NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

JOÃO FILIPE PEREIRA RODRIGUES

Barchelor in Computer Science and Engineering

3D SENSING CHARACTER SIMULATION USING GAME ENGINE PHYSICS

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
December, 2022

DEPARTMENT OF
COMPUTER SCIENCE

3D SENSING CHARACTER SIMULATION USING GAME ENGINE PHYSICS

JOÃO FILIPE PEREIRA RODRIGUES

Barchelor in Computer Science and Engineering

Adviser: Rui Nóbrega
Assistant Professor, NOVA School of Science and Technology

Examination Committee:

Chair: Maria Armanda Grueau
Associate Professor, NOVA School of Science and Technology

Rapporteur: Rui Prada
Associate Professor, IST, Lisbon University

Adviser: Rui Nóbrega
Assistant Professor, NOVA School of Science and Technology

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
December, 2022

3D Sensing Character Simulation using Game Engine Physics

Copyright © João Filipe Pereira Rodrigues, NOVA School of Science and Technology, NOVA University Lisbon.
The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

Acknowledgements

To my adviser, Rui Nóbrega, who was always supportive and available to guide me throughout the entire process.

To my mother and father, who have always been supportive of my academic decisions and cared for me throughout the entire degree. To my brother who always supported my decisions and was always there to listen to me whenever I needed.

To my close friends and colleagues, who made my experience in FCT NOVA a memorable one, and were always there to help me when I needed and made my days much more enjoyable.

Abstract

Creating visual 3D sensing characters that interact with AI peers and the virtual environment can be a difficult task for those with less experience in using learning algorithms or creating visual environments to execute an agent-based simulation.

In this thesis, the use of game engines was studied as a tool to create and execute visual simulations with 3D sensing characters, and train game ready bots. The idea was to make use of the game engine's available tools to create highly visual simulations without requiring much knowledge in modeling or animation, as well as integrating exterior agent simulation libraries to create sensing characters without needing expertise in learning algorithms. These sensing characters, were be 3D humanoid characters that can perform the basic functions of a game character such as moving, jumping, and interacting, but also have simulated different senses in them. The senses that these characters can have include: touch using collision detection, vision using ray casts, directional sound, smell, and other imaginable senses. These senses are obtained using different game development techniques available in the game engine and can be used as input for the learning algorithm to help the character learn. This allows the simulation of agents using off-the-shelf algorithms and using the game engine's motor for the visualizations of these agents. We explored the use of these tools to create visual bots for games, and teach them how to play the game until they reach a level where they can serve as adversaries for real-life players in interactive games.

This solution was tested using both reinforcement learning and imitation learning algorithms in an attempt to compare how efficient both learning methods can be when used to teach sensing game bots in different game scenarios. These scenarios varied in both objective and environment complexity as well as the number of bots to access how each solution behaves in different scenarios. In this document is presented a related work on the agent simulation and game engine areas, followed by a more detailed solution and its implementation ending with practical tests and its results.

Keywords: 3D Sensing Characters, Game Bots, Game Engine, 3D Animation, Simulation Visualization, Agent Simulation,

Resumo

Criar visualizações de personagens 3D com sentidos que interagem com colegas de IA e com o ambiente virtual pode ser uma tarefa difícil para programadores com menos experiência no uso de algoritmos de aprendizagem automática ou na criação de ambientes visuais para executar simulações baseadas em agentes.

Nesta tese foi estudado o uso de motores de jogos como ferramenta para criar e executar simulações visuais com personagens 3D, e treinar bots para jogos. A ideia foi usar as ferramentas disponíveis do motor de jogos para criar simulações visuais sem exigir muito conhecimento em modelação ou animação, para além de integrar bibliotecas de simulação de agentes externas para criar personagens com sentidos sem precisar de conhecimentos em algoritmos de aprendizagem automática. Estas personagens 3D são humanoides que podem desempenhar as funções básicas de uma personagem de um jogo como mover, saltar e interagir, mas também terão simulados neles diferentes sentidos. Os sentidos que estas personagens podem ter inclui: o tato, colisões, visão, som direcional, olfato e outros sentidos imagináveis. Estes sentidos são obtidos usando diferentes técnicas de desenvolvimento de jogos disponíveis no motor de jogos, e podem ser usados como inputs para os algoritmos de aprendizagem automática para ajudar as personagens a aprender.

Esta solução foi testada usando algoritmos de *Reinforcement Learning* e *Imitation Learning*, com o intuito de comparar a eficiência de ambos os métodos de aprendizagem quando usados para ensinar bots de jogos em diferentes cenários. Estes cenários variaram em complexidade de objetivo e ambiente, e também no número de bots para que se possa visualizar como cada algoritmo se comporta em diferentes cenários. Neste documento será apresentado um estado da arte nas áreas de simulação de agentes e motores de jogos, seguido de uma proposta de solução mais detalhada para este problema.

Palavras-chave: Personagens 3D Sensoriais, Bots de Jogos, Motor de Jogos, Animação 3D, Visualização de Simulações, Simulação de Agentes,

Contents

List of Figures

List of Tables

Acronyms

ABMS	Agent-Based Modeling and Simulation 1
AEC	Agent-Environment Cycle 23
EDTI	Exploration via Decision-Theoretic Influence 12
EITI	Exploration via Information-Theoretic Influence 12
HOV	High Occupancy Lanes 9
HRI	Human-Robot Interaction 16
MARL	Multi-Agent Reinforcement Learning 13, 22
NEI	Nuclear Engineering Institute 17
POMDP	Partially Observable Markov Decision Process 21
POSG	Partially Observable Stochastic Games 23
RL	Reinforcement Learning 21
RTE	Real-time Embedded systems 17
TRE	The Robot Engine 16
VoI	Value of Interaction 12

1 Introduction

Agent-based simulations are the representation of the actions and interactions of autonomous agents in a set environment and is used to study and understand the behavior of a system and what controls its outcomes [29]. Over the years, this simulation model has grown in popularity since it can be used for numerous areas. In biology, agent-based simulations can be used to study phenomena such as forest insect infestations [38] or invasive species [1]. Still in relation with natural sciences, these simulation can also be used in epidemiology in order to study and predict the progression of epidemic diseases[1]. On other areas such as economics, agent-based models and simulation have been used in an attempt to predict market progressions [43], or areas such as traffic management for better road planning [14], amongst others. Nowadays, research in this area focuses on both improving and creating algorithms to make more realistic simulations, and reducing the number of iterations necessary for the simulation to reach a point where the results are satisfactory.

Currently, these simulations are being employed in several areas such as the study of traffic flow by simulating driver agents in order to design better road networks improving the overall quality of life [14]. There is also a large number of studies in crowd behaviors, building evacuations, and others using a game environment, taking advantage of the game engine's tools to create a graphical simulation [9]. With the increase of computational power over the years many studies have also emerged using agent-based models in trying to predict the evolution of stock markets to make a profit out of it, as well as predicting the spread of pandemic deceases in an effort to help contain the spread of said deceases. In the gaming area, these Agent-Based Modeling and Simulation (ABMS) have also been emerging due to the entertainment value that is the combining of gaming with agent simulation. Many of the games that exist so far with the incorporation of ABMS also have the purpose of investigating certain social behaviors[2], such as Sugarscape which simulates an artificial society to study human social phenomena. Another game with

[1]Agent Based Epidemic Model, https://cloud.anylogic.com/model/6362c090-dfba-49c1-b071-e48d520cbec9?mode=SETTINGS, Last Access: February 2022

[2]Agent Based Modeling games, https://inesad.edu.bo/developmentroast/2013/09/5-agent-based-modeling-games-that-teach, Last Access: February 2022

this purpose is SimPachamama[3], which is a policy game where the player takes the role of mayor of a small Bolivian community, who has to put into place different policies in order to achieve maximum well being of citizens while reducing deforestation of the surrounding Amazon.

1.1 Motivation

Games have always used interactive characters and Non-Playable Characters (NPCs) to make the game more interesting, appealing and relatable. Agent and Character Simulation has been used over time for many different purposes [29], from scientific research, such as predicting the spread of pandemics [22], to leisure [20] with games having NPCs to add more interactivity or make the game harder. Using these simulations requires a vast knowledge of the different AI, machine learning, and deep learning algorithms needed for these simulations. Additionally, many of these simulations also need graphic environments to show the results with rendering techniques and physics simulations. All these techniques require specialized knowledge that for a computer graphics developer may not be available, specially in the AI and Machine Learning fields.

Another factor that is also present in agent simulation, is the need to create sensing characters. Many of the ABMS scenarios require characters that interact with the environments or with their peers. To achieve this, the creation of characters that can sense their surroundings by simulating vision, touch, distance amongst other factors is needed. Creating these senses from scratch also needs complex knowledge in programming things such as ray tracing for vision and distance, or physics for touch.

All these mentioned needs create a problem for those who want to create character simulations but do not have the technical expertise to develop all the aforementioned technologies. To solve the problems created by the need for a graphical environment and creating sensing characters, it can be interesting to use game engines to visualize the environment and simulate the senses. Being in constant development to facilitate the process of creating games, game engines seem to be a good tool to create and train characters and environments since they already have render and physics engines that implement virtual environment rules such as ray cast distance, gravity or collisions. Besides the use of game engines, over the years many tools have been created to facilitate the creation and execution of these simulations by providing utilities that facilitate the testing and sharing of different machine and deep learning algorithms such as OpenAI Gym [8] and PettingZoo [39].

Taking these factors into consideration, in this work we studied the use of a game engine to create, train and visualize simulated sensing characters and the surrounding environment, as well as seeing how effective it was to create game ready bots. To solve the problem of the expertise needed for the agent simulation algorithms, we used some

[3]Simpachamama, https://www.inesad.edu.bo/2018/03/11/simpachamama/, Last Access: February 2022

of the many external libraries that have been made that allow using AI and machine learning algorithms without having to develop them. In this work, the use of tools like Unity3D, and machine learning libraries, such as OpenAI gym and Unity Machine Learning Agents, was explored to create game bots that learn how to play games with the aid of these algorithms.

1.2 Research Questions

To follow the idea of creating sensing learning bots and visualizing character simulations in a game engine with resource to external AI and Machine Learning libraries, in this section some research questions followed by a more detailed explanation will be presented.

To guide the research work and the design and development of the proposed experiment, the following research questions were defined:

1. How to integrate off-the-shelf agent simulation algorithms in a game-engine-based simulation environment.
2. How do the bots though with different learning techniques behave in different game scenarios.
3. Is it worthwhile to use machine learning to teach bots to play games.

The first question aims to find how to use already existing agent simulation libraries in a game engine to provide an easy to use tool to visualise agent simulations without the need for complex knowledge of these algorithms. Finding how to incorporate the different libraries in a game engine, and then how to communicate the state of the agents and environment between game engine and library in order for the library to simulate and the game engine to show the simulation is key to solve this problem.

The next question is related to comparing different learning algorithms, such as reinforcement learning and imitation learning, in different game scenarios and see how they fair off in games with varying agents and environment complexity, as well as compare their progression with increasing training time. With this it becomes possible to assess which learning techniques are recommended for teaching bots in different scenarios.

The last question is about whether using reinforcement learning and imitation learning techniques to train and teach bots to play the game is worthwhile compared to hard-coding the bots behaviour as done traditionally in games. With this question we aim to discuss the level of difficulty of efficiently implementing these techniques to the difficulty of manually coding every behavior of the bots.

1.3 Objectives

In order to address the research questions, this dissertation focused on using a game engine as a tool to create simulations and exploring agent simulation libraries that can be used in diverse tools such as game engines. Therefore it set out to study and implement the creation and execution of sensing character simulations in a game engine to use it as a tool of visualization, with the use of simulation libraries to allowed testing different off-the-shelf algorithms without having to develop much code.

Given the complexity of the stated problem, this thesis was split into several objectives:

- **Creating simulation environments:** Using the game engine's graphical tool to build environments for the simulations. The environment is one of they key elements of the simulations since its components will interact and influence the agents.
- **Creating simulation agents:** Using human-like game-objects as a base to create agents. Animating and coding all the interaction the agent has with the environment as well as other agents, as without it there is no simulation.
- **Integration of AI and machine learning libraries:** Integrate agent simulation libraries in the game engine, in order to run and visualize the created simulation in it using different algorithms.
- **Creation of different game scenarios:** Created diverse game scenarios that allow to test the algorithms for teaching bots in varying conditions to see which algorithms perform in different conditions.
- **Visualize and document results:** Visualize the simulations running using different algorithms in different game scenarios and document relevant results in order to compare the different algorithms in metrics such as success of the agents in performing their task and time to learn.

The desired tools to elaborate this theses were the use of a 3D game engine with incorporated physics simulation and a machine learning library that allows the execution of the simulation only by sending it the information of the agents and environment, and adjusting parameters without the real need to develop any machine learning algorithm.

1.4 Contributions

The main contributions of this dissertation is the creation of game scenarios for the execution and visualization of agent simulations. This game scenarios allowed us to explore a specific use of these agent simulations that was teaching bots how to play games, and use it to test the use of game engines as a graphical tool for the simulations. These scenario were simple games that any person without any gaming expertise could easily play.

Besides the creation of game scenarios we also tested different machine learning algorithms by using the before mentioned game scenarios. After developing the game scenarios, we used them to test how two different learning algorithms, reinforcement and imitation learning, would behave when teaching bots how to play these games. By testing the algorithms in different games it was possible to find some conclusions of what influences the learning effectiveness and of how effective these methods are in different games.

The work developed in this dissertation was used for the publication of the paper by João Rodrigues and Rui Nóbrega in 2022 entitled "Character Simulation Using Imitation Learning With Game Engine Physics", published in the proceedings of the 4th International Conference on Graphics and Interaction (ICGI 2022).

1.5 Document Structure

Following the introductory chapter, this thesis contains three additional chapters. The next chapter presents the state of the art review, where the related research areas are analyzed, beginning with agent simulation, its history, and current interaction methods and applications, followed by game engines, giving a brief introduction to them followed by works done in animation and simulation using them. The chapter ends with a revision of tools and libraries to join agent simulation libraries with game engines.

The third chapter describes with more detail the proposed experiment, by exposing its requirements, architecture, and the theory behind the concepts needed to implement the games with learning bots. In the fourth chapter, it is firstly described the set of games chosen to perform the agent based simulations and training, followed by a description of how each game was implemented and how the training process occurred.

In the fifth chapter, it is started by explaining which metrics were used to evaluate the each of the training processes, followed by the exposure of the experimental results accompanied by a discussion of their values. In the sixth and last chapter a conclusion about the entire process and its results, as well as the answer to the proposed research questions is presented.

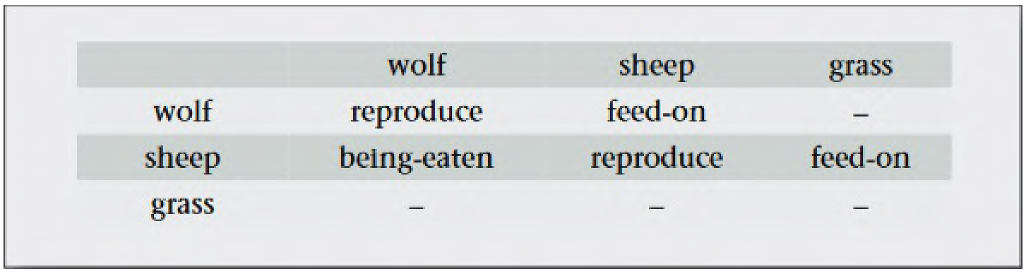
2 State of the Art Review

The purpose of this chapter is to provide an overview of the areas of study related to the subject of this thesis, analyzing their history and also relevant current works that can be used as a basis for the development of the work at hand. To start this chapter, we will begin by seeing a brief definition and history of Agent Simulation, since this is what is needed to accomplish the simulation of the 3D sensing characters. In addition, an analysis of different techniques and procedures to execute Self-play and Imitation on AI agents, as well as some works and projects made with this will be made. Then, we will proceed to a brief explanation of game engines. These are the base for the planned simulation platform, so understanding how they work, as well as seeing how they have been used in prior works for animation and simulation is fundamental. Lastly, some machine learning libraries will be reviewed, since these are needed to supply the simulation platform, with the required learning algorithms.

2.1 Agent-Based Models and Simulation

Agent-Based Modeling and simulation is a paradigm in which simulated humans, animals, or other forms of beings are modeled as agents that interact with their peers as well as their environment. In these agent-based simulations, sometimes called multi-agent simulation, the environment plays a crucial role since it influences all the agents and their interactions and therefore must be carefully taken into account. As said by Franziska Klugl and Ana Bazzan [23], the key idea behind ABMS is to use the simulated agents to produce a phenomenon that should be analyzed, reproduced, or predicted. In ABMS, agents are the active components or decision-makers being modeled and implemented using agent-related concepts and technologies. Rather than simply describing the overall global phenomenon, said phenomenon can be generated from the actions and interactions of the agents in the simulation. "ABMS is particularly suitable for the analysis of complex adaptive systems and emergent phenomena in social sciences, traffic, biology, and others" [23].

In 2009 Charles Macal and Michael North [29] asked the question of why is ABMS



	wolf	sheep	grass
wolf	reproduce	feed-on	–
sheep	being-eaten	reproduce	feed-on
grass	–	–	–

Figure 2.1: Interactions in a prey-predator model [23]

becoming widespread. In their words, it's because we live in an increasingly complex world. The systems that are needed to analyse are becoming more complex in terms of their inter-dependencies and older modeling tools are no longer as efficient as they once were. This authors state that some systems have always been too complex to adequately model, such as economic markets that traditionally relied on the notions of perfect markets, homogeneous agents, and long-run equilibrium because these assumptions made the problems analytically and computationally tractable. With the use of ABMS some of this assumptions can be relaxed and take a more realistic view of these economic systems. Lastly, the increase of computational power has made it possible for these models and simulation to grow, as it is possible to compute large-scale simulation models that just weren't plausible years ago.

In order to create an agent-based simulation, four elements have to be taken into consideration[24]. The first one is the set of agents in the simulation. These are autonomous and independent of the other entities within the simulated environment. Secondly comes the specification of the interactions between themselves and with the environment, since these interactions produce the overall outcome. Next comes the simulated environment which contains all other elements that can be re resources, objects with no active behavior, as well as global properties. Lastly comes the simulation infrastructure, which is the tools used to run and view the simulation. To illustrate what the elements are in a simulation and how they relate, consider a simple environment that simulates the relation between predators (wolves) and prey (sheep) [23] as seen in Figure 2.1. The set of agents are the wolves and the sheep. The interactions of the agents consist of doing a random movement, and when they are close to one another, they would procreate if they are the same species, or eat in the case of predator and prey. In addition, when the sheep are close to an environment object that represents their food (i.e grass), it also eats the grass. The simulated environment of this scenario consists of the spatial representation where grass objects are scattered, and possibly some global variables associated with temperature or humidity to influence the availability of grass. The simulation infrastructure could be the Unity game engine, where we would model the described elements and execute the simulation.

The precise definition of an agent is not something globally agreed upon. It is the subject of occasional debate as the issue arises when one claims that their model is "agent-based"or when one is trying to discern whether such claims made by others have validity. The use of the term "agent-based"to describe a model carries important implications. "Some modelers consider any type of independent component whether it be software or a model to be an agent. An independent component's behavior can range from simple in nature, e.g., described by simple if-then rules, to the complex, e.g., described by complex behavioral models from the fields of cognitive science or artificial intelligence. Some authors insist that a component's behavior must also be adaptive in order for it to be considered an agent. In this view, the agent label is reserved for components that can learn from their environment and dynamically change their behaviors in response to their experiences." [29]. In this thesis, the following set of properties and attributes will be used to describe agents, taken from the description of agents used in Charles Macal and Michael North's work [29].

- An agent is autonomous and self-directed. An agent can function independently in its environment and its interactions with other agents, generally from a limited range of situations that are of interest.
- Agents are modular or self-contained. An agent is an identifiable, discrete individual with a set of characteristics or attributes, behaviors, and decision-making capability. The discreteness requirement implies that an agent has a boundary in a sense and one can easily determine whether something (that is, an element of the model's state) is part of an agent, is not part of an agent, or is a characteristic shared among agents.
- An agent is social, interacting with other agents. Agents have protocols or mechanisms that describe how they interact with other agents, just as an agent has behaviors. Common agent interaction protocols include contention for space and collision avoidance; agent recognition; communication and information exchange; influence; and other domain or application-specific mechanisms.
- Agents interact with their environment as well as with other agents. An agent is situated, in the sense that its behavior is situationally dependent, which means that its behavior is based on the current state of its interactions with other agents and with the environment.
- An agent may have explicit goals that drive its behavior. The goals are not necessarily objectives to maximize as much as criteria against which to assess the effectiveness of its decision and actions. This allows an agent to continuously compare the outcomes of its behaviors to its goals and gives it a benchmark for possibly modifying its behavior.
- An agent may have the ability to learn and adapt its behaviors based on its experiences. Individual learning and adaptation require an agent to have memory, usually in the form of a dynamic agent attribute.

As said in chapter 1, this system of ABMS in practice is used in a wide range of areas.

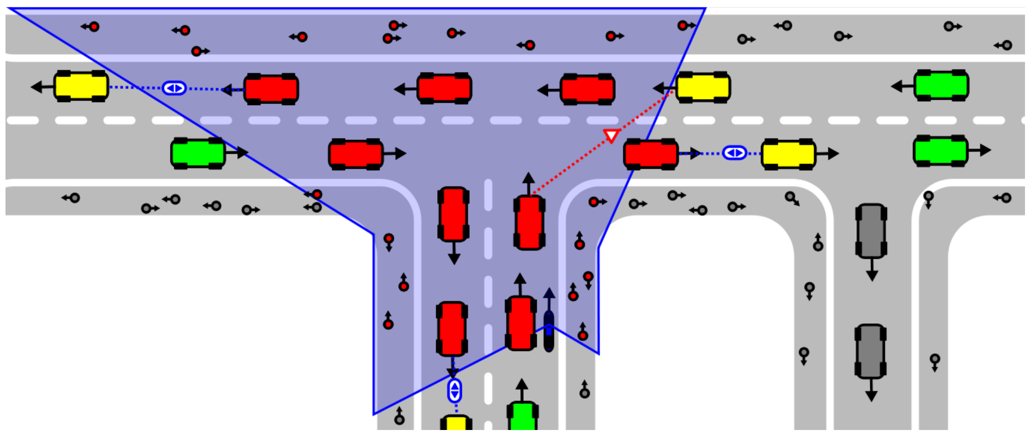


Figure 2.2: Still image of traffic simulation performed by Institute of Visual Computing[1]

One of the areas mentioned was traffic management, on which Priyadarsini Ghadai et al. [14] performed a study showing how simulations using the agent-based model could help traffic engineers reduce congestion, in particular by the use of High Occupancy Lanes (HOV) lanes. To execute this first it was needed to model the vehicles with maximum speed, acceleration, and deceleration. Then the drivers, where their decisions were split into macro and micro-decisions. Macro decisions consisted in defining the destination and route taken, and micro-decisions consist of the actions taken by the driver at each point of time, such as accelerating, overtaking, and turning. Afterward, the environment consists of the road network on which vehicles will travel that is made up of link segments that have one or more lanes, operate in one or both directions, and have properties such as length, number of lanes, and speed limit. Then the approach for the simulation was to treat vehicles as individual units instead and analyze what behavior emerges when the vehicles are given simple rules to follow. Each vehicle moved according to the vehicle ahead, speeding up or slowing down to match its speed while maintaining a safe distance. The authors of this work concluded that using this system for traffic modeling over the prior ones provided more realistic simulations, although to ensure an even more realistic simulation more entities would need to be added to the simulation such as busses, trucks, emergency vehicles, pedestrians and many others that affect the traffic flow in real life. In Figure 2.2 we see an example of a traffic simulation done by Professor Dr. Rainer Herpers and Sven Seele that include some of these extra entities in having pedestrians and different kinds of vehicles.

2.1.1 Self Play Simulation

Self-play reinforcement learning is when agents learn by exploring and playing with themselves. It has seen success when applied in many game scenarios, however, the

[1]Agent Based Traffic Simulation, https://vc.inf.h-bonn-rhein-sieg.de/?page$_id = 1025, *LastAccess* : *February*$2022

Figure 2.3: OpenAI Five agents playing Dota[2]

process for self-play learning is unstable and more sample-inefficient than general reinforcement learning especially when used in imperfect information games [3]. AI learning using direct supervision will not scale to unbounded complex tasks, many have worked on unsupervised exploration and skill acquisition methods. Current indirect exploration methods scale poorly with the environment complexity and differ from the way organisms evolve on Earth. “The vast amount of complexity and diversity on Earth evolved due to co-evolution and competition between organisms, directed by natural selection”. Multi-agents auto-curricula have been used to solve the various type of multiplayer games both in classic discrete games (Backgammon [40], Chess [36]) and continuous real-time games (Dota, Starcraft) as illustrated in Figures 2.3 and 2.4.

Environment affects the capacity for qualitative observation as the more complex the environment is, it becomes harder to see whether the actions taken are positive or not. Simpler environments produce easier-to-evaluate scenarios. “as environments increase in scale and multi-agent auto-curricula become more open-ended, evaluating progress by qualitative observation will become intractable.” [3]. In 2019, OpenAI researchers Bowen Baker et al. [3] performed a study to show how effective self-play RL can me in teaching cooperative and competitive agents using, in which two teams of agents were placed in an environment to play hide and seek as shown in Figure 2.5. This agents had no knowledge in how to play hide and seek, and besides walking around they could also move a set of boxes and ramps and lock them in place. This agents didn’t have any

[2]OpenAI Dota Simulation, https://openai.com/blog/openai-five/, Last Access: February 2022

[3]Starcraft agent simulation, https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii, Last Access: February 2022

Figure 2.4: AlphaStar simulation on StarCraft II[3]

incentive in interacting with the world objects, the score system only considered whether the hiders were not yet found, awarding them one point and and punishing the seekers with a negative point, or whether the seekers had found the hiders, switching the points awarded around. Simulating this scenario over and over again, it was possible to see that the losing team was the one that would learn new strategies. At first the seekers were consistently winning so the hiders learned to move boxes to block seekers entry to their zone, then seekers had to learn how to use ramps to climb boxes, and it got going to the point were the agents would find bugs in the phisics programing and would send objects flying out the map, or they would send themselves flying to get somewhere unreachable by normal means.

In 2020, another work by Łukasz Kaiser et al. was done on agents learning using RL [21], this time teaching agents how to play different atari games. This study however aimed at understanding why AI takes longer to play a game than a human and showing how video prediction models can enable agents to learn atari games in fewer iterations. Lukasz Kaiser says that humans learn how to play a game faster than a simulated agent because humans possess an intuitive understanding of the physical processes that are represented in the game: we know that planes can fly, balls can roll, and bullets can destroy aliens. We can therefore predict the outcomes of our actions. Using policies that allow for video prediction, it was shown that agents learned how to play atari games in way lesser iteration than previous studies done before.

Throughout this thesis, we've only been focusing on agent-based simulation in a high-level view, mentioning definitions, utilities, and works done in the area with a focus on the results of the simulations and not so much on the algorithms that lie under. Though

[4]OpenAI Hide and Seek, https://openai.com/blog/emergent-tool-use/, Last Access: February 2022

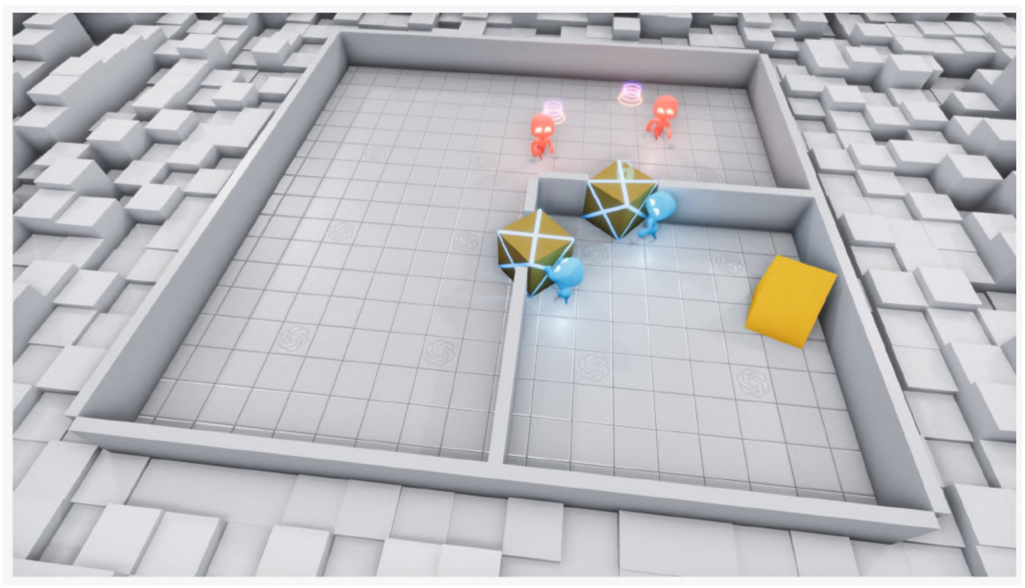


Figure 2.5: Hide and Seek simulation performed by OpenAI[4]

the focus here is in that high-level view, this wouldn't be possible if others hadn't done the low-level work, and studied the actual algorithms involved in executing these simulations. In 2020 Tonhhan Wang et al. [44] made a study comparing the use of two algorithms for influence-based multi-agent exploration, Exploration via Information-Theoretic Influence (EITI) and Exploration via Decision-Theoretic Influence (EDTI), showing how to optimize each of these algorithms. The first algorithm, EITI, uses mutual information to capture interdependence between the transition dynamics of agents, while EDTI uses Value of Interaction (VoI), to characterize and quantify the influence one agent's behavior has on the expected returns of other agents. By optimizing either as a regularizer to the value function, agents are encouraged to explore where they can exert influence on other agents for learning sophisticated multi-agent cooperation strategies. After optimizing the regularizers and performing tests on different scenarios they concluded that the use of both of these facilitates effective exploration on all the tasks by exploiting interactions, and in the comparison between the two, they concluded that EDTI performs slightly better due to its ability to filter out interaction points that lead to no reward points.

