Implementing Machine Learning in Video Games to Simulate Subjects

Thesis

Master's in Information, Communication and Multimedia Technologies

---

Candidate: Joshua Lima Damasio

Student ID: A035026

Supervisor: Cláudia Sofia Borlido de Freitas

Cosupervisor: Agostinho Gil Teixeira Lopes,

Alieldin Ayman Ali Mahmoud

Abstract

In every scientific field there is often a gap in what is researched, and the technologies used by relevant industries for their products. With artificial intelligence and the game industry this gap is significant; researchers work on discovering more efficient and dynamic algorithms while the industry requires cost-effectiveness and reliability. The purpose of this study is to analyze the current available technologies and techniques that have been developed by researchers and define the requirements for implementing them in videogames in a cost-effective manner. Specifically, the study will validate the possibility of creating subjects that learn how to behave through Reinforcement Learning and determine its cost-effectiveness and potential in comparison to commonly used alternatives.

Keywords: Artificial Intelligence; Machine Learning; Reinforcement Learning; Implementation in Videogames; Non-Playable Characters; Multiple Model Reinforcement Learning.

Resumo

Em todos os campos científicos costumam haver separações entre aquilo que é pesquisado e as tecnologias utilizadas pelas indústrias relevantes. Com a inteligência artificial e a indústria de videojogos esta separação é significante; pesquisadores tipicamente trabalhem na descoberta de algoritmos mais eficientes enquanto a indústria necessita de custo-efetividade e confiabilidade. Este estudo foi desenvolvido para analisar tecnologias e técnicas atualmente disponíveis que foram desenvolvidos por pesquisadores e definir os requerimentos para a sua implementação em videojogos de uma maneira custo-efetivo. Especificamente, este estudo valida a possibilidade de criar sujeitos que aprendem como comportar-se através do Reinforcement Learning e determina o custo-efetividade e potencial em comparação a alternativas geralmente utilizadas.

Palavras-chaves: Inteligência Artificial; Machine Learning; Reinforcement Learning; Implementação em Videojogos; Personagens Não Jogáveis; Multiple Model Reinforcement Learning.

Table of Contents

List of Figures

Listings

Table of Abbreviations and Acronyms

AI	Artificial Intelligence
CPU	Central Processing Unit
DI	Dependency Inversion, Dependency Injection
FSM	Finite-State Machine
GPU	Graphics Processing Unit
ICM	Intrinsic Curiosity Model
LSTM	Long-Short-Term Memory
ML	Machine Learning
MMORPG	Massively Multiplayer Online Role-Playing Game
MMRL	Multiple Model Reinforcement Learning
NERO	NeuroEvolving Robotic Operatives
NN	Neural Network
NPC	Non-Playable Character
PPO	Proximal-Policy-Optimization
RL	Reinforcement Learning
RPG	Role-Playing Game
rtNEAT	Real-time NeuroEvolution of Augmenting Topologies
SAC	Soft-Actor Critic

1 Introduction

The use of Artificial Intelligence (AI) in everyday technology has been growing exponentially the last couple of decades. This growth has been often limited to academic achievements; the industry is still heavily reliant on older and simpler methods, especially in the field of videogames. The industry's lag in incorporating newer technologies to produce potentially better solutions is one of the motivations for this study. As such it will focus on analyzing the practicability and reliability of implementing machine learning in videogames.

1.1 Context

From basic chat bots to automated vehicles to warehouse packing robots, AI is finding a place in all kinds of technology. There are many benefits to using AI, and with the advancements in computer hardware it can be easily applied to various processes that were once manually managed. One common way for researchers to test their AI algorithms is to see how it performs in playing games, be it board games such as Chess or Go, or videogames such as Super Mario World. There has been great accomplishments through these tests, as they evaluate different aspects of AI, such as Google's Deepmind beating the Go champion (Reynolds, 2017), a game that is virtually impossible to brute-force and requires intuition, or OpenAI that defeated a professional Dota 2 team, a five against five strategy game that requires a high amount of cooperation (Piper, 2019).

1.2 Motivation

The advancements in AI have introduced a lot of promise to various fields, from education to medicine and even automotive. One would expect that these advancements would also enhance the gaming experience – after all, games typically use some form of AI whether it is through aiding in the creation of virtual environments using procedural generation or creating smarter enemies to fight the player.

Unfortunately, the average AI being used for Non-Playable Characters (NPCs) such as enemies in videogames is extremely simplistic. Given the importance of NPCs in videogames, one would expect the game industry to be among the first to implement new AI technologies, however, it has mostly stagnated in this regard. At the best of cases, NPCs use a type of advanced Finite-State Machine (FSM) that may consider features such as personality and the relationships between subjects. However, FSMs use manually programmed behaviours and are not a true AI – there is no learning or discovering in their development.

This stagnation and the consistent lack of intelligent NPCs throughout a variety of games has been felt personally and among other researchers (Gruenwoldt, Katchabaw, & Danton, 2005), and is one of the main motivators for choosing this as the subject for this study.

1.3 Problem and Objectives

Even with many breakthroughs in the field of AI, little has been seen used in the game industry, especially with NPCs. While it holds much promise for creating highly realistic

characters with complex behaviour patterns, it is set aside possibly for a combination of reasons that may include its perceived cost, complexity of implementation, unreliable outcomes, and a lack of experienced developers. Instead, manually programmed behaviours such as FSMs are typically used as they often provide easier and reliable solutions; however, the static nature of FSMs offers limited complexity and can be difficult to maintain in highly interactive virtual environments. Such scenarios are ideal for the use of an advanced AI based on machine learning. This study will focus on the implementation of subjects in virtual environments with behaviours determined entirely by machine learning. Furthermore, it will analyze how modern object-oriented programming and related technologies could assist in the implementation of the AI.

The points below represent all the objectives to be achieved through the project. Each objective will be further detailed in the Project Execution section of this study.

- Reinforcement learning prototype with manually created model
- Basic subject-to-target implementation
- Basic subject with task implementation
- Subject-to-target implementation with threats
- Subject with target and task implementation
- Subject-to-target with environment perception
- Combination of above subjects into a single environment

1.4 Chapters Organization

In the State of the Art chapter a bibliographic study and research is conducted on the current state of artificial intelligence and the challenges that have been identified within both academia and the industry. With the problem properly identified and described, possible solutions can be defined and the requirements necessary to implement them. Modern techniques and technologies are also studied in order to determine their practicality for simulating subject behaviour. In the Workplan chapter we define the project, procedures, and timelines. Each project, divided into scenarios, attempts to solve a different type of problem that may be found in a typical videogame. The details of each of these projects, their implementation, and their relation to the works previously discussed are stated in the Development chapter. The Analysis chapter details the qualitative and quantitative results of every project and additional observations that were not considered or foreseen during the bibliographic study. Finally, this study is summarized and settled in the Conclusion chapter and thereafter continued by the Future Work chapter in which the research and results are taken into consideration to define ideal future prospects of study.

2 State of the Art

This study conducts an overview on the current state of videogames and the factors that influence their design and development, such as the importance of advancements in software development. The problem is identified, defined, and related to the current subject of the study. Having this defined, we discuss the benefits and drawbacks of using ML for simulating NPC behaviour in a variety of types of games and settings. Finally, an analysis is performed on the implementation of modern AI in videogames and the AI that will be used for the development projects of this study.

2.1 Videogames

For thousands of years humans have been playing games, a challenge or competition purposefully made difficult to determine when an individual or group is better at a given task. With advancements in technology, games eventually took on a virtual form called videogames, and like traditional games, typically present either a challenge or competition. Although traditional games could present a physical challenge, intellectual challenge, or a mix of both, videogames almost always focus on the latter. Videogames have introduced a variety of new ways for humans to play games and tell stories from sports-based videogames to interactive worlds.

2.1.1 History

Videogames have been evolving closely together with the advancements in computation. The limitations of that which can be added to a game is often tied with the

computational power of the average player and the developer's budget. However, another contributing factor is the technologies available that can be used for the development of videogames. Today, a single average game developer should be able to make the original game of Super Mario Bros. without much difficulty, not only thanks to the computational power available, but also the technologies used to enable the creation of games. An instance where one can see the benefits of state-of-the-art technology in game development could be in the comparison of Square Enix's Final Fantasy VII and Final Fantasy IX. Both games had a similar development budget of about USD $40 million and were developed for the same console, the main difference is that Final Fantasy IX was released three years after, near the end of the original PlayStation's production (Leone, 2017; Alder, 2000). Even though both games had the same limitations in budget and hardware, Final Fantasy IX presented superior graphics, a far more detailed world and enhanced user experience.

As game's evolved together with computers, it is expected that the subjects found within them would also evolve to become more intelligent and realistic. This is true in some cases when we compare older games with newer ones that are decades apart, but AI development in games has mostly stagnated, at least in comparison to the advancements of AI in academia. Most games still use subjects with entirely programmed behaviours or FSMs. One of Valve's most renowned games, Half-Life 2, introduced a fairly complex approach on FSMs for its NPCs and enemies, allowing the AI to make decisions as a group (Millington, p. 10) and even consider the relationship between each subject (Valve Developer Community, 2019). It was a pioneer in AI for its time, however, the action for every behaviour was still manually defined and programmed, it only added additional criterias in the AI's decision tree. Another company, Bethesda, implemented a sophisticated AI for

NPCs in one of their flagship open-world videogames, The Elder Scrolls V: Skyrim (Skyrim), and achieved incredible results. Over time however, bugs with their AI started to become known among its player-base and even exploited. For instance, players discovered that if you covered an NPCs head with a pot or basket, thus covering their "eyes", they could not see you and would not react to anything you did, such as stealing (Dobra, 2011). The NPC's AI, which was most likely based on FSMs, was not designed for a situation where it had something blocking its vision. In other cases, the NPC had an extreme lack of situational awareness and did not take into consideration the severity of the events happening around it. This created for some unrealistic and dramatic situations, for example, if the player chose to assist guards defending a town and ended up accidentally destroying town property or harming an ally, the guards, oblivious to what was happening, would consider the player an enemy and attack them. These types of unrealistic scenarios ended up reducing the game's immersion. The sophisticated AI worked as a double-edged sword for the game, while incredibly realistic for its time it also emphasized the flaws of its design (Bedingfield, 2019).

One of the more common uses of AI in videogames is for procedural generation, such as for allowing the AI to generate the designs of "levels" or platforms, locations, creatures, or entire planets on its own. Procedural generation has drawn a lot of interest in the industry for good reason: it is easy to define limitations and provides an easy way to quickly make large and unique virtual environments. The advancements of this technology have introduced an increasingly popular genre of videogames that are known as sandboxes. Sandboxes are videogames with highly interactive virtual environments that can often be completely changed by its users and subjects, hence the naming "sand-box". While the concept has been

around for decades, creating a fully interactive environment was not possible only until recent years, mostly due to the lack of available processing power.

Many games, such as Minecraft, Starbound, RimWorld, and No Man's Sky, to name a few, have all been released in the past decade and provide highly interactive virtual worlds. However, to create these games, the developers had to make cuts on other features. For example, the initial versions of Minecraft contained no NPCs, and the ones that exist in the game today are not impressive when it comes to its AI. No Man's Sky had a highly anticipated release only to disappoint many players on what they thought would be a highly complex universe, but instead found the equivalent to a series of superficial planets and animals with a sophisticated reskin (Heather, 2016).

While many currently available sandbox games cannot implement sophisticated AI without going through an entire redesign, we could use their example as reference to understand how ML could have improved at least the subjects found within them. With Starbound and No Man's Sky, the subjects in their game would vary planet to planet. In Starbound's initial versions for example, the design was quite simple: Several templates were designed for each part of the animal, and some parts would be dependant on the animal's nature – wings would only apply to animals that fly, for instance. When the world was generated, along with its animals, randomization would be used to determine what template would be used for each part. This method allowed for the creation of hundreds of apparently unique animals, but only superficially. The animals' behaviours were highly generic and had no relation to the environment they found themselves in. This did not go unnoted as Chuckefish's director, Finn Brice, mentioned "*one of the lessons we learned really early on is that you can add as many variables as you'd like to a procedural generation algorithm,*

and you don't get any closer to a satisfying experience unless you add context and meaning." (Haske, 2016). Adding meaning and contextual information to a subject's behaviour is one of the greatest challenges for procedurally generated subjects. Chucklefish eventually improved their algorithms to include this information.

