VILNIUS UNIVERSITY

FACULTY OF MATHEMATICS AND INFORMATICS

STUDY PROGRAM: INFORMATION TECHNOLOGIES

Creation of an Artificial Intelligence System for a Game with Complex Rules

Dirbtinio intelekto sistemos kūrimas žaidimui su sudėtingomis taisyklėmis

Bachelor's final thesis

Author: Ruslan Kovaliov

e-mail: ruslan.kovaliov@mif.stud.vu.lt

Supervisor: Linas Būtėnas

Vilnius

2023

Table of contents

Key terminology 3
Abstract 4
Santrauka 5
Introduction 6
1. Analysis 8
  1.1. The concept of artificial intelligence 8
  1.2. Machine learning 9
    1.2.1. Supervised learning 10
    1.2.2. Unsupervised learning 11
    1.2.3. Reinforcement learning 12
    1.2.4. Comparison of learning styles 13
  1.3. Artificial intelligence in video games 15
  1.4. Analysis of similar systems 16
  1.5. Measuring complexity of turn-based games 20
  1.6. Reinforcement learning general implementation 21
2. System Design 23
  2.1. Description of the CubeWars game 23
  2.2. Description of existing AI algorithm 27
  2.3. AI testing sub-system 32
  2.4. AI system design considerations 34
3. System implementation 35
  3.1. Used technologies 35
  3.2. Integrating ML-Agents with the game environment 36
  3.3. Training the agent in a simplified environment 42
  3.4. Training the agent in the full environment 47
  3.5. Training results 49
  3.6. Evaluating performance 52
Conclusion and recommendations 54
Limitations and future work 55
References 56
Appendices 60
  Appendix A – Unity Engine Workflow [41] 60

Key terminology

Neural network – a computer system modeled based on the human brain and nervous system that is often used in machine learning.

AI Model – in artificial intelligence, a program that has been trained on a set of data or in a specific environment to recognize certain types of patterns that exist in that data or environment.

Agent – in artificial intelligence, an agent is anything that can perceive the environment and take actions autonomously in order to achieve different goals.

Episode – in machine learning, a sequence of actions, states and reward consisting of an arbitrary number of steps that usually corresponds to the agent finishing a task, reaching a lose condition or some sort of time limit.

Policy – in reinforcement learning, it is a mapping of the observations from the environment to the probability distribution of actions the agent can take. During training, the agent adjusts the parameters of its policy function to maximize the long-term reward.

Curriculum learning – in type of reinforcement learning where the difficulty of the task the agent must solve gradually increases as the learning progresses.

Reinforcement learning – a type of machine learning where a learning agent acts in a simulated or real environment and receives feedback for its actions in the form of rewards with the purpose of making it find the optimal sequence of actions in order to maximize its reward.

Abstract

The purpose of this project is twofold. The first objective is to design a video game AI for a computer-controlled opponent for a turn-based strategy game CubeWars. The second objective is to create an environment where this AI would be able to play against older algorithms and human players to evaluate its effectiveness.

CubeWars has been in development since 2021. It is developed using Unity Engine, a comprehensive framework for game development that is available for free. The new AI system for the game was developed using machine learning methods. Because Unity programming is done in C# or JavaScript and the most popular machine learning libraries are available in Python, the Unity ML-Agents toolkit was used to accomplish this goal. ML-Agents is a third-party, open-source add-on for Unity that facilitates data exchange between the Unity game code and lifecycle methods and a neural network running in a Python environment.

Several different models were trained using different parameters and evaluated in games against the benchmark script, which was the only AI algorithm used in CubeWars prior to this project.

Santrauka

Dirbtinio intelekto sistemos kūrimas žaidimui su sudėtingomis taisyklėmis

Dirbtinis intelektas – tai labai populiari mokslinių tyrimų kryptis su taikymais skirtingose industrijose, tokiose, kaip gamyba, sveikatos priežiūra, pilotavimas... ir vaizdo žaidimai, kas mūsu laikais yra labai didelė rinka pramogų sektoriuje. Vaizdo žaidimų dirbtinis intelektas yra tema, kuri vis labiau populiarėja mokslo srityje, nes vaizdo žaidimai gali būti naudojami kaip dirbtinio intelekto sistemų projektavimo ir testavimo aplinkos, o praktinis šių tyrimų pritaikymas gali būti panaudotas vaizdo žaidimų produktų kokybei gerinti.

Šio projekto tikslas – sukurti dirbtinį intelektą kompiuteriu valdomam priešininkui nebaigtam ėjimų kovos strateginiam žaidimui, preliminariai pavadintam „CubeWars“, ir tuo pačiu metu sukurti aplinką, kur šitas dirbtinis intelektas galėtų žaisti prieš senesnius algoritmus ir žaidėjus, kad įvertinti jo efektyvumą.

Žaidimas „CubeWars“ buvo kuriamas nuo 2021 m., naudojant „Unity Engine“ – žaidimų kūrimo sistemą, kurią galima naudoti nemokamai. Nauja žaidimo dirbtinio intelekto sistema buvo sukurta naudojant mašininio mokymosi metodus. Kadangi Unity programavimas atliekamas, naudojant C# arba JavaScript kalbas, o populiariausios mašininio mokymosi bibliotekos naudoja Python, šiam tikslui pasiekti buvo naudojamas Unity ML-Agents įrankių rinkinys. ML-Agents yra trečiosios šalies atvirojo kodo priedas, skirtas Unity, kuris palengvina duomenų mainus tarp Unity žaidimo kodo ir gyvavimo ciklo metodų bei neuroninio tinklo, veikiančio Python aplinkoje.

Keletas skirtingų modelių buvo išmokyti naudojant skirtingus parametrus ir buvo įvertinti žaidimuose naudojant statini algoritmą, kuris buvo vienintelis dirbtinio intelekto algoritmas, naudojamas „CubeWars“ iki šio projekto pradžios.

Introduction

Artificial intelligence is a very prominent area of research nowadays that is also seeing widespread application in many different industries, such as manufacturing, healthcare, transportation and many others. According to a survey by the McKinsey consulting company, in 2021 57% of surveyed companies have adopted AI in at least one function, compared with 45% in 2020 [22]. The size of the global artificial intelligence market is expected to grow by $75 billion by the end of 2023, compared to 2019. Machine learning, which is a category of AI is also seeing steady growth and analysts expect it to reach $44 billion by 2027. More and more businesses around the world are starting to utilize machine learning – in a 2020 survey, 54% companies reported that use of machine learning lead to growth in productivity. Companies like Google, Tesla and Facebook are among the more well-known users of artificial intelligence technologies. [3]

Video games are another industry where AI is extensively used. Nowadays the video game industry is among the biggest parts of the entertainment industry, whose growth was further boosted by the COVID pandemic and it has now surpassed movies and music in size. According to an article by the World Economic Forum, the gaming industry is expected to continue to grow, as demonstrated in figure 1 [28].

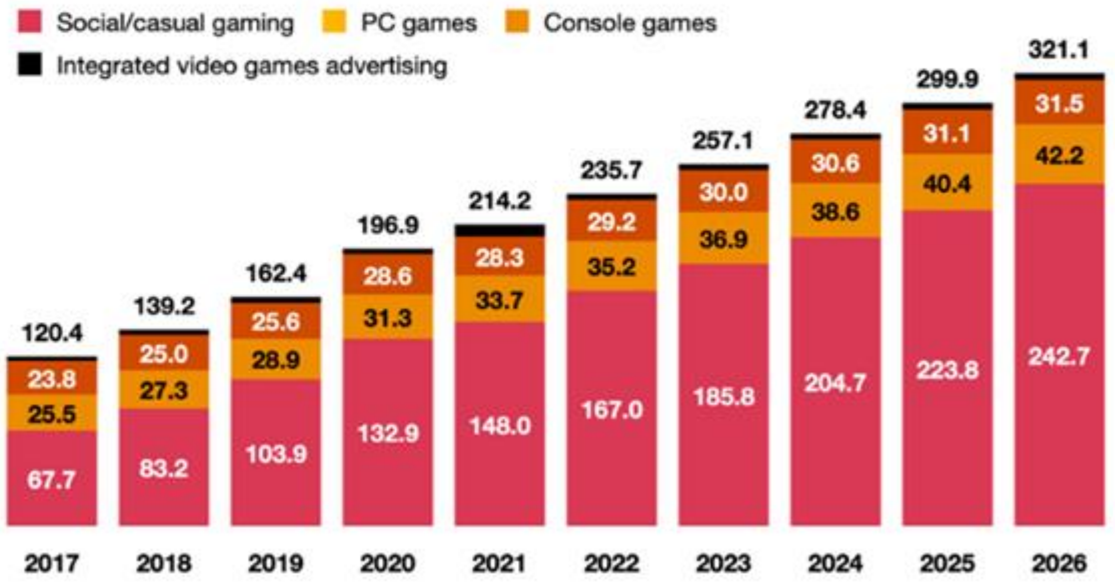


Figure 1: The global video game market. [28]

Video game AI is an important aspect of developing video games because many video games rely on computer-controlled actors to provide the player with immersion and challenge. AI in video games does not have to be able to outperform the player at any time, but rather it should be designed in such a way to make the game interesting for the player. Many techniques and algorithms exist for creating AI in video games, but there is still no definitive answer to which approach works best for which genre and whether it is possible to create a system that could be applied to any game, regardless of its type [17]. Answers to these questions could lead to the improvement of the quality of video game products.

The goal of this project is twofold. The first goal is to design a video game AI for a computer-controlled opponent for a turn-based strategy game CubeWars. The second goal is to create an environment where this AI would be able to play against older algorithms and human players to evaluate its effectiveness.

The objectives of this paper are as follows:

- Analyze the tools and techniques used to create AI for turn-based video games
- Analyze similar systems by other authors
- Design and implement a new AI system that would work inside CubeWars
- Design and implement a testing environment for the newly created AI system inside CubeWars

1. Analysis

In this chapter, the analysis of scientific literature and other information sources on the topics of artificial intelligence, its types and applications in video games is presented and similar systems are described and compared. Various methodologies that were considered or used in the implementation of the system, which is the main objective of this paper, are discussed together with their advantages, drawbacks and peculiarities. Important topic concepts, such as complexity of turn-based video games, are also explained and analyzed.

1.1.The concept of artificial intelligence

Artificial intelligence is a broad area of study concerned with making computers analyze data and make decisions. Artificial intelligence can be defined as a system that displays intelligent behavior. This system should be able to analyze its environment and take a necessary action with some degree of autonomy in order to achieve a specific goal. [24]

Another definition of AI describes it as a branch of computer science that involves developing computer programs to complete tasks that would otherwise require human intelligence. AI algorithms should be able to learn and solve problems by making logical decisions. [30]

Artificial intelligence is commonly categorized by narrowness of problems it is able to solve. Generally, AI systems are divided into three categories [34]:

- Artificial Narrow Intelligence (ANI) or Domain-Specific AI – an AI that is designed to solve problems in a specific domain that is governed by the rules and boundaries of that domain. Examples include AI systems that are trained to play chess or drive electric cars. This is the most common type of AI nowadays and this level of AI has been successfully applied in various industries.
- Artificial General Intelligence (AGI) – an AI that is able to generalize what it learned in one domain to similar domains or unrelated domains. Such an AI should be able to, for example, learn to ride a bike after learning to drive a car with minimal changes to the model. Nowadays, the problem of generalization in artificial intelligence is still hard to solve and such systems are mostly experimental and not widely used practically.
- Artificial Super Intelligence (ASI) – currently a theoretical type of AI that is able to reach a level of intelligence that is comparable to or surpasses a human’s in every field. With the current state of AI research and development, this scenario is not possible to achieve but it may become possible in the future.

This work is concerned with the concept of artificial narrow intelligence, because the purpose is to make an AI that would be able to play a specific game with specific pre-defined rules. This AI system certainly could be applied to other turn-based games with identical or similar rules in the future, but it would not be used to, for example, play completely different types of games, like games that work in real time.

1.2.Machine learning

Because many of the similar systems that were analyzed in this paper use machine learning to achieve intelligent behavior of computer-controlled agents in turn-based games, it is necessary to explain and analyze this concept before going further.

Machine learning is a very prominent area of artificial intelligence that has attracted a lot of attention from researchers and has been successfully applied to various fields. Machine learning refers to the ability of a system to automatically acquire, integrate and then develop knowledge from large-scale data and then expand that knowledge autonomously by discovering various patterns and relationships in data. This results in the computer learning to perform certain tasks without being explicitly programmed to do so. [31]

Figure 2 shows a representation of the relationship between artificial intelligence, machine learning and deep learning, another concept that simply refers to machine learning systems that use more advanced algorithms for learning as explained by Abdellah & Koucheryavy [1].

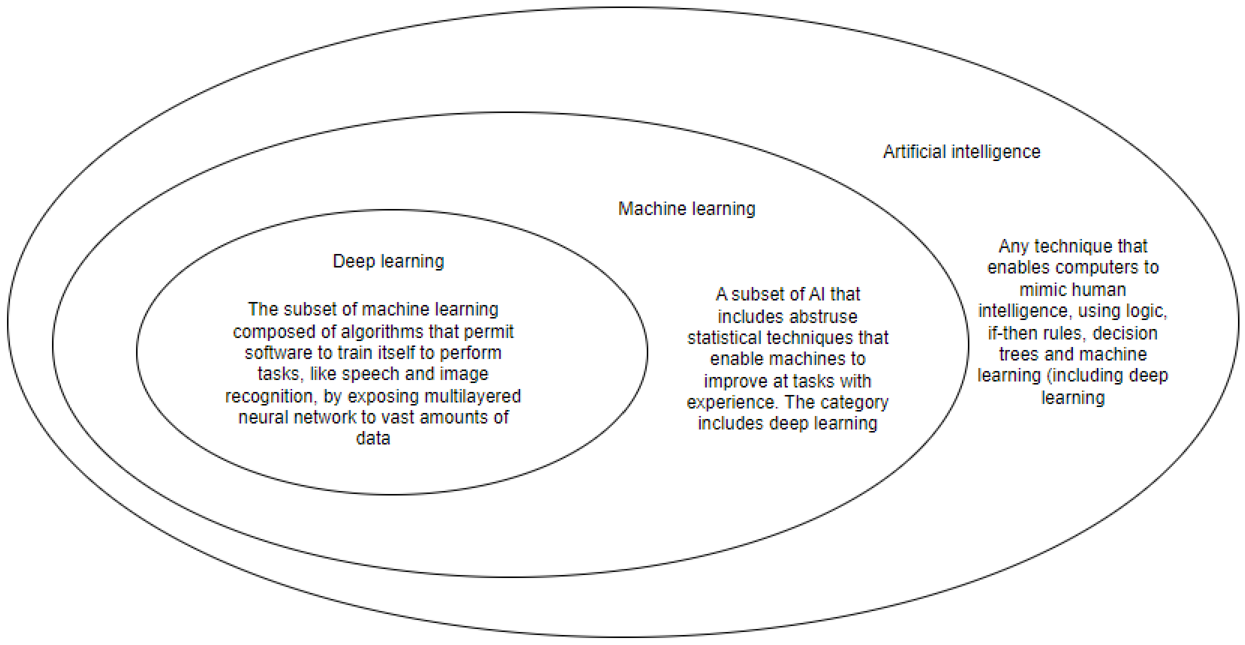

Artificial intelligence
Machine learning
Deep learning
The subset of machine learning composed of algorithms that permit software to train itself to perform tasks, like speech and image recognition, by exposing multilayered neural network to vast amounts of data
A subset of AI that includes abstruse statistical techniques that enable machines to improve at tasks with experience. The category includes deep learning
Any technique that enables computers to mimic human intelligence, using logic, if-then rules, decision trees and machine learning (including deep learning

Figure 2. Relationship between AI, ML and DL. [6]

As can be seen, all machine learning and deep learning systems are part of artificial intelligence, but not all artificial intelligence is machine learning or deep learning.

Machine learning algorithms can be categorized according to different criteria. One of the most common criteria present in scientific literature is how the learning process happens. Often, the following learning types are described: supervised learning, unsupervised learning and reinforcement learning. [7]

1.2.1. Supervised learning

Supervised learning involves training the AI using labeled data. The algorithm is provided with input variables that are labeled. In other words – the developer tells the AI which unit of data corresponds to which output. The algorithm is then “trained” to provide the expected output for each unit of data. Afterwards, the algorithm is tested using similar data that is not labeled [12]. The flowchart for this process is presented in figure 3.

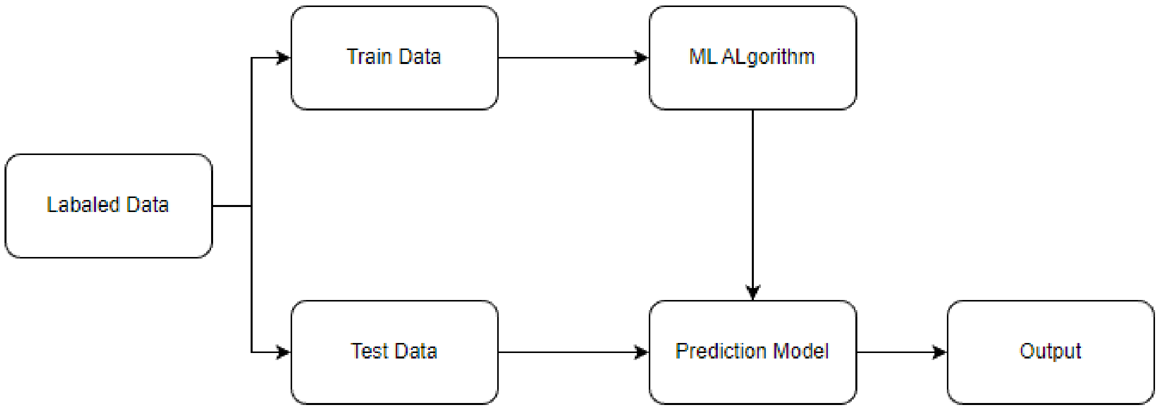


Figure 3. Supervised learning AI flowchart [12]

Figure 4 describes how unsupervised learning AI actually learns to map input variables to desired output [21]:

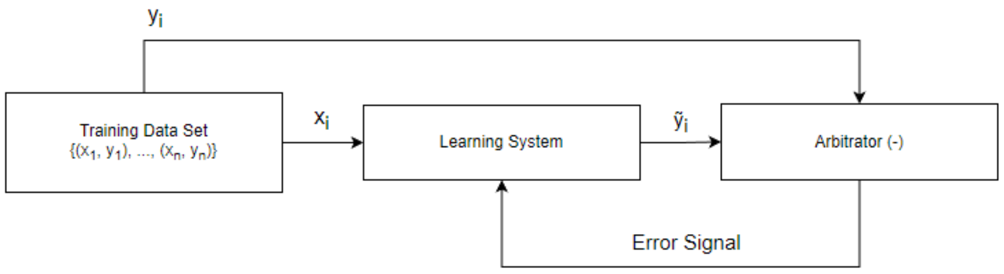


Figure 4. Supervised learning process. [21]

Here, (x_i,y_i) denotes the labeled data sample, where x represents the input and y represents the output by means of being a label of input x. During this process, the system receives the input x_i and for this input, generates an output denoted as $\tilde{y}_i$. The provided output is the compared with the expected output y_i using an arbitrator, which is also sometimes called the loss function or cost function. The difference between the provided and expected output is the error, which is used to adjust the parameters of the model. The goal of the learning process then becomes to minimize the output of this cost function, because the lower the result, the smaller the difference, the more similar is the provided output to the expected output.

A typical example of supervised learning is instructing the AI to map numbers drawn as images to the actual numeric value they represent. The training data consists of a set of labeled number images and then, after the training process concludes, the algorithm is provided with a set of unlabeled images to see how accurately it assigns values to the images.

