Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

Maze Game

Project Engineering

Year 4

Lokesh Panti

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Atlantic Technological University

2022/2023

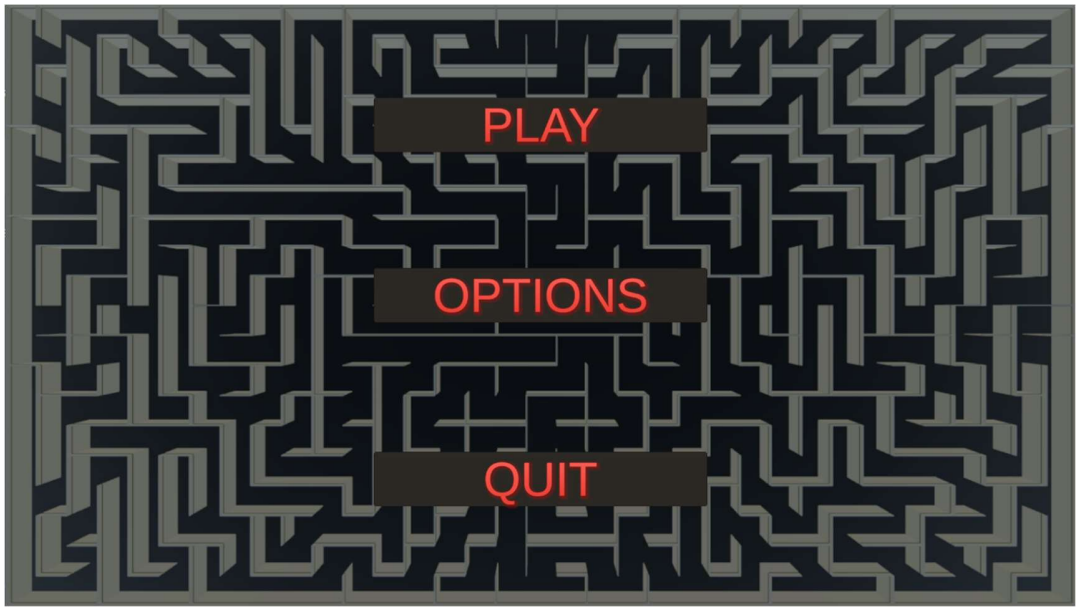

Figure 1

Main Menu screen to initiate the game.

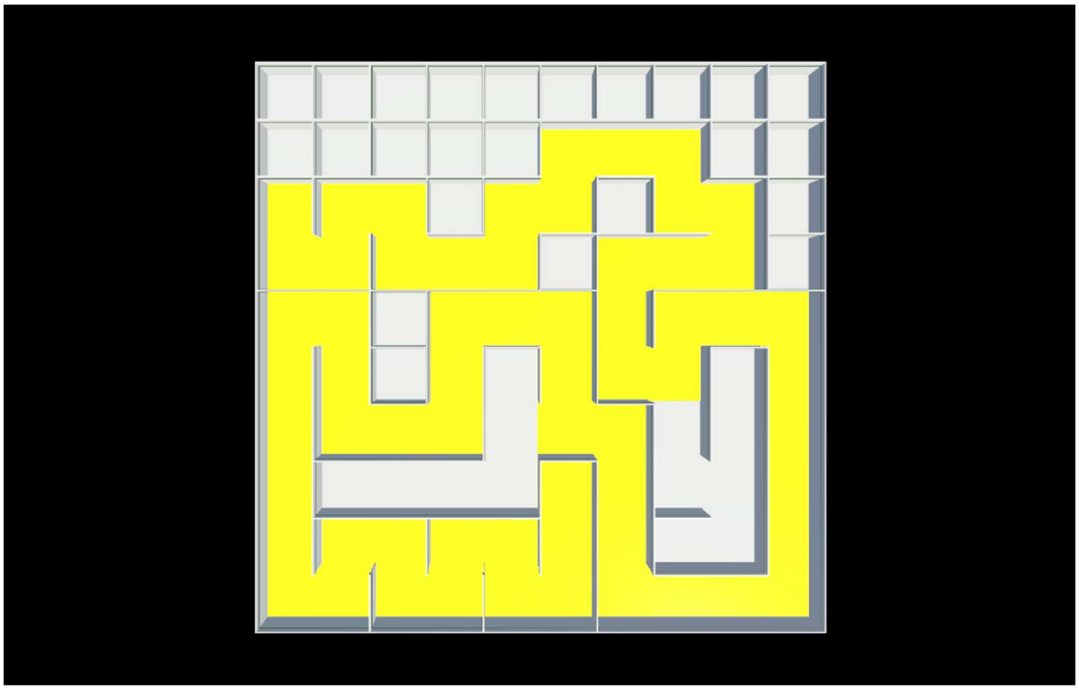

Figure 2

Random Maze generating at the beginning of the game.

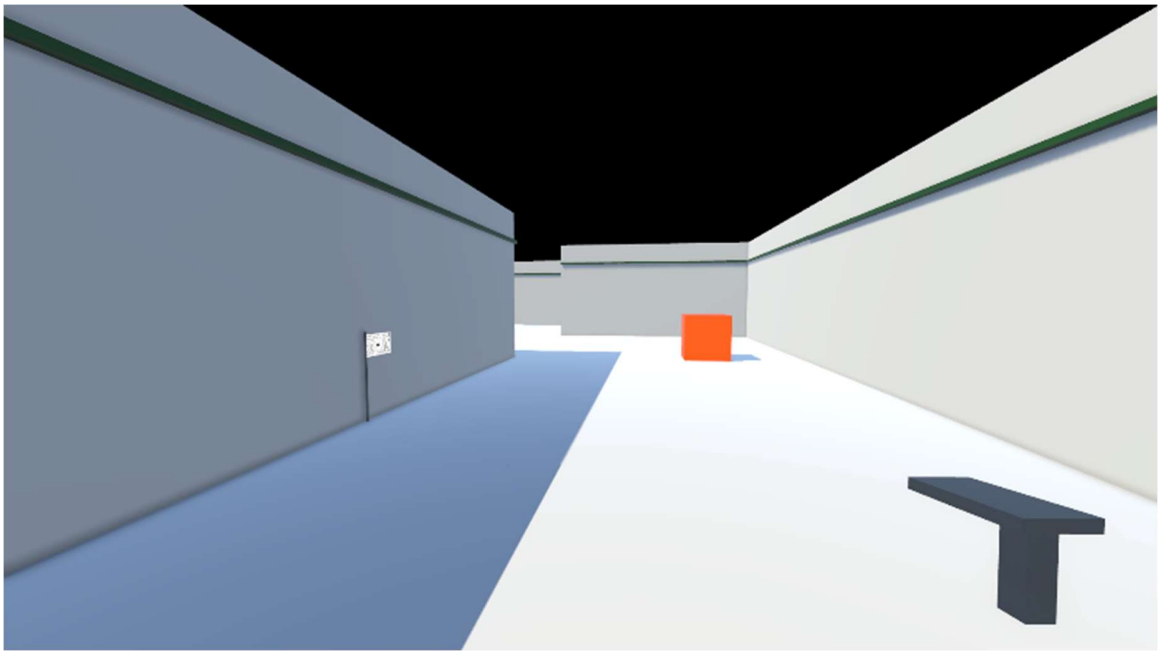

Figure 3

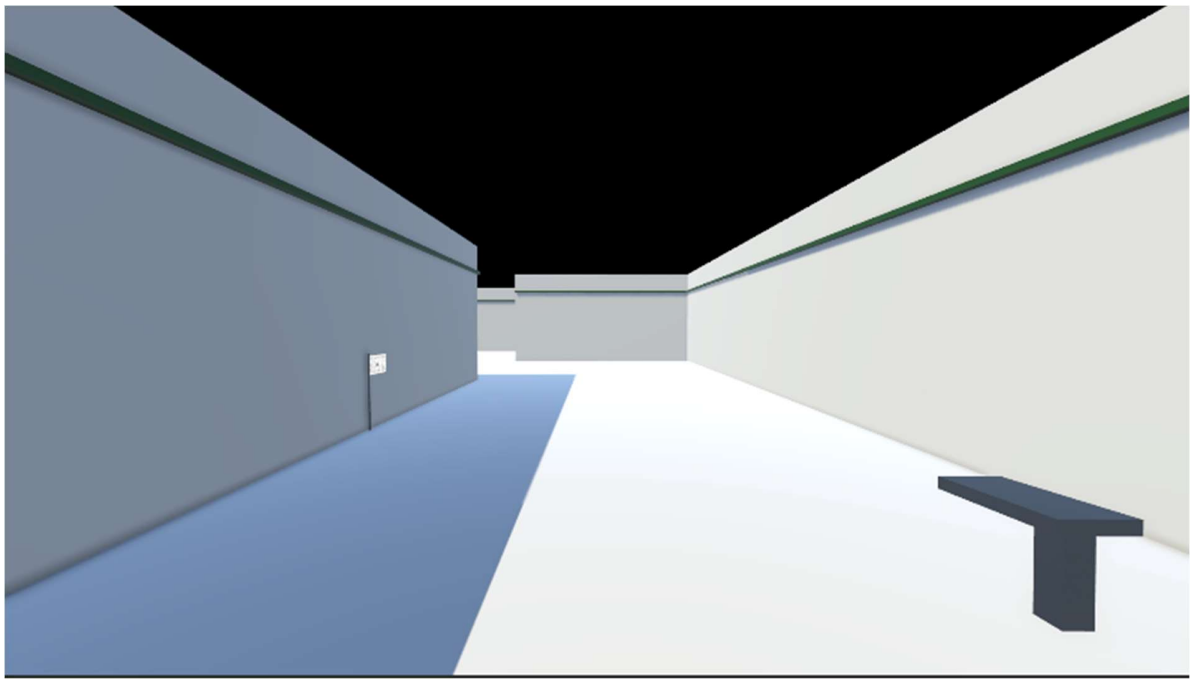

Figure 4

After Shooting the agent.

