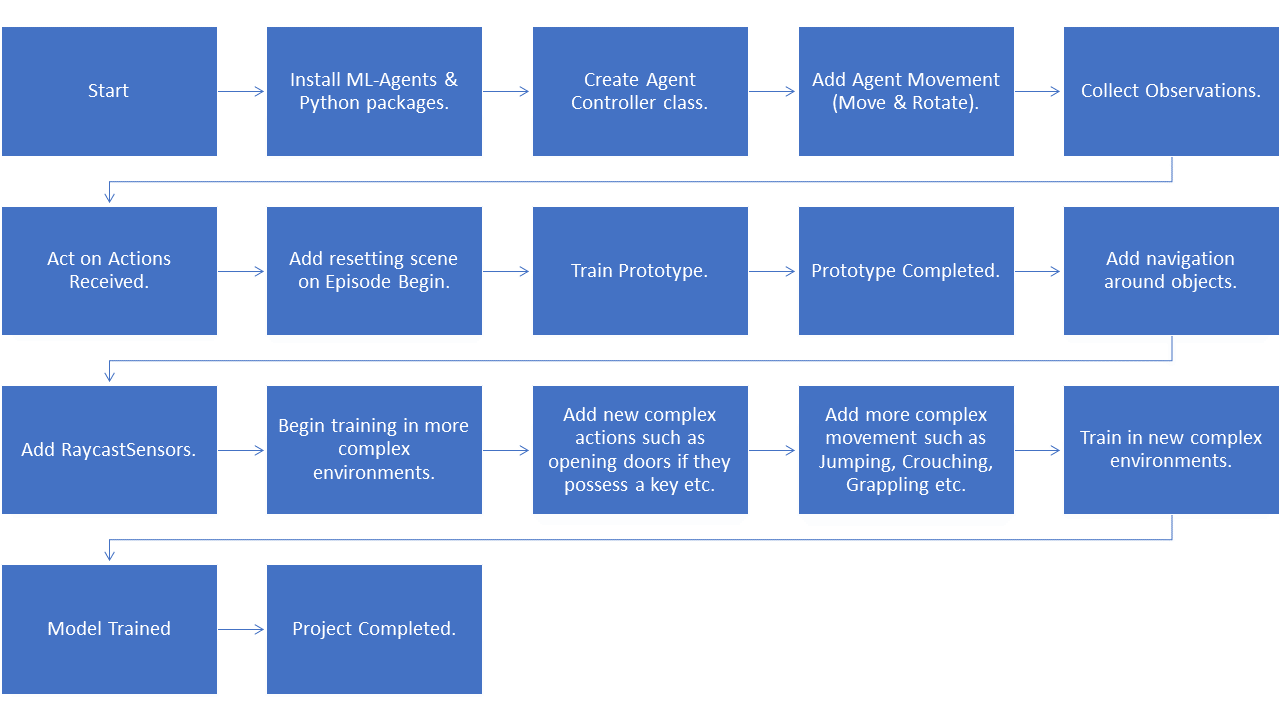
## 4.1 Technologies Used

The technology primarily used is a package provided within the Unity Engine called ML-Agents which utilises Pytorch, a commonly used machine learning tool, to train models. The Unity Engine then provides the tools to run the model within their game or environment through this package. This package is an extension of a previously created package within Unity called Barracuda which has wider usages for neural networks and artificial intelligence, while ML-Agents is built specifically for machine learning.

