FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Simulating and upgrading PiGaming Mixed Reality setup with Artificial Intelligence

José Nuno Amaro Freixo

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Armando Jorge Miranda de Sousa

July 17, 2020

Simulating and upgrading PiGaming Mixed Reality setup with Artificial Intelligence

José Nuno Amaro Freixo

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Hugo José Sereno Lopes Ferreira

External Examiner: Prof. Eurico Farinha Pedrosa

Supervisor: Prof. Armando Jorge Miranda de Sousa

July 17, 2020

Resumo

Realidade virtual e aumentada têm sido tópicos de elevado interesse nos últimos anos. Realidade mista é outra variação que permite que jogos evuluam usando tecnologias para adicionar objetos virtuais e interagíveis a cenários reais. Isto pode ser feito utilizando uma câmara e um projetor onde objetos reais e projeções podem interagir virtualmente num cenário, sem requerir que utilizadores necessitem de aparelhos visuais.

O PiGaming é um sistema interativo demonstrador de robótica baseado em realidade mista implementado no Robotic Operating System (ROS) e contém vários jogos, onde dois, três ou quatro robôs são reais e qualquer outro elemento de jogo é virtual. O PiGaming contém três jogos: PiTanks, um jogo de tiros do estilo "deathmatch"; Robot factory, um jogo de entregas inspirado em ambientes industriais; e Robot race, um jogo de corridas baseado em número de voltas.

O sistema funciona projetando objetos como paredes ou balas numa tela 2D pousada numa superfície, onde são também colocados os robôs. O cenário de jogo é obtido a partir de uma câmara e as posições dos robôs são dadas por marcadores ArUco posicionados em cima deles. O controlo dos robôs é baseado em input humano e na lógica de jogo.

Na fase inicial desta dissertação, existia um simulador incipiente baseado em Gazebo e uma inteligência artificial naive baseada em máquinas de estado finitas, componentes estas implementadas para o jogo PiTanks.

Durante este trabalho, um simulador eficiente foi desenvolvido. Isto permitiu simulações rápidas, o que por sua vez permitiu o desenvolvimento de técnicas de Inteligência Artificial no tempo restante do trabalho.

O foco principal esteve em melhorar a experiência dos jogos do sistema e facilitar o seu desenvolvimento futuro: com o expandível simulador de elevada performance mencionado anteriormente que funciona para todos os jogos existentes e possíveis jogos futuros. As melhorias à Inteligência Artificial do PiTanks para tornar o jogo mais interessante foram conseguidas através de aprendizagem por reforço. O sistema proposto é baseado em Q-Learning e codifica o estado de jogo discretizado bem como as ações possíveis. A função de recompensa proposta penaliza maus posicionamentos e disparos disalinhados ao oponente, e recompensa disparos alinhados quando o adversário está em linha de visão. A fase de treino ocorreu durante 200 jogos de 2 minutos cada e o jogador automático do distema final conseguiu demonstrar estratégias de jogo comparáveis a um jogador humano iniciante.

Abstract

Augmented and virtual reality have been hot topics as of the past recent years. Mixed reality is another variation that allows gaming to thrive by adding virtual and interactable objects to a real scene. This can be done by use of a camera and a projector where real objects and projections interact virtually on the scene, without needing additional visual apparatus on user's side.

PiGaming is a mixed reality-based interactive robotics demonstrator system implemented in Robotic Operating System (ROS) with several games, where two, three or four robots are real and all other game elements are virtual. PiGaming includes three games: PiTanks, a deathmatch shooter-style game; Robot factory, an industrial-inspired delivery game; and Robot race, a lap-based racing game.

The system works by projecting virtual objects such as walls or bullets onto a 2D mat on a surface, where the real robots stand and move. The game scene is acquired through a camera and the position of the robots is given by ArUco markers on top of them. The control of the robots is based on human inputs and the logic of the games.

At the starting point of this dissertation, there was an incipient simulator based on Gazebo and a naive artificial intelligence system based on finite state machines available for the PiTanks game.

Throughout this work, an efficient simulator was designed. This allowed for fast simulations and this, in turn, enabled the development of Artificial Intelligence techniques in the remainder of this work.

The main focus was on improving the gaming experience of the system and facilitate its future development: with the aforementioned high performance and easily expandable simulator that works for all of the existing games and possible future games. Improving PiTanks' Artificial Intelligence to make it more engaging was achieved by using reinforcement learning. The proposed system is based on Q-Learning that encodes the discretized game state and possible actions. The proposed reward function penalizes bad positionings and shooting a misaligned opponent whilst searching for aligned shots with direct line-of-sight to the opponent. Training took 200 games of 2 minutes each and the final system's automated player was able to perform as well as a human player at a beginner level.

Acknowledgements

Before anything else, I will begin by deeply thanking not only my parents but also my brother, cousins, uncles, and aunts, who have supported me from a very young age, allowed me to have a wonderful childhood, and become who I am today. To not include any of them would not only bring me shame but it would also not feel fair given the role they all played throughout my life.

Next, I want to thank my Supervisor and Professor Armando Jorge Miranda de Sousa not only for allowing me to work on this project while always providing feedback and incentivizing me to bring out the best I could give for its improvement, but also making sure I was keeping up with the project's development and keeping my motivation up.

Lastly, I could never finish this topic without acknowledging the dear friends I have made over these past five years of this Master's degree, who have seen and helped me through some of the darkest points in my life but still pushed through with me by their side. Through desperation, banter, and a couple of beers, without them, I would have never gotten the motivation to see those hardships through. To them, an enormous thank you.

José Freixo

“ I see now that the circumstances of one’s birth are irrelevant.
It is what you do with the gift of life that determines who you are. ”

Mewtwo

Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 Motivation . . . 2
1.3 Objectives . . . 3
1.4 Contributions . . . 3
1.5 Document Structure . . . 4

2 Starting point and state of the art **5**
2.1 System overview . . . 5
2.1.1 PiTanks . . . 5
2.1.2 Robot factory . . . 7
2.1.3 Robot race . . . 8
2.1.4 Gazebo simulator . . . 9
2.1.5 Artificial Intelligence . . . 10
2.2 Simulators in ROS . . . 10
2.2.1 Gazebo . . . 11
2.2.2 CoppeliaSim . . . 11
2.3 Reinforcement Learning . . . 12
2.3.1 In robotics . . . 12
2.3.2 Deep Learning . . . 14
2.3.3 Q-Learning . . . 15

3 Proposed improvements **19**
3.1 Simulation . . . 19
3.2 PiTanks' Artificial Intelligence . . . 20
3.3 System architecture . . . 21

4 High performance simulator **23**
4.1 Developing a simpler simulator . . . 23
4.1.1 Objects . . . 23
4.1.2 Distance unit . . . 26
4.2 Communication . . . 27
4.3 Validation and conclusions . . . 30

5 Artificial intelligence with Q-Learning **31**
5.1 Implementing Q-Learning . . . 31
5.1.1 Communication . . . 33
5.2 State and action spaces . . . 34

5.3 Reward function 35
5.4 Performance and conclusions 37
5.4.1 Evaluation against previous AI 37
5.4.2 Evaluation against human players 38

6 Conclusions and Future work **41**
6.1 Future work 42

References **43**

List of Figures

1.1 PiGaming's physical setup . . . 2

2.1 Original Polulu 3pi (left) and modified Polulu 3pi (right). . . . 6
2.2 PiTanks' game map. . . . 6
2.3 Robot factory's game map. . . . 7
2.4 Robot race's game map. . . . 8
2.5 PiTanks' simulated Gazebo environment. . . . 9
2.6 PiTanks' finite state machine AI . . . 10
2.7 Standard reinforcement learning model. . . . 12
2.8 RoboCup's Standard Platform League game. . . . 13

3.1 Custom modelled robots for the previous simulator in three points of view 20
3.2 PiGaming's initial ROS architecture with the simulation module. . . . 21
3.3 PiGaming's new proposed ROS architecture with the new simulation module. . . 22

4.1 PiTanks in the new simulator. . . . 24
4.2 Robot factory in the new simulator. . . . 26
4.3 Start of game communication. . . . 27
4.4 Game update communication. . . . 27

5.1 System communications regarding machine learning. . . . 34
5.2 Numerical representations of the robot's directions. . . . 35
5.3 First agent (blue): Match where both players kept standing still. . . . 37
5.4 Second agent (blue): Match where the agent bombarded the previous AI early on. 38

List of Tables

5.1 Rewards determined by the first AI's reward function. 36
5.2 Rewards determined by the second AI's reward function. 36
5.3 Agent results with each reward function against previous AI. 37

Abreviaturas e Símbolos

ROS	Robotic Operating System
AI	Artificial intelligence
API	Application programming interface

Chapter 1

Introduction

This chapter shows an overview of the work done. First, a brief contextualization about the system is given. Secondly, the motivations for this dissertation are described. Thirdly, the intended goals to be achieved are detailed as well as their overall results and contributions to the project. Fourthly, what contributions were made to the project. Lastly, the document's structure is summarized.

1.1 Context

The work done in this masters dissertation was made with the goal of furthering the development of PiTanks [5], a project previously developed by several students from both MIEEC and MIEIC.

PiGaming is an interactive robotics demonstrator system implemented on ROS [21] which features several mixed reality games, where two, three or four robots are real and all other game elements are virtual. This demonstrator currently includes three games:

- PiTanks: a timer-based deathmatch shooter-style game;
- Robot factory: an industrial-inspired delivery game;
- Robot race: a lap based racing game.

The system works by projecting virtual objects such as walls or bullets onto a 2D mat on a surface, where the robots stand and move. Therefore, all games have a top-down view and their scenes are always a 2D plane. The game scene is acquired through a camera and the position of the robots is read through the detection of ArUco markers [19][7] placed on top of them. The control of the robots is based on human inputs on joystick controllers and the logic of each of the games.

The game PiTanks specifically has two extra components to allow play without requiring real robots: Simulation and AI [6]; these components, however, are fairly restricted and unsuited for future development. As such, for this dissertation, it was proposed to generalize the simulation aspect to all of the games as well as upgrading the overall gaming experience of the system.

1.2 Motivation

The first driving motive for this dissertation is the improvements to the simulation. Considering the starting point of this dissertation, in order to develop new games or make improvements to both Robot factory or Robot race, testing can only be done with the system's physical setup. This setup can be seen in Figure 1.1.

Given the setup's bulkiness and limited availability, game testing cannot be done every time it is required, which hinders the development process of the games. As such, the leading force behind the generalization of the simulation component of the system is to smoothen the development stage of future projects, better preparing the system for new games, different game modes, or improvements for the existing games.