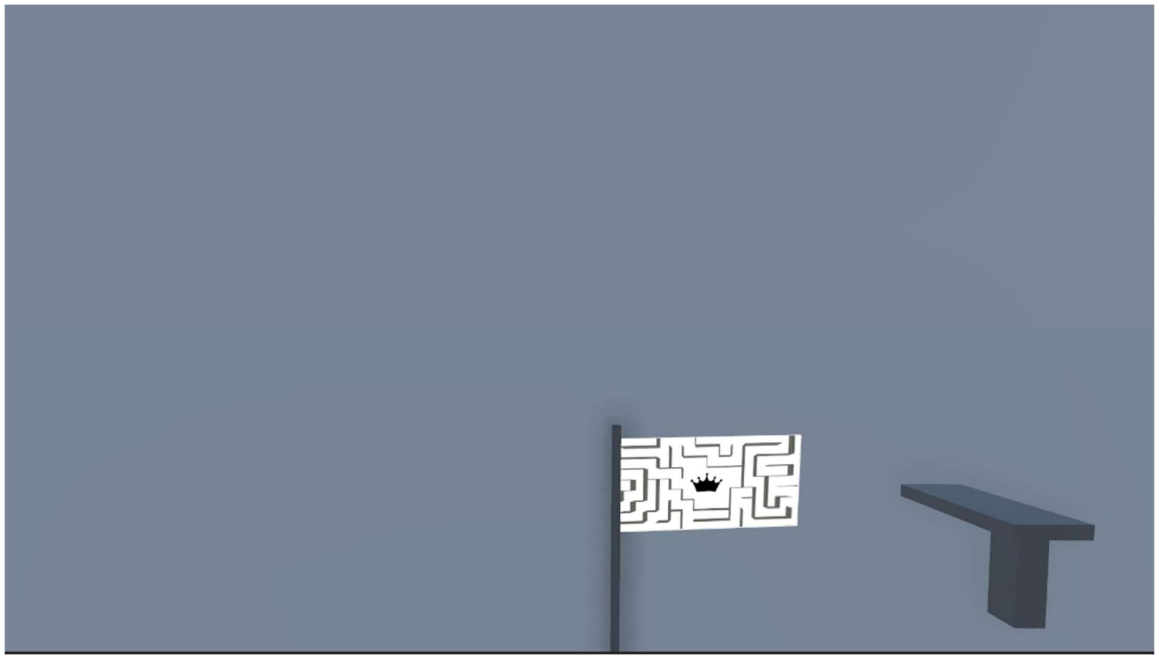

Figure 5

Reaching the flag

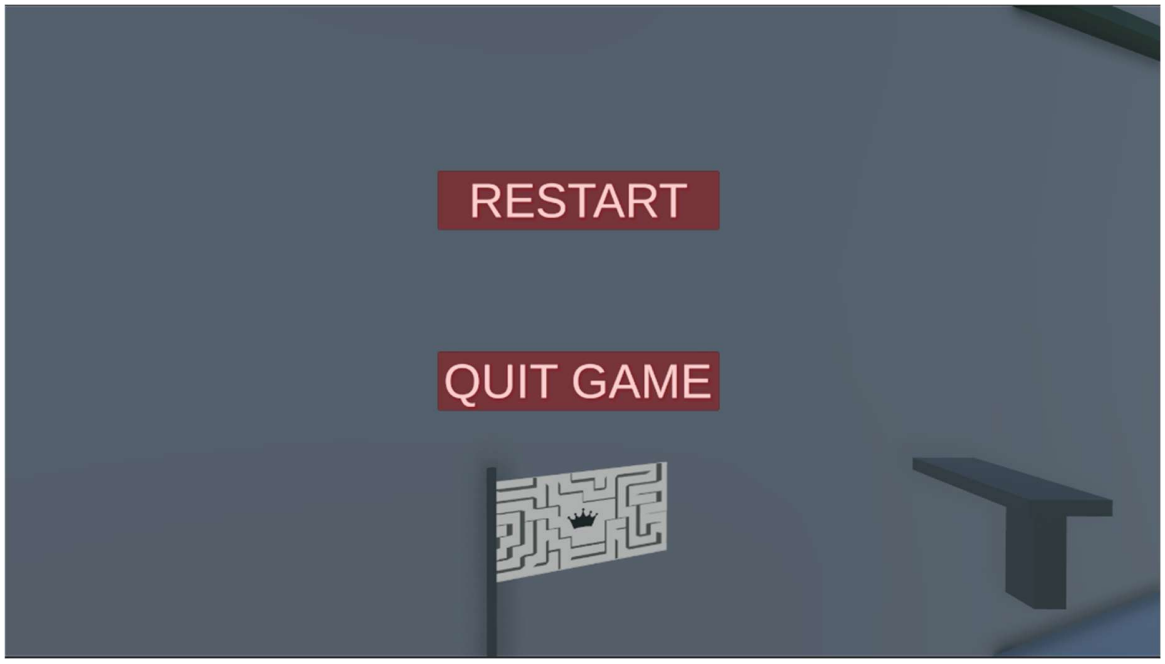

Figure 6

Game over

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Galway-Mayo Institute of Technology.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

_____

Acknowledgements

Special thanks to Michelle Lynch, who was my supervisor for my final year project. Her knowledge in machine learning was very helpful. I would also like to thank my lecturers, Brian O'Shea, Paul Lennon, and Niall O'Keeffe for reviewing and giving feedback on my project.
I would also like to thank my fellow classmates and members of my stand-up group, for giving feedback on different parts of the project report, poster, PowerPoint presentation and video.

1 Summary

The objective of my final year project, "The Maze Game," is to study the behavior of Machine Learning Agents [1] (ML-Agents) in a randomly generated maze environment. The game was developed using Unity Game Engine version 2021.113F1 [2] and coded in C#. The environment, player control, flag, and ML-Agents were all designed, coded and implemented in the game.

The maze environment is created through a script that generates a random maze of any size specified by the player. The player has basic controls such as mouse-based camera movement and keyboard-based movement (WASD or arrow keys). The objective of the game is to navigate the maze and reach the flag, which serves as the end goal.

The ML-Agents were trained using reinforcement learning, where they receive rewards based on their decisions. The agents receive observations every 2-3 seconds and make decisions based on these observations. Reaching the flag results in a reward, while running into a wall or getting stuck results in reduced rewards. This encourages the agents to make more logical decisions and navigate the randomly generated maze effectively.

The player also competes with the ML-Agents to reach the flag and earn points. The first to reach 5 points wins. The player has a gun that can shoot and damage the ML-Agents, and after 3 bullets, the agent is killed and must start again from the beginning. If the ML-Agent reaches and touches the player, the player is reset to the start and must navigate the maze again.

In conclusion, "The Maze Game" provides an interactive platform to study the behavior of ML-Agents in a randomly generated maze environment. The game offers a unique challenge for both the agents and the player as they navigate through the maze and reach the end goal.

Table of Contents

2 Poster

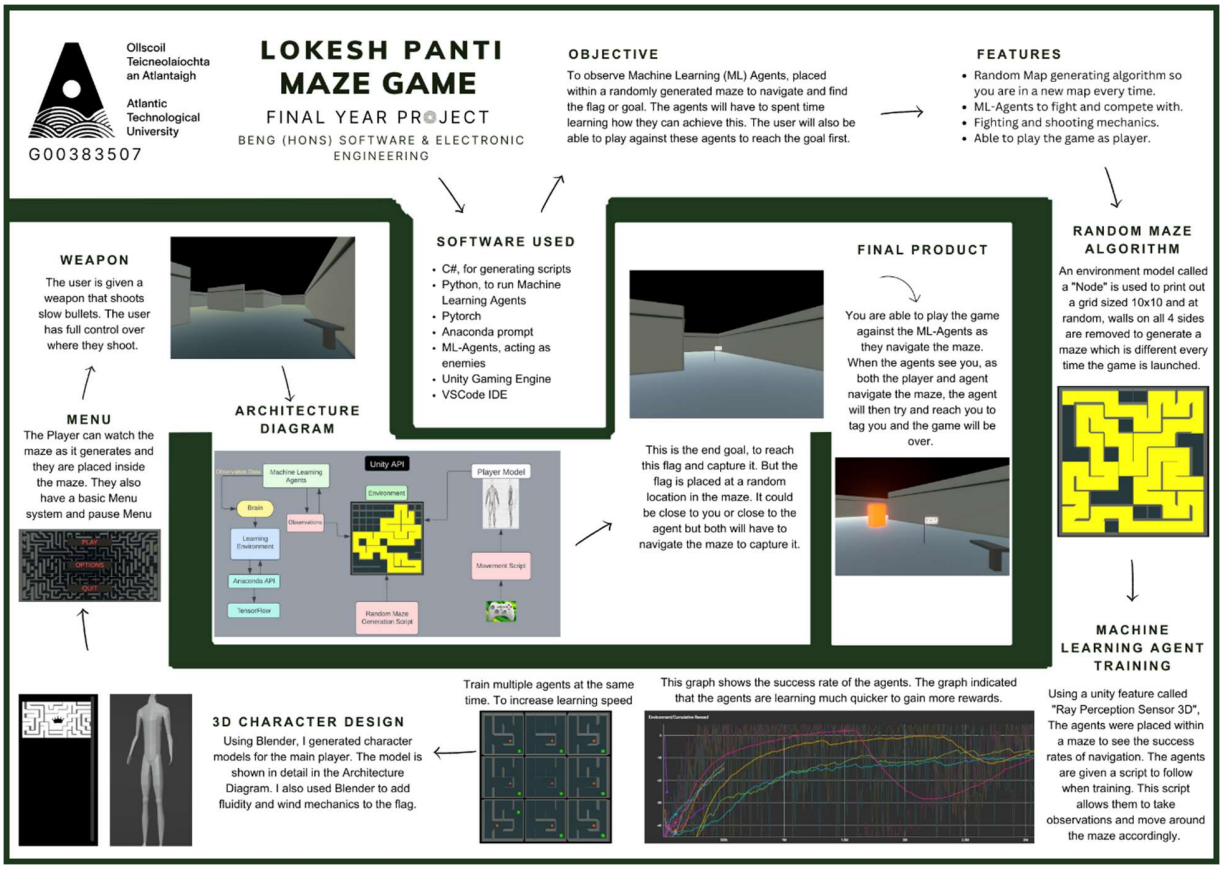

Figure 7

3 Project Planning

The bulk of my project planning is done on my personal OneNote. This OneNote is updated every week with details of what I have accomplished that week, details of what work I have pending, and details of the work from the fellow stand-up team members. I also used Jira to keep track of the different training tests I had running, tests I have done, and tests that are finished.

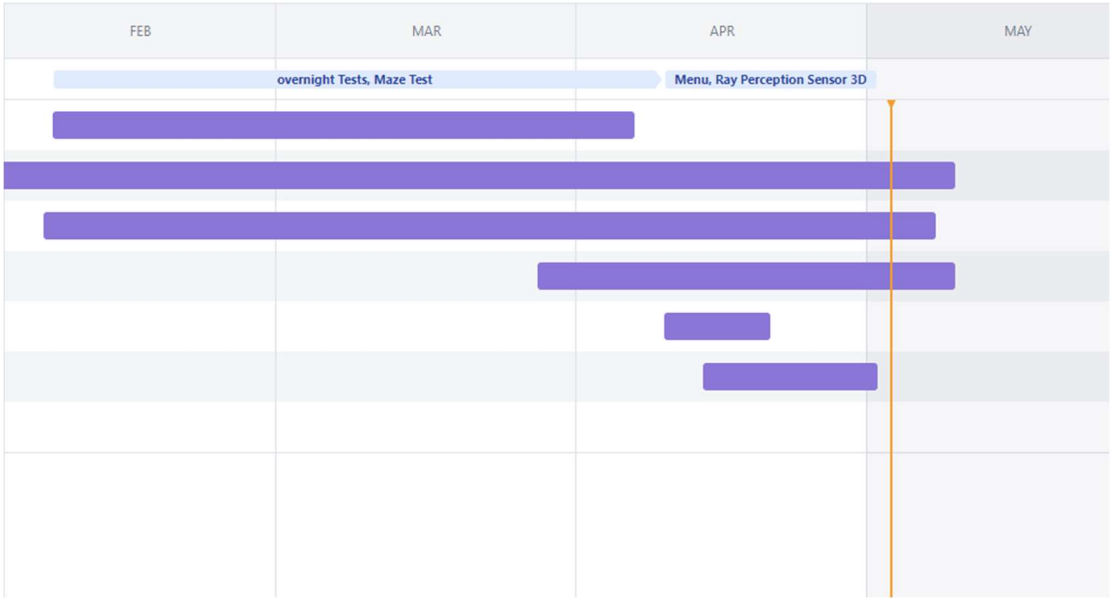

Figure 8

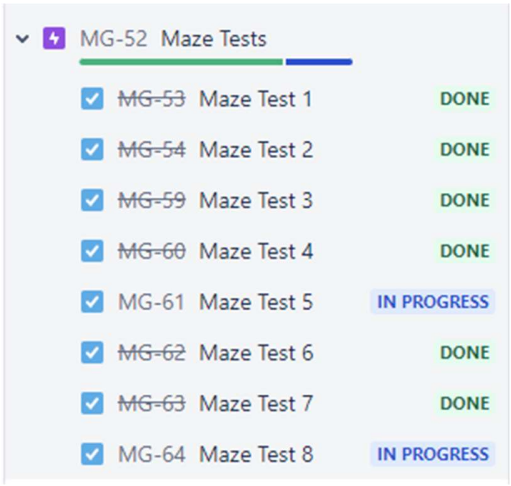

Figure 9

4 Introduction

4.1 Project Goal

This project focus around the learning and training of machine learning agents (ML-Agents). The project also offers the results of these agents into a game format to be viewed and studied upon.