RimWorld differentiates a bit from the other games mentioned in this section as the player micro-manages a colony and does not have explicit control over a single character. While the AI is impressive, and includes a variety of personality traits, its subjects lack contextual awareness. For instance, colony members may choose to walk through a shoot-out or raid to gather resources instead of staying in a safe location or choosing a safer route; likewise, animals will not avoid grazing near predators or other kinds of threats.

The next big leap in creating realistic sand-box games might not be through the improvement of graphics, larger virtual worlds, or deeper levels of interaction between player and environment, but, instead, possibly through highly realistic animals and subjects that emerge, interact, and behave according to their environment. A game where animals will move from one location to another for better grazing or move in herds and packs to increase their chances of survival; where NPCs will leave the city to gather resources, adventure, discover things for themselves, or give meaningful tasks to players to achieve things they cannot on their own. ML provides a means to achieve these realistic subjects that would be otherwise impractical and costly through FSMs.

2.1.2 The Gap

In many games, such as massively multiplayer online role-playing games (MMORPGs), NPCs are fixed in a single location and do not react to players. It is

understandable - games such as Skyrim require high amounts of investment and are not dependent on constant updates like MMORPGs. Not all companies have the budget to create and maintain a game with complex AI or see its economic advantages, which is one of the reasons the gap exists between the AI used in games and those currently available. Georgios Yanakakis noted about this gap stating, "*there is a difference in what is valued, with academics valuing new algorithms and new uses of algorithms … and AI developers in industry valuing software architectures and algorithmic modifications that reliably support specific game designs.*" (Yannakakis & Julian, 2018, p. 14) Part of this divide may be due to the different backgrounds dominating the academia and game design: Currently many of the researchers that work in the field of AI, especially machine learning, come from a statistics or mathematics background, and although they use computers and programming languages for developing their models, they do not necessarily need to understand software architecture. This is not an issue for most business applications that need to use machine learning as it is far easier to adapt software applications to accommodate for new technologies. For games, however, it is a bit more complicated as software architecture holds a far more important role in the game's design. Due to this, the use of advanced AI needs to be part of the game's entire development process.

While the game industry has many uses for AI in developing games, FSMs are still the most common method in creating a simplistic AI for a subject or NPC. The use of FSMs allows one to create simple and reliable AIs; for this reason, it is a popular choice when creating NPCs that serve as the player's enemy or for subjects that will be used for a short moment. The downside of using state machines is its inflexibility and simplicity as they will only iterate through their predefined states. The behaviour becomes unpredictable for

scenarios that it was not designed for. Furthermore, FSMs with greater complexity require increased maintenance making it an impractical choice for environments that may suffer a lot of variation or for subjects that hold an important role (Merrick, 2008).

2.2 Artificial Intelligence

The definition that this study will use for artificial intelligence as defined by the author (Millington, 2019, p. 4): "*Making computers able to perform the thinking tasks that humans and animals are capable of.*" The definition is clear enough to show that we do not wish to focus on pre-defined knowledge of the solution. For example, how does one know that solving a simple equation, something that computers do regularly, is not a thinking task? Typically, it is only possible to solve an equation once its rules are understood and identified. One may know how addition, subtraction, division, and other mathematical operations works, and putting these operations together they discover the solution to the equation. This is not considered a thinking task as it is simply applying rules to a given input to determine the output. Now, instead, if there are multiple equations that one must solve, each using a variety of unknown variables and rules, but only the output is known, how could they be solved? One approach is to use statistics and probability to determine the values of each variable and the rules: The more equations one is given, the more probability they will have to determine the correct values and operations. This process of discovering the value of each variable and the meaning of each operation is what would be considered the "learning" of some areas of artificial intelligence. When the machine understands the meaning of each part

of an input and their importance after processing through large amounts of data, it could determine the correct output of new data with significantly high accuracy.

2.2.1 Traditional techniques

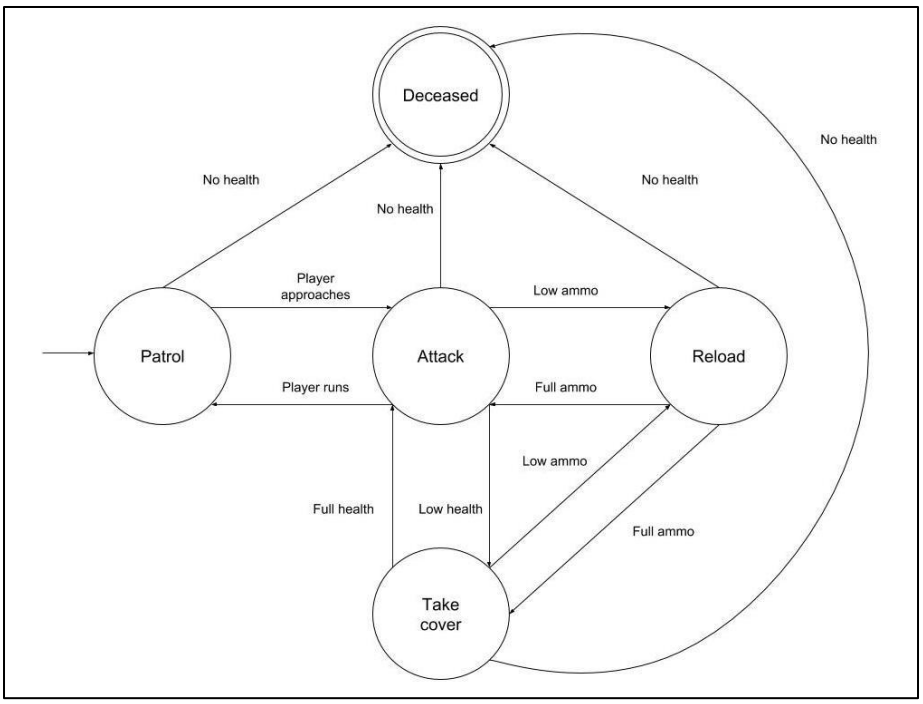


Figure 2.1. Example of a finite state machine. From "Theory of Computation: Finite State Machines", by Marcus Sanatan, 2019 (https://stackabuse.com/theory-of-computation-finite-state-machines/)

2.2.1.1 Finite-State Machines

FSMs are essentially a recursive decision tree. Conditions are defined for each possible scenario that the subject may encounter itself in. When the condition is met, the subject will execute a series of actions and possibly iterate through another series of conditions. The conditions also have a priority and can override less urgent states that may be in the process of execution. For instance, the FSM in Figure 2.1 will only activate the

"Attack" state if the NPC has full ammo, full health, and is approached by a player. If the NPC runs out of ammo, the "low ammo" condition is triggered and the "Reload" state is activated, overriding its current "Attack" state.

2.2.1.2 Planning Scripts

Planning Scripts were introduced in the game F.E.A.R. and combine the use of FSMs and the A* algorithm, an algorithm typically used for pathing that finds the result with least cost between two points. In this case, a goal and a number of actions are provided to the AI, and the A* algorithm calculates the sequence of actions to achieve the goal with the least cost. It introduces more reliability by removing state validation from static actions to an objective-based algorithm. (Orkin, 2006)

2.2.2 Neural Networks

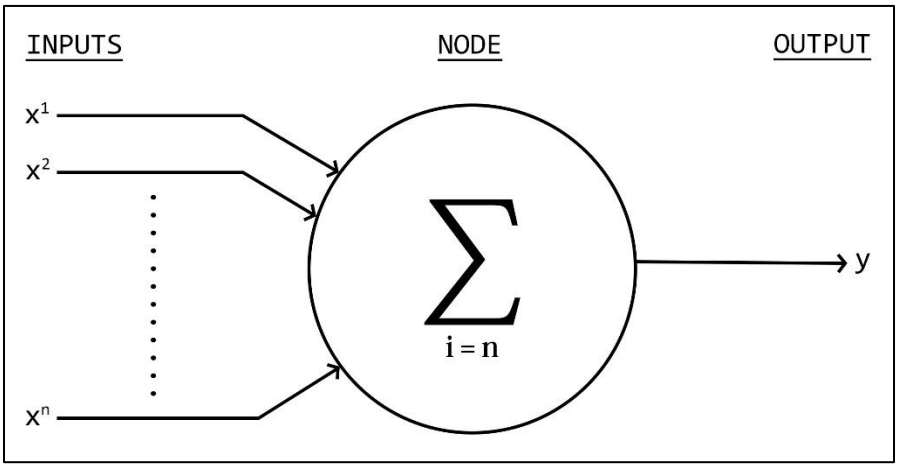


Figure 2.2. Example of a simple artificial neuron. Adapted from "*A Comprehensive Study of Artificial Neural Networks*" by V. Sharma, S. Rai and A. Dev, 2012, *International Journal of Advanced Research in Computer Science and Software Engineering,* 2, p. 279

Neural networks (NNs) are commonly used when designing models based on Machine Learning. A neural network comprises a series of artificial neurons that each calculate an output based on a given input. Each neuron can evaluate a different part of the input and is attributed a weight that defines its importance in the final decision of the network. When a neural network is processing data, it is typically not possible to determine what part of the input neuron is evaluating, this is what is defined as the "black box" problem (Benítez, Castro, & Requena, 1997). Neural networks that use a basic activation function such as sigmoid provide an output for single-class identification and are common for binary problems. The softmax activation function can be used for non-binary problems as they allow the network to have multiple outputs.

2.2.3 Machine Learning

Machine Learning (ML) is a subcategory of Artificial Intelligence that uses Bayesian statistics and algorithms to enable the machine to learn the best probable solution for given problems (Millington, 2019). The neural network's configuration, such as the number of nodes and layers in the network, and its learning rate, are known as hyperparameters. Tuning the hyperparameters is an important part of model training as they should reflect the complexity of the given problem. Incorrect hyperparameter tuning can cause long training sessions or unreliable results.

Within the field of ML there are three main subcategories, each ideal for solving different kinds of problems: Supervised learning, unsupervised learning, and reinforcement learning. The first two subcategories require a large amount of preprocessed data for the model to receive acceptable results. The data is sent as an input to the model, the model then

processes the data and provides an output, also known as a prediction. When the model provides an incorrect prediction, it modifies the algorithms weights and biases and tries again. This process will continue until it runs out of data or until it reaches an acceptable accuracy defined by the developer. The difference between supervised and unsupervised learning is how the model discovers and labels information. Supervised learning models will attempt to associate the data with a given output, but unsupervised learning models will only attempt to find important information or patterns within the data.

The other subcategory is the one this study focuses on: Reinforcement learning. It will be detailed in the next section.

2.2.4 Reinforcement Learning

Reinforcement Learning (RL) is a subcategory of ML together with supervised and unsupervised learning and differs from them in that it uses a virtual subject, known as an agent, and does not require preprocessed data to train its model. Instead, the data, or input, is contextual information of the environment that the agent finds itself in. The amount and specificity of the data can be determined by the developer based on the problem that is being solved. Typically, to ensure that the model does not become bias, many trials are attempted with a varying environment. Varying the elements of an environment helps the model determine what information is important, especially when these elements are associated with the agent's reward scheme. Every agent holds a reward scheme that is used to evaluate its actions. This reward scheme can vary in complexity depending on the nature of the problem. Throughout these trials a behavioural policy is updated based on the best performing agents, which is a strategy that determines how the agent should act and how much variation there

should be in its actions for each new trial (Merrick, 2008). A reward signal can be given to the agent when it reaches certain objectives, such as its proximity to a point in space, or when it reaches a final goal, such as the point in space it needs to move itself towards. The agent updates its behavioural policy based on the reward signal received. To further add, it is important for a reward scheme to provide both positive and negative reward feedback and vary the values in a meaningful way. For instance, minor objectives should provide a fraction of the reward that would be given when the agent fully resolves the problem; penalties should be given based on their perceived impact. Using a proper reward scheme can greatly decrease the time it takes to train the agent.