In the same year, Jiachen Yang et al. [45] showed how the CM3 architecture (Cooperative Multi-Goal Multi-Stage Multi-Agent) learns significantly faster than direct adaptations of existing algorithms on three challenging multi-goal multi-agent problems. These problems were cooperative navigation in difficult formations, negotiating multi-vehicle lane changes in the SUMO traffic simulator[5], and strategic cooperation in a Checkers environment. Their work focused on the multi-goal multi-agent setting where each agent is assigned a goal, and then it must learn to cooperate with other agents with possibly different goals. To do this Jiachen Yan described the complete CM3 learning framework

[5]Simulation of Urban Mobility, https://sumo.dlr.de/docs/index.html, Last Access: February 2022

by defining a credit function, deriving a new cooperative multi-goal policy gradient with localized credit assignment, motivating the possibility of significant training speedup via a curriculum for Multi-Agent Reinforcement Learning (MARL), describing function augmentation, and finally synthesizing all components into a synergistic learning framework. After doing this and running experiments on the defined problems, they concluded that CM3 obtained significantly higher performance, faster learning, and overall robustness than other existing MARL methods, displaying the strengths of both independent learning and centralized credit assignment.

2.1.2 Imitation Learning

Imitation learning is a learning paradigm in which agents learn by mimicking a given behaviour by mapping the set of observations to the actions taken from a set of demonstrations (the mappings to learn from) [18]. This paradigm is gaining popularity because it facilitates teaching of complex tasks with minimal expert knowledge of them. Generic imitation algorithms can also reduce the problem of having to design reward functions by teaching using only the demonstrations available. Within the paradigm a distinction can be made by algorithms that aim to mimic humans such as Generative-Adversarial Imitation Learning (GAIL) and Behavioral Cloning, and algorithms that mimic other agents such as Self-Imitation Learning.

Imitation Learning can be performed either by mimicking demonstrations provided by human agents or artificial agents [47]. In 2018, Junhyuk Oh et al. [31] made a study about self-imitation learning, where artificial agents gradually learned how to play Atari games by mimicking their best past behaviour. The idea with this work, is that exploring past good experiences indirectly lead to a deeper exploration depending on the domain. To further explain this, Junhyuk Oh et al. gave an example of the Atari game *Montezuma's Revenge* where at a certain point the agent needs to pick up the key and open the door. According to the author "Many existing methods occasionally generate experiences that pick up the key and obtain the first reward, but fail to exploit these experiences often enough to learn how to open the door by exploring after picking up the key."... "On the other hand, by exploiting the experiences that pick up the key, the agent is able to explore onwards from the state where it has the key to successfully learn how to open the door.

2.1.3 Visualization of Agent Learning

In this thesis, we focus on the visualization of simulated agents rather than on the learning algorithms themselves. To provide a graphical visualization of a simulation there are a lot of tools available from game engines, to python coding using simulation libraries (more on these in section 2.2.1). In this section we will look at some projects that provide a real-time graphical view of the simulation, focusing on that visual component and not on the simulation itself.

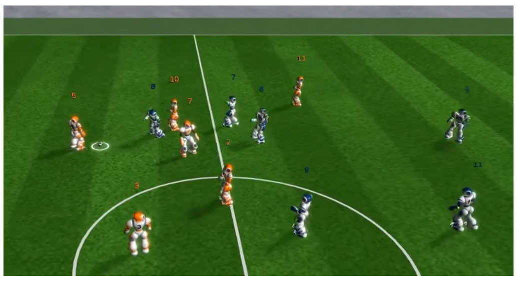


Figure 2.6: RoboCup 3D Simulation 2019 final[6]

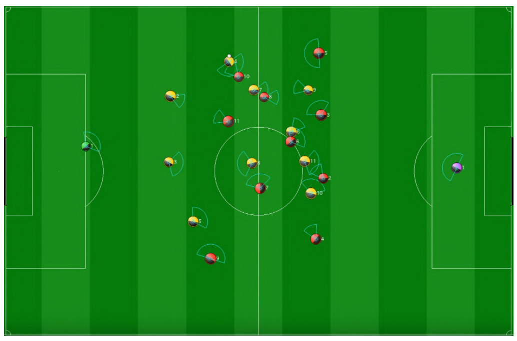


Figure 2.7: RoboCup 2D Simulation 2019 final[7]

An interesting project to look at is the RoboCup [33]. Since 1997 it has been held the robot cup, in which robots play a soccer league. In this tournament each team programs their robots, aiming for them to learn soccer better than the other team's robots, and was made in proportionate evolution in learning algorithms expectation each year's robots to be better and better [27]. Although robot learning is not the aim of this thesis, what latter was added to the robot cup is. In past years it was created the simulation league, where this time instead of actual robots, simulated robots competed in the game of soccer. This category has both the 2d simulation and the 3d simulation. The past editions of these competitions can be watched online, and see how the simulations are shown. These simulations use a platform created by robocup. In figures 2.6 and 2.7 is shown the visual representation of RoboCup 3D and 2D simulations respectively, that allow the spectators to watch the match and see how each simulated team plays.

Another work that allows for an interesting visualization of agent simulations is the one already mentioned in section 2.1.1 done by Bowen Baker et al. [3] from OpenAI, where it is possible to see how agents play hide-and-seek as seen in Figure 2.5 and analyze their learning only from watching the simulation, without needing to analyze numbers to make conclusions. For this, they used python as their main tool for showing the simulations and using Mujoco library for the agent's movements and OpenAI Gym for the simulation algorithms.

2.2 Game Engine Visualization

A game engine's main use is to produce games by rendering its graphics, producing music and sound effects, and allowing external manipulation of the scene by reading inputs from an input device. Usually, a game engine is made up of Rendering Engine, Animation Engine, Physics Engine, Artificial Intelligence Engine, Network Engine, 3D Sound Engine, and a Map Editor. The rendering engine renders the objects out on the

[7]RoboCup 3D simulation, https://www.youtube.com/watch?v=edHyjLC49G4, Last Access: February 2022

[7]RoboCup 2D simulation, https://www.youtube.com/watch?v=BVMat$_{h}Axss, LastAccess : February$2022

screen, the animation engine expresses the object's motion, the physics engine is responsible to simulate an object's gravity, weight, collision, parabola, and centrifugal force. The AI engine controls every entity (NPC) not manipulated by the player, the network Engine makes each user contact the server sharing one space and interaction based on network and the sound Engine generates the game sound data to the game progress state [30].

Although the main purpose of a game engine is, as its name says, to create games, the features of one allow for more uses such as animations and simulations. The render, animation, and physics engines can facilitate the production of animation, since one can just pick up a scene to animate, code the intended movements, and then just play out the result which will follow physics without having to animate frame by frame every movement, and collision. In the same line as animation, these tools can be used to produce simulations. If instead of coding every movement to occur, the focus is on coding how different materials behave and then creating scenarios with different placements of said materials, the result is a simulation of how the materials behave in contact with each other. A more concrete example of this is a fluid simulation [17], where you code different behaviors for liquids and then simulate how the liquid behaves in a given scenario. With the use of the AI engine an agent simulation can be created [20]. This time the focus is on coding the agent behavior and learning algorithms and then playing out the scene with the resulting simulation.

What makes the use of a game engine alluring for these animations and simulations is how it allows a user to view these creations, without having to create low-level code to render and animate the scene since the game engine deals with this internally. For agent simulation, it interests the most how a game engine allows manipulating the motion of the agent's models and how to create a simulation environment for the agents to play out simply by adding game-objects to a scene, and manipulating it's properties with no need for extensive lines of code describing the shape, physics and motion of every object in the simulation.

2.2.1 Human-like Models and Animation

Another functionality that makes a game engine useful for the creation of digital content is that it allows the creator to import resources from external sources, making it possible to develop a project without having all the skill-sets needed for it as long as it's available online for use or purchase. This means that there is no need to know how to create a model, or animated since there are lots of these available in assets stores. Although it's nice to be able to import models and animations, someone had to do it to make it available, raising the questions of why is it important to know and understand how to make and manipulate humanoid models, and how to do it. Wenheng Chen [10] said that "Human motion modeling is crucial in many areas such as computer graphics, vision and virtual reality"in his work on diversified human notion generation. With the progress in graphic computing over time, society has begun to expect more and more realistic human-like graphics in their digital entertainment, thus making it crucial to know how to create humanoid models, with realistic motion animation.

In 2015 Cristoph Bartneck et al. [4] wrote a paper on the use of Unity 3D game engine to control human-like robots. In regards to using a 3D game engine, they said that "Unity 3D allows non-programmers to use a set of powerful animation and interaction design tools to visually program and animate robots.", and were motivated by the fact that most of the tools used for Human-Robot Interaction (HRI) are prototype hardware and software packages that most require classical programming in order to control the interactions. With this work, they aimed at building software that would allow similar ease at controlling robots. To achieve this they created The Robot Engine (TRE), built on top of Unity 3D since it possessed the desired properties of having an easy-to-use graphical user interface for animating and controlling interactions, the ability to communicate with external hardware, and process multimedia sensory data, and is available for multiple operating systems. In Figure 2.8 it's exemplified how TRE uses the different components to control a robot using Unity as its base. As the main component, Unity acts as the central part of the system, using the scene as a means to show the state of the robot and manipulate it with the help of the Animator controller, taking inputs to form speakers and camera as other means to control the robot, and communicating with the robot using an Arduino.

Using the structure in Figure 2.8, TRE was built, and in order to use it, one simply had to import the TRE package into unity, place the human-like models into the scene, define the parameters on the script, test the connection with the Arduino device, and the TRE is ready for use. The only thing a user needs to do in order to use TRE is to create the animation controller using Unity's tools. To test this, Cristoph Bartneck et al. imported a model of a Lego Robot and of an InMoov robot[8], and tested using Unity 3D to control both of these. After this, they concluded that using a game engine as software to control humanoid robots has the advantages of making it easier to develop stand-alone software and that it already contains many tools that have been developed specifically for animating biological life forms.

Focusing more on the digital part of animation, another use that game engines have seen growing is for creating animated films. Both 3D games and animated films have one common goal that is high-quality 3D video sequences [2]. The difference in achieving this same goal is that video games require real-time rendering, making it that they often sacrifice quality to get better frame rates. On this topic, Artur Bąk and Marzena Wojciechowska [2] wrote that the evolution of game engines, in order to aid game developers to achieve more realistic 3D graphics, have allowed their use as a tool for animated films. The main advantage of the mage engine is the generation of a smooth sequence of high-quality 3D frames in real time [19]. It also allows for easy change of the scene configuration, where the view is obtained almost immediately. With this, the use of a

[8]InMoov robot, http://inmoov.fr/, Last Access: February 2022

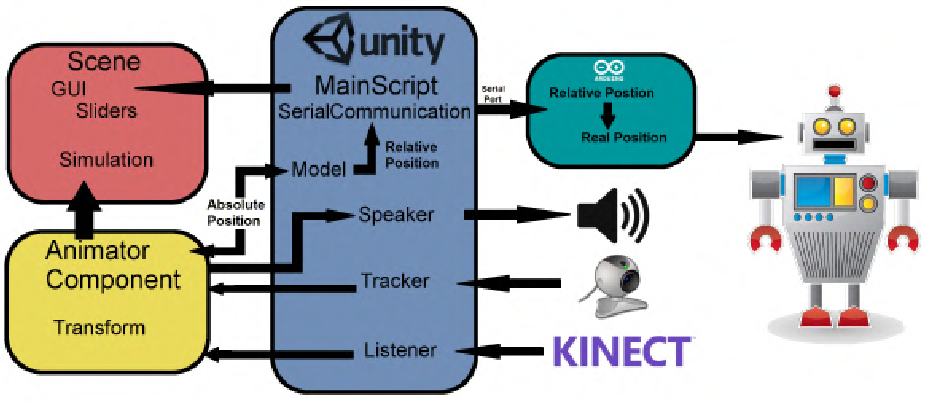


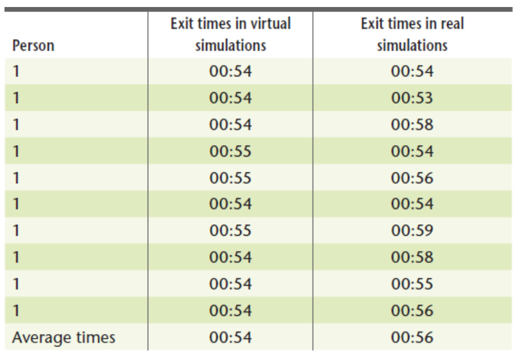
Figure 2.8: Flow diagram of The Robot Engine [4]

game engine does not necessarily cut steps from the traditional linear pipeline of creating animated films [13], but allows for many of these to be carried out simultaneously. In their work, Artur Bąk and Marzena Wojciechowska listed some benefits of using a game engine as a tool for creating animation films. This included the interactive creative process with ease of editing and fast output, earlier creative decision making, reduced production time and reusable assets, and brand consistency. But using a game engine for animation also comes with drawbacks, mainly being the quality, since algorithms for real-time rendering won't be as high quality as algorithms for offline animation [2].

2.2.2 Visualization of Simulations

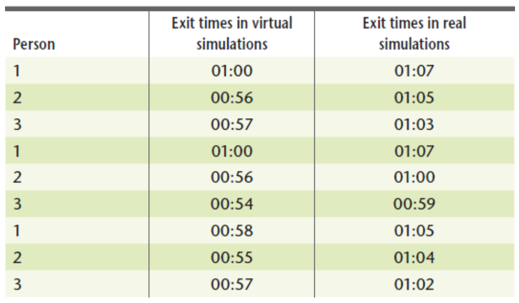
One other application that game engines have seen rising over the years is their use for all kinds of simulations. According to Michal Pasternak et al. "Simulation and visualization are some of the main methods for testing and validation in the context of the development of Real-time Embedded systems (RTE), especially when testing with real hardware is costly or dangerous" [32]. Being game engines powerful visualization tools, they make a useful tool to simulate these RTE systems in a cost-free way. Besides this, game engines can also be used for simulation of other types of systems such as virtual reality, material simulations, agent simulations, as seen throughout this thesis, and some others.

In 2008 Antônio Mól and Carlos Jorge [9] made a research on the use of game engines for VR simulations in order to help in evacuation planning for buildings. In this research they used a 3D model of the Nuclear Engineering Institute (NEI) of the Brazilian Commission of Nuclear Energy, to perform preliminary evacuation tests using the VR Simulation prior to real-life evacuation tests. Their motivation to use a game engine to develop the VR simulations is that they are independent of specific scenarios, meaning that researchers can use the engine's source codes to create totally new scenarios and



Person	Exit times in virtual simulations	Exit times in real simulations
1	00:54	00:54
1	00:54	00:53
1	00:54	00:58
1	00:55	00:54
1	00:55	00:56
1	00:54	00:54
1	00:55	00:59
1	00:54	00:58
1	00:54	00:55
1	00:54	00:56
Average times	00:54	00:56

Figure 2.9: Exit times in single person exits [9]



Person	Exit times in virtual simulations	Exit times in real simulations
1	01:00	01:07
2	00:56	01:05
3	00:57	01:03
1	01:00	01:07
2	00:56	01:00
3	00:54	00:59
1	00:58	01:05
2	00:55	01:04
3	00:57	01:02

Figure 2.10: Exit times in multiple people exits [9]

applications. Besides this, game engines are relatively inexpensive, most of them being free and only need to pay for resources you can't create yourself. Lastly is that game engines include networking capabilities, which allow for multi-user simulation. Antônio Mól and Carlos Jorge used the Unreal Engine to build their simulations. In it, they used the 3D Model of NEI and coded the VR simulation. The interesting take in this work is that the evacuation times obtained in the VR simulation and real-life were similar, with the difference in average time in one person exits being two seconds, and in multiple people exits being around 7 seconds as seen in Figures 2.9 and 2.10.

Liliana Zarco et al. [46] wrote in 2021 about the advantages and setbacks of using game engines simulations for ultra-flexible production environments, as well as comparing different game engines for modeling and controlling these simulations. The general idea of ultra-flexible production environments is to use mobile platforms to carry parts from workstation to workstation in an automated manner [7]. An example of these ultra-flexible production environments can be seen in Figure 2.11. In the words of Liliana Zarco "Ultra-flexible production environments require platforms capable of representing, modeling, controlling, and visualizing highly complex systems. Game engines have evolved rapidly in recent years, proving that they can represent very complex systems in a reliable and user-friendly way." [46].

Gamification for controlling these ultra-flexible production systems makes use of video game elements and their development environments in the technical context of the factory with the objective of creating precise control and a positive experience for the user. By creating an enhanced visual experience, operators can efficiently perform tasks and acquire new skills. In addition, game engines can exploit resource redundancy, allowing efficient integration of new resources into simulations. According to these authors, one of the great potentials of game engines in this area is the fast development and data redundancy since, nowadays, developers can reuse significant portions of their core software components and invest in custom software engineering. Besides this, game engines allowing hierarchical data architectures, reuse of resources, and also rearranging its relations helps with the development of these systems since they use hierarchical models that have

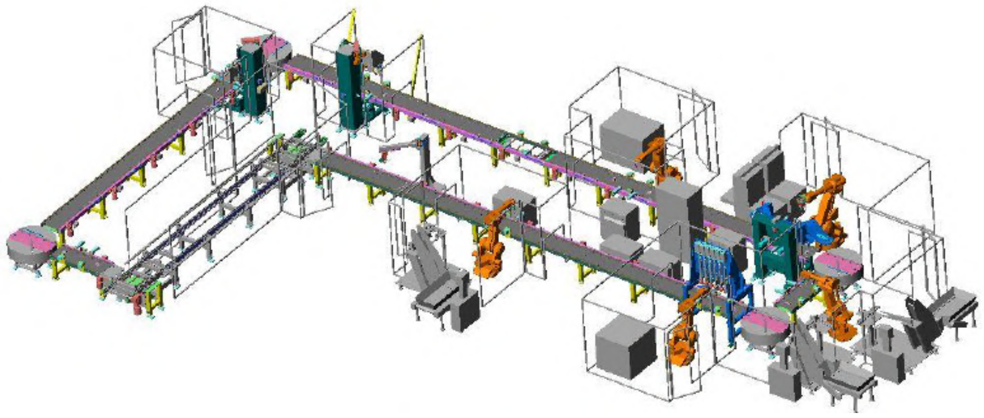


Figure 2.11: Ultra Flexible Production Environment in a robotic assembly line [7]

already been standardized, such as the Unified Robot Description Format which is an XML format for representing a robot model.

As for the setbacks of using game engines for ultra-flexible production systems, the authors states that the integrated physic engines have limitations for these systems. These simulations require lots of precise different physics such as the ability to detect collision between a relatively high number of objects (between 250 and 500) at interaction frame rates from 60Hz to 1kHz and the computed responses of dynamic objects to collisions have to result in a realistic behavior. Besides this, the movement constraint between objects must obey their defined limits and the game engine's physics should support the realistic simulation of a screw mechanism, performing a stable collision and friction computation for complex geometric objects. The problem is, that no game engine physics motor excels at all of these needs of the system, some excel in realistic collisions, others in optimization for higher frame rates, but no game engine can perform all of these simulations at the desired level.

Some other works have been done concerning the use of game engines for simulations. SimGen is a tool for generating simulations and visualizations on the Unity Game Engine. In general, this tool creates scripts to execute a simulation of real-time embedded systems in unity, using predefined meta-objects [32]. In Figure 2.12 is shown the simulation of a rover executed in a unity build with the use of the generated scripts. To control the simulated rover, the execution a java file in the CMD is needed, connecting it with the right IP and ports, and use commands defined in the meta-objects to control the simulation.

In the area of fluid simulation, Wefeng Hu et al. [17] did research in the use of game engines in order to achieve real-time interaction fluid simulations. According to the authors, although there are many real-time simulations using animation software, there is a lack of interactive simulations. In their works, they used the Unity3D container fluid simulation combined with the Smoothed Particle Hydrodynamics method [28] to create

[9]Rover Simulation using SimGen, https://youtu.be/4ROt2N6i6KA, LastAccess: February 2022

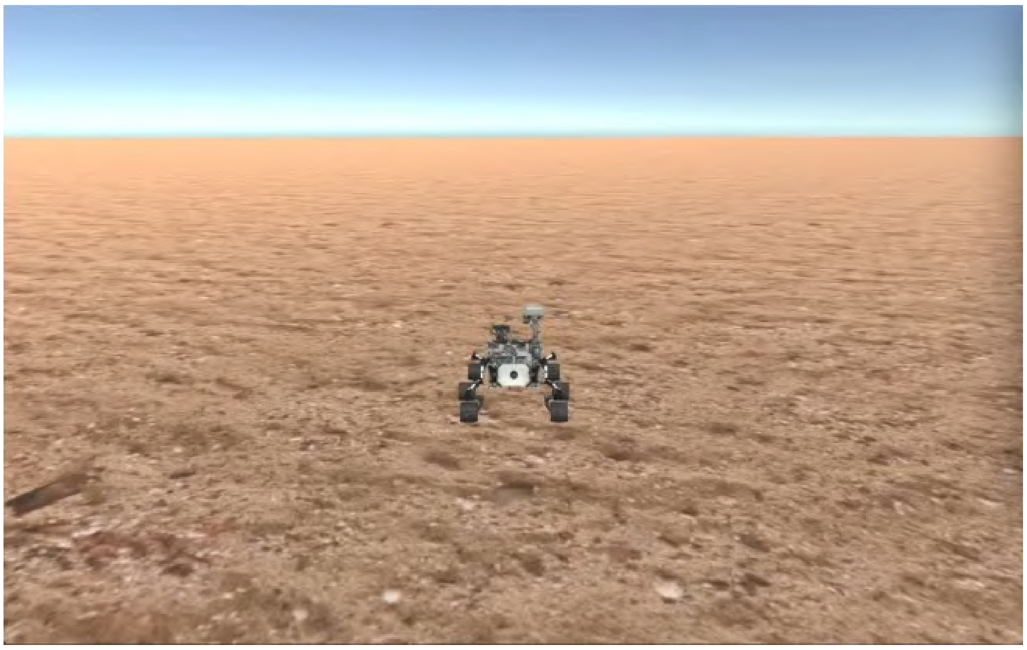


Figure 2.12: SimGen generated simulation of a rover in Unity[9]

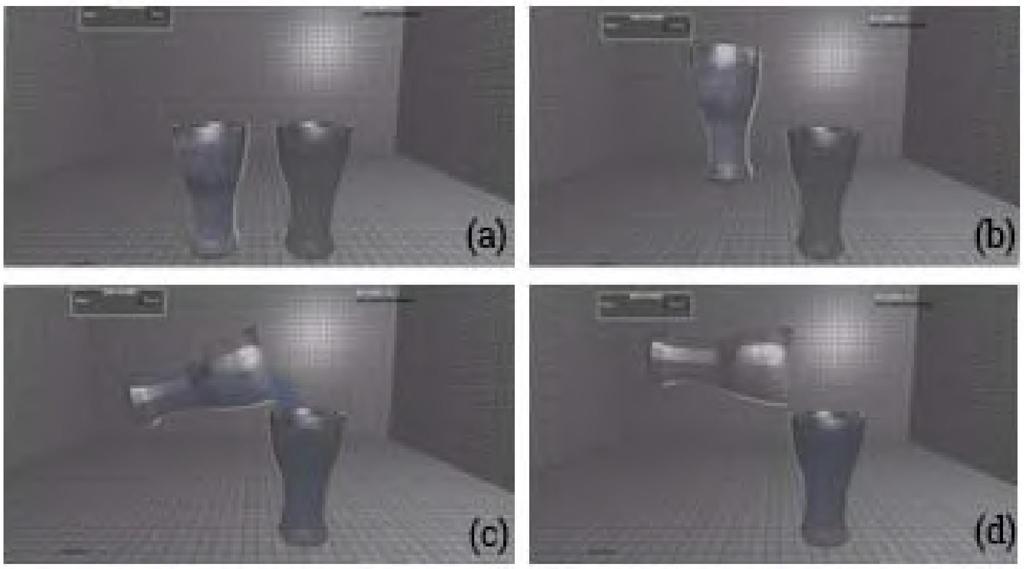


Figure 2.13: Interactive simulation of liquids [17]

a 3D real-time interaction of pouring liquid from one container to another. In Figure 2.13 it can be seen the final result of the interactive simulation of pouring water from one glass to another.

2.3 Visualization and Interaction with Simulation Libraries

In order to create sensing agents for our game engine environment, it is crucial to find a library that allows simulating the agent's actions without the need to write complex algorithms. Luckily, nowadays, there are an increasing number of these that are open source and allow for indiscriminate integration in any kind of work as long as it complies with the library specifications. In this sub-chapter we'll be looking into some of these libraries, what they offer and aim to achieve, a general view of how they work, and what is needed to integrate these libraries with a game engine.

2.3.1 OpenAI Gym

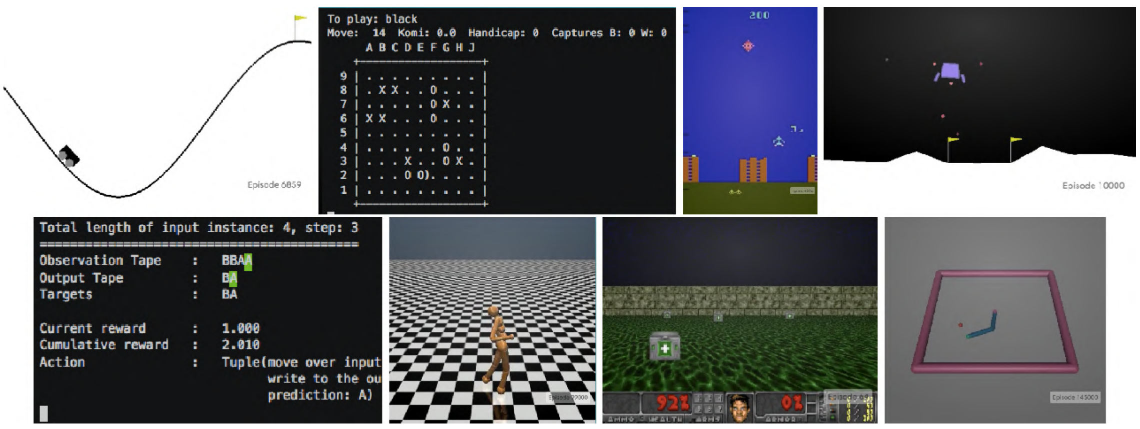
OpenAI Gym[10] is a toolkit for Reinforcement Learning (RL) research [8] as it includes a growing collection of benchmark problems that expose a common interface and a website where people can share results and compare the performance of algorithms. This library focuses on the episodic setting of RL, meaning that the agent's experience is broken into a series of episodes where the agent's initial state is sampled from distribution, and the interaction proceeds until the environment reaches a terminal state. The objective in episodic RL is to maximize the expectation of total reward per episode and achieve a high level of performance in as few episodes as possible.

When designing OpenAI gym, its authors made decisions based on their previous experiences developing RL algorithms and using previous benchmarks collections. One of these design decisions is providing abstraction only for the environment and not for the agents. According to Greg Brockman et al. [8] this was to maximize convenience to the users and allow them to implement different styles of agent interfaces as there are numerous ways of agents taking input and learning from it. Another of these design choices is to emphasize the sample complexity and not just the final performance. This means that instead of evaluating the algorithm solely based on the final performance score, OpenAI gym also takes into account the time it takes to learn (sample complexity). One way of doing this sample complexity evaluation is to define a threshold and count the number of episodes necessary for the score to reach that threshold. Lastly, they chose to have strict versioning for environments, meaning that if an environment changes in any way, its name will be updated with a new version number. This is because results between changing environments are incomparable since agents might act differently because of the changes.

OpenAI Gym contains a set of environments formalized as a Partially Observable Markov Decision Process (POMDP) [37]. See Figure 2.14 for an example of one of these environments. The following environments are some of the ones included as of now [8]:

- Classic control and toy text: small-scale tasks from the RL literature.

[10]OpenAI Gym website, http://gym.openai.com/, Last Access:February 2022



```
To play: black
Move:  14  Komi: 0.0  Handicap: 0  Captures B: 0 W: 0
     A B C D E F G H J
   +-------------------+
 9 | . . . . . . . . . |
 8 | . X X . . O . . . |
 7 | . . . . . O X . . |
 6 | X X . . . O . . . |
 5 | . . . . . . . . . |
 4 | . . . . . . O . . |
 3 | . . . X . . O X . |
 2 | . . . O O). . . . |
 1 | . . . . . . . . . |
   +-------------------+
```

```
Total length of input instance: 4, step: 3
==========================================
Observation Tape    :   BBAA
Output Tape         :   BA
Targets             :   BA

Current reward      :   1.000
Cumulative reward   :   2.010
Action              :   Tuple(move over input
                              write to the ou
                              prediction: A)
```

Figure 2.14: Environments included in the OpenAI Gym [8]

- Algorithmic: perform computations such as adding multi-digit numbers and reversing sequences. Most of these tasks require memory, and their difficulty can be chosen by varying the sequence length.
- Atari: classic Atari games, with screen images or RAM as input, using the Arcade Learning Environment [6].
- Board games: currently, we have included the game of Go on 9x9 and 19x19 boards, where the Pachi engine [5] serves as an opponent.
- 2D and 3D robots: control a robot in simulation. These tasks use the MuJoCo physics engine, which was designed for fast and accurate robot simulation [41]. A few of the tasks are adapted from RLLab [12].