The goals are:

- To create a working game for the user to play as player.
- To efficiently and accurately train the ML-Agents to navigate a randomly generated maze and find the goal.
- Write C# scripts for object-oriented gaming, player control, ML-Agents training scripts and environmental scripts for the random maze generation.

4.2 Project Report

This report contains:

- Approach to the project.
- ML-Agents training and learning.
- Explanation of the C# code.

5 Project Architecture

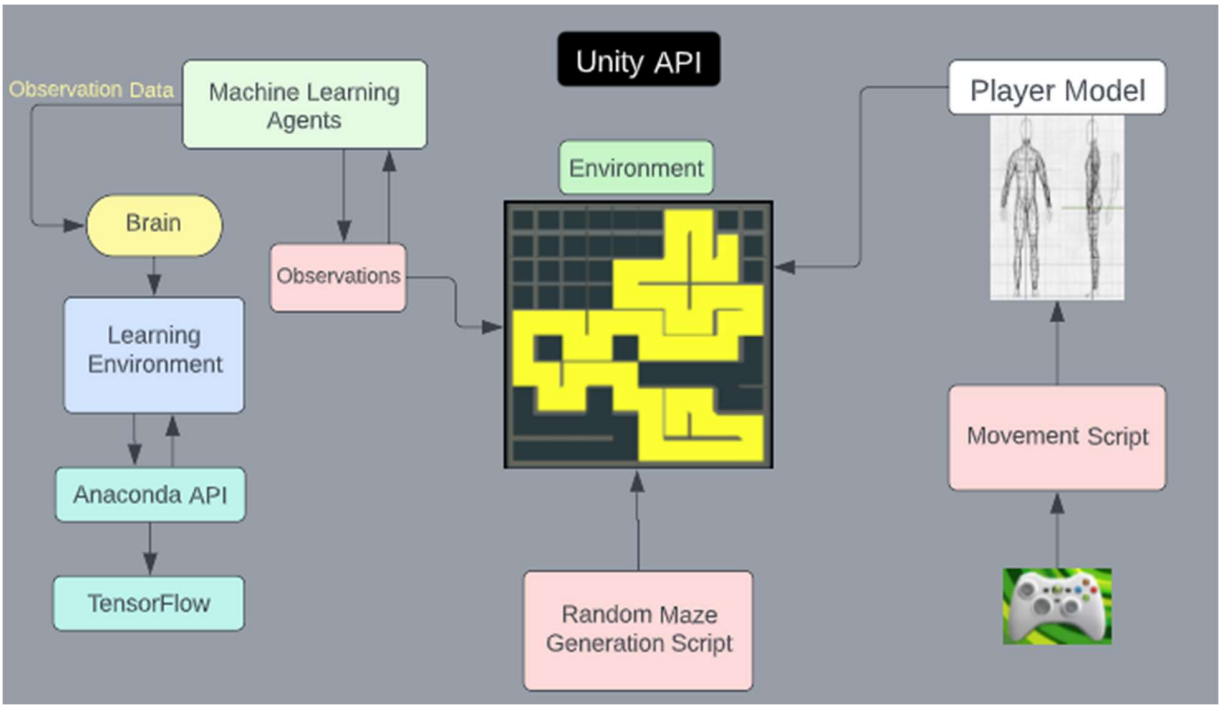

Figure 10

6 Virtual Environment

In unity, a virtual environment can be set up inside a 3D scene that can simulate game objects to user specifications, in this project a virtual environment is used to train ML-Agents. Virtual environments can be used to simulate scenarios where the agents can learn and improve upon their behaviour and decision making. These environments provide the agents with visual, audio and sensor data which can be used to make decisions. The virtual environment in my project the agents have access to visual and sensory data. Virtual environments can also be used to train various types of agents including, reinforcement learning, imitation learning and deep reinforcement learning agents. The flexibility and versatility of virtual environments make them an essential tool for training ML-Agents.

6.1 Python Installation

In this project I used Anaconda prompt to do all the command work for setup and training the ML-Agents. To install the prompt, go to this website and install https://www.anaconda.com/ .

The virtual environment can be set up in your Unity project file. First you must make sure you have the correct version of python installed. To check if you have the correct version of python open command prompt and run this command “python”. If python is not available, then a prompt should appear to install the latest version of python or go to the python website to install https://www.python.org/downloads/ .

6.2 Virtual Environment Setup

Once you have python installed, open anaconda prompt to start setting up the virtual environment. Navigate to the Unity project folder inside the anaconda prompt, once inside the project folder run this command ‘python -m venv <EnvironmentName>’. This creates the virtual environment where the agents will be trained.

```
C:\Users\G00383507@atu.ie\OneDrive - Atlantic TU\Project\Maze Game\Maze Game ver1>python -m venv Venv
```

To activate the virtual environment run this command in the prompt ‘<EnvironmentName>\scripts\activate’, this command will run the activate script inside the virtual environment to start.

```
C:\Users\G00383507@atu.ie\OneDrive - Atlantic TU\Project\Maze Game\Maze Game ver1>venv1\scripts\activate
```

6.3 PIP Install

PIP is a package manager for python that simplifies the installation and management of python packages. It allows users to install, upgrade and uninstall packages within their virtual environment. Pip retrieves packages from the Python Package Index (PyPI), a repository of Python packages, and installs them along with their dependencies.

Run this command to install pip: 'python -m pip install –upgrade pip'.

```
python -m pip install --upgrade pip
```

6.4 PyTorch Install

PyTorch is an open-source machine learning framework for Python that enables developers to build and train neural networks. PyTorch is highly flexible making it a valuable tool for deep learning applications. It contains a library for performing computations using data flow graphs to see representation of the learning models and other tools for research and production environments, like automation, GPU acceleration.

To install PyTorch go to the PyTorch website, navigate to install and in the specifications change them to suit your OS, Python, and latest version of CUDA. A command will be given like so: 'pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu11'

6.5 ML-Agents Install

To install ML-Agents within your virtual environment, follow these steps:

- Type `pip install mlagents` and hit enter. `pip install mlagents`
- Wait for the installation to complete.
- To verify type `mlagents-learn –help`.

This will setup the ML-Agents packages within your virtual environment. To work with ML-Agents inside Unity, follow these steps:

- In Unity open package manager by navigating to Window > Package Manager.
- Inside package manager change the Packages dropdown to 'Unity Registry'.
- Search inside the package manager for ML-Agents and install.

7 Environment

In Unity, an environment is a virtual 3D space where game objects, ML-Agents and other assets are placed and interacted with, enabling developers to create complex and dynamic scenarios. The environment can be customised using various tools and features, including physics simulations, audio effects, and visual effects to create a immersive and engaging experience. Overall, Unity's environment is a powerful platform for building a wide range of interactive applications.

7.1 Unity vs Unreal Engine

The reason why I chose Unity over Unreal Engine is Unity is a popular choice for beginner game developers due to its ease of use and intuitive user interface. Additionally, Unity has a larger community of developers and resources available online, making it easier to find support and learn new skills.

However, Unreal Engine offers advance features and better graphics capabilities, making it a better option for larger and more complex projects. As for training ML-Agents Unity is the preferred choice, as it has built-in support for ML-Agents and offers a more straightforward integration with the ML-Agents toolkit.

7.2 Maze Scene

Unity has a feature called 'scene'. A scene is a 3D environment you can create to separate parts of your game into different sections. Unity allows interaction between different scenes allowing the game to have menus and interactive systems.

This scene contains the essentials of our game like the player, ML-Agent, flag, maze generation, and weapons. The training of the ML-Agents will also be done here until they are able to navigate a maze to their full ability.

7.3 Player Model

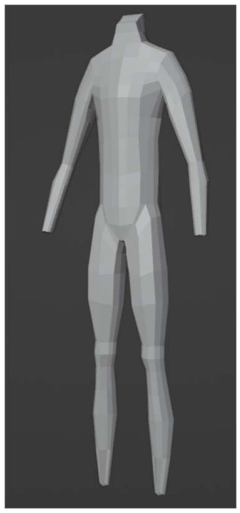

Figure 11 [3]

This player model is created using Blender. Blender is a free open-source 3D creation software used to create 3D models. Blender allows the user to add visual effects and animations to the models. It provides a range of tools for modelling, sculpting, rigging, animation, simulation, rendering, and compositing. Blender is widely used in the gaming industry for creating animations and visual effects for films and videos.

This is the player model that I created for the game. The inspiration for this player model came from the film X-Men. They contain a character in the movie called Sentinels who have a similar body type.

7.4 ML-Agent Model

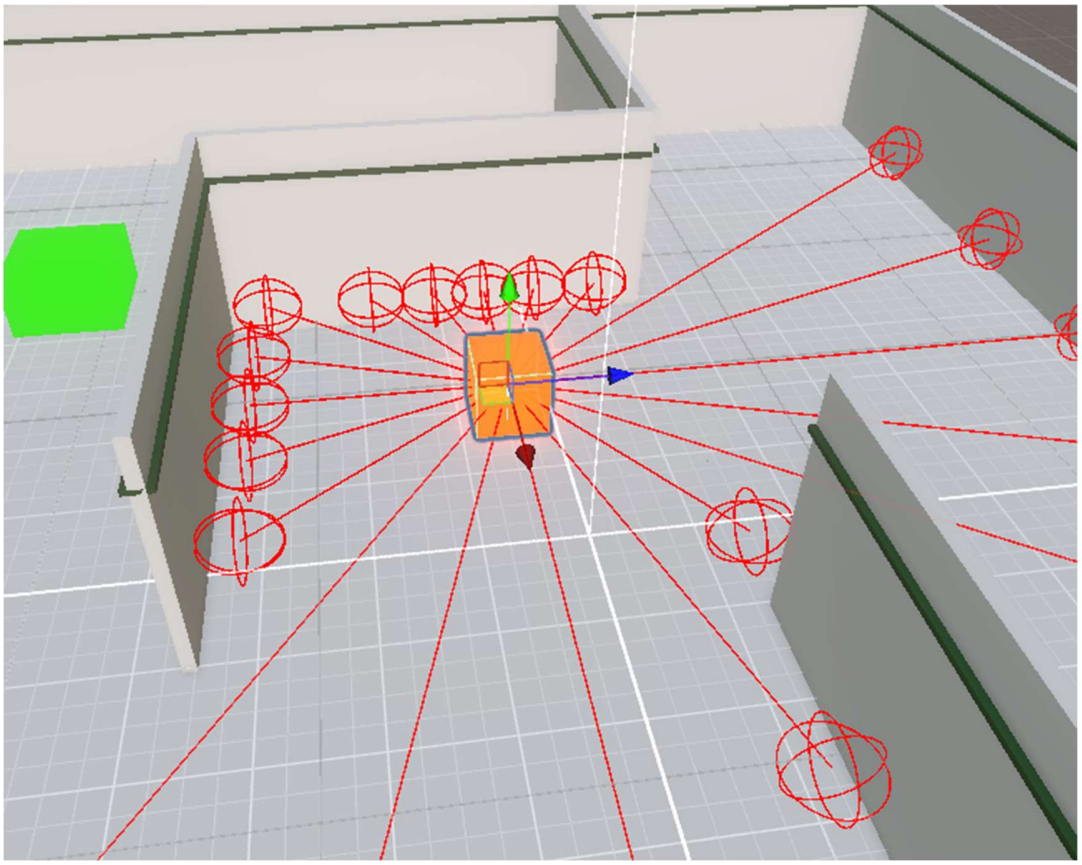

Figure 12 Ray Perception Sensor 3D [4]

This is the model I have for the agent, instead of having body parts and training the agents to walk I chose a simpler model for easy training purposes. This model is a cube that has several components attached to it like Rigid body to give the agent solidity physics, Box Collider so the agent can collide with walls and floor, Ray Perception Sensor 3D which allows the agent to see and finally a mesh renderer to determine the shape of the object.

