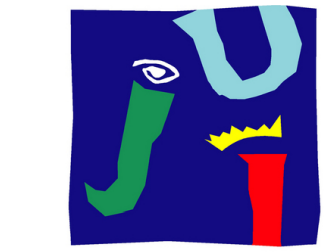

UNIVERSITAT JAUME·I

Integration of Machine Learning in the Tower Defense subgenre of video games

Iván Gomis Moreno

Final Degree Work
Bachelor's Degree in
Video Game Design and Development
Universitat Jaume I

June 6, 2024

Supervised by: Jorge Sales, PhD.

To those who believed in me

Acknowledgments

First I want to thank my parents for supporting me, without them I would not have made it this far. To my friends who have already graduated and still have accompanied me on this journey, especially to Jorge, Olga and Sabina.

I would also like to thank Jorge Sales for his supervision of this project.

I also want to thank my partner, Laia, she has helped me in the translation of this project and has reminded me why I should keep working hard.

I also would like to thank Sergio Barrachina Mir and José Vte. Martí Avilés for their inspiring LaTeX template for writing the Final Degree Work report, which I have used as a starting point in writing this report.

Abstract

This document presents the Final Degree Work report of Iván Gomis Moreno in Bachelor's Degree in Video Game Design and Development.

The main concept is making a tower attack video game using Unity, employing the various systems that this game engine provides for the creation of non-playable characters, specifically, ML Agent Libraries. The AI implementation will possess the capability to adapt and learn based on its current conditions and interactions with the player, enabling it to respond to specific events variably, thus achieving unpredictable behaviours.

Furthermore, several behaviours, actions, and systems developed for the enemy's creation can be repurposed across various video game genres.

Keywords

Unity, artificial intelligence, tower defense, video game, ML-Agent

Contents

Contents **v**

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**
1.1 Work Motivation . . . 1
1.2 Objectives . . . 2
1.3 Environment and Initial State . . . 3

2 Planning and resources evaluation **5**
2.1 Planning . . . 5
2.2 Resource Evaluation . . . 7

3 System Analysis and Design **11**
3.1 Requirement Analysis . . . 11
3.1.1 Functional Requirements . . . 12
3.1.2 Non-functional Requirements . . . 12
3.2 System Design . . . 13
3.3 System Architecture . . . 17
3.4 Interface Design . . . 17

4 Work Development and Results **21**
4.1 Work Development . . . 21
4.1.1 First Steps: Creating the assets . . . 21
4.1.2 Adding animations and camera . . . 24
4.1.3 Basic setup . . . 27
4.1.4 Player mechanics . . . 30
4.1.5 AI design . . . 34
4.1.6 AI behaviour . . . 35
4.1.7 AI implementation . . . 37
4.2 Results . . . 42

5 Conclusions and Future Work **45**
5.1 Conclusions 45
5.2 Future work 46

Bibliography **47**

List of Figures

2.1 Gantt chart depicting the initial project planning 9
2.2 Gantt chart depicting the tasks performed in the project 10

3.1 Final HUD visualisation 18
3.2 Final create unit HUD visualisation 19
3.3 Final sell unit HUD visualisation 19

4.1 Modelled units 22
4.2 Hand-drawn video game map 23
4.3 Final result of the environment 24
4.4 Assault Mech Bone Renderer component 25
4.5 Two Bone IK Constraint component for left Assault Mech leg 25
4.6 Artillery Hammer Foot Solver script 26
4.7 Scene CameraControls script 27
4.8 Unit Tank Script 27
4.9 Unit Behemoth Script 28
4.10 Behemoth Bullet Script 29
4.11 Enemy base Health System Script 30
4.12 Map with stabilised nodes. Nodes can be seen because of its Collider Box . . 30
4.13 Map with established nodes. Nodes can be seen because of its Collider Box . 31
4.14 Map with path nodes functionality 32
4.15 Tank in the convoy destroying minerals for currency 33
4.16 Harvester collecting minerals for currency 36
4.17 Visualisation of the debug *gridSystem* 37
4.18 Visualisation of valid grid positions to which the AI could build a turret . . . 38
4.19 AIBehavior Behavior Parameters component 39
4.20 AIBehavior Grid Sensor component 40
4.21 Visualisation of the Grid Sensor component during runtime 41

List of Tables

2.1 Total project cost 8

3.1 Case of use «CU01. Camera movement» 13
3.2 Case of use «CU02. Camera zoom» 14
3.3 Case of use «CU03. Path generation» 14
3.4 Case of use «CU04. Convoy menu opening» 15
3.5 Case of use «CU05. Buy Units» 15
3.6 Case of use «CU06. Sell Units» 16
3.7 Case of use «CU07. Agent intelligence» 16
3.8 Case of use «CU08. Reaction to stimulus» 17

CHAPTER 1

INTRODUCTION

Contents

1.1 Work Motivation **1**
1.2 Objectives **2**
1.3 Environment and Initial State **3**

This chapter shows the main motivations for the development of this project, as well as the objectives to be achieved and its first stage of development.

1.1 Work Motivation

Strategy games are a type of video game genre that requires the player's planning and critical thinking to achieve victory. Players typically control a group of units or structures to defeat an enemy or complete an objective. Strategy game subgenres include real-time strategy (RTS) games, turn-based strategy (TBS) games and tower defence (TD) games, among many others.

Specifically, tower defence games are a subgenre of real-time strategy games in which players build towers to defend a base from enemy attacks. Enemies are usually waves of units advancing towards the player's base. Players must use their towers to destroy the enemies before they reach the base. Strategy game mechanics are usually simple to learn but difficult to master. The reason for this is that players must take into account several factors when building their towers, such as the type of enemy, the attack range of the tower, and the cost of construction. Most importantly, they must also manage their resources effectively so that they can build enough towers to defend the base.

The TD market is a growing market. On a global scale, it is expected that this market/the TD market will reach 2,311.3 million dollars by 2030, and is growing at a compound annual growth rate (CAGR) of 10.38 [1].

The growth of the TD market is due to several factors, including:

- The popularity of mobile gaming. TDs are a type of genre that is well suited to mobile devices, as they are simple to play with one hand.
- The growing popularity of free-to-play games. Many TD games are free-to-play, making them more accessible to a wider audience.
- Innovation in the genre. TD developers are constantly innovating in the genre, adding new mechanics and features to attract new players.

Thus, due to its great importance and to avoid researching more libraries during the development of this project, it will be developed with Unity for PC. As can be foreseen with the above-mentioned, the project will consist of the realisation of a tower defence type game. I will be employing Artificial Intelligence (AI) as a disruptive element, which holds significant market potential within the world of PC games. In addition, I intend that this integration adds value to the tower defence games dominating the market.

This will be achieved by shifting the traditional dynamic of the genre, making the player the one in command of controlling and managing the waves of units attacking the base and the enemies. On the other hand, the aim of the enemies is to protect the base by placing defences. These enemies will be controlled by an AI developed with machine learning, which will incorporate the player's previous game records into its database. Hence, allowing the AI to devise new and challenging strategies for the player. Finally, in regards to the game concept, this project has drawn inspiration from the following games:

- Bloons TD 6[1]: It includes a player vs. player mode in which besides having to defend against waves of balloons heading to the base, you can also use your economy points in sending extra balloons to your opponent.
- Anomaly 2[2]: As in this proposal, the game consists of directing a convoy of vehicles through a route of previously placed towers while trying to accomplish the objectives of the mission.

1.2 Objectives

Taking into account the driving forces behind this project's development, here are the objectives I aim to accomplish with this endeavour:

[1]Bloons TD 6 is a 2018 tower defence game developed and published by Ninja Kiwi, where various monkeys pop "bloons" [26].

[2]Anomaly 2 takes the RTS tower-offence concept from Anomaly Warzone Earth to a new level. The core elements of the original (tactical planning and the on-field Commander to support troops in combat) are spiced up by a number of important new features [23].

- Improve the knowledge of the Unity game engine.
- Gain proficiency in utilising the diverse range of systems offered by Unity for the implementation of artificial intelligence, with a focus on learning and comprehension.
- Apply artificial intelligence techniques to craft an agent characterised by unpredictability.
- Make the agent increase difficulty through games.
- The agent reacts to stimulus that it perceives through his perception.
- Grasp the concepts of machine learning and explore its practical applications within the context of my project.
- Learn how to implement machine learning in Unity.

1.3 Environment and Initial State

One of the things I was most concerned about when I started this project was the implementation of the AI tools provided by Unity. Throughout my studies I have been taught the basic functions of Unity and how to implement various techniques, structures and algorithms in it. During the Artificial Intelligence course I was also taught how to implement basic AI creation algorithms, but never with the complexity I expected in this project. So, I started looking for documentation and tutorials about Machine Learning in Unity, and to my surprise, the youtuber and developer CodeMonkey had a great variety of tutorials on how to properly use Unity's ML-Agent library [20]. Tutorials found on Unity's official site also helped me during the development of the project [24].

But first of all, I really needed to find out what Machine Learning is and how it works in general terms. To do so, I went to the university library hoping to find a book that would satisfy my needs. Doing a quick search on the web I found a book that caught my attention: *Inteligencia Artificial Fácil: Machine Learning y Deep Learning Prácticos* [25]. This book teaches, with practical cases using python how Machine Learning works.

Armed with ample learning resources on Machine Learning and ML-Agent libraries, I've chosen to broaden the scope of my AI creation concept. My initial concept was based on the video games Starcraft II (Blizzard Entertainment, 2010)[3], Company of Heroes

[3]StarCraft II is a military science fiction video game created by Blizzard Entertainment as a sequel to the successful StarCraft video game released in 1998. Set in a fictional future, the game centres on a galactic struggle for dominance among the various fictional races of StarCraft [32].

2 (Relic Entertainment, 2013)[4] and Supreme Commander 2 (Square Enix, 2010)[5]. In these games, the enemy AI tries to find the player's location and then obliterate him. In Supreme Commander 2 you are able to choose the spawn location, and change the strategies to prevent the AI from destroying you. The positive aspect of this AI is that it is quite good at adapting the player strategies [22]. On the other hand, in my other two references there's no unit commander and AIs are developed with behaviour trees. The mechanics are well thought out in both games although they have some problems in providing a real challenge to the player, because the AI is reasonably predictable.

[4]Company of Heroes 2 is a real-time strategy video game developed by Relic Entertainment and published by Sega for Windows, Linux, and OS X. It is the sequel to the 2006 game Company of Heroes. As with the original Company of Heroes, the game is set in World War II but with the focus on the Eastern Front, with players primarily controlling the side of the Soviet Red Army during various stages of the Eastern Front, from Operation Barbarossa to the Battle of Berlin [27].

[5]Supreme Commander 2 is a real-time strategy (RTS) video game developed by Gas Powered Games and published by Square Enix as the sequel to Supreme Commander [33].

CHAPTER 2

Planning and resources evaluation

Contents

2.1 Planning 5
2.2 Resource Evaluation 7

This chapter shows the planning for the project as well as the software programmes and other resources used.

2.1 Planning

At the project's outset, my understanding of Machine Learning was limited, prompting me to base my initial planning on rough estimations. We are not talking about the development of an entire video game, we are talking about a technical demonstration[1]. To complete this task, we need to model assets, create sprites, and integrate algorithms for player interaction, ultimately achieving a playable demo. However, the specified tasks diverged from the final ones, as some algorithms I intended to implement were already available in Unity systems. Coupled with the need to research and study a new library and encountering additional challenges, the workload for this task has exceeded the initially estimated hours. Tasks were often executed concurrently rather than sequentially.

[1] A technical demonstration (or technical demo), also known as demonstrator model, is a prototype, rough example or otherwise incomplete version of a conceivable product or future system, put together as proof of concept with the primary purpose of showcasing the possible applications, feasibility, performance and method of an idea for a new technology. They can be used as demonstrations to the investors, partners, journalists or even to potential customers in order to convince them of the viability of the chosen approach, or to test them on ordinary users [34].

- **Information gathering and study (40 hours)**: search for research articles to enhance my understanding of AI. Additionally, I plan to enrol in Machine Learning courses and explore for tutorials and information available to familiarise myself with ML-Agent libraries.
- **Creation of the assets (39 hours)**:
  - **Assault Mech (3 hours)**: this ally unit must give the impression of high agility and high firing capacity.
  - **Artillery Hammer (4 hours)**: an artillery vehicle modified to have a 360º rotation and long range. As a consequence, it has sacrificed much of its armour.
  - **Heavy Tank (3 hours)**: a tank that sacrifices part of its firepower to obtain powerful armour.
  - **Blaster (3 hours)**: a relatively small basic enemy tower that alone does not represent a major danger, but can be a challenge in large groups.
  - **Storm Reaper (2 hours)**: a tower that launches lightning bolts that strike a target and spread to up to two extra targets.
  - **Behemoth (3 hours)**: a highly armed tower that attacks with strong energy pulses that can damage several units within range.
  - **Harvester (4 hours)**: a tower that extracts minerals from the underground for 60 seconds. Then, it is destroyed.
  - **Mineral Collector (1 hour)**: a mineral object that ally units can shoot to grab.
  - **Map (6 hours)**: the game environment where the units will move through.
  - **Heads-Up Display (HUD) (4 hours)**: the HUD has an user-friendly design. The player can access the convoy creation menu through the main button. In this menu, the player can control the management of the convoy units.
  - **Visual Effects (6 hours)**: this includes explosions and bullet effects that have been realised with Unity's particle system.
- **Mechanics concept design (2 hours)**: specify the rules and actions that players will have the ability to execute.
- **Mechanics design (45 hours)**:
  - **Path Development (17 hours)**: creation of an undirected graph and an algorithm for the convoy units to move through it.
  - **Animate units (6 hours)**: use of dynamic animation to animate the movement of the units.