2.2.5 Multiple Model Reinforcement Learning

A promising technique that may be efficient and practical in the field of video game AI is a variation of multiple model reinforcement learning (MMRL). The main concept is that every agent holds multiple models, each model being an expert system responsible for a specific task or environment. A controller and predictor are assigned for each model; the controller is responsible for executing the model and retrieving its result while the predictor determines the model's loss for the given problem. When calculating the ideal model to be used, the predictors and controllers are weighted by a softmax function known as the "responsibility signal". The result of the responsibility signal is determined by the predictor with least error and activates the associated model (Doya, Samejima, Katagiri, & Kawato, 2002). Proper implementation of such technique could increase the flexibility of the AI and significantly reduce training times as it moves from a large single-model approach to multiple

small highly expert systems that can be reused in various scenarios and throughout numerous agents.

2.2.6 Machine Learning in Games

As mentioned, ML in games, especially for defining NPC behaviour, is still not widely used if at all. However, there have been a couple notable releases in the past few decades, such as in 2001, Lionhead Studios developed Black & White, a game where you play god over a village and a large creature. The game implemented a mix of RL and other AI technologies to simulate the behaviour of the creature based on how the player interacted with it. The player had the choice to penalize or reward the creature for actions that it performed. The player's choices for given actions would be memorized and processed through RL (Evans, 2000), altering the creatures behaviour based on those choices. Another studio, Uber Entertainment, developed a game known as Planetary Annihilation where the main enemy AI uses artificial neural networks to decide the best actions to defeat the player. Furthermore, the AI also learns and improves for every game it plays (Jurney, Robbins, & Sunshine-Hill, 2012).

2.2.7 Behaviour in Artificial Intelligence

Currently, it is difficult to find a single, universal definition for behaviour, as this is studied in a variety of different fields such as neuroscience, psychology, and sociology, using different methods and observations. However, the definition by Gpts Hemakumura and Rainis Rusian (2018, p. 93) seemed to fit well for the intention of this project: "*Human behaviour refers to the range of behaviours exhibited by humans and that is typically*

influenced by culture, attitudes, emotions, values, ethics, authority, persuasion, coercion and/or genetics." Expanding on this definition and for the purpose of this project, the following would most likely be more accurate: "*Human behaviour would be best described as the range of actions and reactions exhibited by humans, typically influenced by the environment, culture, attitudes, emotions, values, ethics, authority, persuasion, coercion and/or genetics*." The project of this study will not attempt to simulate highly complex social behaviours between subjects or groups of subjects, such as those that are affected by culture, persuasion, and authority. The focus will be on the actions and decisions of the subjects and how they are influenced by the presence of other objects and subjects found in their environment. This study will consider behaviour as connecting "where" the subject finds itself with "what" the subject does and/or is, to explain the reason for its decision.

Behaviour in NPCs typically only involve the range of actions that they can take – the reasoning behind their actions is simply to meet the needs of their developer. The NPC will only do as it was programmed. RL will not allow us to enable the computer to think and understand the reasoning behind its actions, but it may at least provide us with more flexibility and more realistic subjects. Such results have already been achieved by Teemu Heinimaki and Juha-Matti Vanhaputa, where they developed an advanced AI machine that contained parameters to represent human personality traits such as bravery and authority in their work *Implementing artificial intelligence: a generic approach with software support* (2013, p. 35).

2.2.8 Unity ML-Agents

The Unity ML-Agents framework, as of version 1.0.2, comes ready-to-use with two Deep RL algorithms: Proximal-Policy-Optimization (PPO) and Soft Actor-Critic (SAC). These are baseline algorithms that can solve a wide range of problems but may need to be further developed and customized if one wishes to use them in a practical application such as a game. The algorithms can be extended with the Intrinsic Curiosity Model (ICM) and a Long-Short-Term Memory (LSTM). The ICM introduces the attribute of curiosity or exploration to the behavioral policy and the LSTM allows the model to hold information of previous iterations, working as a "memory" for the agent. The algorithm and extensions can all be enabled and managed through the model's hyperparameters (Juliani, et al., 2018).

2.3 Benefits of Simulating Subjects with Machine Learning

As mentioned earlier, certain videogames, particularly those that are strictly online, use NPCs and other game subjects mostly for decorative purposes or for highly basic interaction between player-NPC, such as NPCs responsible for selling goods. These games typically suffer a lot of variation over time as their players expect continuously updated and new content from its developers. It is with these games that implementing sophisticated AI is the most challenging yet could be the most beneficial. The main issue faced is that among all the game's updates, developers need to be assured that their NPCs will behave exactly as they expect them to even with the new changes to their environment. This is what leads most to use a basic or no AI implementation during development. However, by using abstraction and game design appropriated for RL, we could implement an AI that would behave as

initially designed even with updates and changes to its environment. Additional training would only ever be necessary when we add new behaviour to our subject.

2.3.1 Reliability

Proper programming allows for reliable subjects whether using FSMs or ML for your AI. However, foreseeing future requirements could be very difficult and is what usually leads to code maintenance and alterations. This lack of foresight could affect FSMs especially, as one would need to know of all the scenarios that the subject might find itself in. As a simple example, consider a developer creates a state known as the attack state. It is a simple state: if the subject sees an enemy the subject is in range, it will attack the enemy. With proper implementation, the state should work without issues for most cases. However, it needs to address for scenarios with added complexity. If multiple enemies are seen by the subject, the state needs additional logic to take appropriate actions – at this point it is not known how the subject will behave until this is defined. When designing FSMs, all these considerations must be considered, and even further if certain NPCs will behave differently for the same state.

Using ML forces developers, at least slightly, to take approaches that would create a flexible and reliable agent, as they must consider what information is sent to the agent during its development process and how to train it effectively without bias, considering a variety of scenarios. Since we explicitly define and limit this information, the agent will never process a different amount of information and should never find itself in a state, that is, a combination of inputs, completely different than what it encountered during its training process. By limiting the input information and ensuring comprehensive training sessions, it is possible to create reliable agents even for scenarios with high degrees of complexity variation.

2.3.2 Cost

The cost of using ML could potentially be lower than manually programmed alternatives such as FSM and Planning Scripts. This is due to the workload of creating the subject's decision tree being diverted to the agent rather than the developer. In other words, the cost is moved from manual programming to CPU calculations. Manually programmed alternatives require developers to design the actions and states for every NPC with an AI; adding new actions or NPCs increases the workload and complexity significantly (Orkin, 2006). With ML, the behaviour of the subject is defined during the training process. Furthermore, because traditional states found in the finite-states are not anymore necessary for defining behaviour, they can instead be used for creating triggerable events and animations, if beneficial. With ML solutions, much of the work is diverted into the creation of effective training scenarios and reward schemes. Development and training costs could be even further reduced with the implementation of MMRL.

2.3.3 Real-time Adaptation

Some games have already introduced the use of ML to scale the difficulty based on the player's performance (Tan, Tan, & Tay, 2011), or even improve the AI's behaviour in real-time. An example of the latter can be seen in a project developed by a group of researchers from the University of Texas, NeuroEvolving Robotic Operatives (NERO), a game intended for both academia and industry leaders to develop RL algorithms. In NERO, the player must train soldiers to reach a goal using an effective reward scheme (Gold, 2005). In another paper, Kenneth et al. introduce a powerful ML method known as Real-time

NeuroEvolution of Augmenting Topologies (rtNEAT). The rtNEAT allows for agents to dynamically change and improve within the game (Stanley, Bryant, & Miikulainen, 2009).

This sort of adaptation could also be seen with “companion” NPCs – subjects who accompany the player throughout the game’s story. With ML we are given the possibility of enabling various behaviours based on the environment that the companion finds itself, or even the possibility of the companion “learning” and progressing in skill together with the player. Perhaps with advancements in ML we could achieve NPCs that would hold personality traits that greatly influence their actions, such as creating events of betrayal if they’re opportunistic or loyalty if they’re brave and committed; such events could have more significance as they would not be manually defined by the developers and could present different results for every playthrough. Furthermore, companion NPCs could strategize and interact based on the decisions of the players, for instance, choosing to fight with a ranged weapon if the player chooses to fight with a melee weapon.

2.4 Implementing Machine Learning in Games

When implementing ML in games we believe it is important to use state-of-the-art software engineering practices to create cost-effective and reliable solutions. Properly structured video games intentionally designed for ML can easily create and use diverse models that can be used in a variety of situations that would be otherwise preprogrammed.

2.4.1 Abstraction

The use of abstraction in programming can and likely should be used for the implementation of ML in videogames or simulations. The main idea of abstraction is to create a base class that will be “inherited” by multiple child classes. The base class can be a virtual class, an abstract class, or an interface. A virtual class is a regular class that allows some of its functions to be overwritten by its child classes. An abstract class and interface are both abstract in nature and cannot be instantiated, however, abstract classes can implement behaviour while interfaces can only define behaviours that must be implemented by child classes. All classes that are not an abstract class or interface are known as concrete classes as they are responsible for implementing functionality. The use of abstraction in programming brings many benefits, such as reducing the amount of development and maintenance required as many functions can be reutilized or allowing us to reference the parent object instead of the child object or its implementation. This sort of referencing is part of a design pattern known as dependency injection which will be discussed in the next section.

With regards to application and game design there are a variety of ways design patterns could be used and implemented. We will use a simple example and demonstrate how one could use abstraction to represent a variety of animals in an application. A house cat and a tiger are two very different creatures, however, they both fall under the feline species, which falls under the mammal and finally animal category. If we were to represent a house cat and a tiger as classes, they would be two separate concrete classes inheriting from the same abstract parent class, feline. The feline abstract class may have some predefined properties that are shared by all felines, such as having fur and four legs, but may require additional

definitions unique to each one, such as the characteristics of its fur, its sex, and its age, that are defined by the concrete class. Furthermore, classes could have a combination of inherited abstract classes and interfaces, allowing for added behaviour and definition. The tiger class could inherit from the feline abstract class and an interface that defines behaviour associated to wild animals. You cannot instantiate the feline, mammal, or animal classes, the same way you cannot create a “feline”, but you can use them as a requirement for your application’s logic.

Within RL, there are three main components to consider: the model, the agent, and the target. All three of these components could benefit from the use of abstraction. For instance, imagine we wish to create an agent that will represent an animal, such as a Cat in a game.

```
public interface IAgent
{
      int HealthPoints { get; set; }
      Vector3 Position { get; set; }

      void DoAction();
      void AddReward();
      ...
}

// IAnimalAgent inherits IAgent
public interface IAnimalAgent : IAgent
{
      bool IsPredator { get; }
      ...
}

// Cat class implements IAnimalAgent and its parent IAgent
public class Cat : IAnimalAgent
{
      public int HealthPoints { get; set; }
      public Vector3 Position { get; set; }
      public bool IsPredator { get; private set; }

      private IModel Model { get; set; }
      private ITarget Target { get; set; }

      public Cat(IModel model, ITarget target)
      {
            Model = model;
            Target = target;
      }

      public void DoAction()
      {
            ...
      }

      public void AddReward()
      {
            ...
      }
}
```

Listing 2.1. Example implementation of Cat agent class using C#

First, we will create the necessary components as abstract classes or interfaces and define their properties and behaviors. Second, we create the animal class, the Cat class, which will inherit and implement the Agent abstract class. In order to instantiate a Cat class, we

must provide it a model and a target, defined as constructor parameters. Finally, in the context of video games, we associate the Cat class to a virtual object in the video game environment.