7.5 Flag / Goal

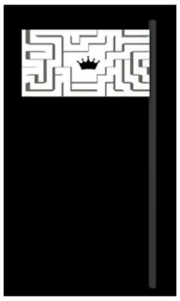

Figure 13

The goal in this game is a Flag. The design for this flag is a background of my maze with a crown representing a flag. This flag contains a reward for the agent and player.

7.6 Maze Generation

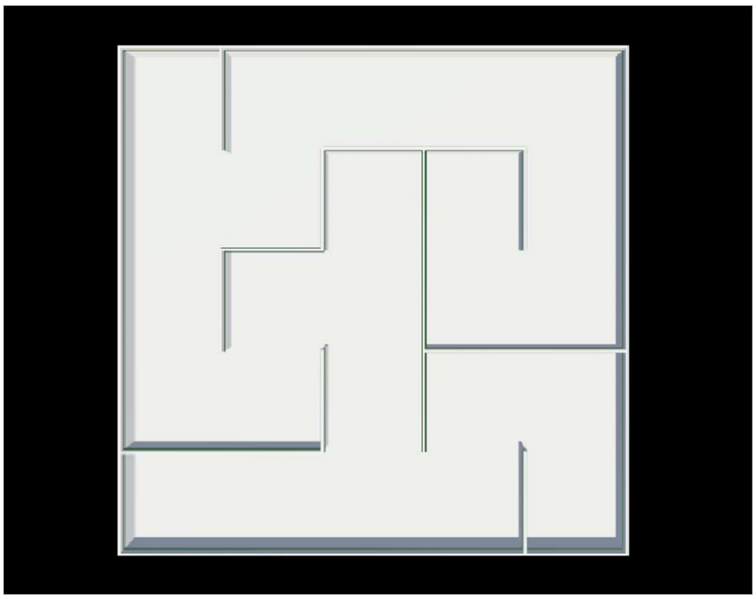

Figure 14

This is the product of the random maze generation. The maze size is set by the player in the options menu. After the maze is generated the player, agent and flag are placed in their desired locations. The agent and player are placed at direct opposites, bottom left and top right respectively and the goal is placed at a random location.

7.7 Weapon

Figure 15

The weapon for this game is a simple design that I created in Unity. This weapon is much more minimalist and shoots blue glowing bullets. The bullets damage the agent and after a few bullets the agent is reset to its original position.

8 Random Maze Generation

Every time the game is launched a random maze is generated which can be set to a certain size by the player before they start playing. The maze is generated using a model called a node. A node is a 1x1 floor with four walls surrounding it. The nodes are then placed on a grid specified to the size of the maze set by the player [5].

The code uses a custom class called “MazeMap” to represent the nodes in the grid. Each node has four walls, which are represented as Boolean values, and the nodes have three states, AvailableNode, CurrentNode, and CompletedNode.

8.1 Grid

```
for(int x = 0; x < size.x; x++){
    for(int y = 0; y < size.y; y++){
        //center the code at (0, 0) instead of random locations
        Vector3 nodePos = new Vector3(x - (size.x/2f), 0, y - (size.y/2f)) * nodeSize;
        // instantiate the new node...set it as parent
        MazeMap newNode = Instantiate(nodePrefab, nodePos, Quaternion.identity, transform);
        nodes.Add(newNode);

        yield return null;
    }
}
```

This code creates a 2D grid of nodes centred at (0, 0) with dimensions determined by the “size” variable. It instantiates a new “nodePrefab” object for each node in the grid, sets the node position and adds it to a list called “nodes”. The use of “yield return null”” suggests this is running as a coroutine in Unity.

8.2 Algorithm

```
Random.seed = System.DateTime.Now.Millisecond;

 currentPath.Add(nodes[Random.Range(0, nodes.Count)]);
 currentPath[0].SetState(NodeState.Current);
```

Using a while loop that loops until all the possible nodes are Complete Nodes. The code creates two empty lists to store the possible next nodes and their directions. It then calculates the current node’s position, using the “nodes” list, “currentPath”, and “completeNodes” lists. This calculation is used to backtrack through the maze to find any nodes that were unvisited and continues generating the maze.

```
//create loop to generate the next node and load the map
while(completedNodes.Count < nodes.Count){
    //check nodes
    List<int> possNextNodes = new List<int>();
    List<int> possDirections = new List<int>();

    //we are getting the poistion so the node can back tack to find a un-used node and continue generating the map
    int currentNodeIndex = nodes.IndexOf(currentPath[currentPath.Count -1]);
    int currentNodeX = currentNodeIndex / size.y;
    int currentNodeY = currentNodeIndex % size.y;
```

8.3 Position and Direction

```
if(currentNodeX < size.x - 1){
    if(!completedNodes.Contains(nodes[currentNodeIndex + size.y]) && !currentPath.Contains(nodes[currentNodeIndex + size.y])){
        // telling the current node what direction it can move in

        possDirections.Add(1);
        possNextNodes.Add(currentNodeIndex + size.y);
    }
}
// same as the if statement before but we are checking if the current ndoe is near the left edge wall
if(currentNodeX > 0){

    if(!completedNodes.Contains(nodes[currentNodeIndex - size.y]) && !currentPath.Contains(nodes[currentNodeIndex - size.y])){
        possDirections.Add(2);
        possNextNodes.Add(currentNodeIndex - size.y);
    }
}

// checking the node above the current node
if(currentNodeY < size.y -1 ){
    if(!completedNodes.Contains(nodes[currentNodeIndex + 1]) && !currentPath.Contains(nodes[currentNodeIndex + 1])){
        possDirections.Add(3);
        possNextNodes.Add(currentNodeIndex + 1);
    }
}

// checking the node below the current node
if(currentNodeY >  0 ){
    if(!completedNodes.Contains(nodes[currentNodeIndex - 1]) && !currentPath.Contains(nodes[currentNodeIndex - 1])){
        possDirections.Add(4);
        possNextNodes.Add(currentNodeIndex - 1);
    }
}
```

This code checks the adjacent nodes of the current node and adds them to the list of possible next nodes if they have not been visited yet.

The code first calculates the position of the current node using the “nodes”, “currentPath”, and “completedNodes” lists. It then checks if the current node is near the right or left edge of the maze, and if there is an unvisited node to the right or left, it adds the node to the list of possible next nodes, along with a direction value of 1 or 2, respectively.

Similarly, it checks if there is an unvisited node above or below the current node and adds it to the list of possible next nodes. The direction value of 3 or 4 is assigned to the node above or below, respectively.

The “possDirections” list keeps track if the possible direction to move from the current node and “possNextNodes” list keep track of thepossible next nodes indices, these lists are later used in the code to choose the next node to visit.

8.4 Selecting Node

```
if(possDirections.Count > 0){
    int choseDir = Random.Range(0, possDirections.Count);
    MazeMap choseNode = nodes[possNextNodes[choseDir]];

    switch(possDirections[choseDir]){
        case 1:
        choseNode.RemoveWall(1);
        currentPath[currentPath.Count-1].RemoveWall(0);
        break;
        case 2:
        choseNode.RemoveWall(0);
        currentPath[currentPath.Count-1].RemoveWall(1);
        break;
        case 3:
        choseNode.RemoveWall(3);
        currentPath[currentPath.Count-1].RemoveWall(2);
        break;
        case 4:
        choseNode.RemoveWall(2);
        currentPath[currentPath.Count-1].RemoveWall(3);
        break;
    }


    currentPath.Add(choseNode);
    choseNode.SetState(NodeState.Current);
}
```

This code chooses the next node to be visited and connecting it with the current node. It checks if there are any possible directions to move by checking if “possDirections” list has any elements. If there are any directions available, it chooses one direction randomly using “Random.Range()” method and removes the walls between the current node and the chosen node based on the direction.

It then adds the chosen node to the “currentPath” list and sets its state as “current”. The “SetState” function changes the colour of the node to show it as current node visually. The switch statement chooses which wall to remove based on the directions chosen. The wall between the current node and the chosen node is removed by changing the corresponding boolean value in the “MazeMap” class using the “RemoveWall()” function.

This block of code is responsible for removing the walls between each node to generate a random maze every time the game is launched.

8.5 Back Tracking

```
else{
    completedNodes.Add(currentPath[currentPath.Count-1]);
   CompNodes.Add(currentPath[currentPath.Count-1]);

    currentPath[currentPath.Count-1].SetState(NodeState.Completed);
    currentPath.RemoveAt(currentPath.Count-1);
}
```

In the while loop, the algorithm checks for the possible next nodes to move on from the current node. If there is no possible direction to move forward, it means the algorithm has reached a dead-end, and backtracking is required. The last node in the current path is then removed, and the algorithm continues checking for the possible directions to move forward. The completed nodes list keeps track if the nides that have been visited and are parts of the generated maze. Once the algorithm has visited all the nodes, the maze generation is completed.

8.6 Material Change

```
switch (state)
{
    case NodeState.Available:
        floor.material.color = Color.red;
        break;
    case NodeState.Current:
        floor.material.color = new Color(255, 230, 0);
        break;
    case NodeState.Completed:
        floor.material=compNode;
        break;
}
```

This code uses a switch statement to change the colour of a floor based on the current state of the node. If the state is “Available”, the floor is coloured red. If the state is “Current”, the floor colour is yellow, and if the state is “Complete” the floor is changed to a specified material. A material defines the appearance of an object by specifying its visual properties such as colour, texture, and shading.