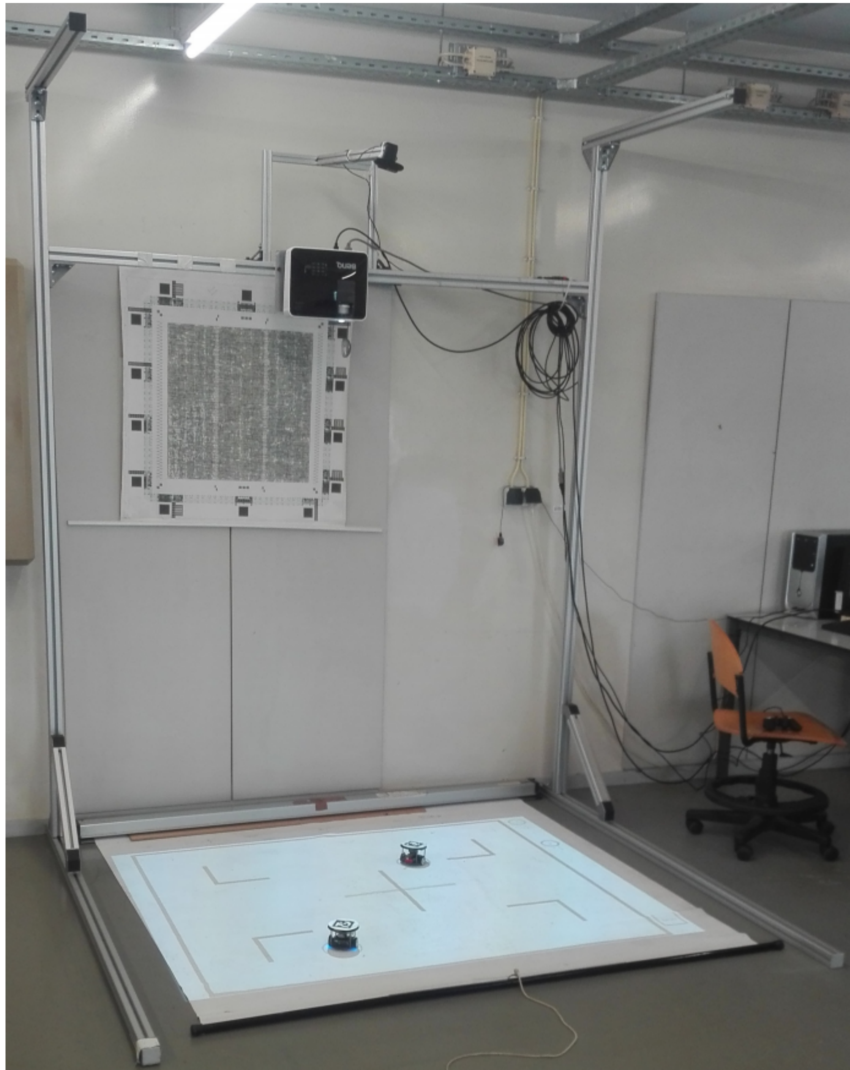

Figure 1.1: PiGaming's physical setup

The second motive is the addition of a more competitive and interactive AI for PiTanks. Despite already having a state machine-based AI, introducing the concept of reinforcement learning with a new self-taught AI could not only expand PiGaming's system's gaming value by bringing a bigger challenge to more experienced players, but also allow it to slowly develop undiscovered strategies that could enhance the game's depth.

1.3 Objectives

Given the aforementioned motivations for this dissertation, there are two main objectives to be achieved:

- Implementing a high performance simulator;
- Developing a self-taught AI for PiTanks.

By providing a high performance simulator, all games, including possible future ones, should be able to smoothly run on the simulated environment, which in turn should allow for reinforcement learning algorithms to be applied more efficiently, as well as improve playability and fidelity with the physical system when testing the games.

The newly developed self-taught AI should both be able to consistently beat its previous state machine based implementation, as well as have decent chances of beating human players.

1.4 Contributions

Regarding the simulation aspect, the simulator was developed with the purpose of having a higher performance while being as lightweight as possible and staying loyal to the physical system.

The fidelity to the physical system was planned to be tested by running the games both with and without the physical setup while performing a given set of actions for both cases, thus gathering data about the position of the robots on each and comparing them, allowing to tweak the simulator accordingly to be loyal to the physical setup. This was however deemed unfeasible due to the physical system's inaccessibility during the development phase of this dissertation as a result of the pandemic outbreak and confinement order.

An additional detail of the new simulator is that it stores information not only about the robots, which was the only required aspect for the games to run without the physical setup, but also the virtual objects. The reason for this inclusion was for the simulator to also serve as the environment that contained all required information for the machine learning algorithms to gather and train the new AI.

Regarding the AI aspect, Q-Learning was used as the first step into self-taught AIs for this project.

At the end of this dissertation, it is possible to play all three games in the simulated environment, as well as PiTanks in a 1 versus 1 scenario against the self-taught AI.

1.5 Document Structure

Besides this brief introduction, this dissertation contains five more chapters.

In chapter 2, PiGaming's state regarding the beginning of this dissertation's development phase is discussed while emphasizing the focused features' shortcomings, as well as the current state of the art, specifically in robotics simulation and usage of reinforcement learning in real-time games.

In chapter 3, possible solutions and improvements for the shortcomings analyzed in the previous chapter are considered and the chosen approaches for the improvements are justified by taking into account the project's state and requirements. Besides this, the initial architecture of the project is detailed along with the necessary changes made.

In chapter 4, the new simulator's development is detailed in addition to the changes made to the core game engine of the project. This includes the simulator's position within the architecture of the whole system. Lastly, the simulator is compared to the previously implemented one, and the results of the implementation of the earlier are presented.

In chapter 5, the developed AI's implementation is described. The interactions between it and the simulator detailed in the prior chapter are detailed. Lastly, the AI's performance against the previously implemented one and a small number of human players is evaluated.

Lastly, in chapter 6, the conclusions for the work done in this dissertation are discussed as well as possible future additions and improvements to the PiGaming system as a whole.

Chapter 2

Starting point and state of the art

This chapter describes the project's state at the start of this dissertation as well as the focused shortcomings that justify the work done in this dissertation. The state of the art related to simulation in robotics and the usage of reinforcement learning in real-time games is also discussed.

2.1 System overview

PiGaming [5] is an interactive robotics demonstrator system implemented in ROS which features several mixed reality games, where two, three, or four robots are real and all other game elements are virtual. PiGaming currently includes three games: PiTanks, a timed deathmatch shooter-style game; Robot factory, an industrial-inspired delivery game; and Robot race, a lap-based racing game.

The games are categorized as mixed reality games due to the interactions between real robots and virtual objects. As such, actions performed by the robots are able to cause changes in the virtual objects and vice-versa. With this, all game rules and game modes present in the system are based on the interactions between both the real and the virtual domains.

The system works by projecting virtual objects such as walls or bullets onto a 2D mat on a surface, where the robots stand and move. The game scene is acquired through a camera and the position of the robots is given by ArUco markers on top of them. To control the robots, up to four joystick controllers are used depending on the number of players and the chosen game. To send the controllers' input to the robots, an additional wireless XBee module is connected to the computer running the system. The robots used are Polulu 3pi [1] modified with a small cover which holds the ArUco markers. Both the regular and modified robots can be seen in Figure 2.1.

2.1.1 PiTanks

PiTanks is a timed deathmatch shooter style game and has two play modes: free-for-all and teams, requiring a minimum of two and a maximum of four players to be played. The game's map can be seen in Figure 2.2.

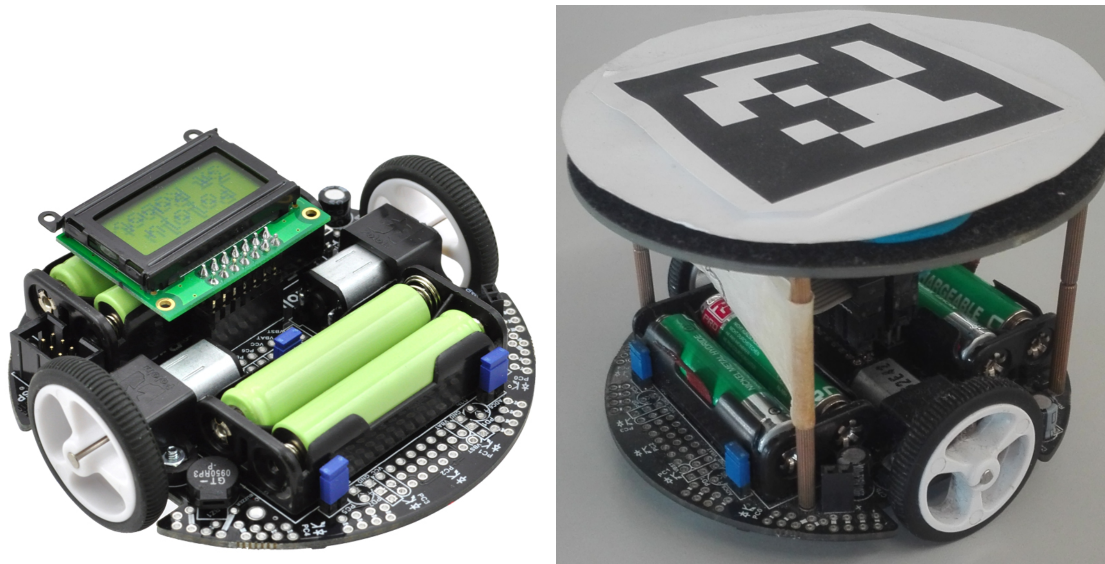

Figure 2.1: Original Polulu 3pi (left) and modified Polulu 3pi (right).

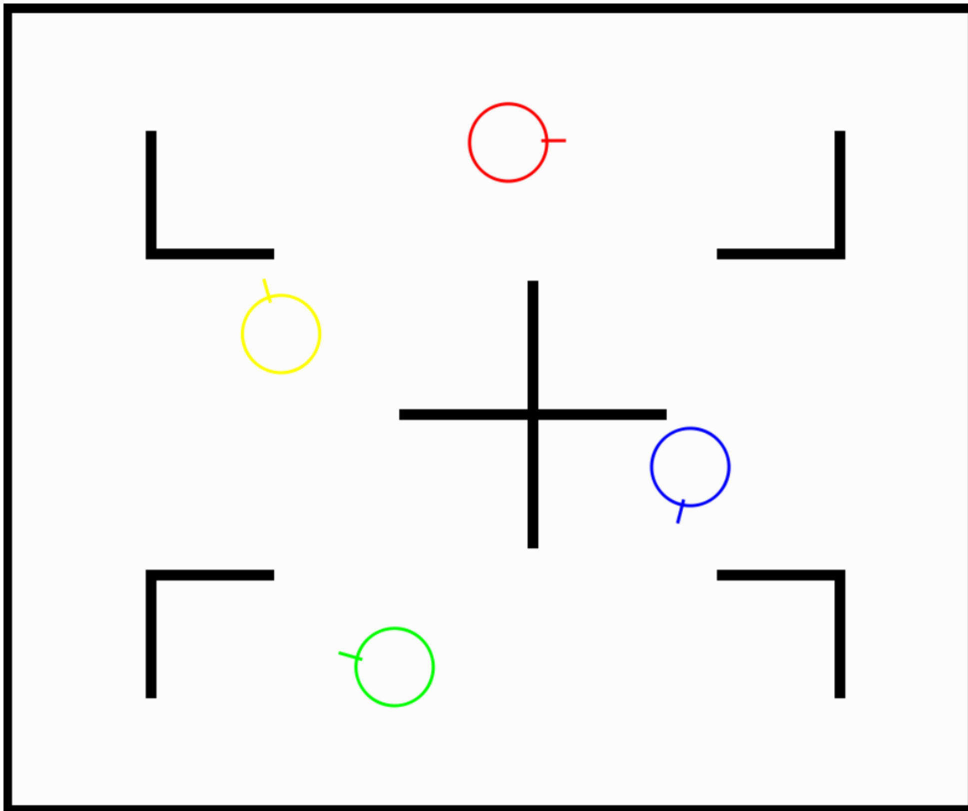

Figure 2.2: PiTanks' game map. [6]

Each robot represents a tank, which is able to move and shoot virtual bullets. These robots navigate through a virtual map with indestructible outer walls that keep them inside and destructible inner walls that can be used as cover. The biggest difference from most shooter games is that there is no concept of health or hit points, meaning that the scores are calculated by how many shots a player or team have hit opponents with and how many shots they have taken from opponents, this game mechanic means that all players keep playing until the game time is over, instead of instantly losing once their hit points reach zero and having to wait for the other players to finish the game. There is no standard game time, it is always agreed upon and selected before the start of a game.

2.1.2 Robot factory

Robot factory is a competitive game which simulates an industrial environment, the game's map can be seen in Figure 2.3. Two teams of two robots each compete and try to move pallets from their entry warehouse to the appropriate industrial machines and then to their exit warehouse. Different colours are used to distinguish which warehouses belong to which team.