In regards to integrating OpenAI gym with a game engine, it is a difficult task. In this work, we aim to find if using a game engine's graphical tools is a worthwhile way of creating environments and simulations with sensing agents, without needing much knowledge in simulation algorithms. Meanwhile, OpenAI gym focuses on simulation using already incorporated abstracted environments which complicate the use of environments created in game engines. Besides this, OpenAI gym also leaves to the user the programming of the agent's learning algorithms. Although they offer a community to share and compare algorithms, this still requires the user to have some knowledge in which algorithms to select.

To read more on works done with OpenAI Gym, one of them has been referenced in section 2.1.1, or go to OpenAI Gym website to see full documentation.

2.3.2 PettingZoo

PettingZoo[11] is a library of diverse sets of multi-agent environments using a Python API. PettingZoo was developed with the objective of accelerating research in MARL, by

[11]PettingZoo website, https://www.pettingzoo.ml/, LastAccess: February 2022

making work more interchangeable, accessible, and reproducible similar to what OpenAI's Gym did for single-agent RL [39]. Although PettingZoo's API inherits many features of a Gym, is unique amongst other RL APIs in that it's based around the novel Agent-Environment Cycle (AEC) games model. In the words of J.K Terry et al., the need for a library for MARL research comes from the fact that as of October 2021, "While a massive number of Gym-based single-agent reinforcement learning libraries or code bases exist (as a rough measure 669 pip-installable packages depend on it at the time of writing), only 5 MARL libraries with large user bases exist [25, 26, 34][12,13]"

When developing PettingZoo both as a library and an API, its authors centered it around two main goals. The first one is for it to be like a gym, in the sense they wanted the API to look and feel as if working with a Gym and include numerous reference implementations of games with the main package. Reusing as many design metaphors from Gym as possible will help its massive existing user base to understand PettingZoo's API without much trouble. The second goal is for PettingZoo to be a universal API for MARL, meaning that it needs to support all use cases and types of environments. With this in mind, several technically difficult cases arise that need to be carefully considered such as environments with large numbers of agents, with agent death and creation, where different agents can be chosen to participate in each episode, and learning methods that require access to specialty low-level features [39]. In addition to this, the other two softer goals that J.K Terry et al. had when developing PettingZoo, are ensuring the API is simple enough for beginners to easily use, and making the API easily changeable if the direction of research drastically changes.

As mentioned in section 2.3.1 for OpenAI, most single-agent RL libraries and APIs use environments based on POMDP, however, MARL does not have a universal mental and mathematical model. One of the most popular models for MARL is the Partially Observable Stochastic Games (POSG) [16]. In a POSG, all agents step together, observe together, and are rewarded together. This creates a couple of problems for MARL. The first problem is that POSGs don't allow access to all information you should have since the agent's rewards are summed and returned at once. In a multi-agent game, this combined reward is often the composite reward from the actions of other agents and one might want to attribute the source of this reward for various learning purposes. The second problem is that POSGs based APIs aren't conceptually clear for games implemented in code. The main example of this is race conditions which occurs because simultaneous models of multi-agent games are not representative of how game code normally works. The problem occurs when two agents are able to take conflicting actions (i.e. moving into the same space), and needs to be resolved by the environment (i.e. collision handling).

Motivated by the problems just mentioned, J.K Terry et al. developed the before mentioned AEC. The idea with AEC is that agents sequentially see their observation,

[12]Tianshou MARL API, https://github.com/thu-ml/tianshou, LastAccess: February 2022

[13]Autonomous Learning MARL library, https://github.com/cpnota/autonomous-learning-library, LastAccess: February 2022

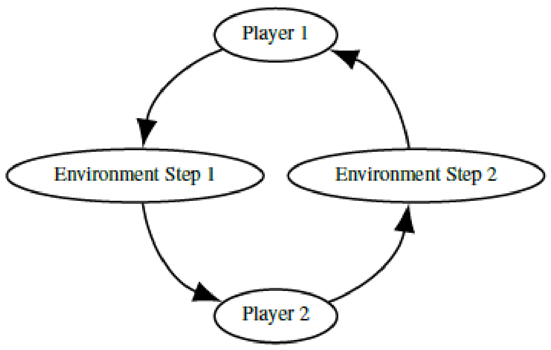


Figure 2.15: The AEC diagram of Chess [39]

take actions, are granted rewards from the other agents, and then the next agent to act is chosen. Another conceptual feature of this is the existence of the "environment"agent. When this agent acts in the model it indicates the updating of the environment itself, realizing and reacting to the other agent's actions. This allows for a more comprehensive attribution of rewards, cause of death, amongst others. Figure 2.15 is an example of the AEC diagram in a two-agent chess game. According to the authors, modeling multi-agent environments sequentially for APIs has the following benefits [39]:

- It allows for clearer attribution of rewards to different origins, allowing for various learning improvements.
- It prevents developers adding confusing and easy-to-introduce race conditions.
- It more closely models how computer games are executed in code.
- It formally allows for rewards after every step as is required in RL.
- It is simple enough to serve as a mental model, especially for beginners.
- Changing the number of agents for agent death or creation is less awkward, as learning code does not have to account for lists constantly changing sizes.
- It is the least bad option for a universal API, compared to simultaneous stepping. Simultaneous stepping requires the use of no-op actions if not all agents can act which are very difficult to deal with, whereas sequentially stepping agents that could all act simultaneously and queuing up their actions is not especially inconvenient.

As of now, PettingZoo has available many environments for its users. Like many other libraries, this list of environments contains a set of diverse Atari games such as Pong, Space War, Mario Bros, and many others. In addition, they offer some environments created by them called the butterfly environments. Then they also have environments for multiple classic games such as chess and backgammon. A list of configurable environments with massive numbers of particle agents is also offered. To top this off, they also offer a set of non-graphical communication tasks and three cooperative environments.

In regards to integrating PettingZoo with a game engine, although it has incorporated environments similar to what OpenAI Gym has, PettingZoo facilitates integration with outside environments by making environments expose only basic and essential functions making it easier to create a wrapper from an environment created to PettingZoo. This reduces the task of integrating PettingZoo's simulations to a game engine, by finding a way of wrapping the game engine's environments in a way PettingZoo's API will accept and manage the communication between both tools. As for the learning algorithm side of the problem, it's still up to the user to decide which algorithms to use in its agents, either by programming entirely or using available algorithms shared by the community.

2.3.3 Unity Machine Learning Agents

Unity Machine Learning Agents[14](ML-Agents) is a toolkit that allows games and simulation to serve as environments for training intelligent agents. This library is specific for Unity's 3D game engine and provides implementations of state-of-the-art that enable game developers and researchers to easily train agents for 2D, 3D, and VR/AR games. With ML-Agents, agents can be trained using reinforcement learning, imitation learning, neuroevolution, and some other methods with the provided simple-to-use Python API [20].

Having been developed specifically for a game engine, these trained agents can be used for multiple purposes such as controlling NPC behavior, automated testing of game builds, and evaluating different game design decisions pre-release. This toolkit is beneficial for both game developers and AI researchers as it provides a central platform where advances in AI can be evaluated on Unity's environments and then made accessible to game developer communities.

With ML-Agents, the training of the agent's behaviors using a variety of methods is possible, but to do so, it's needed to define three entities at every moment of the game (called the environment). The first is the observations, that is, what the agent perceives about the environment. These can be either numeric or visual observations. Numeric observations measure attributes of the environment from the point of view of the agent, while visual observations are images generated from the cameras attached to the agent and represent what the agent is seeing. Secondly, there is the action that the agent can take. For these, there is also a distinction between action existing continuous or discrete ones. The difference between both can be understood by the example of an agent moving in a grid or moving freely on the map. An agent moving in a grid can be seen as a discrete action where it matters only if he moves north, south, east, or west, while the agent

[14]Unity ML Agents website, https://unity.com/products/machine-learning-agents, Last Access: February 2022

moving freely can be seen as two continuous actions that determine direction and speed. Lastly, there is the reward signal, which is a scalar value that indicates how well the agent is performing. These rewards need not be provided at every moment but only when the agent performs an action that is either good or bad.

Having been developed by Unity3D, ML-Agents offers the following features[42]:

- 18+ example Unity environments
- Support for multiple environment configurations and training
- Flexible Unity SDK that can be integrated into your game or custom Unity scene
- Support for training single-agent, multi-agent cooperative, and multi-agent competitive scenarios via several Deep Reinforcement Learning algorithms (PPO, SAC, MA-POCA, self-play).
- Support for learning from demonstrations through two Imitation Learning algorithms (BC and GAIL).
- Easily definable Curriculum Learning scenarios for complex tasks
- Train robust agents using environment randomization
- Flexible agent control with On Demand Decision Making
- Train using multiple concurrent Unity environment instances
- Utilizes the Unity Inference Engine to provide native cross-platform support
- Unity environment control from Python
- Wrap Unity learning environments as a gym

To offer its diverse functionalities ML-Agents contains five high-level key components. The first one is the Learning Environment. which contains the Unity scene and all the game agents. The unity scene provides the environment in which agents observe, act, and learn. Secondly, there is the Python low-level API for interacting and manipulating the learning environment. This API is not a part of unity and is used to communicate and control the Academy during training, as well as using the API to use Unity as the simulation engine for own machine learning algorithms. Then there is the external communicator that simply connects the learning environment and the Python low-level API. Afterward, there are the Python Trainers which contain all the machine learning algorithms that enable training agents. Finally, there is the Gym Wrapper, which wraps the unity environment into OpenAI's gym. This however can only be used with single-agent environments.

On the ML-Agents website it is available many resources showcasing or teaching the utilities of ML-Agents. In it, it's possible to see some of the works done using this, such as AI learning soccer as illustrated in Figure 2.16, or agents learning how to fly an airplane or learning how to kart racing. Besides these gaming environments, ML-Agents has also been used in a project to automate the task of making breakfast, teaching agents how to flip pancakes from pan to plate and a robot that dodges obstacles to deliver butter. In addition, this library has also been used in teaching a robot to collect metallic waste,

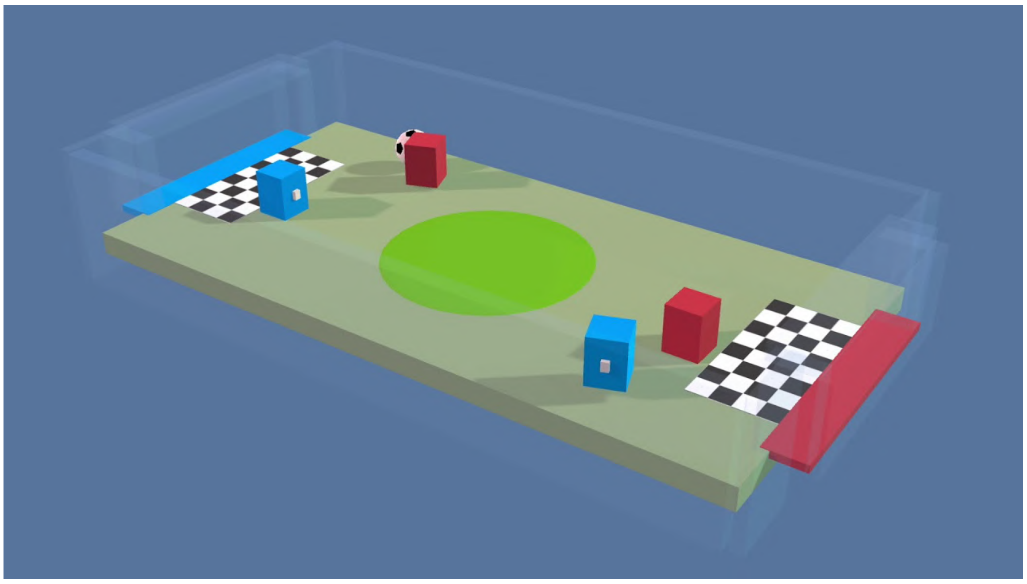


Figure 2.16: ML-Agents teaching AI to play soccer[16]

Table 2.1: Pros and Cons of Agent-Based Models and Simulations

Pros	Cons
Ability to Model Heterogeneous Populations	Weak dealing with Homogeneous Data
Models both Discrete and Continuous Models	Computationally expensive
Doesn't require knowledge of studied Phenomena	
Can incorporate randomness into Model	

meaning that with enough research this can one day be applied in a real-life scenario, and has also been used in teaching AI how to park a car. Besides these mentioned scenarios and others present on the website, a game that has used ML-Agents as a tool to improve its gameplay is also mentioned. This game is Source of Madness[15], which is a rogue-lite game created by Carry Castle that uses machine learning to procedurally generate millions of monsters, making it that you play against different enemies each play-through.

2.4 Discussion

In this section, a brief discussion about agent simulation and the different machine learning algorithms will be made.

Throughout this thesis, one of the topics talked about was agent-based simulations. This paradigm allows for modeling and simulating some more complex paradigms with

[15]Source of Madness game, https://sourceofmadness.com/, LastAccess: February 2022

more efficiency, and for this work in specific, it makes it possible to execute the simulation of sensing characters, having them learn their environments gradually by exploring it. Looking into some of the advantages and disadvantages of agent-based models and simulations, ABMS excel in modeling heterogeneous populations, meaning that it can perform well in scenarios where the agents perform and behave differently to each other, and where the agents interacting with different environments assets can change the outcome of the simulation. This advantage, make for a disadvantage in homogeneous models where agents are supposed to behave the same way every time independent of what happens in the environment. Besides this, ABMS can deal with both discrete and continuous models as seen in section 2.1.1 with works done in games such as Chess (discrete) and Dota (continuous) and can insert randomness into the model, which can simulate some factors impossible in deterministic models such as errors and other random events. Another interesting factor about agent-based simulations is that most of the time one does not need to know about the studied phenomenon to obtain results, since these are obtained through visualization and analysis of the progression in the agent's behavior. Although these simulations are appealing, the more complex they are, the more computational resources are needed to execute the simulations since simulations using agent-based models are computationally expensive. In Table 2.1 a summary of these pros and cons is listed.

In Table 2.2 some key comparisons between the three machine learning libraries seen in section 2.3 is made. As seen, OpenAI Gym has the disadvantages of not allowing for multi-agent simulations, as well as requiring reinforcement learning algorithms. In another hand, it contains sample environments for the user to test and allows connection to the Unity3D game engine using the Unity ML-Agents Wrapper. PettingZoo has the upsides of allowing multi-agents simulation as well as containing sample environments to test algorithms, but once more this one doesn't contain any sample RL algorithms and has the downside of, at the time of writing, not having any form of environment wrapper that allows connection with game engine environments. In relation to the downside of requiring the algorithms, both these libraries encourage their users to share their algorithms and result in their community, making it a bit easier for inexperienced users to find algorithms to use in their projects. Lastly, Unity ML-Agents has the big upside of being developed specifically for a game engine, making it easier than OpenAI to make the connection. Besides this, ML-Agents also allows multi-agent simulations and contains both sample algorithms and environments ready to use. Although not mentioned in the table, ML-Agents also have the possibility for learning through other methods besides RL, by having imitation algorithms.

[16]AI learns soccer using ML-Agents, https://www.youtube.com/watch?v=qN7umlE4ZmQlist=WLindex=85, Last Access: February 2022

Table 2.2: Comparative Table Between Machine Learning Libraries

	OpenAI Gym	PettingZoo	Unity ML-Agents
Single-Agent Simulation	Yes	Yes	Yes
Multi-Agent Simulation	No	Yes	Yes
Sample Environments	Yes	Yes	Yes
Sample Algorithms	No	No	Yes
Game Engine Connection	Yes	No	Yes

3 Simulation and Training Workflow

As mentioned in previous chapters, the goal is to create and run agent simulation loops using game engine physics and use this as a base to teach game bots using both reinforcement learning and imitation algorithms and compare both results in different types of games. To get a better grasp on how to do this, in this chapter we are going to look at the overall architecture and workflow of the simulations followed by more detailed concepts of how the simulations and the bots' training work. This will give a generic view of which kinds of games this training process should be effective.

3.1 Architecture

To create a game engine-based simulation loop that can be used to teach bots, two key components are needed: The Game Engine and a Machine Learning Library. The game engine is used to create the autonomous agents and the surrounding environment, as well as performing physics calculations that allow us to define the environment rules (collisions, speed, gravity) and emulate the agent senses. As seen in Figure 3.1 all of these calculations will produce observations, such as, which objects the agents see, and the distance to objects amongst others. These are provided to the Machine Learning library that will use the observations to calculate which actions to take and perform the learning according to the selected algorithm and configurations.

Besides these key components, human input can also influence the simulation. In reinforcement learning algorithms some predetermined knowledge can be provided to the agents to accelerate the learning process, although it's not necessary. In imitation learning algorithms based on human behavior, some sort of demonstration is required to allow the algorithms to emulate what humans would do during the course of the games.

Going into further detail in each component, the game engine is responsible for the agents, the environment, and physics calculations. The agents represent our game bots that have to learn to play games. With the help of the game engine physics motor, the bots can have emulated in them different senses such as vision, touch, and hearing assisted by ray-tracing, collision detection, and audio engines. The environment represents the game

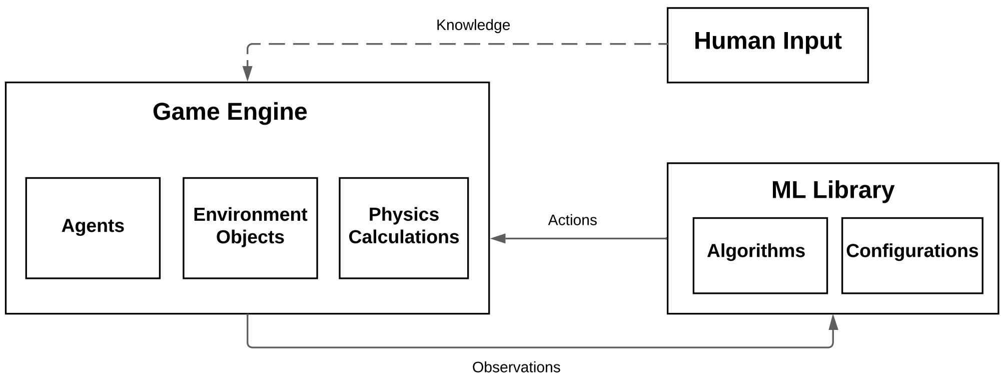


Figure 3.1: General architecture for the game engine based simulation loops

that is going to be played by the bots. If we take as an example the game of volleyball, as seen in Figure 3.2, the environment will be composed by a field, a ball, the net, an outside area, and borders that limit the space where bots can roam around. Besides the visual representation of the game, the environment is also what defines the game rule. In the example used these rules would be the regular rules of a volleyball game. The physics engine is responsible for handling the bots' movement, and collision detection amongst other possibilities such as the senses emulation as mentioned before.

The machine learning library is what is responsible for performing the bots' learning. This library should have available different off-the-shelf machine learning algorithms so that they can be used without requiring the expertise to develop the needed learning

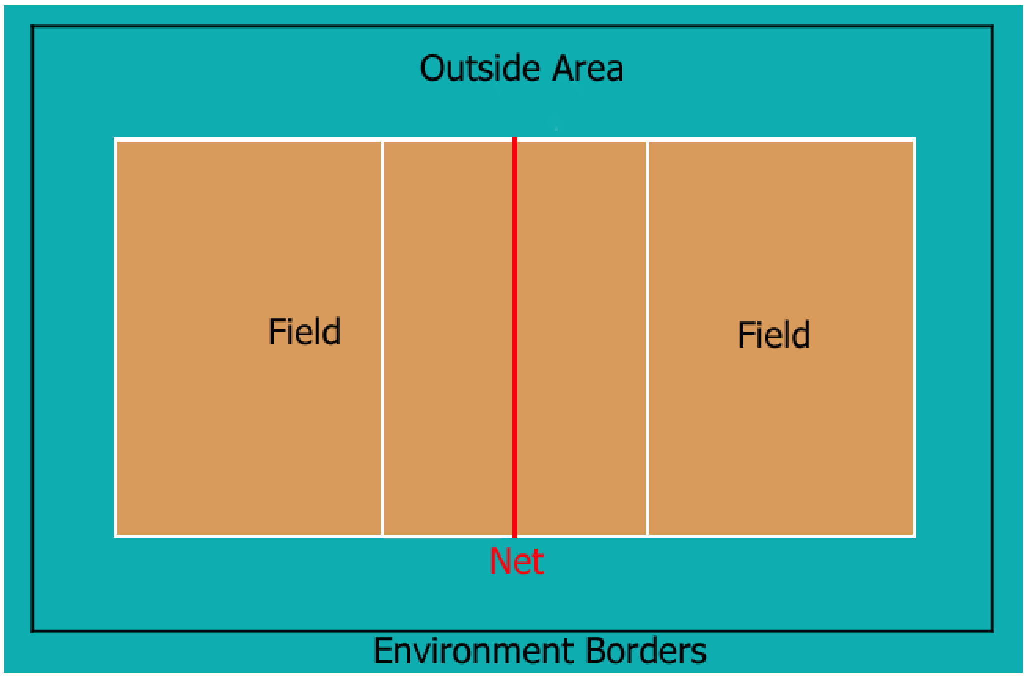


Figure 3.2: Environment representation of a Volleyball Game

algorithms. This library will receive from the game engine the different observations and rewards, and use them to compute which actions should be the correct ones and infer conclusions based on the rewards compared to the observations it has. This library should be configurable so that the users can select different algorithms, and change the parameters of the algorithm to the desired ones.

3.2 Simulation and Training

In this thesis, **Reinforcement Learning** and **Imitation Learning** methods were the focus of the study. As mentioned before, the goal is to use both these techniques to teach bots how to play different game scenarios and compare the results obtained with both methods in different games.

To better understand how these learning techniques can be used to teach bots how to play games, first, it's good to understand how to create scenarios with learning bots. A general flow of a Game Engine and an ML library can produce self-learning bots can be seen in Figure 3.1. Creating the scenarios is quite simple for those with game development experience. The first step is to envision the desired interactive game, then, build that game using the Game Engine scene editor by placing different game objects and scripts that control the desired interactions.

After the game environment is built, each bot connects with the ML library interface, it's behavior takes into consideration the actions that it receives from the system and the observations from the environment it interacts with. Each bot sends a set of variables and values that constitute the observations, and a ray-cast from the bot to the world with hit history.

In the next sections, we are going to look in more detail at how both Reinforcement learning and Imitation learning algorithms can be used to teach bots how to play games.

3.2.1 Reinforcement Learning

As seen in Section 2.1.1, the aim of reinforcement simulation is that agents learn how to behave in a scenario by exploring it autonomously and gradually learning as it obtains rewards and starts understanding what is beneficial and what is not.

For this thesis, we used reinforcement learning algorithms to teach bots how to play games by having the bots play them autonomously with no initial knowledge of the game, and learn how to play it by playing the game out repeatedly, and getting better at the game by getting rewards and understating what got them the rewards.

One of the key elements for reinforcement learning is the definition of the reward system. The reward system is what is going to tell the bots that they did something positive or negative by awarding or removing reward points. In game scenarios, the main form of obtaining a reward should be the main goal of the game. This means that in a game of football we would reward a team by scoring a goal and penalize a team by suffering a goal. For these kinds of rewards, the two approaches that can be used are to give a flat reward for doing the objective or give a reward based on the time taken (more time to score/suffer a goal means lesser rewards given/taken).

Depending on the game, giving rewards only on the main objective might not be enough, in which case rewards can also be given for the secondary objective. These objectives are actions that don't win the game by themselves but contribute positively to it. If we reuse the football example, we can consider giving rewards for blocking a shot, stealing the ball from the opponent, or making a successful pass. Although these rewards can help the bots get better at the game, they should be significantly smaller than the rewards for the main goal of the game. If we give too much importance to secondary objectives, the bots might think that completing the main objective is useless as they could just repeat the secondary ones. Going back to the football example, two bots from the same team could just keep passing the ball between each other to get a higher reward than trying to score a goal.

Now that we talked about rewards we can have a better look at the learning process using reinforcement learning as seen in Figure 3.3. From the game, we can start the training process in which the bots and the reinforcement learning algorithm enter a cycle in which the bots communicate with the algorithms what observations they have and how much reward they have obtained, and the algorithm processes this information to learn and compute which actions the bot should do next. After the training ends we can analyze the results by both seeing how the rewards obtained progressed through the training and seeing the bots in play using the knowledge obtained in the previous training. If the results are not yet up to the pretended level, the training process can restart from the endpoint of the previous training.

Of the Reinforcement Learning algorithms, there are three of them that are commonly used. These algorithms are Proximal Policy Optimization (PPO) [35] and Soft-Actor Critic (SAC) [15] for single-agent scenarios, and MultiAgent Posthumous Credit Assignment (MA-POCA) [11] for multi-agent scenarios. PPO uses a neural network to approximate the ideal function that maps an agent's observations to the best action an agent can take in a given state. Meanwhile, SAC is an off-policy that can learn from experiences collected at any time during the past. As experiences are collected, they are placed in an experience

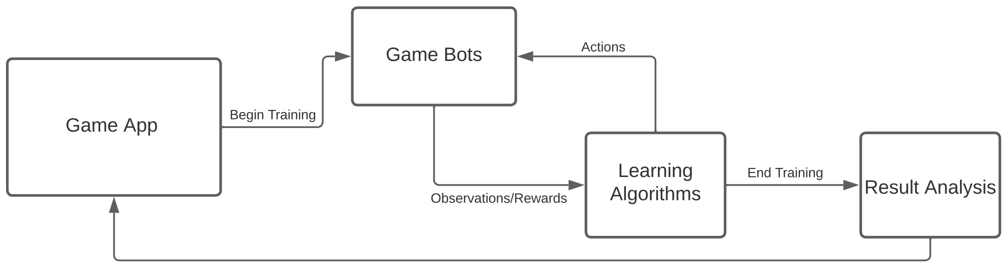


Figure 3.3: Reinforcement Learning Training flowchart

replay buffer and randomly drawn during training. Lastly, MA-POCA is a novel multi-agent trainer that trains a centralized critic, a neural network that acts as a "coach"for a whole group of agents.

3.2.2 Imitation Learning

As mentioned before, the goal of Imitation Learning is to get records of humans playing games and use the obtained data to train bots.

In the context of this thesis, these records will be called **demonstrations**. A demonstration is a data structure that saves the observations obtained using human gameplay, at each moment and maps it to the actions they took so that bots can decide their actions by finding the actions in the demonstrations that have similar observations to what they have at that point.

Focusing on the question of how can demonstration records help bots learn, let's first look at what exactly are these demonstration records. To obtain a demonstration, some sort of program or script attached to a bot controlled by a human is needed so that it can generate the demonstrations. This program will record the actions the user took at each moment and map it to the observations available and ray trace hits so it can be saved in a data structure to later be used.

This data structure forms a demonstration for imitation learning because the bots can then use the data in these data structures to decide what actions to take, by comparing their observations with the observations present on the demonstration file, and see which one looks the most similar and perform the action taken with that observation on the demo.

This means that to generate demos that can be used for learning, we can have human users playing the game scenarios and completing the tasks to serve as inspiration for the bots. This is possible because we can have humans control the bots by creating a typical game character controller and shifting the bots' actions to the human inputs rather than the library inputs.

The idea with this is that for more complex scenarios that take very long for bots to learn, and that the tuning of the configuration becomes harder, we can take advantage of these demonstrations to accelerate the process and create usable bots faster. For this, first, we have to create a game that is designed for both the agents and users to play out. We also need to define the observations and rewards the agent gets. If this game involves multiple agents simultaneously it has to support multiplayer so multiple users can also play it out simultaneously. After all, this is done, we proceed to have users play the game to record the demonstrations. The more users and the more time the better. Upon collecting all the demonstrations, the training of the bots can be started. Then the final result of the bots can be analyzed, and if it's still not up to expectations, the process can be repeated to collect more demonstrations with the possibility of using the current bots as adversaries to the users. A flowchart of this process can be seen in Figure 3.4.

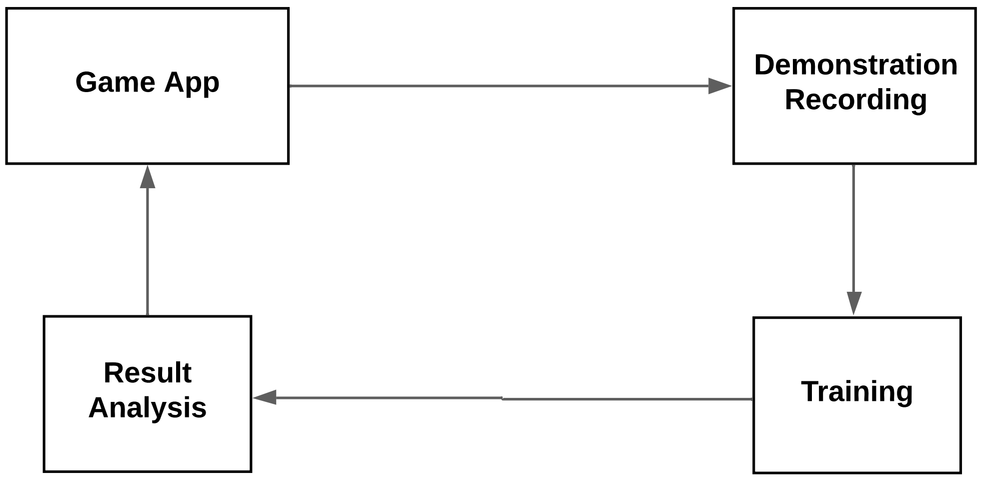


Figure 3.4: Demonstration Record process flowchart

For the imitation algorithms that mimic human behavior, there are two of them that are commonly used. These algorithms are the Generative Adversarial Imitation Learning (GAIL) and Behavioural Cloning. GAIL is an imitation algorithm that uses an adversarial approach to reward the agent for behaving similarly to a set of demonstrations. This training method does not require the definition of a reward system, since it distinguishes whether an observation/action is from a demonstration set or produced by the agent, and then examines a new observation or action and gives it a reward based on how close it believes this new observation/action is to the provided demonstrations.