9 ML-Agents Training

9.1 Overview

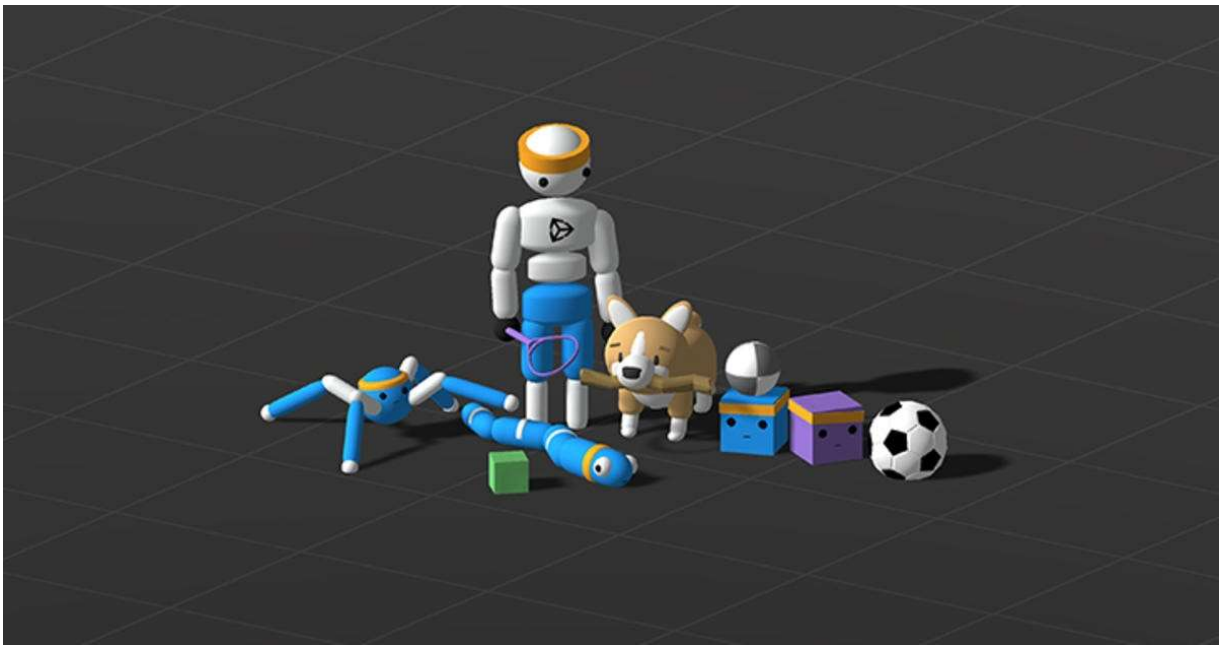

Figure 16 ML-Agents [6]

Machine learning has become a game-changer in many industries, and the gaming industry is no exception. Unity Technologies has developed a powerful toolkit called ML-Agents that allow game developers to integrate machine learning, various reinforcement learning algorithms into their Unity project.

With ML-Agents, developers can create agents that learn and adapt to their environment. This enables endless possibilities for game AI, enabling game developers to design complex behaviours for NPCs (Non-Playable Characters) and game characters. Unity allows game developers to create complex scenarios for training and testing and optimise game mechanics for a better player experience.

The toolkit supports a variety of machine learning algorithms and techniques, including deep reinforcement learning [7], imitation learning, and evolutionary strategies. This flexibility provides developers with a range of options to choose from to find the algorithm that best suits their needs. Instead of creating a player vs player environment, training ML-Agents would allow

the player to fight against the agent, given enough time to train the agents to navigate the maze.

9.2 Training Type

Navigating a maze is a complex task that requires learning and decision-making in a dynamic environment. There are two different types of learning I used to train the agents, deep reinforcement learning, and exploitation and exploration trade off [8].

Training agents to navigate a maze is challenging due to the high dimensionality of the problem and the complexity of the environment. The agent needs to learn to identify its current location, the location of the goal and avoid obstacles. Additionally, the agent needs to learn how to balance the exploration and exploitation to maximize its expected reward while avoiding getting stuck.

9.2.1 Training Algorithms

Two of the most widely used algorithms for training ML-Agents in Unity are Proximal Policy Optimization (PPO) and Self-Organizing Curiosity (SOC). PPO is a policy gradient-based algorithm that uses stochastics gradient ascent to update the agent's policy. The objective is to maximize the expected cumulative reward. PPO is known for its ability to achieve stable and efficient training with good sample efficiency. It can handle the complexity of the maze environment and balance exploration and exploitation tradeoff.

SOC, on the other hand, is an exploration strategy that encourages agents to explore their environment by rewarding them for novel experiences. SOC relies on a curiosity-based intrinsic motivation system to drive exploration and learn new skills. While SOC can be effective in some scenarios, it may suffer from slow convergence and difficulty in learning complex tasks, like maze navigation, where there are many possible actions and states to consider. As such, PPO is generally considered a more reliable choice for navigation mazes.

9.2.2 Exploitation and Exploration Trade-off

Exploration-exploitation trade-off is another critical aspect of training agents to navigate a maze. The agent needs to explore the environment to learn about its structure and find the optimal path to the goal. However, if the agent only explores and does not exploit its current knowledge, it can lead to inefficient learning. Conversely, if the agent only exploits its current knowledge and does not explore, it can get stuck in suboptimal solutions.

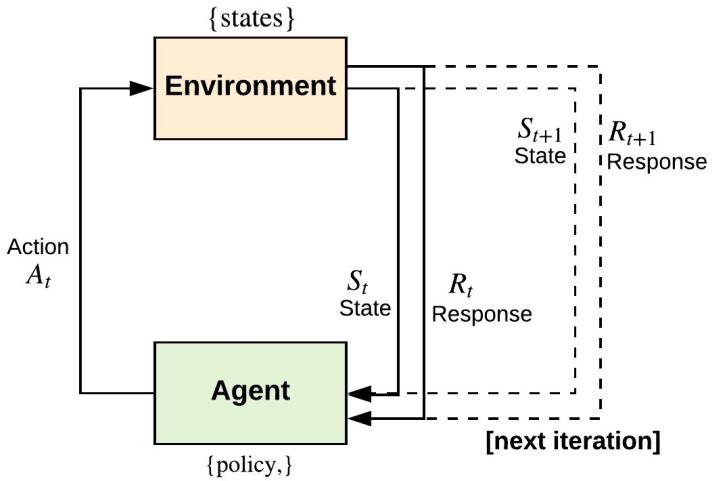

Figure 17 Exploration and Exploitation Trade off [8].

9.3 Approach

ML-Agents must be trained very carefully, if the agents do not take in correct observations or receive incorrect instructions, the brain model the agent is creating using the observations and instructions will be created with defects.

In the beginning I had a very basic approach towards training the ML-Agents. The very beginning I only had the agents moving in one direction, but training to avoid walls and reach the goal more efficiently.

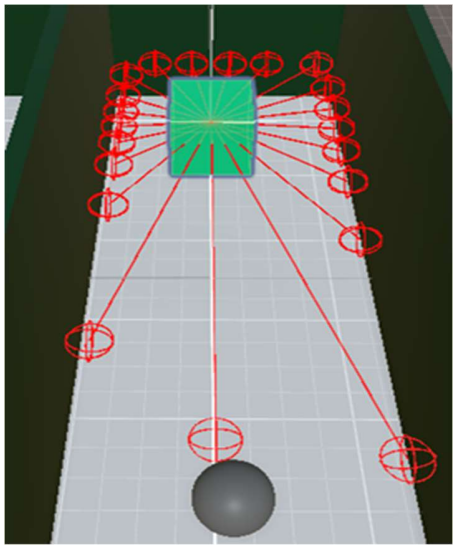

Figure 18

This image shows perfectly how the agents can see their surroundings. Unity has a component called ‘Ray Perception Sensor’, this component gives the agents rays which shoot out from the centre allowing the agents to see their surroundings. The rays are customisable to send the correct data to the agent.

9.4 Ray Perception Sensor

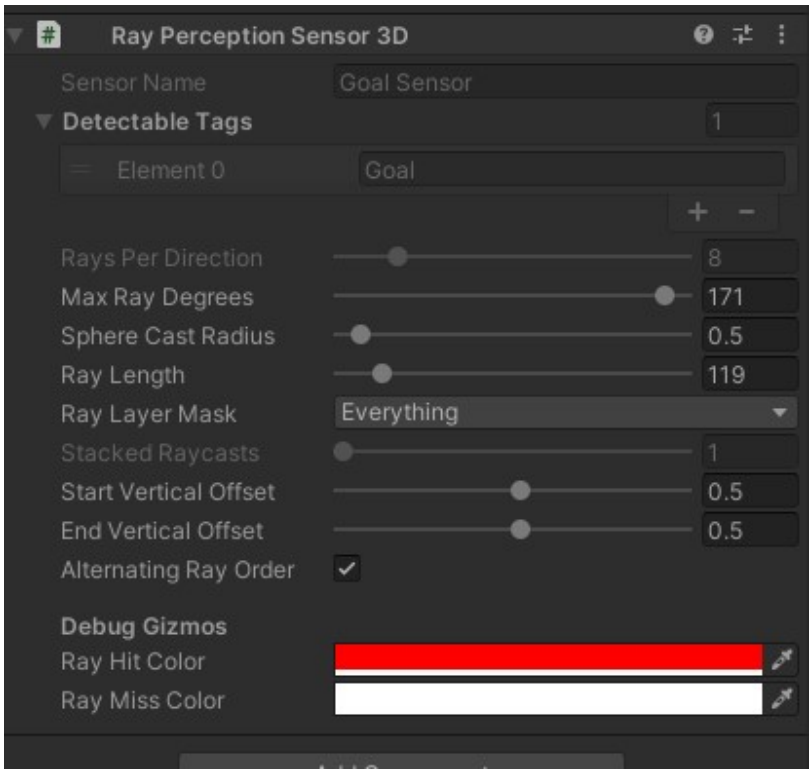

Figure 19

- Max Ray Degrees: The maximum angle between two adjacent rays in the sensor's field of view.
- Sphere Cast Radius: The radius of the sphere used to cast the rays.
- Ray Length: The length of each ray cast by the sensor.
- Ray Layer Mask: The layers to include in the raycast.
- Start Vertical Offset: The vertical offset of the starting position of the raycast.
- End Vertical Offset: The vertical offset of the ending position of the ray casts.

9.5 Goal Finding Training

To train the agents to reach the goal, the first step is to modify the environment to optimise their learning process. Rather than a maze, an open area with four walls on the edges was introduced. This allows the agents to navigate and explore more freely without being hindered by complex obstacles. By adapting to his simplified environment, the agents can focus on learning how to achieve their objective more efficiently. The modification of the environment is a critical aspect of training the agents, as it can significantly impact the rate at which they acquire the necessary skills to succeed.

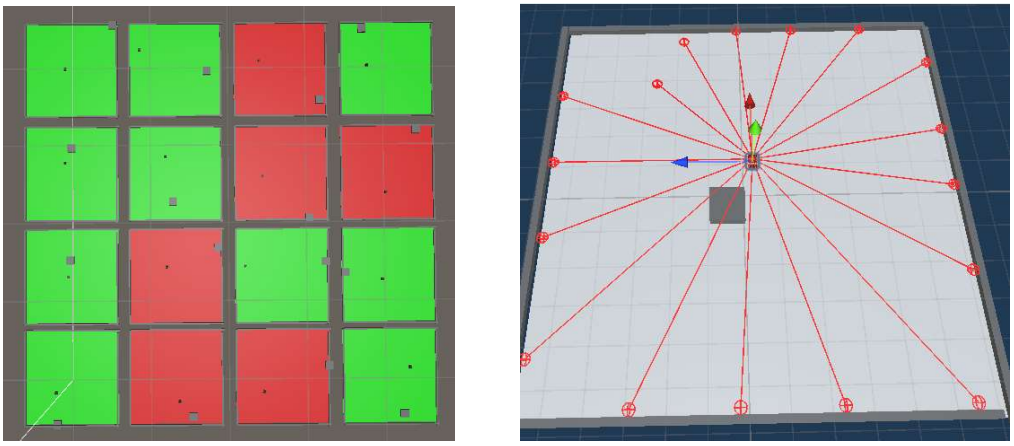

Figure 20

This is the environment changes, for demonstration purposes, if the agent successfully navigates to the goal, the floor will change colour to green, and if the agent fails to reach the goal the floor colour changed to red.