  - **Units mechanics (8 hours)**: creation and programming of a health, aiming and firing system for towers and allied units.
  - **Convoy Creation (8 hours)**: creation and programming of a HUD with which units can be purchased and added to the convoy and then sold.
  - **Economy (6 hours)**: programming of an economy system where the player can obtain resources by destroying towers and mineral collectors.

- **AI Concept design (15 hours)**: outline the objectives and prerequisites for the AI, along with identifying suitable systems and algorithms for implementation.
- **AI design (32 hours)**:
  - **AI Grid (12 hours)**: creation and programming of a mesh capable of detecting units, obstacles and towers built to facilitate AI decision making.
  - **AI Economy (4 hours)**: programming of an economy system where the AI can obtain resources by using Harvesters.
  - **AI Observations (35 hours)**: programming the various responses that the agent will have to know what is happening in the environment (convoy position and turrets built).
  - **AI Actions (45 hours)**: programming AI reactions to allow the AI to build turrets when it is necessary.
  - **Training system (35 hours)**: allow the AI train to generate a neural network capable of beating the player.

2.2 Resource Evaluation

There are many different types of professional profiles in the video game industry, and salaries can vary greatly based on an employee's level of expertise. After a brief check on job search portals, I discovered that a junior video game programmer typically makes 21,500.00 € gross annually, or 1,254.00 € net. Given that the project was produced over a four-month period, the approximate total cost would be 5,016.00 € net (see table 2.1).

Besides from the salary, the cost of equipment should also be included in the total cost of production. The laptop was purchased at approximately 800.00 €. I include a detailed list of the laptop components, despite its ageing components, the tasks were completed successfully.

- **Model**: Lenovo Legion Y520-15IKBN
- **CPU**: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
- **GPU**: NVIDIA GeForce GTX 1050 4G GDDR5
- **RAM**: 16GB DDR4

- **Display**: 15.6" 39.6 cm, 1920x1080 pixels FHD IPS AG (SLIM)

Lastly, the cost of the software totals 0.00 € as the programmes are either free or available as a student version.

- **Unity**: the game engine used for the development of the project.
- **3DS Max 2023**: computer software used for the modelling of the assets.
- **Krita**: free and open-source raster graphics editor used to make UI elements.
- **Jira**: a web page designed to facilitate task management, aiding in the organisation of project activities [2].
- **GanttProject**: an open-source project management application utilised for generating Gantt diagrams to aid in project development [16].
- **Google Docs**: an online word processor enabling document creation and formatting. Employed for initial project submissions such as the technical proposal and Game Design Document (GDD).
- **Overleaf**: online LaTeX editor that has been used for the realisation of this report [21].

Software Costs	
Development Software	0.00 €
Hardware Costs	
Laptop	800.00 €
Other Production Costs	
Salary	5,016.00 €
Total	**5,816.00 €**

Table 2.1: Total project cost

Task name	Start date	Finish date	Status	Approximate Hours Employed	Hours used
Final degree project preparation	**08.12.2023**	**01.02.2024**	**Finish**	**14h about**	**10**
Technical proposal	08.12.2023	09.12.2023	Finish	4h about	2
GDD	24.01.2024	01.02.2024	Finish	10h about	8
Asset creation	**02.01.2024**	24.04.2024	**Finish**	**32h about**	**33**
Assault Mech	02.01.2024	02.01.2024	Finish	3h about	3
Artillery Hammer	03.01.2024	03.01.2024	Finish	3h about	4
Heavy Tank	03.01.2024	03.01.2024	Finish	3h about	3
Blaster	07.01.2024	07.01.2024	Finish	3h about	3
Storm Reaper	07.01.2024	07.01.2024	Finish	3h about	2
Behemoth	07.01.2024	07.01.2024	Finish	3h about	3
Harvester	22.04.2024	22.04.2024	Finish	3h about	4
Mineral Colector	01.04.2024	01.04.2024	Finish	1h about	1
Map	13.01.2024	28.01.2024	Finish	5h about	6
HUD	30.03.2024	30.03.2024	Finish	5h about	4
Research on Machine Learning in video games	**01.12.2023**	03.05.2024	**Finish**	**40h about**	**40**
Lecture of Machine Learning book	01.12.2023	28.12.2023	Finish	25h about	20
Watch videos and tutorials of ML-Agent	29.01.2024	03.05.2024	Finish	15h about	20
Player gameplay development	31.01.2024	17.04.2024	Finish	**52h about**	**62**
Import assets and make animations	31.01.2024	04.02.2024	Finish	6h about	6
Health, traking and shooting of turrets and vehicles	07.02.2024	29.02.2024	Finish	*8h about*	8
Explosion and bullets effect	05.02.2024	13.02.2024	Finish	*8h about*	6
Path development	13.02.2024	14.03.2024	Finish	*14h about*	17
Convoy creation	15.03.2024	26.03.2024	Finish	6h about	19
Economy	01.04.2024	01.04.2024	Finish	4h about	2
HUD implementation	31.03.2024	31.03.2024	Finish	6h about	4
AI implementation	**19.04.2024**	10.05.2024	**Finish**	**80h about**	**80**
Grid for turrets	19.04.2024	22.04.2024	Finish	12h about:	3
Harvester economy	*22.04.2024*	22.04.2024	*Finish*	*4h about*	*2*
ML-Agent Implementation	*23.04.2024*	26.04.2024	*Finish*	*16h about*	*15*
AI training	29.04.2024	03.05.2024	Finish	24h about:	50
AI incorporation	*06.05.2024*	*08.05.2024*	*Finish*	*16h about*	*4*
Player training	*08.05.2024*	10.05.2024	*Finish*	*8h about*	6
User test #1	11.05.2024	12.05.2024	**Finish**	**10h about**	**10**
Feedback app design	11.05.2024	12.05.2024	Finish	5h about	5
Error and bug feedback	11.05.2024	12.05.2024	Finish	5h about	5
Revision #1	13.05.2024	15.05.2024	**Finish**	**16h about**	**16**
Troubleshoot layout #1	13.05.2024	13.05.2024	Finish	8h about	8
Troubleshoot programming #1	14.05.2024	15.05.2024	Finish	8h about	8
User test #2	18.05.2024	19.05.2024	**Finish**	**10h about**	**13**
Feedback app design	18.05.2024	19.05.2024	Finish	5h about	6
Error and bug feedback	18.05.2024	19.05.2024	Finish	5h about	7
Revision #2	20.05.2024	22.05.2024	**Finish**	**16h about**	**10**
Troubleshoot layout #2	20.05.2024	20.05.2024	Finish	8h about	5
Troubleshoot programming #2	21.05.2024	22.05.2024	Finish	8h about	5
Prepare Presentation	06.04.2024		**In progress**	**30h about**	**30**
Powerpoint	16.05.2024		Open	5h about	
Project Memory	06.04.2024	15.05.2024	Close	25h about	30

Total hours used	**Total real hours used**
300h about	304

Figure 2.1: Gantt chart depicting the initial project planning

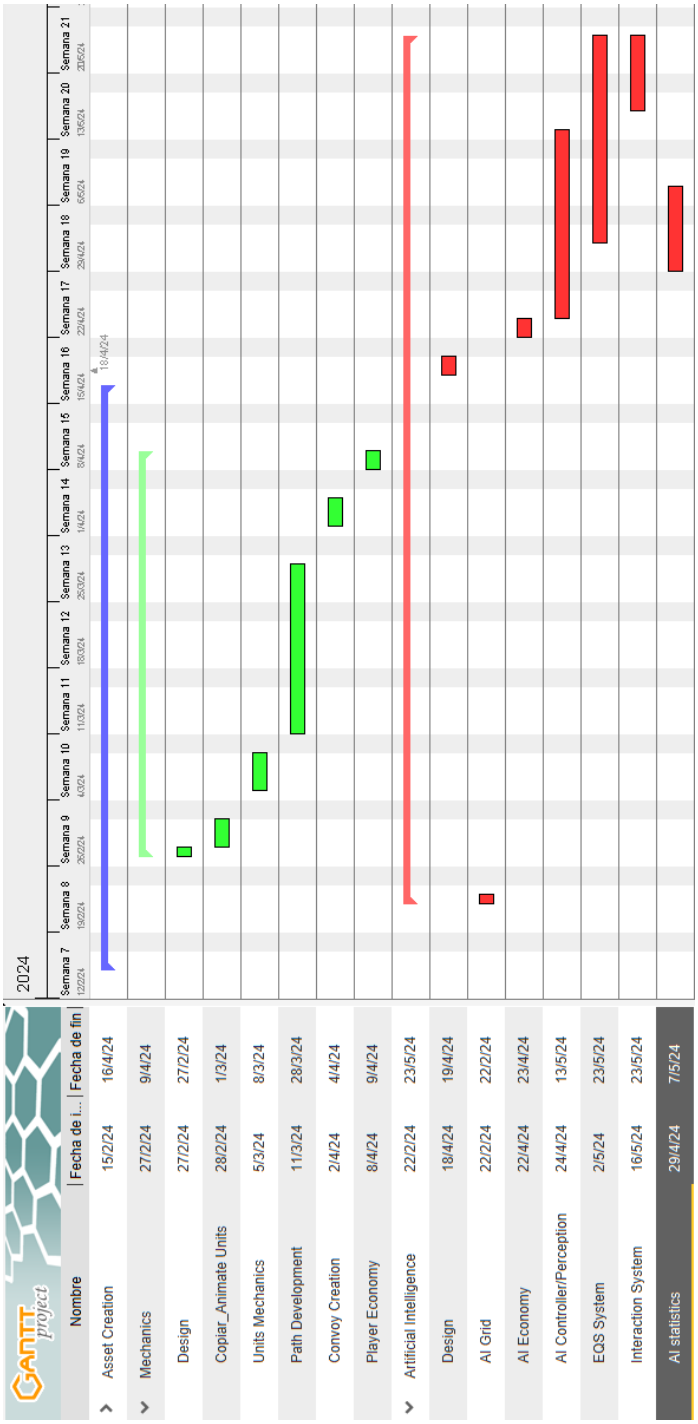

Figure 2.2: Gantt chart depicting the tasks performed in the project

CHAPTER 3

System Analysis and Design

Contents

3.1 Requirement Analysis **11**
3.2 System Design **13**
3.3 System Architecture **17**
3.4 Interface Design **17**

Chapter 3 presents the requirements analysis, design and architecture of the proposed work, as well as, where appropriate, its interface design.

3.1 Requirement Analysis

As previously discussed, the objective of the artificial intelligence under development is to attain dynamic behaviour, imparting a sense of learning and adaptation as the game unfolds.

For this project, the player can move the camera around the environment with the WASD keys and move the cursor at the edge of the screen. The player may press a create convoy button or C key at any time to open the convoy menu. The player can also press a pause button or P key at any time to open the options menu. The player is able to click on the arrows indicating the convoy's path to change it. The player can press the buy button to select any affordable unit. The player can also press any created unit button to sell it.

On the other hand, the agent who will take on the role of main enemy will have a dual purpose. First, to protect their bases so that the player's convoy does not destroy them. To do this, the agent will be able to see which units the player has built, to build

the counter turrets. The position where these towers are established is also important due to the existence of obstacles that act as cover. The next objective will be to cover 75% of the map with towers. As a result, we give the player a sense of urgency that prevents the camping. The agent will have to prioritise building as many turrets as possible in the shortest time possible to win.

The agent has a series of statistics that will dictate which will be its behaviour. These statistics are: Mineral Collected, Position of the convoy and Turrets built. The agent's interactions with various elements of the game will vary depending on the values of specific statistics. However, irrespective of its current state or statistics, when the agent is low on resources, its top priority will be to construct a Harvester to bolster its economy. In case he does not have enough minerals to build the Harvester, he will have to wait passively to have the necessary minerals to continue building turrets.

3.1.1 Functional Requirements

A functional requirement defines a function of the system that is going to be developed. This function is described as a set of inputs, its behaviour, and its outputs [28]. The functional requirements may be: calculations, technical details, data manipulations and processes, and any other specific functionality that defines what a system is supposed to achieve. Let's examine the project's requirements:

- **R1**. The player can move the camera around the environment.
- **R2**. The player can zoom in and zoom out the camera.
- **R3**. The player can interact with path arrows clicking on them.
- **R4**. The player is able to open the convoy menu by pressing the convoy button or C key.
- **R5**. The player can buy convoy units.
- **R6**. The player will be able to sell units.
- **R7**. The agent will be able to build turrets.
- **R8**. The agent can react to stimuli from the environment.

3.1.2 Non-functional Requirements

Non-functional requirements impose conditions on the design or implementation (e.g., to meet performance, safety, or reliability constraints). Some of the non-functional requirements shown are accessibility, backup policy, certification, documentation, extensibility, interoperability, open source code or not, performance, compatibility between platforms, portability, quality, scalability, and usability [29].

- **R9**. Communication between scripts must be efficient through the use of inheritance.
- **R10**. The use of design patterns is essential for the development of the project.
- **R11**. Artificial Intelligence will consistently execute the anticipated actions.
- **R12**. The code must be reusable for other video games in the industry.

3.2 System Design

This section outlines the logical and operational design of the system to be implemented. Subsequent pages detail the use cases for both the player and the artificial intelligence agent.

Requirement:	R1
Actor:	Player
Description:	Each time the player presses one of the movement keys, the camera can move around the environment by pressing W, A, S or D.
Preconditions:	1. The player must not be in the convoy menu. 2. The player must have units on the field.
Normal sequence:	1. The player presses W, A, S or D keys. 2. The camera moves in the direction assigned to the key.
Alternative sequence:	None.