1.2.2. Unsupervised learning

Unsupervised learning, as the name implies, involves training the AI only with inputs, with no labels. The model is expected to be able to, from a set of unorganized and unlabeled data, classify, label and group the data without any external guidance when performing this task. The main goal of unsupervised learning is to discover patterns in the data that could have escaped the eye of human analysts. In unsupervised learning, there are no "correct" pre-defined categories and it is up to the AI to determine the differences between data items provided and perform tasks with those items based on the differences observed. [13]

An example of how unsupervised learning could work would be if the system was provided with a set of data (for example, different animals), that share some common characteristics (for example, number of limbs) but also have observable differences. It then falls on the AI to determine, how this data can be grouped (for example, some animals have fur, while others do not) by going over the data items and analyzing the differences and afterwards, actually grouping the data according to its own identified criteria. An example workflow is presented in figure 5.

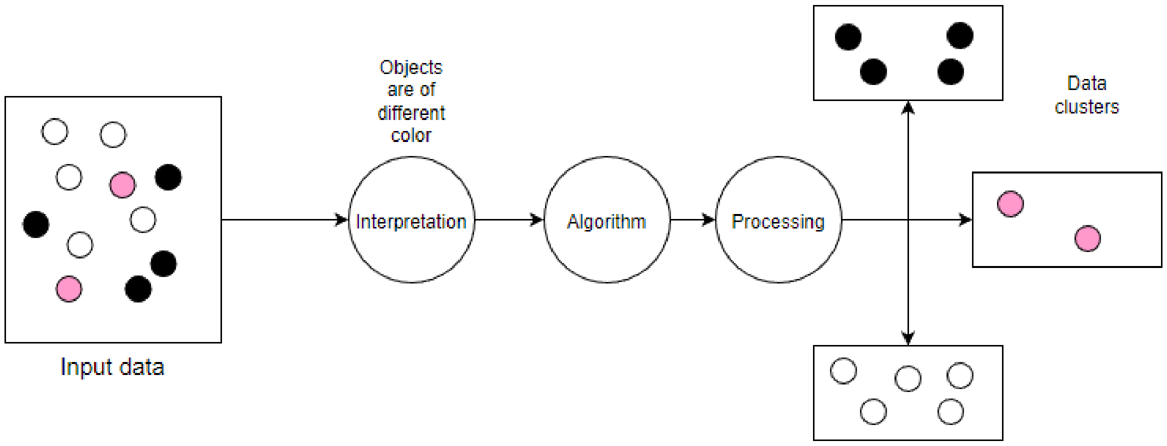


Figure 5. Data grouping (clustering) via unsupervised learning process. [25]

Unsupervised machine learning can identify various patterns in the data that would not be possible with supervised learning. It is also less expensive and time consuming to set up, because the developer does not need to spend time labeling training data sets. However, using it also carries certain risks. It is less accurate than the supervised learning process and can produce results that are harder to interpret because they are not bound to specific pre-defined categories, so while they are easier to set-up, analyzing the results becomes more difficult. [13]

1.2.3. Reinforcement learning

The principle of reinforcement learning is to have the AI base its actions on feedback from previous actions. This learning method exposes the AI to an environment where it performs certain actions using trial and error. The AI learns from past actions and makes better decisions in subsequent attempts. This is accomplished by giving the AI "rewards" or "punishments" for making good or bad decisions. Unlike in previous methods, the AI does not have access to labeled information and instead of the simple tasks of categorization or clustering, it can perform a variety of actions and reach different states. [8]

An example of how reinforcement learning works would be an example where an AI is asked to choose between 2 paths. At the beginning, it has no knowledge of the situation, so it will act based on random choice. Then the AI is given feedback on the path it chose. Assume that path A is shorter than path B, so the AI will be given a reward if it chooses path A. The next time this situation repeats, the AI will choose path A based on previous experience. For complex problems, the AI will be expected to make a series of choices some of which can be linked together (for example, if the AI made a choice in step 1, it will be expected to make a certain choice in step 2).

In figure 6, 2 reinforcement learning diagrams are presented. Diagram "a" shows the classical reinforcement learning schema, where "t" represents the iteration or step of the process, "s" represents a state of the environment, "r" represents the reward and "a" represents the action the AI can perform. At each step "t", the AI performs certain actions that change the environment the AI operates in, which in turn can influence available actions for the next step. The goal of the process is to maximize the reward obtained from performing "t" actions.

In the diagram "b" a more advanced reinforcement learning algorithm utilizing deep learning is shown. In that example policy "π" is the function that must be maximized by the algorithm to obtain the biggest possible reward [15].

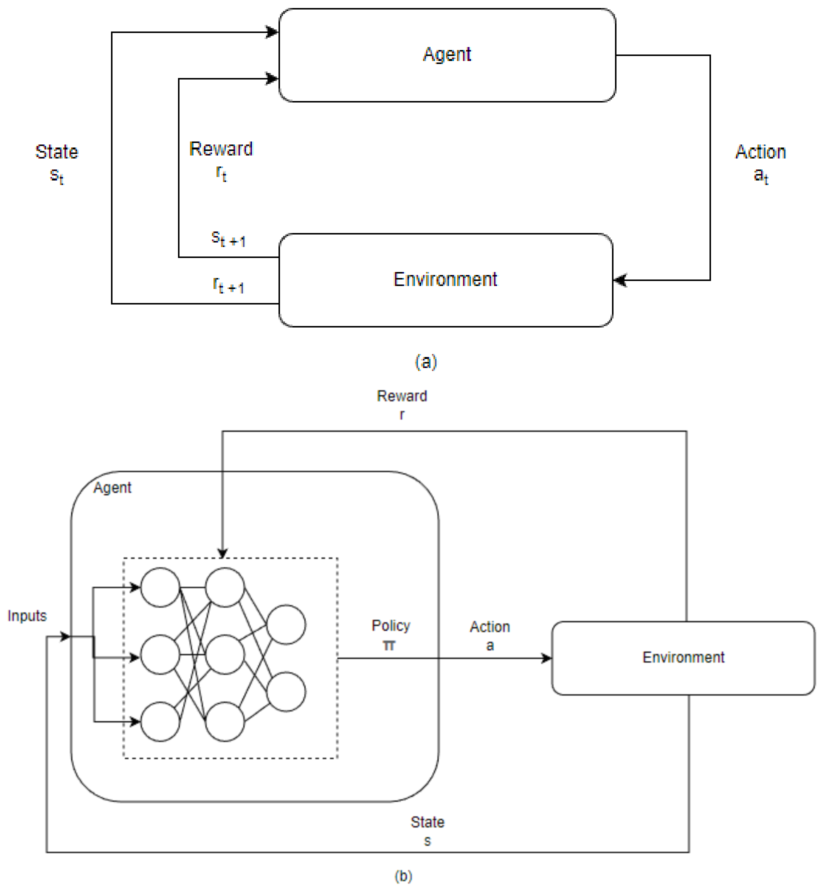


Figure 6. a. Classical reinforcement learning. b. Deep reinforcement learning. [15]

Reinforcement learning is often used in the development of game AI, where the AI player is trained by playing millions of matches against itself or other AI algorithms, where the goal is to find a set of consecutive steps that would result in a victory (maximized reward). [13]

1.2.4. Comparison of learning styles

Each of the learning methodologies described above comes with its all advantages and disadvantages and are suitable for solving different problems. A comparison of the learning methodologies based on some common criteria is presented in the table 1.

Table 1. Comparison of learning methodologies. [4]

Criteria	**Supervised Learning**	**Unsupervised Learning**	**Reinforcement Learning**
Input Data	Input data is labelled.	Input data is not labelled.	Input data is not predefined.

Problem	Learn pattern of inputs and their labels.	Divide data into classes.	Find the best reward between a start and an end state.
Solution	Finds a mapping equation on input data and its labels.	Finds similar features in input data to classify it into classes.	Maximizes reward by assessing the results of state-action pairs
Model Building	Model is built and trained prior to testing.	Model is built and trained prior to testing.	The model is trained and tested simultaneously.
Applications	Deal with regression and classification problems.	Deals with clustering and associative rule mining problems.	Deals with exploration and exploitation problems.
Algorithms Used	Decision trees, linear regression, K-nearest neighbors	K-means clustering agglomerative clustering	Q-learning, SARSA, Deep Q Network
Examples	Image detection, Population growth prediction	Customer segmentation, feature elicitation, targeted marketing, etc.	Drive-less cars, self-navigating vacuum cleaners, etc.

Each of these learning methodologies is typically associated with certain tasks that they excel at. Siadati (2018) provides a visual representation of the different methodologies and their related tasks, which is demonstrated in figure 7.

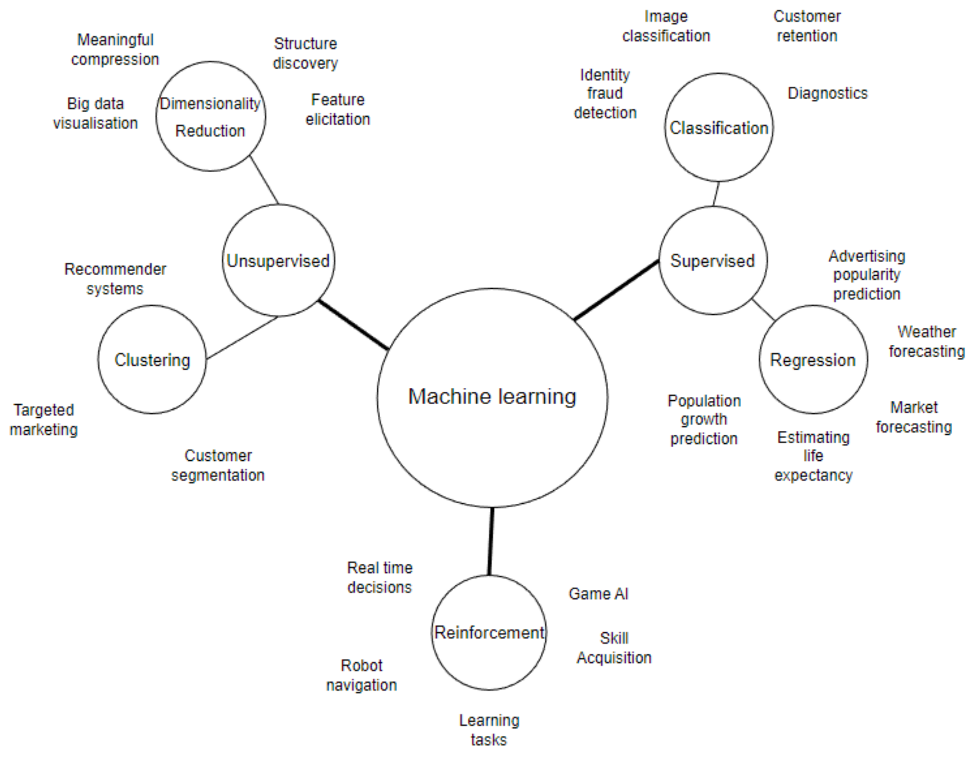


Figure 7. A sample of ML algorithm types and most common tasks. [35]

Reinforcement learning will be the primary focus in the following sections, as the reinforcement learning methodology is most suitable for training agents in video games.

1.3.Artificial intelligence in video games

Artificial intelligence can be applied in video game development in different ways. Xia, et. Al. (2020) [43] outline several areas of application in their paper, which include:

- Designing in-game agents that act intelligent and believable. These agents are able to make decisions effectively and pose a challenge or increase the immersion of the player.
- Generating levels and other procedural content.
- Modeling the player's experience and adapting the game's setting to suit the player

Another very popular area of AI application in video games is pathfinding – making the agents correctly navigate the game world, while avoiding obstacles and inaccessible areas. This is one of the most heavily researched areas concerning video game AI and powerful algorithms like A* have been created and successfully applied in this field. [9]

AI can also be used to upscale the resolution of in-game images and textures, enhancing the game's visual qualities and even create new images on the basis of existing ones using generative techniques. [23]

There are even computer games where entire scenarios, game rules, entities and visual components are completely or partially generated by an AI – one example is AI Roguelite, a game created by an independent developer and information about it is available only on the Steam gaming platform [36]. Games of this type are still highly experimental and are appreciated more for their novelty rather than the gameplay experience that they can provide.

This paper will focus particularly on the development of an AI system for decision making – the creation of a computer actor that would be able to take advantage of various game rules and interactions to play against a human opponent and provide a reasonable amount of challenge.

1.4.Analysis of similar systems

In this section, results of existing research into the topic of AI decision-making agents for computer games will be presented, discussed and summarized. Designing and implementing an intelligent computer opponent in a video game is a research area that has gained some amount of traction, as video games provide for interesting testing grounds for designing AI systems. [2]

Santoso & Supriana (2014) [37] proposed an AI system for a simple turn-based strategy game that is played on a small 23x23 board without obstacles, where each side controls a certain amount of soldiers with several characteristics and every soldier type having a strength or weakness against other types. Screenshots of their system is provided in figure 8.

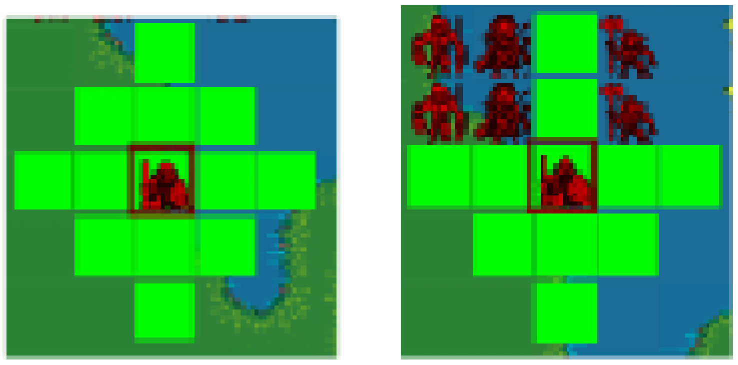


Figure 8. Visualization of the game system. [37]

The proposed system makes the computer controlled player choose from a list of hard-coded strategies rather than choosing specific actions, which significantly help the AI agent's learning by limiting the number of decisions it can make and all the computations of possible branching paths stemming from those decisions. An algorithm based on Spronck's dynamic scripting is then used to filter the possible strategy selection for each unit and the resulting system is trained using a mixture of reinforcement learning and the Minimax algorithm. The proposed encoding of different actions as a list of strategies can be applied to this project, as the state space of CubeWars is even more significant than in the authors' game, which exponentially increases the difficulty of computing the best action.

Sestini, et. Al (2019) [32] propose a system for a rogue-like game (game with randomly generated levels where the characters can only take 1 action at a time) called DeepCrawl that utilizes reinforcement learning exclusively. The relatively small and simple board and limited action selection translated well into using RL because the board states and actions could be easily encoded as inputs for the algorithm. The authors recommend designing the AI in such a way so that it doesn't always take the best action available to create an illusion of fairness. To implement their system, the authors created the game using the Unity Engine and used Unity ML-Agents to train the AI. For training, the authors used a curriculum – a type of reinforcement learning environment that changes based on certain conditions. In this system's case, it changes over time, depending on how long the agent has been training in order to make sure that the agent learns the required behavior – for example, in early stages of training, enemies are made much weaker so that the agent learns to fight and defeat them and receive the positive reinforcement associated with those activities. Figure 9 and table 2 shows how the authors' curriculum system works.

Table 2. DeepCrawl curriculum phase details. [32]

Phase	Agent's HP	Enemy HP	Loot quantity
1	20	1	20%
2	[5, 20]	10	20%
3	[5, 20]	[10, 20]	20%
4	[5, 20]	[10, 20]	[10%, 20%]
5	[5, 20]	[10, 20]	[5%, 20%]

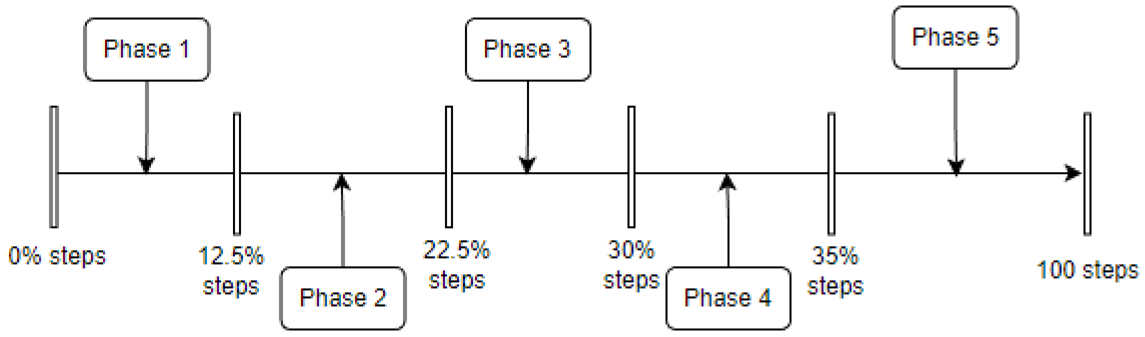


Figure 9. DeepCrawl curriculum progression. [32]

The percentages in the timeline represent the percentage of training steps the agent has performed – the idea is to make the training environment more difficult as more time has passed.

Hyrkäs (2015) [17] Used reinforcement learning to teach an AI to play an existing strategy game called King of Thule. The author used a style of reinforcement learning called adversarial learning where both the player and the opponent are learning agents and both parties' actions contribute to training the neural network to make correct decisions.

Dockhorn, et. Al. (2020) [11] propose Stratega, a turn-based strategy game AI training platform where it is possible to customize the game rules and have the agents learn the game with updated rules. The game template they use for their system is quite simple (square grid, single action per unit) so it wouldn't be suitable to adapt it to CubeWars, but the authors' system does possess some interesting characteristics that could be adapted in this project, like the forward planning model, which allowed the AI to calculate the theoretical outcomes of various actions and pick the ones which result in the best outcome. For these calculations the authors use the Monte Carlo Tree Search (MCTS) algorithm. The authors also provide interesting insight into the methods of estimating the threat level of various units on the field, like their proposed Strength Difference Heuristic, that calculates a unit's threat level by summing its values of its attributes divided by the maximum value of these attributes on the map.

Rill-Garcia (n.d.) [29] created an AI that could play Pokemon, which is a popular game series, where two teams of creatures fight against one another and each creature has a strength, a weakness and a few abilities it can use. As far as turn based computer games go, Pokemon is very simplistic – there is no movement on the game board, no complicated characteristics and each side can only have one creature active at a time, meaning that one match is a string of "1 versus 1" battles. The author argues that finding an optimal winning strategy even for such a simple game is an extremely difficult computational task (PSPACE-hard in EXPTIME) so he recommends using a machine learning approach instead. The proposed system uses Q-Learning, which is a type of reinforcement learning algorithm that works well when the number of actions an agent can take at any given moment is small and consists of discrete numbers. The AI was trained on different types of data – first on data from existing matches, then in simulations against itself and then data from games it played in real time. The author also suggests that it would be possible to use reinforcement learning together with the Minimax algorithm for this task.

Another system by Amato & Shani (2010) [2] involves using reinforcement learning to train an AI to play Civilization IV, a popular turn-based strategy game that is commercially available. Civilization IV is a game that is much more complex than CubeWars, however the authors reduce the complexity of the proposed system by making the AI choose only 1 of the 4 possible high-level strategies instead of choosing individual actions, similar to how it works in Santoso's & Supriana's (2014) [37] system. Moreover, the strategies that the AI can choose already exist in the game. Nevertheless, these two systems show that abstracting the possible selection of actions to a limited number of higher-level strategies can make the AI's job much easier by exponentially reducing the complexity of decisions. Because limiting the choice of actions to one of 4 possible strategies makes the learning agent's action space quite small, the authors used the Q-Learning algorithm for training.