9.6 Training Script

These are the Unity Machine Learning Agents (ML-Agents) packages used in the script. It imports necessary classes and namespaces for creating agents that can perceive their environment using sensors, take actions using actuators and learn through reinforcement learning.

```
0 references | LOKESH PANTI - STUDENT, 29 days ago | 1 author, 1 change
public override void Initialize()
{
    orig = new Vector3(this.transform.position.x, this.transform.position.y, this.transform.position.z);
    Target = this.transform.parent.transform.Find("Goal").gameObject;
}
```

Figure 21

This is a method in C# used to initialise a Unity game object's position and target location for the ML-Agent training. It sets the original position of the agent to its current position and defines the target location as a child game object named "Goal" of its parent game object.

```
public override void OnEpisodeBegin()
{
    stepsTaken = 0;
    this.transform.localPosition = new Vector3(Random.Range(minX, maxX), 0, Random.Range(minZ, maxZ));
    Target.transform.localPosition = new Vector3(Random.Range(minX, maxX), 0, Random.Range(minZ, maxZ));
    prevDistanceToGoal = Vector3.Distance(transform.localPosition, Target.transform.localPosition);
}
```

Figure 22

This code is used to reset the environment and the agent's position at the start of each episode in a reinforcement learning task. The method initializes the step count to zero and randomizes the position of the agent and the target location within a specified range. It then calculates the distance between the agent and the target location using the "Vector3.Distance" method and stores it as the "prevDistanceToGoal". This method ensures that the agent is starting in a new environment and has a new target to reach each time an episode begins, allowing it to learn and adapt to new situations trough the reinforcement learning process.

```
public override void CollectObservations(VectorSensor sensor)
{
    sensor.AddObservation(transform.localPosition);
    sensor.AddObservation(Target.transform.localPosition);

}
```

Figure 23

This code is used to collect observations from the environment and add them to a sensor for the ML-Agents. This method adds the current position of the agent and the position of the target location to the sensor, which is then used as an input for the agent's decision-making process.

```
0 references | LOKESH PANTI - STUDENT, 21 days ago | 1 author, 3 changes
public override void OnActionReceived(ActionBuffers vectorAction)
{
    float moveX = vectorAction.ContinuousActions[0];
    float moveZ = vectorAction.ContinuousActions[1];
    Target = this.transform.parent.transform.Find("Goal").gameObject;

    float currentDistanceToGoal = Vector3.Distance(transform.localPosition, Target.transform.localPosition);

    Vector3 moveDirection = new Vector3(moveX, 0, moveZ).normalized;
    Vector3 newPosition = transform.position + moveDirection * Time.deltaTime * Movespeed;
    AddReward(-0.001f);

    RaycastHit hit;
    if (Physics.Raycast(transform.position, moveDirection, out hit, rayDistance, wallMask))
    {
        if (hit.distance <= 0.5f)
        {
            AddReward(+0.1f);
            return;
        }
    }

    transform.position = newPosition;

    Vector2Int currentState = GetDiscretePosition(transform.localPosition);
    if (recentVisitedStates.Contains(currentState))
    {
        AddReward(-0.2f);
    }

    // Update the recent visited states queue
    recentVisitedStates.Enqueue(currentState);
    if (recentVisitedStates.Count > maxRecentVisitedStates)
    {
        recentVisitedStates.Dequeue();
    }

    float newDistanceToGoal = Vector3.Distance(transform.localPosition, Target.transform.localPosition);
    float distanceReward = (prevDistanceToGoal - newDistanceToGoal) * 0.1f;
    AddReward(distanceReward);

    prevDistanceToGoal = newDistanceToGoal;
    stepsTaken++;
}
```

Figure 24

The OnActionReceived method is called each time the agent takes an action. The method receives a vector of continuous actions that represents the agent's movement in the c and z directions. The code then calculates the distance between the agent's current position and the target location using the "Vector3.Distance" method. It then adds a small negative reward (-0.001f) to encourage the agent to reach the target location quicky

If the agent's movement direction intersects with a wall, the code checks if the distance to the wall is less than 0.5 units. If so, the agent receives a positive reward (+0.01f) and returns. The code also updates the agents position and calculates the new distance to the target location.

Overall, this method updates the agent's state and reward based on its current action, position, and the environment it interacts with, allowing the agent to learn more efficiently.

9.7 Reward System

```
void OnCollisionEnter(Collision col)
{
    if (col.gameObject.CompareTag("Goal"))
    {
        float reward = 5f ;
        SetReward(reward);
        Debug.Log("Hit Goal");
        floorMeshRenderer.material = winMaterial;
        EndEpisode();
    }
    else if (col.gameObject.CompareTag("Wall"))
    {
        SetReward(-6f);
        Debug.Log("Hit Wall");
        floorMeshRenderer.material = loseMaterial;
    }
}
```

Figure 25

This code is called when the agent collides with an object in the environment. The method checks the tags of the object the agent collided with and assigns a reward to the agent based on the tag.

If the agent collides with the "Goal" object, it assigns a positive reward of 5 to the agent, which indicates a successful completion of the task. The method then sets the material of the floor object to a "winMaterial" and ends the current episode, the episode is then restarted to reset the position of the goal so the agent can begin navigating the maze again.

If the agent collides with a "Wall" object, it assigns a negative reward of -6 to the agent, which indicates a failure to complete the task. The method sets the material of the floor object to a "loseMaterial" and the agent continues in the same episode, until it collides with the goal.

This reward system is good for navigating a maze because it provides clear and concise reinforcement signals to the agent base on its actions. The positive reward for reaching the goal encourages the agent to move towards the goal location, while the negative reward for hitting a wall discourages the agent from taking actions that lead to collisions with walls. This reward system is aligned with the goal of navigating a maze, which is to reach the goal as fast as possible while avoiding obstacles.

The material change is used for visualising the progress of the agent and makes it easier for me to understand the training process.

9.8 Maze Training

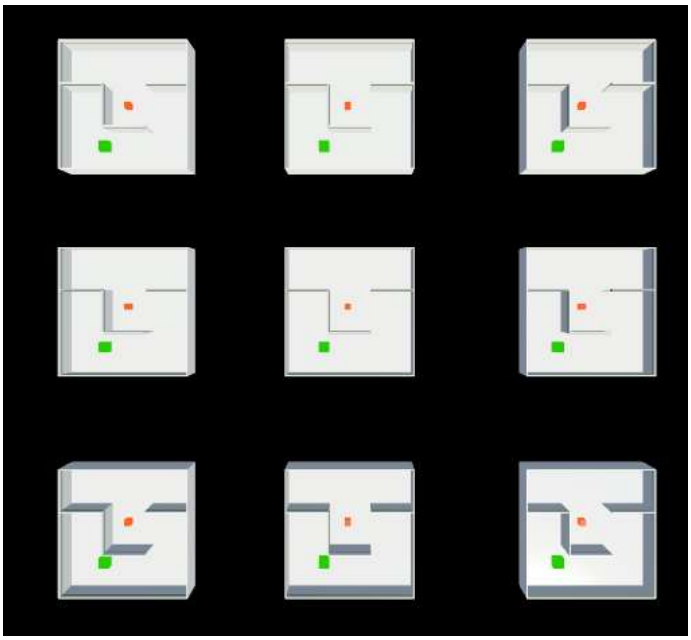

Figure 26

To begin maze training, the agents were placed within a small 3x3 maze and were programmed to find the goal while avoiding walls. The agents were successful in finding the goal but were still facing problems as ML-Agents still need time to learn. These agents were successfully navigating and moving around the walls to find the goal.

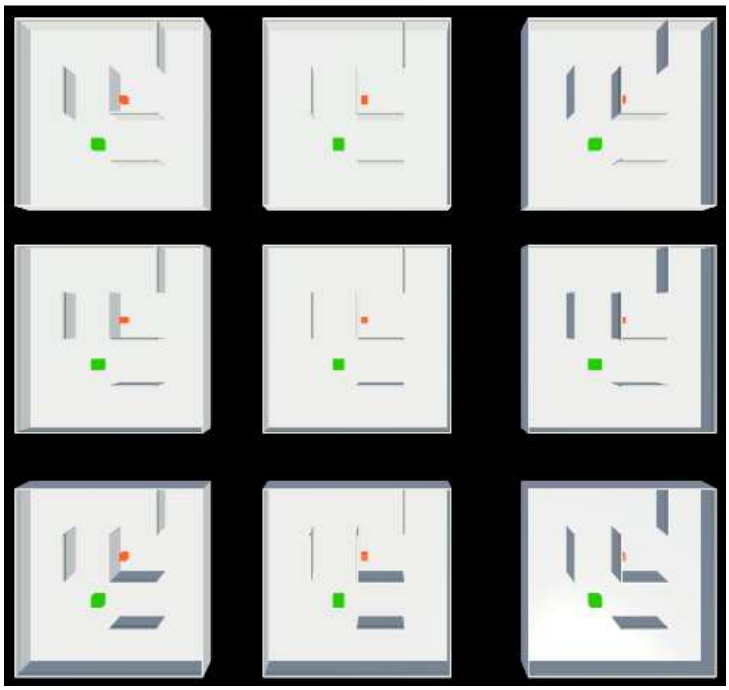

Figure 27

Once the agents were fully successful in navigating a 3x3 maze, they were placed in a maze of a bigger size, but some obstacles like the walls shown above are placed at random so the agent would learn more efficiently how to avoid walls and find the goal. The agents were navigating this premature maze successfully and were actively avoiding walls.

9.9 Brain

The Brain component in Unity ML-Agents is what enables the agents to learn from its experiences in the environment and make decisions based on that learning. During training, the Brain receives input data from the environment and generates actions for the agent to take. The results of the agent successfully navigating the 4x4 maze and actively avoiding walls were stored in this Brain model, and this brain model was used to train the future mazes.

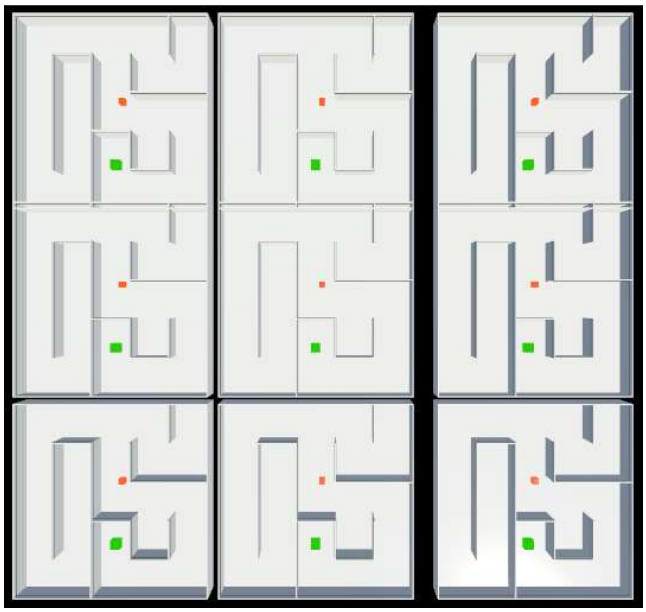

Figure 28

The agents were placed in this new environment which is a larger 5x5 maze and are training to navigate this maze. The agents can navigate around this maze and successfully find the goal. The complexity of the environment changes with every training scenario. This enables the agent to develop the skills they need to adapt to new environments and overcome new challenges. The results from this training experience are stored within the Brain.