## 4.2 Functional Specifications

|  |
| --- |
| **Must Have** |
| Suitable Unity environment with all necessary packages installed. |
| Suitable Python environment with all necessary packages installed. |
| Method which provides the necessary data inputs (observations) to the agent. |
| Method to take model outputs and translates it into actions. |
| **Should Have** |
| Defined Heuristic method to allow the developer to test outside of training. |
| Method which determines if the agent is facing the correct direction. |
| Method that determines that the agent has either hit their target or has hit a wall. |
| Method that gets additional inputs such as Raycast sensors and adds them to the observations. |
| **Could Have** |
| Method that will handle the resetting of the scene when each episode begins. |
| Methods that provide visual feedback during the training process to visually gauge that the agent is acting as expected. |
| **Would Have** |
| Method that will return feedback of data on the beginning of the episode. |
| Methods that will run a NavMeshAgent in the same environment and will compare the time of arrival to the reinforcement learning model. |

## 4.3 Implementation Plan



*Figure 1*

## 4.4 Class Diagram

A screenshot of a computer program

Description automatically generated

*Figure 2*

## 4.5 Prototype

For the prototype a basic model which will orient itself and will move towards the target will be created. The aim of this prototype is to show that navigation is possible and that it is extendable in the future.

### 4.5.1 Tasks

|  |  |
| --- | --- |
| **Details** | **Status** |
| Create a Unity Project. | Completed. |
| Download & Install the ML-Agents Unity Package. | Completed. |
| Install Python. | Completed. |
| Download & Install Pytorch. | Completed. |
| Download & Install the ML-Agents Python package and its dependencies. | Completed. |
| Create the 3D environment (ground and walls). | Completed. |
| Create the agent and the target. | Completed. |
| Create the Agent Controller script. | Completed. |
| Create basic movement & rotation. | Completed. |
| Create Heuristic inputs. | Completed. |
| Collect observations (the agent’s position, agent’s rotation, targets position, if the agent is looking at the target) | Completed. |
| Handle received actions (outputs of the training) and call the Move() and Rotate() methods with the outputs. | Completed. |
| Create agent & targets position and rotation resetting on episode begin. | Completed. |
| Create logic to validate if the agent is looking at the target. | Completed. |
| Create visual feedback in response to agent actions. | Completed. |
| Train the agent. | Completed. |

### 4.5.2 Process

The process began by doing the installation of Unity’s ML-Agents package in both the Unity Engine and Python. There was no issue with installation in the Unity Engine, but the Python installation presented some issues. ML-Agents had not been updated since 2022 at the time of installation and so was lacking support for newer python versions. It only supported Python versions 3.8 and 3.9 so version 3.8 was installed to remedy this.

Next Pytorch and the ML-Agents packages were installed mostly without issue, but I got an issue where a specific version of Protobuf must be installed separately. There was also an issue where the Onnx package had to be installed separately. This package is necessary as onnx is the file format that the trained model is saved in and, if forgotten, training would have been lost.

Next was the creation of the environment in Unity. It was a simple environment for the prototype. It consisted of a ground and four walls. The agent and the target were also placed down. The agent was just a capsule, and the target was a sphere.

A video game of a game

Description automatically generated

*Figure 3*

A C# script called AgentController was then created which implemented methods for basic movement. The agent is constrained to only move forwards and backwards and can rotate right to left.

It also implemented a class from called Agent from ML-Agents and certain override methods were implemented which allowed the class to give data inputs to the model. It also collected outputs from the model and moved and rotated the agent accordingly. It also handles giving rewards and punishments to the agent based on the actions taken during training. Finally, it also handled randomly positioning both the agent and the target each time an episode began as well as randomly rotated the agent.

It can take in data as numbers and Unity allows for it to take in any data structure which can be broken down into numbers such as the Vector3 structure which is three floating point numbers. It was given the position in three-dimensional space of the agent and the target which are both Vector3 structures, it was given the agents rotation which is a Quaternion which is four numbers and finally it was given a Boolean value which would translate into a binary zero or one value.

It was then ready to train, and it worked for a small while but then it continually stopped as soon as it started. This was due to the default configuration for training in ML-Agents. However, a custom configuration can be passed to the python package when calling the python command. It turns out that only fifty thousand steps can be done as the limit in the default configuration, however this was resolved by setting the limit much higher in the custom configuration.