Kimura & Kokolo (2020) [19] created a grid based turn-based strategy environment called TUBSTAP and designed an AI that could play games in that environment using a combination of reinforcement learning and search tree algorithms. To make learning easier, the authors recommend making some of the possible decisions for the learning agent and then letting it make the rest of the decisions itself. For example, in their scenario the possible decisions involve choosing, which unit to move, choosing where to move and choosing whom to attack, so the authors limit the AI from choosing which unit to move and make the choice for it. The AI then receives the chosen unit as the input, together with other observable parameters and must decide what to do with that unit. The authors train the agent using various map configurations, some of which are presented in figure 10.

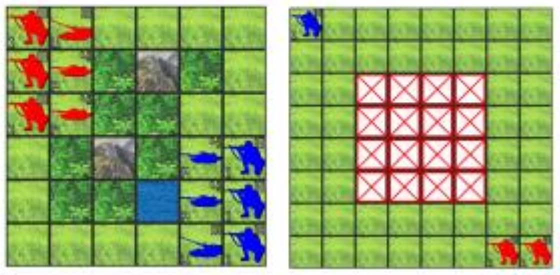


Figure 10. 2 examples of TUBSTAP scenarios. [19]

Zhang, et. Al. (2019) [44] used a similar approach. They created an AI for a turn-based mobile game about creature battles that could only select ability-target combinations from a list of possible pairings. Their system involved a combination of decision trees, regression and deep learning. The authors have tried a reinforcement learning model but claimed that it had poor results, however they explain that it was likely due to a simplistic reward function and that results could be improved with a better one. In their experiment, the reinforcement learning based model only reached a 33.7% win rate against the benchmark script, while their DT + LR + DL model managed to reach a 57.6% win rate.

The similar systems that were analyzed can be separated into three main categories – systems that use reinforcement learning exclusively, systems that use other types of algorithms (for example, search algorithms) exclusively and systems that use a combination of both (for example, an algorithm is used to filter the input or limit possible output and then ML is utilized to train the agent). The summarized findings are presented in table 3.

Table 3. Summary of similar systems.

Related work	**Category**	**Tools and techniques**
Santoso & Supriana [37]	ML + Algorithm	Turning action selection into a selection of a high-level strategy. Filtering strategies based on Spronck's dynamic scripting. Minimax algorithm.
Sestini, et. Al. [32]	ML	Unity Engine. Unity ML Agents. Reinforcement learning. Sampling from a list of top predicted actions instead of choosing the best action all the time. Curriculum learning.
Hyrkäs. [17]	ML	Reinforcement learning. Adversarial learning.
Dockhorn, et. Al. [11]	Algorithm	Forward planning model. Strength Difference Heuristic. Monte Carlo Tree Search.
Rill-Garcia [29]	ML	Reinforcement learning (Q-Learning).

		Training the agent on different types of data.
Amato & Shani [2]	ML	Turning action selection into a selection of a high-level strategy. Reinforcement learning (Q-Learning).
Kimura & Kokolo [19]	ML + Algorithm	Limiting the agent’s actions by taking some of the actions for it and providing them as input.
Zhang, et. Al. [44]	ML + Algorithm	Limiting the agent’s actions by taking some of the actions for it and providing them as input. Tested both reinforcement learning and deep learning combined with search algorithms.

As can be seen from this summary, the majority of works used either ML exclusively or ML combined with other algorithms, but only one of the works didn’t use ML at all. The reason for this is that turn-based strategy games have a huge search space, much larger than traditional board games, which makes it very hard to compute the next move. Moreover, turn-based computer games often have various characteristics of different pieces and topographic elements that don’t exist in traditional board games. Also, they don’t have a fixed game-board size and configuration and map layouts can change from game to game [19]. While it certainly possible to design sophisticated AI systems for computer games using algorithms like Minimax or decision trees or make simpler but still effective systems that can meet the game design goals of providing a human player with a challenge or an interesting situation to solve, many researchers, especially recently, have been turning to machine learning or some combination involving it.

1.5.Measuring complexity of turn-based games

Measuring the complexity of a game like CubeWars mathematically is difficult and existing scientific literature is focused on either measuring the complexity of board games or measuring the complexity of path-finding algorithms of computer games.

As this work is focused specifically on the decision-making component of artificial intelligence rather than path-finding and CubeWars is similar to zero-sum turn-based board games, measurement methods used in papers focused on board games could be applied here.

Most common ways of measuring game complexity are state-space complexity and game-tree complexity. State-space complexity refers to the number of different game positions that are theoretically possible to achieve. Game-tree complexity refers to the total number of distinct plays for a given game [16]. Game-tree complexity is more difficult to compute, so state-space complexity is the more widely used approach to calculate game complexity. There are also other methods, like decision complexity and computational complexity, which are less common. [20]

For example, tic-tac-toe, a very simple game, has a state-space complexity of 4. This number is achieved by taking the 3 possible states of each square (X, O or empty) to the power of 9 (the number of spaces on the board). The resulting number 19863 indicates the number of different possible board states. Some of them are illegal, for example a board of all Xs or all Os – this is called upper-bound state-space complexity, and this approach is used because it is much easier to calculate. Because for more complex games the numbers get exponentially larger, the log of base 10 of the result is used. $\text{Log}_{10}(19863)$ approximately equals 4 and thus the state-space complexity of tic-tac-toe is 4. [14]

Chess, which is considered to be a complex game, usually includes 32 pieces and 64 board spaces. When calculating state-space complexity, it is assumed that the first piece can be placed in any of 64 spaces, then the second piece in the 63 remaining spaces and so on. The fact that some of these configurations are impossible in a real game is ignored. It should also be kept in mind that some of the pieces in chess are identical – there are 8 pawns and two pairs of advanced pieces in each set. Thus the state-space complexity of chess can be given by the following calculation: $64!/(32!8!^2 2!^6)$ [33]. The result of the calculation is $4.6 * 10^{42}$, which indicates all the possible board configurations given the conditions above. As a result, the state-space complexity of chess is 43, due to rounding up. These are just a few examples how game complexity can be calculated using the most common methods.

1.6.Reinforcement learning general implementation

Before describing the specific implementation details of reinforcement learning in the context of CubeWars, it is necessary to explain some general concepts in reinforcement learning implementation. Reinforcement learning is implemented through the following steps:

1. A learning environment is created. This environment has to provide the agent with the necessary data that it must use to make its decisions. The agent does not need to get all the data from the environment, but only what it needs to accomplish its task. In the case of this project, the environment for the agent is the game described in the previous sections.
2. A reward function is created. The reward function determines, which actions the agent is rewarded for. The reward in this case refers to a simple number and it can be implemented in many different ways. For example, an agent can be given 1 point for a successfully completed task and 0 points in any other case (this type of reward function is called sparse rewards) or it can be given an arbitrary number of points depending on how well it performed after every action (this is called shaped rewards).
3. An algorithm is chosen. Reinforcement learning has many different algorithms that determine, which mathematical functions will be used to find the optimal actions given the known state. As a result, some algorithms require a smaller number of data to learn, but others learn more slowly but can potentially reach greater results over time. [39]. Reinforcement learning algorithms can be categorized into different groups according to certain criteria. One popular dividing factor is whether the algorithm is model-based or model-free [26]. Model-based algorithms are given the model of the environment (they are provided with information on what action would correspond to which change in state), while model-free algorithms don’t get that information. Model-based algorithms require more effort to set up as the environment model has to be described and fed to the algorithm, but they are useful in cases where the environment is always the same and

operates using deterministic rules, for example, in board games like chess or go. In such cases the agent is able to learn very effectively and gain the ability to plan ahead. Model-free algorithms are more popular because they are easier to set up and can work in cases where a model of the environment is too difficult to translate into a function. Model-free algorithms are also typically separated into two categories: policy-based and value-based (Q-learning). Policy based algorithms build a policy (the agent's desired strategy) and iterate it over time, while value-based methods build an action-state table, called the Q-table. Figure 11 provides a useful summary of different RL algorithm types.

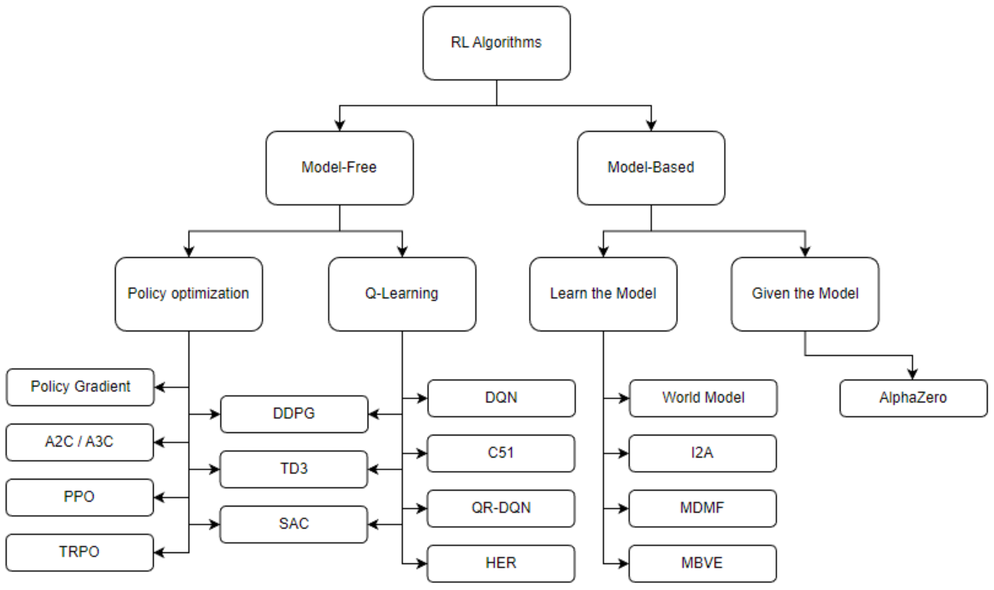


Figure 11. Types of RL algorithms. [26]

Different algorithms also require different types of input. For example, Q-Learning algorithms can only work, when the agent's actions can be laid out as a set of whole numbers and every number would correspond to a specific action, while algorithms like PPO can work with these actions as well as actions that are given as a floating-point number in a range. Algorithm choice is also limited by the libraries or frameworks you are working with.

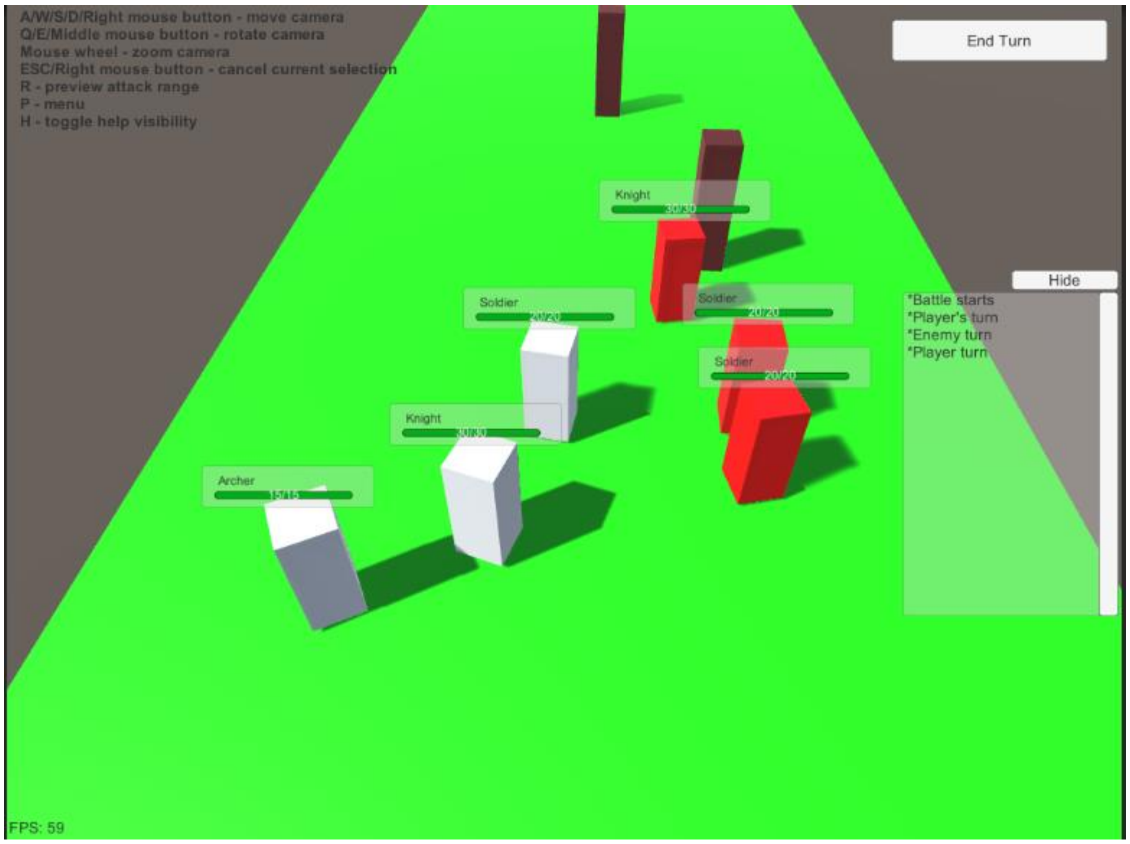
4. The algorithms' hyper-parameters are optimized. Reinforcement learning algorithms come with a variety of hyper-parameters that influence learning. They include learning rate (how much the neural network parameters of the algorithm are allowed to change each update), discount factor (how much the agent values the immediate reward it can get now versus the potential reward in the future), entropy regularization (an indication of how random the actions an agent performs can be) and many others. Usually, reinforcement learning libraries offer a pre-configured set of hyper-parameters for each algorithm but offer the possibility of modifying them.
5. The agent is trained a set number of steps, which can take a different amount of time depending on the complexity of the environment. Depending on the results, environment parameters, the reward function and/or the algorithm and its hyper-parameters are changed and training is restarted.

2. System Design

In this section the design of the system will be described. Because the AI system is being created for CubeWars, a video game project that has been worked on prior to starting this paper, the game, its specifics and rules as well as the already existing work-in-progress AI algorithm that it uses will also be described.

2.1.Description of the CubeWars game

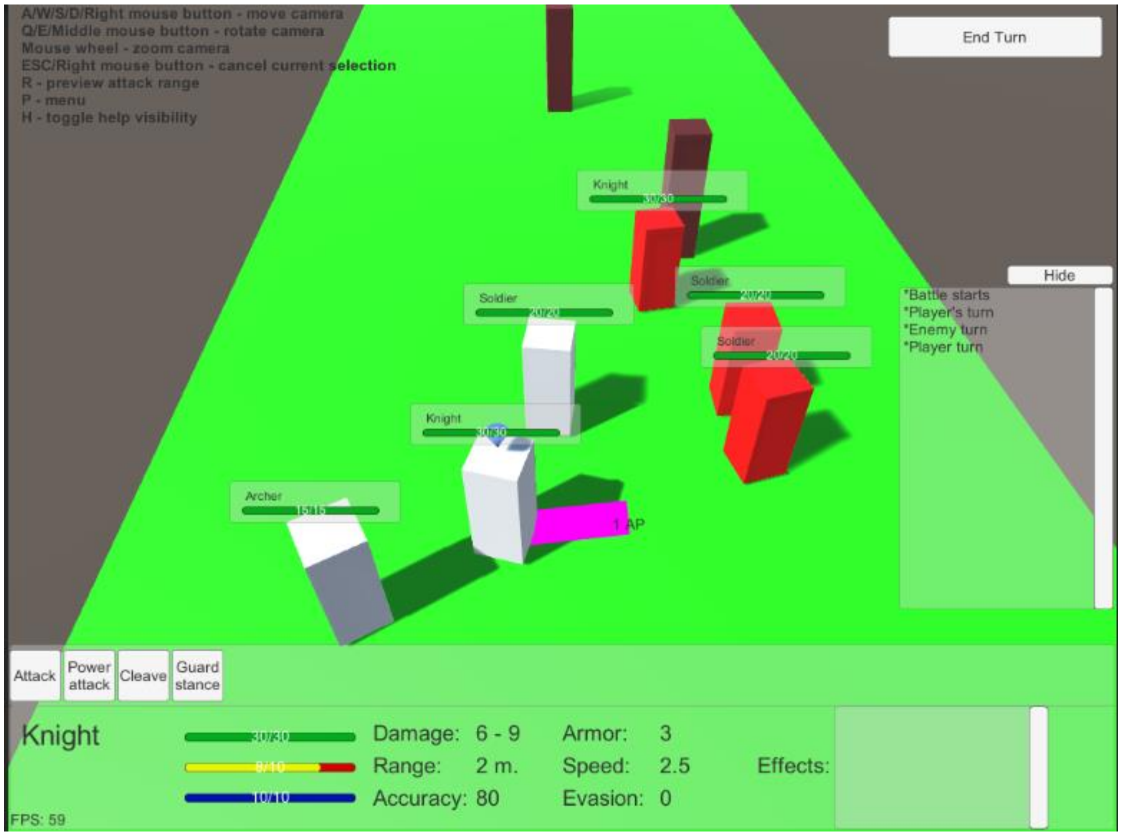
CubeWars is a work in progress project developed using the Unity engine. Currently, the game takes place on a field where 2 players' armies fight each other in turn-based combat, similar to a board game. One of the armies is controlled by a human and the other by the computer. Each player's army consists of a varying composition of different types of units, each having access to different possible actions (a soldier can strike in close range or concentrate, increasing its accuracy, a knight can enter a defensive stance, an archer can shoot arrows and etc.). A screenshot from the game showing 3 player-controlled and 3 computer-controlled units positioned close to each other is provided in Figure 12.


A/W/S/D/Right mouse button - move camera
Q/E/Middle mouse button - rotate camera
Mouse wheel - zoom camera
ESC/Right mouse button - cancel current selection
R - preview attack range
P - menu
H - toggle help visibility

*Battle starts
*Player's turn
*Enemy turn
*Player turn

Figure 12. Screenshot of CubeWars.

Each unit also has 3 resources that it works with – hit points (HP), energy and action points (AP). Hit points represent the unit's health. They decrease if a unit is hit by an attack and increase when it is healed. If a unit's HP goes down to 0, the unit is killed and is removed from the battlefield. Energy is a resource units spend to use abilities. Most abilities, other than simply attacking with a weapon, cost energy. Currently, there is no way to restore energy in the middle of the game so every unit possesses a finite amount of energy. AP are used to perform any type of action, from moving to attacking and using abilities. Each unit has a certain amount of AP that it can use and that amount replenishes at the start of the unit's turn. Unlike, for example, chess, where each player can only make one action during their turn, in this game a player can keep making actions as long as any of their units have AP remaining to do so. Figure 13 shows the information panel for a selected unit (knight), where each of the resource is represented using a colored bar: HP is the green bar, energy is the blue bar and AP is the yellow bar.