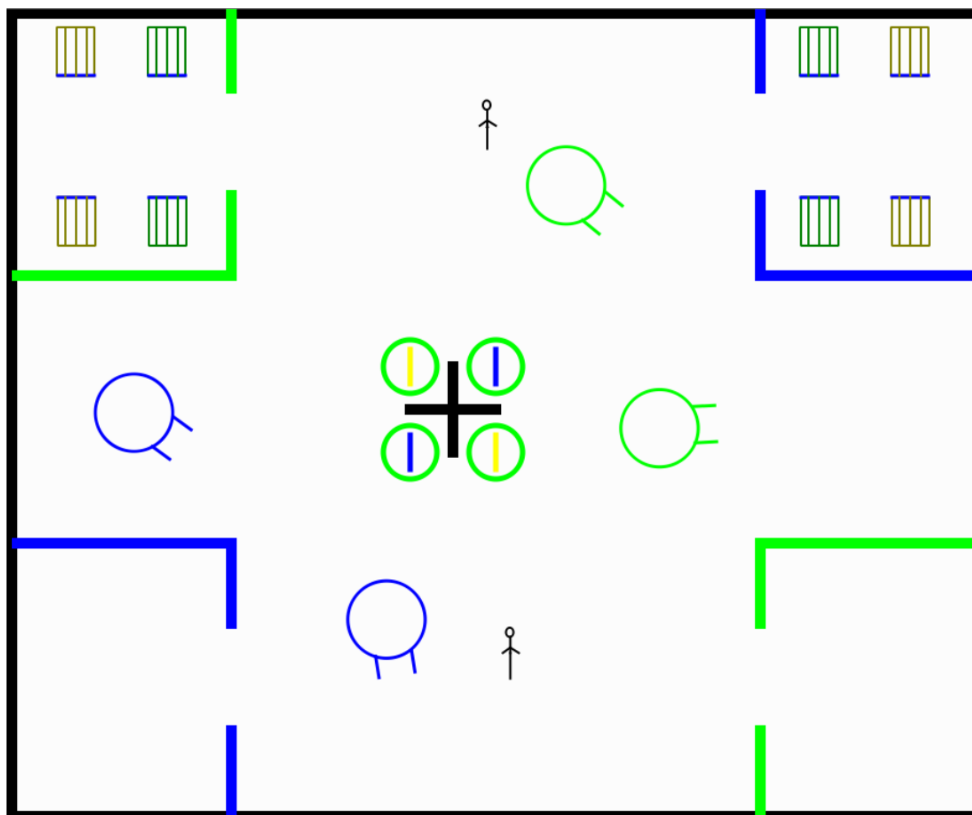

Figure 2.3: Robot factory's game map.

The first team to move all of their processed pallets to their exit warehouse is declared victorious. Both teams share the industrial machines, meaning that proper management of their use and teamwork between robots of the same team are fundamental to being more efficient than the

opposing team. If a pallet is placed in the wrong machine, it will get damaged and become unable to be processed. Besides the robots, their warehouses, the machines, and the pallets, there are also NPCs that represent human factory workers, these NPCs move to machines that have appropriate pallets in them in order to process them. If a robot moves over an NPC, the robot gets immobilized for three seconds as a penalty. Every pallet needs to be processed twice in order to be considered complete, only then it is accounted for when it is dropped in the exit warehouse.

2.1.3 Robot race

Robot race is a racing game where robots compete to see which can complete three laps the fastest, the game's map can be seen in Figure 2.4. A particular feature that distinguishes this game from most racing games is the fact that each robot has its own start and finish line.

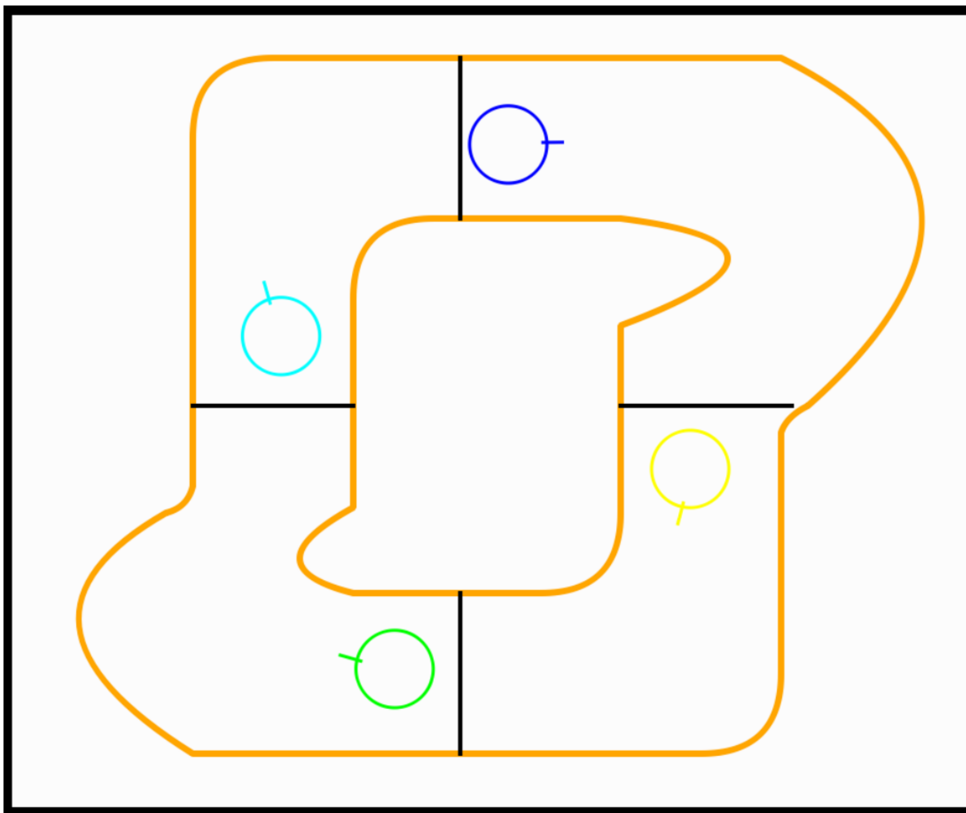

Figure 2.4: Robot race's game map.

Compared to the other two previous games, the robots' velocity is slightly increased in this game to make the races more interesting and fast-paced. To prevent players from going out of the race track's bounds, the robots' speed is reduced by a ratio of $\frac{2}{3}$, significantly slowing them down. The game is also prepared to not count laps that are made through the middle of the track, by making use of the other robots' start and finish lines as checkpoints which must be cleared for the lap to be counted.

2.1.4 Gazebo simulator

PiTanks also features a custom simulator and an artificial intelligence module [6], which makes it possible to play with simulated robots as one would with real robots, as well as play against an AI in a one versus one scenario.

This simulator for PiTanks is based on Gazebo [13] and despite its usefulness of allowing the use of custom modelled robots that are a decently accurate simulated representation of the robots, which allow the game to be played without the real robots, it has a few shortcomings.

As can be seen in Figure 2.5, the real-time factor of this simulator fluctuates around 0.5 in a 64-bit Windows 10 machine, this means that the game runs at roughly half the speed it otherwise would if it was played with real robots instead of the simulated ones. This issue is mostly due to the usage of Gazebo, which could be considered as too complex for what is needed to be simulated for the PiGaming system specifically, therefore adding unnecessary complexity given that the in-game representation of the robots are simple circles.

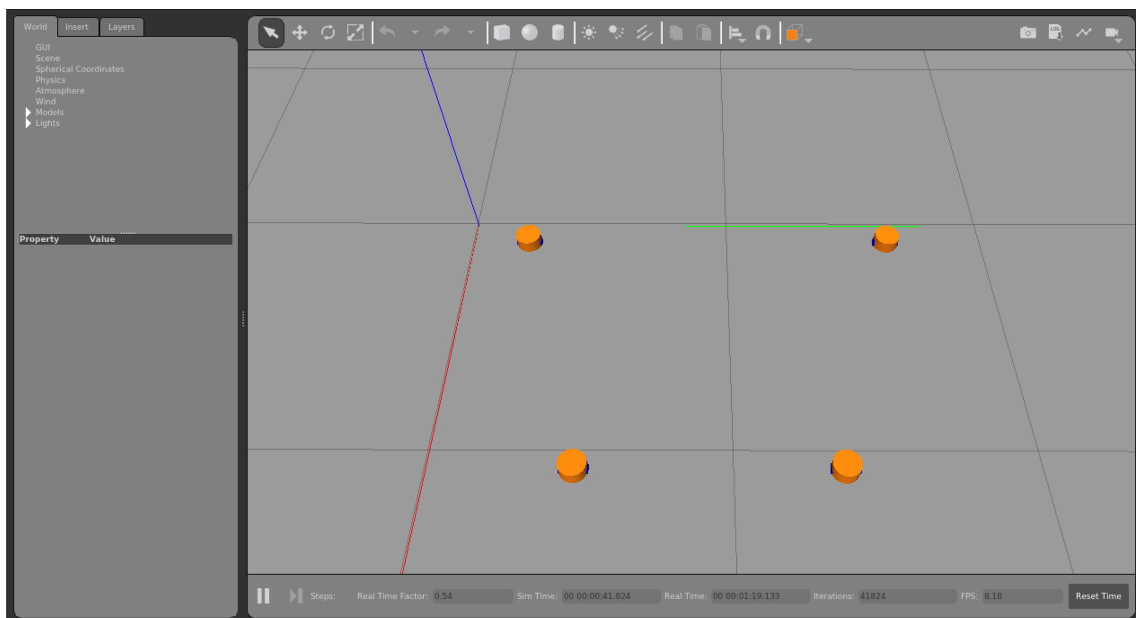

Figure 2.5: PiTanks' simulated Gazebo environment.

Additionally, the dimensions of the game objects in the game engine and in the simulator are completely different, this means that whenever the system needs to communicate with the simulator or vice-versa, the units always need to be converted. This detail makes it not only difficult to understand why these calculations are needed for future developers, but also take unnecessary processing time since this communication happens twice, converting to the simulator and deconverting to the game engine, for every frame that the game runs.

2.1.5 Artificial Intelligence

The AI is based on finite state machines and features both offensive and defensive behaviours depending on if the AI is winning or losing. These behaviours are displayed in Figure 2.6.

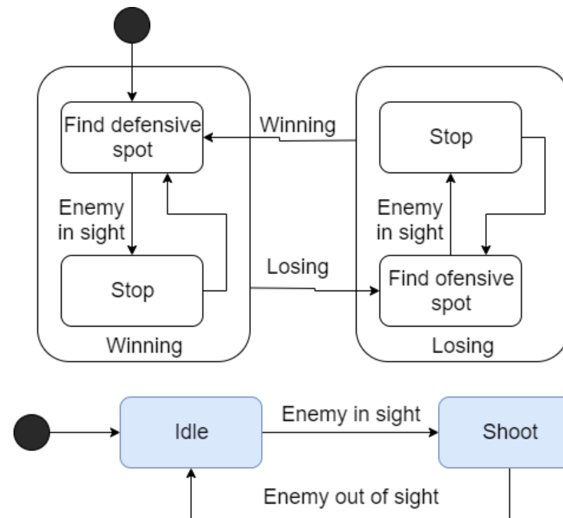

Figure 2.6: Simplified view of PiTanks' finite state machine AI.

However, this approach can be both easily exploitable once known how the AI behaves, and malfunction at critical times, resulting in strange situations where it seems to be stuck in a specific behaviour and allowing other players to easily win against it by staying in the same spot while continuously shooting.

2.2 Simulators in ROS

Since the first focus of the work in this dissertation is employing a more lightweight and generic simulator in the system, it is of importance to analyze several relevant simulators that fit this criteria.

When it comes to choosing which simulator is best suited for the system, there are several important factors to take into account:

- Integration and compatibility with ROS [21], a flexible framework with several useful libraries and tools for writing robot software: since ROS is the base of the PiGaming system;
- Performance under stress conditions: this is due to PiGaming sometimes requiring the use of four different robots at the same time which when simulated may cause the system to lag and thus slow down the gameplay;
- Future usage: PiGaming has been in development for several years now and it is intended to keep being upgraded in the future, thus requiring a simulator that stays relevant.

Considering the aforementioned factors, the two most commonly used simulators in the robotics industry were analyzed: Gazebo and CoppeliaSim.

2.2.1 Gazebo

Despite being the implementation base for the previously developed simulator for PiTanks and one of the reasons as to why this dissertation exists, it is still worth going into detail about its capabilities for possible performance-enhancing options.