Then we have behavioral cloning, which is a policy that trains the agent to mimic the exact actions of a demonstration. This policy can't generalize past what is in the recordings, which makes it work well when it exists recordings for nearly all states, which reinforces the idea that the more demonstrations the better. These imitation algorithms are not exclusive, meaning they can be used together. In particular, behavioral cloning works well when paired with GAIL since it allows the bot to infer what can be a good action if the observation values are not explicitly the same as one in the demonstration set, but similar to it.

3.3 Games

Now we are going to discuss which types of games we aim to teach the bots how to play. In an ideal world, we could use these kinds of algorithms to create usable bots for every game, but that is unrealistic since very complex bots would need a lot of computational power and time to the point where it is easier to create a bot with manually programmed behaviors. Creating a bot that follows the player and helps them in every available task in the game could be too heavy for self-learning bots, but teaching it how to drive a car around a circuit is a much lighter and doable task.

In this section, we are going to narrow down which type of games we consider that these teaching methods can be specifically useful, as the training process is reasonably resource and time hungry to obtain results that are satisfying and ready to ship to a full-fledged game.

Firstly, it is recommended games with a single objective. Single objective games will produce better results since the self-learning bots won't be divided by having to complete multiple objectives which can lead to it being less efficient at completing the tasks to even being unable to complete a single task since it got overloaded with information about multiple tasks that lead to being unable to understand what is good and what is not.

This, however, doesn't mean that using self-learning bots for games with multiple goals is unachievable, it simply is recommended to keep the number of goals as low as possible. Then we have the complexity of the tasks. These methods don't do well with complex tasks, meaning that the game should be something with a reasonable amount of actions, obstacles, and variables. The complexity of tasks that the method can handle varies with the number of tasks, meaning that a game with a single goal can handle a more complex task than a game with many goals in which the bot has to focus on learning various tasks.

Lastly, we have the number of bots in play. Games with a single bot or multiple bots with the same behavior and objective are simpler to teach. In these games, the algorithm only has to worry about a single type of behavior and all the findings through training can be used for every bot in play. Games with bots with different behaviors or objectives are harder to achieve since the algorithms have to focus on either teaching multiple behaviors, or how to play the same behavior in a symmetric objective.

To better understand these concepts, we are going to see some real game examples, and how they would fit into each of the mentioned categories. The first game we have is a game of basketball free throw shots. In this game, the single goal is to throw the ball from the free throw mark and hit it inside the hoop. This game should be one of the simpler ones to teach since it has a single goal with low complexity that is hitting the hoop from the same spot, and also only has a single agent.

The second example is a racing game, in which multiple bots drive a car around the same circuit with the objective of finishing first. Once more this game only has one goal of driving around the circuit, however, this one has multiple agents with the same behavior, and is a bit more complex than the previous one as it not only has to learn to drive the circuit, it also needs to learn how to overtake/defend against the other bots racing.

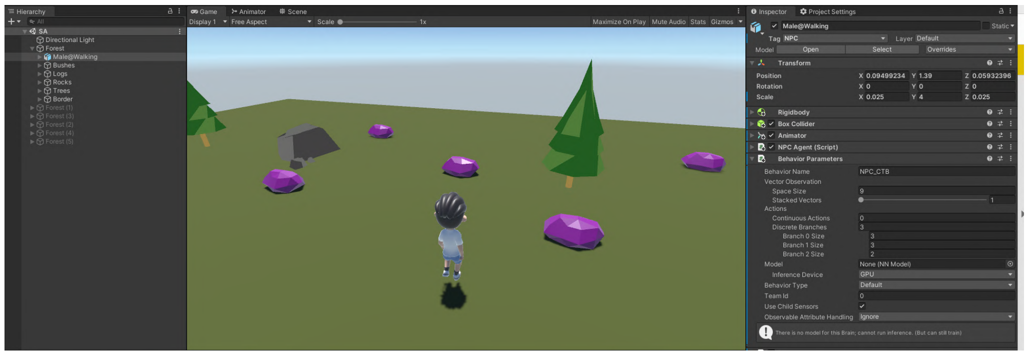
The last example is the game of football. This game has two goals scoring a goal and preventing the opposition's goal. This game is more complex than the previous ones since the bots have to learn all the rules of the game (no handballs, fouls, corners kick, etc...), and the multiple bots have the same behavior, but symmetric objectives depending on the team. If we want a game with different behaviors, we can split the football game into the attacking team and defending team, in which case the teams have different behaviors as they have different objectives.

3.4 Used Technologies

In this section, we are going to discuss with tools and technologies that were used to develop the game engine-based simulation loops to train self-learning bots in game scenarios.

For the Game Engine, Unity 3D was chosen to develop this project, as it offers an intuitive and accessible interface to develop games, and is free meaning that anyone can replicate the work done on their one and expand on it. Besides this, Unity 3D is accompanied by an asset store, where community members can out their assets for sale, or provide them for free, making it extremely useful to get assets and tools to create and develop games. Lastly, this game engine also offers a variety of plugins that can be integrated into games that offer a wide range of utilities to facilitate different aspects of game creation.

As for the Machine Learning library, the ML-Agents Unity Plugin was used. Being a plugin developed by Unity, compatibility and connecting the environments with the algorithms is not an issue. With this, the only focus in development is creating the game scenarios and testing out the different algorithms for teaching bots. Aside from this, ML-Agents offers a wide range of reinforcement learning, self-play, and imitation learning algorithms, and has available robust documentation that makes it easier to understand how to better use each algorithm and how to tune the algorithm parameters.



```
Male@Walking
Tag NPC                      Layer Default
Position    X 0.09499234   Y 1.39   Z 0.05932396
Rotation    X 0            Y 0      Z 0
Scale       X 0.025        Y 4      Z 0.025
Behavior Name                  NPC_CTB
Vector Observation
  Space Size                   9
  Stacked Vectors              1
Actions
  Continuous Actions           0
  Discrete Branches            3
    Branch 0 Size              3
    Branch 1 Size              3
    Branch 2 Size              2
Model                          None (NN Model)
  Inference Device             GPU
Behavior Type                  Default
Team Id                        0
Use Child Sensors              ✓
Observable Attribute Handling  Ignore
There is no model for this Brain; cannot run inference. (But can still train)
```

Figure 3.5: Used Technologies.

4 Games with Self-Learning Agents

In this chapter, we are going to see the details of how we created games with self-learning bots and how these bots were trained to play the games. Firstly we are going to talk about which games were chosen, how they play out, and why they were chosen, followed by the details of how the games were implemented to support the self-learning algorithms. Lastly, we are going to see how the training process was prepared and done going into the detail of the parameters used for each training.

4.1 Implemented Games

In this section, we are going to look at the rules and details of the chosen games. The games chosen were **Clean the Bush**, **Carry the Box** and **Capture the Flag**. Besides the rules of the games, in this section is also mentioned the controls of the game for a human player to play the game, ending with an explanation of why the game was chosen as a testing example. A summary of these games characteristics can be seen in Table 4.1.

4.1.1 Clean the Bush

In the game Clean the Bush, the player is in a natural park with bushes, rocks, trees, and tree trunks but the bushes are all polluted so they look purple instead of green, as can be seen in Figure 4.1. Faced with this unpleasant situation, the goal of the player in this game is to clean the bushes until they are no longer polluted and turn back green. To complete this game with a higher score, the player should clean the bushes in as little time as possible, while also avoiding colliding with the other objects.

To complete this task, the player can move forward and backward using the **W** and **S** keys, although the backward speed is slightly slower than the forward speed. Besides moving, the players can rotate left and right using the **A** and **D** keys, and start their cleaning action while pressing **F**.

In Figure 4.2 we can see a game state where the player has already cleaned some of the polluted bushes, signaled by them turning green. Meanwhile, the game is not yet

Table 4.1: Games Characteristics

	Clean the Bush	Carry the Box	Capture the Flag
Agent Type	Single Agent	Multi Agent	Multi Agent
Gameplay Type	Individual	Cooperative	Cooperative Competitive
Positive Rewards	Cleaning a Bush	Picking up a Box, Delivering a Box	Capturing a flag Recover a flag Pick up a ball Hit player with ball
Negative Rewards	Colliding with Obstacles	Colliding with Obstacles	Obstacle Collision Getting hit by ball Losing Ally Flag
Observations	Ray-Trace Vision Self Position Cleaning Action State Is Inside bush State	Ray-Trace Vision Self Position Delivery Zone Position Carrying Box State	Ray-Trace Vision Self Position Flags Position Ally Field Position Has a Flag State Has a Ball State

over as in the distance is possible to see that not all bushes have been cleansed as some of them are still purple.

This game was chosen as the first testing game, as it is a good example to test self-learning techniques in simple environments. This game is a game that has a singular way of gaining rewards by cleaning the bushes, and a single way of losing rewards by colliding with obstacles. This means that the bots only have to learn how to walk into the bushes, perform cleaning while in there and avoid the obstacles in the way. If the self-learning bots are unable to learn a scenario as simple as this, then these methods to create bots are useless as they won't be able to successfully teach the bots more complex games.

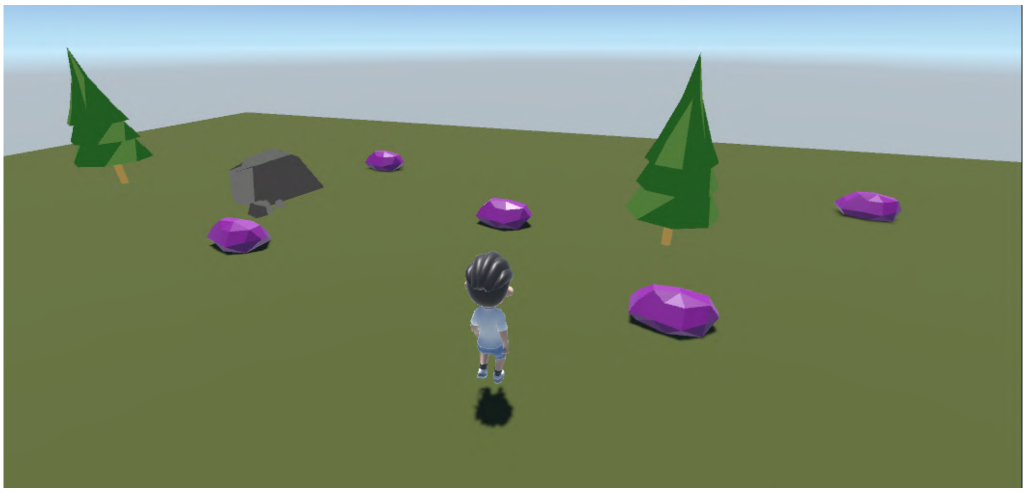


Figure 4.1: Screenshot from Clean the Bush game in the initial state with all bushes polluted.

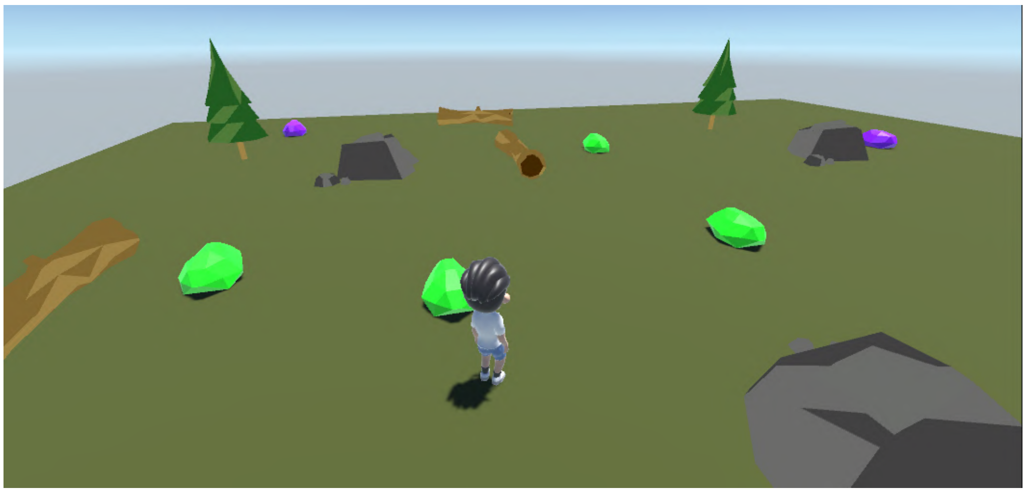


Figure 4.2: Screenshot from Clean the Bush game after the player cleaned some of the bushes.

4.1.2 Carry the Box

In the game Carry the Box, two players are in a warehouse site and have to transport the boxes from the initial zone to a carriage as seen in Figure 4.3. The trick here is that the zone has two floors: The upper one where the boxes are, and the lower one where the carriage is. So to complete the game the players start each on a different floor, and the one on the upper floor carries the boxes to a ramp that leads to the lower floor, and the player that plays on the lower floor waits for the boxes to drop and carries them to the carriage.

So to complete the game, the players have to work together to carry the boxes to the

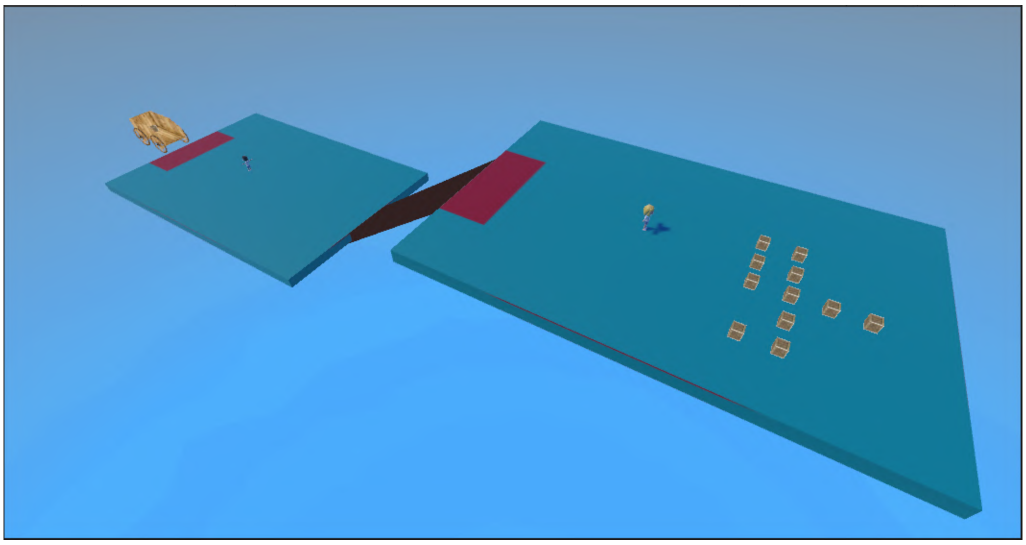


Figure 4.3: Screenshot from Carry the Box game in the initial state with all boxes in the upper level.

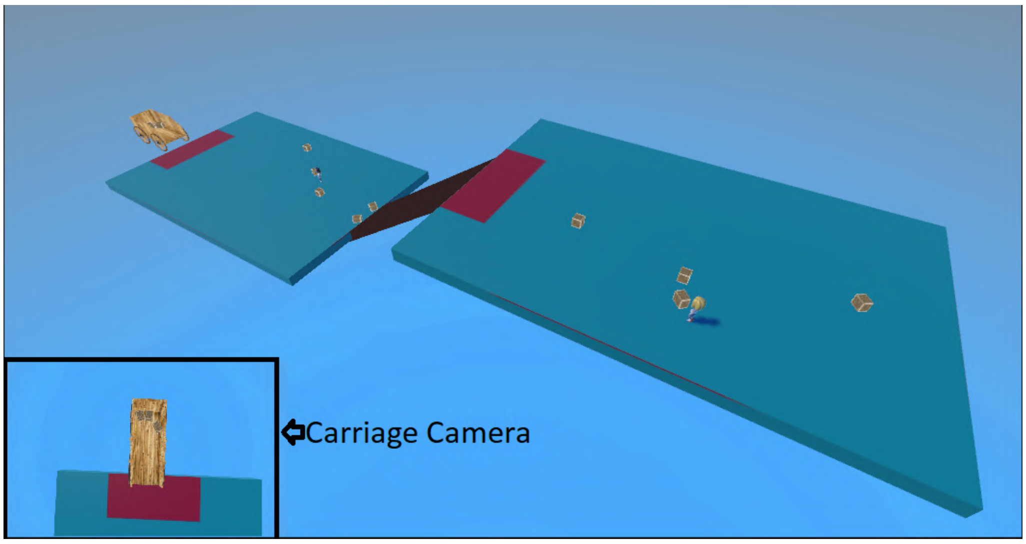


Figure 4.4: Screenshot from Carry the Box game after some of the boxes were moved.

carriage as quickly as they can. To do this, the players can move forward and backward using the **W** and **S** keys and rotate left and right using the **A** and **D** keys. To hold the box the players simply have to touch them and they will pick them up, being unable to pick up another box while they are still holding a box. To drop the box and the destined zone, they simply have to enter the red zone and the box will be dropped moving it to either the ramp or the carriage depending on whether the box is on the upper or lower floor.

In Figure 4.4 we can see a game state where the players have already moved some of the boxes, having some of them being held by the players, others at the carriage, some on the lower floor and ramp, and the rest still in the initial upper floor.

This game was chosen as the second test as it provided a good example of a simple scenario with some variations. As for the reward system, it is as simple as the previous one, the player receives rewards for carrying a box successfully to the destination, and loses rewards for colliding with the borders of the area. In the player department, this one seems more complex as it has two players cooperating to get the same results, but with a closer look, we can see that it is not the case. Although there are two players that play different roles, their tasks as exactly the same. Each player needs to find a box, pick up the box and carry it to the red area. This means that in reality, the bots of this game can learn from the same behavior with no need to distinguish teams. A well-trained brain for a bot in the upper level should be a well-trained brain for a bot in the lower level as well.

4.1.3 Capture the Flag

The last tested game was the game Capture the Flag. In this game, two teams of three players compete to steal the opposing team's flag and return it to their teams' side. The field is made of two symmetric halves with inner walls to provide hiding spots for the

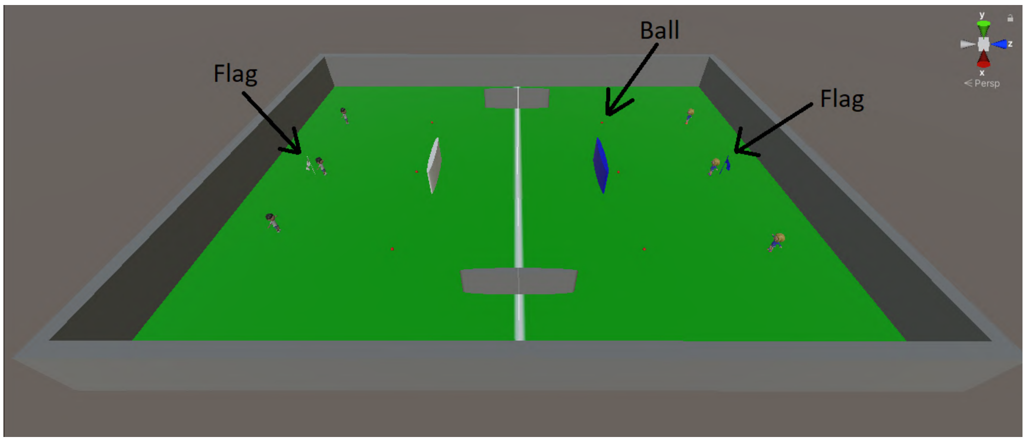


Figure 4.5: Screenshot from Capture the Flag game.

players. Besides the walls, it is also placed in the field six balls (initially three in each field) that the players can pick up and throw to stun the enemies and make them drop the flag in case they're holding it. If a flag is dropped outside its spawn area it can be picked by the rival team to resume their heist on the flag, or retaken by the ally team in which case the flag automatically transports to the spawn point. In Figure 4.5 it is seen the capture the flag field as described before.

To win this game, players have to compete with the opposing team and cooperate with the ally team to steal the opposition flag first, while defending the team's flag. To do this, the players can move forward and backward using the **W** and **S** keys and rotate left and right using the **A** and **D** keys. Besides the controls already present in the other games, the players can strafe left and right using the **Left Arrow** and the **Right Arrow** keys. To pick up the balls the players just have to walk through it, as it will automatically pick them up, and can throw them in the direction they are facing by pressing **Space-bar**.

In Figure 4.6 we can see the point of view of a player playing the game, in which he

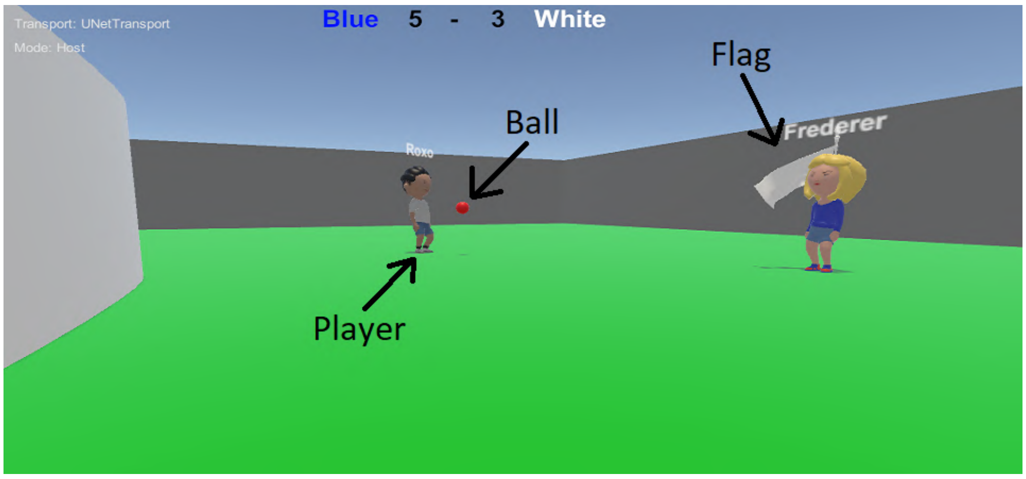


Figure 4.6: Point of view image of a player in Capture the Flag.

can see two other players, one from the white team holding a ball, and one from the blue team stealing the white team flag. Since in the game multiple episodes are played, it is also possible to see the scoreboard in which the blue team is winning 5 to 3 against the white team.

This last game was chosen to provide results in a more complex game scenario. In this game no only we have a total of six players, but the behaviour of the bots needs to be split into two teams so each bot knows how to play in cooperation with their team but also in competition with the opposing team. Besides this, this game also has a more complex reward system. Obviously the main source of gaining and losing rewards is the game result, if your team wins you get rewards if your team loses you lose rewards. But since the game have many secondary interactions some of these also need to be considered for rewards.

First of all, it makes sense to reward players for capturing/recovering a flag since it is the most important task to win the game. Hitting players with a ball can also be considered for rewards. If you hit a member of the opposing team it makes sense to get rewards since you stun an enemy for a couple of seconds and may cause him to drop the flag. Meanwhile hitting an ally player you should lose rewards since you are setting your team back by incapacitating a member for a couple of seconds. With this game we'll be able to see how the self-learning bots fair off in a more complex game with more variants, seeing how they do in both the cooperative and in the competitive parts of the game.

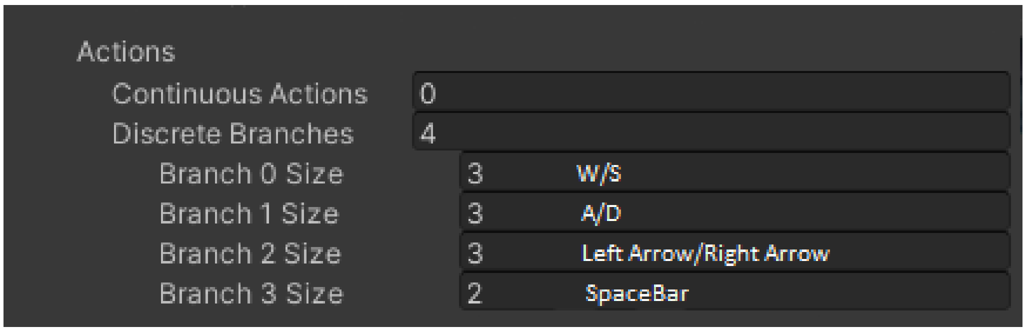
4.2 Game Implementation

In this section, we are going to see how the games were implemented to support the different needed components to create the self-learning bots. First, we are going to review what was needed to do in each game to teach bots how to play using reinforcement learning. Then we are going to talk about the adaptations that are needed to create bots that learn through imitation and how to create game scenarios where human players can play and record demonstrations that bots will use to learn the game. Lastly, as a follow-up on imitation learning we are going to see how to implement a multiplayer game that allows multiple human users to play and record the demonstrations simultaneously for games that require multiple bots to interact with each other.

For the context of this thesis, we consider an **Episode** to be each full iteration of a game played. This means than an episode is the entirety of events that happen from the moment a game round starts to be played to the moment that same round ends either by the objective being completed or by exceeding the time limit.

4.2.1 Reinforcement-Learning

As seen in Section 3.2.1, Reinforcement Learning bots learn by exploring the game autonomously by doing random actions and slowly learning how to perform their actions



Actions
Continuous Actions 0
Discrete Branches 4
Branch 0 Size 3 W/S
Branch 1 Size 3 A/D
Branch 2 Size 3 Left Arrow/Right Arrow
Branch 3 Size 2 SpaceBar

Figure 4.7: Action specification in the Unity3D inspector

as they receive/lose rewards as the algorithm analyses the observations the bot has when a reward is given, and as rewards accumulate the algorithm starts to understand patterns of actions leading to observations and observations leading to rewards.

Since the main components needed to teach bots using Reinforcement Leaning are **Actions**, **Rewards** and **Observations**, in this section we are going to talk about what is needed to implement these components using a generic game engine and machine learning library followed by a specification of how Unity 3D and the ML-Agents plugin allowed the implementation of the different components.

The **actions** for the bots are equivalent to what the input controls are for humans. The actions are what will tell the bot that it should move forwards or backward, turn left or right among all the other interactions the player needs to do. As mentioned before, human users need to use input devices like a keyboard to perform their actions, like using the **WASD** keys to move as is the case in all the games implemented. But the bots cannot press keys on a keyboard, so they need another method to process their actions. This method is through numeric values. With numbers, we can map the values given by the algorithms to actions similar to keyboard presses.

Reinforcement learning and most agent-based machine learning algorithms can interpret actions in either continuous or discrete values. In continuous values, the range usually goes from -1 to 1 as normalized values help algorithm efficiency. Meanwhile, the range in discrete values needs to be specified as they go from zero to the desired integer. In our games, since the actions from the human perspective are all keyboard inputs, it makes sense to use discrete values. Since you can't half press a key on the keyboard using a continuous value will only increase learning difficulty, but if the motion control was done through a joystick analog or other device that is sensitive to how much it is pressed then a continuous is the better choice.

In all of the implemented games, at least two discrete values were needed to describe the movement. The first one to describe motion (**W** and **S** keys), raging from 0 to 2, where 0 represents the state where no key is pressed, and each one of the remaining states for each of the **W** and **S** keys. For rotation the same logic is applied, where we have a three option discrete value, where 0 describes no pressed key, and 1 and 2 represent **A** and

Table 4.2: Mapping of Inputs to Discrete Values

	0	1	2
Movement	No movement	Forwards	Backwards
Rotation	No Rotation	Left	Right
Actions	No Action	Clean	

D. In the Clean the Bush game, besides the movement the **F** can be pressed to clean a bush, this was represented by another discrete value with 0 for no key pressed and 1 for **F** key pressed. This same logic was applied for the extra actions in capture the flag, the strafe movement can be represented by a three option discrete value and the throw ball command by a two value discrete value as seen in Figure 4.7 and in Table 4.2.

To do this using Unity3D game engine and the ML-Agents plugin, we can use a script from the ML-Agents which is the Agent script. This script must be attached to every bot that needs to learn, and establishes the connection between the algorithm and the game engine. In it, there is a function called **OnActionReceived**, on which the algorithm sends the numeric values requested in the game engine as shown in Figure 4.7, and then the developer has to map the values to the actions as one would map the keyboard presses.

The **rewards**, as mentioned before, is what triggers the algorithm to try to map actions and observations to what is good or not. This makes the definition of a reward system one of the key elements in reinforcement learning which without the bots are unable to learn effectively. To see how game engines can be used to implement efficient and easy-to-make reward systems we are first going to see what rewards are given in each game followed by an explanation of which game engine tools can be used to detect reward opportunities and give them to the bots.