Table 3.1: Case of use «CU01. Camera movement»

Requirement:	R2
Actor:	Player
Description:	Each time the player rolls the middle button of the mouse, the camera zooms in and zooms out.
Preconditions:	1. The player must not be in the convoy menu. 2. The player must have units on the field.
Normal sequence:	1. The player rolls in the middle button of the mouse. 2. The camera approaches the environment.
Alternative sequence:	2.1. The camera moves away from the environment if the player rolls out the middle button of the mouse.

Table 3.2: Case of use «CU02. Camera zoom»

Requirement:	R3
Actor:	Player
Description:	Each time the player presses one of the path arrows, it cycles through the different available paths.
Preconditions:	1. The player must not be in the convoy menu. 2. The player must have units on the field. 3. It is necessary that the nearby roads are not part of the main road and are not the path from which you are coming.
Normal sequence:	1. The player presses one of the arrows placed on the environment. 2. The arrow changes its direction. 3. From the point where the arrow places, a new path is generated.
Alternative sequence:	None.

Table 3.3: Case of use «CU03. Path generation»

Requirement:	R4
Actor:	Player
Description:	The player can open the menu convoy by pressing convoy button or by pressing C key.
Preconditions:	1. The player must not be in the convoy menu.
Normal sequence:	1. The player presses the convoy button or C key. 2. The game pauses. 3. The menu convoy opens.
Alternative sequence:	2.1. In the case that the menu convoy is open, it closes.

Table 3.4: Case of use «CU04. Convoy menu opening»

Requirement:	R5
Actor:	Player
Description:	Player can press the buy button and select which unit is going to be built.
Preconditions:	1. The player must be in the convoy menu. 2. The player must have less than 6 units on the field. 3. The player must have enough money to afford at least one of the buildable units.
Normal sequence:	1. The player presses the buy button. 2. The player presses the button of the unit which he wants to be built. 3. The buttons of the convoy are updated. 4. When the player exits the convoy menu, the unit bought will be built.
Alternative sequence:	None.

Table 3.5: Case of use «CU05. Buy Units»

Requirement:	R6
Actor:	Player
Description:	Player can press any unit button and sell that unit to have a refund.
Preconditions:	1. The player must be in the convoy menu. 2. The player must have units on the field.
Normal sequence:	1. The player presses any unit button. 2. The player presses the sell button. 3. The buttons of the convoy are updated. 4. When the player exits the convoy menu, the unit sold will be destroyed.
Alternative sequence:	None.

Table 3.6: Case of use «CU06. Sell Units»

Requirement:	R7
Actor:	Agent
Description:	The agent is capable of building turrets.
Preconditions:	1. The agent must have a ML-Agent attached and configured. 2. The selected position must be within the grid volume. 3. The selected position must not be in a position where another turret, obstacle, road or mineral is placed. 4. The agent must have enough money to afford at least one of the buildable turrets.
Normal sequence:	1. The agent's ML-agent component is started at the beginning of the game. 2. The agent decides the position of the next turret. 3. The agent decides which turret is going to be built. 4. The agent builds the selected turret.
Alternative sequence:	3.1. In case that the agent does not have enough money, the next turret to build will be a Harvester.

Table 3.7: Case of use «CU07. Agent intelligence»

Requirement:	R8
Actor:	Agent
Description:	The agent perceives various stimuli present within the game environment and responds accordingly.
Preconditions:	1. The agent must be able to build turrets (see CU07 in 3.7). 2. The agent must have the component ML-Agent.
Normal sequence:	1. The agent's ML-agent component is started at the beginning of the game. 2. The agent perceives a stimulus. 3. The ML-Agent modifies the values of the variables in the class. 4. The decision making of the agent is affected according to the stimulus perceived by the agent.
Alternative sequence:	None.

Table 3.8: Case of use «CU08. Reaction to stimulus»

3.3 System Architecture

This project has been developed in Unity version 2022.3.5f1. For running games made with this version of Unity Engine the recommended system requirements are:

- **Operating System**: Windows 7 (SP1+), Windows 10 and Windows 11.
- **Processor**: x86, x64 architecture with SSE2 instruction set support.
- **Memory**: 4 GB RAM or more.
- **Graphics Card**: DX10, DX11, DX12 capable.

These are the requirements recommended by Unity for games developed in Unity Version 2022.3 [15]. However, as the project graphics are made in low poly[1], we can not take these requirements to the letter, since the render times for these models are shorter than the average rendering time of Unity projects.

3.4 Interface Design

Since this project is the incorporation of artificial intelligence in a type of video game and not about developing a complete video game, only making a technical demo, the player interface is rather simple to facilitate the understanding of potential players. Therefore,

[1]Low poly refers to a style of polygon mesh in 3D computer graphics characterised by a relatively small number of polygons.

the interface consists of two buttons, the option menu button and the convoy creation menu located on the left side of the screen. The player currency appears at the bottom of the player. Meanwhile, at the top you can see the total health of the two enemy bases on the left side, and the percentage of terrain built by the enemy on the right side. The path followed by the convoy is marked with a blue line. At each intersection, the arrow appears with which the path can be changed. Finally, you can see the life of each unit and each tower built. In this way the screen is freed from useless information (see Figure 3.1).

For the convoy creation menu, a simple interface has also been chosen. This is composed of six buttons corresponding to the 6 positions of the convoy. By pressing the buy button, you can always add a unit from the convoy queue. When clicking on the buy button, a drop-down list of available units appears. By hovering the mouse over the unit buttons, all the information about the unit can be seen, so the player can decide which unit to buy at that moment (see Figure 3.2).

If the player clicks on the button of one of the units on the field, another drop-down with the sell button appears. This is done in case you want to add a repair button or an upgrade button in the future. Hovering the mouse over the sell button will also show the information of the unit to be sold. The money appears at the bottom of the buttons (see Figure 3.3).

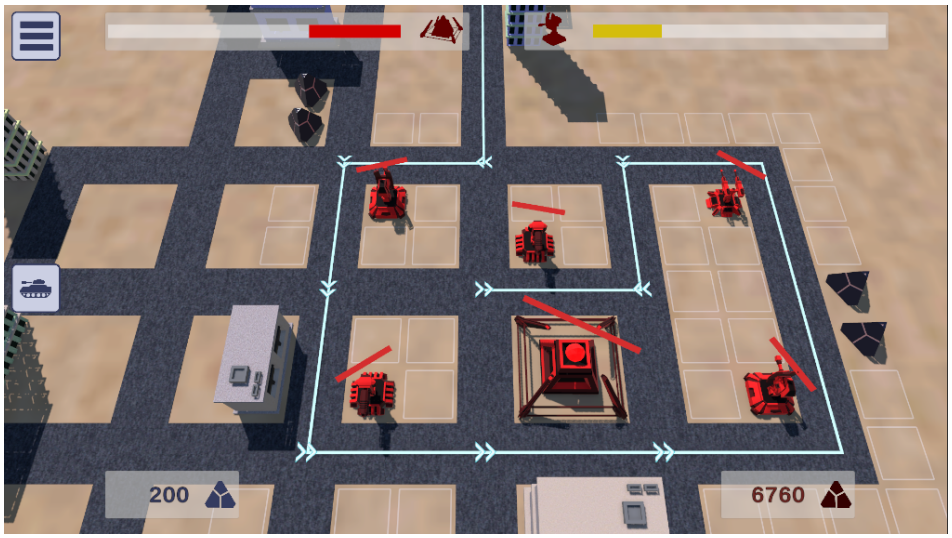

Figure 3.1: Final HUD visualisation

Figure 3.2: Final create unit HUD visualisation

Figure 3.3: Final sell unit HUD visualisation

Chapter 4

Work Development and Results

Contents

4.1 Work Development **21**
4.2 Results **42**

This chapter shows in depth the creation of the various behaviours and actions that the agent and the player can perform, as well as the design process. It also illustrates some of the problems that the project has gone through and how they have been solved.

4.1 Work Development

The work will be elucidated in chronological order to provide a comprehensive understanding of the process, including any changes, additions, and challenges encountered throughout the project.

4.1.1 First Steps: Creating the assets

When the project was conceived, it was not expected to develop a complete video game. Eventually, it was understood that a well-designed assets and environment would be beneficial to illustrate the final result. However, the assets were designed as low poly models to reduce the time spent on its design. At the end, the final result agrees with the characteristics of a technical demonstration.

In order to facilitate the creation of the models, I have relied on semi-futuristic weapons and vehicles that facilitate low poly modelling. Therefore, as you can see in Figure 4.1, I have a laser weapon mixed with conventional weapons, or anthropomorphic units, next to tanks.

To illustrate the correct functioning of the AI, I modelled different types of units where the AI could choose from, so a rock, paper, scissors model was followed[1]. Following this premise, I listed the units that have been modelled and catalogued them in our model.

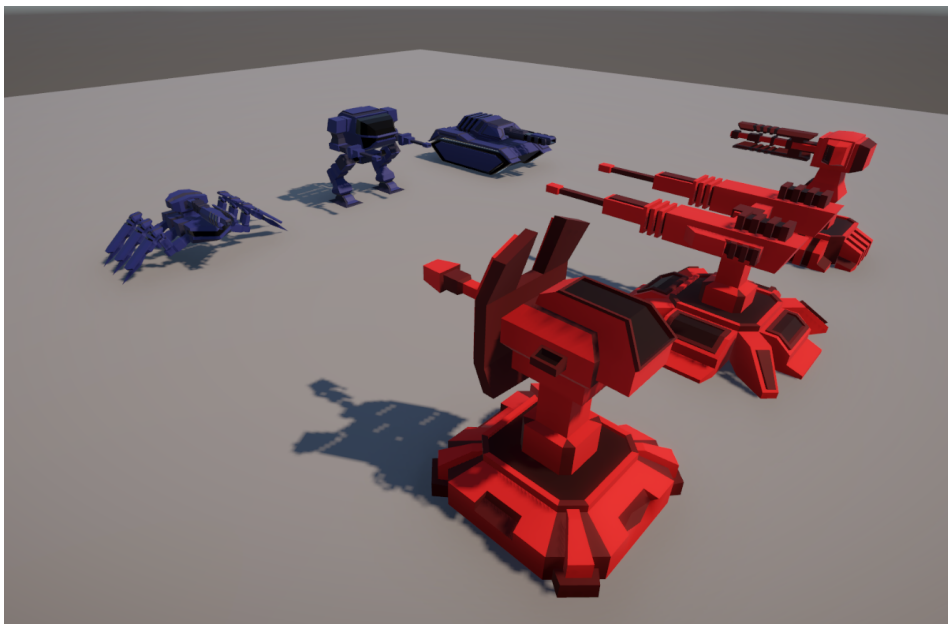

Figure 4.1: Modelled units

To begin with, I modelled the Assault Mech. This is the player's standard unit. I tried to give the impression of a light and versatile unit. As I wanted it to be a fast-firing unit, I decided that the mech's arms should be light canons.

The Artillery Hammer is an arachnid artillery unit. It was originally planned to add 8 legs, but due to a problem with the animations, it was reduced to 6 legs. The idea is to make it a lightly armoured but high damage unit. This was achieved by reducing the size of the main body. At the same time, I tried to make the main gun prove that it is a very powerful unit.

The Heavy Tank, as the name suggests, is a heavy tank. It has been given a loaded look to give the impression of a rough unit. You can see that the gun is similar to that of the Artillery Hammer, this is because they are designed in much the same way, but the gun of the Heavy Tank has been made smaller to show less firepower.

Moving on to the agent units, they were originally designed to be less powerful than the player units. This is because during the course of the game, there will be more towers than player units. Still, every player unit has a tower counterpart. The Assault Mech's

[1]Rock-Paper-Scissors has 3 outcomes: a win, loss, or tie. Rock defeats scissors, scissors defeats paper, and paper defeats rock. Players who play Rock-Paper-Scissors have an equal probability of winning, assuming that both players choose options completely random [19].

counterpart is the Blaster. The Blaster is like the typical starting tower defence of any Tower Defense. Like the Assault Mech, it has two light cannons that fire simultaneously.

The Storm Reaper is the counterpart to the Artillery Hammer, replacing the large, massive damage cannon with a slightly more moderate damage laser. In exchange for the damage reduction, it was thought to have its armour upgraded. It was therefore decided to make the turret body larger than that of the Blaster. To show that the Storm Reaper really is the counterpart of the Artillery Hammer, a similar cannon was designed, but with a different orientation.

Finally, the Behemoth, as is obvious, is the counterpart to the Heavy Tank. The Behemoth is not only the most heavily armoured unit, but also the one that deals the most damage. An attempt has been made to give this impression by creating a sort of shield around the turret and making the gun of a larger calibre than the other guns. The idea of this unit was that its bullets would do area damage.

The modelling process was carried out with a consistent structure. Each unit comprises a main body, accompanied by a rotating element on the horizontal axis and another rotating element on the vertical axis. With this, what I achieve is that when importing the model into Unity, the movement of the unit to aim at its target is more dynamic and easier to achieve. In the player units, particularly in the two anthropomorphic ones, the lower part of the model has been divided into three parts: the upper leg, the lower leg, and the foot. As previously noted, this segmentation enables the utilisation of Unity's dynamic animations, which will be elaborated on in the following subsection.