This Brain model is now used as a basis for training the agents to navigate random maze of size 5x5. This means that the agents can navigate a wide variety of different mazes and are not limited to a single scenario.

10 C# Scripts

C# [9] scripts are an essential component of creating interactive games and applications in Unit. C# is a high-level programming language that allows developers to write code that interacts with Unity's game engine, creating objects, modifying scenes, and handling user inputs. In Unity, C# scripts are usually attached to game objects, and they provide the logic and functionality for those objects. C# scripts can perform various function such as handling collisions, changing materials, and generating random mazes [10].

10.1 Player Movement Scripts

```
// Update is called once per frame
void Update()
{

    isGround = Physics.CheckSphere(ground.position, groundDistance, groundMask);

    if(isGround && velocity.y < 0){
        velocity.y = -2f;
    }

    float horizontal = Input.GetAxisRaw("Horizontal");
    float vertical = Input.GetAxisRaw("Vertical");


    Vector3 move = transform.right * horizontal + transform.forward * vertical;

    controller.Move(move * speed * Time.deltaTime);

    velocity.y += gravity * Time.deltaTime;

    controller.Move(velocity * Time.deltaTime);



}
```

Figure 29

This code handles player movement. The update function is called once per frame and checks whether the player is on the ground by performing a sphere cast. If the player is on the ground, it sets the velocity to -2f, which simulates gravity. The code then reads input from the horizontal and vertical axis and calculates the players movement direction. This direction is then used to move the player using the "controller.Move" function, which applies the movement to the players position. Finally, the code applies gravity to the players vertical velocity and moves the player using the same function.

10.2 Gun Script

```
// Update is called once per frame
void Update()
{

    if(Input.GetKeyDown(KeyCode.Mouse0)){
        var bullet = Instantiate(bulletPrefab, bulletSpawn.position, bulletSpawn.rotation);
        bullet.GetComponent<Rigidbody>().velocity = bulletSpawn.forward * bulletSpeed;
    }


}
```

Figure 30

This code is responsible for shooting the agent. The Update function is called once per frame and checks if the mouse right click is pressed. If pressed creates an instance of a bullet object at the “bulletSpawn” position and rotation using the “Instantiate” function.

The newly created bullet object is given a velocity that is calculated by multiplying the forward direction of the “bulletSpawn” object with the “bulletSpeed” variable. The velocity is applied to the bullet's rigid body using the “GetComponent<Rigidbody>()”.velocity function, causing it to move forward in the direction the “bulletSpawn” is facing.

10.3 Main Menu

```
public class MainMenu : MonoBehaviour
{

    public void PlayGame()
    {
        SceneManager.LoadSceneAsync(1, LoadSceneMode.Single);
        Cursor.lockState = CursorLockMode.Locked;
    }

    public void QuitGame(){
        Application.Quit();
        Debug.Log("Player Has Quit");
    }
}
```

Figure 31

The main menu script navigates through the different scenes. The starting scene is the main menu where the player will click play, and the scene is changed to “MazeScene” so the player can play the game.

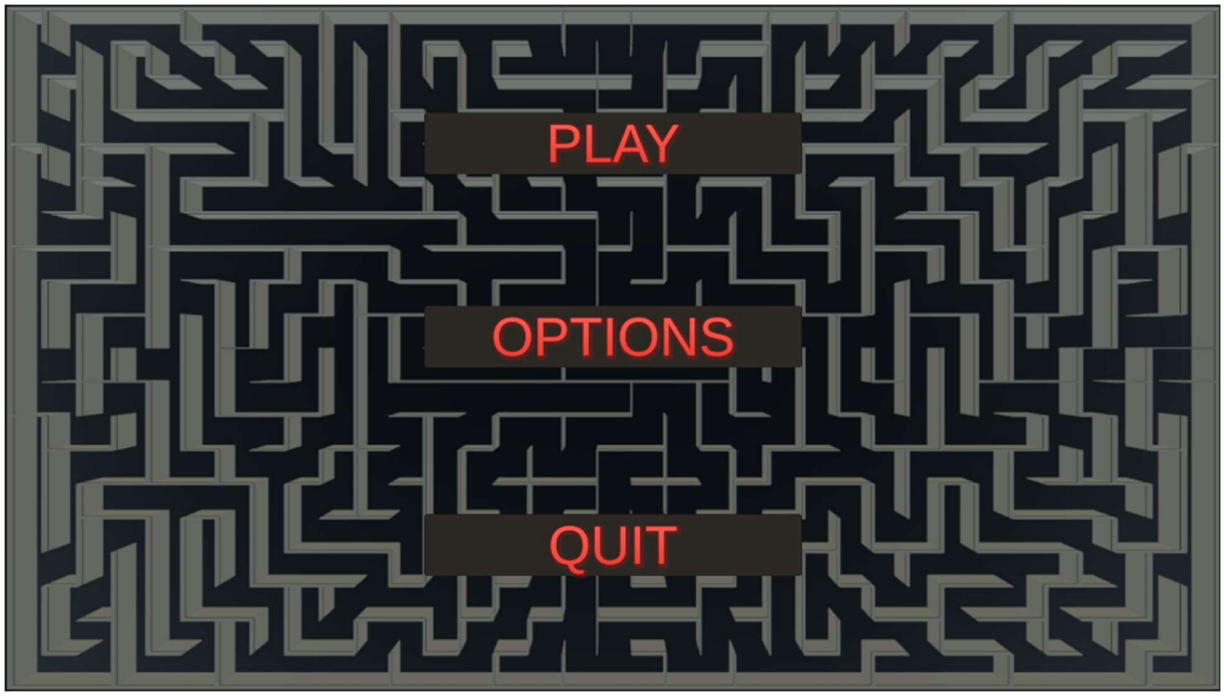

Figure 32

10.4 Camera Look

```
void Update()
{
    if (escapeMenuController != null && escapeMenuController.escapeMenu.activeSelf)
    {
        // Don't move the camera if the Escape Menu is active
        return;
    }

    float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity * Time.deltaTime;
    float mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity * Time.deltaTime;

    xRotation -= mouseY;
    xRotation = Mathf.Clamp(xRotation, -90f, 90f);

    transform.localRotation = Quaternion.Euler(xRotation, 0f, 0f);
    playerBody.Rotate(Vector3.up * mouseX);
}
```

Figure 33

The code checks if the escape menu is active, and if it is, the camera wont move. Otherwise, it retrieves the input from the mouse to control the cameras movement. The mouse input is multiplied by a sensitivity factor and the time that has elapsed since the last frame to create a smooth camera movement. The code then clamps the camera's vertical rotation to prevent it from rotating too far up or down. Finally, the code updates the camera's local rotation based on the vertical rotation and rotates the player's body based on the horizontal rotation.

11 YAML File

A YAML [11] file is a configuration file that specifies various parameters and settings for training ML-Agents. These parameters are essential to customising the training process and achieving the desired results. A YAML file defines the path to the Unity environment that the ML-Agents will train on. This allows the training process to interface with the Unity engine and receive observations and actions from the environment.

The YAML file also defines the type of training algorithm that will be used to train the ML-Agents. There are several training algorithms available in ML-Agents, including PPO, SAC and DDPG, each with their unique advantages and disadvantages. The YANL file also allows for the customisation of hyperparameters such as learning rate, batch size and number of training iterations, which can significantly impact the training process efficiently.

Here are some of the hyperparameters used in this project:

- batch_size: This is the number of experiences that the agent collects before updating its parameters. A larger batch size can lead to more stable training but also requires more memory and computational resources.
- buffer_size: This is the number of experiences that the agent stores in its replay buffer. The reply buffer is used to sample experiences for training. A larger buffer size can lead to better sampling of experiences, but again , requires more memory.
- learning_rate: This hyperparameter controls how much the agent updates its parameters based on the gradient of the loss function. A smaller learning rate can result in slower learning but can help the agent avoid overshooting the optimal solution.

12 Training Results

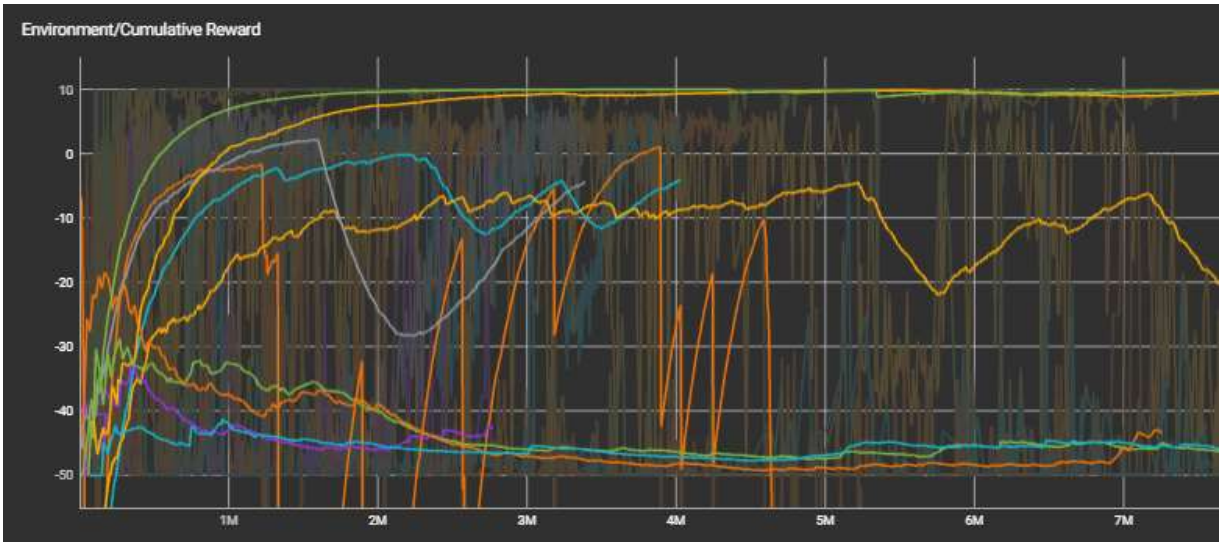

Figure 34

These are the results of the different training scenarios of the ML-Agents, each unique in their own way. The graph above is the cumulative rewards of the agents while they train. From the graph, it is evident that the agents were able to learn quickly and efficiently how to reach the goal in most of the test cases. They received high rewards indicating successful completion of the task. However, there were some scenarios where the agents failed to reach the goal and returned to receiving negative rewards. This is a major challenge that I face while training the agents, as they need to explore the maze while avoiding getting stuck in dead ends.

To overcome this challenge, I modified the environment to optimize the agents learning process. Starting with a 3x3 and increasing the complexity of the maze as they learn. This allowed the agents to adapt to new environments and learn how to explore and exploit more efficiently.