Every one of these components will have predefined behaviors that must be implemented by concrete classes. The model interface will have a function declared to perform a calculation on the given inputs and return an output, for our example we will call this function GetAction. As this will be the main function for every model, we are certain every inheriting concrete class will need to implement this function. With the agent, we may create an interface and declare a vague function such as DoAction that will be used differently by each concrete agent. We could also instead extend the Agent class by creating another abstract class or interface called the Animal Agent class which will be inherited only by animals. For a panda, the Action function may vary between eating, sleeping, and roaming, but for a lion, the Action function may vary between hunting, attacking, eating, and sleeping. These actions will usually only be defined by the concrete class and depend on the output of the model (Toftedahl, 2019). For the target, we might only want to hold the information of its position and declare that as a property. In the case of the panda agent, its target would be bamboo. The target may need to hold an additional property called IsConsumed that determines if it were consumed or not, essentially serving as its health points. The lion's target would be a gazelle, and thanks to object-oriented programming, we can implement multiple interfaces in a single class, allowing us to make the gazelle both an Agent and a Target and implement the behaviors for each interface.

2.4.2 Dependency Injection

As mentioned above, through abstraction we can use abstract classes as the requirements for a class's construction and define what concrete implementation to use at run-time from higher-level modules. This means when we design concrete classes such as the Cat class above, we do not have to be concerned with the model and target classes used as long as they implement the respective interfaces. This approach is part of what is known as Dependency Injection (DI), as we are *injecting* dependencies from high-level modules to low-level modules. Using DI comes with many benefits, but one of the major purposes for this context is the ease of changing and using multiple different implementations of a single interface. As the higher-level modules are no longer dependant on the concrete requirements of lower-level modules, we can provide different implementations for different needs without having to modify the code of dependant classes.

In the context of game development, most game editors provide a means of flexibly adding and replacing dependable classes to virtual objects. This is somewhat a form of dependency injection. In Unity, these dependable classes, including the Cat class above, are known as scripts and can be associated to virtual objects known as prefabs. Prefabs represent an object in the game world and can hold any number of scripts. As such this allows developers to easily replace dependant classes without having to modify code. The Cat class would need to be slightly altered to allow the model and target variables to accept scripts through the editor. The Cat class would then become as the following:

```
// Example of Cat class in Unity
public class Cat : IAnimalAgent

{
      public int HealthPoints { get; set; }
      public Vector3 Position { get; set; }
      public bool IsPredator { get; private set; }

      // SerializeField tag allows scripts to be associated
      // through the game editor
      [SerializeField] private IModel model;
      [SerializeField] private ITarget target;

      public void DoAction()
      {
            ...
      }

      ...
}
```

Listing 2.2. Example implementation of Cat class in Unity

One can create several Cat prefabs each holding a different combination of models and targets and associate the prefabs to different environments or scenes. These scenes are independent of the prefabs associated to them as they are not responsible for their instantiation, instead this responsibility is moved to the game engine.

2.4.3 Abstraction with Unity ML-Agents

We can use abstraction through Unity's ML-Agents as their framework is open-source and based off the concept of learning agents and RL. There is a bit of difficulty in modifying and extending the models as they are based on Python and connected to Unity using wrappers. Through the hyperparameters, it is possible to easily switch model algorithms; Unity ML-Agents comes with two algorithms out-of-the-box which are the PPO algorithm that uses approximation to find the best choice in each state or SAC algorithm that

focuses more on model updating. By default the model uses a feedforward neural network but if our agent requires the use of memory, we may easily enable the use of recurrent neural networks through the hyperparameters (Juliani, et al., 2018). The scope of this thesis focuses mainly on the implementation of ML in subjects and as such the baseline algorithms and models provided by Unity ML-Agents will suffice.

To create a training environment, its only necessary to implement an agent and a target, although this makes it necessary for the agent to manage information related to the environment and target. Instead, it is best to create a separate class that will manage environment objects and possibly the target as well. This class is then responsible for adding variation between sessions, such as moving obstacles and targets to random locations, and for managing goals and requirements if any are given. This way, the agent is independent from the environment and its changes, allowing developers to reliably use it in different scenarios.

3 Workplan

In this chapter we will review the methodology used for gathering information on the related technologies and methods necessary for implementing RL into agents in a virtual environment. An overview is given regarding the types of virtual environments and scenarios that will be validated. Finally, we detail how the project will achieve the results and the general timeline of this study's development.

3.1 Methodology

The methodologies used for this project were bibliographic research and practice-based research. For bibliographic research we mostly used books, articles, and other studies regarding the implementation of AI in games, game development, and different RL models. These materials were across different fields, such as computer science and neuroscience, to ensure that various aspects of AI were considered. The samples used for comparing with the results of the simulations are based on chosen subjects in videogames currently or previously sold on the market. These videogames include World of Warcraft developed by Blizzard, a dated yet important game that pioneered the MMORPG genre, The Elder Scrolls V: Skyrim, which introduced sophisticated AI for its time, and Monster Hunter: World, an action role-playing game developed by Capcom that introduced a variety of behaviours and interactions to different monsters and animals. The practice-based research is to define the technologies and techniques that are ideal for simulating the behaviour of subjects in video games. The simulation's success is based on the complexity of the model's implementation in a virtual

environment, the amount of resources required to train it, and its reliability to complete the given task.

3.2 Procedures

The project uses Unity's ML-Agents framework to develop simple generic scenarios that can be applied to a variety of different virtual environments. It is important to note that the environments are stationary as time is not a factor in the training process, however, the elements found within them change for every iteration to reduce the bias of the model. The purpose of these scenarios is to create an interactive virtual world that is manipulated by either an animal-like or human-like subject and observe how they behave to achieve their goals.

3.3 Project Execution

Before having begun the implementation of any sort of agent, a prototype was developed with the purpose of understanding how to design and implement RL in a virtual environment. The prototype only needed to use a basic feedforward neural network and act in a 2D space; it did not use any premade framework. This was enough to gain a basic understanding of where bugs and inefficiencies could occur when implementing models in different environments and using different frameworks.

To tackle a variety of different problems, both in complexity and nature, three environments were designed. The first environment is aimed to represent basic animals that demonstrate no advanced intelligence. The goal of the animal in this environment is to search

for food while avoiding dangers. We hope that this will demonstrate the effectiveness of using RL for creating subjects with simplistic and predictable behaviour and that have a minimal amount of spatial awareness.

With the second environment we focus on subjects with tasks or jobs whose problems may vary in requirement. We decided to create an environment where the subject will have to navigate and interact with objects in the area to gather a resource and bring them back to a designated location. In this environment it is important that the agent does not have information about the number of targets, the type of resource its gathering, or the requirements for completing the goal. The focus is to create a simple and flexible agent that can be used for a variety of task-based problems.

Finally, in the last unique environment we simulate agents that require a high amount of precision and dynamic comprehension of their surroundings. For our scenario, the agent is tasked with “pushing” an object to a designated location. The difficulty of the task depends on the shape and size of the object and the distance of the goal, which changes for each training session. The agent will also find obstacles between itself and the goal that it must learn how to maneuver through.

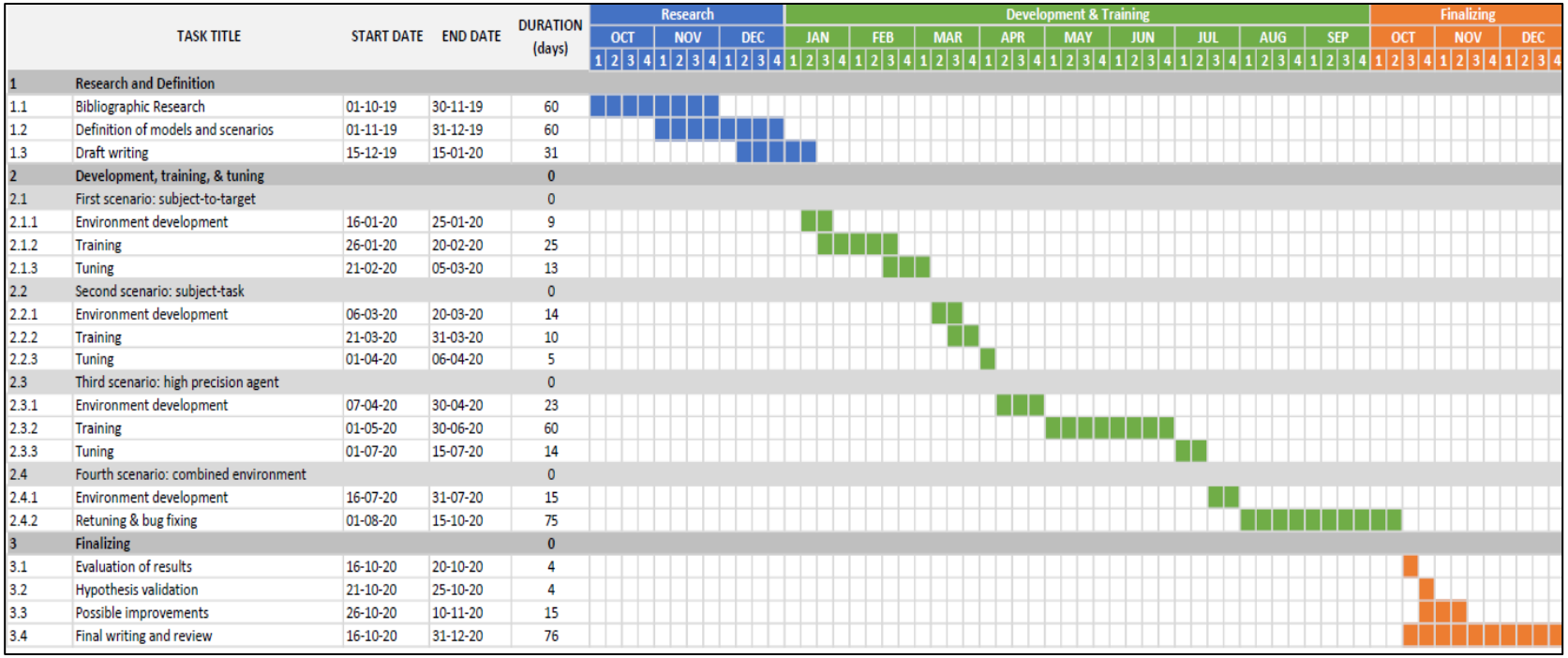
After having achieved satisfactory results in all these scenarios, we combined them into a single environment to create a sort of “ecosystem”. The agents do not interact with each other - they behave as if they were alone in their training environment. However, doing this allows us to demonstrate the reliability of the model when it finds itself in highly differing environments.

Table 3.1. Technologies, software, and their versions

Name	Version
Microsoft Visual Studio Community 2019	16.1.5
Unity	2019.2.0f1
Unity ML-Agents	1.0.2
Python	3.6.5
TensorFlow	2.0.0
C#	7.0
.NET	4.8

In Table 3.1 are defined the technologies, software, and their related versions used for the development of this study's project. The only important note is that during the writing of this thesis, Unity ML-Agents is in its alpha development phase and as such only offers primitive functionality. Unity and Visual Studio were used for the creation of the virtual environments, Unity ML-Agents which is based on Python and TensorFlow was used for training the agents.