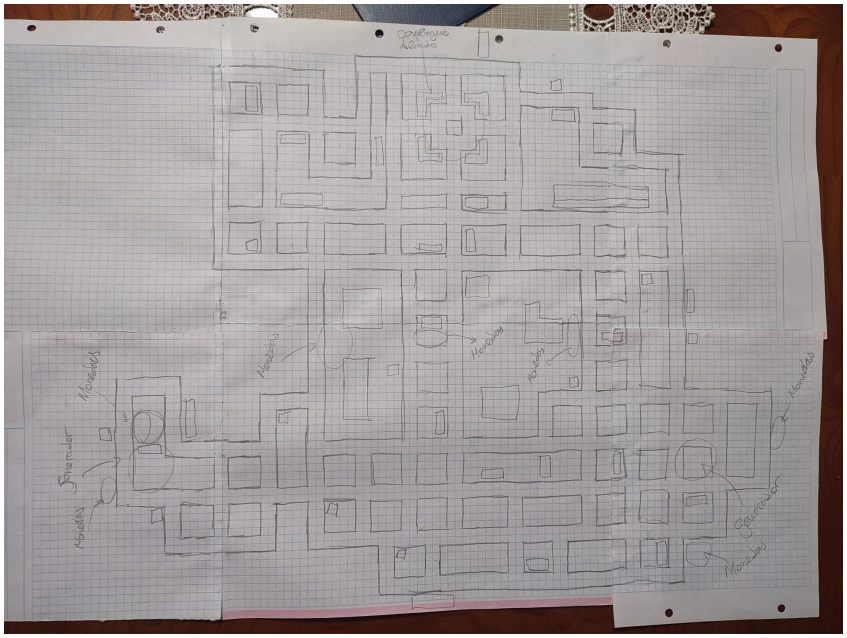

Figure 4.2: Hand-drawn video game map

Following the completion of modelling the main units in the video game, the focus was

directed towards modelling the environment. The design process was informed by the level design of the Anomaly 2 campaign. Extracting segments from the expansive levels, a meticulous manual merging process was undertaken to create the level showcased in the image (see Figure 4.2). Subsequently, numerous buildings were meticulously modelled and adorned with unique materials, strategically positioned throughout the map to ensure comprehensive coverage. It is worth noting that the idea was to represent a city at war so the buildings the plan consisted of designing varied types of buildings and obviously, buildings seriously affected by war. But as I have mentioned before, it was not intended to take many hours of the project modelling the environment, so only two building models were designed and combined with each other to give dynamism to the city.

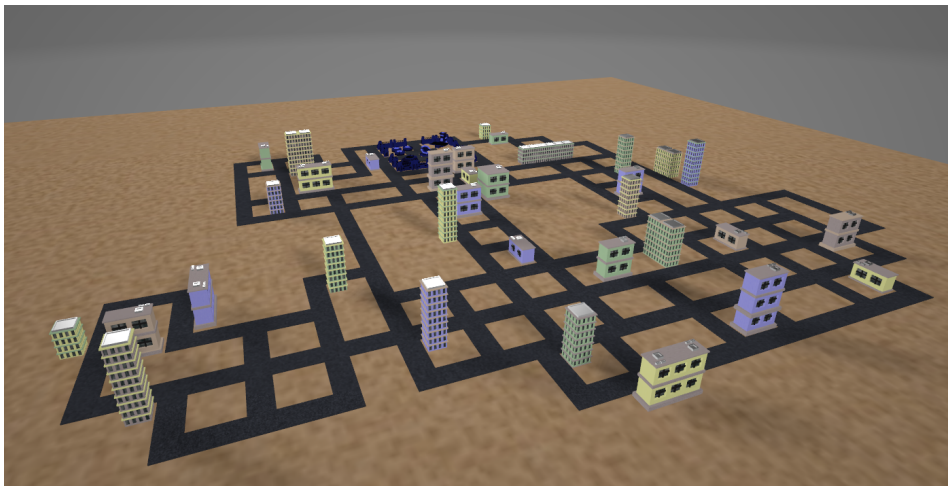

Figure 4.3: Final result of the environment

4.1.2 Adding animations and camera

Once the basic assets were modelled, the next step is the animation of the assets. In this case, the Assault Mech and the Artillery Hammer had to be animated, as they are the only models with legs. As I wanted to speed up this part of the development, I was investigating a way to build these animations procedurally. It was found that in Unity 2022.3 the Animation Rigging library [11] was added in its vanilla version. This library allows us to rig a model, and animate it procedurally through code.

To implement the rig in a model, just add a *Rig Builder* component and a *Bone Render* component to the model, which adds all the parts that compose the model as bones. Then for the constraints, I only have to add what type of constraint I need. In the case of the models we are aborting, I need a *Two Bone IK Constraint* component on each leg to move them correctly.

For the realisation of the animations, a script called *Foot Solver* was created which works as follows. First, all the legs are set to be fixed on the ground. This way, even if

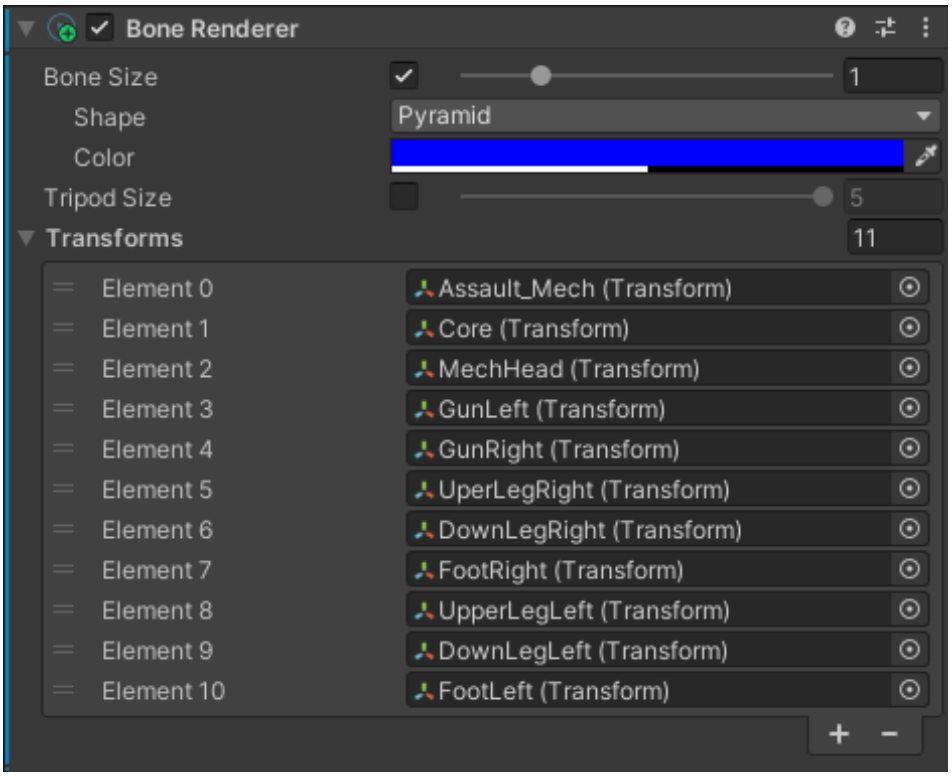

Figure 4.4: Assault Mech Bone Renderer component

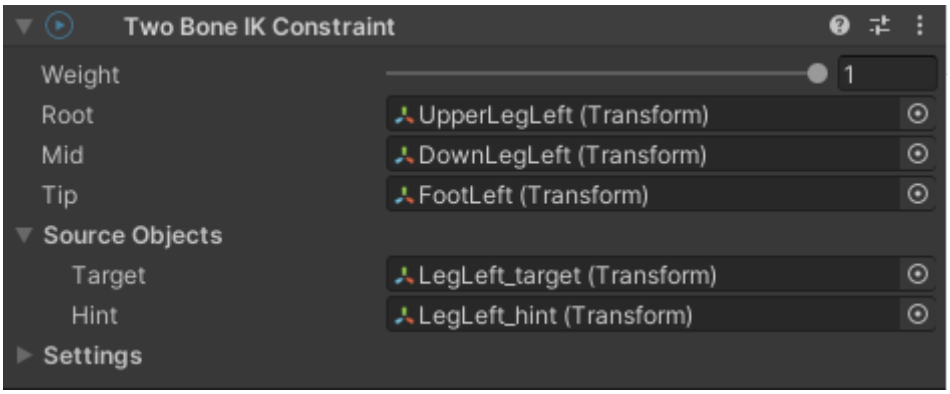

Figure 4.5: Two Bone IK Constraint component for left Assault Mech leg

the body moves, the base of the legs will adhere to the environment. Then a target is created for each foot that follows the position of the model's body. This will later allow them to move the bottom of the leg to that target. The next step is to launch a Raycast[2] from the target in order to move the target in the Y-axis so that it can overcome terrain obstacles. To know when to move the leg, simply check the distance to the target and when the distance between the two points is greater, move the leg towards the target giving it a certain height to simulate a step. Finally, just repeat the process for all the legs of the model. In this case, the order of the legs is set by the order in which they have been added to the leg array.

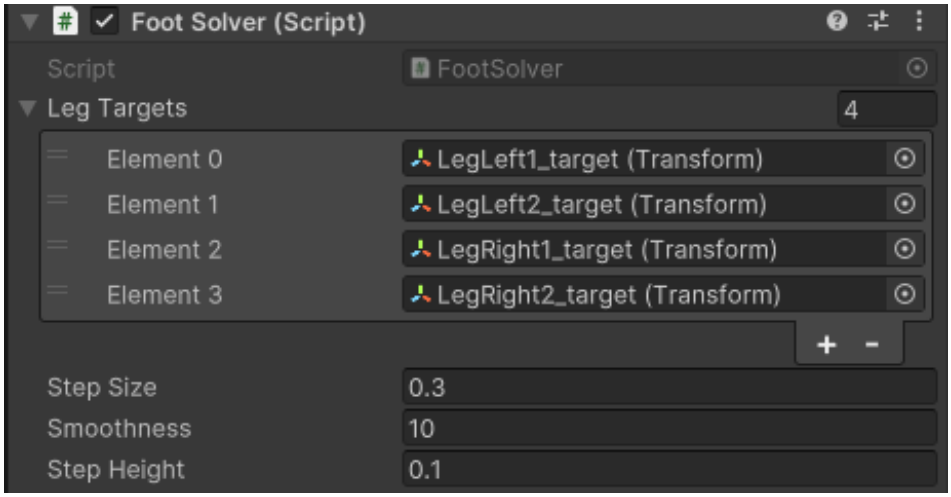

Figure 4.6: Artillery Hammer Foot Solver script

In addition to these animations, other animations have been realised in the classical way. All units (both player units and turrets) have an Animator Controller attached, with a non-moving idle animation and a firing animation that I will mention later. This animation is a simple backward movement of the cannon giving a sense of recoil. Finally, an animation has been added to the player base antenna in an attempt to add some dynamics to the environment.

To continue with the development of a user-friendly yet pleasant experience of our tech demo, a camera capable of moving around the environment conveniently was required. A further positive addition would be a camera with a zoom lens. As a result, the player can see the events in the scene. For this it was decided to use the Cinemachine package. This package is able to set up a functional camera in a quick and easy manner. To develop our camera it was enough to create a *Transposer*[3] camera that follows a gameObject placed in the scene, that is the one that actually moves.

[2]Raycast casts a ray, from an origin point in a determined direction, hoping to collide to any other object in the game scene.

[3]This Virtual Camera Body algorithm moves the Virtual Camera in a fixed offset to the Follow target. It also applies damping.

I created a script *CameraControls* that using the keys W, A, S and D would move this object around the environment. As the camera is attached to this object, the camera follows it and I get the desired movement. For the zoom, I access the *FollowOffset* property of the Cinemachine to move the camera up and down interacting with the mouse wheel.

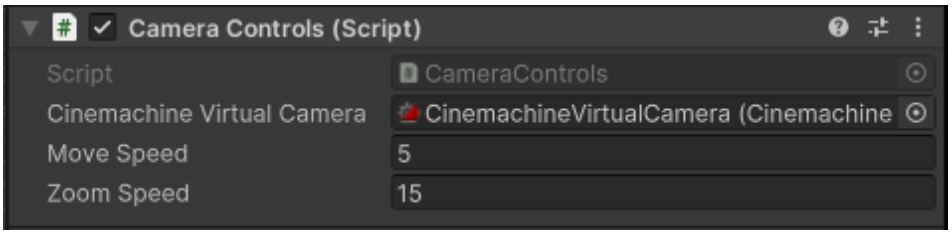

Figure 4.7: Scene CameraControls script

4.1.3 Basic setup

After a way to observe the entire environment was created, a basic setup was established for the correct development of the player mechanics and artificial intelligence. This includes scripts for the correct functioning of the convoy units and turrets, projectiles for all units, and a functional life system. Also, at this point in development, multiple visual effects were created to better visualise what is happening on screen.