In **Clean the Bush** the reward system is quite simple as the only influencing factors are if the bot is cleaning the bush and if it collided against an obstacle. So in this game the bot receives a little bit of reward for each moment he is cleaning the bush which

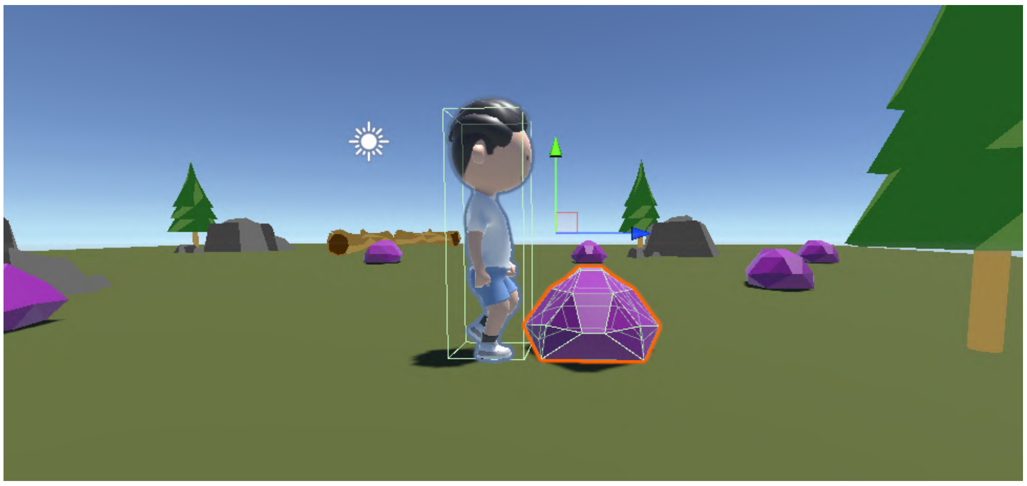


Figure 4.8: Colliders of the bot and the bush in Clean the Bush game.

accumulates to a substantial reward as more bushes get cleaned, and the bot loses some rewards each time it collides with an obstacle. This way the bot gets incentives to find and clean bushes, but also to avoid the obstacles in the way. In **Carry the Box**, the system is really similar as the bot receives rewards for picking up boxes and for dropping them at the objective area, and loses rewards for colliding with borders and obstacles.

In the **Capture the Flag** game, the reward system is more complex as more variants need to be looked at. First of all, since this game is a multiple bots game with different teams in the behavior, rewards are given as a team, This means that the rewards one bot gets, all the bots from the same team will also get. Successfully taking the opposition flag you allied spawn zone is what prompts the game over screen with a victor and defeat, so completing this task awards a large number of rewards to the winning team, and penalizes the losing team the same large amount of rewards. To retrieve the enemy flag to your spawn area, first, you need to get a hold of the enemy flag, so catching the opposing team flag also wins rewards. Likewise, recovering the allied flag after it being stolen gets the bots some rewards. Then we have the ball interactions. Hitting an enemy with a ball gives rewards while hitting an ally loses rewards. Getting hit by a ball will also result in rewards being taken. Lastly, running into walls will result in rewards lost, since we don't want our bots to bump into them constantly.

So now that we saw how rewards were given in different games we can talk about how game engines allow us to detect the mentioned events and give rewards for them. The main tools in a game to detect such events are collision detection and variable. For instance, in the Clean the Bush game where only two events cause rewards change, we can use collision detection to know when the bot bumps into obstacles to trigger the reward loss. To know if a bot is cleaning, we can use collision detection to know when a bot is inside a bush, and then check if the bot's variable that represents if it is cleaning or not, and the variable that represents if the current bush is polluted or not, giving it rewards if

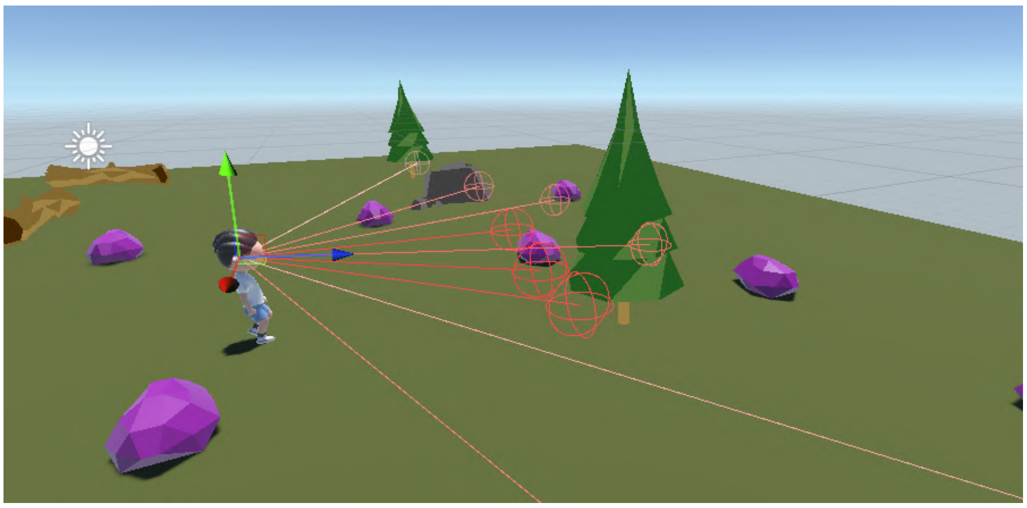


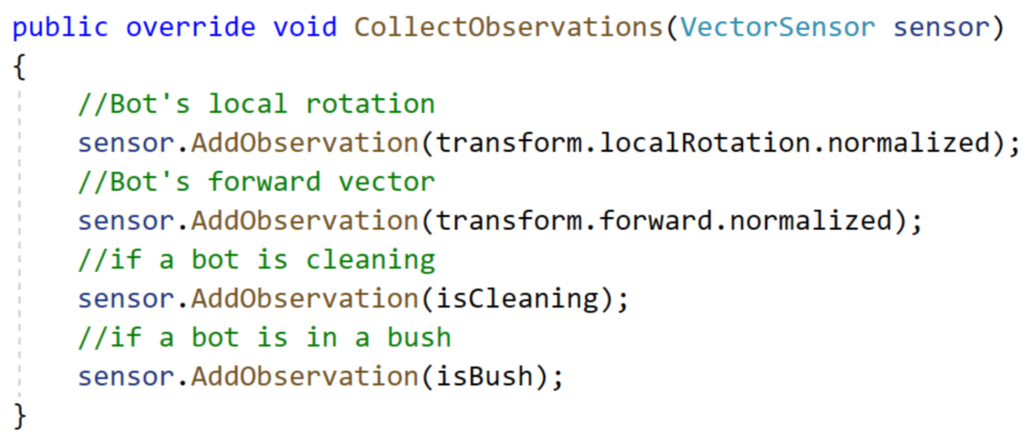
Figure 4.9: Colliders of the bot and the bush in Clean the Bush game.

it's performing the cleaning action while inside a polluted bush. This collision detection is facilitated when using a game engine since they have pre-built collider systems that automatically detect contact with other colliders. In Figure 4.8 we can see the Unity3D collider system that is used to detect collisions.

The last key component, the **observations**, is a representation of what the bot sees and knows through numeric values. This numeric representation of what is seen is what allows for the algorithms to discover similarities in these numbers each time a reward is given, and discover what to look for and what to avoid. These observations will most of the time be ray trace hits to simulate vision, as well as the representation of the bot's world coordinates, the local rotation, and the vector that represents the forward direction the bot is facing. In the Clean the Bush game, besides the mentioned observations, the bot also knows if he is performing the cleaning action or not. Similarly, in the Carry the Box game the bot knows if he is carrying a box, as well as the world coordinates of the destiny location.

In capture the flag, the bot has a larger number of observations as more things are in play. Besides the common observations mentioned, the bot knows if it holds a ball, if it holds a flag, the world coordinates of the allied flag and of the enemy flag, and a vector pointing toward the enemy flag (to facilitate the bot understanding the objective), and a vector pointing towards home spawn so the bot knows where to take the flag.

To represent these observations, both Unity3D and ML-Agents have some tools that facilitate the development process. The ML-Agents contains a script that provides ray-tracing vision, as seen in Figure 4.9, which automatically supplies to the algorithm the numeric representation of the ray trace hits. In the Agent script mentioned before, it is also available the function **CollectObservations** that allows us to supply the rest of the observations using typical Unity scripting tools. Everything that represents a world coordinate or a vector can be represented using a Vector3 variable, and every flag-typed variable can be passed as observation through its boolean value as seen in the code snippet in Figure 4.10.



```
public override void CollectObservations(VectorSensor sensor)
{
    //Bot's local rotation
    sensor.AddObservation(transform.localRotation.normalized);
    //Bot's forward vector
    sensor.AddObservation(transform.forward.normalized);
    //if a bot is cleaning
    sensor.AddObservation(isCleaning);
    //if a bot is in a bush
    sensor.AddObservation(isBush);
}
```

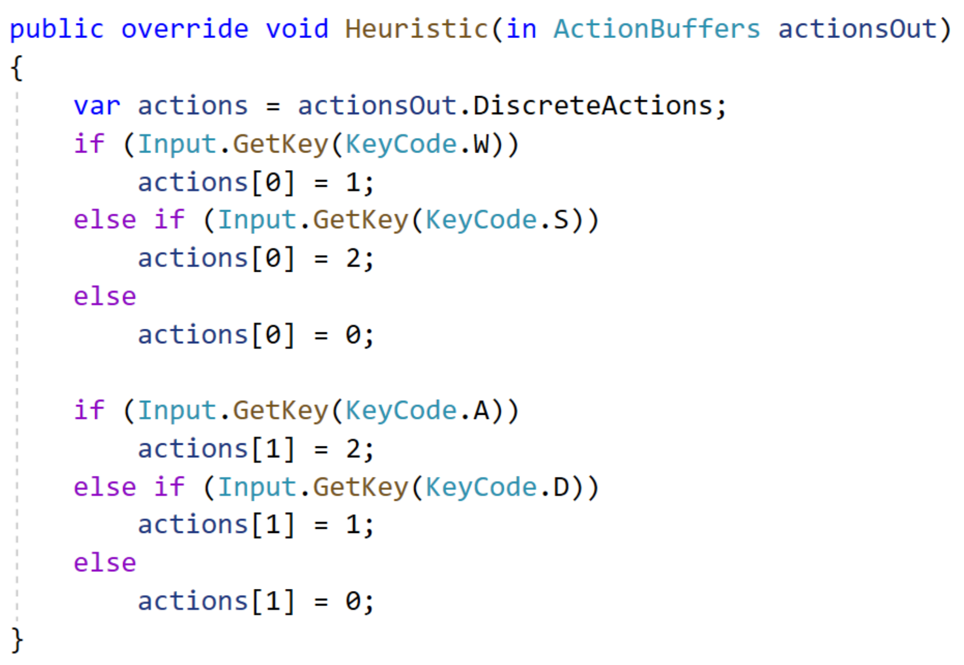
Figure 4.10: Code snippet of the CollectObservations function in Clean the Bush game.

4.2.2 Imitation

As seen before in section 3.2.2, imitation learning uses a sample of demonstrations to teach the bots how to play by mimicking the behavior in the demonstrations. Although these demonstrations can be human-based on computer-based, in this thesis the focus was on teaching bots through human behavior, so only humane imitation techniques were used. In this section, we are going to see what was needed to obtain these demonstrations based on human behavior and how to use them to teach bots.

In this paradigm, two of the main components mentioned for Reinforcement Learning still stand: the **Actions** and the **Observations**. In Imitation Learning, rewards are not necessary as the algorithms have automated reward systems based on similarity to demonstrations, but they can still be optionally used since Imitation Learning can be used simultaneously with Reinforcement Learning. As for the other components, they work the same as in Reinforcement Learning. The actions need to have a numeric representation so the algorithm can tell the bot what to do, and observations are a numeric representation of what the bot sees and knows.

Now the question is, how to record a human-player game session, and save it as a demonstration file that can be interpreted by the algorithms. First, let us remind that a demonstration is a data structure that maps the actions the player takes to the observations it has in that instance. This means that to form a representation we need to convert the player input (such as pressing WASD keys to move) to the numeric representation the bots have for those actions. As seen in the previous section, the numeric representation for actions is not hard to obtain, since they are either a continuous number between -1



```
public override void Heuristic(in ActionBuffers actionsOut)
{
    var actions = actionsOut.DiscreteActions;
    if (Input.GetKey(KeyCode.W))
        actions[0] = 1;
    else if (Input.GetKey(KeyCode.S))
        actions[0] = 2;
    else
        actions[0] = 0;

    if (Input.GetKey(KeyCode.A))
        actions[1] = 2;
    else if (Input.GetKey(KeyCode.D))
        actions[1] = 1;
    else
        actions[1] = 0;
}
```

Figure 4.11: Code snippet from Heuristic function to represent WASD movement as discrete values.

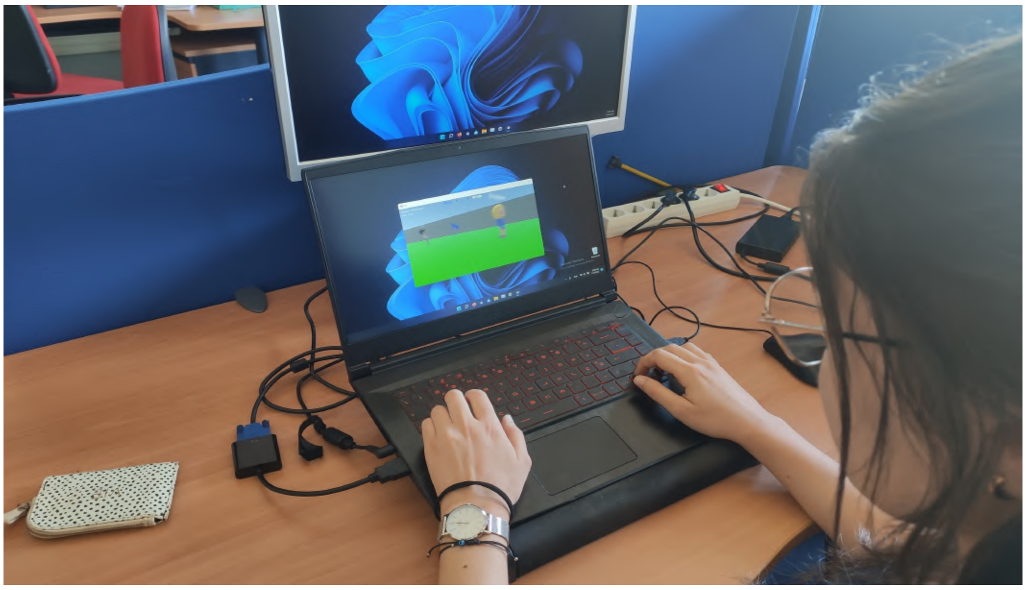


Figure 4.12: User playing the Capture the Flag game to record a demonstration.

and 1 or a discrete integer number between zero and the desired value. We even exemplified the numeric representation of every action in every game through discrete values in Section 4.2.1, such as movement being a value from 0 to 2 to represent no movement, forwards, and backward.

In the Agent script from the ML-Agents plugin mentioned in the previous section, there is a **Heuristic** function that eases the conversion from human input to numeric representation. Instead of having a dedicated script for the player movement, we can use this function to describe the inputs as numeric values as seen in Figure 4.11, and then the **OnActionReceived** mentioned before will take the numeric representation and perform the movement/actions as if it was provided by the algorithm.

We have now seen how to convert human inputs to numeric values that the algorithm and the bot understand, now we just need to have a system that saves the mapping between the user input and the observations. One way would be to map at every other frame of the game a list of actions taken and map it to a list of observations obtained. In the ML-Agents plugin, there is a script that automatically does this. Once the script is attached to a player with the Agent script, it will record the play session and save a file with all the mappings of actions to observations.

With all this done, it is simply needed to have the game ready and let human users play it to record demonstrations as seen in Figure 4.12. It is possible to train bots using multiple demonstrations at once, and the more demonstrations are provided the better the final result should be as there is a bigger sample of mapping for the algorithms to find similarities to the bots' current state. Worthy to note that only demonstrations of players who know how to play the game are useful, as the algorithm has no way to know if a player is playing badly, meaning that providing a demonstration of a bad player will

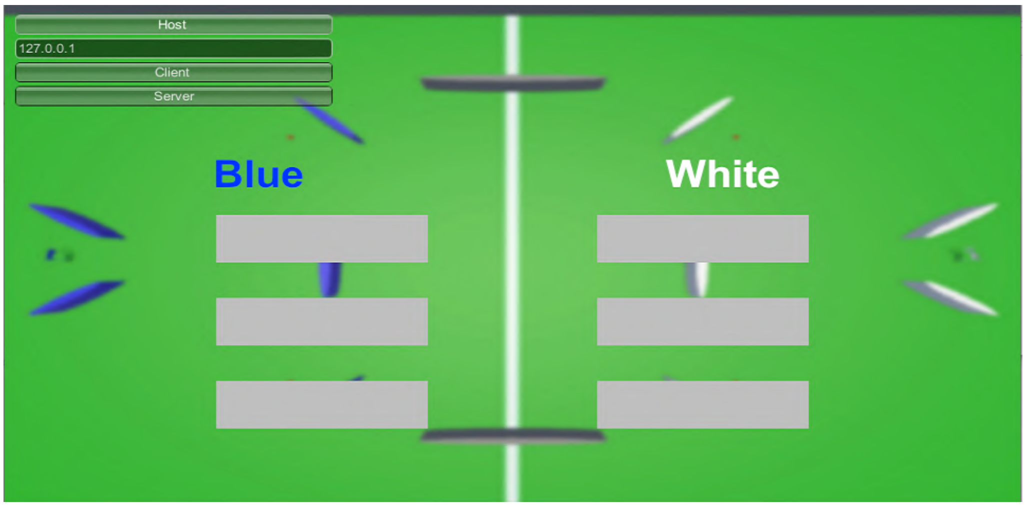


Figure 4.13: Capture the Flag Connection Screen

create the chance the algorithm takes those mapping as an example for the bot to follow.

4.2.3 Multiplayer

Now all that is left to be able to train our bots is to look at the Capture the Flag game. With this game being, at its nature, a $3vs3$ game, it supports multiple users playing at the same time, meaning that a multiplayer component must be implemented for this game. This is needed so that the human users can play and record demonstrations at a competitive level by playing other humans so that the comparison between demonstration observations, and the bot's observations is as realistic as possible so the learning is more effective.

The first step in designing the multiplayer is defining how the connection is made. For this, we used the Unity Netcode plugin which allows us to connect to a host player simply by putting the host's public IP address. To establish this connection we created a menu, as exemplified in Figure 4.13, which has the option to start a game as either a host or a server, or to connect as a client with a space to input the host IP to establish the connection. After establishing the connection, there is the option to input a game name for better immersion, the option to choose which team the player will compete on, and the list of connected players divided by the selected team.

Then we have to look at the network object. Network objects are those that need to be constantly synchronized between all connected players so that the game state is consistent. The objects usually are those whose state (position, rotation, variable, etc...) are dynamic and in constant change. All other objects who are static and see no relevant change during gameplay don't need to be synchronized since their state should remain the same through all connected players regardless. If we take a game of football as an example, the ball would be a network object since it's in constant motion and is essential that all players see it the same while the goal posts wouldn't be a network object, since

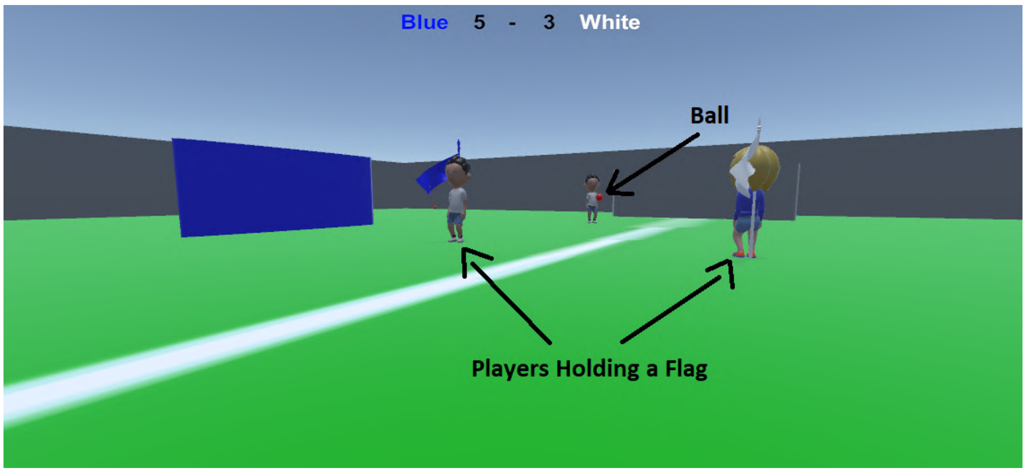


Figure 4.14: Screenshot from the Capture the Flag game, where it is shown the three network object types: the players, the flags and the balls.

their position is always the same no matter what happens in the game.

For the capture the flag game, we have three groups of objects that will be network objects and thus synchronized: Players, balls, and flags.

The flag and the ball are the simpler network objects. The flag mostly coordinates its position, having as only variables a boolean that states if the flag is picked up by a player or not. The Ball is pretty similar since it mostly coordinates positions, and a boolean if it is picked up, but also has a variable about the user who last picked up the ball.

The players, like all others, also have to coordinate their positions. But besides this, they also have information about which ball and/or flag they are holding, the inputs pressed by the user so it can be supplied to the ml-agents interface, and if it is stunned so it can't move after being hit. In Figure 4.14 we see a screenshot that displays all network objects in action: the two players, the flag, and the ball both being held by players.

Having defined how to establish a connection and which objects need to be synchronized, we can use the Unity Netcode plugin to create the online connection and handle the synchronization. With this plugin, we can create a Manager Script that reads inputs

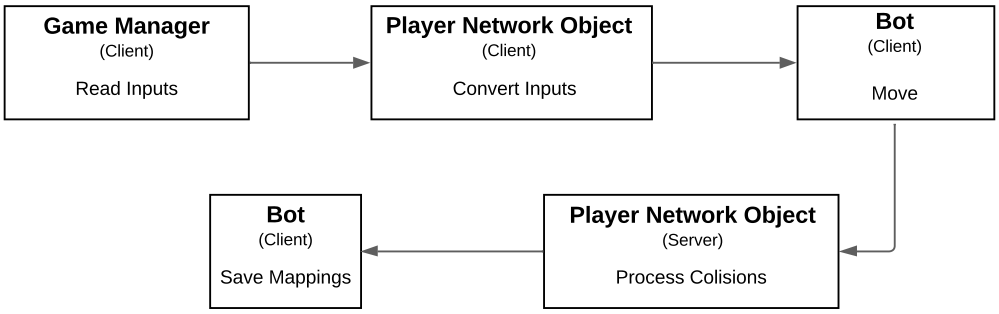


Figure 4.15: Multiplayer Execution Flow

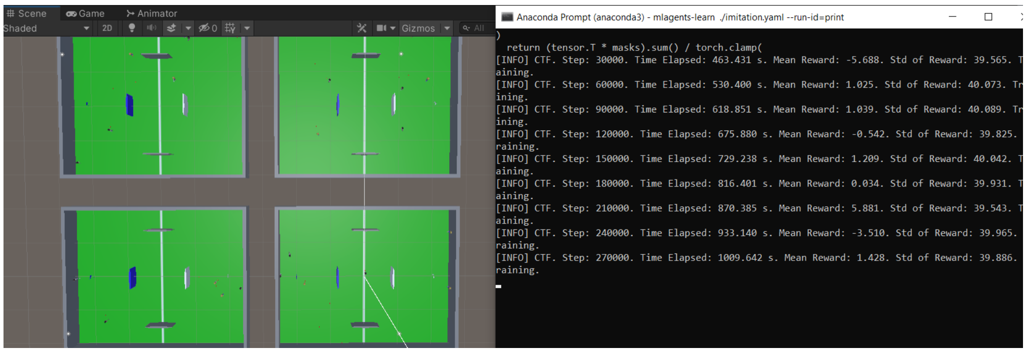
and communicates with the server to send and receive new game states. The server is responsible for all collision calculations so that these are consistent amongst all players. This means that upon receiving information about a state change in a player, the server will calculate all collisions and send them back to all players so everyone has a consistent game state.

Now we can take a better look at how the game will flow with the multiplayer as shown in Figure 4.15. Upon the start of the game players will starts pressing inputs that are read by the Game Manager to move their characters. These inputs are then sent to the player network object on the client side and converted into discrete integer values that can be read by the agent. The agent script will read the values and perform the movements and actions upon command. After moving, on the server side, the collisions are calculated and processed causing the player to be stunned, catch the balls and flags, and upon receiving the result of the collision detection the player-controlled bot can save the mappings of actions/observations to save in the demonstration.

4.3 Training

After seeing how we prepared games for training bots with Reinforcement and Imitation Learning algorithms it's time to proceed to the training phase of the bots. In this thesis, we will focus more on training through Reinforcement Learning using the PPO and MA-POCA algorithms, and Imitation Learning using Behavioural Cloning and GAIL algorithms.

During training, it is generated for each bot a log file with different relevant metrics for evaluation. The contents of these logs will be further explained in Section 5.1 where the metrics used to evaluate the training results will be exhibited and discussed. To initiate training, the configurations of the algorithms and a form of connection to the game engine environment are needed. Using Unity3D and ML-Agents plugin this can be achieved using the python-based ML-Agents library that reads a configuration file that dictates the used algorithm and used parameters, and automatically detects a unity environment to connect with and start training.



```
Anaconda Prompt (anaconda3) - mlagents-learn  ./imitation.yaml --run-id=print
)
  return (tensor.T * masks).sum() / torch.clamp(
[INFO] CTF. Step: 30000. Time Elapsed: 463.431 s. Mean Reward: -5.688. Std of Reward: 39.565. T
aining.
[INFO] CTF. Step: 60000. Time Elapsed: 530.400 s. Mean Reward: 1.025. Std of Reward: 40.073. Tr
ining.
[INFO] CTF. Step: 90000. Time Elapsed: 618.851 s. Mean Reward: 1.039. Std of Reward: 40.089. Tr
ining.
[INFO] CTF. Step: 120000. Time Elapsed: 675.880 s. Mean Reward: -0.542. Std of Reward: 39.825.
raining.
[INFO] CTF. Step: 150000. Time Elapsed: 729.238 s. Mean Reward: 1.209. Std of Reward: 40.042. T
aining.
[INFO] CTF. Step: 180000. Time Elapsed: 816.401 s. Mean Reward: 0.034. Std of Reward: 39.931. T
aining.
[INFO] CTF. Step: 210000. Time Elapsed: 870.385 s. Mean Reward: 5.881. Std of Reward: 39.543. T
aining.
[INFO] CTF. Step: 240000. Time Elapsed: 933.140 s. Mean Reward: -3.510. Std of Reward: 39.965.
raining.
[INFO] CTF. Step: 270000. Time Elapsed: 1009.642 s. Mean Reward: 1.428. Std of Reward: 39.886.
raining.
```

Figure 4.16: Unity training environment and ML-Agents console

Table 4.3: Configuration Parameters for RL Training

trainer_type	PPO	MA-POCA
batch_size	1024	2024
buffer size	10240	20240
learning_rate	3e-4	3e-4
epsilon	0.2	0.2
lambd	0.95	0.95

In Figure 4.16 we show an example of the Unity training environment as well as the ml-agents console in the middle of a training session. To ease the evaluation of the results for the imitation algorithms, during demonstration recordings the human-controlled bots generate the same log files as the AI bots. The only difference is during training some default values may generate for bots since these have a time limit on each episode (Each iteration of the game played) which may lead to unfinished scenarios compared to demonstrations where humans always finish their tasks.

Now that we spoke about how to start training and that during the process logs are generated to help analyze the results, it is left to speak about the training parameters. These parameters are different for Reinforcement Learning and Imitation Learning, and although there are many parameters that can be tweaked during training, only the most relevant ones are going to be mentioned.

For Reinforcement Learning the parameters[1] that were the most relevant during the training were: Batch Size, Buffer Size, Learning Rate, Epsilon, and Lambd. The batch size is the number of experiences/examples produced by the algorithm over each iteration of training. Related to batch size, buffer size is the number of these experiences/examples the algorithm should collect before updating the policy, this is, how the bot behaves. This parameter should be a multiple of the buffer size, so we don't have examples sampled from an iteration split between different buffers. The learning rate is the initial rate for gradient descent, as it corresponds to the strength of each gradient descent update step. In other words, the learning rate is how strongly the buffer experiences affect each policy update, as it shouldn't be too high as it hinders learning early during training, nor it should be too low as it prevents the bot's ability to learn. The epsilon influences how rapidly the policy can evolve during training. It corresponds to the acceptable threshold of divergence between the old and new policies during gradient descent updating meaning that setting this value small will result in more stable updates, but will also slow the training process. Finally, the lambd can be thought of as how much the bot relies on its current value estimate when calculating an updated value estimate. Low values correspond to relying more on the current value estimate (which can be high bias), and high values correspond to relying more on the actual rewards received in the environment (which can be high variance). The values used for each of the policies of RL (PPO for Clean the Bush and

[1]ML-Agents Parameters, https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md , Last Access: September 2022

Carry the Box, and MA-POCA for Capture the flag) can be seen in Table 4.3.

For imitation learning, mostly the default values were used, with the most relevant being the strength of imitation being set to 1, which maximizes the learning through imitation. Since imitation can be used in junction with reinforcement learning there are setting where having an imitation learning strength lesser than one makes sense to balance what the bot learns with imitation, and what he learns with RL, but since the focus of the thesis is with pure imitation, we leave the strength set to one.

5 Evaluation

To evaluate the results of the different training, several metrics were defined to let us compare the performance of the bots while learning with reinforcement learning and with Imitation Learning. In this chapter, first, we are going to discuss which metrics were used in each game and how they help us evaluate the performance of the bots. Then, we will expose the various results of the different training sessions and discuss what it shows us about the bot's progression and performance.

5.1 Metrics

In this section we are going to see which metrics were chosen to evaluate the bots level of play in the different implemented games, and why these metrics were chosen. These metrics must be values that in some capacity measure the quality of play from the bots, such as measuring times for speed analysis, or track certain variables that influence the result of the game.

5.1.1 Clean the Bush

To analyze the results of the entire bot training process using Reinforcement Learning, Imitation learning and the resulting usable bots, we first defined metrics to evaluate the training process in reinforcement learning displayed in Table 5.1 , followed by metrics to evaluate the resulting bots and the human recording sessions, finishing by metrics to evaluate bots though with imitation learning.