A/W/S/D/Right mouse button - move camera
Q/E/Middle mouse button - rotate camera
Mouse wheel - zoom camera
ESC/Right mouse button - cancel current selection
R - preview attack range
P - menu
H - toggle help visibility

*Battle starts
*Player's turn
*Enemy turn
*Player turn

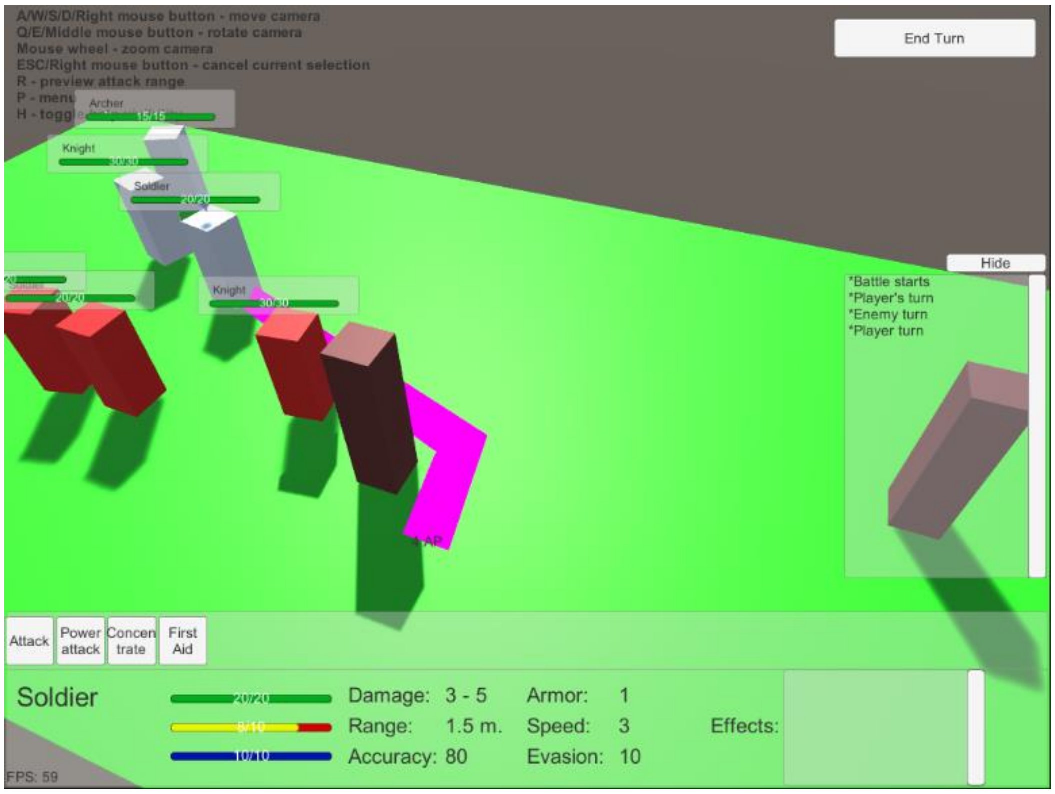
Knight
30/30
8/10
10/10
Damage: 6 - 9
Range: 2 m.
Accuracy: 80
Armor: 3
Speed: 2.5
Evasion: 0
Effects:

Figure 13. Information panel of the knight.

The screenshot also shows some of the statistics units possess, including: damage (how much HP does a unit's attack take away from its target), range (how far does the unit's weapon reach), accuracy (how likely are the units attack's to hit), armor (how much damage is mitigated when this unit is attacked), speed (how many units of distance can the unit move per 1 AP) and evasion (how likely is the unit to avoid being hit).

Every combat interaction in the game is calculated using certain mathematical rules. Consider an example where a knight attacks an archer with its normal attack. The knight's attack can deal from 6 to 9 damage and it has an accuracy of 80. The archer has 0 armor and 10 evasion. This means that the knight has a 70% chance to hit the archer (base accuracy of 80 minus the 10 evasion of the target) when it attacks, and if the attack hits, it will reduce the archer's HP by a random number between 6 and 9. Attacking the archer will reduce the knight's AP by 3, meaning that if the knight started its turn next to the archer, it could make 2 more attacks afterwards.

Movement and positioning of units in the game is handled using a 3-D coordinate system, however at the moment, units can only move on a 2-axis flat plane. A unit can move from its origin coordinates to any coordinates on the plane, provided that they are not obstructed by other units or obstacles. Depending on the unit's speed, it can move a certain distance for every AP spent. A path-finding system using Unity's NavMesh library was implemented to make units navigate around obstacles. Figure 14 demonstrates path calculation for a human controlled unit, where the purple line represents the path the unit will take upon movement confirmation and the number above the end point of the line represents how much AP will be deducted upon performing the movement.



A/W/S/D/Right mouse button - move camera
Q/E/Middle mouse button - rotate camera
Mouse wheel - zoom camera
ESC/Right mouse button - cancel current selection
R - preview attack range
P - menu
H - toggle

*Battle starts
*Player's turn
*Enemy turn
*Player turn

Soldier 20/20 8/10 10/10
Damage: 3 - 5 Armor: 1
Range: 1.5 m. Speed: 3 Effects:
Accuracy: 80 Evasion: 10

Figure 14. Path calculation example.

Currently the player is able to pick one of three difficulty levels when starting the game, and each difficulty has a different battlefield with a pre-determined computer-controlled army associated with it. The positioning and composition of units in each army is imbalanced – on easy difficulty, the player can potentially have more units deployed than the computer opponent, while on medium and hard difficulty, the computer has the numerical advantage.

The sequence diagram in Figure 15 demonstrates how the current iteration of the game is played and what interactions happen between the player and the computer opponent.

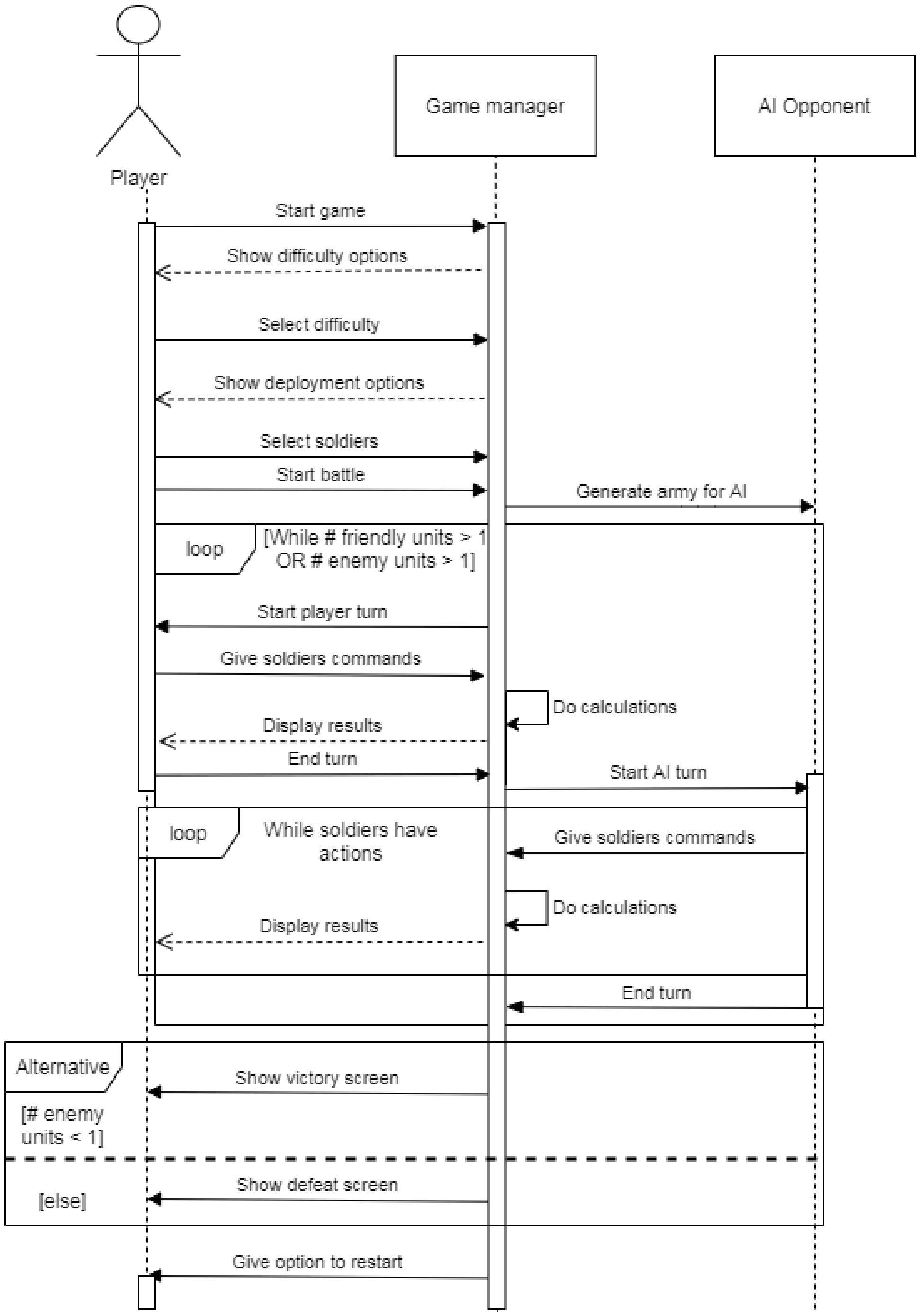


Figure 15. Sequence diagram of CubeWars.

When the game is started, the player selects the difficulty and the desired unit composition. Then combat begins and the player gets to give orders to all their units until the end turn button is pressed. When the player's turn ends, control is given to the computer opponent which gives orders to its units according the logic of the its algorithm and when it can give no more orders, it gives the control back to the player.

From the point of view of game rule complexity, the current iteration of the game has:

- Movement in continuous 2-D space bounded by the X and Z coordinates of the battlefield
- Different possible action choice combinations per unit
- Different possible order in which the player controls its units
- Different target that can be selected to attack
- Different starting configurations of the battlefield

This makes CubeWars more complex than the majority of the games presented and discussed in related work with the exception of commercially available games made by professional studios, like Civilization, as these games have at least one of these decision-making dimensions missing.

The problem with calculating the complexity of CubeWars is that unlike in a game like chess, the board and its configurations can change depending on game settings. For example, the board can be 50 units long and wide or 100 units long and wide, the unit composition can be different and there can be obstacles on the map. If we assume a 25x25 game field with 2 opposing sides, each consisting of 5 units, we can similarly calculate state-space complexity as (25*25)!/(25*25-5*2)!. The log base 10 of this number is 3338. Unlike chess, in this example every “piece” would be a unique unit with different characteristics. Even if the unit types are the same, their parameters can differ. This method also vastly oversimplifies the movement in the game space, as it assumes that units can only move to whole number coordinates. Even with these limitations it can be observed that an average game map will have a much greater state-space complexity than chess. It should be noted that higher state-space complexity does not mean that the game is more difficult to play, but it is more difficult for an AI to solve using brute-force approaches.

2.2.Description of existing AI algorithm

Prior to starting work on this paper, a certain amount of work was already made on the AI of CubeWars. A simple behavioral algorithm was created as a temporary solution so that the player would be able to play the game and interact with the opponent in a limited way.

The algorithm, which shall be referred to as the greedy algorithm or greedy AI consists of a method that is launched whenever the AI player’s turn begins and several smaller methods that run inside it to help it make decisions. Pseudo-code of the main algorithm and its helper methods is provided in Algorithm 1.

Algorithm 1. Greedy AI main algorithm

```
Procedure  :  HandleTurn()
Input      :  -
Output     :  -
Step 1     :  for each unitC in the list of computer-controlled units do
Step 2     :      Define unitP = FindClosestEnemy(unitC)
Step 3     :      Define enemyInRange = false
Step 4     :      Define usableActions = List<Action>
Step 5     :      usableActions = GetUsableActions(unitC)
Step 6     :      if enemyInRange == false then
Step 7     :         call PrepareMovementAction(unitC, unitP)
Step 8     :       end if
Step 9     :       while unitP != NULL do
Step 10    :         usableActions = GetUsableActions(unitC)
Step 11    :         if usableActions count == 0 then
Step 12    :           Break loop
Step 13    :         end if
Step 14    :       call ChooseAction()
Step 15    :       call UseAction()
Step 16    :     end while
Step 17    :   end for
```

Currently, greedy AI goes through the list of computer controlled units one by one. For every unit, it first finds the closest player controlled unit, and if that unit is out of its attack range, moves its unit to the nearest possible unoccupied space to the player controlled unit. This implementation is very flawed, because if, for example, a player unit is located right outside the firing range of an AI-controlled archer unit, the AI will move the archer very close to the player. Once the AI controlled unit is standing in such a position that it can reach the player's unit with at least some of its actions, it will pick actions at random while it has resources to use them and if the player unit is still alive. If the AI unit kills a player unit and still has AP available, it will move to or attack the next nearest player unit. Below is the pseudo-code of helper methods. Only the helper methods that help the AI make its decision are presented, while other methods that handle other parts of the game, like using an action by a unit on another unit or moving a unit to target coordinate are omitted, as they are not part of the AI algorithm, but of the game's logic as a whole.

Algorithm 2. Finding the closest unit that the computer-controlled unit unitC can attack

```
Procedure  : FindClosestEnemy(unitC)
Input      : Unit unitC
Output     : Unit closestEnemy
Step 1     : Define closestDistance = 10000
Step 2     : Define closestEnemy = NULL
Step 3     :   for each player controlled Unit unitP do
Step 4     :     if unitP != NULL do
Step 5     :       Define distance = Vector3.Distance(unitC, unitP)
Step 6     :       if distance < closestDistance then
Step 7     :         closestDistance = distance
Step 8     :         closestEnemy = unitP
Step 9     :       end if
Step 10    :     end if
Step 11    :   end for
Step 12    : return closestEnemy
```

Algorithm 2 finds the closest opponent's unit to the current computer-controlled unit by calculating the Euclidean distance between them using Unity's built-in methods.

Algorithm 3. Building a list of actions that computer-controlled unit unitC can use

```
Procedure  :  GetUsableActions(unitC)
Input      :  Unit unitC
Output     :  List<Action> usableActions
Step 1     :  Define usableActions = new List<Action>
Step 2     :  for each action in unitC.actions do
Step 3     :    if unitC can pay action cost and (actionRange >= Vector3.Distance(unitC,
              unitC.preferredTarget) or (action.Type == SELF and action.Type == SUPPORT
              and no duplicate action effect on unitC) do
Step 4     :      usableActions.Insert(action)
                end if
Step 5     :  end for
Step 6     :  return usableActions
```

Algorithm 3 simply checks which actions the current computer-controlled unit can use by comparing its resource costs to available resources of the unit and checking whether or not the target is in range of the action or alternatively, if an action has to be used on the unit itself and applies an effect to its user that already is active.

Algorithm 4. Finding the coordinates to which computer-controlled unit unitC should move to reach its preferred target and moving there

```
Procedure  :  MoveToTarget(unitC)
Input      :  Unit unitC
Output     :  -
Step 1     :  if unitP == NULL do
Step 2     :    return
Step 3     :  end if
Step 4     :  define apCostToTarget = 0
Step 5     :  define movementTarget = null vector
Step 6     :  define apCostToTarget = cost to move for unitC to prefferedTarget
Step 7     :  if apCostToTarget > unitC.currentActionPoints do
Step 8     :    define difference = abs(unitC.currentActionPoints – apCostToTarget)
Step 9     :    movementTarget = (Vector3.Distance(unitC, prefferedTarget – difference + 0.05)
              * unitC.movementSpeed) * Vector3.Normalize(prefferedTarget - unitC) +
              unitC.position
Step 10    :  else
Step 11    :    movementTarget = (Vector3.Distance(unitC, prefferedTarget – unitC.width) *
              Vector3.Normalize(prefferedTarget - unitC) + unitC.position
Step 12    :  end if
Step 13    :  movementTarget = FindBestClosestMovablePoint(movementTarget)
Step 14    :  define movementCost = cost to move from unitC to prefferedTarget
Step 15    :  Pay movement cost for unitC
Step 16     :  move unitC
```

Algorithm 4 allows the computer player to select the place where its unit should move to. It checks whether or not it can reach its preferred target during the current turn and then moves to the place right next to its target using the width of the unit's game object to make sure that it doesn't move into its target. If it cannot reach it, it instead moves to the closest place on the line between the two units that its remaining action points can allow. Before moving to the target it also checks if the movement point is blocked or not and if it is it finds the closest available unblocked point that it can move to. This is accomplished by algorithm 5.

Algorithm 5. Finding the closest unblocked coordinate to coordinate movementTarget

```
Procedure   :  FindBestClosestMovablePoint(movementTarget)
Input       :  Vector3 movementTarget
Output      :  Vector3 movementTarget
Step 1      :  define offset = unitC.width
Step 2      :  if no other game object exists at movementTarget and movementTarget is on field
               do
Step 3      :    return movementTarget
Step 4      :  else
Step 5      :    define lowestDistance = 100000
Step 6      :    while lowestDistance == 100000 do
Step 7      :      define targetCoords = Vector3[]
Step 8      :      initialize targetCoords with 8 coordinate points 1 unit away from
                   movemenTarget on x and/or z axis
Step 9      :      for each coordinate in targetCoords do
Step 10     :        if  no other game object exists at coordinate and coordinate is on field do
Step 11     :          define coordinateToCheck = null vector
Step 12     :          if prefferedTarget != NULL do
Step 13     :            coordinateToCheck = prefferedTarget
Step 14     :          else
Step 15     :             coordinateToCheck = movementTarget
Step 16     :          end if
Step 17     :          define distanceFromPointToTarget = cost to move for unitC to
               coordinateToCheck
Step 18     :          if lowestDistance > distanceFromPointToTarget && unitC can move to
               coordinate do
Step 19     :            lowestDistance = distanceFromPointToTarget
Step 20     :            movementTarget = coordinate
Step 21     :          end if
Step 22     :        end if
Step 23     :      end for
Step 24     :    if  lowestDistance == 100000 do
Step 25     :      define coordinateToCheck = null vector
Step 26     :       if prefferedTarget != NULL do
Step 27     :        coordinateToCheck = prefferedTarget
Step 28    :       else
Step 29    :         coordinateToCheck = movementTarget
Step 30    :       end if
Step 31    :     For i = 0 to 7 do
Step 32    :       define distanceFromPointToTarget = cost to move from targetCoords[i] to
                   coordinateToCheck
Step 33    :       if lowestDistance > distanceFromPointToTarget do
Step 34    :         lowestDistance = distanceFromPointToTarget
Step 35    :         coordinate = targetCoords[i];
Step 36    :         if i == 7 do
Step 37    :           lowestDistance = 100000
Step 38    :         end if
Step 39    :       end if
Step 40    :     end for
Step 41    :   end if
Step 42    :  end while
Step 43    : end if
Step 44    : return movementTarget
```

The purpose of this algorithm is to check whether or not the coordinate that was selected by the previous algorithm that the unit should move to is blocked or not. If it's blocked, the algorithm attempts to find the closest unblocked coordinate and select it instead. It checks the 8 nearest coordinates from the target coordinate by 1 distance on the x and/or z axis. If not suitable coordinate is found, it takes one of those 8 coordinates and checks around them, continuing until a point is found.

Algorithm 6. Out of the list of actions that computer-controlled unit unitC can use, choosing one to use

```
Procedure  : ChooseAction(usableActions)
Input      : List<Action> usableActions
Output     : Action chosenAction
Step 1     : define chosenAction = NULL
Step 2     : if effects active on unitC do
Step 3     :   for i = 0 to usableActions.count – 1 do
Step 4     :     if (usableActions[i] is of type SELF) do
Step 5     :       if duplicate effect exists on unitC do
Step 6     :         usableActions.remove(i)
Step 7     :       end if
Step 8     :     end if
Step 9     :   end for
Step 10    : end if
Step 11    : for each action in usableActions do
Step 12    :   if action is of type SELF and enemy is in range and unitC.currentActionPoints
               >=6 do
```

```
Step 13   :       chosenAction = action
Step 14   :       **return** chosenAction
Step 15   :       **end if**
Step 16   :     chosenAction = usableActions[random[0, usableActions.count - 1]]
Step 17   :   **end for**
Step 18   :   **return** chosenAction
```

Algorithm 6 chooses an action for the unit to use. Currently it is a very simple implementation where the action it chooses is chosen at random, unless the action applies a supporting effect on the unit, in which case it is eliminated from the list to prevent an endless loop because using actions with duplicate effects that are already active is not allowed.

2.3.AI testing sub-system

In order to test the effectiveness of AI algorithms developed for this project, a testing ground needed to be created. Prior to this, CubeWars had no means of allowing two computer controlled opponents to play against each other – one of the sides had to be a player character. The testing ground would need to not only allow two computer controlled players to battle, but also control the speed of the simulation in order to observe a large number of games in a short time and also document the results. The use case diagram in Figure 16 shows the expected functionality of this sub-system.