Gazebo [13] is an open-source 3D dynamic simulator and is the default used simulator for ROS projects. Despite being different projects, Gazebo is developed with the philosophy of being specifically used with ROS, which makes its integration and compatibility some of its biggest strengths. Since it is the most commonly used simulator in the robotics community, it possesses an extensive amount of documentation as well as open discussions and forums, which in turn allow its workflow to be fairly straightforward.

When it comes to robot modelling, it is not only possible to use several, although not many, predefined ones, but also import custom-made designs created outside of Gazebo. This feature allows users to define their own robots in a way that most closely fits the features of the real robots that are intended to be simulated.

Regarding physics, ODE is Gazebo's default engine. However, it is possible to integrate other physics engines manually if deemed necessary. The issue with these engines when it comes to usage in the PiGaming system is that since they handle 3D environments, their level of complexity is fairly greater than what is required within PiGaming which only features two dimensions.

Lastly, in terms of performance in multi-robot simulation, Gazebo shows fairly decent results [17] in regards to CPU usage as well as keeping up the real-time factor.

2.2.2 CoppeliaSim

CoppeliaSim [18], formerly known as V-REP, is a 3D simulator and is one of the simulators in robotics projects with the largest number of functionalities and tools. It is most distinguishable from other simulators because it allows for high fidelity when it comes to physics, by featuring several physics engines such as Bullet, ODE, and Vortex.

Unlike Gazebo, it is not developed with the intention of being specifically used alongside ROS. However, there are several different interfaces and plugins which allow the development of ROS projects without complications.

It possesses a large number of native robot models while also allowing users to import and edit their own models inside the simulator itself. Alongside a being well documented, this is one of the features that make CoppeliaSim's UI to be considered one of the most intuitive and accessible.

When it comes to performance, due to its high focus on realism, it falls behind significantly in multi-robot simulations [17]. A small scene with 5 robots holds a real-time factor of 0.38 and a 200 p.p. increase in CPU usage.

2.3 Reinforcement Learning

Since the current state machine-based AI for PiTanks is at times faulty and naive, reinforcement learning could prove to be an important step into developing a more challenging AI that appeals to more experienced players.

Reinforcement learning [9] is an area of machine learning which focuses on the process of making an agent learn a certain behaviour through trial-and-error interactions with an environment that possesses a finite amount of possible states, and a finite set of actions per state.

There are two primary approaches when it comes to reinforcement-learning problems. The first is to explore the state space of the environment with the goal of finding a behaviour with a good performance. The second is using statistics and dynamic programming methods to estimate the value of taking actions in states of the environment.

In a standard reinforcement learning model, an agent interacts with its environment by performing actions, making it transition from one state to the next. When an action is executed, an immediate reward is provided to inform the agent of how beneficial the action taken for that given state was. The long-term value for the action is the sum of its immediate reward and the achievable value from the new state. This approach results in influencing the selection of an action by the potential reward of future actions in future states, thus leading to more promising decisions the more agent trains. Ultimately, the objective of a reinforcement learning agent is to maximize its total obtained reward.

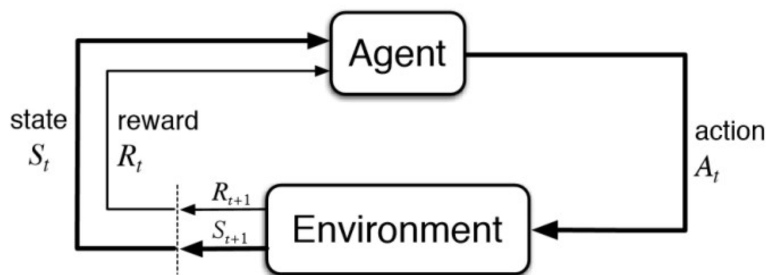

Figure 2.7: Standard reinforcement learning model.

There are several approaches and algorithms when it comes to reinforcement learning. As such, in this subsection, the usage of reinforcement learning in robotics is explored and several state of the art reinforcement learning algorithms and developed AIs are discussed.

2.3.1 In robotics

Reinforcement learning in robotics [12] varies significantly from other reinforcement learning problems, mostly due to most problems in robotics being represented by continuous and high-dimensional states and actions. In addition, it is frequently unreasonable to assume that the acquired state is complete and noise-free, which leads to the agent responsible for the robot's actions

oftentimes not being able to specifically determine in which state it is or misjudge one state for another.

Besides this inaccuracy, issues with training when it comes to real physical systems are reproduction, cost and time: since replicating initial state conditions is almost close to impossible; every trial setup is costly, can lead to malfunctions and result in damaged hardware; and given the real-time limitation, the time frame for a robot to learn the behaviour it is expected to can at times be unreasonable.

Despite this, advances in simulation have bypassed these physical issues and lead to a much more efficient application of reinforcement learning in the robotics domain. More specifically, RoboCup [11] is a worldwide robotics initiative that has withstood these issues and made great advancements with reinforcement learning in robotics.

RoboCup

RoboCup [11] is an initiative that attempts to foster AI and intelligent robotics research by offering a standard problem. This problem is using a soccer game as a platform for a wide range of AI and robotics research, where several different kinds of robot teams face each other. A game of RoboCup's Standard Platform League can be seen in Figure 2.8.

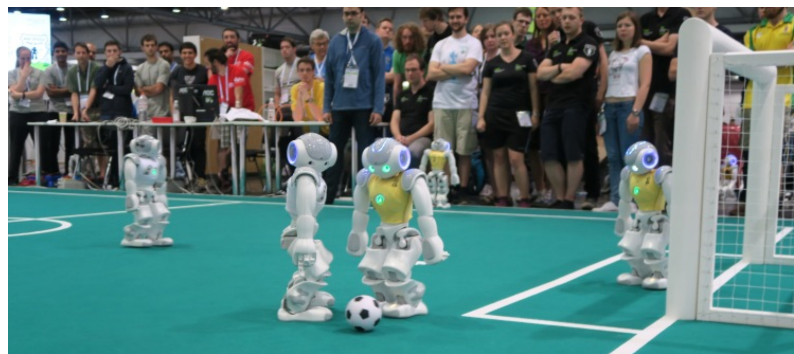

Figure 2.8: RoboCup's Standard Platform League game.

This competition has promoted the development of many kinds of intelligent behaviours through reinforcement learning over the years.

Keepaway Soccer [22] is a subtask of regular RoboCup soccer developed through reinforcement learning and its goal was to develop a maintain ball possession strategy for the robots to use in RoboCup. While there were already hand-coded strategies which made use of several macro-actions, such as holding the ball, passing the ball and moving to an open space to receive a pass, their effectiveness was not close to optimal and it seemed possible for the robots to maintain possession of the ball for longer periods of time, thus leading to the development of this behaviour with reinforcement learning.

The results of the new keepaway behaviour with reinforcement learning showed better episode durations, meaning more time with ball possession, after very few hours of training, and after

several more hours, the robots were able to maintain possession of the ball for double the amount of time they could with hand-coded strategies, thus proving that an approach through reinforcement learning for the development of AI strategies could provide significant improvements.

Half Field Offense [10] is an expansion of the aforementioned keepaway behaviour, and its task was to develop a strategy that would allow offensive robots from one team to outsmart the defensive team's robots, including its *goalkeeper*, to score a goal, while mostly assuming that the defending team would have more players. Just like its predecessor this task was episodic, unlike it however was that it was not relevant how long the offensive team could keep possession, instead, what was important was how consistently it was able to score goals without losing possession of the ball or letting it get out of bounds.

The resulting behaviour was able to achieve a success rate of 32% with inter-agent communication and 23% when each agent learned independently. When compared to hand-coded strategies for the same behaviour, it was able to surpass their average success rate of 12.5% within less than 5000 episodes of training.

2.3.2 Deep Learning

Deep Learning [14] is an area of machine learning which uses multiple processing layers to progressively extract higher-level features from raw input data. Over the years, deep learning methods have significantly improved the state of the art not only in game AIs but also speech recognition, object recognition and detection, and other domains such as drug discovery and genomics. Through backpropagation, deep learning determines how much a machine should change its internal parameters which are used to compute the representation in each layer from the representation in the previous layer.

Although most frameworks for deep learning have their core libraries implemented in C++, such as Tensorflow [3] or PyTorch [16], to have increased efficiency, their API's have better support and documentation for the Python language, which besides not being the most efficient, can at times not be suited for specific projects, which makes the usage of these frameworks intricate.

As breakthrough as it may be, deep learning tends to take a much larger degree of time to train when compared to other training methods, despite having demonstrated its capacity to produce highly effective results.

AlphaZero

AlphaZero [20] is a deep reinforcement learning program that was able to master several board games such as chess, shogi, and Go by only taking in as input the current state of the board and outputting a vector of move probabilities and their scalar move value. These probabilities and values were learned exclusively through self-play and without any domain knowledge except for each game's rules.

This program demonstrated superhuman performances and was able to defeat Stockfish, the world champion chess program, winning 155 times and losing 6 out of 1000 matches in chess; in shogi, AlphaZero defeated Elmo 98.2% of games when playing black and 91.2% overall.

AlphaStar

AlphaStar [4] was the first deep learning AI system to beat a professional player at the game of Starcraft 2, a real-time strategy game, in january 2019, which represented a milestone in the progress of AI.

Initially, AlphaStar's agents were trained by supervised learning from anonymized human games released by Blizzard, Starcraft's developer, which allowed it to learn by imitation the basic micro and macro strategies commonly used by players; this lead these initial agents to defeat the built-in Elite level AI in 95% of games. These agents were then used to seed a multi-agent reinforcement learning process, simulating a continuous league where agents played against each other; new competitors were dynamically added to this league by branching from existing competitors. With the progression of this league and the creation of new competitors, new strategies emerged and others were refined.

OpenAI Five

OpenAI Five [15] is a Deep Reinforcement Learning agent that was able to learn the game Dota 2, a multiplayer online battle arena, through self-play and was able to defeat the reigning Dota 2 world champions in april 2019, this was the first time ever an AI was able to defeat an esports world champion team.

Besides the world champions, OpenAI Five became available for the Dota 2 community to freely play against, in these community matches OpenAI Five won 99.4% of over 7000 games. This proved the viability, if not superiority, of reinforcement learning in complex environments such as Dota 2.

2.3.3 Q-Learning

Q-Learning [23] is an off-policy reinforcement learning algorithm that attempts to find the best possible action to take given the current state in order to advance to a more promising state. It is considered off-policy since it learns through exploration, meaning that it seeks to learn a policy that maximizes the total reward through the exploration of random actions.

The Q stands for quality, which is represented by how advantageous it is to take a certain action in a given state.

In order to represent these values in every possible action in every possible state, a Q-Table is used. A Q-Table is a matrix with the shape *[S,A]*, where S is the number of possible states and A is the number of possible actions an agent can take at any given moment, and it functions as a reference table for the agent to choose the best possible action based on their Q-value. Every Q-value starts at 0 when training begins, meaning that at the beginning of training, the agent does not

know which action leads to a more promising future state, however, through exploration, the agent updates the Q-values of the chosen actions through the delayed reward it receives. The longer the agent trains, the more these Q-values converge, making the agent more confident in the actions they choose in future episodes.

To update a Q-value, Q-Learning follows a Bellman equation as a simple value iteration update, by using the weighted average of the old value and the new information:

$$Q^{\text{new}}(s_{\text{t}}, a_{\text{t}}) \leftarrow Q(s_{\text{t}}, a_{\text{t}}) + \alpha * (r_{\text{t}} + \gamma * maxQ(s_{\text{t+1}}, a) - Q(s_{\text{t}}, a_{\text{t}}))$$

Where:

- t: is the current time step;
- s_{t}: is the current state;
- a_{t}: is the current action taken;
- $Q^{\text{new}}(s_{\text{t}},a_{\text{t}})$: is the new Q-value for performing action a_{t} in state s_{t};
- $Q(s_{\text{t}},a_{\text{t}})$: is the old Q-value for performing action a_{t} in state s_{t};
- $maxQ(s_{\text{t+1}},a)$: is the maximum Q-value of all of the actions for state $s_{\text{t+1}}$;
- α: is the learning rate, which determines the extent of how much the new information overrides the old value;
- r_t: is the reward received for transitioning from state s_{t} to $s_{\text{t+1}}$;
- γ: is the discount factor and has the effect of valuing earlier rewards received higher than those received later thus leading to the convergence of the Q-values obtained throughout the agent's training;

CLASS$_{\text{Q-L}}$

CLASS$_{\text{Q-L}}$ [8] is a Q-Learning based algorithm that was developed to complete Wargus games, a real-time strategy game similar to Warcraft 2 where players control armies of units from several classes.