Table 5.1: Metrics for Clean the Bush game

	Metric
A1	Episode Times
A2	Time in Bushes
A3	Pollution Cleaned
A4	Time Cleaning

The first metric defined, A1, was the episode times (each episode represents an episode of the bot cleaning the bushes) through the course of training. These times include the total episode time, the time the bot took to clean 10 out of the 12 available bushes and the time the bot needed to clean half of the bushes. Assuming the bots can learn to beat the objective within the time limit, this allows us to see the progression of the bot getting faster at beating the game and analyze if there was a threshold where the bot couldn't get faster. Having the times at certain checkpoints also allows us to comprehend if at certain point the bot was able to clean bushes when many were available and only struggled to find the last few ones or if there simply was a general struggle in cleaning in the first place.

The next metric, A2, was the time spent in bushes by the bot though the training process. This metric allows us to analyze the progress of the bot learning that the bush is where rewards can be obtained, and in conjunction with the next metric A3, the amount of pollution cleaned, also permits to identify if there was a point where the bot was in bushes but wasn't cleaning, and how efficient the bot was in cleaning the bushes once it was there. The amount of pollution cleaned, is a value created to quantify how much the bot has cleaned. Each unit of this value is equivalent to 2 seconds spent cleaning a bush, which is why in conjunction with time spent in bushes it allows us to infer the bot efficiency once inside a bush since the required time to clean a bush is equal to 2 times the amount cleaned.

The last training related metric, A4, was the time spent performing the clean action. This metric aims to see if the bot is defaulting to cleaning all the time instead of cleaning only when inside a bush. Time spent cleaning can be compared to time spent in a bush to see how much time the bot was performing the cleaning action compared to the time it should have spent performing that action.

Then we analyze average times and values of the previous mentioned metrics, as well as its deviations from the mean and compare it to results obtained by human users during demonstration recordings. This values aim to see if the level of the final bot is on par with the level during final episodes of training as well as compare it to the human level and see if it performs better or worse. To display the imitation learning results, these metrics will be displayed but as a progression graph as more demonstrations were used to train the bots.

5.1.2 Carry the Box

To analyze the results of the entire bot training process using Reinforcement Learning, Imitation learning and the resulting usable bots, we first defined metrics to evaluate the training process in reinforcement learning shown in Table 5.2, followed by metrics to evaluate the resulting bots and the human recording sessions, finishing by metrics to evaluate bots though with imitation learning.

Metric B1 is the episode times (each episode represents an episode of the bot carrying

Table 5.2: Metrics for Carry the Box game

	Metric
B1	Episode Times
B2	Boxes Carried
B3	Time Carrying Boxes
B4	Rewards

the bots) through the course of training. Assuming the bots can learn to beat the objective within the time limit, this allows us to see the progression of the bot getting faster at beating the game and analyze if there was a threshold where the bot couldn't get faster.

The second metric defined, B2, is the number of boxes a bot successfully carried in each episode. This allows us to observe how long the bot took to learn to complete its objective, and compare it to the previous values to access how efficient the bots are. Metric B3 is the time the bots spent carrying the boxes during the course of training. This allows us to analyse the progress of the bot learning that it has to carry the box, and then also see the progress if the bot gets faster at getting the boxes to the objective zone.

The last metric, B4, is he rewards obtained by the bot over time. This allows us to have a general view of how was the progression of the bot not only in learning the objective, but also in avoiding penalties.

To evaluate the level of the bot after the training, we analyze average times and values of the previous mentioned metrics, as well as its deviations from the mean compared to same values obtained by human players. This allows us to see if the level of the final bot is on par with the level during final episodes of training, as well as compare the level to the human level. To display the imitation learning results, the same metrics will be displayed but as a progression graph as more demonstrations were used to train the bots.

5.1.3 Capture the Flag

Same as the other games, to analyze the results of the entire bot training process using Reinforcement Learning, Imitation learning and the resulting usable bots, we first defined metrics to evaluate the training process in reinforcement learning as displayed in Table 5.3 followed by metrics to evaluate the resulting bots and the human recording sessions, finishing by metrics to evaluate bots though with imitation learning.

In metric C1, we will compare the overall average episode time obtained during play. With episode times, it is possible to see if any of the bot teams were able to beat the other within the time limit and if they got faster at it. Then in metric C2 the teams' performance will be compared by analyzing ratio between wins, losses, and draws, between the 2 teams, by comparing average episode times on wins, and the average rewards obtained by the bots on those teams. This allows us to see if one team learned better than the other.

In metric C3 we start by compare the rewards obtained by the bots through the training process, as this allows us to see how the bots did in learning to play the game and

Table 5.3: Metrics for Capture the Flag game

	Metric
C1	Teams Win Ratio
C2	Episode Times
C3	Rewards
C4	Times when Winning

avoiding penalties. Lastly in metric C4 we will compare the teams' performance during play, by comparing average episode times on wins, and the average rewards obtained by the bots on those teams. This allows us to infer a few conclusions on which conditions the bots can obtain wins when in a certain team, if the win spread is similar to both teams or if one team learned better than the other.

To evaluate the level of the bot after the training, we are going to look at averages and deviations of the previous mentioned metrics and compare it to the human level obtained during demonstration records. To evaluate the bots of imitation learning we are going to see the progression of these metrics when different demonstration samples are used, with some added variances. One of these variances is the average episode time. For this one, we will study the average times with all episodes, and exclude draw games to how these times progress with different amounts of demos and if removing the episodes where the bots defected and were unable to complete the game affects in any way the comparisons between averages.

5.2 Training Results

In this section we are going to see the results obtained in the metrics explained in the previous section for each of the games, with the aim to understand how well bots learned each of the games and try to answer the research question of wetter these methods of teaching bots are worthwhile to create game ready bots.

5.2.1 Clean the Bush

To train the bots in the game of clean the bush, the bots were put in an environment with a total of twelve bushes to clean and some spread obstacles to create difficulty to the bot. For better training results, in each episode the spawn point of the bot and of the bushes are randomized to increase variance and to encourage the bots to actually learn to clean the bushes instead of memorizing a set of commands that will always successfully complete the task.

When the bots are training, six parallel environments are running simultaneously to increase learning speeds. Since each bot will have comparable behaviour to its parallel foes only the results of a single environment will be viewed.

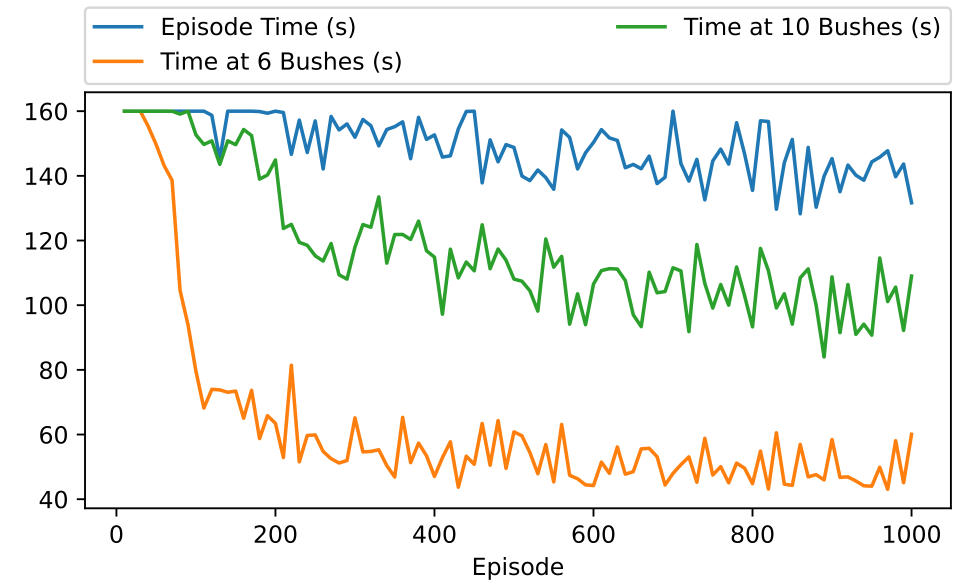


Figure 5.1: Episode time progression through the reinforcement learning training process.

5.2.1.1 Reinforcement Learning

In Figure 5.1 we can see the progression of the time, in seconds, each episode took through the training process. We can see that in the beginning, the bots couldn't complete the game. Very quickly the bot started learning that it should be cleaning bushes and near episode 100 it could already clean half the bushes in approximately 70 seconds, while still struggling to clear the 10 bushes checkpoint. Then after the 200th episode, it

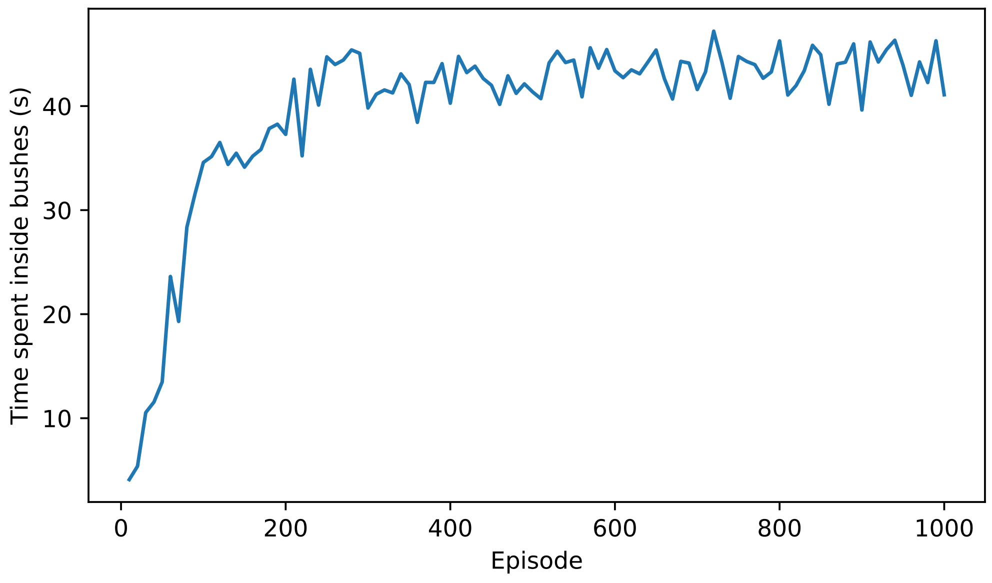


Figure 5.2: Time spent inside bushes by the bots through the reinforcement learning training process.

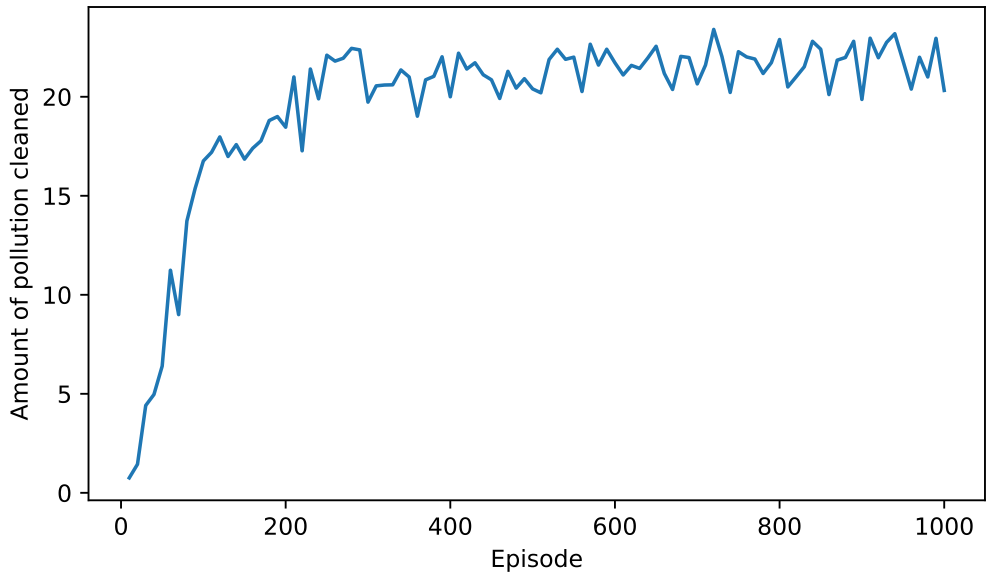


Figure 5.3: Amount of pollution cleaned from bushes by the bots through the reinforcement learning training process.

started being able to clear the 10 bushes check point consistently seeing it time gradually drop to 110 seconds by episode 400 and getting times between 90 seconds to 100 seconds by the end of training. By episode 400 the time needed for half the bushes had already stabilized near the 50 seconds. As for cleaning all the bushes, after episode 250 the bots started seeing some successes, but with times very near to the time limit and still very inconsistently. Until the end of training it gained some consistency in cleaning all 12 bushes and had an average episode time near the 140 seconds.

In Figure 5.2 it is exhibited the amount of time, in seconds, the bot spent inside the bushes throughout the training episodes. This progression has similar tendencies with the episode times, as bush times increase in a similar faction that episode times decreased. Since it takes the bot 4 seconds to clean each bush, to clean all 12 bushes the bot needs at least 48 seconds. We can see that during training, the bot could not consistently achieve those values, but could get really close and on some occasions achieve them from episode 300 forwards. These values also show that the bot early on realised that it should be performing the cleaning action, and that the difficulty was in finding the bushes. If the opposite were to happen (it knew how to find the bushes but couldn't perform the cleaning action), then the bush times would have increased faster then the episode times would have decreased as the bots would spend time inside bushes without cleaning them.

This last conclusion is further reinforced by Figure 5.3 where it is shown the amount of pollution cleaned per episode over the training process. The values displayed are piratically the same as time spent inside the bush, that strengthens the conclusion that the bot learned right away that it should be cleaning and that the biggest obstacle in training was finding the bushes.

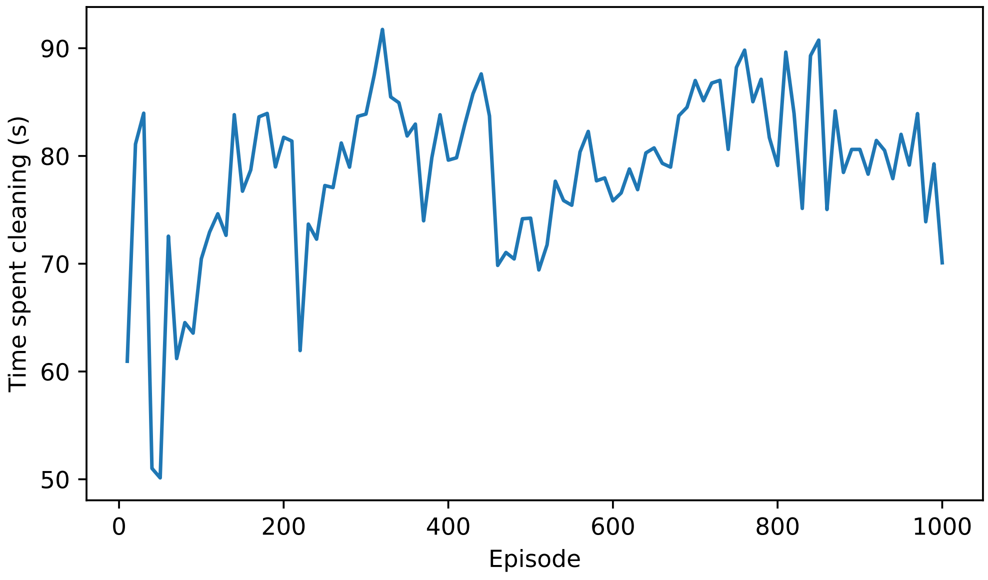


Figure 5.4: Time spent performing the clean action per episode through the reinforcement learning training process.

In Figure 5.4 we can see the progression of the amount of time, in seconds, the bot spent performing the cleaning action. The amount of time the bot spent cleaning was irregular throughout the the training process. The only consistency is that in all episodes the bot spent more time cleaning than it is necessary to clean all bushes, since the lowest value was 50 seconds while only 48 seconds are needed to clean all 12 bushes. This means that the bot has a problem of over-cleaning (cleaning while outside a bush, or inside an already cleaned bush). Over training, we can also see that evolution of cleaning times often shift between increasing values and decreasing values. This means that there are times the bots sees it is receiving reward penalties for cleaning when not necessary and tries to decrease the action, but soon after that it sees some prejudice in the other rewards (not cleaning while inside a polluted bush), and decides that the insurances of over-cleaning outweigh the penalties. A possible solution to this problem is increasing the penalties for cleaning when not necessary to make it strong enough to force the bot to learn to clean only when inside the bushes.

After the training was over, we used the obtained result to make a bot play the game for some time, and see how it performed. In Figure 5.5 it is exhibited the episode and checkpoint times obtained by the bot during the course of nearly 250 episodes to see how was the performance compared to the end training values and if it was consistent in obtaining them. As it was to be expected, the performance of the bot in the course of the episodes played was consistent, having an acceptable level of variance that can be justified by the randomness of each episode bush dispersion. As for the times obtained during the episode, they tend to be very similar to the values obtained in the last episodes of training, with a tendency to be slightly lower which means that the final result has a

Table 5.4: Average metric values obtained in the game by the bots and humans.

	Bot		Human	
	Average	StdDev	Average	StdDev
Episode Time (s)	127.28	26.16	69.95	2.33
Time at 10 Bushes (s)	84.00	16.26	58.31	2.60
Time at 6 Bushes (s)	43.79	7.69	35.29	2.15
Time Spent in Bush (s)	47.22	2.17	48.88	1.39
Time Spent Cleaning (s)	76.58	11.06	51.29	2.13
Amount of Pollution Cleaned	23.42	1.05	24.00	0.00

slight better performance than in training.

The results of the play session with the final bot result were also computed to calculate the averages and standard deviations of the metrics analyzed during training. In addition, these same values were also computed for all the demonstration recording sessions played with human users, so that a comparison with the human level can be made. These values are displayed in Table 5.4.

In a first look, the first thing that stands out is that the bot still fall short of the human level by a significant margin. In time needed to complete the game, the bot averages 127 seconds, while humans only average a time of 70 seconds. This difference of 57 seconds mean that on average humans only need 55% of the time the bot needs to complete the game which is nearly half. If we analyze the times obtained in each checkpoint we can see that this difference is significantly reduced. To clean 10 of the 12 bushes the difference falls to 26 seconds, and now the human users on average needed 69% the time the bot needs, and on the 6 out of 12 bushes check point the difference is of 9 seconds and

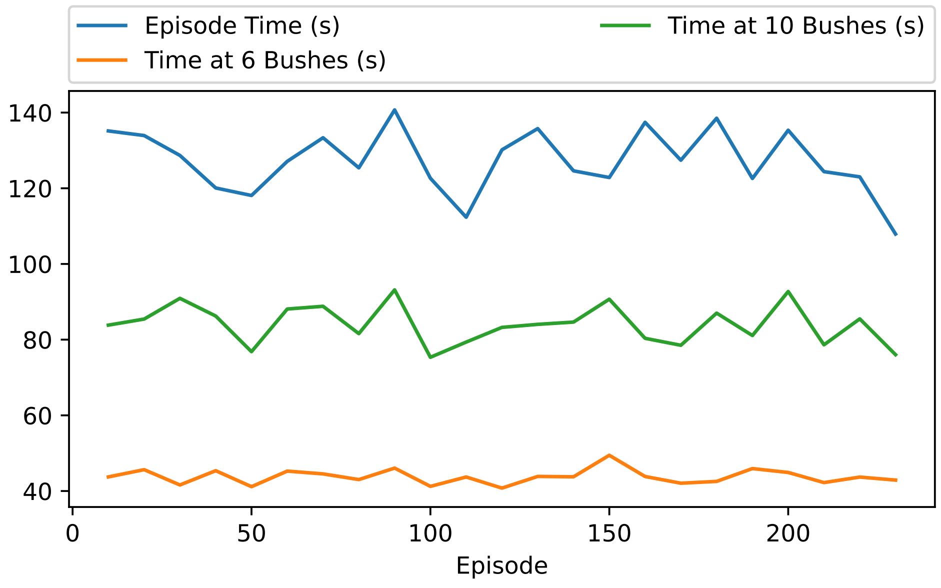


Figure 5.5: Time spent inside a bush and amount of pollution cleaned from bushes by the final result bot after training.

represents the humans taking on average 80% of the time the boots took to clean those bushes. This means that the bot falls short to the humans in finding bushes. While most of the bushes are still polluted and finding one is easy the bot can stay relatively close to the human times, but as fewer bushes are available the more time the bot takes to find a bush compared to humans.

Looking into time spent inside a bush and amount of pollution cleaned, the values between humans and bots were very similar. Since both are able to consistently complete the game, it makes since that there is not much variance in these values since for the most part they only correlate with the ability to complete the game and not the speed in which the game was beaten (only case where this is not true is when the player stays in the bush without cleaning, which is no the case as seen before). The last metric left to be seen is the time spent performing the clean action. Once again, the bots perform considerably worse, as on top of the 47 average seconds they spend inside a bush the spend an additional 30 seconds performing the clean action when they shouldn't be. Meanwhile the human users only spent on average an additional 2 seconds cleaning which can be justified by reaction times. Although this last metric doesn't have a direct impact on performance, it is still important this means that visually the bot is doing the cleaning animation while outside of bushes which is not ideal for a video game experience.

As for variance of the results, once more the bots perform worse than the human users, as the humans had very little standard deviation in all metrics, while the bots have high deviation in time related metrics. This shows that humans were very consistent in completing the game independently of the random dispersion of bushes. Meanwhile the bots are heavily affected by this randomness as some bush dispersions can create to the bots more difficulties in finding the bushes which influence a lot the time the bot takes to beat the game.

5.2.1.2 Imitation Leaning

A total of twelve distinct ten minute game sessions were played to obtain demonstration recordings. The results for these demonstration games are displayed in Table 5.4. In these results, the most relevant numbers are the average time that each episode took, which was 72.96 seconds, the time spent in a bush per episode which was 49.50 seconds, and that the amount of time spent cleaning in each episode averages 51.86 seconds which is only 2.36 seconds more to the time spent in a bush which tells us that the users who player were efficient in their clean action usage. To train the bots, diverse training sessions were carried out started by using 20 minutes of demonstration and adding extra 20 minutes each new train until all 120 minutes were used.

In Figure 5.6 we can see the progression of episode times and the cleaning checkpoints as more time of demonstration was used to train the bots. In these results is is possible to see that bots trained with up to 40 minutes of demonstrations, couldn't even complete the first checkpoint of cleaning 6 bushes, and when using 60 minutes of demonstrations

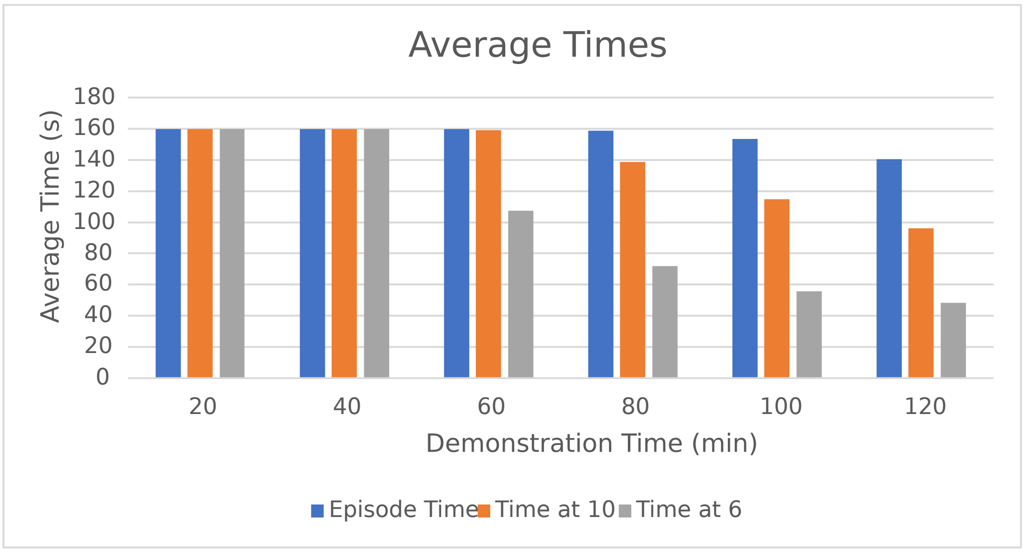


Figure 5.6: Average episode times obtained in bots trained by Imitation with different demonstration samples. Each episode has a time limit of 160 seconds.

the bots could complete the 6 bushes checkpoint, but still couldn't complete the others. Only after using a sample size with over an hour of demonstrations the bots became able of completing all the checkpoints consistently and start beating the game. After the bots became capable of completing the game it can be seen that as more demonstration time was used the faster they became having every checkpoint drop its time. To clean 6 bushes, bots trained with 60 minutes of demonstrations needed on average 107 seconds while bots trained with all 120 minutes only needed on average 48 seconds to clean the same amount. To clean 10 bushes the same drop can be seen as the times dropped from 138 seconds to 96 seconds with bots trained with 80 and 120 minutes of demonstrations respectively. The total episode time also decreases going from 158 seconds to 140 seconds.

In Figure 5.7 we see the average amount of pollution the bots trained with different demonstration times were able to clean along with the standard deviation of these time. This metric provides a bit more insight into how the level of the bots progresses as it isn't depended in bots achieving a certain checkpoint. Once more, it is possible to see that the level of the bots gradually increase as more demonstration time was used to train, as in the beginning the bots could barely clean any pollution, then with 40 minutes of demonstration bots could clear the equivalent of 2.5 bushes (each bush has an amount of 2 pollution), and the gradually increasing the amount cleared up until the end where bots could clear an amount of 23 pollution which means on average it cleaned 11.5 bushes per game. Looking at the deviations, we can see that overall all bots were consistent in the amount they would clean as the amount cleaned never tented to deviate more than 2.5 from the average amount.

In Figure 5.8 we see the times the bots spent inside a bush and performing the cleaning action. Looking at times spent in bush first, we can see that once more this time increases as more samples were used to train, and that the times are very close to the minimum time needed to clean the amount of pollution each bot cleaned on average (each unit of pollution cleaned corresponds to 2 seconds cleaning). This shows that the bots don't spent more time than necessary inside of a bush independently of how much time of demonstrations was used to train. Looking at times spent performing action, we can see that even in imitation the bots have a hard time of understanding that they should only clean while inside a bush, spending way more time doing the cleaning action than they were inside a bush.

To summarize what can be concluded, it can be seen that more time of demonstrations means increase of performance for the bots. When more samples were used the bots got faster at doing their tasks and were able to clean more bushes. As for comparison to both human level and reinforcement learning, it can be seen that even when all 120 minutes of demonstrations were used, this method falls short to reinforcement learning trained bots and these last ones were faster at cleaning the bushes. This means that both these methods are still quite far to the human level, as time needed to clear the first check point of 6 bushes are 8 to 11 seconds faster compared to both RL and Imitations, and the time humans needed to clear the game on average are 57 to 70 seconds faster than the bots trained with both these methods.

5.2.2 Carry the Box

To train the bots in the game of carry the box, the bots were place in an environment with two levels (the upper and the lower level), where the upper bot had to carry a total of 6 boxes and the lower bot had to carry a total of ten boxes (four initial boxes to accelerate learning plus the six boxes from the upper level). For better training results, in each

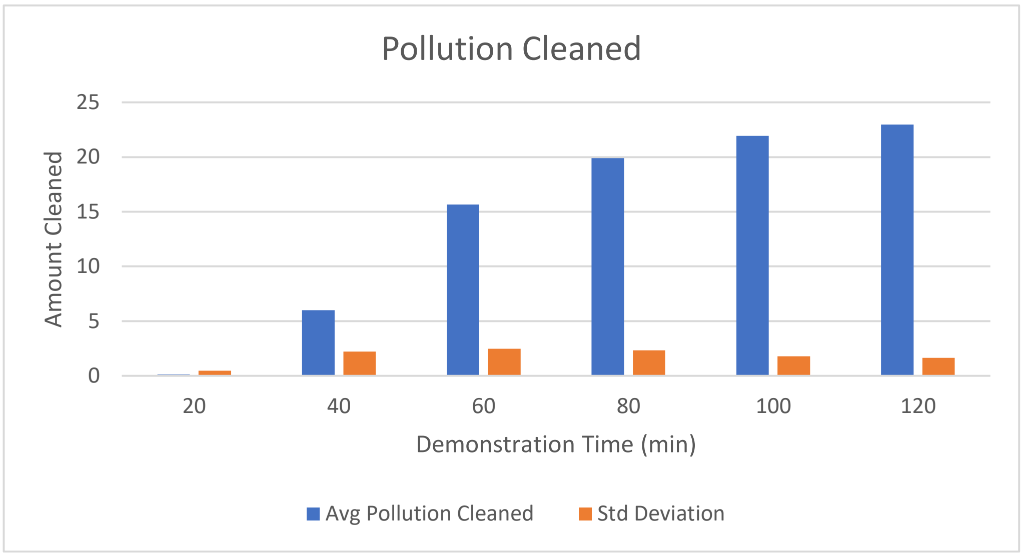


Figure 5.7: Average amount of pollution cleaned from bushes by bots trained with Imitation.