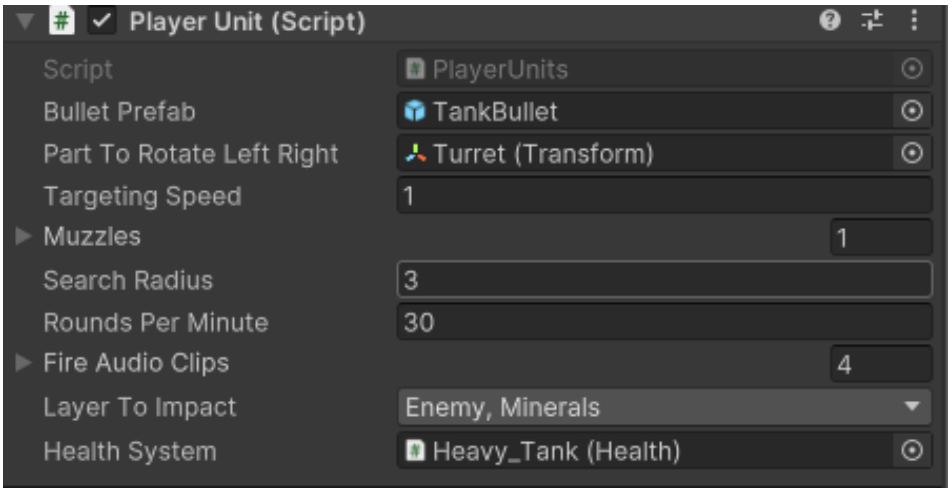

Figure 4.8: Unit Tank Script

The idea was to keep the code clean and reusable in as many scripts as possible. This is why I tried to separate the visual part of the project from the functional part. This means that, for example, when a unit is taken out of some life, an event is called, to which the visual part subscribes, which takes care of updating it only when it is

necessary. This allows in addition to that mentioned to reduce the number of calls that are made in the Update loops. Also, any script that has been created in this project has been made with the idea that it would be compatible with any other project of these characteristics, so that both the player's units and the AI towers would be able to rotate on their horizontal axis, rotate on their vertical axis and shoot. For this reason, a Unit script was created to handle all the movement of these agents.

This script is responsible for updating the target I want to shoot against, in other words, the aiming. It is also the one in charge of horizontally rotating the body of the unit towards the target. Vertical rotation was not included because the player's units were not prepared for it. In this script I can also find the target firing system.

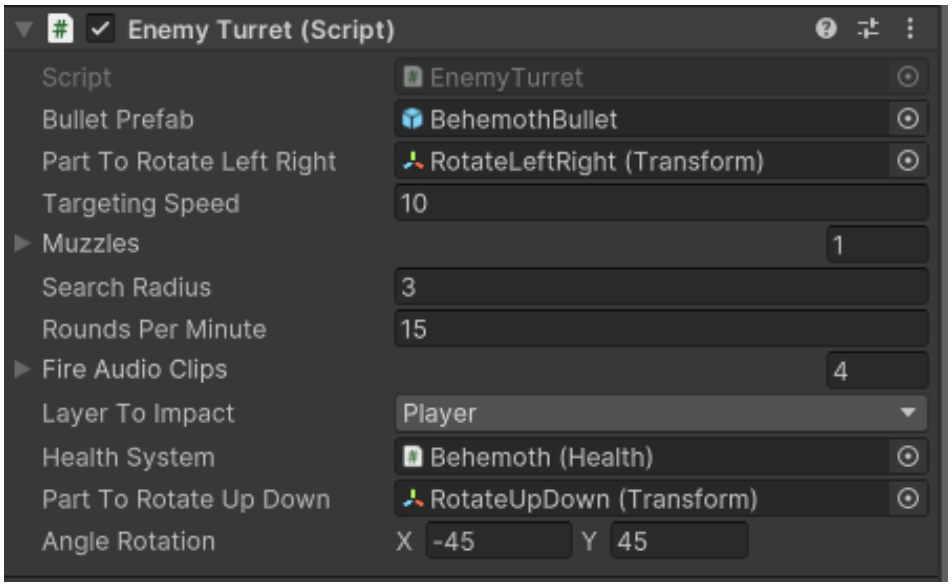

Figure 4.9: Unit Behemoth Script

All the aiming system was done using the collider physics [12] system provided by Unity. When something enters the physics sphere around the unit, as long as it doesn't already have a target, the unit starts to rotate its body towards the target. A Raycast is then launched at that target from the cannon, and if there is a collision, it is fired. You can see in the enemy script (see Figure 4.9) that, in addition to the same parameters that you have in the allied unit, you have a Part To Rotate Up Down component. This component is the one that rotates on the vertical axis. The next parameter is the minimum and maximum rotation that this element can reach. In this way, I ensure that the rotation does not exceed certain limits and that the rotation of the unit is seen correctly.

The only slightly different unit is the Storm Reaper, as it lacks a bullet prefab. As this tower has to fire a beam at the enemy, a Line Renderer [13] was created to act as a laser beam which was set as the point of origin to the gun barrel and the point of impact to the unit it is firing at. The unit itself does the damage, as opposed to the other units

where it is the bullet. The damage per second is obtained by multiplying the damage assigned in the variable by the Time.deltaTime.

At this time they were also provided with a sound system to make a firing sound every time a bullet was fired. This system, to maintain the philosophy of separating functionality from visuals, is done with an event system. At the same time, the shot uses collision physics [12] to know when it has hit a unit and therefore subtracts life from our life system. A particle system has also been added to this bullet to create the shot effect.

Regarding the bullet, it was created with the intention of having a maximum distance it can reach. The visualisation of it was done with a Line Renderer [13] to which applying a speed gives the effect of having a large movement. Like the health system that we will see later, it has a Big Gun parameter with an extra prefab. The Behemoth unit has an area explosion so when this flag is activated and we have the prefab assigned, apart from showing the impact prefab, it also shows the blast prefab doing area damage.

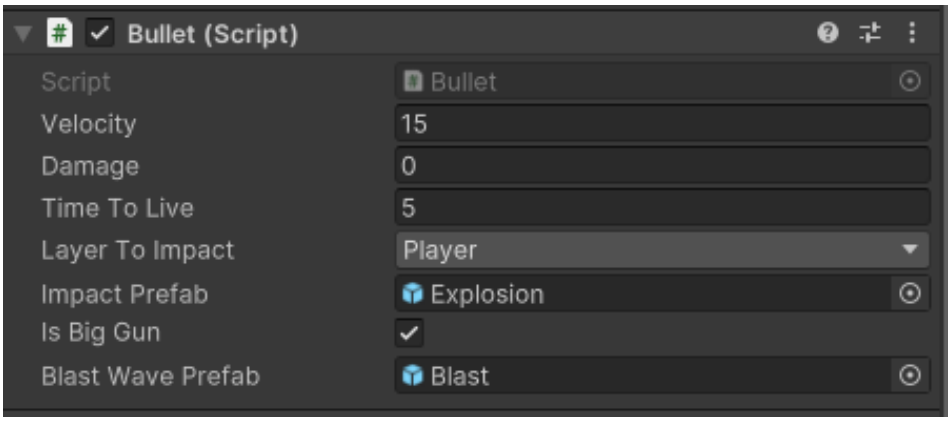

Figure 4.10: Behemoth Bullet Script

On the other hand, and as mentioned briefly in the previous paragraph, a life system was implemented. Each unit is given a variable life that when it reaches 0, it removes the object and triggers an event that triggers an explosion (also created with the particle system). Later, this was changed so that it was the ConvoyManager (the player's information manager, which will be discussed later) that handled the destruction of the allied unit, and the AIBehaviour that handled the destruction of the enemy tower. This allowed us to make the life system communicate with the Unit script, being universal for both types of units, without repeating code. As can be seen in the Figure 4.11 there are various parameters that do not correspond to what would be a normal health system. These are the boolean Has2death and the integer Object Value. The first boolean is used to distinguish between the death of normal units and the death of the enemy base. Normal units simply explode on death. But the enemy base, in addition to exploding, generates a shockwave that does area damage. Then I have the Object Value which is used to add that number to the total currency the player has.

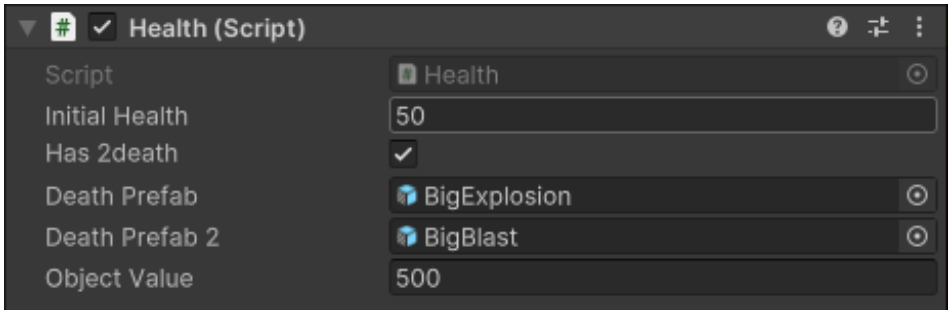

Figure 4.11: Enemy base Health System Script

4.1.4 Player mechanics

Once the development of a basic setup for the tech demo was finished, it was time to focus on the player mechanics. This includes giving units a way to move around the environment, and a basic HUD with which to create units and sell units. It was also important to develop an appropriate economic system.

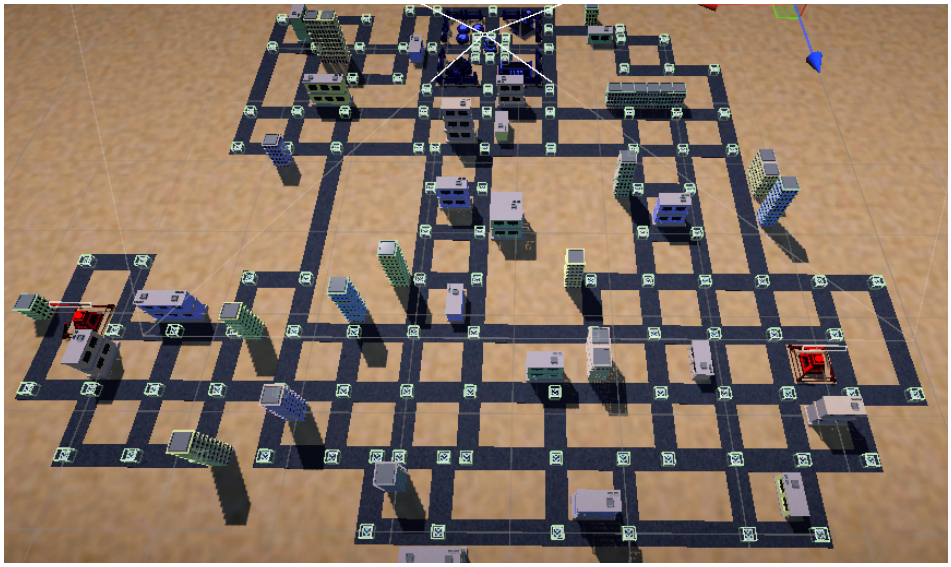

Figure 4.12: Map with stabilised nodes. Nodes can be seen because of its Collider Box

The initial idea was to create an undirected and unweighted graph[4] in which each node corresponds to an intersection in the game environment. For this, an undirected graph class was created. This class would be in charge of building the graph at the beginning of the game. This was achieved by adding all the nodes inside an object and retrieving all the node components from the children of the object. I then traversed all

[4]Set of vertices connected pairwise by edges. The edge has not any specific direction or weight.

the nodes and launched a Raycast in all four directions. A Collider [12] was previously set up at each node. This allows that when the Raycast hits another node, it will be added as a neighbour node of the node we are evaluating. An important detail is that we have to distinguish when a node is an intersection or not. To do this, once all possible neighbour nodes have been added to the node we are evaluating, we check if the number of neighbour nodes is greater than 2 to mark it as an intersection or not.

The project required that this network could be accessed from anywhere in the project. This is why, in order to have a clean and efficient code, the Singleton design pattern[5] was used for the creation of this class. Later we will see that this design pattern is repeated in several classes of the project with the same needs as the network.

The next challenge posed by the project was to make a group of units, in other words, a convoy of units that had the ability to move around the graph following each other. This challenge was similar to the movement of the snake in the classic video game Snake[6]. For this reason, the movement of the convoy is very much based on the movement of the snake in this video game. The head of the convoy is actually the one that is moving all over the map and in the meantime, it is generating some markers. These markers are the points to which the units that follow it are going to move. So we have a convoy head and units that follow it all over the graph without being randomly moved around the graph.

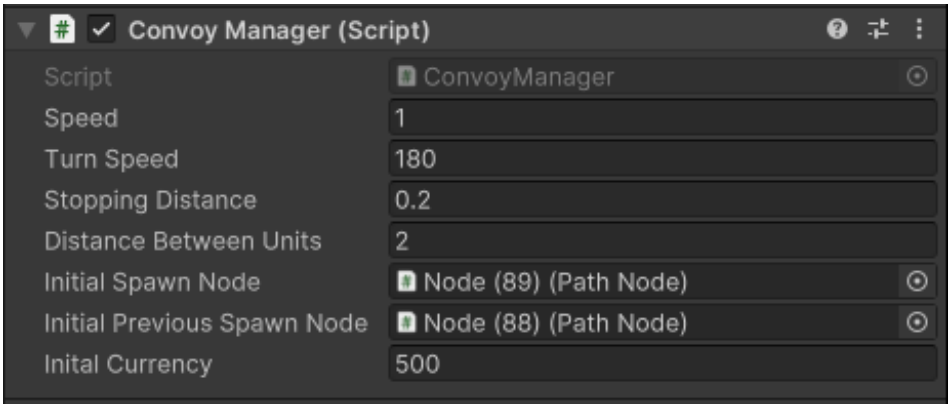

Figure 4.13: Map with established nodes. Nodes can be seen because of its Collider Box

[5]In software engineering, the singleton pattern is a software design pattern that restricts the instantiation of a class to a singular instance. One of the well-known "Gang of Four" design patterns, which describes how to solve recurring problems in object-oriented software, the pattern is useful when exactly one object is needed to coordinate actions across a system [30].