3.4 Gantt



	TASK TITLE	START DATE	END DATE	DURATION (days)
1	Research and Definition			
1.1	Bibliographic Research	01-10-19	30-11-19	60
1.2	Definition of models and scenarios	01-11-19	31-12-19	60
1.3	Draft writing	15-12-19	15-01-20	31
2	Development, training, & tuning			0
2.1	First scenario: subject-to-target			0
2.1.1	Environment development	16-01-20	25-01-20	9
2.1.2	Training	26-01-20	20-02-20	25
2.1.3	Tuning	21-02-20	05-03-20	13
2.2	Second scenario: subject-task			0
2.2.1	Environment development	06-03-20	20-03-20	14
2.2.2	Training	21-03-20	31-03-20	10
2.2.3	Tuning	01-04-20	06-04-20	5
2.3	Third scenario: high precision agent			0
2.3.1	Environment development	07-04-20	30-04-20	23
2.3.2	Training	01-05-20	30-06-20	60
2.3.3	Tuning	01-07-20	15-07-20	14
2.4	Fourth scenario: combined environment			0
2.4.1	Environment development	16-07-20	31-07-20	15
2.4.2	Retuning & bug fixing	01-08-20	15-10-20	75
3	Finalizing			0
3.1	Evaluation of results	16-10-20	20-10-20	4
3.2	Hypothesis validation	21-10-20	25-10-20	4
3.3	Possible improvements	26-10-20	10-11-20	15
3.4	Final writing and review	16-10-20	31-12-20	76

Figure 3.1. Gantt chart

The above Gantt chart shows the timeline of the study and project development. Most of the time was spent on the actual project development, nearly half of the total duration. Aside from learning how to develop environments using Unity and implementing Unity ML-Agents, the retuning and bug fixing in the fourth scenario contributed greatly to the amount of time needed to finalize development as this often led to the need for retraining agents in their original environments.

4 Development

In every scenario the basic architecture contains an agent class, a target class, a manager class, and a goal class where applicable. The agent, target and goal all handle their respective virtual objects within their limits, while the manager class handles environment resetting and communicates relevant information between the target, goal, and agent. Keeping the design abstract and similar throughout all scenarios was a necessity to properly implement the agents in the final Ecosystem scenario.

It is important to note that environments and models may have been extremely simplified as the usage of these projects are specific only for this study.

4.1 Basic Animal Scenario

The first scenario represents an animal in search for food. We assume that the animal is herbivorous. The animal must approach the source of food and interact with it to consume it. Only after the target is completely consumed the agent is rewarded. For added complexity, the environment has a basic enemy to represent a predator that slowly follows the agent. If the predator reaches the agent, the agent “dies”, receives a penalty and the trial is reset. The enemy is only added for additional complexity and creating a more realistic environment. The presented problem demonstrates the effectiveness of using RL in simplistic subjects with minimal spatial awareness.

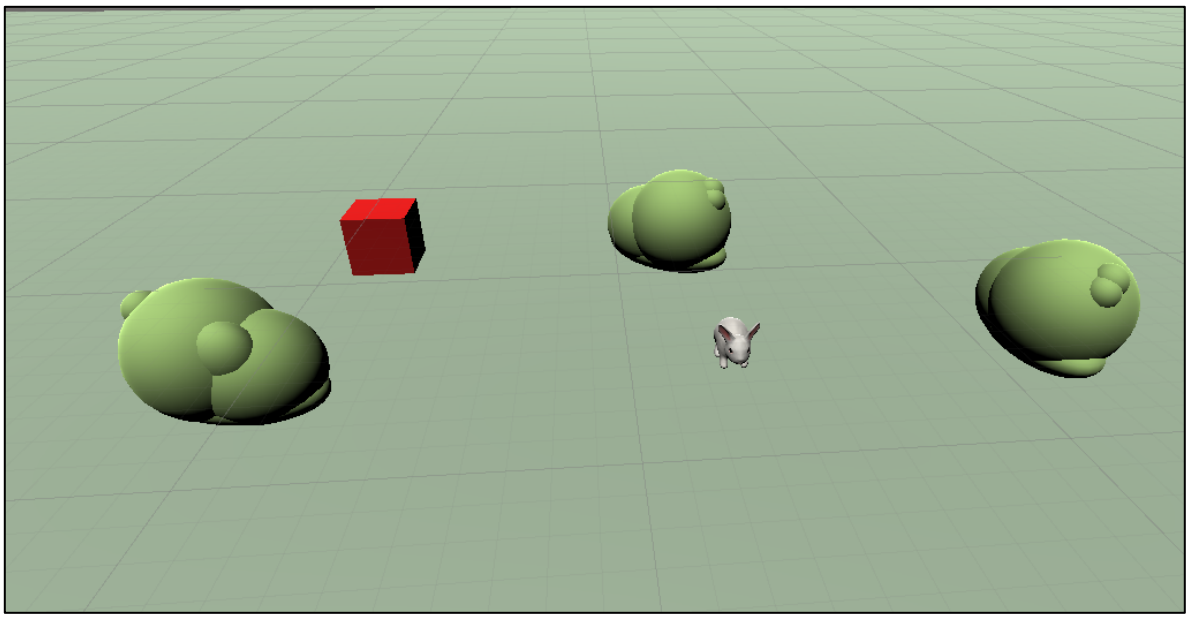


Figure 4.1. Image of animal training scenario

To simulate a realistic environment, we added several sources of food within the scene. One of the challenges faced with this is how to define the inputs when the number of sources can vary. The first solution was to add a limit to the number of sources that the agent can process, for instance, three sources at most. This solution requires the targets' positions and an additional Boolean value to determine if that target exists or not, as there may be less than three sources in the environment. This approach relays the decision making to the model, and consequentially drastically increases the training time. We chose not to take this approach as it adds unpredictability to the model's behaviour. The second solution that we implemented is to provide a single target's information to the input variables and change the target in real-time. In this approach some of the decision making is defined by the developer and can be made different for every agent.

Either of the solutions above defined are viable approaches to model design and may be implemented in a variety of ways to meet the needs of the developer. As our study is the

implementation of RL in video games, we believe at this point in time reliability still holds more value than the variance that can be achieved using the first approach.

4.2 Resource Collector Scenario

For the second scenario we wanted to focus on NPCs that have a larger task with multiple objectives. For our problem, we decided to create an agent that needs to gather resources to build a structure, such as a house. The agent must move to one of the resources it requires, collect it, and then return it to a structure. In this scenario the agent finds itself in two main states: the first state when it has nothing and must gather a resource, and the second state when it holds a resource and must carry it to the designated location.

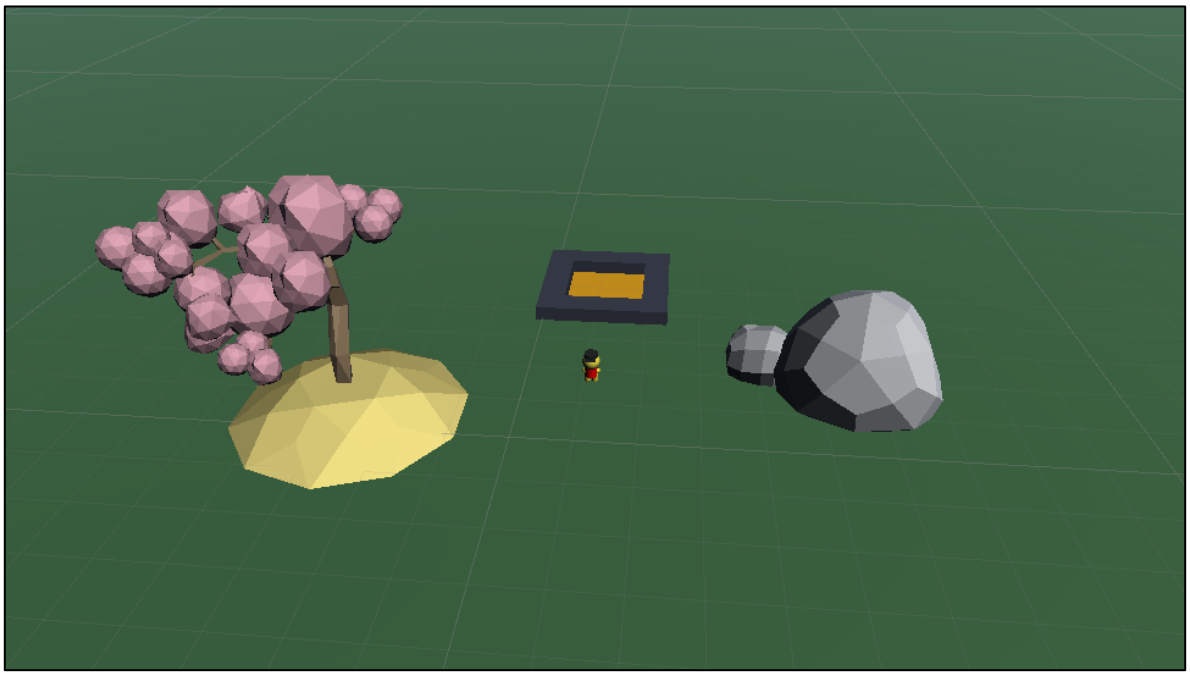


Figure 4.2. Image of resource collector training scenario

Initially, the agent was given information about all the resources required and their amounts. However, it was later realized that this was not the most efficient way to designing

the agent and that it does not need this information. Instead, this information is managed by the manager class; the agent is given information of its next target in real-time. Additionally, we use a basic implementation of finite-states to manage actions and animations from the agent.

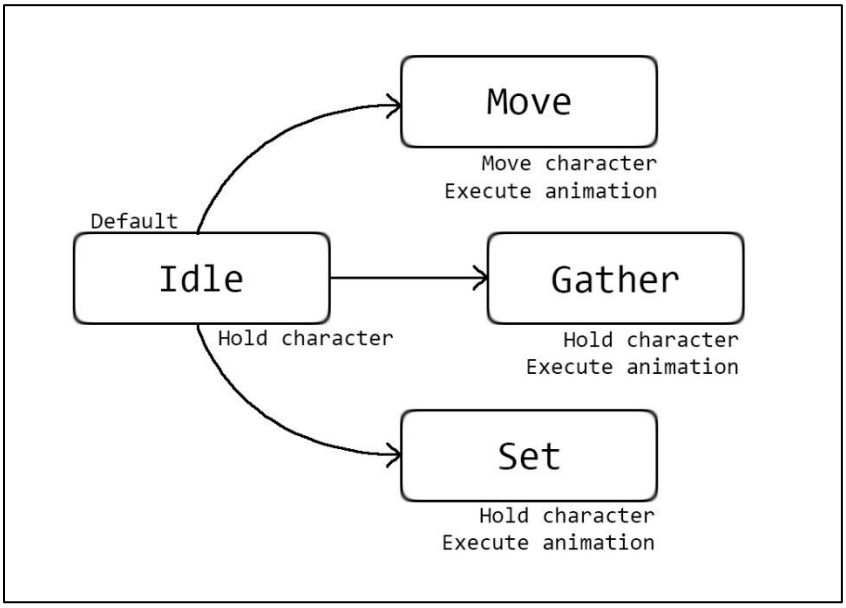


Figure 4.3. FSM integrated with RL, actions, and animations

These states are triggered depending on the model's prediction and environment conditions, rather than predefined states. The agent will default to the idle state if it does not move or interacts with another object. It can only enter the gathering state when it approaches the defined target, furthermore, the gathering state automatically locks the agent in position and executes the action and necessary animations.

The agent solved the problem effectively within a reasonable number of steps and presented very few failing trials after being fully trained. Additionally, adding new and different targets for the agent or increasing the number of resources required did not affect

its performance. As such, it is possible that this same model could be used for several different applications due to the abstraction of the inputs.

4.3 Hauler Scenario

The third and last unique agent scenario was focused on agents with qualitative and high precision tasks. We chose to simulate an agent that must “push” a target object to a designated goal. Furthermore, we wanted it to perceive the scene using “Raycasts”, a method of retrieving object information by casting a sensor in a specified direction and distance, so that it can learn to reliably maneuver in a variety of different environments. To add complexity many obstacles of varying sizes and shapes were added to the scene. The agent not only has to find a path through the obstacles, but also must ensure that the target object will fit in the various gaps towards the goal.