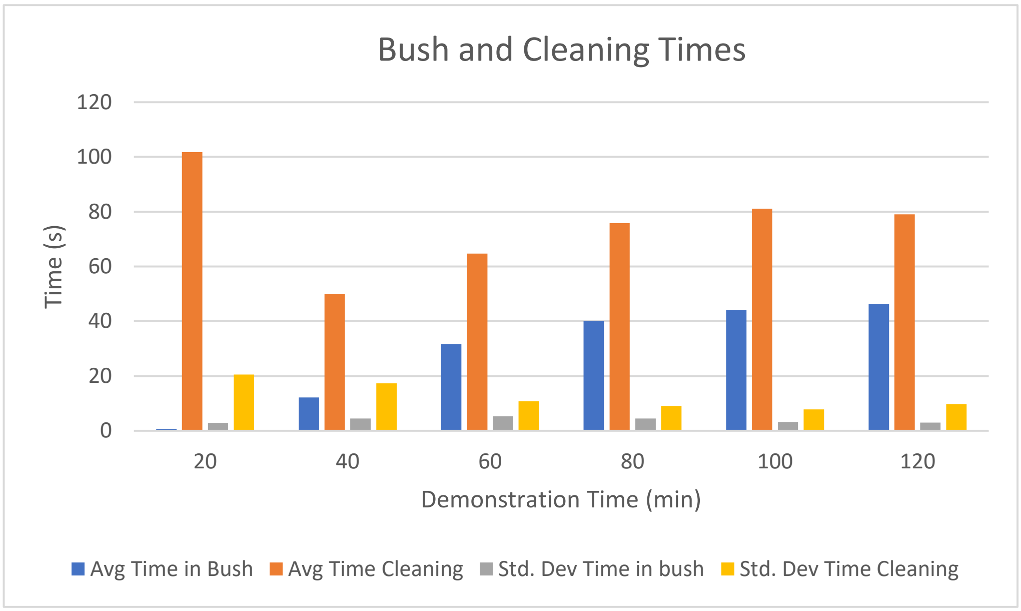


Figure 5.8: Average time spent inside of bush and average time spent performing the cleaning action by bots trained with Imitation.

episode the spawn point of the bot and the boxes are randomized to increase variance and to encourage the bots to actually learn to find and carry the boxes instead of memorizing a set of commands that will always complete the game successfully.

When the bots are training, six parallel environments are running simultaneously to increase learning speeds. Since each pair of bot will have comparable behaviour to its parallel foes only the results of a single environment will be viewed.

5.2.2.1 Reinforcement Leaning

The first metric seen for the game of Carry the Box is the time each of the bots needed to complete their task which can be seen in Figure 5.9. Throughout the training process the bots got gradually faster at carrying their boxes, only stabilizing their times after episode 1000. One noticeable difference is that the lower level bot has a more steady decrease of times as well as them being more consistent while the upper level bot have bigger spikes in time decreases as well as having more inconsistent times. This can be explained due the fact that every box in the upper level is randomly place having different distances to the destined box drooping spot, while the lower level bot only has the initial 4 boxes with randomized positions while the 6 that drop from the upper level have similar positions. This causes the upper level times to have higher variance as luck in spawn positions can influence the times a lot while the lower level has some variance in the first boxes but most of them have the same position to the destination leading to more consistent and less luck dependent times.

In Figure 5.10 we see the progression of the number of boxes carried in each episode

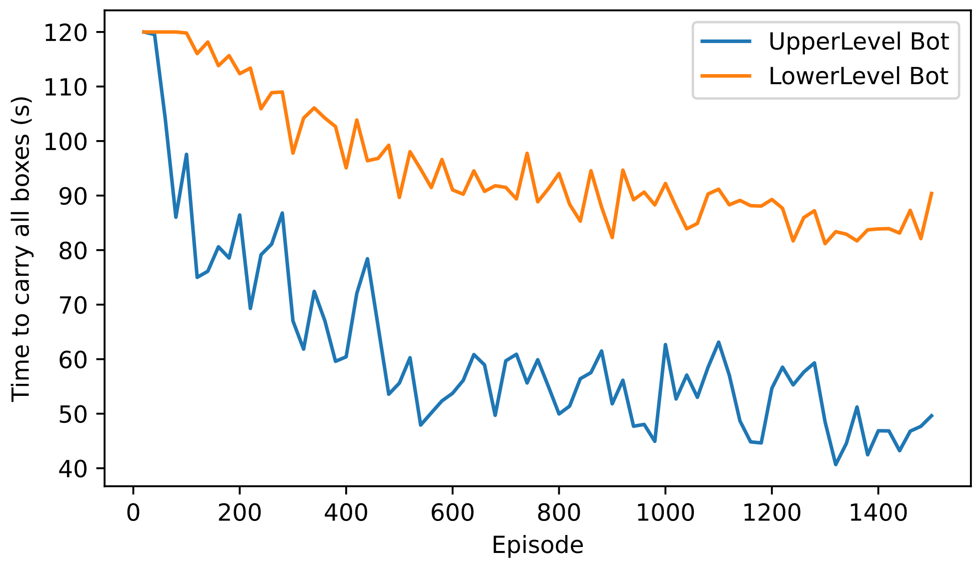


Figure 5.9: Progression of the time of each game iteration (episode) through the reinforcement learning training process

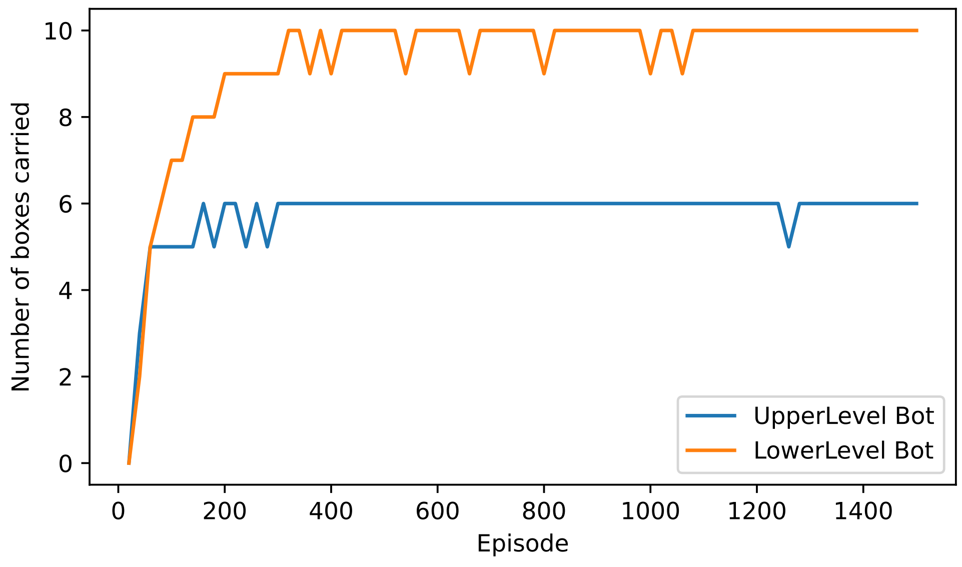


Figure 5.10: Number of boxes successfully carried to the destination by both bots during the reinforcement learning training process.

by each bot. With this we can see that the bots took about 200 episodes for the upper level bot, and 300 episodes for the lower level bot, to learn how to carry all the boxes to the objective zone. This can be seen as it was near these marks the bots successfully carried 6 and 10 boxes respectively during training. After this mark, each bot needed near 100 extra episode to consistently carry all boxes in nearly all remaining episodes with exception of a few.

In both of the metrics viewed so far, it can be seen that both bots tend to learn at similar rates, with the lower level bot taking some more time to carry all boxes and needed a few extra episodes to successfully carry all boxes as it has to deliver more boxes than its upper level counterpart. This strengthens the idea the even though they are performing in different places with slight differences in their task, the same behaviour can be used for each bot as the core of their job is the same, so knowing how to perform in one level means the bot also knows how to perform in the other level.

In Figure 5.11 we see the time each bot spent carrying a box. The progression of this metric is similar to the progression of boxes carries, as they peak and stabilize around the same number of episodes. This shows that by the time the bots consistently carried all boxes, they knew how to go to the box delivery spot as fast as they could. This also means that the variance in time to complete the objective is in getting to the boxes and not in carrying them to the objective area. With this it is also possible to conclude that most of the learning done from episode 300 further was in picking up the boxes faster as this is the only way left to decrease the times since the time spent carrying boxes was constant throughout the episodes.

In Figure 5.12 we see the rewards obtained by the bots over the episodes they played during the training process. With these values we can see how the bots did in terms of learning how to do the positive actions but also avoid the negative ones. We can see that the rewards increased very quickly, only having negative rewards in the first few episodes and stabilizing after 200 episodes for the upper level bot and 400 episodes for the lower level bot. This just reinforces that the bot could complete the game consistently after 400 episodes of learning and that afterwards all progression was on time as this doesn't affect

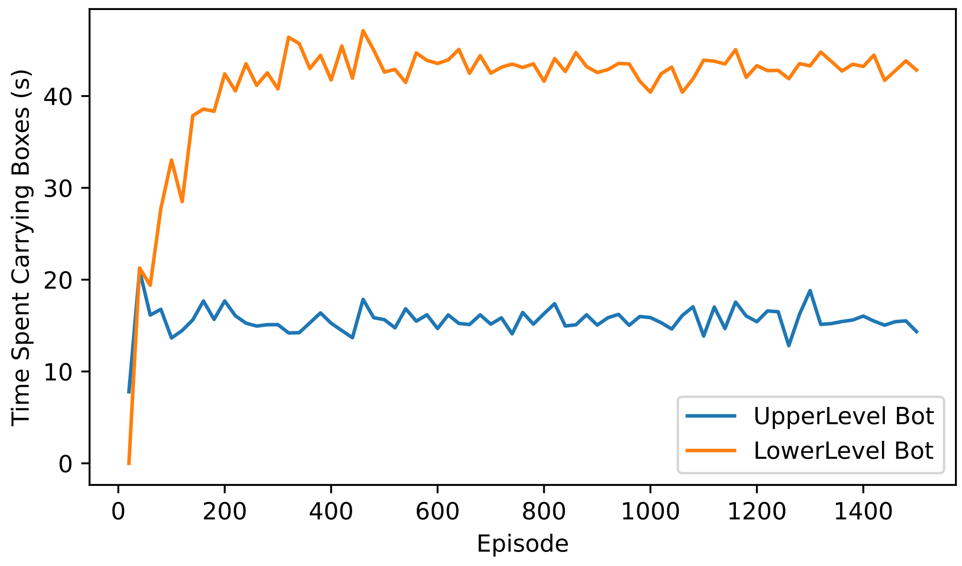


Figure 5.11: Time a bot spent carrying a box through the reinforcement learning training process.

Table 5.5: Averages obtained by bots after training with Reinforcement Learning.

	Upper Bot		Lower Bot	
	Average	StdDev	Average	StdDev
Episode Time (s)	84.02	10.59	84.02	10.59
Time at 3/4 Boxes (s)	14.37	6.84	24.39	4.25
Time at 6/8 Boxes (s)	46.70	20.34	61.50	8.23
Time Carrying Boxes (s)	15.52	3.67	43.53	3.18
Boxes Carried	5.97	0.28	9.97	0.38
Rewards	4.3	0.39	7.40	0.37

rewards.

In Tables 5.5 and 5.6 we see the average results for each metric obtained firstly by the bots after the training process, then by the human users during demonstration recording sessions. If we compare the human level with the lower bot which played with the same condition, carrying 10 boxes total, we see that once more the level of reinforcement learning still falls short quite a bit. On average the bots take 84.02 seconds to carry all boxes while humans only needed 69% of the time (58.38 seconds) to carry the same amount. This difference is seen in all the check points with the humans only taking 69% and 68% of the time the bots took to get to the 4 boxes and 8 boxes checkpoint. A good portion of this time is spent while carrying a box, since on average the bots spend more 14.54 seconds with a box in their hands. As for rewards the values are very close since both successfully complete the task, but the bots still fall short by a tenth for occasionally colliding against the borders getting them a penalty.

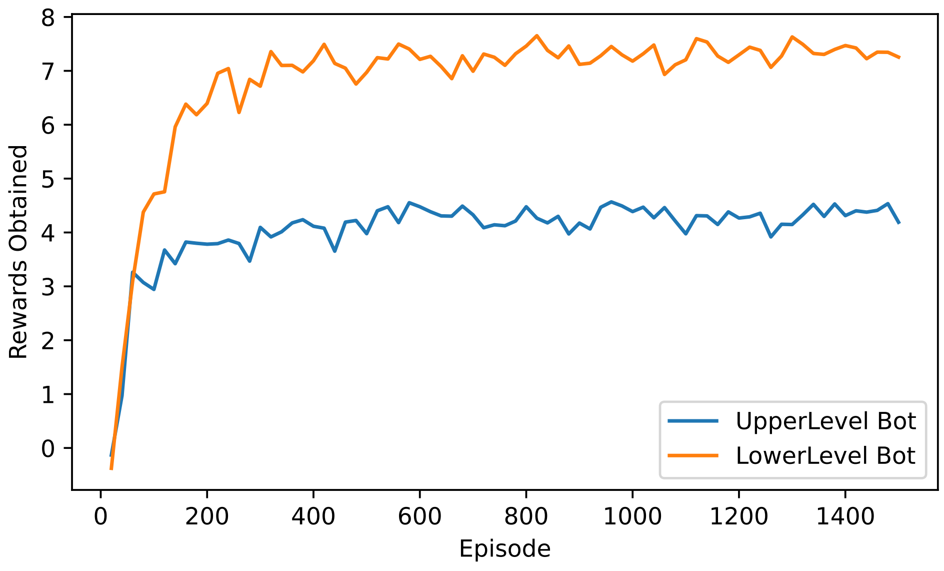


Figure 5.12: Rewards obtained by both bots through the reinforcement learning training process.

Table 5.6: Averages obtained by the human users during demonstration recording sessions.

	Average	StdDev
Episode Time (s)	58.38	6.07
Time at 4 Boxes (s)	16.94	2.35
Time at 8 Boxes (s)	41.88	5.01
Time Carrying Boxes (s)	28.99	3.19
Boxes Carried	10.0	0.0
Rewards	7.5	0.0

5.2.2.2 Imitation Learning

A total of twelve distinct ten minute game sessions were played to obtain demonstration recordings for a total of 120 minutes of demonstration time. The setting for these game sessions was a single level with a total of ten boxes. The results for these demonstration games are displayed in Table 5.6. In these results, the most relevant numbers are the average time that each episode took, which was 58.38 seconds, the time spent carrying a box per episode which was 28.99 seconds, and that the amount of rewards obtained in each episode averages 7.5. To train the bots, diverse training sessions were carried out started by using 20 minutes of demonstration and adding extra 20 minutes each new train until all 120 minutes were used.

In Figure 5.13 we can see the progression of the episode times as more demonstration time was used to train the bots. Overall we can see that independently of which level the bots played, the times the bots needed to complete both the checkpoint and the game dropped as more demonstrations were used to train. This progression shows that even though the demonstrations were recorded in a scenario with a single level of boxes to carry, this doesn't affect the learning capability of neither bot as they only need to learn how to pick up a box and how to drop it at the desired locations which works the same for both levels of play. As for the speed the bots carried the box, we can see that even with only 20 minutes of demonstration, the upper level bot could already complete it's objective even if barely. As it needs to carry more boxes, the lower level bot only managed to complete its task within the time limit after training with at leas 60 minutes of demonstration.

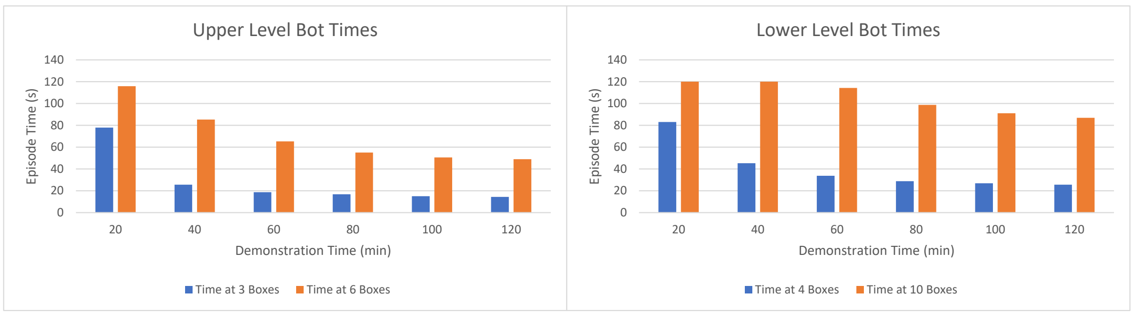


Figure 5.13: Average episode times obtained by the bots trained with imitation.

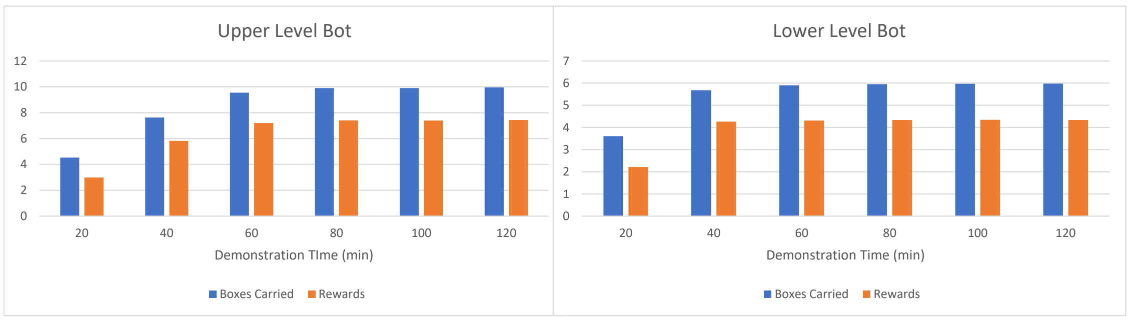


Figure 5.14: Number of boxes successfully carried to the destination and rewards obtained by the bots trained with imitation.

In Figure 5.14 we see the number of boxes successfully carried and the rewards obtained by the bots trained with different demonstration times. Analyzing the number of carried boxes, we can see that with any amount of demonstrations the bots could complete at least a part of their jobs, with the upper level bot having an average of 4.5 boxes with 20 minutes of demonstration which is nearly half of the objective, and the lower level bot having an average of 3.6 boxes which is more than half of its objective. It is also possible to see that with only an hour of demonstration time, both bots could complete their objective consistently. Looking at rewards we can see that they grow linearly with the growth of boxes carried, which indicates that at no point in the different training the bots had any form of penalties. This shows that even with 20 minutes of demonstrations, the bots could avoid colliding against the walls.

In Figure 5.15 we see the amount of time each of the trained bots spent on average carrying a box. This metric is used to measure how much time a bot spends with a box in its hands to see whether the bots are efficient and deliver the box to the destined place right away or if they take long to accomplish that task thus spending more time carrying a box. Looking at the results for the first time we can see different behaviour depending on which level the bot is playing. The lower level bot saw its times decrease as more demonstration time was used to train, which demonstrates that the bot got increasingly

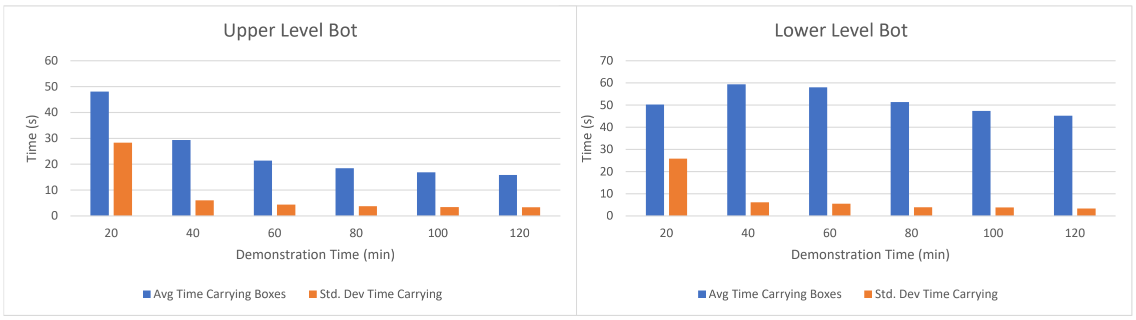


Figure 5.15: Average times that a bot spent carrying boxes and standard deviation of the values obtained by the bots trained with imitation.

more efficient in delivering the boxes once it got the hold of it. Meanwhile the same behaviour cannot be seen in the lower level bot as in the beginning it sees the carrying times increase and then after 60 minutes of demonstration only sees a slight decrease in times. This can be justified by the fact the the lower level bot saw an increase in the number of boxes it had to carry as the other carried all boxes to the lower level. More demonstration time would be needed to see if the carrying times would keep decreasing as the bot got more efficient.

In conclusion, it can be said that the tendencies seen in the clean the bush game can be also seen in this game. Firstly, the more time of demonstrations used to train means increase of performance for the bots. When more samples were used the bots got faster at doing their tasks and were able to carry all the boxes faster. As for comparison to both human level and reinforcement learning, it can be seen that even when all 120 minutes of demonstrations were used, this method falls short to both RL trained bots and the human level.

5.2.3 Capture the Flag

To train the bots in the game of capture the flag, the bots were place in an environment with four obstacle walls and six randomly dispersed balls both evenly scattered in each teams field. For better training results, in each episode the spawn point of the balls are randomized to increase variance and to encourage the bots to actually learn to find find the balls to throw at enemies. When the bots are training, four parallel environments are running simultaneously to increase learning speeds. Since each pair of teams will have comparable behaviour to its parallel foes only the results of a single environment will be

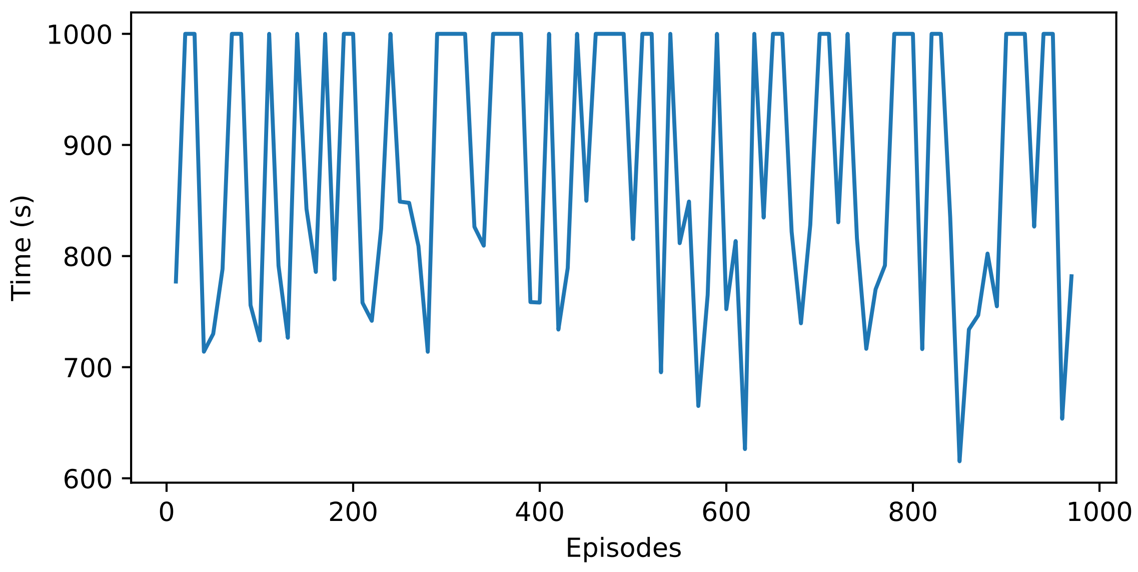


Figure 5.16: Time progression of each game iteration (episode) through the reinforcement learning training process.

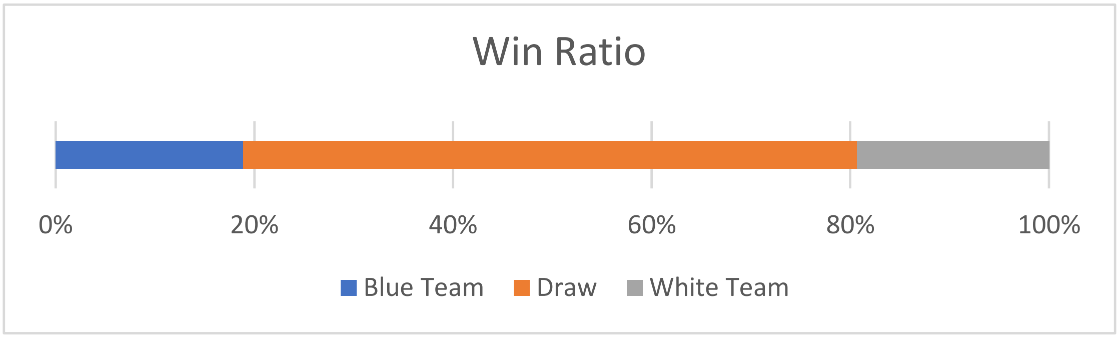


Figure 5.17: Win ratio of the teams during the reinforcement learning training process.

viewed.

5.2.3.1 Reinforcement Leaning

In Figure 5.16 we see the time progression each episode took through the training process with reinforcement learning. In this game we see that reinforcement learning trained bots are having a harder time learning the game. At any point of the process we see a clear consistent decrease in episode times, and game draws keep happening as neither team is capable to consistently beat the other.

Looking into Figure 5.17 where the win ratio of the teams during the training process is displayed, it is possible to verify that the draw is the most common result with it being the end result of slighly over 60% of games played. As for the games that ended with a team winning, it is also possible to see that the teams won a similar amount of games

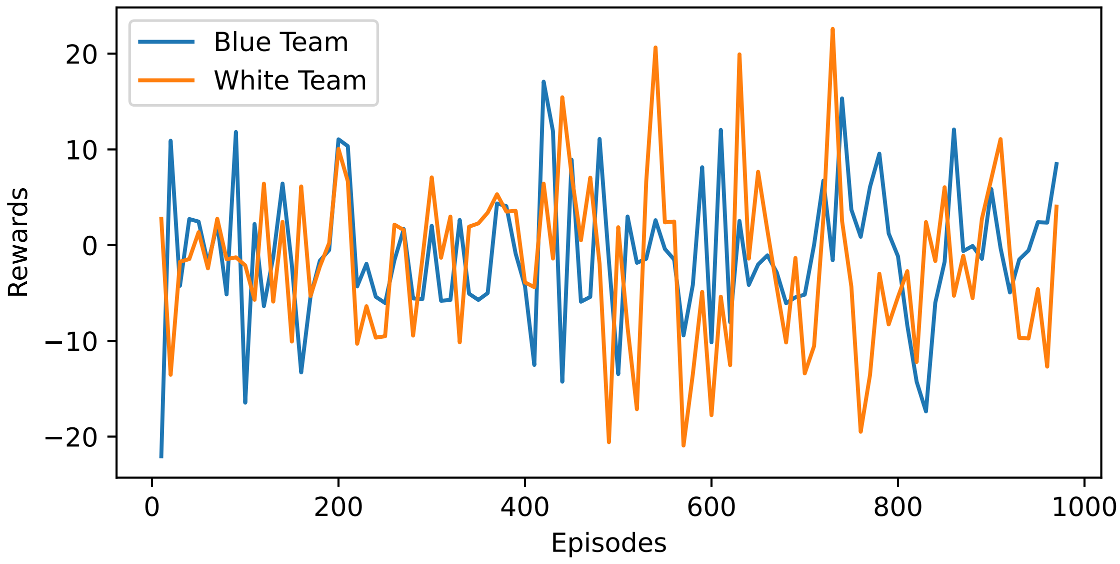


Figure 5.18: Reward progression of both team through the reinforcement learning training process.

Table 5.7: Recording Games Results

	Blue	White	Avg Time(s)	StdDev
(1*vs*1)	58	22	35.481	12.273
(2*vs*2)	34	19	31.839	17.224
(2*vs*2)	30	14	34.306	9.547
Total	122	55	34.098	10.248
%	68.9	31.1		

both having nearly 20% of game wins.

In Figure 5.18 it is shown the progression of the reward of both teams through the reinforcement learning training process. Once more these results are not very indicative of performance evolution from the bots as the rewards keep floating through the course of training. The only inference possible to make from this is that the bots interact with the ball and throw them as rewards are not symmetric and both teams manage to get positive rewards simultaneously. But is is also seen that the bots are having a hard time avoiding the obstacles as rewards tend to be negative.

5.2.3.2 Imitation Learning

Three distinct game sessions were played to obtain demonstration recordings. In total 300 minutes demonstrations were obtained. These game sessions were played in a set of 1*vs*1 and two sets of 2*vs*2. The results for these games are displayed in Table 5.7. In these results, the most relevant numbers are the average time that episodes took to finish which is 34 seconds, and the fact that the blue team won every game set played with an episode

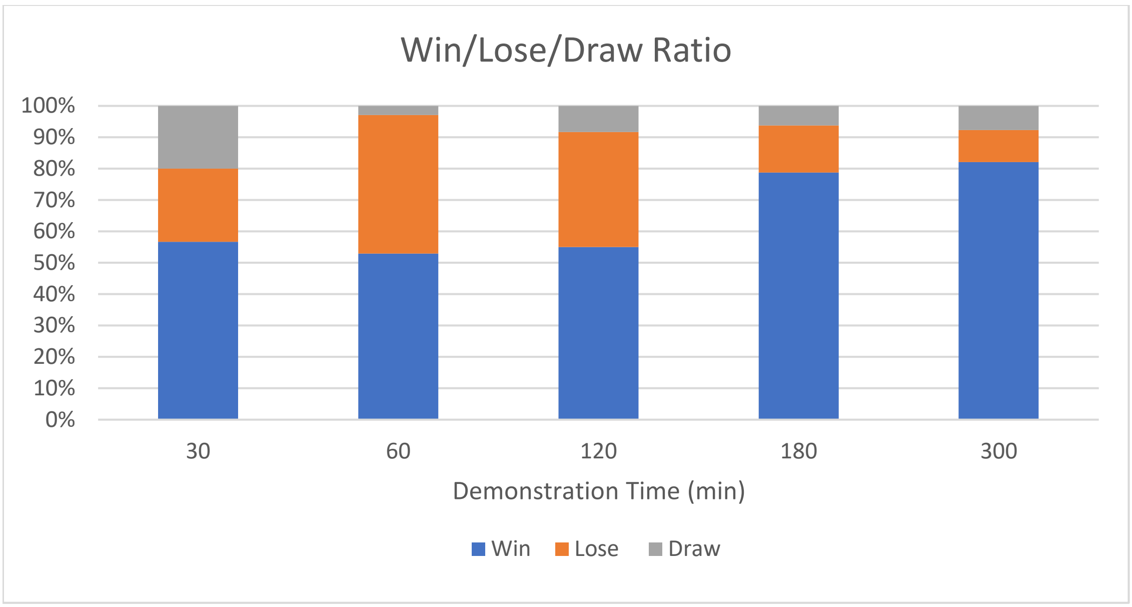


Figure 5.19: Win/Lose/Draw ratio from the blue team team perspective in bot teams trained with different demo samples.