[6]Snake is a genre of action video games where the player manoeuvres the end of a growing line, often themed as a snake. The player must keep the snake from colliding with both other obstacles and itself, which gets harder as the snake lengthens. It originated in the 1976 two-player arcade video game Blockade from Gremlin Industries where the goal is to survive longer than the other player. The concept evolved into a single-player variant where a snake gets longer with each piece of food eaten (often apples or eggs). The simplicity and low technical requirements of snake games have resulted in hundreds of versions (some of which have the word snake or worm in the title) for many platforms [31].

Before I continue with the development of the convoy class, it is worth noting the placing of an initial path for the convoy. This algorithm was made so that when a way to change the path is enabled, this algorithm could also be used. The algorithm created uses the current node to which the convoy head is heading and the previous node. Within the node class, an algorithm was made to search for a valid node to which the convoy could move. This node algorithm went through its neighbours looking for a neighbour that was not already part of the path and was not the previous node, then added that neighbouring node to the path and searched for a next valid node and repeated the loop until the selected neighbouring node was within the path.

Once the initial path was established, a visualisation was added to the nodes. This visualisation corresponds to an arrow and a Line Renderer [13] going from node A to node B. Interaction with the arrow image was enabled to switch between the four available directions and the node class was modified again by adding an index so that it could cycle through all valid neighbouring nodes. Therefore, every time an arrow was clicked from the visual, an event was triggered to the convoy manager. This event was in charge of modifying the variables so that a new path would be created from the point where it was pressed. As can be seen, the philosophy of separating visuals from functionality is always present.

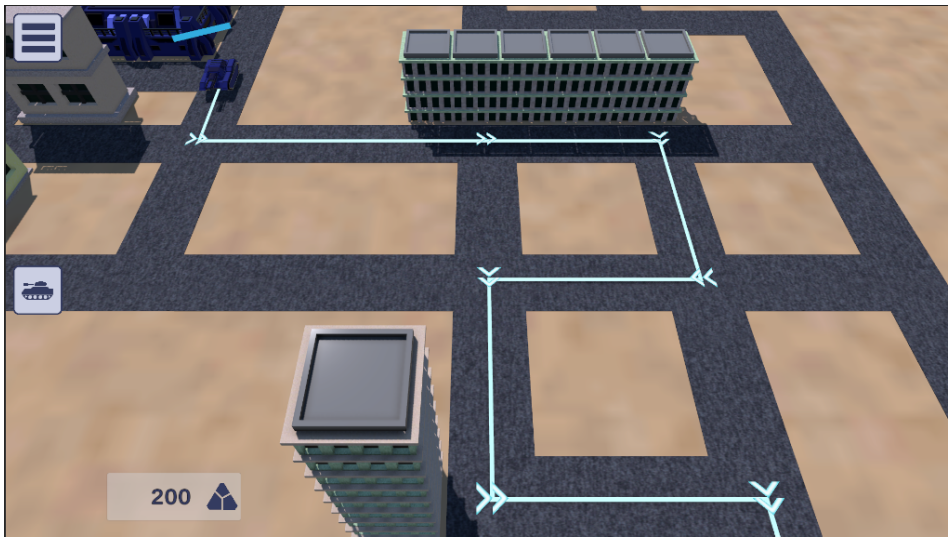

Figure 4.14: Map with path nodes functionality

With all this I already had a main unit which was followed by several units that were the body of the convoy moving through the network. The player could also interact with the nodes of the network to change the path the convoy followed. The next step was to allow the player to create and destroy convoy units. For creation, what was done was to create a timer in which if there were still units to be created when this timer exceeded the minimum distance between units, the unit itself would be created. At the same time, because of the way the code was initially designed, the destruction of the unit simply

involved removing the unit from the convoy's list.

These two methods of adding units and removing units allow us to create a simple HUD. In this HUD, clicking on the buy unit button displays the three available units and clicking on one of them adds that unit to the convoy. At the same time, if you click on the button for one of the units in the convoy, the sell button appears. If this button is clicked, the unit to which the button belongs is removed. To make it clear which unit the player was buying, an event was added where when the player hovered the mouse over the button all the unit information would be displayed. The synchronisation of the buttons with the units in the convoy was done through events. As the convoy manager class was starting to be used at many points in the project, the decision was made to convert it, as with the graph class, into a Singleton.

The unit information was starting to fill the convoy manager class and the unit classes with visual elements. Faced with this problem, it was decided to use the Unity Scriptable Object [14] tool in a way that would relieve the load on the classes. For this reason a Scriptable Object [14] was created to store the information that appears in the HUD.

Finally, to finish off the development of the player mechanics, an economy system was added. This system relies on minerals scattered around the map, in particular near enemy bases. When an allied unit detects an ore it shoots at it and destroying it gives it a certain amount of currency. Both the towers and the enemy base also give currency when they are destroyed. It is important that the enemy base gives some kind of reward because its blast wave is capable of destroying the entire convoy. In the situation where the convoy has been completely destroyed, the convoy HUD reappears to generate a new convoy that will start inside the player's base. If there is not enough money to buy a new unit, the player loses.

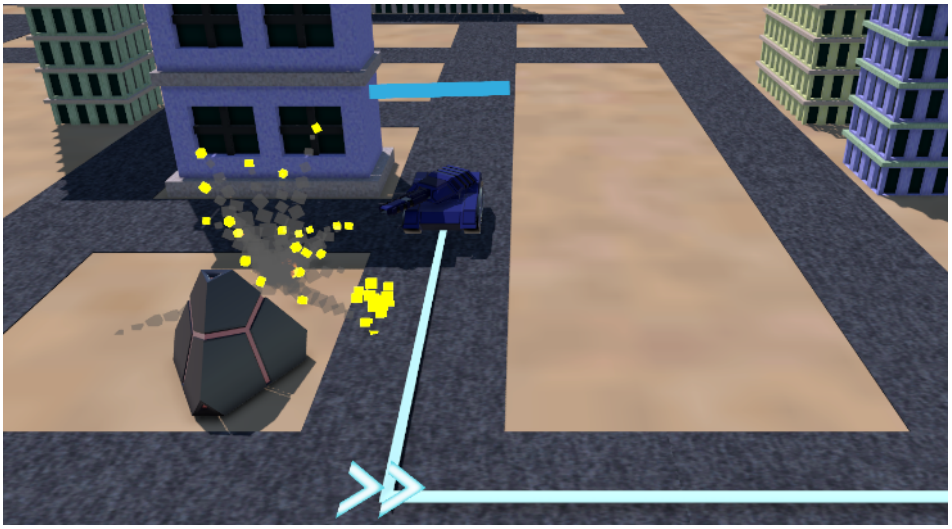

Figure 4.15: Tank in the convoy destroying minerals for currency

4.1.5 AI design

With all the elements already developed I was ready to create and implement an artificial intelligence using machine learning. The process of designing artificial intelligence went through several phases, each of which involved finding different strategies for implementing artificial intelligence. First of all we have to know that this artificial intelligence has to be developed by reinforcement learning. This is the type of learning in which machines learn and perfect their techniques based on their own experience.

Reinforcement learning is a technique within the field of machine learning [3]. In this approach, a machine learns by itself through interaction with its environment. The machine seeks to maximise the rewards it receives and minimise the penalties. It does this by making decisions and performing actions that bring it benefits. Every action performed by the machine receives feedback. This feedback can be positive if the action was beneficial or negative if it was detrimental. The machine uses this feedback to adjust its behaviour. In this way, it learns which actions are most appropriate in different situations. Reinforcement learning is similar to the learning process in living things. In nature, organisms learn through experience and feedback from the environment. In addition, reinforcement learning allows machines to develop long-term strategies. This means that they consider not only the immediate benefit, but also the future consequences of their actions.

To do this, we must first define the state and the environment. We have to be clear that in a normally developed artificial intelligence the goal is to achieve the desired action in the shortest possible time, which means at the lowest possible cost. In our case, what we want is to prevent the player from achieving the goal in as much time as possible, while at the same time achieving his own goal at the lowest possible cost. The goal is that given a state, the AI will be able to predict the best strategy to stop the player. In this case, the state being the current situation of the environment. Specifically, the position of the bases, the direction of the convoy, the distance from the convoy to the bases, the position of the towers already built, and even the current money of our AI. Needless to say, the environment would be the game itself.

With all this, I first thought of approaching the problem of building this artificial intelligence using Q-Learning. This strategy based on learning by error is based on storing all the data in a table known as a Q-Table. This Q-Table is a map that associates each state with all possible actions and their respective utility values. The agent then uses these values to select optimal actions according to each situation. Q-Learning is based on the Bellman equation which is a calculation that expresses the optimal value as the addition of the immediate rewards and the expected value of future rewards. This strategy is very useful when the action spaces are finite and not very large. In my case, I have the advantage that I have a well-defined grid that I can use to create our Q-Table. The problem is that the space is quite large, so this option was discarded.

I thought of some other machine learning strategy that, having a finite state space, would have more capacity, so the creation of a neural network fulfilled both characteristics. A neural network is a machine learning architecture that allows us to generalise knowledge. Through a training process, it is able to find patterns in the data and apply

this acquired knowledge to data that has not been previously seen. It is important to highlight that this type of reinforcement learning strategies do not need previous data to work and therefore follow an arbitrary policy[7].

Although this is the strategy that was finally followed to realise the project, the implementation of a critical actor was also considered. This means that there would be two neural networks working simultaneously. The first one would be in charge of policy, which means that it would be in charge of evaluating the states for a correct action, while the other one would be a neural network built with Q-Learning. This would be in charge of evaluating how promising the action to be performed is. Another strategy that was tried to be developed was to create a multiple intelligence. In this strategy each tower would have its own artificial intelligence developed by Q-Learning as the action space would be smaller. These intelligences would be able to communicate with each other so that they could decide which one would be built first and where. Both strategies were discarded due to the complexity of their development.

As mentioned above, in the end it was decided to develop a neural network. The only problem was that the current environment was too complex to understand what was being done. Therefore the decision was made to reduce our environment. In this case a smaller map was designed, the number of bases was reduced to one, and finally the number of towers that could be built was reduced to two, being the Harvester and the Blaster. Once the neural network will work for this case, the existing neural network would be upgraded for the complex environment that was originally designed.

4.1.6 AI behaviour

But before applying machine learning to my technical demo, I must establish some rules to facilitate the operation of artificial intelligence. The final objective is to obtain a function to which I can pass as attributes a position where the tower will be built and a number that will be assigned to each tower.

First of all, the remaining tower called Harvester was designed. The Harvester is in charge of the economy of our artificial intelligence. There can only be two of these units in the environment and they have a limited lifespan. This means that they will generate currency for a certain period of time until they are destroyed. At that point, if the AI wants to get more currency it will have to build another Harvester. As this is a technical demo, AI information has been introduced in the HUD. In this case, on the opposite side of the HUD to where we have the player's currency, the AI's currency has been added.

To find out the positions where turrets have been built or where the enemy convoy is, a grid system has been designed with several classes that fulfil this function. First we have a *gridPosition* structure that stores the X and Z position of the grid. This structure is added to a *gridObject* class that will be responsible for storing which units are inside

[7]A policy is essentially a mapping from observations to actions. An observation is what the agent can measure from its environment and an action, in its most raw form, is a change to the configuration of the agent.

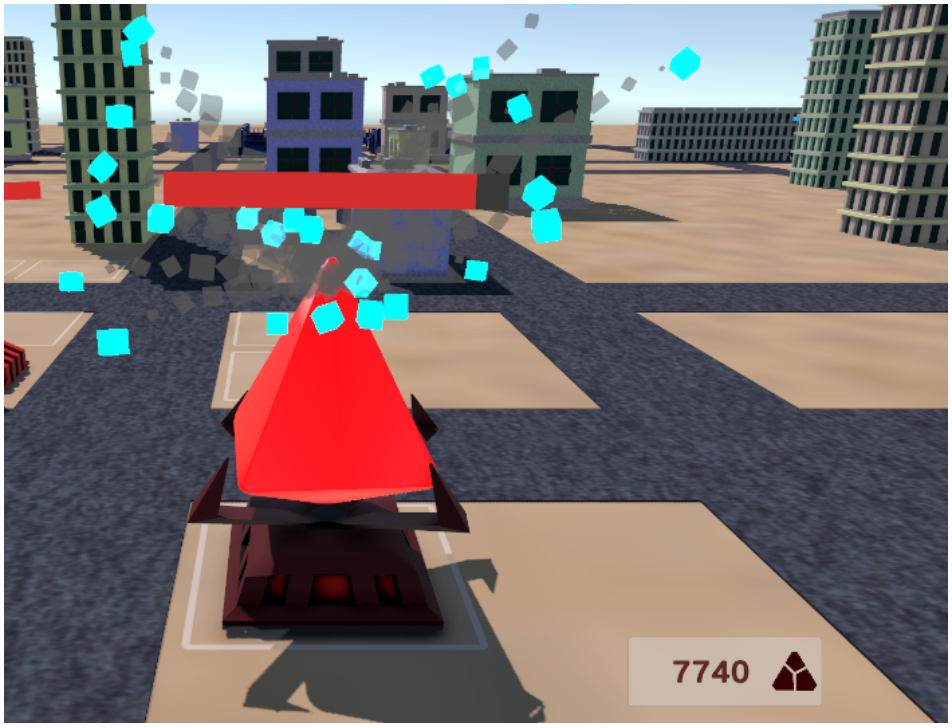

Figure 4.16: Harvester collecting minerals for currency

the grid position. In addition, this gridObject will have a boolean that will mark if that grid cell is buildable or not. In a *gridSystem* class we store an array of *gridObjects* so that the *gridSystem* class will store all the information and modify all the *gridObjects* accordingly. Finally, all this is modified in a *levelGrid* which is the object that will be in our scene. The *levelGrid* has the dimensions of the matrix, the size of the cell and is a class that is also used as a Singleton in order to have a single instance within the whole project.