The strategy used to implement Q-Learning in this environment was to have multiple Q-Tables each representing one of the classes of units, this was due to different classes having different sets of actions. This way, by having a single table for all units of the same class, the learning process was significantly sped up compared to updating Q-values for every single unit's action.

To reduce the number of possible states, several state features that had too many possible values were generalized. These included features such as gold, resources like wood and food, and number of units; for instance, gold was discretized to have 18 possible representations, from 1 meaning 0 gold and 18 meaning more than 4000 gold.

Wargus had a built-in scoring system to determine which player was the winner at the end of the game. However, this value was not very indicative of determining by how much one team won since at times quick victories would have a smaller score differential than much longer games. Instead, each match was repeated ten times to obtain a more statistical value of which player performed better.

This algorithm was tested in a small map of Wargus, in which it was able to consistently beat all of its opponents after very few iterations of training, proving to be a quick learner. The opponents used to train and evaluate $\text{CLASS}_{\text{Q-L}}$ were AIs that come with the Warcraft distribution.

Chapter 3

Proposed improvements

This chapter explains the approaches that were chosen from what was analyzed in the previous chapter, as well as explain why the other ones were discarded. An overview of the changes in the system's architecture is also provided to highlight what was changed during the development of this project.

3.1 Simulation

Given the importance of requiring an appropriate simulation module, not only for the second part of this dissertation, the self-taught AI, but also for future improvements of the PiGaming system as a whole, there are three main factors that must be guaranteed:

- A real-time factor of at least 1 when simulating up to four robots;
- Straightforward integration in ROS;
- Expandability for future development;

The real-time factor is the quotient between the elapsed simulation time and real-time, if this value can be assured to stay at values of 1 or greater, the game will never run slower than it is intended when simulated. Since all games in PiGaming are real-time games, guaranteeing this factor ensures that the simulated games will stay faithful to their physical counterparts.

Considering the fact that PiGaming has been developed in ROS from its very beginning and is currently not planned to change, having a simulation module that fits the ROS framework is a necessity especially when dealing with communication with other modules of the PiGaming system.

Since it is expected that PiGaming will continue to receive improvements over the years to come, making certain that its simulation module is easily expandable and generic is of great importance to allow future games and upgrades to be easily incorporated.

Taking into account [17], Gazebo seems to show the best values in regards to the real-time factor when used to simulate small scenes, such as the ones in PiGaming, with up to five robots.

However, as shown in the previous chapter, when used in PiTanks, Gazebo struggles to keep up a real-time factor value greater than 0.55 which is much less than the requirement of 1. And since the custom modelled robots [6] that can be seen in Figure 3.1 were already purposefully modelled to be simple thus avoiding complicated physics calculations, improving the real-time factor was deemed unfeasible.

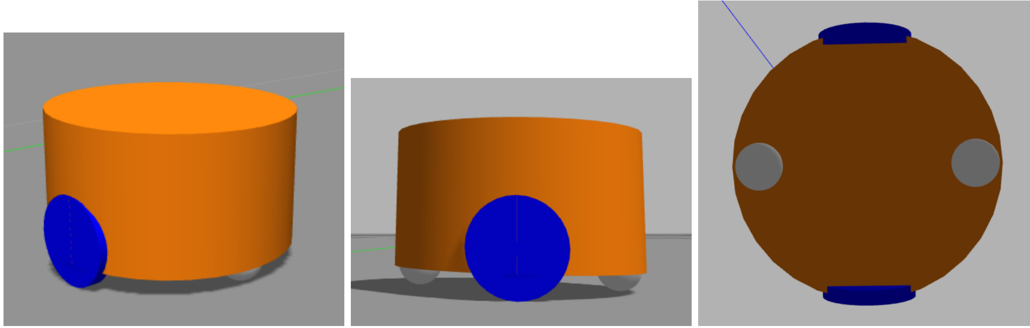

Figure 3.1: Custom modelled robots for the previous simulator in three points of view [6].

CoppeliaSim, although having a higher level of realism, lacks even more than Gazebo when it comes to keeping up high values of the real-time factor in small scenes with five robots.

With both most common simulators in the robotics industry not being suited for PiGaming, both due to their high levels of complexity, and PiGaming's games being simple 2D minigames with basic game rules, it was decided to implement a completely new 2D simulator from scratch as a new ROS package module for PiGaming. This new simulator would feature simple objects with basic information, such as position and orientation, since the whole purpose of implementing a simulation module in the system was to replace the camera component of the physical setup by informing the game engine where each robot was located. This approach would allow: minimal resource usage since only object positions and geometrical collisions would be calculated; one-to-one object dimensions by focusing on pixels as the main unit; all game objects' information to be stored in one centralized module for the new AI component to access.

3.2 PiTanks' Artificial Intelligence

With the goal of exploring reinforcement learning to develop a more challenging AI for PiTanks, there were two major concerns when it came to choosing which approach was more effective and efficient:

- Simple integration within the ROS C++ framework;
- The amount of time it would take to train the AI;

Since PiGaming was developed in C++ and within the ROS framework, several setbacks were present at the very beginning, due to most reinforcement learning frameworks only having proper

implementations and documentation in the Python language. Although, even with little to no documentation, there were some of these frameworks which also included C++ libraries of their implementation, such as PyTorch's LibTorch [16], including these libraries within ROS proved to be a difficult task and had very little online support. This does not mean that usage of this library in the future is inconceivable, and if an approach is found that allows its usage, it should certainly prove beneficial for the project's future.

Secondly, given the limited amount of time to develop and demonstrate decent results, selecting a time-consuming approach in both these aspects would severely hinder the progress towards reaching the second goal of this dissertation.

With these limitations, even acknowledging that the obtained results would not be faultless, Q-Learning was the chosen approach to support the integration of the new self-taught AI for PiTanks.

3.3 System architecture

Having determined that new modules would need to be included within PiGaming, it was necessary to closely observe and adjust, if necessary, the architecture of the system.

At the beginning of the development of this dissertation, the architecture of the system when executed to run with the simulation module was composed of six modules which can be seen in Figure 3.2. The observed diagram was acquired through *rqt_graph* which creates a dynamic graph of what modules are currently running as well as the messages that are being exchanged by them through topics. The arrows show the direction in which messages are being sent in the respective topics.

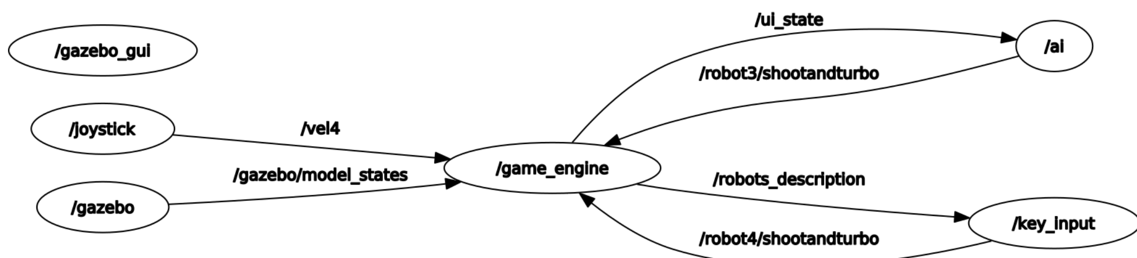

Figure 3.2: PiGaming's initial ROS architecture with the simulation module.

The */game_engine* module is the core of the whole system and where most of the game information is processed and distributed to the other modules. This module is where all game rules are implemented and it is also responsible for the system's UI.

The */joystick* module is in charge of receiving and handling, as the name suggests, the inputs received from the joystick controllers connected to the machine that is running the system. It handles information such as the robot's velocities, linear and angular, as well as if they are shooting bullets or using turbo, which grants a slight increase to their speed.

The */key_input* module is a replacement to the */joystick* and its only purpose is to be used in debug mode, as a means to control the robots through the computer's keyboard thus not requiring

the use joystick controllers. This is its only purpose since it is only able to read one input at a time, making it unfeasible to have multiple players using it at the same time.

The */ai* module is responsible for handling the decision making of the old state machine-based AI and sending its output control messages of one robot to the *game_engine*.

Lastly, the */gazebo* and */gazebo_gui* are, respectively, responsible for simulating the robots and sending their positions to the */game_engine*, and showing their positions to the user.

Considering that the new simulator module is replacing Gazebo, its introduction means the removal of both modules related to Gazebo. Similarly, by introducing a reinforcement learning AI, it would take the place of the previously implemented AI module. These changes result in the proposed architecture shown in Figure 3.3. Despite being replaced, the previously mentioned modules are still implemented within the system, and if proven to be useful in the future, they could and should resurface.

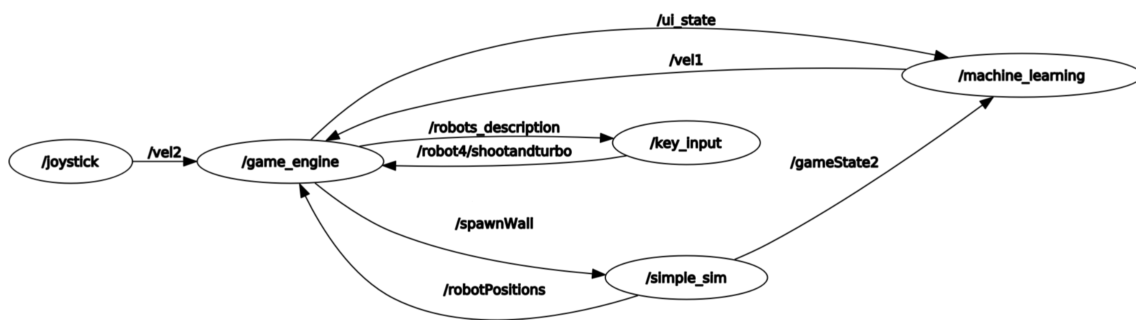

Figure 3.3: PiGaming's new proposed ROS architecture with the new simulation module.

Instead of sending information about the robots' movements to Gazebo, the */game_engine* module now sends it to the new simulator module */simple_sim* as well as all other game-related objects' information. This way, the new simulator will contain all game state information, and to correctly replace Gazebo, its only requirement is to calculate the robots' positions given the inputs sent by the */game_engine* and send them back.

The */key_input* module was slightly modified to allow all robots to be able to shoot bullets. Prior to this change, only the first two robots were able to shoot in debug mode.

The new */machine_learning* module was implemented to continuously receive the game state information from the */simple_sim* module in order to train itself. And the previous AI's topics were also repurposed for this new one in order to fit the */game_engine* module's requirements.

Chapter 4

High performance simulator

In this chapter, the implementation process of the new simulator is described. Besides this, the overall changes to the system to harmonize its communication with the simulator are documented. Lastly, the simulator's performance is compared to Gazebo's.

4.1 Developing a simpler simulator

4.1.1 Objects

Since the goal of this simulator was to have a higher performance in order to stay relevant and expandable within PiGaming, the focus of its implementation was on representing the game state with efficient objects. Considering that all games within the system are 2D with a top-down view, the only logical approach to represent the objects were their horizontal and vertical positions, orientation and their sizes.

Considering that the in-game representation of the robots are circles, it was decided that they would be represented in the same way within the simulator, thus only containing their position, orientation and radius when it came to their physical aspects. However, those three alone are not sufficient to indicate the robots' information that players can see in PiTanks. Thus, by adding their score represented by how many shots they've hit subtracted by how many shots they've taken, the team to which they belong, and finally their id, internally represented within the simulator, all aspects that define one robot are represented.

Additionally, as wasn't required by the game engine when running the games with the physical setup, only robot collisions had to be implemented within the simulator. Since all other kinds of collisions were already implemented within the game engine and the real robots would collide regardless, implementing robot with robot collisions in the game engine was unnecessary but absolutely required in the simulator.