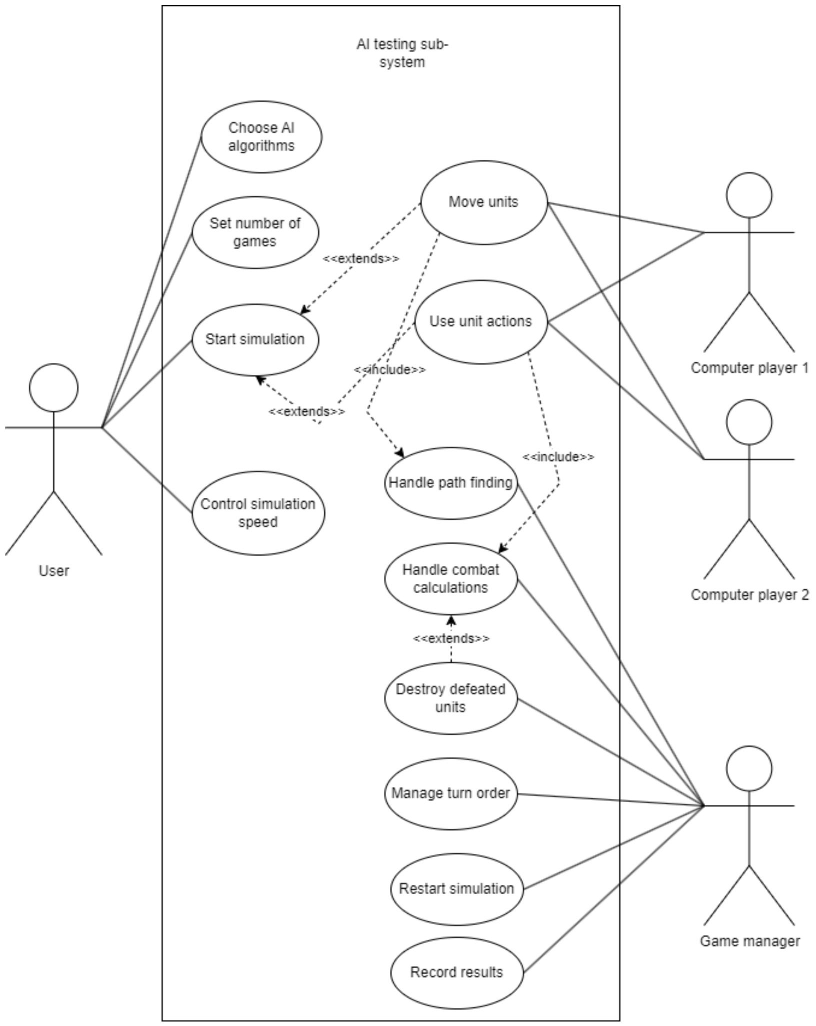


Figure 16. Use case diagram for the AI testing sub-system

The sub-system should allow the user to be able to select, which AI algorithm would play against which and set a number of games that the algorithms should play against each other and then start the simulation. The user should also be able to control the speed of the simulation – at the very least, there should be an option to skip all instances of the game animating the movement of objects in the world or waiting after an action is performed. When the simulation is started the two chosen AI algorithms must play against one another by using the rules and mechanics of the game that were described above. The game manager handles all mathematical operations that happen in the game and shows the results to the user. It also controls the flow of the game – for example, once one of the players finish acting and end their turn, the game manager correctly initiates the turn of its opponent. Once the game ends with the victory of one of the players, the manager must also correctly restart the simulation. Lastly, all necessary variables, such as the win and loss statistics of the AI players, number of games played, the number of turns it took to win the games and others have to be recorded, preferably in a file that can be viewed and analyzed whenever necessary.

2.4.AI system design considerations

Video game AI can be implemented using various ways, some of which were presented in section 1.4. For the development of an AI system for CubeWars, it was decided to use reinforcement learning-based methods. There are several reasons for this. Reinforcement learning seems to be a very popular choice for the creation of decision-making AI in turn-based games –the vast majority of systems that were analyzed used either reinforcement learning or some combination of reinforcement learning with other algorithms in their work. As was mentioned before, turn-based computer games often feature very complex states and rules and as a result become very difficult to solve by computing every possible move, on top of having too many factors to consider in order to write behavior scripts manually [19]. Apart from this, the author of this paper has had his professional internship in a company where he also worked with reinforcement learning AI, so a certain amount of familiarity with the subject has already been attained, which also influenced the choice of this methodology in order to achieve the goal of this paper. In the following sections implementation of two types of reinforcement learning application will be described – raw reinforcement learning, where the system is designed in such a way that the reinforcement learning agent has direct control over all aspects of the game and can make decisions in the same way as a player, and assisted reinforcement learning, where the agent can make a part of the decisions and some of those decisions are abstracted to a higher level strategy or action sequence as was recommended by Amato & Shani [2] and Santoso & Supriana [37]. The first approach could result in a smarter AI, as it would have access to all the actions a real human player would, but there is no guarantee that the agent would be able to understand such a complex game world. The second approach would result in AI that is more limited in its capabilities, but one that would have a higher chance of successfully learning how to play the game.

3. System implementation

The following section details the tools and methods used to implement the described system as well as key decisions that were made and what was created as a result.

3.1.Used technologies

Below the technologies used for implementation of the system, their peculiarities and reasons for choosing them are described.

Unity Engine [40] – Unity is a game engine (a software framework designed for developing games) created in 2005. It offers a vast toolset of working with 3D graphics, navigation and audio and easily integrating them with game logic. Unity allows users to create behavior scripts either in C# or in JavaScript and make these scripts interact with objects in the game world through various methods like “Start”, “Awake” or “Update” inherited from its special classes that are executed automatically at specific times. A diagram showing the workflow of various Unity methods and events is provided in appendix A.

Unity Engine was chosen for this project for numerous reasons, but the most important being that the game has already been created and worked on for a year in Unity, so it made sense to continue using it for this project. Unity Engine is free to use and comes with a large amount of documentation and learning material, making it very attractive for inexperienced game developers.

ML-Agents [42] – a third party open-source toolkit for training and running machine learning AI agents in the Unity environment. By default, Unity Engine is not equipped to handle machine learning so this toolkit was created to give it this capability. ML-Agents provides a bridge between the Unity Engine and PyTorch, a Python-based machine learning and neural network framework. As Unity itself is unable to handle machine learning, data from unity is sent to the Python-based trainer and the output it generates is then sent back to unity, which uses the output to control the game and make changes to the environment. ML-Agents utilizes reinforcement learning and works the following way:

1. An Agent object is put into the Unity environment.
2. The game requests a decision from the Agent object (either on demand or periodically).
3. When a decision is requested, the Agent object collects the necessary data from the environment.
4. The data is sent to the Python-based trainer.
5. The trainer analyzes the data and sends a decision back the Agent.
6. The Agent uses the decision to perform an action. Which action is performed depends on how the decisions are mapped to actions. For example, in a game where the agent can move either left or right, decision “0” can be used to move left and decision “1” can be used to move right.
7. The agent receives a reward based on how the environment changes from its decision. How rewards are calculated and given is completely arbitrary. For example, if the game’s goal for

the agent is to move right, after every action it is possible to award it a negative point for moving left and a positive point for moving right. If the goal is to move to a certain point, then the agent could be awarded after every move based on how close or far it is from that point or it can also be awarded only when it reaches that point.

The ML-Agents toolkit allows training agents using either proximal policy optimization (PPO) or soft actor-critic (SAC) algorithms and also provides a method of implementing support for other algorithms like Q-Learning if the user needs them. It also gives the ability of changing the various parameters of the algorithms by editing a YAML configuration file. Apart from regular reinforcement learning, the toolkit allows training agents from recorded games through imitation learning. ML-Agents was chosen for this project because it is the only known toolkit for handling machine learning in Unity without needing to program the implementation yourself.

PyTorch [27] – a Python machine learning framework used by many other machine learning libraries and tools, including ML-Agents. Its primary purpose is performing complicated calculations using data flow graphs, which is how neural networks are represented in computers. The PyTorch framework is not interacted with individually and is mainly used because it is a required dependency for ML-Agents.

TensorBoard [38] – a visualization toolkit for machine learning and part of the larger TensorBoard library for Python. It is used to generate graphs of machine learning agents' performance. The ML-Agents toolkit is integrated with TensorBoard and generates statistics files that can be opened and viewed in it, which makes TensorBoard the obvious choice for a visualization tool.

3.2.Integrating ML-Agents with the game environment

Prior to starting work on integrating the ML-Agents toolkit with the game, two simple environments were created to better understand the working principles and peculiarities of ML-Agents.

The first environment was a simple game where a cube and a ball were put on flat, rectangular shaped plane surrounded by walls. The cube was assigned to be the agent and the ball – the goal. The agent was given the ability to change its X and Z coordinates equal to the value of the decision it made, which was coded as a 2 floating point numbers in the range [-1; 1], corresponding with the changes in the X and Z axis. The agent was rewarded with 1 point for colliding with the ball object and -1 point for colliding with a wall, after which the simulation was restarted. The environment is shown in Figure 17.

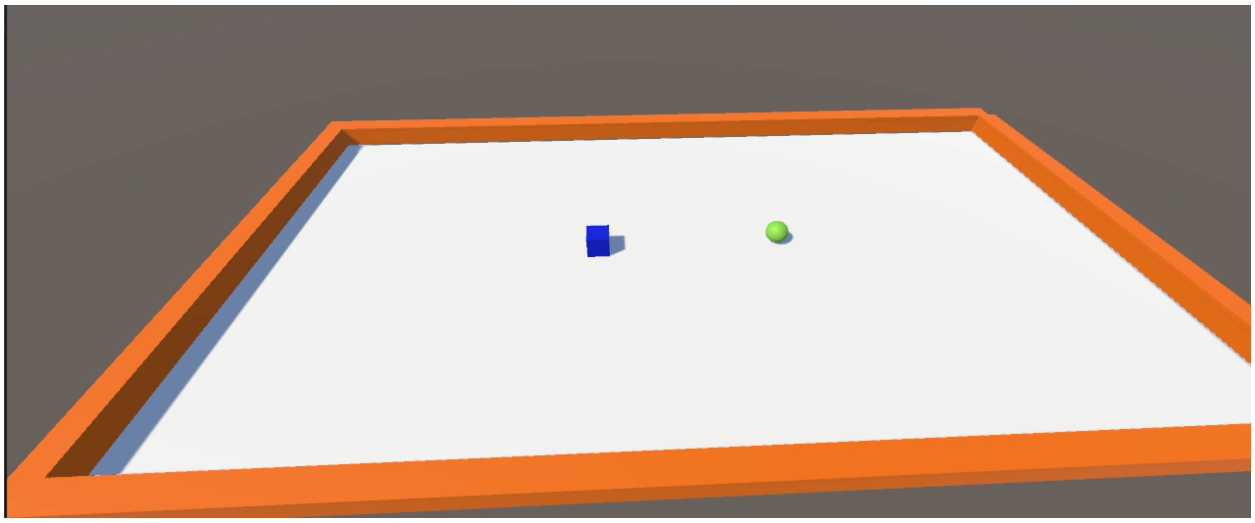


Figure 17. Simple 3D game environment.

The simulation was run until a peculiar behavior was spotted – the agent initially tried to move around but after some point it just started moving in place. This happened, because it learned what it believed to be the optimal policy – because it was unable to find the goal (or found it very rarely) but frequently hit the walls, for which it received a penalty, it concluded that the optimal course of action was to simply move in place and avoid getting negative points.

From this observation, the environment was adjusted in the following way: the goal was moved closer to the agent and a time limit was set – if the agent would be stuck in place for too long, the episode would also end with a negative score to the agent. With these changes, the agent managed to reach the goal more frequently and started doing it consistently after 100000 training steps.

After this another experiment was done – the goal's coordinates were changed while the agent was moving to it, but instead of moving to the goal, the agent kept moving to the place where the goal always appeared during training.

The experiment was restarted again, this time the position of the goal was randomized in a small area around its initial position. After 100000 training steps the agent once again was able to reach the goal consistently and moving the goal while the agent was moving to it caused the agent to alter its trajectory. However an interesting behavior was observed: during training the goal always appeared on the right side of the agent, so while the agent was moving to the goal, if the goal were suddenly moved to the left, the agent would start moving in place. The agent was retrained in an environment where the goal could appear on either side, as well as above or below the agent and only then it learned to correctly move and reach the goal in any direction. This coincides with the recommendation given by Sestini, et. Al. [32] that the agent should be exposed to as many different states as possible because if it never acts a certain way during training, its performance will be sub-optimal.

Afterwards a second simple environment was created where the agent had to escape from another moving object for as long as possible, but this time the game was turn-based – every time an agent acted, the other object would act as well and then control was passed to the agent. This environment was created simply to test how ML-Agents performs in a turn-based setting and training the agent to achieve a certain goal was not the primary concern.

After these experiments, ML-Agents had to be integrated into the already quite complicated architecture of CubeWars. A typical game of CubeWars involves several game objects called Units, which can be of several types. These units can be controlled by a human player or the computer. When a player gives a command to the unit, various manager classes handle all the necessary game logic, like calculating route to destination or calculating outcome of a combat action. The game flow is controlled by a state machine consisting of several preparation states and game loop states. The game starts in the menu state, during which the user selects initial game parameters and then moves into the deployment state where the user selects which units it wants to use for the game. Then the main game loop starts, consisting of the turn states for both players, the processing states that happen after a turn is finished, which are necessary to handle post-turn effect calculation and game object list updates and the end game state, which displays the result of the game once it finishes and offers the ability to restart. How AI works in this setting is that when its turn begins, the algorithm will simply iterate over a list of units it controls and keep issuing orders to them while the units still have enough action points to act and afterwards end its turn. This process is visualized in Figure 18.

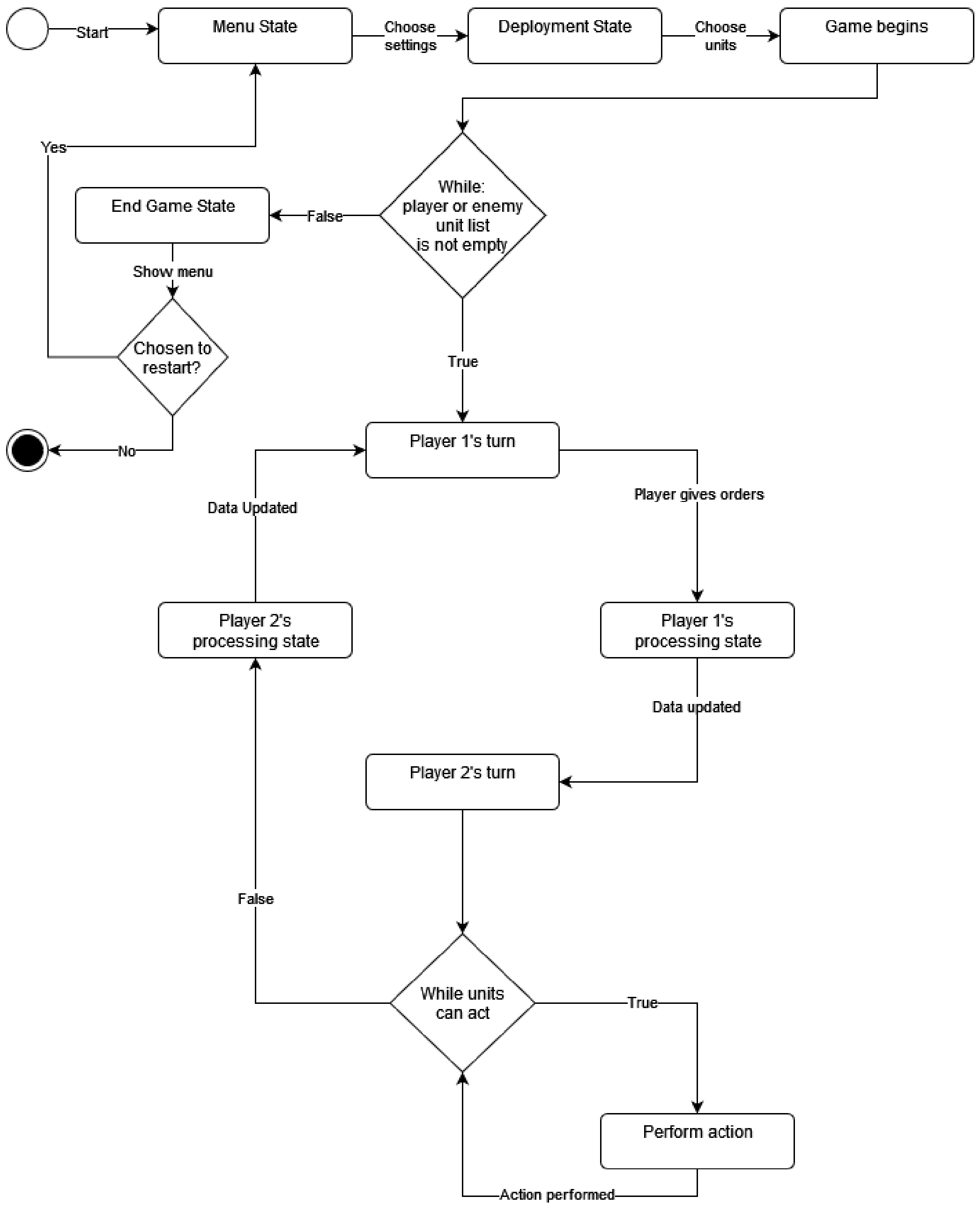


Figure 18. CubeWars game flow and states.

The above diagram assumes that player 1 is a human and player 2 is computer controlled. The addition of Unity ML-Agents complicates the process, as it is now necessary to adjust the game flow in such a way that 2 AI algorithms could play against one another, so that the learning agent could train. The agent runs in a separate Update loop and expects to receive decision request signals that automatically trigger the process of collecting data, sending it to the trainer, receiving the best action and then taking that action. Because CubeWars has certain rules that must be followed, it is also necessary to forbid the agent from taking actions the game doesn't allow (for example, attempting to move to a non-existent space or attack an enemy that is out of range). Fortunately, ML-Agents comes with the ability to set filters on actions through an automatic listener method that is ran before an action is taken. Figure 19 shows the updated game flow that involves 2 computer-controlled players playing against each other, one or both of which can be controlled by ML-Agents. Because the scenario when both player 1 and player 2 are computer-controlled does not involve a deployment phase and the game restarts automatically when one of the players wins, other parts of the process are appropriately changed.