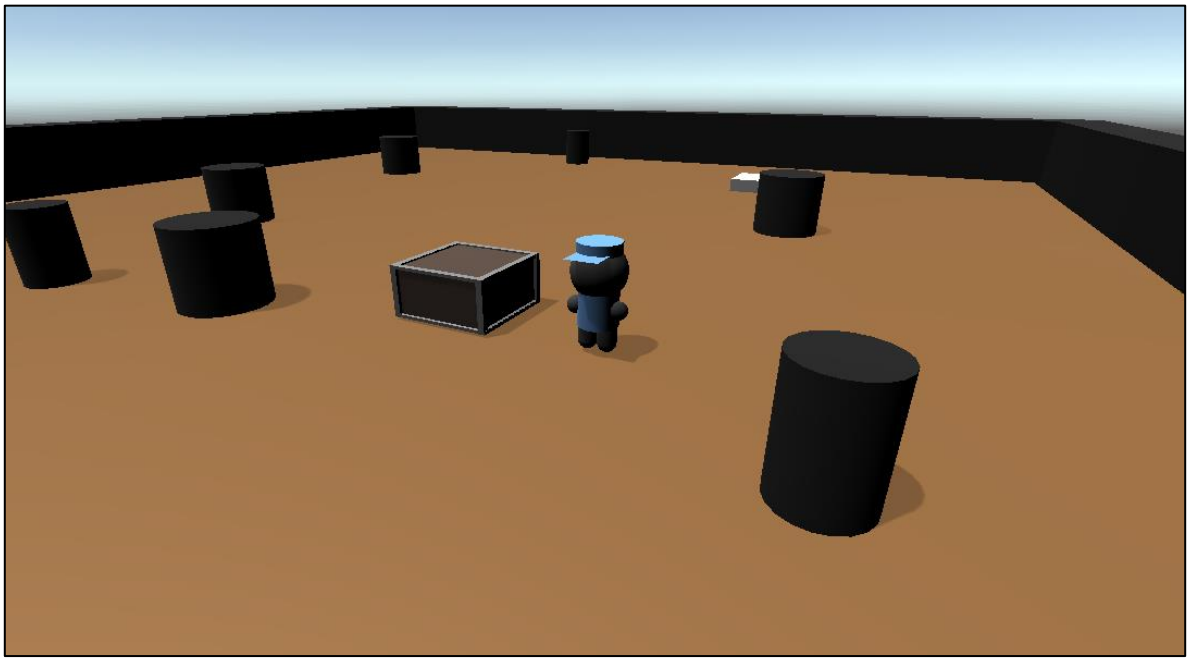


Figure 4.4. Image of hauler training scenario

Due to the nature of this scenario, the costs of training would be much higher than the previous two. For this reason, we initially thought it would be better to train the agent in a static environment. However, using a dynamic environment with varying difficulties happened to shorten the learning process. In any case, as expected, the cost of training was still tens of times greater than our previous environments. Much of this is due to the use of raycasting and the increased amount of inputs necessary. In this scenario, we have obstacles in varying shapes and sizes, so it's important for the agent to not only know if one of its rays hit an obstacle, but also the properties, such as the position and dimensions, of the obstacle that was hit. For each ray used, the amount of inputs must be increased by seven, one for informing if the ray hit an obstacle, and six for each property.

Even after training the agent for about 20 to 30 million steps, we find that it does not effectively solve the problem, obtaining a success rate of about 60 percent. This limitation could be due to the complexity of the problem for the current base algorithms being used for the model's training. Another possibility is the improper configuration of the hyperparameters. With the results achieved it was not possible to effectively integrate the hauler agent into the ecosystem environment, thus it was decided to abandon further training of this model and focus on the remaining two.

4.4 Ecosystem Scenario

The Ecosystem scenario is the final and most important of this study. It serves as a tool to qualify the reliability and flexibility of the models developed and trained in the previous scenarios. The development of this scenario serves also as an integration test, as it

may reveal bugs and issues in our models when implemented in a different environment. Furthermore, it requires a software architecture that enables and manages three different agents and environments. For this scenario to work effectively, it was necessary to implement and improve on some of the designs used in the previous scenarios. The hauler scenario was also considered as part of the design even though abandoned and would likely work with a properly trained model.

One of the main design decisions was giving better definitions to goals and targets to be closer to things that may be found in video games. We defined targets that provide resources as a "source", such as a food source, water source, stone source, etc. Sources always contain a collection of the resource they provide and may hold more than one type of resource. Generics, an abstract design concept, was used to simplify the creation of sources. Goals were defined as a "structure" as they typically represent some sort of structure or building in the environment and may have material requirements. Like sources, they also use Generics to simplify the creation process and hold a collection of resources. However, they can only consume resources rather than provide them.

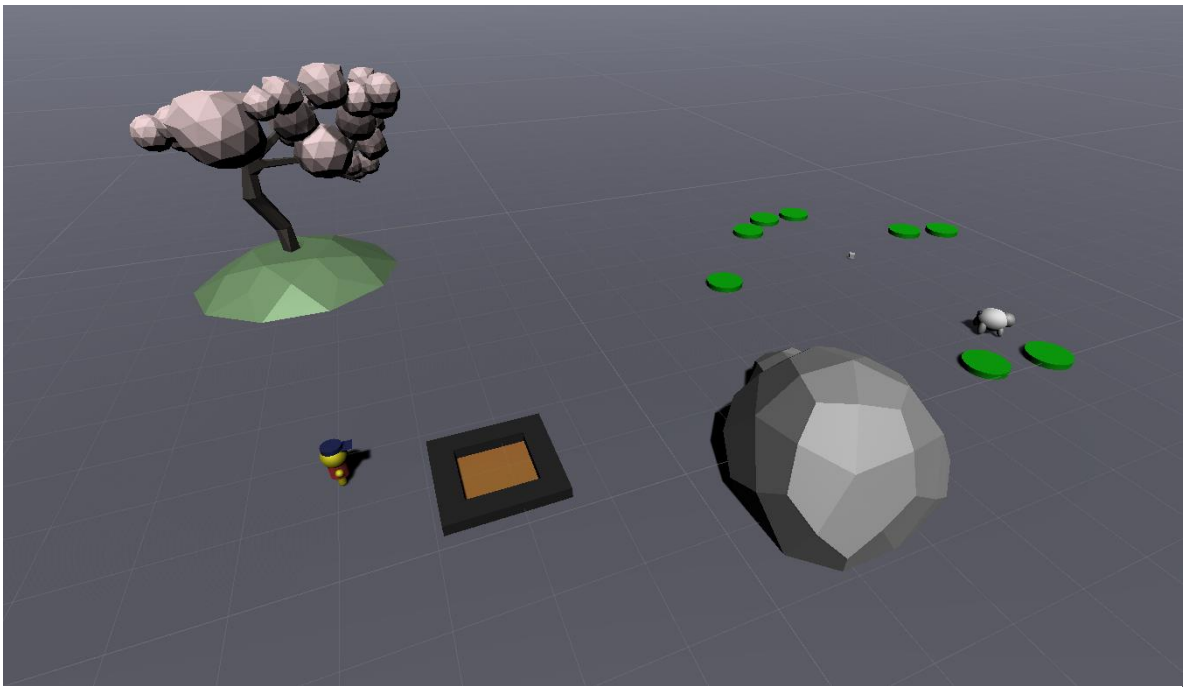


Figure 4.5. Image of ecosystem environment

Another design decision was allocating information on available targets and goals to the EnvironmentManager class. While this was done in some of the previous environments, for the Ecosystem it was necessary to ensure that the EnvironmentManager only provided targets and goals related to the agent. To accomplish this, the targets, goals, and agents are all associated through the inspector to the game object prefab that holds the EnvironmentManager class. This way it is possible to divide the environment into parts as required. This allows us to create very different scenarios and agents all within the same scene.

Various challenges were encountered when integrating the pretrained models to the Ecosystem's environment. Most of these challenges were related to inconsistencies between the training and Ecosystem environments, such as the scales and positions, which introduced data that the pretrained models could not process. This resulted in a highly degraded

performance by the agents. To work around these limitations, it was necessary to simplify the model's inputs or retrain them in a more normalized training environment, or both. In industry video game development, it is likely these issues will not be encountered as scenes are typically normalized in size and scale throughout the entire game.

5 Analysis

In our analysis it is discussed the performance of our projects, observations defined during development, and the final results. The performance is compared qualitatively to current market videogames. For this comparison, we chose three videogames with distinctly different genres and that use NPCs differently, to ensure that our results are compared for a variety of different uses. In the section after, observations and limitations are discussed in regard to this project's development and implementation in production videogames. Finally, with the results of our project we take a more quantitative approach and observe the duration of the training and model characteristics for each successful scenario.

5.1 Performance

When comparing between RL and traditional AI methods it is important keep in mind the benefits of using RL in video games, such as the diversion of development costs to processing time, that cannot be achieved using traditional methods, and the potential and continuous improvements of ML that can be used throughout the development of a video game.

While World of Warcraft may be a dated videogame it is still relevant in the industry and uses AI in a variety of different ways. As with most MMORPGs, the majority of its NPCs are fixed at a location and may have little to no interaction with the player, such as city guards and shopkeepers. A number of NPCs move around and "interact" with their environment, such as a guard captain or a travelling merchant, but are typically limited and have no

interaction with the player – these NPCs are only for creating the illusion of a realistic virtual world. For some specific quests, NPCs may follow and even join the player in combat, but the NPC has no other special behaviour. Finally, most of the monsters and animals in World of Warcraft have no special AI. While it's not possible to know exactly how they were designed, it is likely that they have a designated location in the world where they will "spawn", and then move around in circles until interacted by a player. World of Warcraft was a pioneer in the MMORPG genre and as such, many of its features, including the levels of AI found with their subjects, can be found in a number of other different games within the same genre. The results achieved from our study are superior to what is currently used in this MMORPG, and with proper implementation and game design, RL could be a better alternative to traditional methods.

As already mentioned, Skyrim introduced an advanced AI for its time, along with its disadvantages. Much of the realistic behaviours and interactions from NPCs are manually programmed, which can include how they behave based on their location, time of day, and who they are interacting with. Such behaviours would be highly challenging to implement using RL and would likely require the use of auxiliary or hybrid systems to create a more simplified and robust overall AI. However, these advanced techniques were not implemented for animal subjects or simple monsters. Typically, these are set in a specific environment, for example fish being located in rivers and ponds, and given basic instructions, such as fleeing when harmed. They have no special interaction with their environment, and as such, have a simpler AI than what was developed in our study.

Monster Hunter: World introduces somewhat the opposite of what can be found in Skyrim. Instead of highly realistic NPCs, it introduces highly realistic natural environments

and beasts that all interact with each other. The game has a variety of different environments each with their own unique creatures and plants. Even without player influence, creatures may interact with other creatures, such as larger beasts hunting and consuming smaller prey, or they may interact with the environment, such as by consuming plants. However, even though the AI is fairly advanced, it is important to note that each environment is fairly small and contained, with a limited number of creatures, as compared to games such as Skyrim which have a very large and vast single world. Due to this, it is easier to develop elaborate environments and creatures as they are smaller in size and numbers, respectively.

5.2 Observations

During the development of the project various observations were made related to limitations of current technologies, difficulties with implementation, and practical development improvements that reduced training times and/or improved the model's flexibility.

5.2.1 Data association

Machine learning in general faces an issue with associating input data. All observations of the environment are typically bundled together into a single array and then passed through the model. The model must create an association on its own through training which leads to increased training times. While this limitation is beyond the scope of our study, it would have made the agent's learning process a lot faster if it were possible to associate data to their virtual objects. For instance, in our first scenario, it would have been

beneficial to associate the enemy's velocity and position to the enemy itself. This way when processing those values, the model would understand that those values have no relation to any other object aside from the enemy, decreasing its training time.