Lastly, given that Gazebo was only able to simulate the PiTanks game, for this dissertation, the only objects besides the robots that were implemented in the simulator were bullets and walls, which are the only other kinds of objects that exist in PiTanks. Figure 4.1 shows PiTanks running

in the simulated environment and the following code blocks show their implementation within the simulator.

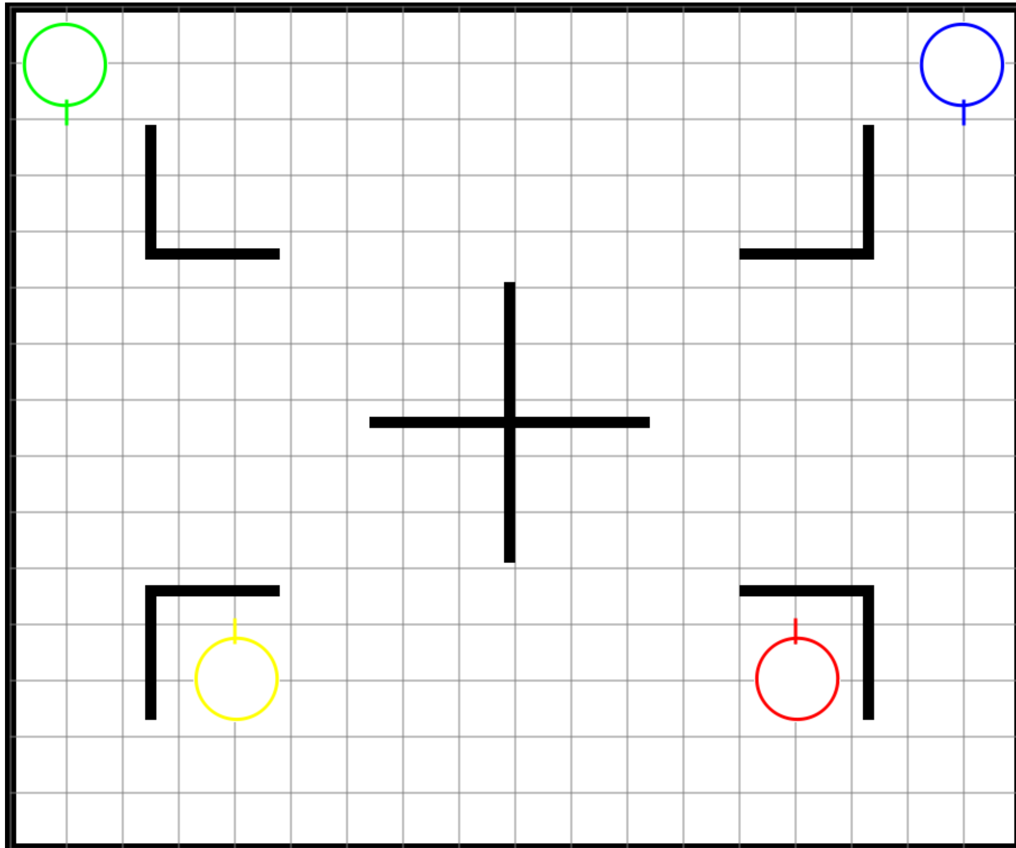

Figure 4.1: PiTanks in the new simulator.

```
class Robot {
    private:
        double x;
        double y;
        double radius;
        double angle;
        int score;
        int teamId;

    public:
        Robot(double x, double y, double radius, double angle, int teamId);
        double getX();
        void setX(double x);
        double getY();
        void setY(double y);
        double getRadius();
        void setRadius(double radius);
        double getAngle();
        void setAngle(double angle);
```

```
        int getScore();
        void setScore(int score);
        int getTeamId();
        void setTeamId(int teamId);
};
```

```
class Wall {
    private:
        pair<double, double> point1;
        pair<double, double> point2;
        double width;
        int health;
        bool indestructable;

    public:
        Wall(double x1, double y1, double x2, double y2, double width, int
            health);
        pair<double, double> getPoint1();
        void setPoint1(double x, double y);
        pair<double, double> getPoint2();
        void setPoint2(double x, double y);
        double getWidth();
        void setWidth(double width);
        int getHealth();
        void setHealth(int health);
        bool isIndestructable();
        void setIndestructable(bool ind);
};
```

```
class Bullet {
    private:
        double x;
        double y;
        double angle;
        int robotId;

    public:
        Bullet(double x, double y, double angle, int robotId);
        double getX();
        void setX(double x);
        double getY();
        void setY(double y);
        double getAngle();
        void setAngle(double angle);
        int getRobotId();
        void setRobotId(int robotId);
};
```

Not including objects from Robot factory or Robot race, however, does not prevent either of them to be played with this simulator since its only requirement is to inform the game engine where

the robots are, it simply means that not all game information for those games are present in the simulator's current state. Figure 4.2 shows Robot Factory running in the simulated environment.

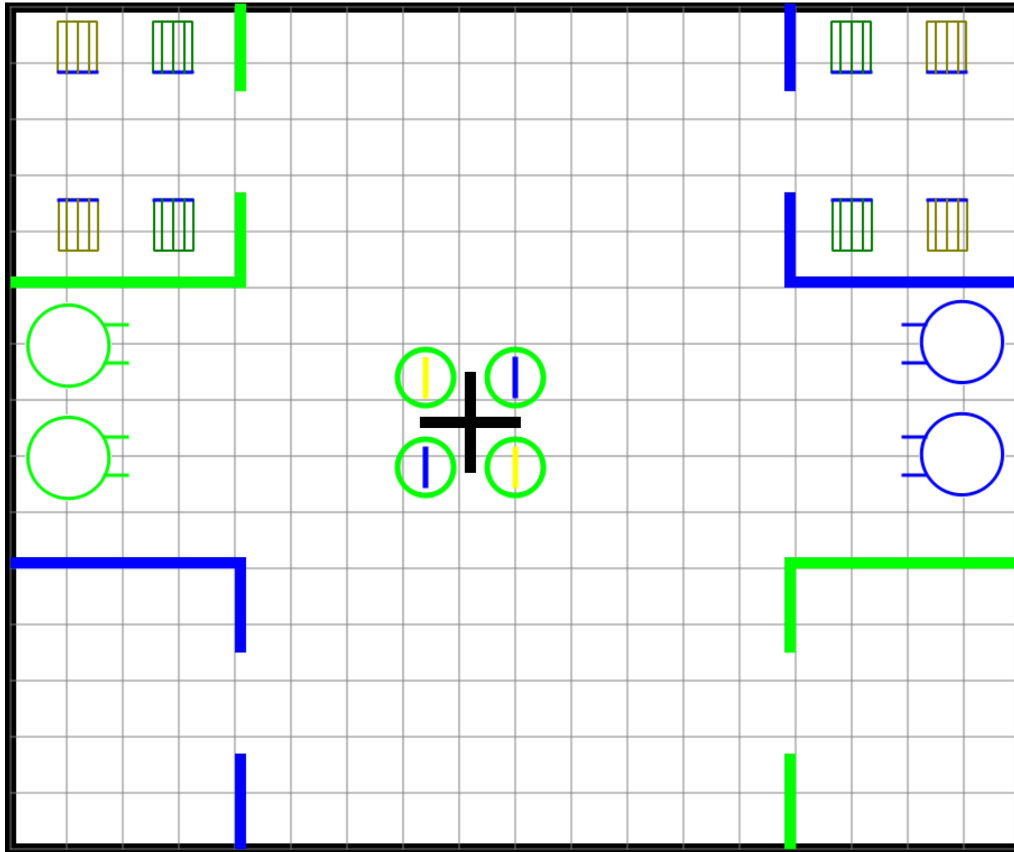

Figure 4.2: Robot factory in the new simulator.

4.1.2 Distance unit

Given the fact that Gazebo's distance unit was the meter, yet the game engine's was the pixel, it would constantly keep converting and deconverting the robot's position and velocity values. The conversion ratios of these values were reasonable considering Gazebo's level of realism, although, their existence not only took a toll on the system which delayed the gameplay but also made it unnatural to compare the robots' positions within the game screen and the simulated screen.

To make up for this issue and remove these conversions, the new simulator's units would mimic the ones existing in the game engine, meaning that a robot's position, orientation and radius would be the exact same both in the game engine and in the high performance simulator. This change would also allow information such as velocity to be kept in pixels/s instead of the SI m/s.

4.2 Communication

Previously, the game engine's only requirement when it came to communicating with the simulation module, was sending information about when to spawn a robot, move a robot, and delete a robot. With the decided change to make the simulator hold all game state information, it was now also required to send information about both the walls and the bullets. Since the game engine already implemented the creation, updates, and deletion of both these objects, the only required addition was to inform the simulator whenever these cases happened. Figure 4.3 shows how this information is processed when the game starts while Figure 4.4 shows how it is processed at every game update.

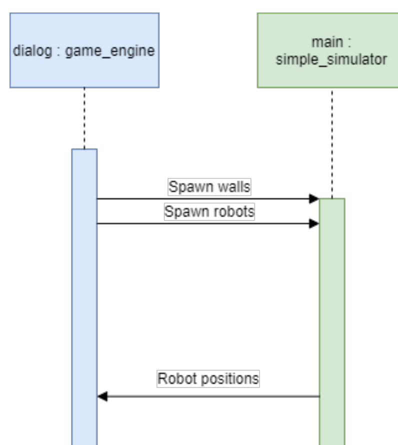

Figure 4.3: Start of game communication.

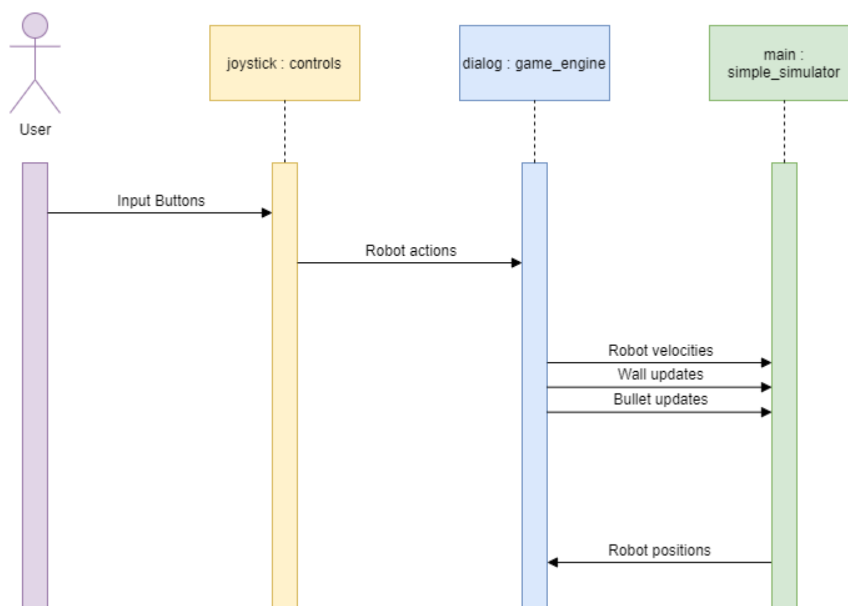

Figure 4.4: Game update communication.

To avoid overflooding the communication topics and to keep the game's frame rate consistent, the messages to the game engine regarding those were sent with a frequency of 50 Hz. This value was specifically chosen to match the frequency at which the */joystick* module sends the controllers' input information to the game engine which is the same value. This consistency results in the objects, both in the game engine to be drawn on the screen and in the simulator, to be updated only in times that there could be a change.