*Figure 4*

A screenshot of a computer program

Description automatically generated

*Figure 5*

It was left train for a long time and when seeing how it was doing, an odd behaviour was discovered. If it began by facing away from the target, it moved backwards towards the target. This was undesirable behaviour, so a reward was added to the method which handles validating if the agent touched either a wall or the target and now gives a larger reward if the agent was facing the target when it hit it.



*Figure 6*

This prototype was then considered finished for the purposes of what was needed. To go on from here the next step is to get it navigating around obstacles and to get it to utilise Raycast sensors. As of the conclusion of the prototype the following image is the results of how large a reward is being obtained and how long the episode is taking. As can be seen, the results began to plateau and thus something new may need to be introduced.

A screenshot of a graph

Description automatically generated

*Figure 7*

# 5 Implementation

## 5.1 Sprint 1

### 5.1.1 Tasks

|  |  |  |
| --- | --- | --- |
| Task No. | Description | Status |
| 1 | Remedy the issue presented while training the prototype where it goes backwards like a car before moving to the target. | Completed |
| 2 | Clean up the code and separate responsibilities of scripts. | Completed |
| 3 | Expand current basic movement to include left/right movement. | Completed |
| 4 | Define all future observations and actions of the agent. | Completed |

### 5.1.2 Task 1

During the completion of the prototype, a defect occurred whenever the rotation of the agent had it facing roughly away from the target. In this occasion, the agent would move backwards similarly to that of a car backing out from a parking spot before moving more normally towards the target. This was likely a consequence of the remediation of a different defect which gave the agent a larger reward for facing the target to encourage it to face the target when moving towards it. Due to this, it developed this behaviour to orient itself this way.

To resolve this advice was given that to resolve this the agent may need to be encouraged to move quicker. To this end, the Unity ML-Agents examples were looked at and it was discovered that Unity had encountered a similar issue and remedied it by giving a miniscule negative reward every time OnActionReceived is called to encourage the agent to move as quickly as possible. Equipped with this knowledge, a similar formula was utilised here. (Unity, 2020)



*Figure 8*

### 5.1.2 Task 2

For the purposes of the prototype all logic was handled in the AgentController class. This is not good practice and so needs to be remedied. This remediation began by adding an interface to the WallScript and GoalScript classes. These classes initially were just present to check their types but now implement an interface called ITriggerableObject which contains a method Trigger. This method now contains the logic for triggering.

Next, a lot of magic numbers were present in the AgentController script. This is again not good practice, so all constant numbers were moved to a new class simply called Constants which contained all these numbers for use where needed.

Finally, the visual aid utility methods such as OnDrawGizmos and AddTexture have been moved to the EnvironmentManager which handles the environment of the agent. Later, this class will have greater use generating obstacles and so on but for now it only handles the visual aspects of the training and removes that from the AgentController.

### 5.1.3 Task 3

In the prototype, the agent could only move forward and backwards. While it adjusted its rotation to move left and right. However, for primary implementation it was decided that the agent would not be as limited, and it would also obtain the ability to move left and right.

To do this, it was a simple as increasing the number of continuous actions from two to three and adjusting the move calculation to factor this movement.

The intention was then to continue from the previous point of training with this new movement added but an unfortunate discovery was made. It was unable to continue from the previous point as the number of continuous actions must remain the same to continue from where a model finished. This directly led to the Task 4 as it was imperative that no move progress was lost each time a new action or observation was added.

For this task, an identical scene was made with extended movement and tested. It took the model much longer to learn to perform its task of moving to the target, but it was more consistently accurate once this was accomplished. This would indicate that as new actions are added training time will increase but will potentially pay off more in the long run.

A graph on a screen

Description automatically generated

*Figure 9*

### 5.1.4 Task 4

This task was moved in quickly due to the pressing issue outlines in Task 3 of when new actions or observations are added to an existing model it must be retrained. Due to this, a total of 101 observations were defined, 3 continuous actions and 5 discrete actions. Note that each observation and continuous action is a floating-point value.