5.2.2 Input Simplification

When designing an agent one can decide how much information that agent should be responsible for managing. The more information it is required to process will drastically increase the training time but also reduce its flexibility when it finds itself in different environments or with different requirements. For instance, imagine one creates an agent whose task is to build a house. It could simply be given the materials and their required amounts for building the house, and each associated target that holds the materials it needs.

```
// total of 8 observations
AddObservation(target 1 position [vector])
AddObservation(target 2 position [vector])
...
AddObservation(material 1 amount [integer])
AddObservation(material 2 amount [integer])
...
```

Listing 5.1. Example of bad model inputs

The problem with this approach is that the model will base its calculations on the number of targets and the number of required materials needed during its training. Changing any of these parameters, such as adding an additional material or target, would require retraining of the agent. Instead, it is more efficient to divert the responsibility of the material and target requirements to another class. This class would then send updated information of the targets to the agent in real-time and on-demand; at this point, one can add basic logic in the agent class for deciding the next target. Using this method, the agent would only need a

single target, and does not need any information related to the task's requirements as this can also be managed externally.

```
// example of simplified model inputs
// total of 4 observations
AddObservation(current target position [vector])
AddObservation(materials required? [boolean])
...
```

Listing 5.2. Example of simplified model inputs

This simplification creates more flexibility as the number of targets and materials are no longer input parameters for the model, thus changing them does not affect the algorithm in any way. The result of this is that we can freely change the target and its behaviours without requiring any retraining whatsoever; the agent focuses solely on completing a task rather than understanding its requirements.

5.2.3 Bias or Model Inflexibility

When training the model, it is essential to guarantee a wide range of possible scenarios that might be encountered in production. This means that the input values have to vary, possibly significantly, in order to train the model reliably. Not doing so will create a model that is biased to a fixed range of input values and may cause unexpected behavior when introduced into a production environment. We faced this issue when we initially tried to integrate our animal model to the ecosystem, as we had used a different range of values for the targets' positions between both environments. Nonetheless the ecosystem environment proved its usefulness exposing flaws in our pretrained models.

5.2.4 Abstraction with Unity

While developing the various scenarios within Unity it was found that the framework has some limitations regarding abstraction. These limitations affected the time it took to develop scenarios and a proper architecture for implementing the agents, but only after finding acceptable workarounds. One of the main limitations is the lack of support for interfaces with Unity's game objects. When referencing a game object, only the concrete or abstract class is returned. It is necessary to manually convert and validate if a game object implements an interface before you can reference that interface directly. The main issue with this fact is that most of Unity's supported functions use game objects, so it is more cost effective to instead use abstract classes.

5.2.5 Multiple Agent Environments

In all the scenarios, we wanted to create environments that had multiple interacting agents. This is necessary if one wishes to simulate social interactions such as group exploration or herding, hierarchies, communities, relationships, and a variety of other social contexts. Adding information relating to other agents is probably one of the greater challenges in AI as models must have a fixed number of inputs and the number of related agents depends on the social context that one wishes to simulate. A simple approach is to add a fixed number of agents to the model's inputs – with animals, we would only add the related agent's position but task-based agents might require the target and goal of the related agent as well. One must also determine which related agents will be provided to the model.

```
foreach (var relatedAgent in relatedAgentsInRange)
{
      if (relatedAgent is valid)
      {
            AddObservation(relatedAgent position);
            AddObservation(true);
      }
      else
      {
            AddObservation(default position);
            AddObservation(false);
      }
}
```

Listing 5.3. Example of inputs for related agents in a multi-agent environment

For each related agent, the model would require at least four additional inputs – a vector of three values to represent the position, and a Boolean value to determine if the position references a valid agent. This approach becomes costly as one increases the number of related agents but may be ideal for small social contexts with complex interactions. If one wishes to simulate swarm or mob behaviour, however, the above approach would be far from ideal. Instead, it would likely be most cost-efficient and realistic to use a single model that will determine the objective of every agent. The targets and goals for every agent would be updated but their actions will be determined by the individual model.

5.2.6 Multiple Model Reinforcement Learning

It was not possible for us to implement this technique in the given timeframe; it is a method we believe would be very promising in result and efficiency. In the field of game design, this technique might be most suitable when combined with FSMs. Each state, instead of having manually programmed actions, would have a pretrained “submodel”. A main

decision model would hold information regarding the state of the agent, the environment, and other information related to its task. The decision model's main role is to calculate the responsibility signal and provide the necessary information to the active submodel. The submodels would be highly efficient and require less information processing than agents with singular-based models as they hold highly generic and simplistic tasks. Furthermore, due to the simplicity of the submodels, they could be used throughout a variety of environments and agents without the need of retraining. With this sort of application, most of the training required would be with the decision models and their different combinations of submodels. Multiple agents with similar roles or differing complexity could be created at a fraction of the cost. Finally, it may also be possible to stack multiple decision models. For example, the main decision model may activate the "move" model, which could also be a decision model that evaluates whether the agent should run or walk and activates their respective submodels. One could easily see the benefits that this could bring to games that use a lot of static or basic NPCs.

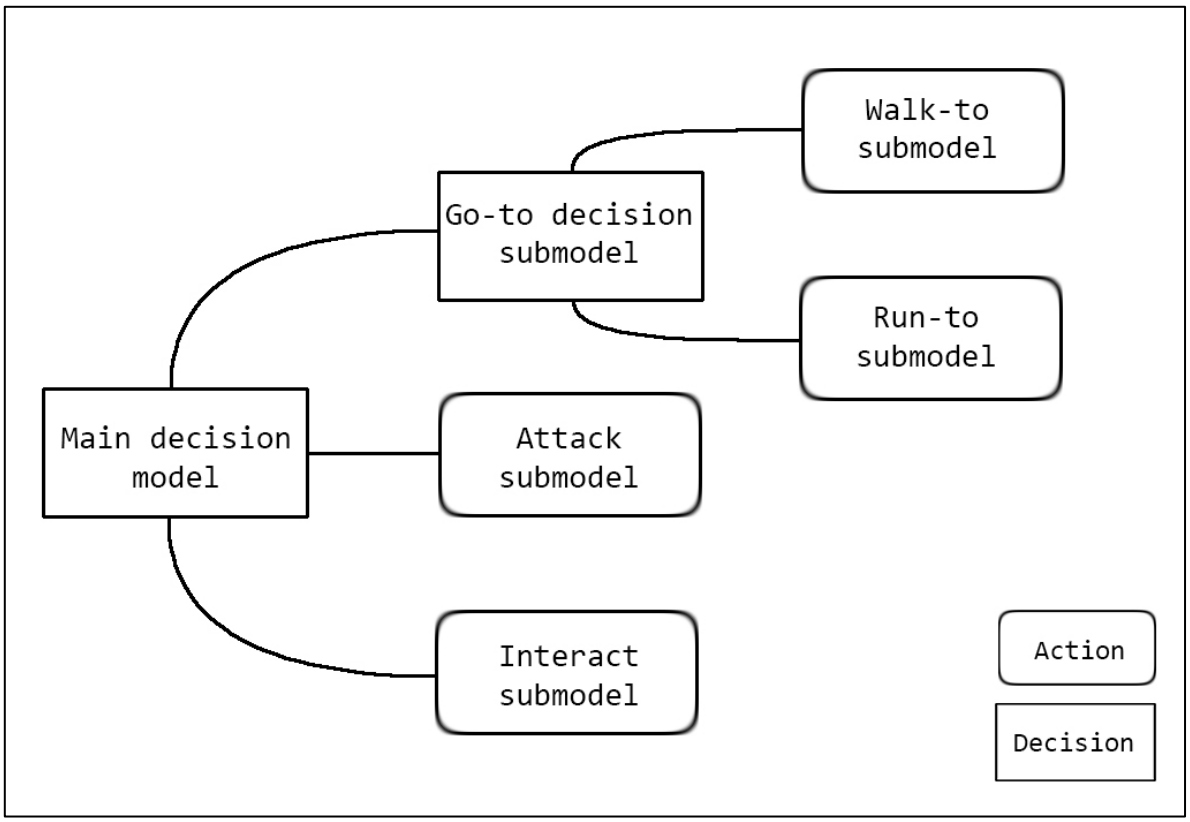


Figure 5.1. Example of using MMRL in a FSM structure

Using this technique with the simulated scenarios, each one would have roughly one or two submodels: one responsible for moving the agent to a specific point in space, and another for executing an action or interacting with an object. The submodel responsible for moving the agent could be the same for the first and second scenario, although not ideal as the inputs and problems are not identical.

5.3 Results

Each scenario took a different amount of time to train as they faced problems of different complexity. The animal scenario was the fastest to train even though its model had more inputs and outputs than the collector. The hauler scenario, which was abandoned,

required roughly 20 million steps, or 10 times the training time of the collector, to produce mediocre results, and never solved the problem effectively.

Both scenarios used the PPO trainer algorithm and a 2-layer neural network architecture. Each agent trained with 32 nodes per layer. Considering that our hardware was nowhere near ideal, training times for agents of this complexity could be reduced to minutes and possibly even seconds when using appropriated servers or cloud computing alternatives. Furthermore, we did not use some of the tools provided by Unity ML-Agents to decrease training times such as curriculum learning and pre-recorded demonstrations.

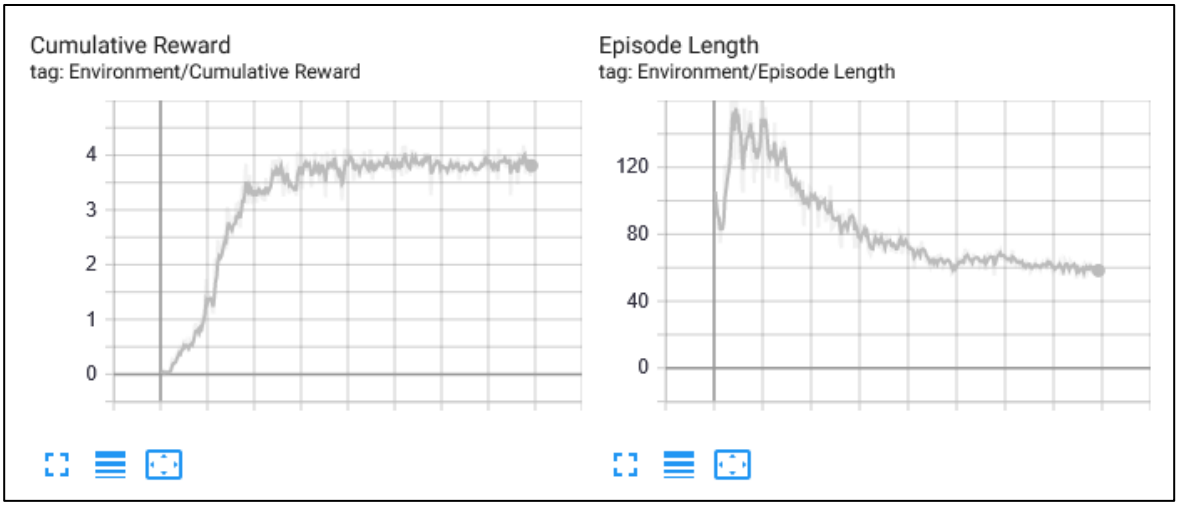


Figure 5.2. Training graph of animal agent

The animal agent used a total of 14 inputs and had 3 outputs – two outputs for moving along the X and Z axis, and one output for interacting with food sources. The agent was able to solve the problem quickly, at roughly 500 thousand steps, and further increased performance at around 1 million steps whereafter it had no significant improvement. The maximum achievable score for this training is 4.5 - 1.5 for every plant consumed and three total plants per iteration. With an average score of roughly 3.8, the animal agent achieved a

success rate of roughly 84%. This scenario had a training duration of about 1 hour and 20 minutes to achieve maximum performance.