The following code block shows the simulator's implementation and how it handles the messages received from the game engine.

```
void Simulator::insertWall(int id, double x1, double y1, double x2, double y2,
    double width, int health) {
    Wall w(x1, y1, x2, y2, width, health);
    walls.insert(make_pair(id, w));
}

void Simulator::removeWall(int id) {
    unordered_map<int, Wall>::iterator it = walls.find(id);
    // Wall was already deleted
    if (it == walls.end())
        return;
    walls.erase(id);
}

void Simulator::updateWall(int id) {
    unordered_map<int, Wall>::iterator it = walls.find(id);
    // The wall was already removed
    if (it == walls.end())
        return;
    int health = it->second.getHealth() - 1;
    if (it->second.isIndestructable())
        return;
    if (health > 0)
        it->second.setHealth(health);
    else
        removeWall(id);
}

void Simulator::insertRobot(double x, double y, double radius, double angle,
    int teamId) {
    Robot r(x, y, radius, angle, teamId);
    robots.push_back(r);
}

void Simulator::removeRobots() {
    robots.clear();
}

bool Simulator::robotCollision(int id, int x, int y) {
```

```
    for (int i = 0; i < robots.size(); i++) {
        // Same robot
        if (i == id)
            continue;
        // Distance between robots is smaller or equal to their summed radii
        if (pow(x - robots[i].getX(), 2)
                + pow(y - robots[i].getY(), 2)
                <= pow(robots[id].getRadius() + robots[i].getRadius(), 2)){
            return true;
        }
    }
    return false;
}

void Simulator::updateRobot(int id, double x, double y, double angle, int score
    ) {
    // Robot does not exist
    if (id >= robots.size())
        return;
    // Robot collides with object, won't move
    if (robotCollision(id, x, y))
        return;
    robots[id].setX(x);
    robots[id].setY(y);
    // Normalize angle between 0 and 2PI
    double ang = fmod(angle, 2*PI);
    if (ang < 0)
        ang += 2 * PI;
    robots[id].setAngle(ang);
    robots[id].setScore(score);
}

void Simulator::insertBullet(int id, double x, double y, double angle, int
    robotId) {
    Bullet b(x, y, angle, robotId);
    bullets.insert(make_pair(id, b));
}

void Simulator::removeBullet(int id) {
    unordered_map<int, Bullet>::iterator it = bullets.find(id);
    // Bullet does not exist
    if (it == bullets.end())
        return;
    bullets.erase(id);
}

void Simulator::updateBullet(int id, double x, double y) {
    unordered_map<int, Bullet>::iterator it = bullets.find(id);
    // Bullet does not exist
```

```
    if (it == bullets.end())
        return;
    it->second.setX(x);
    it->second.setY(y);
}
```

4.3 Validation and conclusions

Due to the unavailability of PiGaming's physical setup during the development of this dissertation, the results of the developed simulator could not be compared to those of the real system. Thus, the only way to check the simulator's validity was to allow a very limited amount of playtesters to play a few games with both the Gazebo simulator and the new high performance simulator.

These playtesters expressed that the gameplay felt much slower when playing with the Gazebo simulator, which isn't an unforeseen conclusion when considering its low real-time factor, as well as emphasizing that the controlled robot kept performing the players' last inputted action even when they were no longer pressing any buttons, at times to a point where they stopped playing altogether to ask if something was wrong. While with the newer simulator, the playtesters stated that they felt much more in control of their robot and that gameplay was much smoother and fluid.

To further validate the simulator, access to the physical setup is necessary in order to compare how input actions differ when applied to the real robots instead of the simulated ones.

Chapter 5

Artificial intelligence with Q-Learning

In this chapter, the development of the new self-taught AI is detailed.

5.1 Implementing Q-Learning

As the previously implemented AI was only purposed for one versus one matches in PiTanks, the new self-taught one would also be focused in the same scenario as a way to provide a simple method of comparison in relation to the old AI. If the new AI was capable of consistently winning matches against the old AI, then its implementation and addition to PiGaming would be considered as an improvement.

Since this project was developed from its beginning within the ROS framework, it became apparent how complex it was to include certain libraries within the project due to conflicts with the framework itself. To work around this complication, and considering that Q-Learning was the chosen approach to implement the self-taught AI, it was decided that the best solution would be to use an implementation of Q-Learning that did not require any external libraries.

Dr. Humphrys [2] provides an explained pseudocode implementation of Q-Learning that was used as a base to solve the HouseRobot problem. This implementation for the new machine learning node is especially useful due to being created specifically to avoid usage of external libraries and by providing a simple integration within the ROS framework. Besides this, it also allows the usage of multiple agents per robot, and although this feature was not explored in this dissertation, it could prove to be advantageous in future development to create AIs with multiple behaviours.

Following the aforementioned implementation, to represent the state and action vectors, an enumerable vector class was implemented:

```
class EnumVector {
    public:
        // Current vector | vec and c always have the same size
        vector<int> vec;
        // Element limit vector: 0 <= vec[i] < c[i]
        vector<int> c;
```

```
        // No. of possible vectors
        unsigned long long int no;

        EnumVector(vector<int> cvec);
        int getVectorId();
        void setVectorFromId(int id);
        void testIds();
        int& operator[](int i);
};
```

Since this vector is enumerable, meaning that all of its elements are 0 at minimum and c[i] at maximum, every vector is represented by a unique id, which will be useful when defining the Q-Table's dimensions.

To make use of this enumerable vector class, a StateActionSpace class which defines the Q-Table, was implemented:

```
class StateActionSpace {
    protected:
        // All  Q values, accessed with x and a Q(x,a)
        vector<vector<float>> vec;
        // The state vector
        EnumVector* xf;
        // The action vector
        EnumVector* af;
        string filename;

    public:
        StateActionSpace(vector<int> cvec, vector<int> dvec, string path,
            string filename);
        float at(EnumVector x, EnumVector a);
        void increment(EnumVector x, EnumVector a);
        void set(EnumVector x, EnumVector a, float value);
        float max(EnumVector state);
        long int totalNoOfExperiences();
        void saveSpace(string path);
        void printSpace();
};
```

Having implemented a Q-Table, it was finally necessary to implement the Agent class that would access and update its values:

```
class Agent {
    protected:
        // These values retain a "somewhat" stochastic policy
        const float gamma = 0.6;
        const float maxQTemperature = 1.0 / 2;
        const float minQTemperature = 1.0 / 50;
        const long int ceiling = 100000;
        float sigma;

```

```
        // Each (x,a) has its own varying alphaQ
        float alphaQ(long int i);
    public:
        // Keep track of Q-values for my actions
        StateActionSpace* Q;
        // noQ(x,a) counts the no. of times we have "visited" (x,a)
        // (Tried action a in state x)
        // so we can have declining alphaQ
        StateActionSpace* noQ;
        // The action I suggest to execute
        EnumVector* ai;
        // Temporary action variable
        EnumVector* af;

        Agent(vector<int> cvec, vector<int> dvec, string path, string filename)
            ;
        // The reward function is what defines me
        // (defined in subclasses)
        virtual float reward(EnumVector state, EnumVector action, EnumVector
            newState);
        int randomAction();
        void updateQ(EnumVector state, EnumVector action, EnumVector newState);
        // The sum of the exp(Q/T) terms
        void calculateSigma(EnumVector state, float qTemperature);
        // Shows how probable each action is for the given state
        void printProb(EnumVector state, float qTemperature);
        // Suggests an action ai
        void suggestBoltz(EnumVector state, float qTemperature);
        // Suggests action with reasonable (declining) temperature
        void suggestReasonable(EnumVector state);
        float reasonableTemperature();
        // No exploration, demo mode
        void exploit(EnumVector state);
        void saveQTable(string path);
};
```

The temperature parameter is used to determine if the agent will explore or not and it is determined by how much the agent has explored before. As such, if the agent has yet to considerably explore the state space, it will most likely explore actions it hasn't yet since their probability is increased due to the high temperature. Consequentially, if the agent has explored most of the state space, the temperature will be low, and thus, the agent is most likely to select the most beneficial action, exploiting what it has already discovered.

5.1.1 Communication

As this implementation itself does not contain the environment from which it is meant to learn from, it was required for it to gather information from the new simulator, which was developed

with the consideration that it would contain all game state information. Thus, all that was needed was for the communications to be made so that:

- The Q-Learning module could access the game state information at any given moment;
- The game engine module received the actions that the Q-Learning module decided to execute.

With this in mind, the communications created between these modules are shown in Figure 5.1.

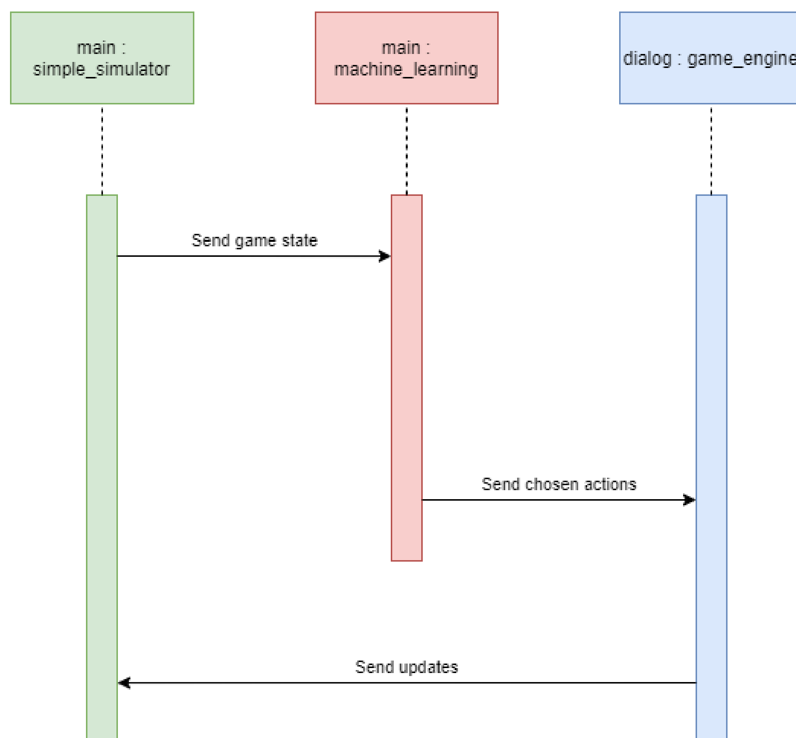

Figure 5.1: System communications regarding machine learning.

5.2 State and action spaces

Having gathered the game state, it was still necessary to define which features were most relevant for the Q-Learning algorithm to learn from. This lead to an issue where if too many features were taken into account, the resulting Q-Table would have gigantic proportions due to including too many different possible states and would always result in the module crashing.

With this, three features were determined to be most relevant and were the ones that were taken into account when building the Q-Table. These features were: The *direction to the opposing robot*; the *direction to the closest inner wall*; and the *direction to the closest incoming bullet*. All of these directions were discretized between the values of 0 and 15 inclusive, this was both to reduce the number of possible states while keeping a decent enough representation. They were

also determined by which direction the robot was currently facing, meaning that if the opposing robot is directly across from the main robot, the value for the *direction to the opposing robot* is 0 regardless of the position or orientation of each. Additionally, the *direction to the closest inner wall* and the *direction to the closest incoming bullet* had an additional value of 16. This value was used when there were no more inner walls left and no incoming bullets. Figure 5.2 shows how the numerical representations of these directions function, these directions are not restricted by distances, meaning that they merely represent from which direction an object is in comparison to the robot regardless of how distant it is to said object.

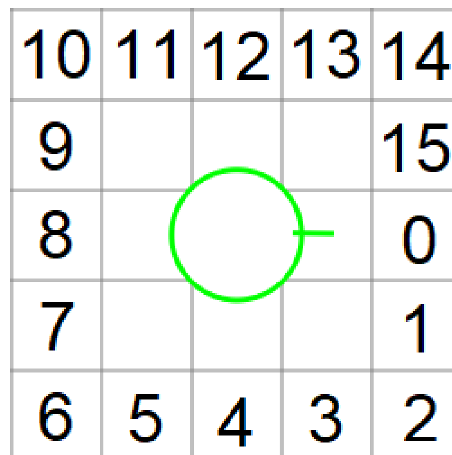

Figure 5.2: Numerical representations of the robot's directions.

With the state space defined, it was needed to define which actions the robots could execute. This was a much simpler process since the actions that any robot can perform in PiTanks were already well defined. In total, there are eighteen possible executable actions at any given moment. These are the result of combinations of three individual actions:

- Linear movement: Moving forward, backward, or not moving;
- Angular movement: Rotating left, right, or not rotating;
- Shooting bullets: Shooting or not shooting;

With both state and action spaces defined, the Q-Table was ready for the module to update its Q-values. There was only one last step to define these updates.