The intention was always to utilise the RayPerceptionSensor3D sensor class to allow the model to “see” around the scene. They shoot seven rays out from the centre of the body in 50 degree increments. There are three RayPerceptionSensor3D classes; one at the feet, one at the centre of the agent, or origin or the transform.position, and finally one at the head of the agent. These had to be implemented now to accurately determine the number of observations.

The observations are Agent Position (3), Agent Rotation (4), Target Position (3), IsLookingAtTarget (1), IsJumping (1), IsCrouching (1), IsGrappling (1), Keys (3, allows for multiple key types if later applicable), RayPerceptionSensor3Ds (84 (7 rays \* 3 for position of each hit + 7 for a hit object type represented as a number \* 3 number of RayPerceptionSensor3Ds))

The continuous actions are Move (2 (forwards/backwards, left right)), Rotate (1).

The discrete actions are Jump, Crouch, Stand Up, Grapple, Use.

## 5.2 Sprint 2

### 5.2.1 Tasks

|  |  |  |
| --- | --- | --- |
| Task No. | Description | Status |
| 1 | Train model using the new observations. | Completed |
| 2 | Combine extended movement into base model. | Completed |

### 5.2.2 Task 1

This sprint was primarily focused on testing and learning about the new movement and observations and how they can be applied. The agent was trained within the existing environment with the new observations. In particular, the RayPerceptionSensors were invaluable moving around as it allowed the agent to determine what it could “see” using tags. The walls were all given the tag “Wall” and the target was given the tag “Target”. These tags could be input to the RayPerceptionSensors as things to observe for.

This proved valuable moving around the area. Firstly, it simply had a target somewhere around it in a room and it had to move to it, and then it had to move around a wall. However, moving around a wall wasn’t fully trained in this sprint and will be discussed in more detail in the next one. This was done in this task mostly as proof that the RayPerceptionSensors were valuable enough to maintain.

The other thing to note is that RayPerceptionSensors are expensive on the system and put the CPU under constant high loads. The size of agents running concurrently reduced from twenty-five agents to sixteen as the CPU couldn’t handle them. Even with this, there were occasions where load spiked to one hundred percent. This may indicate that this is not an appropriate solution for a production application.

### 5.2.3 Task 2

The second thing done was a simple implementation of the code produced in Task 3 of Sprint 1. The code was moved from a separate class which implemented the base class that the Agent was working with. This didn’t affect observation sizes, so it did not add or remove any observations, it simply moved the code that was present in the AgentControllerExtended class to the AgentController class and removed this now unneeded class.

## 5.3 Sprint 3

### 5.3.1 Tasks

|  |  |  |
| --- | --- | --- |
| Task No. | Description | Status |
| 1 | Train model to move around walls. | Completed |
| 2 | Expand upon this so that the wall changes position. | Completed |

### 5.3.2 Task 1

While it was touched upon in the previous sprint, the next objective was for the model to accurately be able to move around walls and to reach the target. To achieve this, new walls were put in the centre of the level. They were placed with a five-unit gap in between them to allow the agent adequate room to navigate around it.

It was able to learn this action remarkably well, with it inheriting its initial movement training from the previous iteration it was able to quickly learn that moving past the wall and looking around would usually yield a reward. It quickly rose in accuracy and was able to do it consistently after around four million steps. It hit a ceiling of around 0.92 which is as expected. Due to the code inserted in Sprint 1 to encourage the agent to keep moving it is constantly losing a small amount of a reward, so it reaching this amount was more than adequate.

### 5.3.3 Task 2

This is a continuation of the previous task. Now that the agent was able to move around walls, it was time to make the environment change more. The first adjustment made was a method called OnEpisodeBegin() was inserted into the EnvironmentManager class which was called whenever the method of the same name was called in the Agent class. This allowed for the environment to be regenerated whenever the episode began again. This allowed for the position of the walls to change. The opening would move left and right along the wall which meant that the agent had to search for it before going through it and reaching the target.