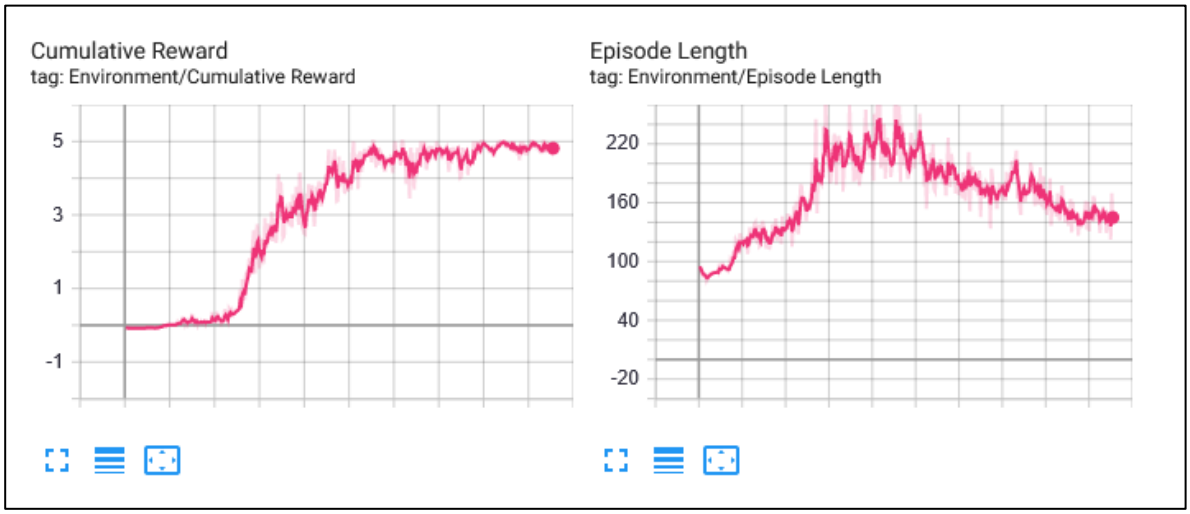


Figure 5.3. Training graph of collector agent

As can be seen in Figure 5.3, the collector agent learned the problem with moderate performance around 1.1 million steps but improved slightly by around 1.7 million whereafter it did not produce significant performance changes, resulting in training times of 1 hour and 50 minutes and 2 hours and 40 minutes respectively. For this problem only 11 inputs were needed. The agent used two inputs instead of three as we chose to not make it a requirement for it to explicitly interact with the objects, only approach them. The maximum performance score for this training was 5.0; the agent averaged around 4.8 achieving a success rate of 96%.

Both agents performed their tasks successfully when integrated into the ecosystem environment without requiring any retraining.

6 Conclusion

With the advancements in AI and computer processing power it would be expected that games would begin to implement highly intelligent and interactive NPCs that use some sort of advanced AI such as RL, and not the current ones found today that are often based on FSMs. While the reasoning for this is hard to justify, it could be pointed towards reasons such as the perceived costs, complexity of implementation, and/or unreliability of the current technologies.

Several games have introduced unique and advanced techniques using FSMs and Planning Scripts. The game Half-Life 2 contained a sort of relational hierarchy between NPCs only using FSMs, while the game F.E.A.R. used Planning Scripts to create dynamic enemies. Both of these games introduced state-of-the-art AI for their time but come with limitations. Manually programmed AI will never be scalable as the entire cost of development is on the developer. Furthermore, integrating advanced social interactions, such as a relationship system, with dynamic states such as Planning Scripts, introduces an extremely high amount of complexity that must be constantly accounted for throughout the game's lifecycle. These limitations create a need for dynamic, flexible, and scalable AI whose effectiveness does not fully rely on developers but instead algorithms.

One of the potential alternatives is ML, an AI approach that finds solutions using algorithms over large amounts of data. With ML, and specifically RL, a subset of ML that does not require a large, predefined data set, we could create complex yet reliable AI for video games. Among advancements in ML, the MMRL approach offers a lot of promise to applying RL in industry video games by dividing decisions and actions throughout an array

of models. This approach, when applied properly, could increase the performance, reliability and flexibility of agents compared to traditional ML and FSMs.

For this study we created a variety of environments, each with a different problem for agents to solve. One of our planned environments had to be abandoned due to hardware limitations, however, the results of the remainder showed promise for implementing ML in video games. The two remaining environments provide a fair amount of complexity and take only a few hours to fully train using a low-end consumer PC. Furthermore, the agents proved to be flexible when changing their task requirements and integrating them into an entirely new environment as they did not require any retraining. Typical video game producers will have teams of developers to help design and create unique models, agents, and environments, and have access to powerful servers. It is likely they would benefit from optimized models and far superior processing performance. As such, the use of RL in video games should be a highly feasible possibility.

Considering the limitations of common AI techniques of FSM and Planning Scripts, and the advancements achieved in recent years in processing performance and advanced AI such as ML, the use of these newer technologies for creating more reliable and sophisticated subjects in video games is most definitely a practical option. Of these advancements with ML, we believe MMRL has the most benefit and potential in its application for simulating subject behaviour. The use of MMRL, or a variant of this, will most likely be necessary for applying ML in large-scale video games and for simulating highly complex individual and social behaviours.

7 Future Work

The study conducted introduced various new technologies that could be further researched and tried that were not previously known to us. Reviewing over the study, we can identify aspects that could have been improved, completed, or used.

One of the initial ideas that was not approached in the study was implementing multiple agent scenarios. Each scenario was to have a phase where multiple agents would be trained to observe their behaviour. To effectively train multiple agents, it would require a new model trainer with a design taking this into consideration, as the rewards would be divided between the agents. This idea was eventually discarded due to the complexity of designing a new model trainer and the increased training times this would introduce.

Returning to this study one of the first challenges we would like to overcome is completing a scenario similar to the hauler, a problem that requires high precision and dynamic perception of the environment. Considering our acquired experience for making these environments, we could explore other technologies and of solely using Unity and Unity ML-Agents.

Finally, as already mentioned several times within the study, we would like to explore the potential benefits of using MMRL in these implementations. The use of MMRL for implementing NPCs in videogames would likely be the main topic of any future works that we would develop, especially considering that it could be easily compared with the results found in this study.

References

Alder, D. (2000, July 29). *NTV Tokyo Interviews Hironobu Sakaguchi*. Retrieved from SquareHaven: http://squarehaven.com/news/2000/07/29/NTV-Tokyo-Interviews-Hironobu-Sakaguchi/

Bedingfield, W. (2019, April 20). *Bad video game AI makes for hilarious surrealist comedy skits*. Retrieved from Wired: https://www.wired.co.uk/article/video-games-surrealism-bethesda

Benítez, J. M., Castro, J. L., & Requena, I. (1997). Are artificial neural networks black boxes? *IEEE Transactions on neural networks, 8.5*, 1156-1164.

Dobra, A. (2011, November 11). *Fan-Made Skyrim Videos Show Off Major Glitches, Stealing Exploit*. Retrieved from Softpedia News: https://news.softpedia.com/news/Fan-Made-Skyrim-Videos-Show-Off-Major-Glitches-Stealing-Exploit-233915.shtml

Doya, K., Samejima, K., Katagiri, K.-i., & Kawato, M. (2002). Multiple Model-Based Reinforcement Learning. *Neural Computation, 14*, 1347-1369.

Evans, D. (2000). *Black & White: Prima's Official Strategy Guide.* Prima Games.

Gold, A. (2005, April). Academic AI and Video games: a case study of incorporating innovative academic research into a video game prototype. *IEEE 2005 Symposium on Computational Intelligence and Games*, pp. 141-148.

Gruenwoldt, L., Katchabaw, M., & Danton, S. (2005, 08). Creating reactive non player character artificial intelligence in modern video games. *Proceedings of the 2005 GameOn North America Conference*, pp. 10-17.

Haske, S. (2016, August 16). *'Starbound' Offers a Totally Different Kind of Space Adventure*. Retrieved from Inverse: https://www.inverse.com/article/19751-starbound-developer-interview-finn-brice

Heather, A. (2016, October 18). *A Look At How No Man's Sky's Procedural Generation Works*. Retrieved from Kotaku: https://kotaku.com/a-look-at-how-no-mans-skys-procedural-generation-works-1787928446

Heinimaki, T. J., & Vanhatupa, J.-M. (2013). Implementing artificial intelligence: a generic approach with software support. *Proceedings of the Estonian Academy of Sciences*, 27-38.

Hemakumura, G., & Ruslan, R. (2018). Spatial Behaviour Modelling of Unauthorised Housing in Colombo, Sri Lanka. *KEMANUSIAAN: The Asian Journal of Humanities*, 25(2).

Juliani, A., Berges, V.-P., Vckay, E., Gao, Y., Henry, H., Mattar, M., & Lange, D. (2018, September 7). Unity: A General Platform for Intelligent Agents. *arXiv preprint arXiv:1809.02627*. Retrieved from https://github.com/Unity-Technologies/ml-agents

Jurney, C., Robbins, M., & Sunshine-Hill, B. (2012, March 5-9). Off the Beaten Path: Non-Traditional Uses of AI. *GDC 2012*. San Francisco, California, United States of America. Retrieved 2020, from https://www.gdcvault.com/play/1015667/Off-the-Beaten-Path-Non

Leone, M. (2017, January 09). *Final Fantasy 7 An oral history*. Retrieved from Polygon: https://www.polygon.com/a/final-fantasy-7

Merrick, K. (2008). Modeling Motivation for Adaptive Nonplayer Characters in Dynamic Computer Game Worlds. *ACM Comput Entertain*, 32.

Millington, I. (2019). *AI for Games* (3 ed.). Florida: CRC Press.

Orkin, J. (2006). Three States and a Plan: The A.I. of F.E.A.R.1Game Developers Conference 2006Three States and a Plan: The A.I. of F.E.A.R. *Proceedings of the GDC-06*.

Piper, K. (2019, April 13). *AI triumphs against the world's top pro team in strategy game Dota 2*. Retrieved from Vox: https://www.vox.com/2019/4/13/18309418/open-ai-dota-triumph-og

Reynolds, M. (2017, May 23). *DeepMind's AI beats world's best Go player in latest face-off*. Retrieved from NewScientist: https://www.newscientist.com/article/2132086-deepminds-ai-beats-worlds-best-go-player-in-latest-face-off/

Sanatan, M. (2019, March 20). *Theory of Computation: Finite State Machines*. Retrieved from Stack Abuse: https://stackabuse.com/theory-of-computation-finite-state-machines/

Sharma, V., Rai, S., & Dev, A. (2012, October). A Comprehensive Study of Artificial Neural Networks. *International Journal of Advanced Research in Computer Science and Software Engineering, 2*(10), 278-284.

Stanley, K. O., Bryant, B. D., & Miikulainen, R. (2009). Evolving Neural Network Agents in the NERO Video Game. *Proceedings of the IEEE*, pp. 182-189.

Tan, C. H., Tan, K. C., & Tay, A. (2011). Dynamic game difficulty scaling using adaptive behavior-based AI. *IEEE Transactions on Computational Intelligence and AI in Games, 3*(4), 289-301.

Toftedahl, M. (2019, September 30). *Which are the most commonly used Game Engines?* Retrieved from Gamasutra:

https://www.gamasutra.com/blogs/MarcusToftedahl/20190930/350830/Which_are_the_most_commonly_used_Game_Engines.php

Valve Developer Community. (2019, September 20). *ai_relationship*. Retrieved from Valve Developer Community: https://developer.valvesoftware.com/wiki/Ai_relationship

Yannakakis, G. N., & Julian, T. (2018). *Artificial intelligence and games* (Vol. 2). New York: Springer.

Appendix

Damasio, Joshua. (2020, December 5). damasioj/ML-Agents-Ecosystem: Initial release of Ecosystem (Version v1.0.0). Zenodo. http://doi.org/10.5281/zenodo.4308006

Damasio, Joshua. (2020, December 5). damasioj/ML-Agents-Resource-Collector-NPC: Initial release of Resource Collector (Version v1.0.0). Zenodo. http://doi.org/10.5281/zenodo.4308012

Damasio, Joshua. (2020, December 5). damasioj/ML-Agents-Animal-NPC: Initial release of Animal environment (Version v1.0.0). Zenodo. http://doi.org/10.5281/zenodo.4308010

Damasio, Joshua. (2020, December 5). damasioj/ML-Agents-Hauler-NPC: Preview release of Hauler scenario (Version v.0.1-preview). Zenodo. http://doi.org/10.5281/zenodo.4308014