When the *levelGrid* is created, from each position of the grid we launch a Raycast with the aim of finding the obstacles in those positions. If the Raycast collides with an obstacle, we mark that position as not buildable. This has allowed us to later not visualise the positions where an obstacle or the road is located.

I have enabled a way for each unit (allied or enemy) to update its position within the grid. This allows us to efficiently update the grid only when necessary. Also when I have to make the input to our artificial intelligence I only have to pass the *gridSystem* or the *levelGrid* with all the information it has to generate a valid response.

Once all the information about the turrets was stored in a grid, a script was created to control the behaviour of the AI. This script is in charge of managing what happens when the bases reach zero health and of establishing in which positions of the grid these bases are located. For the construction of turrets, an action system was created to know when an action was in progress and when the AI was free to do another action. This

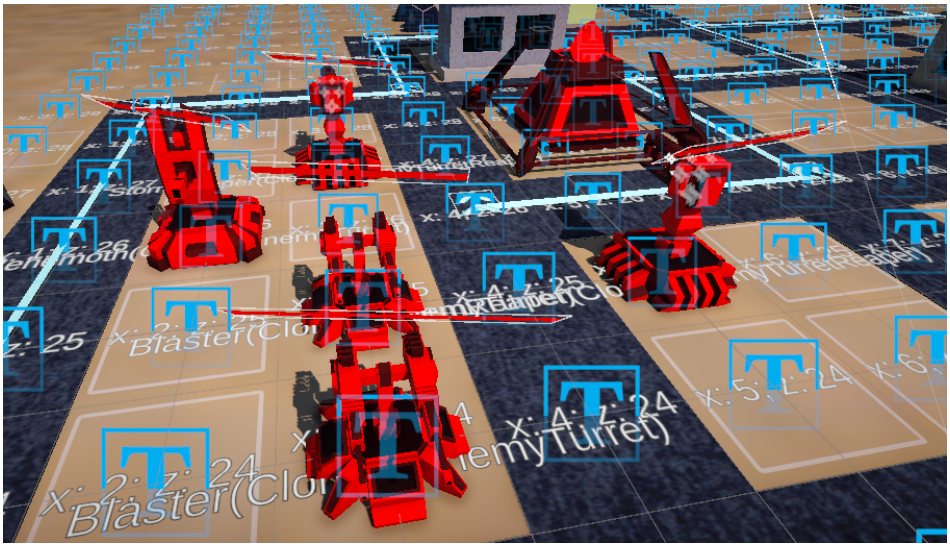

Figure 4.17: Visualisation of the debug *gridSystem*

is because I wanted to limit the number of turrets the AI could build at a time. A test flag was also enabled so that if the player right-clicked on a valid position on the grid, a random turret would be built.

Finally, a script for the turret building action was added. This script has two special functions. The first is the build action which accesses the output of our artificial intelligence. The idea was that the output would be three parameters. The first two correspond to a position in the world that could be transformed into a grid position and an index that corresponds to one of the four turrets that are available to build. On the other hand, there is a function in charge of calculating the valid positions where towers can be built. For this purpose, a maximum construction distance of three positions for the base and two positions for the turrets was established. Within this maximum distance it is checked whether the position is within the mesh, whether the position is occupied by a unit or the base itself and whether it is a buildable area. This list of positions together with the levelGrid information and the current money will be the inputs to our artificial intelligence.

As mentioned before, I wanted to display some information about AI. A visual object was created to show the valid positions where a tower can be built. This way the player can see how the map is evolving and that the AI is working correctly.

4.1.7 AI implementation

As previously mentioned throughout the project, our artificial intelligence was developed thanks to Unity's ML Agent libraries. These libraries allow us to develop a neural network, which is exactly what I need, in a simple way without having to develop the whole neural network framework in a complex way. It should also be noted that both

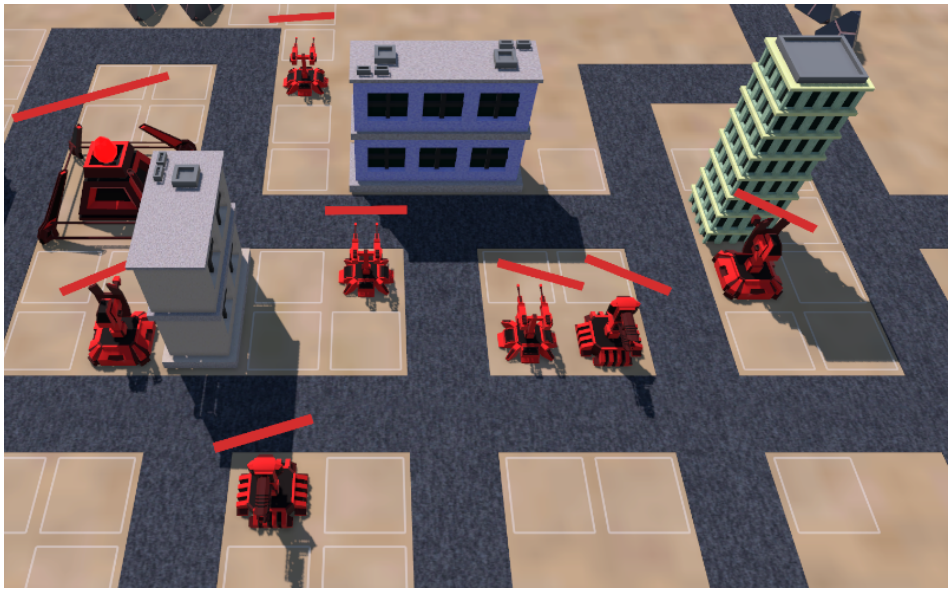

Figure 4.18: Visualisation of valid grid positions to which the AI could build a turret

the images shown and the explanations made in this section are in reference to the full model that was originally imagined and not to the reduced model that was used as a training test, since the full model is the one that was used at the end of the training.

To start training the agent I have to add several components to the overall *Gameobject*[8] of our artificial intelligence. The first component is the *Behaviour Parameters* [4]. This component is used to set the observations, the actions, and the neural network model that our AI will follow. By using these libraries, the environment is defined by the observations. It can be seen in the Figure 4.19 that only a single vector of size 1 has been set for the observations. This observation corresponds to the current money that artificial intelligence has and I pass it to it thanks to the *collectObservations()* [8] method that I added in my *AIBehaviorAgent* script.

As for the actions the AI can take, it can be seen that four discrete branches have been set. This means that the output must be four integers. These integers must be between the values set in the following parameters that can be seen in the Figure 4.19. In this case it has been decided that the first parameter corresponds to whether or not the artificial intelligence should build a turret, that is why the size of this answer is 2. The next two parameters correspond to the X and Z position of our grid. You can see that the number of values that can be returned by these two parameters corresponds to the size of the grid. The last parameter corresponds to the type of turret to be built. A number from zero to three has been assigned to each turret, so that the artificial

[8]The GameObject is the most important thing in the Unity Editor. Every object in your game is a GameObject. This means that everything you think of to be in your game has to be a GameObject. However, a GameObject can't do anything on its own; you have to give it properties before it can become a character, an environment, or a special effect [17].

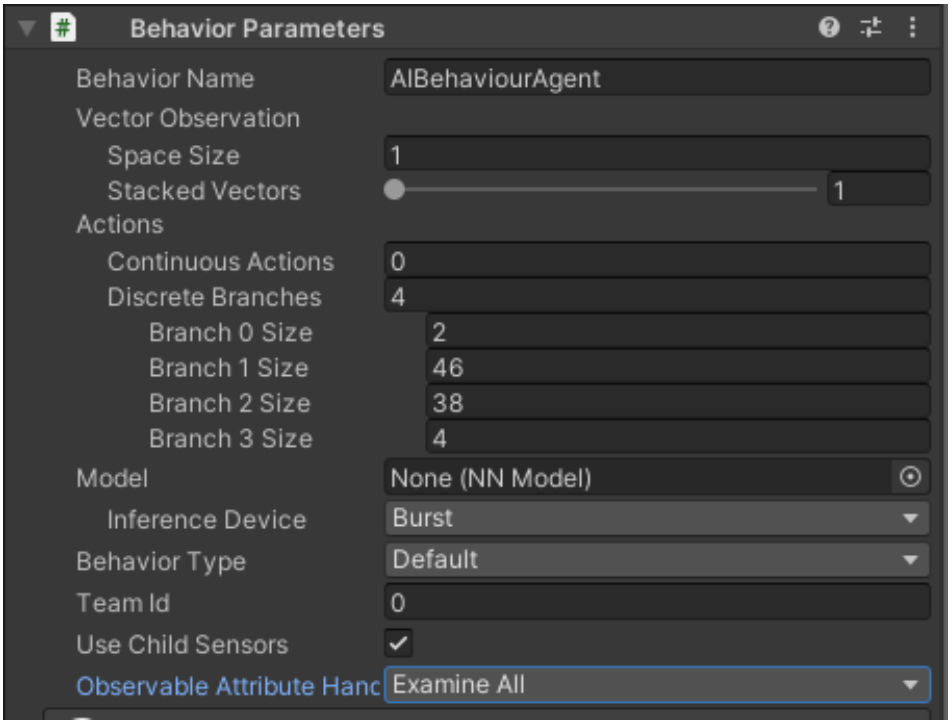

Figure 4.19: AIBehavior Behavior Parameters component

intelligence can easily and efficiently choose which turrets to build. These parameters as well as the observers are added in the *OnActionReceived()* [10] method where I configure how these parameters behave. For these actions to be carried out, I must add a *Decision Requester* [5] component. This component will be in charge of evaluating the observations every time the decision period is fulfilled and will take the pertinent actions.

It is clear that only the current amount of money that artificial intelligence has as an observation is not enough. The rest of the information is passed to our neural network by using a *Grid Sensor* [7] component. This component generates a grid using colliders and establishing already normalised values within each grid position. Several tests have shown that the information necessary for the artificial intelligence to make a correct decision corresponds to the position where a turret is already built, the position where the bases are located, the current position of the convoy units and, finally, the positions where the AI can build a turret. In order for the *Grid Sensor* [7] to retrieve all this information, colliders and tags were added to the elements that did not have them and that were going to be part of these detectors. The good thing about using this Grid Sensor is that we can visually see what information is being sent to the neural network (see Figure 4.21).

To continue configuring our neural network within Unity, I must add both positive and negative rewards to our neural network model. To centralise the reward system,