It developed a strategy of moving from right to left along the wall until it found an area where it was able to move through. However, this sometimes failed due to the limited range of the RayPerceptionSensors. To remedy this, the max range was increased to one thousand which was never going to be reached. This allowed the agent to always see where the wall was, and this strategy was more effective. This displayed a utilisation of strategy and development of a policy which showed the first signs of searching.

A graph of a graph

Description automatically generated

*Figure 10*

Above is the reward cycle for the model. The beginning stages is it learning from the initial learning that was inherited from previous models, then it began the quick rise as outlined in Task 1. Where it falls around six million steps, this is when this task was implemented. The slight dip is when it levelled out a bit and the solution of extending the right was introduced before it beginning to level out and move towards exploitation instead of exploration.

## 5.4 Sprint 4

### 5.4.1 Tasks

|  |  |  |
| --- | --- | --- |
| Task No. | Description | Status |
| 1 | Expand the training area. | Completed |
| 2 | Retrain the model. | Completed |
| 3 | Explore adjusting parameters | Completed |

### 5.4.2 Task 1

In this task, the area was expanded massively. This placed the agent in the centre of the area and providing it four rooms, one in each cardinal direction. The target could be in any of them, and the agent was made to search for them. The EnvironmentManager class handled changing wall positions and moving the target. However, this went very poorly and did not come out as expected. Colliders were added to rooms to attempt to incentivise exploration, but the agent wasn’t capable of handling it.

The agent just appeared to be getting confused. It would attempt to enter a room, find nothing and either usually either hit a wall which ended the episode, or it would get stuck in it. As it was able to find the target so rarely was it couldn’t achieve many rewards it eventually began spinning in circles and from there the model was not useful.

In hindsight, it was too much too quickly and due to this a sprint was wasted. When attempting to fix this, so many changed were made that the Git branch had to be rolled back as the scene and code was too different from what came before. It should have instead been in a smaller room and changing the area within it dynamically instead of dropping it in a much larger place.

### 5.4.3 Task 2

Due to an oversight and accidentally forgetting to commit the model of Sprint 3 before attempting to achieve Task 1 the model had to be retrained. The code had been committed but the completed model had not been. This allowed for the model to be retrained as it was before.

### 5.4.4 Task 3

In this task, an exploration of adjusting the training parameters was undertaken to attempt to make the previous task. This included adjusting the configuration for the agent and attempting to gain a better result. This meant adjusting value such as the number of epochs and the alpha value to try encouraging the agent to explore more. While it did work temporarily, the agent inexplicably reverted to how it previously was. When this occurred, increasing, or decreasing values unfortunately did nothing to fix it.

## 5.5 Sprint 5

### 5.5.1 Tasks

|  |  |  |
| --- | --- | --- |
| Task No. | Description | Status |
| 1 | Expand upon older training area. | Completed |

### 5.5.2 Task 1

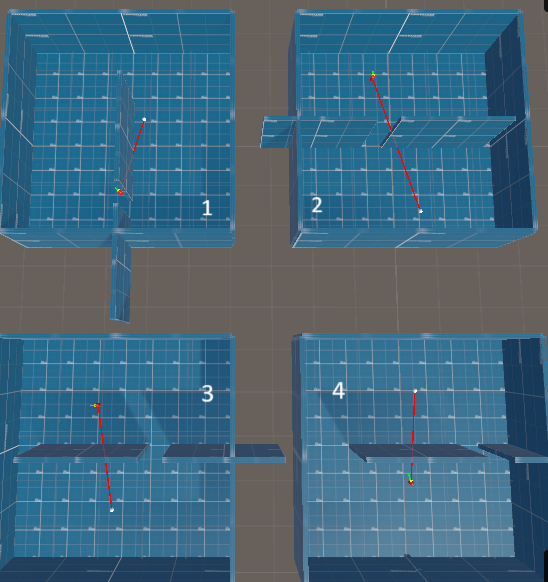
For this sprint, all the time was focused on expanding upon the area used in training prior to Sprint 4. Where it has ended was a room that is split by a wall with a gap in the middle, and that wall would move left or right at the start of each episode based on random chance. For this sprint, the objective was to expand upon this idea. Taking lessons from the previous sprint, this was done incrementally.