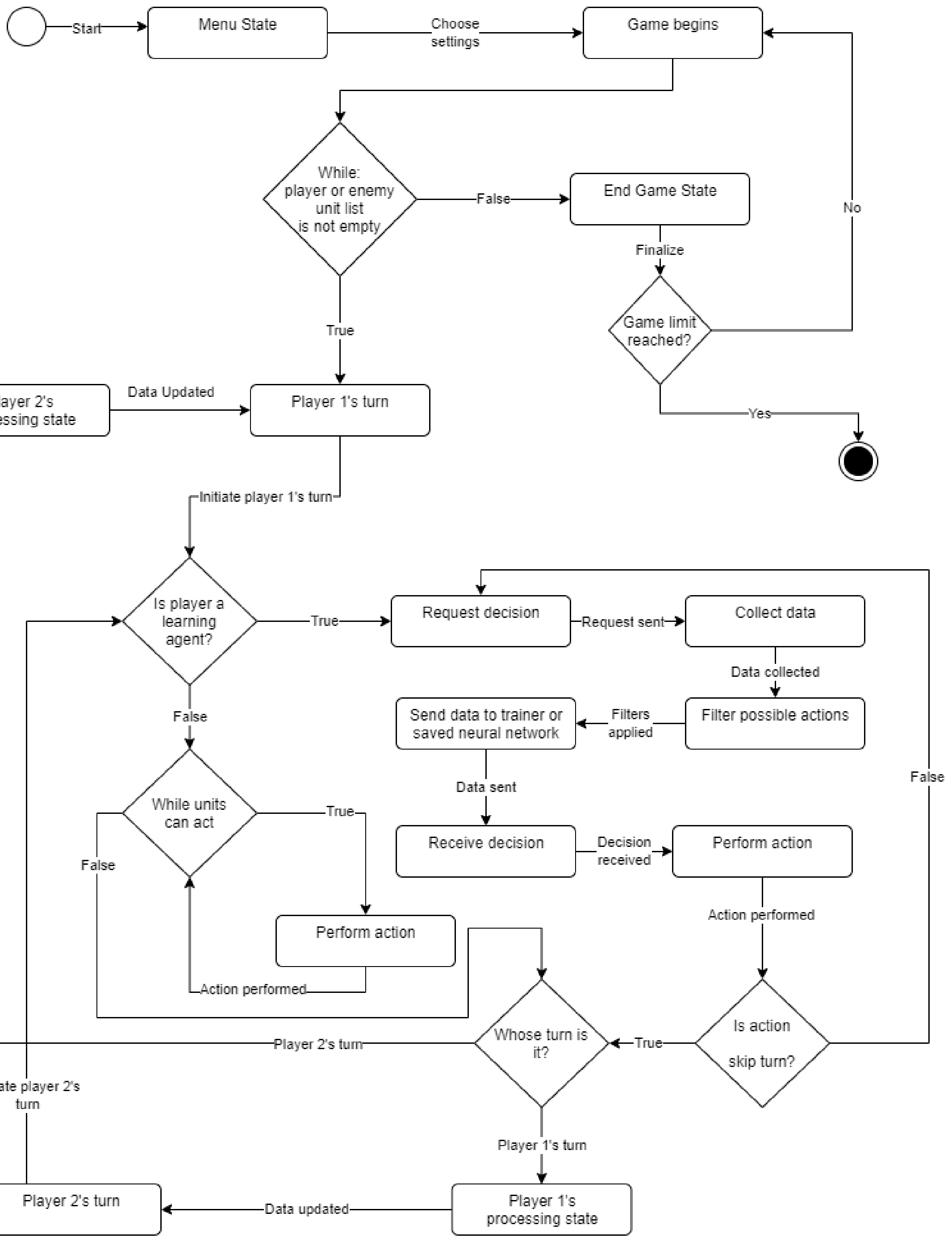


Figure 19. CubeWars game flow and states with ML-Agents.

Furthermore, an additional state manager had to be created to handle the training process specifically. It functions similarly to the main game state manager, but also handles additional control related with resetting the training environment, allocating rewards and other machine-learning related activities. It was added in order to not overload the main game state manager with additional states. Also, because the implementation of ML-Agents based AI turned out to be completely different from how the static AI algorithms were programmed in CubeWars, a change in system architecture had to be made: prior to this, all AI algorithms inherited from the class AIAlgorithm that contained different shared methods that allowed the computer-controlled player to interact with the game world. However, ML-Agents AI classes need to inherit from a special class Agent in order to get access to the necessary methods for collecting observations and performing actions. Because C# does not allow classes to inherit from more than one parent, the system had to be updated to correctly handle the turns of two types of AI algorithms. The UML diagram in figure 20 shows the interactions between the different AI-related classes and the game manager classes.

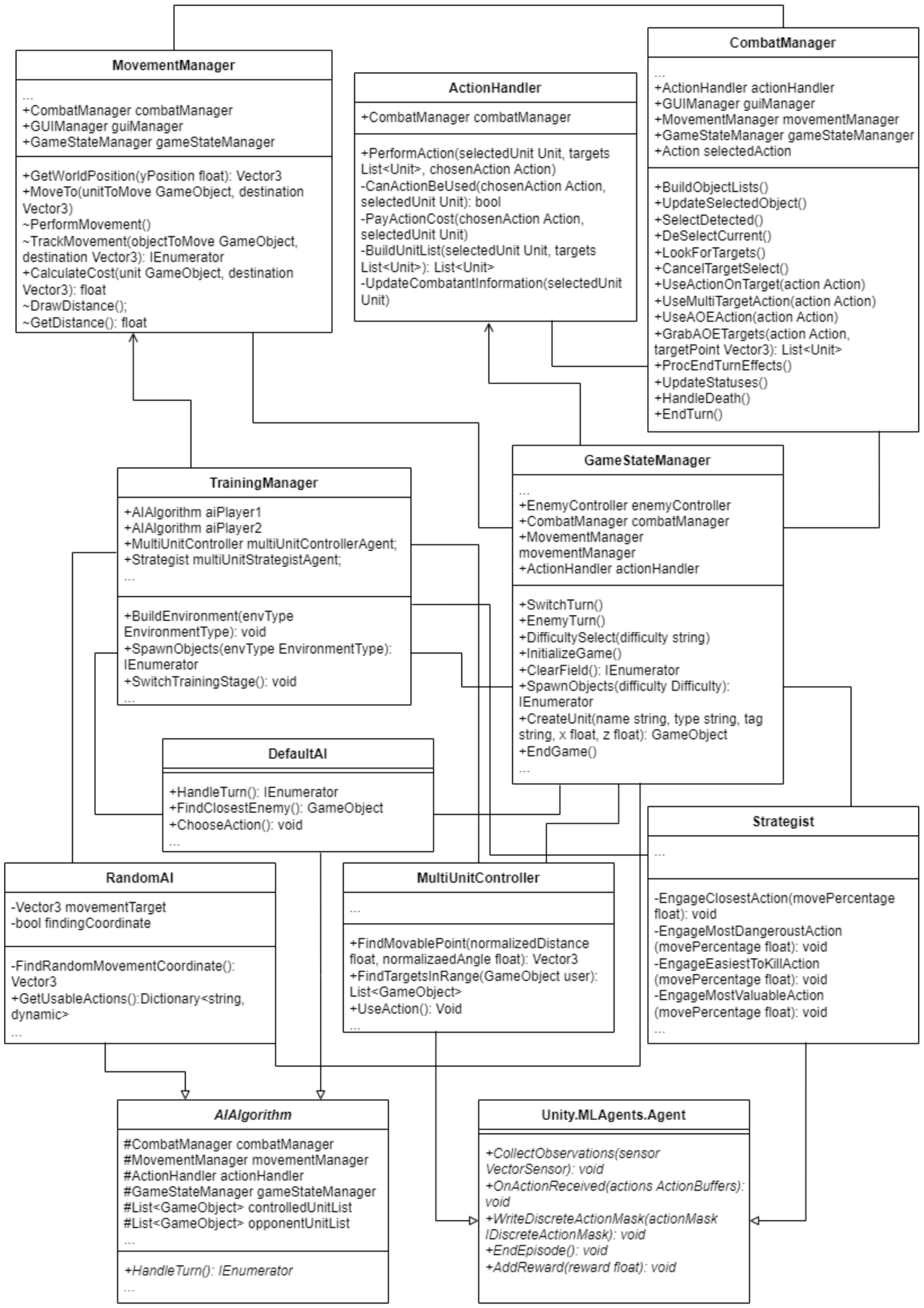

MovementManager
...
+CombatManager combatManager
+GUIManager guiManager
+GameStateManager gameStateManager
+GetWorldPosition(yPosition float): Vector3
+MoveTo(unitToMove GameObject, destination Vector3)
~PerformMovement()
~TrackMovement(objectToMove GameObject, destination Vector3): IEnumerator
+CalculateCost(unit GameObject, destination Vector3): float
~DrawDistance();
~GetDistance(): float
ActionHandler
+CombatManager combatManager
+PerformAction(selectedUnit Unit, targets List<Unit>, chosenAction Action)
-CanActionBeUsed(chosenAction Action, selectedUnit Unit): bool
-PayActionCost(chosenAction Action, selectedUnit Unit)
-BuildUnitList(selectedUnit Unit, targets List<Unit>): List<Unit>
-UpdateCombatantInformation(selectedUnit Unit)
CombatManager
...
+ActionHandler actionHandler
+GUIManager guiManager
+MovementManager movementManager
+GameStateManager gameStateMananger
+Action selectedAction
+BuildObjectLists()
+UpdateSelectedObject()
+SelectDetected()
+DeSelectCurrent()
+LookForTargets()
+CancelTargetSelect()
+UseActionOnTarget(action Action)
+UseMultiTargetAction(action Action)
+UseAOEAction(action Action)
+GrabAOETargets(action Action, targetPoint Vector3): List<Unit>
+ProcEndTurnEffects()
+UpdateStatuses()
+HandleDeath()
+EndTurn()
TrainingManager
+AIAlgorithm aiPlayer1
+AIAlgorithm aiPlayer2
+MultiUnitController multiUnitControllerAgent;
+Strategist multiUnitStrategistAgent;
...
+BuildEnvironment(envType EnvironmentType): void
+SpawnObjects(envType EnvironmentType): IEnumerator
+SwitchTrainingStage(): void
...
GameStateManager
...
+EnemyController enemyController
+CombatManager combatManager
+MovementManager movementManager
+ActionHandler actionHandler
+SwitchTurn()
+EnemyTurn()
+DifficultySelect(difficulty string)
+InitializeGame()
+ClearField(): IEnumerator
+SpawnObjects(difficulty Difficulty): IEnumerator
+CreateUnit(name string, type string, tag string, x float, z float): GameObject
+EndGame()
...
DefaultAI
+HandleTurn(): IEnumerator
+FindClosestEnemy(): GameObject
+ChooseAction(): void
...
Strategist
...
-EngageClosestAction(movePercentage float): void
-EngageMostDangeroustAction (movePercentage float): void
-EngageEasiestToKillAction (movePercentage float): void
-EngageMostValuableAction (movePercentage float): void
...
RandomAI
-Vector3 movementTarget
-bool findingCoordinate
-FindRandomMovementCoordinate(): Vector3
+GetUsableActions():Dictionary<string, dynamic>
...
MultiUnitController
...
+FindMovablePoint(normalizedDistance float, normalizaedAngle float): Vector3
+FindTargetsInRange(GameObject user): List<GameObject>
+UseAction(): Void
...
AIAlgorithm
#CombatManager combatManager
#MovementManager movementManager
#ActionHandler actionHandler
#GameStateManager gameStateManager
#List<GameObject> controlledUnitList
#List<GameObject> opponentUnitList
...
+HandleTurn(): IEnumerator
...
Unity.MLAgents.Agent
+CollectObservations(sensor VectorSensor): void
+OnActionReceived(actions ActionBuffers): void
+WriteDiscreteActionMask(actionMask IDiscreteActionMask): void
+EndEpisode(): void
+AddReward(reward float): void

Figure 20. CubeWars UML diagram for AI-related classes.

3.3.Training the agent in a simplified environment

At first the agent was trained in a simplified version of the CubeWars environment in order to make sure that it interacts with the game correctly. In the simple environment shows in figure 21, the agent was given the ability to control only 1 unit and the enemy also had 1 unit, thus the decision the agent could make involved selecting where to move on the field and which action to use. Because there were only 2 units on the map, it didn't have to consider which unit to select and which target to attack.

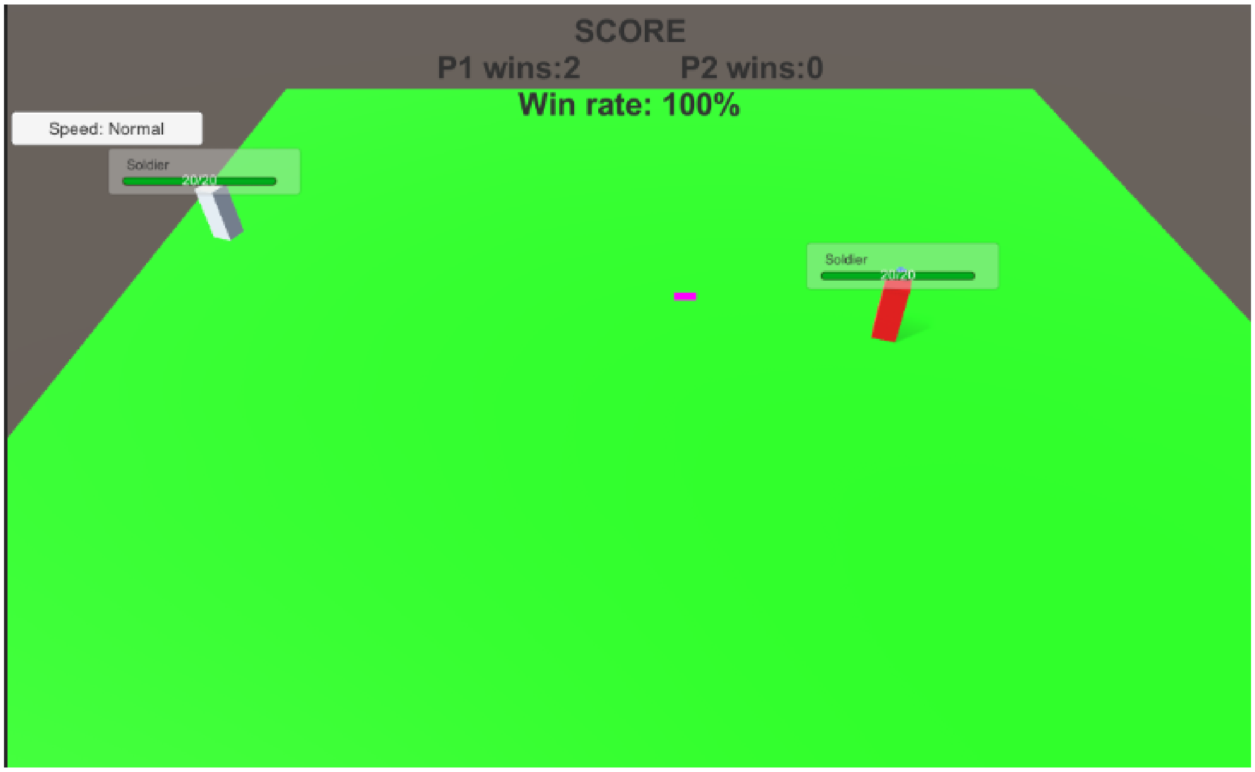


Figure 21. CubeWars 1-versus-1 training environment

In this environment, the agent had access to the following observations:

- All relevant unit variables: their position on the field, characteristics (hit points, energy, damage and others).
- Distance between the agent's unit and the enemy's.
- The coordinates that signify the field's border, beyond which movement is impossible.

When making a decision, the agent passes the observations provided through its neural network and output a series of numbers that correspond to the actions it wants to take. ML-Agents supports two types of actions: discrete and continuous. Discrete actions are encoded as integer numbers, while continuous actions are floating point numbers in the range of -1 to +1. When the agent is set up, it must be provided an action space, which represents the type and number of different actions it is allowed to take. ML-Agents allows the action space to include many different continuous numbers and multiple arrays of discrete numbers. Newer version of ML-Agents also allow mixing the action spaces.

In this environment, the agent was provided with an action space consisting of 1 array of 6 discrete actions and 2 continuous actions. The actions interact with the game in the following way:

- Action 0 makes the agent end its turn
- Action 1 makes the agent move its unit

- Actions 3-6 make the agents use one of four abilities of its unit. Currently in CubeWars, every unit has 4 abilities, except for the archer, which has 3

If the agent chooses to move its unit, the coordinate of the move command is determined by the values of the 2 continuous actions. Continuous actions in ML-Agents are always provided in the range of [-1, 1], so they need to be translated into a valid point on the field. Because currently movement in CubeWars only happens on a flat plane, only 2 coordinates are needed to find this point – X and Z. Knowing the origin point, these can be given by the formulae:

$$X = Xo + \sin\alpha * d, Z = Zo + \cos\alpha * d$$

Where Xo and Zo are the X and Z coordinates of the unit's original position, α is the angle of movement and d is the distance, representing how much from the original position the unit should move. The two continuous values received from the agent can be converted into a distance value in the range of [0, max], where max is the product of the unit's current action points and is movement speed and an angle value in the range of [0, 360]. Knowing the minimum and maximum value of those variables, they can be calculated using the formula:

$$newVal = \frac{(oldVal - oldMin) * (newMin - newMax)}{oldMin - oldMax} + newMin$$

When ML-Agents returns a decision, it always gives one number from every discrete action array and one number for every action, so in this case it outputs 3 numbers. If its discrete action is 1, then it would move to the coordinate calculated from the provided 2 continuous numbers. In every other case, the continuous numbers are not used.

Initially, the agent was rewarded for killing the enemy and penalized for using illegal actions (for example, trying to attack an enemy out of range), similarly to how it was done in Sestini, et. Al. work [32].

Initially, training results were quite poor – because the agent received negative rewards for invalid moves, it learned to not do anything rather than actually try and fight the enemy, so negative rewards for invalid moves were removed (except for negative rewards for trying to move off the game field) and action filters were added that prevented the agent from being to use actions when conditions were not met. This change resulted in better performance. Initially the agent was trained against the greedy algorithm (one that always tries to run towards and attack the closest enemy) and it learned to beat it reliably, but because none of the training parameters were randomized and the opponent algorithm always performed the same action, the agent just learned to stand in place, wait for the enemy to come into range and then use the most damaging actions to kill it, however it couldn't play well in any other scenario – for example, if the enemy was placed right next to its unit at the start of the game, it still skipped the first few turns (because it did it every time during training) and if the enemy was given a long-ranged unit, the agent would never move its unit from its starting place, even as the enemy was attacking it from afar.

Training was the attempted several more times while modifying various parameters, like adding randomization of starting positions and unit types. The agent was then trained against the algorithm that makes random moves in order to be exposed to more different states. Initial results showed an interesting behavior – while the agent now learned to recognize different positions and unit types, it still refused to move to the enemy's unit. This behavior made sense, because the agent was only rewarded for winning and the episode ended when either the agent or the enemy won. So the agent had no reason to do anything but wait until the enemy randomly moved into its attack range and then attack and defeat it.

Following this experiment, a time limit was introduced and the reward function was changed. Now the agent had 30 turns to kill the enemy or the episode was restarted and the reward it gained for killing the enemy was now equal to the remaining turns it had divided by 30 – the agent would gain 1 point if it killed the enemy on the first turn, and 0.5 points if it killed it on turn 15. An additional condition was also introduced – now at the beginning of the training the enemy would start with lower health points and the number would increase as the training progressed.

However, even after these changes the agent still could not learn to move to the enemy in order to attack it. It understood how to attack and would kill the enemy if it ever came close, but when the enemy was far away from it, it had difficulty moving to it using the coordinate system. Its behavior evolved like this: at first it was always moving to random positions but eventually started moving towards one of the field's edges and either staying there or hovering around that position. This behavior is shown in figure 22, where the white cuboid is the agent's unit, the red cuboid is the enemy and the white spheres show the places where the agent tried to move recently.