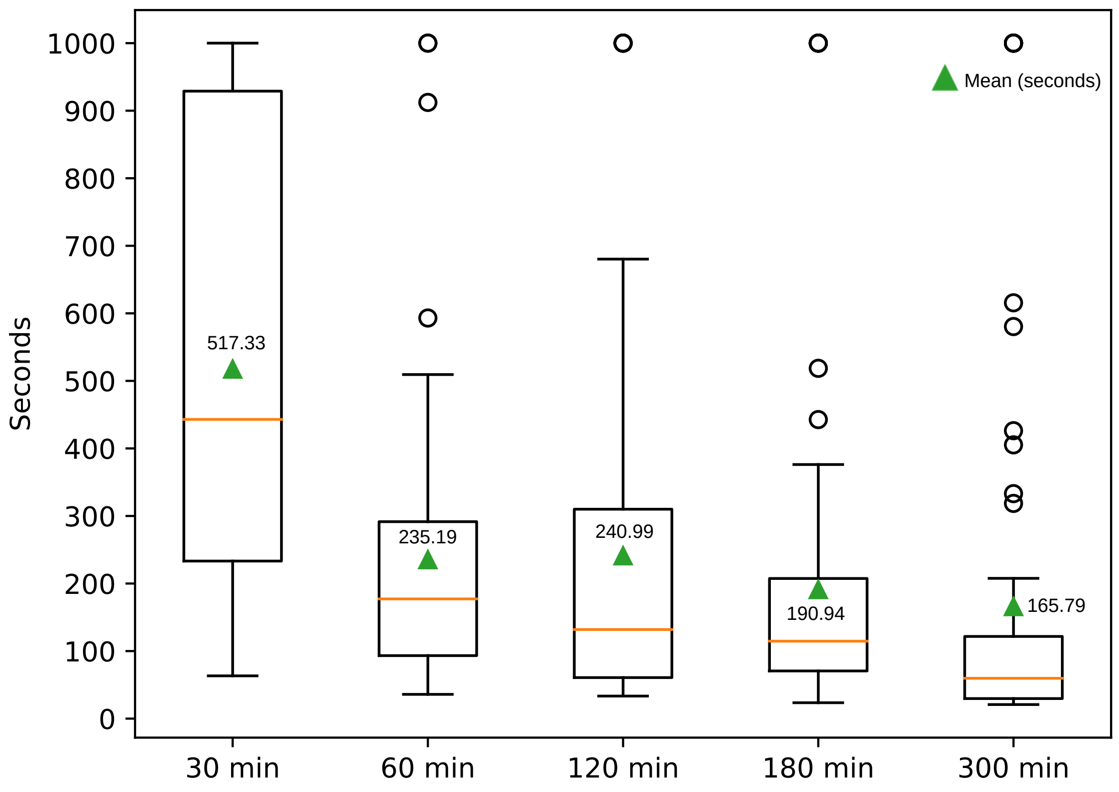


Figure 5.20: Average episode times obtained in games with bots trained with different demonstration samples.

win rate of 68.9%. This disparity in wins can be an interesting detail as we'll analyze how this affects the win rates of both teams when the trained bots play. If we only train a bot with the winning team we could get better results.

Using the recordings we trained bots in two teams. Once again, the blue and white teams were used. Each bot trained using every demonstration available independently if it was a blue human team or a white human team recording. The correlation between the bot and human teams is that the blue bot team starts on the same side as the blue human team, and the same for both white teams.

As mentioned in Section 5.1.3, the first metric that is going to be viewed is the ratio between wins, losses, and draws in the blue team perspective (white team win ration is symmetric) which is displayed in Figure 5.19. After analyzing these numbers and the context behind them there are a number of conclusions one can infer. First, it that the blue team winning most of the game during the recording sessions impacts the bots' results as the blue team maintains a higher chance to win than the white team. This occurrence seems to intensify the more demonstration time was used, as bots seem to become more proficient, and random results seem to dissipate. Secondly is that draws seem to be an odd event, that occurs randomly when the bots find themselves in a state where they are unable to know what to do, and can't find a way to win. Expectations were that this value

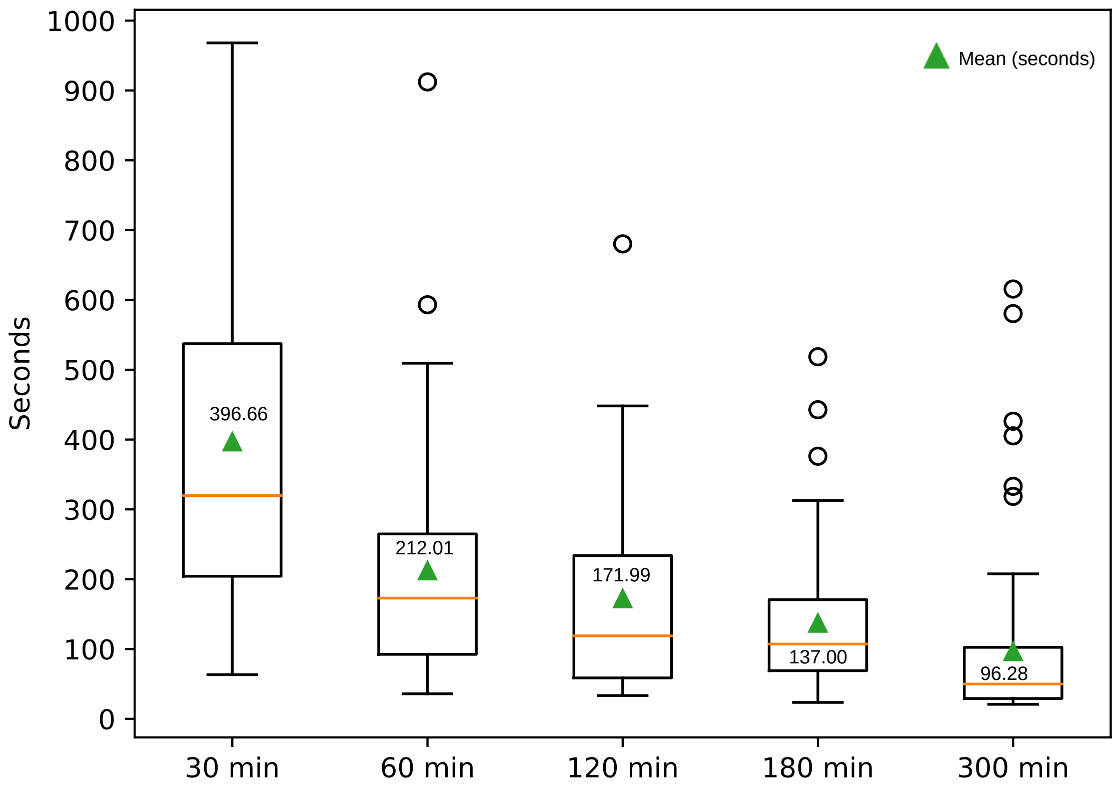


Figure 5.21: Average episode times throughout all episodes excluding games finished with a draw result.

would reduce the more demonstrations used, but with the samples used it seems to be more related to randomness, although the number of samples used also seems to have impact on it.

To better analyze the idea that the time of demonstration used influences the quality of play of the bots, we're going to analyze the overall average episode time for the bots trained with different times of demonstration as well as see what happens to this average if we remove the draws as these seem to be odd events. These numbers are shown in Figures 5.20 and 5.21 respectively.

The results on episode times seem to be pointing in the same direction as the other games trained before. The more demonstration time was used for training, the average episode times become shorter and more compact. The only oddity in these numbers is for 60 and 120 minutes of demonstration where the range of times is larger in bots trained with 120 minutes of demonstration. But even this can be explained on the random occurrence of the draw, as if we exclude the episodes finished with a draw from the times, the time ranges normalise with bots trained with 120 minutes of demonstration having more compact results than bots trained with 60 minutes.

As for comparing the results of the times with and without draws, as obvious without draws the averages times drop. This drop has more effect in plays with more draws.

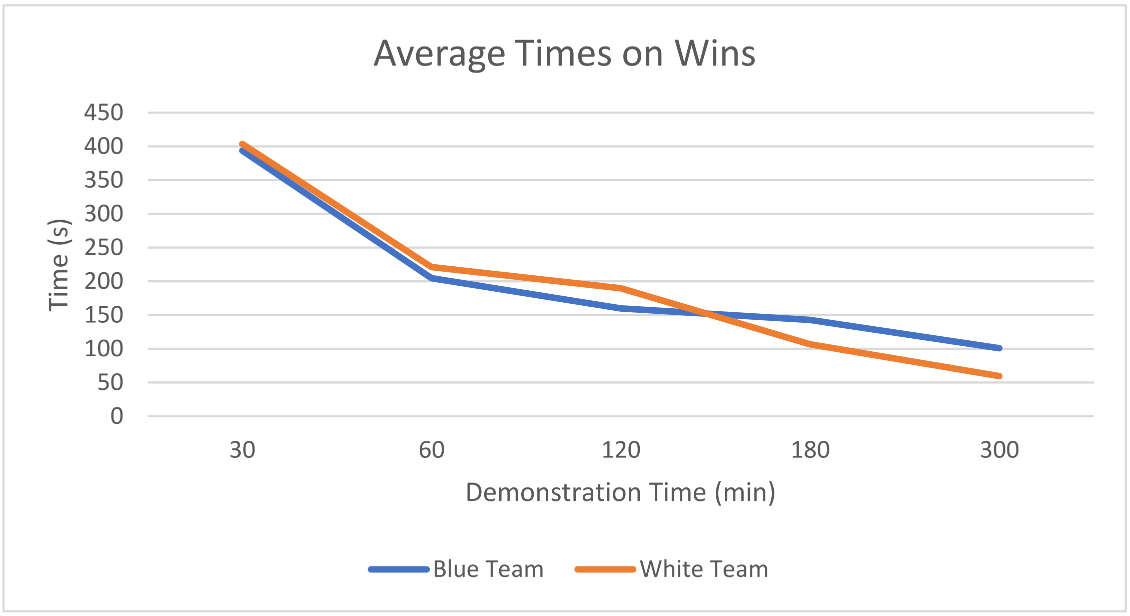


Figure 5.22: Average times obtained in episodes where winning was the final result in bots trained with different demonstration samples for each team.

Besides this, analyzing the averages without draws provides a more regular progression curve of the time drops while the time of demonstration used is increased. If we compare these times with the ones of the users that recorded the demonstrations obtained, we see that it's still far off from those times, as in the best scenario with 300 minutes of demonstration used, the average is 60 seconds slower than the users. But the average curve seems to point out that with more demonstrations for training, the bots could eventually achieve similar average times.

Next, we are going to compare the episode times on the games won between both teams in an attempt to understand in which conditions each team was able to obtain wins, and how this progresses the more demonstrations were used during training. This results can be seen in Figure 5.22.

The analysis of the times during wins by a team, reinforces the idea that the teams' results in recording sessions influence the teams' results for the bots. Overall we can see that for the same time of demonstration, the average on win times begins shorter for the blue team and after 120 minutes of demonstration time used the white team has a shorter times on wins. This is due the fact that in the beginning when both team are really inefficient at the game, the wins are more distributed between teams but the blue one understands the game a bit better which makes them be able to win a bit faster. After they become more proficient at the game, the situation subverts because the games are less random, and the blue team manages to get wins in both short and long games since they understand the game better, while the white team need to get their "perfect game"to obtain a win otherwise they can't beat blue team.

Lastly, we will take a look at how the average rewards for both teams progress as more

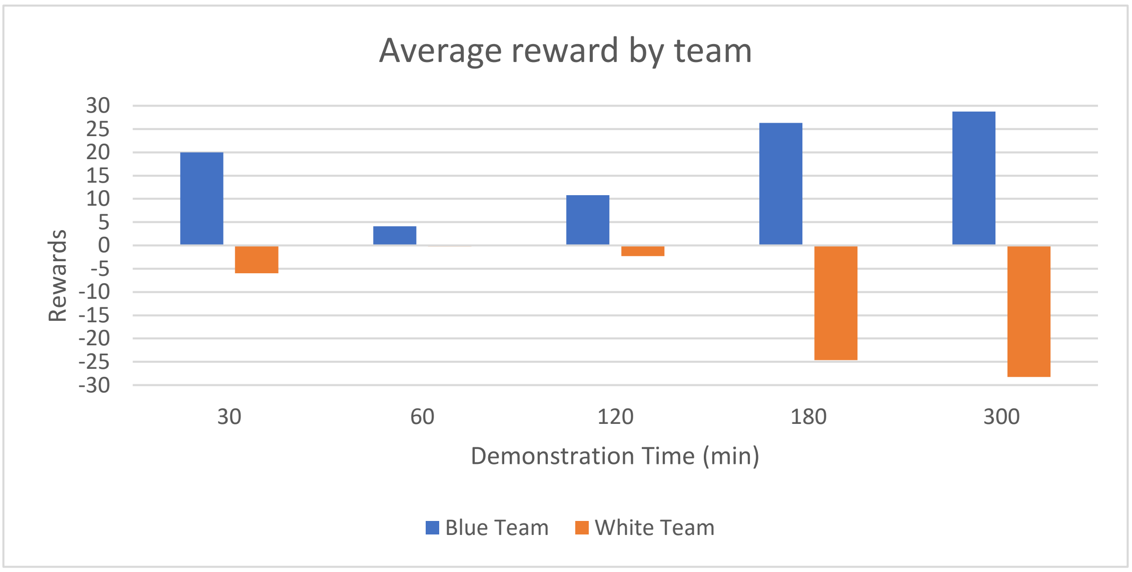


Figure 5.23: Average rewards obtained through all episodes in bots trained with different demonstration samples.

demonstrations were used for training as shown in Figure 5.23. As for the rewards it goes as was to be expected, the blue team tends to have higher rewards as they win more. If we exclude the game with bots trained with only 30 minutes of demonstration time, as this one tends to be more random, we see that the rewards go up for the blue team and down for the white team the more demonstrations were used for the training. There is a big jump in rewards from using 120 minutes of demonstration while training, to using 180 minutes while training as this corresponds to the moment when the blue team starts dominating with win rates higher than 80% as the win rates correlate directly with the average amount of rewards obtained.

6 Conclusion

In this thesis, we present a method to create game engine self-learning bots in the Unity3D that use reinforcement learning and imitation learning algorithms. This technique was achieved using the described ML-Agents plugin. This method was based in the idea that we can create any game environment in Unity, and use the game bots as agents for the simulation to teach them how to play the games. Then based on both visual and numeric results the bots will either have achieved a desired level, or still fall short in which the process can be repeated to improve performance further.

To demonstrate this method, first we created two simple games: **Clean the Bush** and **Carry the Box**. Training with reinforcement learning we could see that the bots manage to start consistently beating their games within the time limit after each individual bot played 300 episodes, which amounts to a total of 1800 learning episodes as each training had 6 parallel environments. Less positive, is the fact that in both games learning stabilized before the training finished, and the end results were still far off from the human level, with the humans being on average 45% faster in the clean the bush game and 30% faster in carry the box. Since learning stabilized it is hard to think that increasing training time would improve the performance further but changing the rewards policy certainly could help. For instances the most obvious way to try and improve the performance is making the rewards time dependent creating an explicit incentive for the bots to be faster.

Training these games with imitation learning, it was possible to see that the core idea that we can loop the training process to improve the bots performance works. Considering that each training with 20 minutes of demonstration is an iteration of the loop described in Section 3.2.2, each increment of 20 minutes of demonstration in training represents the repetition of the process in recording more demonstrations and training once more as a means to improve the performance. As more demonstration time was added the performance saw an improvement and at no point this improvement saw a stabilization where the performance stop improving. As a downside, once more the performance never achieved a level close to human level, staying at a similar level to bots trained with reinforcement learning.

Then we created a $3vs3$ game of capture the flag and created a multiplayer component for the game that could coexist with the ML-Agents plugin, so that we could have multiple people recording demonstrations that could be later used for training. In reinforcement learning, sadly we saw negative results as the bots were unable to successfully learn how to play the game. As for the imitation learning training some more interesting results were seen. During the recordings with humans, the blue team won around 70% of the games, which allowed us to conclude that in team games, a team performing better than the other during the demonstration recordings will also affect the bots' performance depending on which team they are as the bot's blue team also performed better.

Analyzing the training results, it was possible to conclude that the bots do indeed get better the more demonstrations they were provided. This impacted metrics such as, the blue team win rate, which was about 60% when using fewer amounts of demos but then skyrocketed to 80% when more demonstrations were provided (Figure 5.19). After training the level of the bots was still worse than the humans since with 10 demonstrations they performed 3 times slower than humans. That said, the tendency for average times to drop shows that with more demonstrations, the bots could get to a closer level to humans.

After considering the results of the experiments, we can look into our research questions. The first research question on how to integrate off-the-shelf machine learning algorithms in a game-engine in order to execute and see the simulations, we saw that we can easily use game bots as simulation agents, by converting movement and actions to numeric values, such as 0 to stay still and 1 to move forward, and numeric representation of the observations of the bot. By having this numeric representation, it is possible to integrate the algorithms, as they only need to receive the set of observations to process the calculations, and have a set of numbers it can manipulate to command the bot agent. The second question how the bots behaved depending on which training process they used. In the simpler games, we saw that both methods performed similarly, but neither performed optimally so it one could still perform better or differently than the other as the training was tuned to perform better. In the more complex game of capture the flag, imitation learning performed much better than reinforcement learning, as in the first one the bots were able to comprehend how to play and beat the game in a consistent manner even if performing worst than humans, the bots trained with RL simply couldn't consistently finish a game within the time limit and when they did, would be really slow to do so. As for the last research question of these methods of self-learning bots are worthwhile to teach game ready bots, the answer so far is that it is not, as the level of play of these bots could at best be considered beginner level, as most new players of the games would still be able to perform better than the bots. In the future with improvements to the teaching method and training process, a better level could be achieved but with the result obtained it is not possible to consider the method worth for creating bots for games.

As for improvements of the current work, we intend to test the same scenarios by tweaking the reward and observation functions to observe the effects it has on the bots and attempt to improve the performance. For Imitation Learning, we intent to increase substantially the sample of demonstrations available to train, to confirm if the bots can actually get to a better level with an increased sample size. Lastly, we intend to test different types of games with varying levels of complexities and different variables to the games already tested, so different kinds of results can be obtained so that it is possible to infer new conclusions about this method as well as to solidify the conclusions already obtained in this experiment.

For future work, it is interesting to explore the idea of segmented learning, where a game is segmented into incremental parts that can be though sequentially. The idea with this is that a bot can master a complex task quicker, if it is though how to play each part of the task individually, and gradually joining them together. Besides this, it is also interesting to study the concurrent use of both Reinforcement Learning and Imitation Learning to teach bots, as it can be the best of both worlds where the imitations serve as a starting point for the bots to start playing the games, and then gradually learning further from the starting point by creating new knowledge through reinforcement learning.

Bibliography

[1] H. A. Ameden et al. “An Agent-Based Model of Border Enforcement for Invasive Species Management”. In: *Canadian Journal of Agricultural Economics/Revue canadienne d’agroeconomie* 57 (4 Dec. 2009), pp. 481–496. ISSN: 00083976. DOI: 10.1111/j.1744-7976.2009.01166.x (cit. on p. 1).

[2] A. Bąk and M. Wojciechowska. *Using the game engine in the animation production process*. 2020. DOI: 10.1007/978-3-030-14132-5_16 (cit. on pp. 16, 17).

[3] B. Baker et al. “Emergent Tool Use From Multi-Agent Autocurricula”. In: *ICLR 2020* (Sept. 2019). URL: http://arxiv.org/abs/1909.07528 (cit. on pp. 10, 14).

[4] C. Bartneck et al. “The Robot Engine - Making the Unity 3D Game Engine Work for HRI”. In: *24th IEEE International Symposium on Robot and Human Interactive Communication* (2015). DOI: 10.0/Linux-x86_64. URL: https://www.openrobots.org/wiki/morse/ (cit. on pp. 16, 17).

[5] P. Baudiš and J.-L. Gailly. *Pachi: State of the Art Open Source Go Program*. 2011 (cit. on p. 22).

[6] M. G. Bellemare et al. “The Arcade Learning Environment: An Evaluation Platform for General Agents”. In: *Journal of Artificial Intelligence Research* 47 (July 2012). DOI: 10.1613/jair.3912 (cit. on p. 22).

[7] R. Boca et al. “Ultra-Flexible Production Systems for Automated Factories”. In: *2016 IEEE International Conference on Automation Science and Engineering* (2016) (cit. on pp. 18, 19).

[8] G. Brockman et al. *OpenAI Gym*. June 2016. URL: http://arxiv.org/abs/1606.01540 (cit. on pp. 2, 21, 22).

[9] A. Carlos et al. “Using a Game Engine for VR Simulations in Evacuation Planning”. In: *IEEE Computer Society* (2008). URL: http://udn. (cit. on pp. 1, 17, 18).

[10] W. Chen et al. “Dynamic Future Net: Diversified Human Motion Generation”. In: Association for Computing Machinery, Inc, Oct. 2020, pp. 2131–2139. ISBN: 9781450379885. DOI: 10.1145/3394171.3413669 (cit. on p. 15).

[11] A. Cohen et al. *On the Use and Misuse of Absorbing States in Multi-agent Reinforcement Learning*. 2022. URL: www.aaai.org (cit. on p. 33).

[12] Y. Duan et al. “Benchmarking Deep Reinforcement Learning for Continuous Control”. In: *33rd International Conference on Machine Learning* (Apr. 2016). URL: http://arxiv.org/abs/1604.06778 (cit. on p. 22).

[13] R. Dunlop. *Production Pipeline Fundamentals For Film and Games*. 2014. ISBN: 0415812291 (cit. on p. 17).

[14] P. Ghadai et al. “A Study on Agent Based Modelling for Traffic Simulation”. In: *International Journal of Computer Science and Information Technologies* (2016). URL: www.ijcsit.com (cit. on pp. 1, 9).

[15] T. Haarnoja et al. *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. 2018. DOI: 10.48550/ARXIV.1801.01290. URL: https://arxiv.org/abs/1801.01290 (cit. on p. 33).

[16] K. Horák and B. Bošansk´bošansk´y. “Solving Partially Observable Stochastic Games with Public Observations”. In: *33rd AAAI Conference on Artificial Intelligence* (2019), p. 19. URL: www.aaai.org (cit. on p. 23).

[17] W. Hu, Z. Wang, and X. Fan. “Contained fluid simulation based on game engine”. In: Institute of Electrical and Electronics Engineers Inc., June 2017, pp. 545–549. ISBN: 9781509055074. DOI: 10.1109/ICIS.2017.7960052 (cit. on pp. 15, 19, 20).

[18] A. Hussein et al. *Imitation learning: A survey of learning methods*. Apr. 2017. DOI: 10.1145/3054912 (cit. on p. 13).

[19] P. Johnson and D. Pettit. *Machinima: The Art and Practice of Virtual Filmmaking*. 2012. ISBN: 0786461713 (cit. on p. 16).

[20] A. Juliani et al. *Unity: A General Platform for Intelligent Agents*. Sept. 2018. URL: http://arxiv.org/abs/1809.02627 (cit. on pp. 2, 15, 25).

[21] Ł. Kaiser et al. “Model Based Reinforcement Learning for Atari”. In: *ICLR 2020* (2020). URL: https://goo.gl/itykP8 (cit. on p. 11).

[22] K. M. Khalil et al. “An Agent-Based Modeling for Pandemic Influenza in Egypt”. In: *Handbook on Decision Making: Vol 2: Risk Management in Decision Making*. Ed. by J. Lu, L. C. Jain, and G. Zhang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 205–218. ISBN: 978-3-642-25755-1. DOI: 10.1007/978-3-642-25755-1_11. URL: https://doi.org/10.1007/978-3-642-25755-1%5C_11 (cit. on p. 2).

[23] F. Klugl and A. L. C. Bazzan. *Agent-Based Modeling and Simulation*. 2012 (cit. on pp. 6, 7).

[24] F. Klügl, M. Fehler, and R. Herrler. “About the role of the environment in multi-agent simulations”. In: vol. 3374. Springer Verlag, 2005, pp. 127–149. DOI: 10.1007/978-3-540-32259-7_7 (cit. on p. 7).

[25] M. Lanctot et al. *OpenSpiel: A Framework for Reinforcement Learning in Games*. Aug. 2019. URL: http://arxiv.org/abs/1908.09453 (cit. on p. 23).

[26] E. Liang et al. "RLlib: Abstractions for Distributed Reinforcement Learning". In: *35th International Conference on Machine Learning* (Dec. 2017). URL: http://arxiv.org/abs/1712.09381 (cit. on p. 23).

[27] P. U. Lima et al. "RoCKIn and the European Robotics League: Building on RoboCup Best Practices to Promote Robot Competitions in Europe". In: *RoboCup 2016: Robot World Cup XX*. Ed. by S. Behnke et al. Cham: Springer International Publishing, 2017, pp. 181–192. ISBN: 978-3-319-68792-6 (cit. on p. 14).

[28] R. D. Lisio et al. *The Convergence of the SPH Method*. 1998, pp. 95–102 (cit. on p. 19).

[29] C. Macal and M. North. "Agent-based modeling and simulation". In: Dec. 2009. DOI: 10.1109/WSC.2009.5429318 (cit. on pp. 1, 2, 6, 8).

[30] S. S. Noh, S. D. Hong, and J. W. Park. "Using a Game Engine Technique to Produce 3D Entertainment Contents". In: *16th International Conference on Artificial Reality and Telexistence–Workshops (ICAT'06)*. 2006, pp. 246–251. DOI: 10.1109/ICAT.2006.139 (cit. on p. 15).

[31] J. Oh et al. *Self-Imitation Learning*. 2018 (cit. on p. 13).

[32] M. Pasternak et al. "Simgen: A Tool for Generating Simulations and Visualizations of Embedded Systems on the Unity Game Engine". In: *MODELS 2018*. MODELS '18. Copenhagen, Denmark: Association for Computing Machinery, 2018, pp. 42–46. ISBN: 9781450359658. DOI: 10.1145/3270112.3270135. URL: https://doi.org/10.1145/3270112.3270135 (cit. on pp. 17, 19).

[33] *Robocup*. URL: https://www.robocup.org/research/ (cit. on p. 14).

[34] M. Samvelyan et al. "The StarCraft Multi-Agent Challenge". In: *33rd Conference on Neural Information Processing Systems* (Feb. 2019). URL: http://arxiv.org/abs/1902.04043 (cit. on p. 23).

[35] J. Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. DOI: 10.48550/ARXIV.1707.06347. URL: https://arxiv.org/abs/1707.06347 (cit. on p. 33).

[36] D. Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *CoRR* abs/1712.01815 (2017). arXiv: 1712.01815. URL: http://arxiv.org/abs/1712.01815 (cit. on p. 10).

[37] R. S. Sutton and A. G. Barto. *Reinforcement Learning An Introduction second edition*. 2018. ISBN: 978-0-262-19398-6 (cit. on p. 21).

[38] D. A. Swayne et al. "Exploring Forest Management Practices Using an Agent-Based Model of Forest Insect Infestations". In: *2010 International Congress on Environmental Modelling and Software Modelling for Environment's Sake* (2010). URL: http://www.iemss.org/iemss2010/index.php?n=Main.Proceedings (cit. on p. 1).

[39] J. K. Terry et al. "PettingZoo: Gym for Multi-Agent Reinforcement Learning". In: *35th Conference on Neural Information Processing Systems* (Sept. 2020). URL: http://arxiv.org/abs/2009.14471 (cit. on pp. 2, 23, 24).

[40] G. Tesauro. "Temporal Difference Learning and TD-Gammon". In: *Commun. ACM* 38.3 (Mar. 1995), pp. 58–68. ISSN: 0001-0782. DOI: 10.1145/203330.203343. URL: https://doi.org/10.1145/203330.203343 (cit. on p. 10).

[41] E. Todorov, T. Erez, and Y. Tassa. "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109 (cit. on p. 22).

[42] *Unity ML-Agents Toolkit*. URL: https://github.com/Unity-Technologies/ml-agents (cit. on p. 26).

[43] S. Vanfossan, C. H. Dagli, and B. Kwasa. "An agent-based approach to artificial stock market modeling". In: vol. 168. Elsevier B.V., 2020, pp. 161–169. DOI: 10.1016/j.procs.2020.02.280 (cit. on p. 1).

[44] T. Wang et al. "Influence-Based Multi-Agent Exploration". In: *ICLR 2020* (2020). URL: https://sites. (cit. on p. 12).

[45] J. Yang et al. "CM3: Cooperative Multi-Goal Multi-Stage Multi-Agent Reinforcement Learning". In: *ICLR 2020* (2020) (cit. on p. 12).

[46] L. Zarco et al. "Scope and delimitation of game engine simulations for ultra-flexible production environments". In: vol. 104. Elsevier B.V., 2021, pp. 792–797. DOI: 10.1016/j.procir.2021.11.133 (cit. on p. 18).

[47] B. Zheng et al. "Imitation Learning: Progress, Taxonomies and Challenges". In: (June 2021). URL: http://arxiv.org/abs/2106.12177 (cit. on p. 13).