Firstly, the starting positions of the agent and the target were randomly swapped. The agent was placed in a random location around the starting position with a random location all the time, but now it can swap with the target. That means it can have a random location based on its starting position or the targets starting locations, and the target will do so based on the opposite. This was trained successfully; however, it took the agent longer than intended to learn this behaviour as it seems to have developed a bias regarding its rotation from the initial training.

Once this was completed, the next, and main, step was the change the directions even more. From the start, the largest faces of the wall that the agent would see faced a north-south direction. However, they now could face an East-West direction too. To achieve this, the OnEpisodeBegin method in the EnvironmentManager class received a new argument which was a Boolean denoting if this should occur. The agent would generate this and pass it to the EnvironmentManager to ensure that both were in tandem. The agent and the target would also receive appropriate new positions if this occurred.

This leads to the following logic:

* Boolean checking if the direction should be rotated.
* If true, agent and target start in an east-west position with a random position around the starting location.
* If false, agent and target start in a north-south position with a random position around the starting location.
* Boolean checking if the agent and target positions should be rotated. If yes, they will be rotated.
* Call EnvironmentManager. OnEpisodeBegin with argument that shows if direction should be rotated.
* If rotate direction is true, rotate direction to east/west. Also adjust the z value by a random range which moves the gap location.
* Else, rotate direction to north/south. Also adjust the x value by a random range which moves the gap location.



*Figure 11*

Notes: The red line is a line between the agent and the target. The green line is the direction the agent is facing. The agent does not see either of these. They are only for human accessibility.

The above image shows the random locations in action. In area 1, two gaps have been presented by the random movement for the agent to navigate through and to find the target. They are also facing an east/west direction. In the other three it has generated a north/south direction, but they begin in different sides, orientations, and positions.

This training would succeed but it would not achieve as high an accuracy as the previous models, likely as it had to search more regarding direction, orientation and so on. It would achieve an accuracy of around 0.86. The other thing that would become apparent though is that if trained for too long, the accuracy would drop dramatically as if it forgot how to perform the action. In addition, if it was left to run this way for long enough, it would start crashing with the error message “RuntimeError: probability tensor contains either inf, nan or element < 0” which, after research, indicates that something if either a null value or is dividing by zero. To resolve this problem, a rollback back to a previous commit would have to be done. To resolve this problem, a slightly older commit was loaded up and work was continued from there after remediating the differences.

Obstacles were also planned for this stage so that it would have to avoid obstacles while doing this, but due to lack of time and only one sprint remaining it was decided that at least one of the planned for complex actions would be implemented.

## 5.6 Sprint 6

### 5.6.1 Tasks

|  |  |  |
| --- | --- | --- |
| Task No. | Description | Status |
| 1 | Implement obtaining keys and opening doors | Unachieved |

### 5.6.2 Task 1

The final sprint was not achieved. The aim, as outlined in Sprint 5, was to have at least one complex action implemented. In this case, the aim was to have functionality for collecting keys and opening a door. The plan was to have keys instantiate near the agent to be picked up, the agent would then use a discrete action on the door to open it and the move to the target.

To begin with, a key was instantiated near the agent using similar logic to the logic that decides where an agent begins so that the key always spawns near the agent. For the beginning, this would simply be an object which would be created, the agent could observe, they would get a reward upon touching it, and it would be destroyed. However, an unforeseen issue occurred here.

The RayPerceptionSensor takes in a list of tags that it can “see”. What was not known is that each of these tags for every ray is an observation in and of itself. This means that when a key was added with the tag “key”, it had a larger observation size. This issue with observation sizes had been seen in Sprint 1, but since it wasn’t known that each tag is its own observation that would not work. Unfortunately, the only way to resolve it was to train the model again with the key and door tags added. Once this was done, a lot of time was already gone. To note, if adjustments were made to the ML-Agents Python code, it could be fixed without retraining it, but there was no time to go attempting this avenue. It was vital to do this or else the agent couldn’t detect the keys or the door.