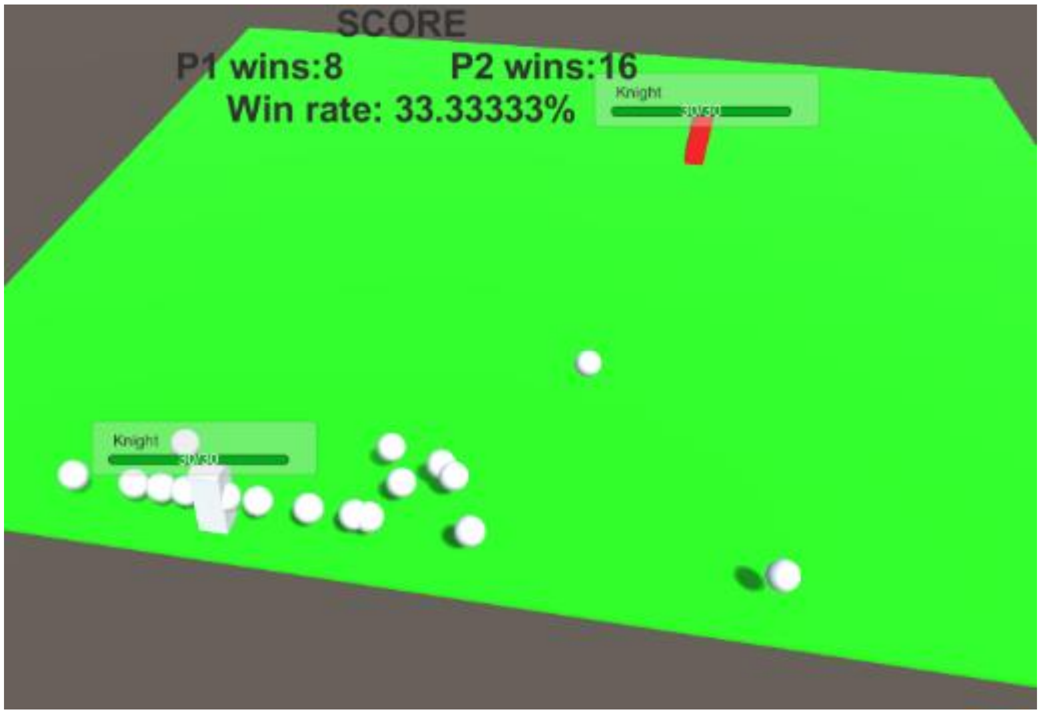


Figure 22. Agent's movement visualization.

Further experiments using different hyper-parameters, like increasing the size of the neural network, changing randomization level and learning rate, training for longer periods of time, however the results were still the same. Both PPO and SAC algorithms that ML-Agents provides were also tested, with SAC resulting in slightly more competent behavior, but it still moved into walls most of the time. Figure 23 show 2 graphs that show the average episodic rewards under same training conditions achieved by the PPO and SAC algorithm. SAC training was terminated earlier because it trained much more slowly than PPO and did not show much improvement with tine.

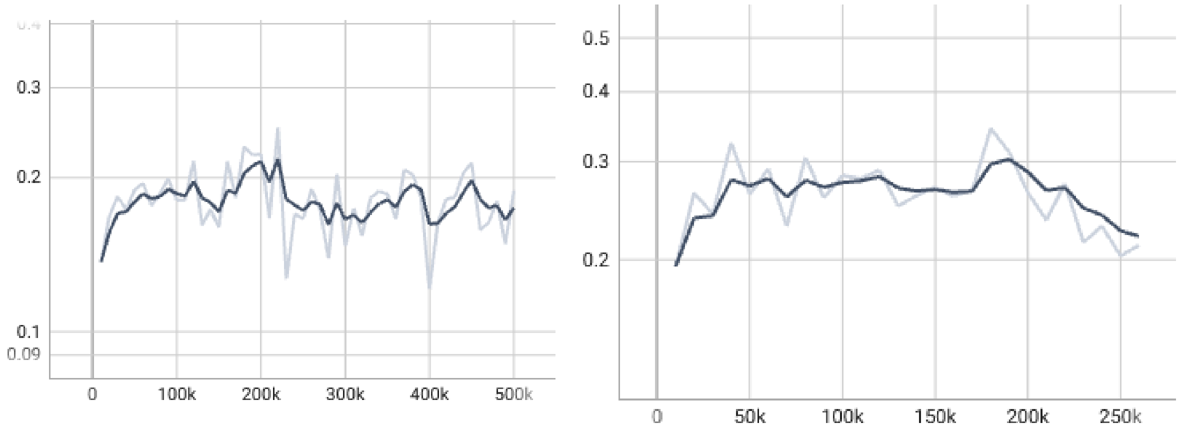


Figure 23. Agent's training result using PPO (left) and SAC (right)

After the last experiment the agent was given an additional discrete action to use. That action simply moved it as close as possible to the enemy. With the addition of this discrete action, the agent learned the progression of first moving to the enemy then attacking it and was able to achieve much higher reward, however as a consequence it never learned to use free movement inside the coordinate space. Figure 24 shows two graphs showing the average episodic reward of the two agents with identical parameters – the left hand graph is the performance of the agent that could only moving using the discrete and continuous action combination and the right hand graphs shows the agent that could use the action that moved it closer to the target.

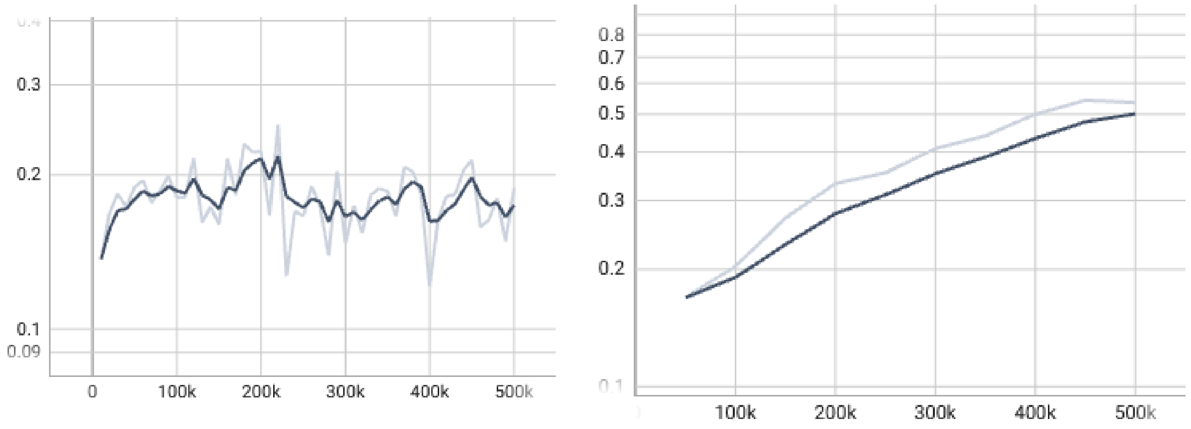


Figure 24. Agent's training result without and with the ability to move to target directly

These experiments showed that even though the agent managed to interact with the environment of CubeWars, it had difficulties learning to play even in the simplified version of it, when it was offered full control over the game (ability to move anywhere on the field and use any action), so in the full environment, where the agent also has to control different units and choose which enemy to attack, in order to improve performance its task would need to be simplified – either by abstracting actions into higher level strategies, as suggested by Santoso & Supriana [37] and Amato & Shani [2] or changing the reward function so that the agent would be rewarded for smaller actions.

3.4.Training the agent in the full environment

In the full CubeWars environment, the complexity of the actions the agent could take increases significantly. Now it has to consider not only where to move and which action to use, but also which of its unit to control and which enemy should it attack. In order to act in such an environment, the agent's observation and action spaces were changed. When neural networks are trained with a certain amount of observation inputs, they always expect the same number of inputs when operating in different environments. For example if an agent is trained in an environment where there are only 2 units, and each unit consists of 20 observable parameters, if placed in an environment where the number of observable parameters is not 40, it will be unable to act. ML-Agents provide the ability to give agents variable length observations with an upper bound of observable entities and a set number of parameters per entity. For example, if an upper bound is set to 10 and each entity has 10 observations and there are only 5 entities in the environment, the agent will be provided 50 observations and the missing observations will be padded with zeros and ignored by the neural network. However, it will still be unable to process an environment with more than 10 observable entities.

The observation for the agent was changed to 5 static variables, representing the edge coordinates of the map and the ID of the currently selected Unit and up to 20 variable observations each consisting of 21 values – the relevant unit information, like position, ID and other characteristics. Because the agent would not be trained in environments where there are more than 10-15 units, an upper bound of 20 was set, allowing to have up to 10 units on each team. Another variable length of up to 4 observable components with 11 values each was added to represent the units' actions.

Two different action space variants were created. The first one is identical to the one described of the simple environment except that another discrete array is added to indicate the target chosen by the agent. Because the agent must first collect the observations then filter possible actions and then choose from the remaining actions, in the filtering steps the agent checks which enemy targets are available within the attack range of its unit and enables as many options in the second discrete array as there are possible targets. Then it is able to choose one of its unit's actions and it will be used on the target chosen from the other discrete array.

It was decided to not let the agent choose units, because this overcomplicates the filtering process – the action filter must be applied before the action is chosen, and if the agent chooses its unit as part of its action, then during the filtering stage there is not enough information to correctly apply the filters. Thus the unit choice is made for the agents. This makes the agent unable to play the game on the same level as a real player, but it ensures the correct flow of information between its operation stages and simplifies the environment, which makes it easier for the agent to solve. This approach was recommended by Kimura & Kokolo [19].

The second variant was created based on the recommendations of Amato & Shani [2] and Santoso & Supriana [37] and observations made during the experiments in the simplified environment. The action space was turned into a single discrete array of 16 actions, which include the individual combat actions of the unit, the action that skips the current unit's turn, the action that delays the current unit's turn and 10 positioning strategies, which replace the agent's ability to freely move in the game world. The positioning strategies involve allowing the agent to approach certain enemy units based on their relative characteristic on the field. 4 main strategies were created:

- Engage closest unit. This makes the agent attempt to come into range of the closest enemy unit. The closest unit is considered the unit, which the current unit would spend the smallest amount of action points to reach and it is found using the following formula:

  $$CU = min\left(Path\big((x,z)(x_i,z_i)\big)\right), \forall_i, i = [0, \dots k]$$

  Where *Path* is the function used by Unity to calculate the shortest path between two points given by *x* and *z* coordinates and k is the number of opponent's units on the field.
- Engage most dangerous unit. This makes the agent attempt to come into range of the enemy unit that is considered the most dangerous. The unit's danger rating is calculated by the formula:

  $$DR = c - \frac{c}{3} * \frac{ec}{em}$$

  Where *c* is the unit's deployment cost, which roughly corresponds to its strength relative to other units, *ec* is current energy and *em* is maximum energy. The unit with the highest danger rating is considered the most dangerous unit among *k* units that are on the field.

  $$DU = \max(DR_i), \forall_i, i = [0, \dots k]$$

- Engage the unit that is easiest to kill. This makes the agent attempt to come into range of the enemy unit that it is most likely to kill. This is determined by the unit's survivability rating, which is calculated as follows:

  $$SR = ch * \left(1 + \frac{e}{100}\right) * (1 + a) + d$$

  Where *ch* is the current health of the unit, *e* is its evasion rating, *a* is its armor rating and *d* is the distance to that unit. The unit with the lowest survivability rating among *k* units on the field is considered the easiest to kill.

  $$LSU = \min(SR_i), \forall_i, i = [0, \dots k]$$

- Engage the most valuable unit. This makes the agent attempt to come into range of the enemy that is considered the most valuable target. The most valuable unit is determined by the value rating, which calculated by dividing the danger rating of the unit by its survivability. The unit with the highest value rating is the most valuable unit.

  $$VU = \max\left(\frac{DR}{SR}_i\right), \forall_i, i = [0, \dots k]$$

Each of these strategies is transformed into 2 discrete actions – one makes the agent move as far as it is able to towards its chosen target, the second only moves a part of the full distance. The remaining 2 actions are fall back, which makes the agent move a certain distance away from the closest enemy, and regroup, which makes the agent move to an ally.

The reward functions for the 2 variants are also different: for the direct control variant, the reward is given after every action depending on how much damage the agent‘s unit inflicted on the enemy and whether or not the action applied any additional effect. For the strategy-using variant, the reward is given after the agent wins or loses the round and is equal to the difference between the remaining HP percentages of its and the enemy‘s units.

Both variants were trained in a curriculum environment where the units the players control and their position is randomized every game and additional training parameters are changed every 5000 - 10000 games – the algorithm the agent is training against is switched from Random to Greedy, the unit spawn position is switched from mirrored (the units for the player‘s are spawned close to each other and each player‘s units start opposite from each other) to random (the units are spawned anywhere on the field) and the unit composition is switched from balanced (both players get an equal number of units of the same types) to random (both players can get any units).

3.5.Training results

Firstly, the variant of the agent that could issue direct movement and action commands was trained for 1.5 million steps, totaling at approximately 5.7 hours. While it did improve its performance over time, it ran into the same problem where it tried to move off the field with its units, so they stayed mostly in one place. It did learn to reliably use abilities to attack the enemy, but could not effectively position its units. As a result it could effectively perform versus the random algorithm, but performed poorly versus the greedy algorithm. Figure 25 shows the agents average reward over time. It was trained in the environment where it first trained versus the random AI but then switched to the greedy AI at around the 500 thousand step mark. The sudden increase at the end happened because the environment automatically switched to the next training stage, where all units were scattered randomly around the map, which was easier for the agent to operate in

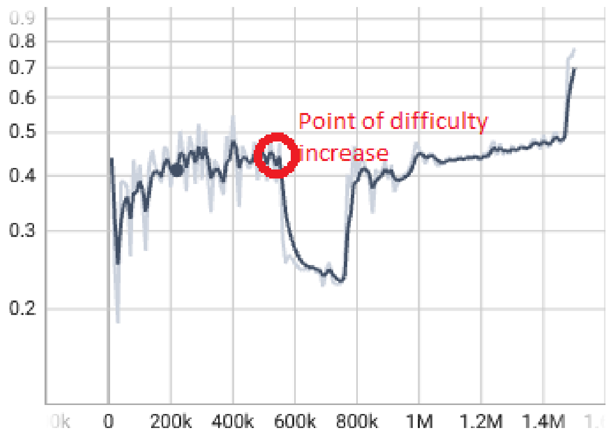


Figure 25. Agent’s average episode reward during 1.5 million training steps

From the results it was concluded that this agent would need a complete rework in order to function properly in this environment, so due to time constraints attention was focused to the strategy-using agent. Several iterations of the strategy-using agent were trained. The first iteration of the agent was trained for 1.5 million steps (approximately 4.3 hours). During the first 400 thousand steps the agent was trained against the random algorithm on a field where each side controlled 3 units, until 1.2 million training steps it trained against the greedy algorithm and towards the end of the training the unit count was increased. Figure 26 shows how the average episodic reward of the agent changed over time. Note that the values on the reward axis differ significantly from the previous agent – this is because the strategy-using agent has a different reward function with the possibility of gaining negative reward.

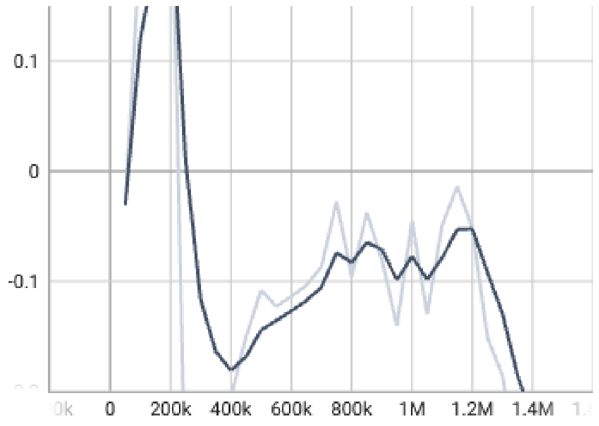


Figure 26. Agent's average episode reward during 1.5 million training steps

A problem was observed with how the agent used the strategies – because it could spend an action to select an enemy unit using one of the "engage" strategies even if it was already in range of the unit, it started wasting a lot of training steps simply switching between targets, which significantly slowed down the training – the average reward barely changed between the 400 thousand and 1.2 point mark. Additional constraints were added to prevent the agent from standing in place and switch between targets. Figure 27 shows the results with these changes:

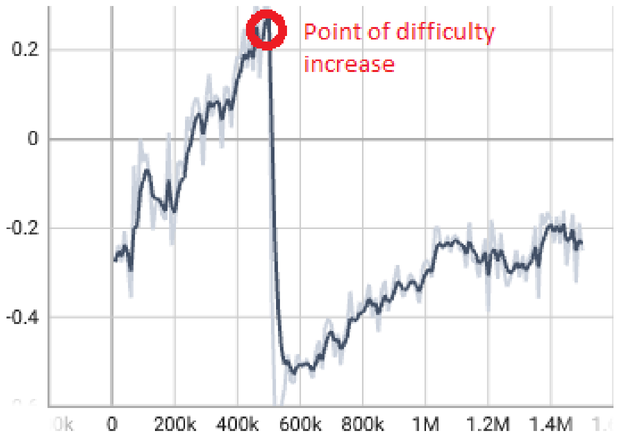


Figure 27. Updated agent's average episode reward during 1.5 million training steps

This time, the difficulty was increased only once – from random opponent with 3 units to greedy opponent with 5 units. This time, the agent showed very slow, but gradual increase all the way to the end of the training steps. Towards the end training also started slowing down, because of the wait action – the agent was using it to delay the turns of its units without any meaningful strategic benefit. Figure 28 shows the same agent training for an additional 500 thousand steps. The reward continues increases, although at a much slower rate, because it spends too many actions just waiting and not changing the environment's state in any way. The total training time was approximately 7.1 hours.

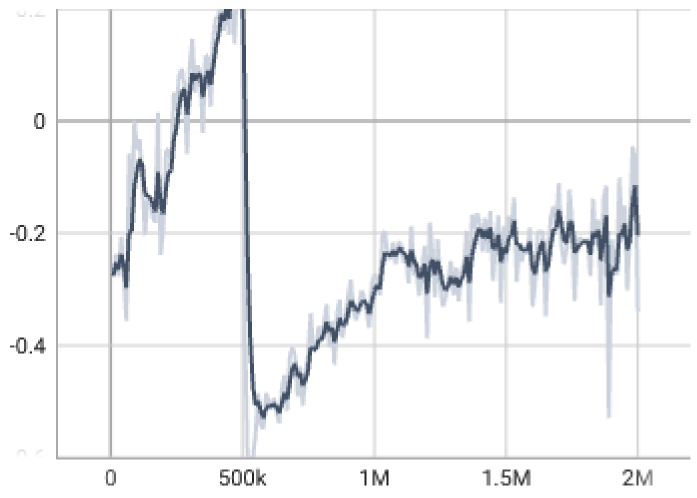


Figure 28. Additional 500 thousand training steps of training

It was decided to remove the wait action – this limits the agent's capabilities but slows down training. Another change that was made is that the behavior of the strategy-using agent was further simplified by changing its ability to use the units' actions – instead of individual actions, it could select from one of 6 action strategies:

- Full attack – spend all of its action points using the unit's normal attack
- Buff then attack – first apply an enhancing effect on itself, then attack with the remaining action points
- Attack using energy – spend all of its actions using the unit's offensive actions, prioritizing those that cost energy
- Apply DoTs – use actions that apply a damage over time effect
- Use AoE attack – use actions that have an area of effect (can target multiple enemy units in a radius)
- Recover – use actions that restore the unit's health

The resulting updated agent had an action space of 17, composed of 10 movement and 6 action strategies and 1 action to skip the turn.

This agent was trained for 2 million steps against greedy AI exclusively, because with an action space abstracted this way, it would have a much easier time finding and engaging the units of random AI even if not exposed to it. It was trained in matches of 5 versus 5 units, increasing to 7 versus 7 at approximately 500 thousand training steps. Figure 29 demonstrates the training results.

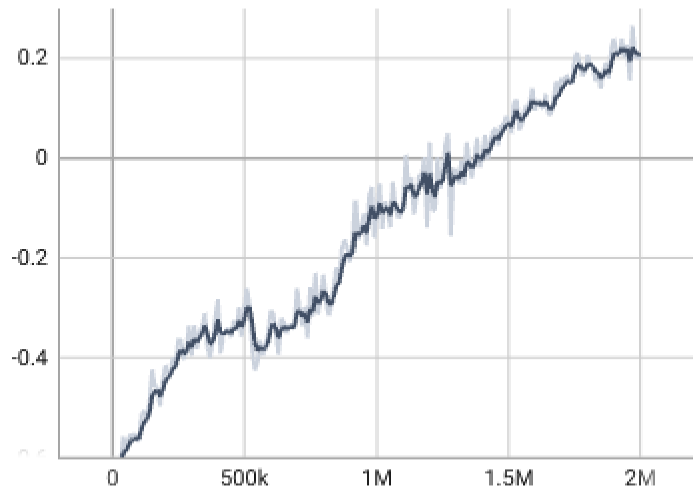


Figure 29. Final iteration agent's average episode reward during 2 million training steps.