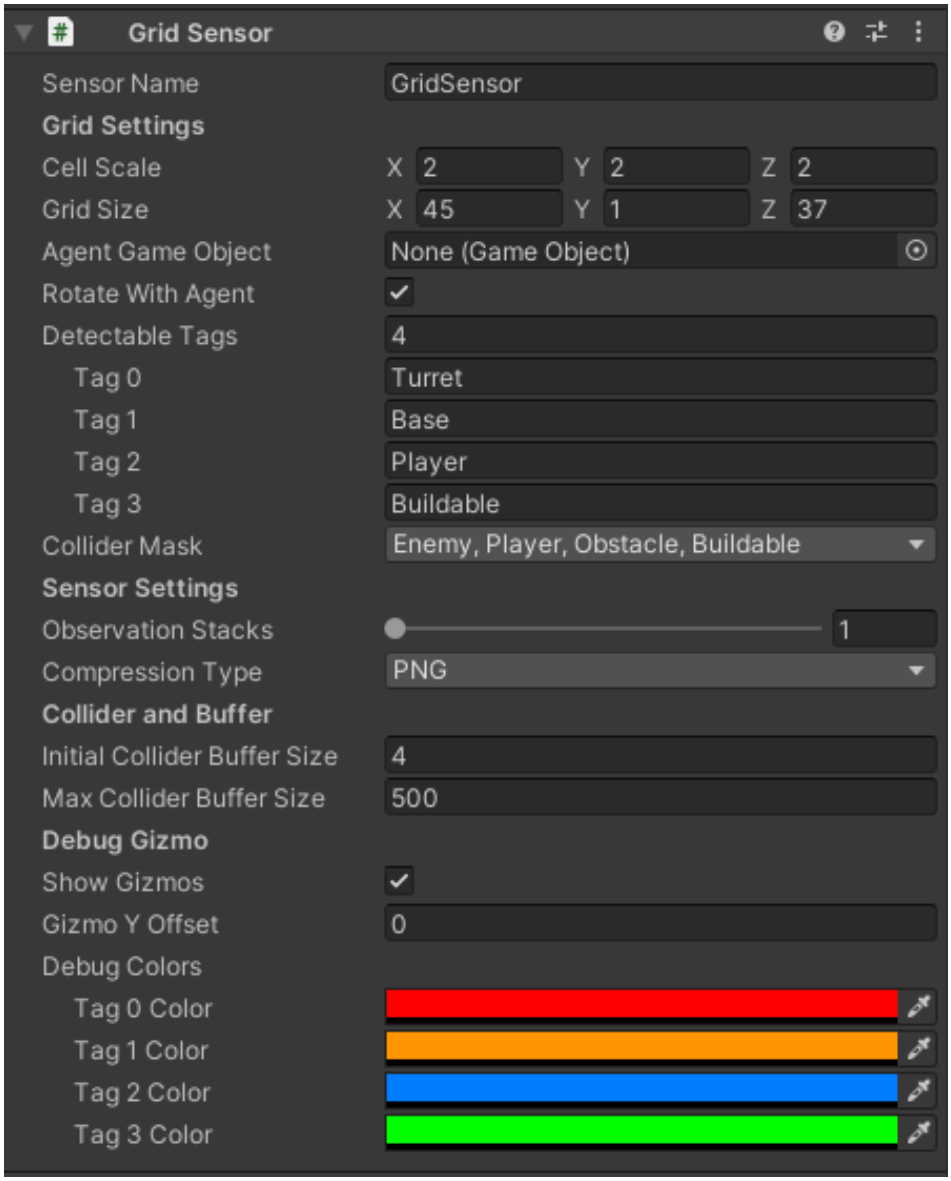

Figure 4.20: AIBehavior Grid Sensor component

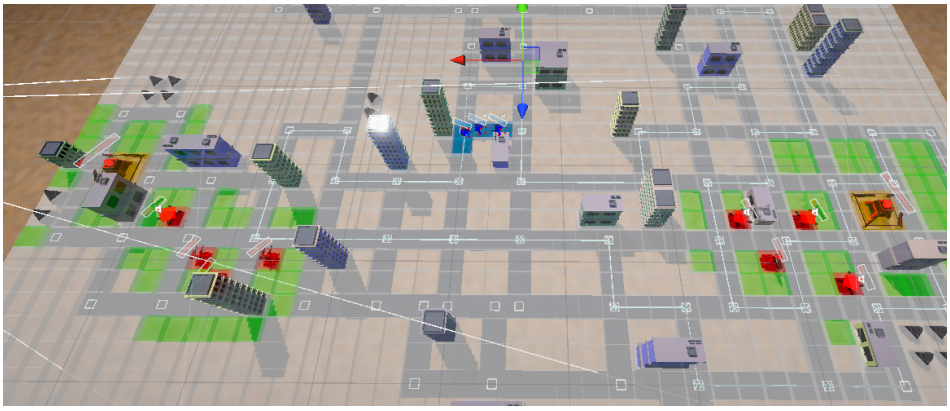

Figure 4.21: Visualisation of the Grid Sensor component during runtime

I created events each time I wanted to give a reward to the artificial intelligence. As previously explained, the objective of the artificial intelligence is to hold out as long as possible against the player's actions. That is why every five seconds an event is triggered that gives a positive reward to the artificial intelligence. I also give positive rewards every time a turret is able to destroy a unit and obviously when the player is defeated. In turn, I give negative rewards every time the player manages to destroy a turret, a greater negative reward when he destroys one of the two bases and an even greater negative reward when the player wins the game. I know that the artificial intelligence has a secondary victory condition which is to fill 75% of the map with turrets. For this reason, every time the artificial intelligence builds a turret I also give a positive reward. With this I finish with the configuration of our neural network inside the Unity environment.

With all these parameters, I would have an artificial intelligence configured with a fully functional neural network. The problem is that by taking completely arbitrary values at the beginning of the training, the artificial intelligence does not challenge the player in the first few games. This is why I came up with a way to perform Imitation Learning [18]. With the ML Agent libraries, I can add a component called *Demonstration Recorder* [6] with which I can record a basic setup for the AI to use as the basis of its training. To do this, I must overwrite the *Heuristic()* [9] method in our agent script. Inside, the functionality that was done in the *AIActionSystem* is incorporated. In this way I can record a basic behaviour with which the AI is able to recognise the patterns of its demonstration in order to train faster. In addition I set up a way to randomise the player's actions. So, infinite games can be generated without needing input from a player. Evidently, for the final game these parameters were removed although they are still available via a training flag.

To finish parameterising our artificial intelligence, I only need to configure a *Q-Agent* file that will be in charge of managing the simulations. Therefore, in order for the model to be able to use Imitation Learning [18], I must add some configuration parameters to

this file. This will allow us to configure what percentage of the demonstration the AI will use to make its decision and what percentage it will use its own heuristics.

With all this configured, I created a build with our training flag activated and proceeded to create a training to generate a neural network competent enough to have the basic rules of the game clear. The AI was able to achieve an average reward of 90% with this configuration. This seemed to us to be a fairly acceptable result to start working towards. Finally, this neural network was incorporated into our *Behaviour Parameters* [4] component so that each execution the artificial intelligence would continue learning but avoiding those first arbitrary results in the first cycles of the general intelligence.

4.2 Results

After the completion of this project I have obtained an artificial intelligence based on Machine Learning. This AI has all the information for making decisions throughout the duration of the video game. The Machine Learning procedure leverages the existing artificial intelligence systems within Unity to adapt its course and furnish the agent with cognitive capabilities, enabling it to respond to environmental cues in an unpredictable manner based on its condition or other variables.

On the other hand, some difficulties were faced in various stages of our technical demo. However, I solved them by using design patterns and writing a clean and understandable code. The fact that I decided from the beginning to separate visuals from functionality has helped us to accomplish our mission. Speaking of AI, creating the grid classes was beneficial for generating the inputs I needed to pass to the Machine Learning process. On the other hand, having all AI behaviour incorporated into multiple scripts assigned to the same Game Object aided us in analysing possible outcomes. Finally, concentrating the output into a single function was key to optimising learning as much as possible. All of this was possible because before the development of Machine Learning, I incorporated an option to test all AI functionalities. Additionally, multiple on-screen outputs, such as the squares where towers can be built, helped us confirm that decision-making was correct.

I have also implemented a system that was not planned for the agent. Using the ObservatorAI system, the agent can perceive its environment, and also with a small configuration, I can when I need to pass an input for the AI, generating less stress to the project. Through this, I have achieved that the agent can perceive where the convoy is moving around the environment, and if it is not a threat to their bases, continue its expansion to achieve its goal of invading 75% of the map. In this way, if the player finally goes for their bases, he will encounter a fortified defence placed by the AI.

While the primary application of this artificial intelligence is envisioned for a tower attack video game, certain elements implemented in this technical demo hold potential utility across various other video games.

- The path generation of the convoy can be applied to any type of tower defence video games.

- The convoy movement can be applied for Real Time Strategy games with a convoy system for fast travel though the environment.
- The AI functionality can be applied for doing a multiplayer game. We simply have to delete AI behaviour and make changes for another player to control this behaviour.
- The structure of the video game can be applied to any strategy games.

CHAPTER 5

Conclusions and Future Work

Contents

5.1 Conclusions **45**
5.2 Future work **46**

In this chapter, the conclusions of the work, as well as its future extensions are shown.

5.1 Conclusions

At the outset of this project, I was not entirely confident that I would achieve satisfactory results. Learning new methodologies from scratch, especially under time constraints, heightened my sense of insecurity. However, the learning curve proved to be manageable, and after some trial and error, I gained a better understanding of the procedure of this project.

Facing initial difficulties ultimately proved beneficial. Embracing new design patterns and adopting clean coding practices allowed me to accelerate the development during the project's later stages. Additionally, I was pleasantly surprised to find that much of the knowledge I gained during my time in college could be directly applied to the Unity engine. Unity itself is an excellent game engine, offering simplicity in use and delivering impressive results with minimal effort.

While implementing artificial intelligence in Unity presented its challenges, understanding its various systems and its interactions proved to be highly valuable. Ultimately, Unity's versatility enabled me to achieve results that would have been much harder to realise in other game engines.

The AI I created exceeds the standards of many strategy games in the video game market, yet it is not flawless. I am rather pleased with the accomplishments achieved in this demo, such as being able to incorporate reinforcement learning into the primary project. Thanks to this project, I have come to the realisation that I have a strong passion for artificial intelligence. Nevertheless, I acknowledge that there is still much more for me to learn. Thus, I will persist in my studies and strive to enhance my skills in order to effectively communicate my ideas.

5.2 Future work

There is still a lot to work on this technical demo to become a real game. The first thing would be to polish certain aspects of the user interface, such as the menus and functioning. On the other hand,an improvement of the path generation and convoy movement would be beneficial. I also want to keep digging more into Machine Learning, as I know I can get more potential out of it and improve the internal mechanism that will improve the agent. Additionally, adding more player units and turrets for the AI, and the ability to improve the ones that are on the field are things that I keep in mind. Finally, the aim would be to develop the demo and publish the complete game on Steam. So, this technical demo is a good start to seek funding that will allow me to continue with the development of the video game.

Bibliography

[1] Astute Analytica. Mobile tower defense games market: A comprehensive review of the global market 2022–2030. https://medium.com/@AstuteAnalytica24/mobile-tower-defense-games-market-a-comprehensive-review-of-the-global-market-2022-2030-cb89751322e2. Accessed: 2024-02-28.

[2] Mike Cannon-Brookes and Scott Farquhar. https://www.atlassian.com/es/software/jira.

[3] Blog de CEUPE. Aprendizaje por refuerzo: Concepto, características y ejemplo. https://www.ceupe.com/blog/aprendizaje-por-refuerzo.html?dt=1714411958969. Accessed: 2024-04-20.

[4] ML Agent Documentation. Class behaviorparameters. https://docs.unity3d.com/Packages/com.unity.ml-agents@2.0/api/Unity.MLAgents.Policies.BehaviorParameters.html. Accessed: 2024-04-24.

[5] ML Agent Documentation. Class decisionrequester. https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.MLAgents.DecisionRequester.html. Accessed: 2024-04-24.

[6] ML Agent Documentation. Class demonstrationrecorder. https://docs.unity3d.com/Packages/com.unity.ml-agents Accessed: 2024-04-24.

[7] ML Agent Documentation. Grid sensors for unity ml-agents - version 2.0. https://github.com/mbaske/grid-sensor. Accessed: 2024-04-24.

[8] ML Agent Documentation. Method collectobservations. https://docs.unity3d.com/Packages/com.unity.ml-agents@3.0/api/Unity.MLAgents.Agent.CollectObservations.html. Accessed: 2024-04-24.

[9] ML Agent Documentation. Method heuristic. https://docs.unity3d.com/Packages/com.unity.ml-agents@3.0/api/Unity.MLAgents.Agent.Heuristic.html. Accessed: 2024-04-24.

[10] ML Agent Documentation. Method onactionreceived. https://docs.unity3d.com/Packages/com.unity.ml-agents@3.0/api/Unity.MLAgents.Agent.OnActionReceived.html. Accessed: 2024-04-24.

[11] Unity Documentation. Animation rigging. https://docs.unity3d.com/Packages/com.unity.animation.rigging@1.3/manual/index.html. Accessed: 2024-03-10.

[12] Unity Documentation. Collision. https://docs.unity3d.com/Manual/collision-section.html. Accessed: 2024-03-20.

[13] Unity Documentation. Line renderer component. https://docs.unity3d.com/Manual/class-LineRenderer.html. Accessed: 2024-03-31.

[14] Unity Documentation. Scriptable object. https://docs.unity3d.com/Manual/class-ScriptableObject.html. Accessed: 2024-04-05.

[15] Unity Documentation. System requirements for unity 2022.3. https://docs.unity3d.com/Manual/system-requirements.html#player. Accessed: 2024-04-21.

[16] GanttProject. https://www.ganttproject.biz/.

[17] Javatpoint. Unity gameobjects. https://www.javatpoint.com/unity-gameobjects. Accessed: 2024-04-26.

[18] Bernard Marr. What is ai imitation learning – a super-simple guide anyone can understand. https://bernardmarr.com/what-is-ai-imitation-learning-a-super-simple-guide-anyone-can-understand/. Accessed: 2024-04-24.

[19] Mobidictum. Rock paper scissors ai. https://www.linkedin.com/pulse/rock-paper-scissors-ai-mobidictum/. Accessed: 2024-04-26.

[20] Code Monkey. Code monkey channel. https://www.youtube.com/watch?v=zPFU30tbyKs&list=PLzDRvYVwl53vehwiN_odYJkPBzcqFw110. Accessed: 2024-04-13.

[21] Overleaf. https://es.overleaf.com.

[22] Mike "Sorian" Robbins. Neural networks in supreme commander 2. https://ubm-twvideo01.s3.amazonaws.com/o1/vault/gdc2012/slides/Summit_AI/Robbins_Michael_Off

[23] Steam. Anomaly 2. https://store.steampowered.com/app/236730/Anomaly_2/. Accessed: 2024-04-26.

[24] Unity. Ml-agents videos. https://unity.com/products/machine-learning-agents. Accessed: 2024-04-23.

[25] A. Vannieuwenhuyze. *Inteligencia Artificial Fácil: Machine Learning y Deep Learning Prácticos*. ENI, 2020.

[26] Wikipedia. Bloons TD 6. https://en.wikipedia.org/wiki/Bloons_TD_6. Accessed: 2024-04-26.

[27] Wikipedia. Company of heroes 2. https://en.wikipedia.org/wiki/Company_of_Heroes_2. Accessed: 2024-04-26.

[28] Wikipedia. Functional requirements. http://en.wikipedia.org/wiki/Functional_requirements. Accessed: 2024-04-21.

[29] Wikipedia. Non-functional requirements. http://en.wikipedia.org/wiki/Non-functional_requirement. Accessed: 2024-04-21.

[30] Wikipedia. Singleton pattern. https://en.wikipedia.org/wiki/Singleton_pattern. Accessed: 2024-04-26.

[31] Wikipedia. Snake (video game genre). https://en.wikipedia.org/wiki/Snake_(video_game_genre). Accessed: 2024-04-26.

[32] Wikipedia. Starcraft ii. https://en.wikipedia.org/wiki/StarCraft_II. Accessed: 2024-04-26.

[33] Wikipedia. Supreme commander 2. https://en.wikipedia.org/wiki/Supreme_ Commander_2. Accessed: 2024-04-26.

[34] Wikipedia. Technology demonstration. https://en.wikipedia.org/wiki/Technology_demonstration. Accessed: 2024-04-26.