Then another issue reared its head, which was the new model didn’t train as well as the previous one. The model worked fine up until keys were added, upon which it seemed to get confused and would rapidly drop in accuracy, with an average drop of 0.15 accuracy every time the model was saved which is every 50,000 steps. It would also stop learning and would spin in circles, but if the keys were removed again the model would work fine. It has been put down to some issue either with PyTorch or with how Unity handles the data as there isn’t anything different about keys that would prompt this.

# 6 Results

While the final sprint was unachieved and the scope didn’t extend as far as originally intended, it did complete all the initial requirements all the same. It satisfied all conditions set out in the functional specifications.

It also achieved the overall goals of being able to autonomously move around without any other assistance. It had no knowledge of where the target was located, or of the direction it was in. Since it started in a different position and orientation each time a new episode began, it always had to orient itself and search for the target. This was done reliant almost solely on how a human being navigates, with it relying mostly on sight using RayPerceptionSensors and knowing its own location. While it had other observational inputs, they were only things such as if they are crouching or if they are jumping.

It also achieved good accuracy for most tasks, with the ultimate model achieving an accuracy of 0.86 out of a total of 1. This was a great result as, considering it lost reward every step to encourage consistent movement, it meant that despite the obstacles imposed on it, it achieved the task in very quick timing.

A graph with lines on it

Description automatically generated

*Figure 12*

While it is easy to blame the software, in the case, it is partially to blame. ML-Agents lacks a lot of the features that PyTorch offers that would have given the tools to fix some of the issues faced. For instance, there are tools present in PyTorch that allow for models to still be runnable with different observation sizes. This is just one example, but in multiple cases an error of some sort has been present that is possible to resolve, but it requires editing the ML-Agents source code. While it is likely as it is for a reason, it is nonetheless disappointing. As touched upon in Sprint 6, the agent appeared to just “forget” previous training when keys were added without is adding a tangible difference to the scene. The model was trained with the key observation, but it just didn’t work when it saw a key which was very confusing.

Other avenues which should have been explored more thoroughly earlier were tuning parameters more thoroughly and exploring using state models. For parameters, they were only explored in any detail in Sprint 4, however they were looked at more casually from as early as Sprint 2. However, a failure was to adequately capitalise on the use of them.

Another area that should have been explored is state models. In certain Unity ML-Agents examples, they used models in a state machine-like pattern. This allowed them to train a model to be very good at a specific task and only that task. They then had logic to determine when to use each model and it allowed them to have an accurate model. This may have been an avenue to explore. It primarily wasn’t explored earlier as the aim was to have a navigation system and I felt that having it as a state machine strayed a bit from that.

Finally, curriculum learning should have been explored. Since it is incremental in nature, the agent may have been able to overcome some of the present challenges.

# 7 Conclusion

In this paper, the evolution of navigation systems was explored within the scope of video games, focusing on the application of machine learning techniques to enhance the navigation capabilities of non-player characters (NPCs) in three-dimensional environments. The paper highlights the limitations of traditional navigation methods, such as waypoints and navigation mesh, which are often rigid and fail to adapt to dynamic changes within the environment. By integrating reinforcement learning it’s been demonstrated that NPCs can achieve a higher degree of autonomy and responsiveness, effectively navigating through complex and dynamically changing environments.

This paper showcases that machine learning can not only enhance the realism of NPC behaviours but also contribute to the efficiency of game design. The Unity ML-Agents package and PyTorch have proven to be valuable tools in training these sophisticated models, offering a new realm of possibilities for game developers.

The implications and applications of the paper extends beyond video games. The principles and techniques discussed in this paper have potential applications in real-world scenarios, such as robotics and autonomous vehicle navigation, where adaptability and responsiveness to environmental changes are crucial.