As can be seen, the results were much better. The agent achieved a consistent positive average reward during training, which meant that it was now winning against the greedy algorithm more often than losing.

All of the agents were trained using the PPO algorithm. Even though SAC performed slightly better than PPO in the simple 1 versus 1 environment, SAC training took too much time so it was decided to focus only on PPO.

3.6.Evaluating performance

The three resulting agents: Controller Agent, which had the ability to use all the game's actions directly, Strategist Agent, which could use 10 movement strategies and the unit's actions directly and Simplified Strategist Agent, which could use the movement strategies together with 6 new action strategies, were evaluated in the testing environment by playing 100 games each against the random and greedy algorithms. The evaluation environment is a 7 versus 7 game on a square field with no obstacles. Both players control the same composition of units and start far across. The results are presented in table 4. The simulation where the Controller agent played against the random algorithm was stopped after 10 games, because it took the agent too much time to find the random agent's units

Table 4. Agents' Evaluation results

Evaluation	Percentage of games won	Average remaining HP	Average opponent remaining HP	Average game length (in rounds)
Controller vs. Greedy	19%	6.44	39.69	7.57
Controller vs. Random	100%	88.5	0	115.4
Strategist vs. Greedy	25%	11.88	35.61	4.61
Strategist vs. Random	96%	46.09	1.95	29.26
Simplified Strategist vs. Greedy	63%	30.61	12.29	4.43
Simplified Strategist vs. Random	99%	82.1	0.38	20.17

As can be seen, the simplified strategy-using algorithm far outperformed the other 2. Controller performed poorly both versus the greedy and random AI – even though it won all its games against the random AI, it took too long to do so, because it didn't learn to move properly and random AI rarely moved into its units' attacking range on its own. Both strategy using algorithms also took longer to beat the random AI than greedy AI, but this is understandable, as they were mostly trained to fight against greedy AI which came into their range themselves and they didn't have an incentive to seek out random AI's units as quickly as possible.

The simplified strategy-using agent can be considered a success as it managed to achieve a higher than 50% consistent win rate against the benchmark algorithm.

Conclusion and recommendations

While most of the resulting ML agents could not outperform the standard greedy algorithm of CubeWars and the successful agent could still only win X% of the time, the results of the various experiments provide interesting insight into using machine learning for a game of this type. The experiments with the agent that was given full control over the game's mechanics (ability to move into any coordinate, ability to use any action) showed that the environment may have been too complex for the agent to learn right away and a more correct approach would be to first teach it individual aspects of the game, like moving on the field.

The agent that could use higher level movement strategies and all available combat actions performed better, but in the end still could not outperform the greedy algorithm. However, the agent that was allowed to use only abstract higher level actions could achieve good performance. Preparing such agents can take a lot of time, because the different high-level strategies they can utilize have to be programmed to work. Moreover they will not be able to play the game optimally as they do not have perfect control over all the game's aspects; however this does significantly help in making a complicated environment more easily learnable by the agents. This approach could be useful when the game becomes more complex, as a limited number of valid strategies take a reasonable time to program, but creating conditions for all the situations where they should and should not be used would be too much work which reinforcement learning helps lessen.

It should be noted, that it is highly likely that the underperforming agents were simply not trained enough due to time and budget constraints. Reinforcement learning is known for being very slow and not sample efficient. For example, OpenAI's project to create a reinforcement learning agent for DotA2, a highly complex real-time strategy game took 10 months to actually train the agent, and training was done on 1536 powerful graphical processing units and even that agent could not utilize the full extents of the game (there were multiple limitations set on the variety of environments the agent could experience). [5]

Another important factor that increases learning difficulty is the randomness of the environment – in CubeWars actions do not always result in the same outcomes, as different percentage chances come into play. Reinforcement learning is also very unpredictable and the smallest change in either the environment or the hyper parameters can yield completely different results, so a lot of trial and error is needed to create a working agent.

However, despite the somewhat lackluster resulting agents, these experiments show that reinforcement learning can work for a game like this even as a small-scale project. Both strategy-using agents showed slow improvements even towards the end of the training periods and given more training time, it's very likely that they could have achieved better performance. The approach to abstract specific actions in order to decrease the action space could also work in combination with other AI algorithms that are not ML-based, which is something that can be investigated further.

Limitations and future work

The resulting system is very flawed due the difficulty of implementing machine learning in a complicated environment and time constraints – two out of the three trained agents could not outperform the static greedy algorithm that took a fraction of the time to develop. However the ML agents were able to play the game and learn and with more training time and more finely tuned parameters performance could be improved.

Because the author plans to continue working on this project in the future, the AI system for it will have to be continually improved. As this work was focused mostly on achieving that with machine learning, other methodologies, such as the Minimax algorithm, that were not attempted due to time constraints, will be looked into in the future. All the capabilities and configurations of machine learning have not also been exhausted and could be attempted in future research, including:

- Using self-play, where two learning agents play against each other.
- Training agents for longer periods of time.
- Using different models for different units.
- Using a more advanced curriculum and training the agent to first learn to perform individual actions in a game, before letting it play.
- Using a more complicated neural network.
- Using other algorithms, like SAC.
- Experimenting with hyper-parameter configurations.
- Using more different configurations of observation spaces, actions spaces and reward functions.
- Using the action abstraction approach in combination with other AI methodologies.

Furthermore, as complexity of CubeWars increases with the addition of new gameplay elements and mechanics, it will need to be reflected in the AI. Machine learning agents will need to be re-trained in the new environments where these new gameplay elements could translate into increased observation and action spaces. If other, non-ML, AI creation methodologies are used, they will also have to be updated.

References

[1] A. Abdellah and A. Koucheryavy, A. Survey on Artificial Intelligence Techniques in 5G Networks. 2020. http://www.sut.ru/doci/nauka/1AEA/ITT/2020_1/1-10.pdf

[2] Christopher Amato & Guy Shani. High-level Reinforcement Learning in Strategy Games. 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS). 2010. http://www.ccis.northeastern.edu/home/camato/publications/LearningInCiv-final.pdf

[3] James Anthony. 60 Notable Machine Learning Statistics: 2022 Market Share & Data Analysis. 2022. https://financesonline.com/machine-learning-statistics/

[4] Kainat Asif. Supervised vs. unsupervised vs. reinforcement learning. 2022. https://www.educative.io/answers/supervised-vs-unsupervised-vs-reinforcement-learning

[5] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław “Psyho" Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, Susan Zhang. Dota 2 with Large Scale Deep Reinforcement Learning. 2019. https://cdn.openai.com/dota-2.pdf

[7] Philip Boucher. Artificial intelligence: How does it work, why does it matter, and what can we do about it? 2020. https://www.europarl.europa.eu/RegData/etudes/STUD/2020/641547/EPRS_STU(2020)641547_EN.pdf

[8] Steven L. Brunton, Bernd R. Noack, Petros Koumoutsakos. Machine Learning for Fluid Mechanics. 2020. https://doi.org/10.1146/annurev-fluid-010719-060214

[9] Xiao Cui and Hao Shi. A*-based Pathfinding in Modern Computer Games. 2010. https://www.researchgate.net/publication/267809499_A-based_Pathfinding_in_Modern_Computer_Games

[10] Devopedia. Supervised vs. Unsupervised Learning. 2022. https://devopedia.org/supervised-vs-unsupervised-learning

[11] Alexander Dockhorn, Jorge Hurtado-Grueso, Dominik Jeurissen, Diego Perez-Liebana. STRATEGA - A General Strategy Games Framework. 2020. https://gaigresearch.github.io/afm/papers/StrategaAIIDE2020.pdf

[12] Salim Dridi. Supervised Learning - A Systematic Literature Review. 2021. http://dx.doi.org/10.13140/RG.2.2.34445.67049

[13] Salim Dridi. Unsupervised Learning - A Systematic Literature Review. 2021. http://dx.doi.org/10.13140/RG.2.2.16963.12323

[14] Erik Eklov. Game Complexity I: State-Space & Game-Tree Complexities. 2019. https://www.pipmodern.com/post/complexity-state-space-game-tree

[15] Rentao Gu, Zeyuan Yang, Yuefeng Ji. Machine learning for intelligent optical networks: A comprehensive survey. Journal of Network and Computer Applications, Volume 157. 2020. https://doi.org/10.1016/j.jnca.2020.102576.

[16] M.J.H. Heule and L.J.M. Rothkrantz. Solving games - dependence of applicable solving procedures. 2007. https://www.cs.utexas.edu/~marijn/publications/solving_games.pdf

[17] Jarno Hyrkäs. Reinforcement learning in a turn-based strategy game. 2015. https://www.theseus.fi/bitstream/handle/10024/101921/Hyrkas_Jarno.pdf

[18] Sergey S. Ivanov and Elena V. Ponomareva. AI Development in Video Games. 2022. https://elar.urfu.ru/bitstream/10995/113861/1/978-5-91256-550-2_2022_007.pdf

[19] Tomihiro Kimura and Ikeda Kokolo. High-performance Algorithms using Deep Learning in Turn-based Strategy Games. 12th International Conference on Agents and Artificial Intelligence - Volume 2, pages 555-562. 2020. http://dx.doi.org/10.5220/0008956105550562

[20] Chairi Kiourt, Dimitris Kalles, Panagiotis Kanellopoulos. How game complexity affects the playing behavior of synthetic agents. 15th European Conference on Multi-Agent Systems, Evry, 14-15. December 2017. https://arxiv.org/pdf/1807.02648.pdf

[21] Qiong Liu and Ying Wu. Supervised Learning. Encyclopedia of the Sciences of Learning. Springer, Boston, MA. Pages 3243–3245. 2012. https://doi.org/10.1007/978-1-4419-1428-6_451

[22] McKinsey & Company. The state of AI in 2021. 2021. https://www.mckinsey.com/capabilities/quantumblack/our-insights/global-survey-the-state-of-ai-in-2021

[23] Piotr Migdał, Bartłomiej Olechno, Błażej Podgórski. Level generation and style enhancement — deep learning for game development overview. International Simulation and Gaming Association (ISAGA) Conference. 2021. https://arxiv.org/pdf/2107.07397.pdf

[24] Eduardo F. Morales, Hugo Jair Escalante. Biosignal Processing and Classification Using Computational Learning and Intelligence. Academic Press, Pages 111-129, 2022. https://doi.org/10.1016/B978-0-12-820125-1.00017-8

[25] Rupika Nimbalkar. *Unsupervised Learning for Machine Learning and AI.* 2021.

https://medium.com/appengine-ai/unsupervised-learning-for-machine-learning-and-ai-df8f011d9d11

[26] OpenAI. User Documentation: Introduction to RL. 2018. https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

[27] Pytorch.org. Pytorch Documentation. 2022. https://pytorch.org/docs/stable/index.html

[28] Simon Read. Gaming is booming and is expected to keep growing. This chart tells you all you need to know. 2022. https://www.weforum.org/agenda/2022/07/gaming-pandemic-lockdowns-pwc-growth/

[29] Rodrigo Rill-Garcia. Reinforcement Learning for a Turn-Based Small Scale Attrition Game. N.D. https://ccc.inaoep.mx/~esucar/Clases-mgp/Proyectos/2018/reinforcement-learning-turn%20%281%29.pdf

[30] Ziyad Saleh. Artificial Intelligence Definition, Ethics and Standards. 2019. https://www.researchgate.net/publication/332548325_Artificial_Intelligence_Definition_Ethics_and_Standards

[31] Jaydip Sen, Sidra Mehtab, Rajdeep Sen, Abhishek Dutta, Pooja Kherwa, Saheel Ahmed, Pranay Berry, Sahil Khurana, Sonali Singh, David W. W Cadotte, David W. Anderson, Kalum J. Ost, Racheal S. Akinbo, Oladunni A. Daramola, Bongs Lainjo. Machine Learning: Algorithms, Models and Applications. 2021. Machine Learning - Algorithms, Models and Applications. IntechOpen. https://doi.org/10.5772/intechopen.94615

[32] Alessandro Sestini, Alexander Kuhnle, Andrew D. Bagdanov. DeepCrawl: Deep Reinforcement Learning for Turn-based Strategy Games. AIIDE-19 Workshop on Experimental Artificial Intelligence in Games. 2019. https://arxiv.org/pdf/2012.01914.pdf

[33] Claude E. Shannon. Programming a Computer for Playing Chess. Philosophical Magazine, Ser.7, Vol. 41, No. 314. 1950. https://vision.unipv.it/IA1/ProgrammingaComputerforPlayingChess.pdf

[34] Igor Shnurenko, Tatiana Murovana, Ibrahim Kushchu. Artificial Intelligence, 2020. https://iite.unesco.org/wp-content/uploads/2021/03/AI_MIL_HRs_FoE_2020.pdf

[35] Saman Siadati. What is unsupervised Learning*?* 2018. http://dx.doi.org/10.13140/RG.2.2.33325.10720

[36] Steam. AI Roguelite. 2022. https://store.steampowered.com/app/1889620/AI_Roguelite/

[37] Sulaeman Santoso and Iping Supriana. Minimax guided reinforcement learning for turn-based

strategy games. 2nd International Conference on Information and Communication Technology (ICoICT), Bandung, Indonesia. Pages. 217-220. 2014. http://dx.doi.org/10.1109/ICoICT.2014.6914068

[38] Tensorflow.org. TensorBoard: TensorFlow's visualization toolkit. https://www.tensorflow.org/tensorboard

[39] Ujwal Tewari. Which Reinforcement learning-RL algorithm to use where, when and in what scenario? 2020. https://medium.datadriveninvestor.com/which-reinforcement-learning-rl-algorithm-to-use-where-when-and-in-what-scenario-e3e7617fb0b1

[40] Unity. Unity Engine. https://unity.com

[41] Unity. Unity User Manual. 2022. https://docs.unity3d.com/Manual/

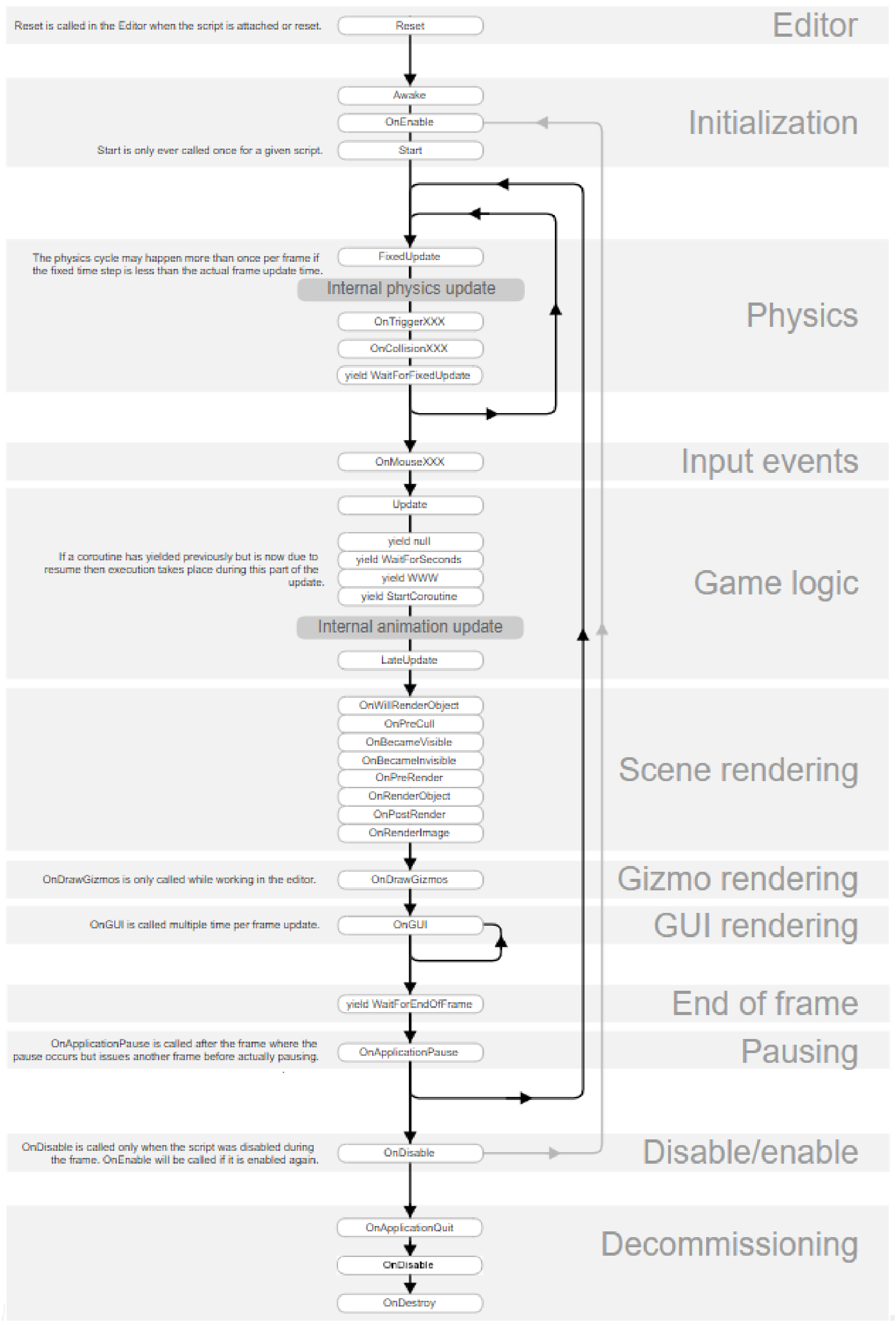
[42] Unity ML-Agents. 2022. https://github.com/Unity-Technologies/ml-agents

[43] Boming Xia, Xiaozhen Ye, Adnan O.M. Abuassba. Recent Research on AI in Games. International Wireless Communications and Mobile Computing (IWCMC), Limassol, Cyprus, pages 505-510. 2020. http://dx.doi.org/10.1109/IWCMC48107.2020.9148327

[44] Like Zhang, Hui Pan, Qi Fan, Changqing Ai, Yanqing Jing. GBDT, LR & Deep Learning for Turn-based Strategy Game AI. 2019. https://ieee-cog.org/2019/papers/paper_7.pdf

Appendices

Appendix A – Unity Engine Workflow [41]



Reset is called in the Editor when the script is attached or reset.
Reset
Editor
Awake
OnEnable
Start is only ever called once for a given script.
Start
Initialization
The physics cycle may happen more than once per frame if the fixed time step is less than the actual frame update time.
FixedUpdate
Internal physics update
OnTriggerXXX
OnCollisionXXX
yield WaitForFixedUpdate
Physics
OnMouseXXX
Input events
Update
If a coroutine has yielded previously but is now due to resume then execution takes place during this part of the update.
yield null
yield WaitForSeconds
yield WWW
yield StartCoroutine
Internal animation update
LateUpdate
Game logic
OnWillRenderObject
OnPreCull
OnBecameVisible
OnBecameInvisible
OnPreRender
OnRenderObject
OnPostRender
OnRenderImage
Scene rendering
OnDrawGizmos is only called while working in the editor.
OnDrawGizmos
Gizmo rendering
OnGUI is called multiple time per frame update.
OnGUI
GUI rendering
yield WaitForEndOfFrame
End of frame
OnApplicationPause is called after the frame where the pause occurs but issues another frame before actually pausing.
OnApplicationPause
Pausing
OnDisable is called only when the script was disabled during the frame. OnEnable will be called if it is enabled again.
OnDisable
Disable/enable
OnApplicationQuit
OnDisable
OnDestroy
Decommissioning