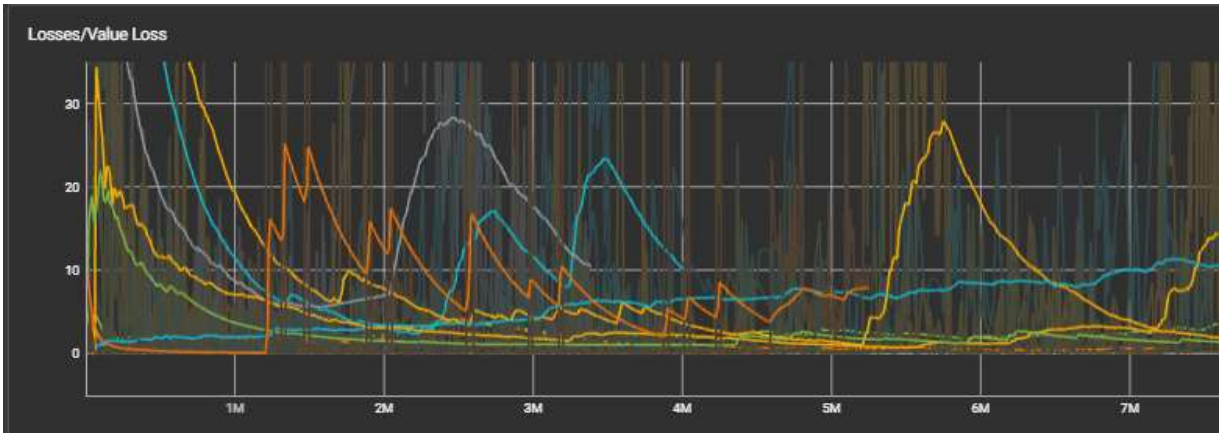

Figure 35

This graph correlates with the graph from earlier, this graph plots the value loss. From this graph it is evident that some of the test cases were successful in reducing the number of rewards lost. This means that the agents were avoiding walls and moving around them to reduce chances of losing rewards and reach the goal successfully.

Here is a graph of a single test case:

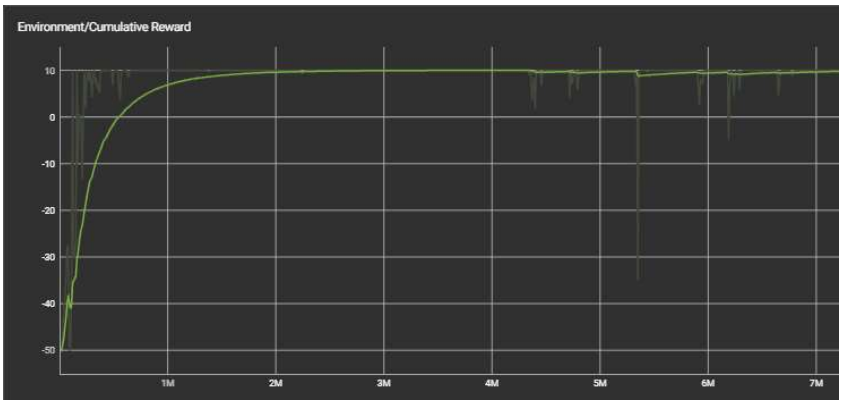

Figure 36

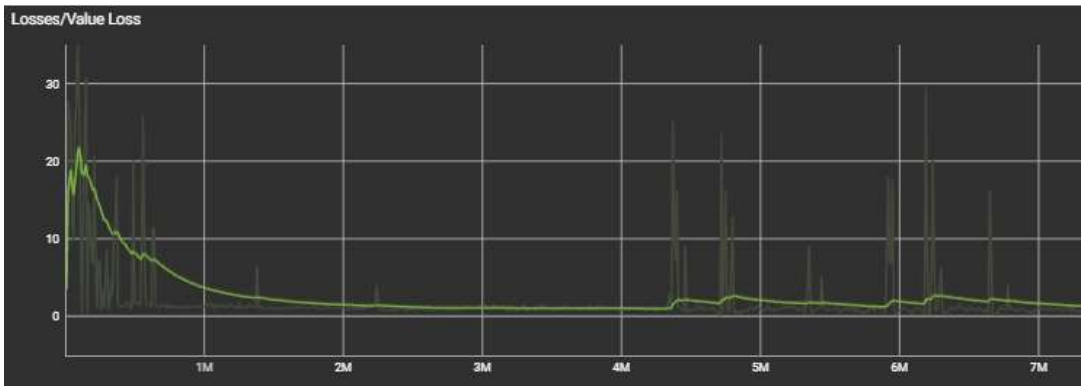

Figure 37

These graphs were plotted using TensorFlow. With TensorFlow in my virtual environment, I can easily be able to plot diagrams like the ones above to showcase the different training tests I have done and visualise how the agents are training. This allowed me to make changes to my code to respond to some of the failed test cases. TensorFlow is an excellent tool I added to my virtual environment, and it has helped a lot.

Another graph which showcases the learning rate of the agents is a graph that plots the episode length of a specific test case. In this graph the episode length being reduced to a low value, meaning that the agents are learning as efficiently as they can and are finding the goal much easier. Finding the goal faster reduces the episode length which created this graph:

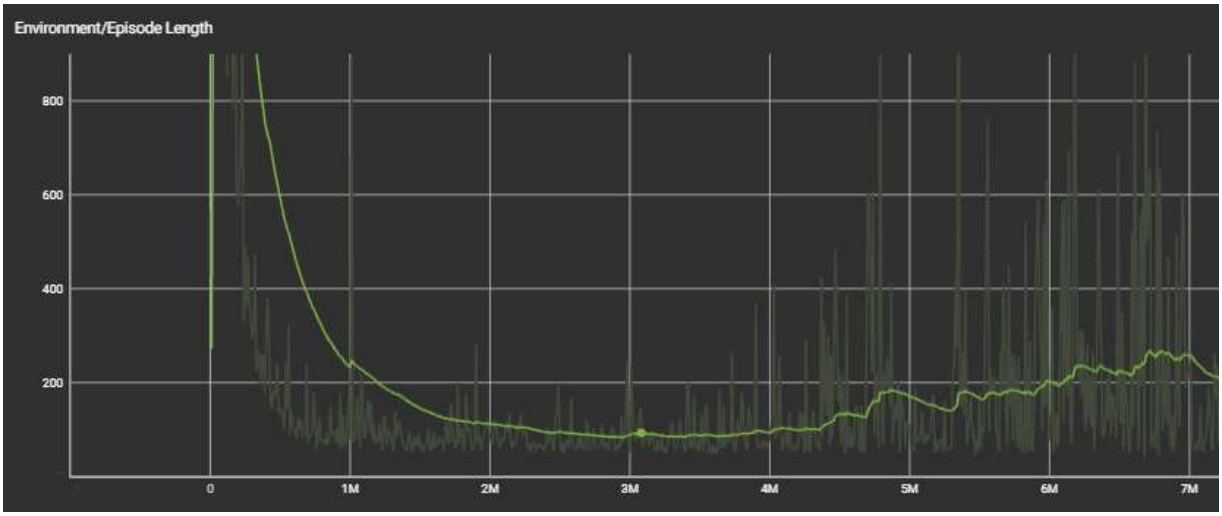

Figure 38

13 Blender

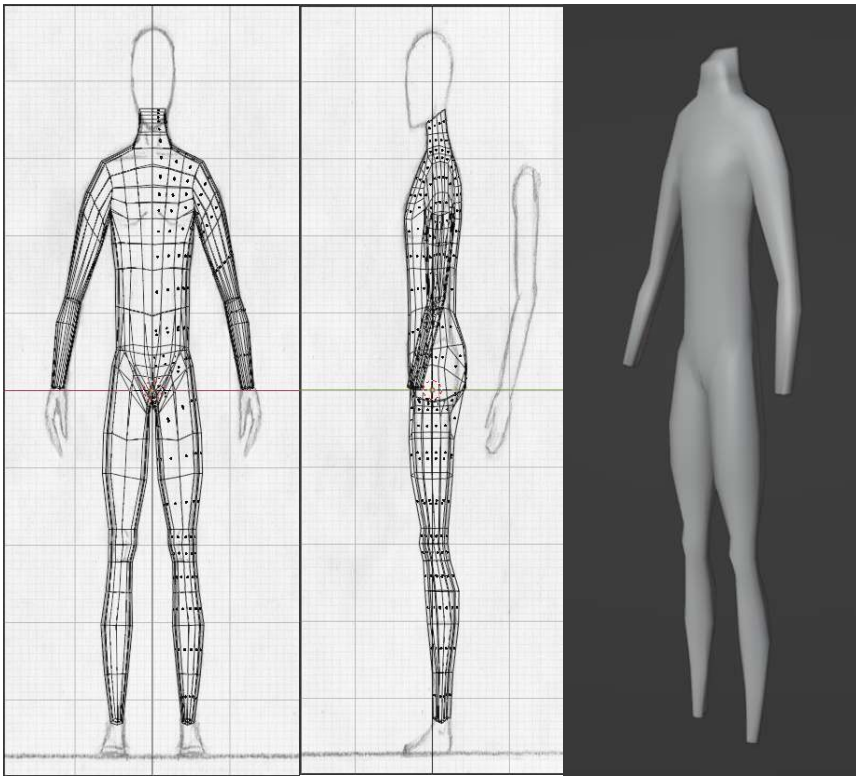

Figure 39 [12]

Using a sketch of a human body for reference I sculpted this model using Blender.

14 Ethics

14.1 Privacy

It is important to ensure that the data used to train the ML-Agents is collected and used in an ethical and transparent manner. Data Privacy laws and regulations must be taken into consideration when collecting and storing data. The use of anonymized data and obtaining consent from individual can also help address data privacy concerns.

15 Conclusion

In conclusion I have the agents successfully navigating a 6x6 maze. Since the maze is random every time, it creates situations which are far too complex for the agent to learn how to navigate a maze of a larger size in time. The game is fully functional, and the player and agent can play against each other.

In the future I hope to train the agents for a sufficient amount of time, so they are able to navigate mazes of higher sizes like 15x15 or a 20x20. Randomly generated mazes of this size are incredibly complex, and the agents would have near perfect decision making to navigate efficiently .

References

[1] "Unity ML-Agents Toolkit.," 3 5 2023. [Online]. Available: https://unity.com/products/machine-learning/ml-agents.

[2] Unity, "Unity.com," 3 5 2023. [Online]. Available: https://unity.com/.

[3] "Blender," Blender, 3 5 2023. [Online]. Available: https://www.blender.org.

[4] "Ray Perception Sensor.," 3 5 2023. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.ml-agents@1.9/manual/RayPerceptionSensorComponent.html.

[5] "Maze Generation Algorithms.," 3 5 2023. [Online]. Available: https://en.wikipedia.org/wiki/Maze_generation_algorithm.

[6] CodeMonkey, "Youtube," CodeMonkey, 3 5 2023. [Online]. Available: https://www.youtube.com/watch?v=zPFU30tbyKs.

[7] "Deep Reinforcement Learning," 3 5 2023. [Online]. Available: https://en.wikipedia.org/wiki/Deep_reinforcement_learning.

[8] "ekababisong," The Exploration-Exploitation Trade-off, 3 5 2320. [Online]. Available: https://ekababisong.org/the-exploration-exploitation-trade-off/.

[9] "C# Programming Guide.," 3 5 2023. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/.

[10] Brackeys, "Youtube," Brackeys, 3 5 2023. [Online]. Available: https://www.youtube.com/@Brackeys.

[11] YAML, "Unity ML-Agents," 3 5 2023. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/Training-Configuration-File.md.

[12] S. Lague, "Youtube," 3 5 2023. [Online]. Available: https://www.youtube.com/watch?v=aAO4C_8y0w8.

[13] O. Santiago, "Youtube," 3 5 2023. [Online]. Available: https://www.youtube.com/watch?v=-vgis1SSIDA.