Ultimately, this research contributes to a deeper understanding of how machine learning can revolutionize navigation systems within video games and potentially influence other technology-driven fields. As game technologies evolve, the continued integration of AI will play a critical role in shaping the future of interactive entertainment and autonomous systems.

# 8 References

Alonso, E., Peter, M., Goumard, D., Ubisoft, J.R. and Forge, L., 2021. Deep Reinforcement Learning for Navigation in AAA Video Games. [online] [*https://arxiv.org/abs/2011.04764*](https://arxiv.org/abs/2011.04764)[Accessed 16 October 2023]

Beeching, E., Peter, M., Marcotte, P., Debangoye, J., Simonin, O., Romoff, J. and Wolf, C., 2021. Graph augmented Deep Reinforcement Learning in the GameRLand3D environment. [online] [*https://arxiv.org/abs/2112.11731*](https://arxiv.org/abs/2112.11731)[Accessed 18 October 2023]

Candra, A., Budiman, M.A. and Pohan, R.I., 2021. Application of A-Star Algorithm on Pathfinding Game. In: *Journal of Physics: Conference Series*. IOP Publishing Ltd. [*https://doi.org/10.1088/1742-6596/1898/1/012047*](https://doi.org/10.1088/1742-6596/1898/1/012047). [Accessed 18 October 2023]

Fang, Q., Xu, X., Wang, X. and Zeng, Y., 2022. Target-driven visual navigation in indoor scenes using reinforcement learning and imitation learning. *CAAI Trans. Intell. Technol*, [online] 7, pp.167–176. [*https://doi.org/10.1049/cit2.12043*](https://doi.org/10.1049/cit2.12043). [Accessed 28 October 2023]

Kapi, A.Y., 2020. A Review on Informed Search Algorithms for Video Games Pathfinding. *International Journal of Advanced Trends in Computer Science and Engineering*, 9(3), pp.2756–2764. [*https://doi.org/10.30534/ijatcse/2020/42932020*](https://doi.org/10.30534/ijatcse/2020/42932020). [Accessed 18 October 2023]

Lawande, S.R., Jasmine, G., Anbarasi, J. and Izhar, L.I., 2022. A Systematic Review and Analysis of Intelligence-Based Pathfinding Algorithms in the Field of Video Games. *Applied Sciences (Switzerland)*, 12(11). [*https://doi.org/10.3390/app12115499*](https://doi.org/10.3390/app12115499). [Accessed 18 October 2023]

Rafiq, A., Asmawaty Abdul Kadir, T. and Normaziah Ihsan, S., 2020. Pathfinding Algorithms in Game Development. In: *IOP Conference Series: Materials Science and Engineering*. Institute of Physics Publishing.[*https://doi.org/10.1088/1757-899X/769/1/012021*](https://doi.org/10.1088/1757-899X/769/1/012021). [Accessed 20 October 2023]

Varghese, N.V. and Mahmoud, Q.H., 2021. A Hybrid Multi-Task Learning Approach for Optimizing Deep Reinforcement Learning Agents. *IEEE Access*, 9, pp.44681–44703. [*https://doi.org/10.1109/ACCESS.2021.3065710*](https://doi.org/10.1109/ACCESS.2021.3065710)*.* [Accessed 21 October 2023]

Zheng, B., Verma, S., Zhou, J., Tsang, I.W. and Chen, F., 2021. Imitation Learning: Progress, Taxonomies and Challenges. *IEEE Transactions on Neural Networks and Learning Systems*, [online] p.2022. [*https://doi.org/10.1109/TNNLS.2022.3213246*](https://doi.org/10.1109/TNNLS.2022.3213246)*.* [Accessed 21 October 2023]

Unity, 2020. *PushAgentBasic Unity ML-Agents Example*. [computer program] Available at: <*https://github.com/Unity-Technologies/ml-agents/blob/release\_13/Project/Assets/ML-Agents/Examples/PushBlock/Scripts/PushAgentBasic.cs#L173-L174*> [Accessed 20 November 2023].