5.3 Reward function

Given the limited amount of features used to define the state, defining a good reward function was a critical part to ensure proper learning. With this, two different agents were trained each with its own reward function.

In the first agent, to tell if the chosen action was beneficial or not, considering that there is only access to the three aforementioned directions, the reward for each action consists of the three factors shown in Table 5.1.

Table 5.1: Rewards determined by the first AI's reward function.

Scenario	**Reward**
Shooting when the opposing robot is in line of sight and not in the same direction as the closest wall	10
The opposing robot is not behind but gets behind with closest wall not in between	-1
Shooting when not looking at opposing robot	-1

Despite being a considerably higher value when compared to the others, the reward 10 for shooting in the direction of the opposing robot while the closest wall is not in the same direction was chosen to ensure that the agent would highly consider states where shooting the opponent were more likely to happen. If the opposing robot and the closest wall are not in the same direction and the agent's direction to the opposing robot is not in between 4 and 12, but is in between 4 and 12 in the next state, then the robot is penalized for facing away and possibly conceding shots.

This reward function, although lacking in rewarding defensive behaviours, was found to be decently suited for an aggressive offensive behaviour, while making sure that the robot isn't constantly shooting randomly, which although there is no inherent penalty for doing so, would lead to an unfun and boring game experience.

For the second agent, Table 5.2 shows how the second reward function determines the value for each of the agent's actions.

Table 5.2: Rewards determined by the second AI's reward function.

Scenario	**Reward**
Shooting when the opposing robot is in line of sight and not in the same direction as the closest wall	10
The opposing robot is behind with the closest wall not in between	-1
Shooting when not looking at opposing robot	-1
Staying still	-10

In contrast to the previous reward function, instead of punishing the agent once for letting the opposing robot get behind it, this reward function keeps punishing the robot as long as the opposing robot is behind it, thus making it select actions that prevent the opposing robot both from getting behind the agent and from staying behind it. Besides this, and a significantly more important factor, stillness was added into consideration to be an extremely negative behaviour both for leading to detrimental states where opposing robots would be given the advantage and to stalling gameplay.

5.4 Performance and conclusions

5.4.1 Evaluation against previous AI

Having the machine learning module fully developed, it was time to begin the training phase of the two aforementioned agents. To do so, each agent trained against itself in 200 two minute matches in order to sufficiently explore the several possible states of the game and update their Q-Tables accordingly to the given rewards they received.

After this stage, the now trained agents were tested by facing the previously implemented state machine-based AI in 100 two minute matches. The results of both agents' performances in these matches are shown in Table 5.3:

Table 5.3: Agent results with each reward function against previous AI.

Not punishing stillness	Number of games
Wins	32
Ties	68
Losses	0

Punishing stillness	Number of games
Wins	94
Ties	6
Losses	0

Although never losing to the previously developed AI, the first agent's reward function did not take into consideration that staying still would lead to a non-negative long-term reward while most other actions would. As such, in 68 of its games against the previous AI, both players stayed hidden for the full duration of the matches with neither ever taking a shot at the other. This scenario can be seen in Figure 5.3 which is a screenshot of one of the 68 matches where no player ever shot and stood still in their respective positions.

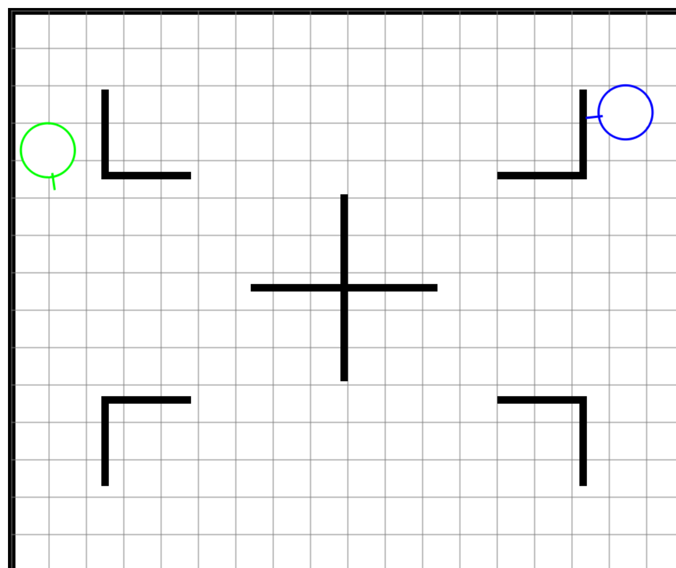

Figure 5.3: First agent (blue): Match where both players kept standing still.

As opposed to the first agent, the second one with the updated reward function that punished stillness was much more aggressive and prone to approaching its opponent, which due to the previous AI being more prone to waiting for the opponent's approach, would lead to the agent consistently getting the first shot and keep bombarding its opponent until the end of the game, this lead to it winning most of its matches. Figure 5.4 shows one of the 94 matches where the agent managed to get a few shots before the previous AI could retaliate.

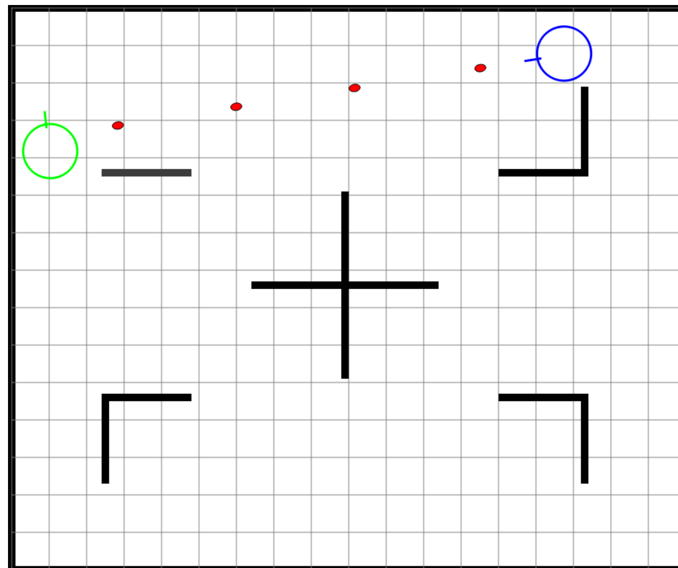

Figure 5.4: Second agent (blue): Match where the agent bombarded the previous AI early on.

It is worth noting that both agents and the previous AI would at times get stuck in the walls due to bugs in the collision detection of the game engine. This issue was a slight setback in both training and evaluation of both agents due to its unpredictability and is one of the causes for some matches resulting in ties.

With the obtained results, the second agent was determined to be superior since it was more proactive and engaging, instead of stalling.

5.4.2 Evaluation against human players

To further evaluate the second agent's performance, it was decided to let two playtesters play both against it and the previous AI to determine if the new agent was decent enough when facing actual human players.

The first playtester, who had a decent amount of knowledge and experience regarding video games, was able to beat both the previous AI and the agent in every single match that they played. When matched against the previous AI, this playtester noted that it had a "terrible" habit of hiding behind the same wall, which lead to them easily defeating it.

When questioned about what they thought of the agent's actions, they expressed that it was significantly more challenging and that they were taken a bit by surprise by how aggressive it was

in comparison. Although the agent was able to hit this playtester a few times, they commented that the agent, despite being less predictable than the previous AI, would leave itself open to shots by being overly aggressive.

The second playtester, in contrast, had little to no experience in video games, and in most matches would sometimes stop looking at the game screen to look at which controls they were pressing. Still, when facing the previous AI, this playtester quickly realized how naive the AI's actions were and it took them little effort and time to figure out how to easily defeat it on their very first match and thus also never losing against it. Against the agent, however, this playtester struggled to keep up with its barrage of shots and never managed to defeat it.

When inquired about their thoughts on both AIs, this playtester stated that the previous AI's behaviour was "a little boring" but a good way to introduce new players to PiTanks. In comparison, they voiced that the agent was too overwhelming for them.

With these results, it was concluded that the new self-taught AI had the capacity of defeating players that were still adapting and learning the game while still being beatable by players with more general experience with video games.

Chapter 6

Conclusions and Future work

By the end of this dissertation, the simulation and artificial intelligence ROS modules in the PiGaming system had received a significant overhaul. From the introduction of a higher performance simulator which replaced the previously implemented Gazebo one, to a machine learning component that could make use of this higher performance to efficiently train machine learning agents that were able to play PiTanks.

The new high performance simulator was able to smoothly run not only PiTanks but also Robot factory and Robot race with a consistent refresh rate of 50 Hz thus preserving the playability of the system's games in a simulated environment and allowing for more convenient debugging by not requiring the use of the system's physical setup and real robots to test the implementation of new features and games in the system.

The new machine learning component focused on Q-Learning was able to take advantage of the simulator's performance as the environment to train two different kinds of agents within an acceptable time frame and properly communicate these agent's actions to the system's game engine, thus allowing the trained AIs to play against real players in the simulated environment. There were however at times issues with the collision detection between the robots and the walls within the game engine, which would occasionally lead to the robots getting stuck in the walls.

The results of the new additions to the system were validated by two playtesters that experienced the system's gameplay with both its previous simulation and AI components and the newly developed ones. These results were limited to the opinions of these playtesters due to the restricted access to the system's physical setup and inability to allow more users to try the upgrades as a consequence of the confinement order from the pandemic outbreak of COVID-19.

It is thus concluded that PiGaming as a system benefits substantially from having an efficient simulator not only to its players, but also to its developers. In regards to the artificial intelligence, however, although there was a step forward in its improvement, there is still room for further upgrades and unexplored machine learning strategies that could lead to more engaging AIs.

6.1 Future work

When it comes to the further development of PiGaming's system as a whole, there are several explored and unexplored areas that could use improvements. As such, the following bullet points provide insight on how the system could be upgraded in the future:

- Simulation fidelity with the physical setup: Despite being faster than the previous simulator, the new high performance simulator is not guaranteed to enact the games exactly the same as the physical system would. As such, data should be gathered both from running the physical setup and the simulated one to ensure the simulator's fidelity;
- Deep Learning AI: Due to time restrictions from the development of the high performance simulator, this area of machine learning was left unexplored for the system. However, it has shown multiple successes in the areas of robotics and gaming and could lead to a more accurate representation of the game's state when compared to Q-Learning and should result in the creation of more challenging agents. LibTorch is suggested as a possible machine learning library to use within PiGaming for including multiple deep learning algorithms;
- Wall collisions: Since these are handled within the system's game engine itself it was not explored in this dissertation. However, it did slightly limit the development of the AIs and is thus suggested to be looked into as a way to provide future improvements to the AI and to less buggy gameplay;
- Gaming depth: Introducing more depth to the games, via power-ups, for instance, could lead to creating more engaging experiences and increase the games' replayability.

References

[1] Polulu 3pi. https://www.pololu.com/product/975. Accessed: 2020-06-28.

[2] Sample code for q-learning. https://computing.dcu.ie/~humphrys/Notes/RL/Code/index.html. Accessed: 2020-04-21.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[4] Kai Arulkumaran, Antoine Cully, and Julian Togelius. Alphastar: An evolutionary computation perspective. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '19, page 314–315, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Hugo Costa, Peter Cebola, Tiago Cunha, and Armando Sousa. A mixed reality game using 3pi robots—"pitanks". In *2015 10th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6. IEEE, 2015.

[6] Sérgio Daniel Marinho de Lima Teixeira. Simulação e melhoramento do pitank com sistema de inteligência artificial. Master's thesis, Faculty of Engineering of University of Porto, 2019.

[7] Sergio Garrido-Jurado, Rafael Muñoz-Salinas, Francisco Madrid-Cuevas, and Rafael Medina-Carnicer. Generation of fiducial marker dictionaries using mixed integer linear programming. *Pattern Recognition*, 51, 10 2015.

[8] Ulit Jaidee and Héctor Muñoz-Avila. Classq-l: A q-learning algorithm for adversarial real-time strategy games. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[9] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[10] Shivaram Kalyanakrishnan, Yaxin Liu, and Peter Stone. Half field offense in robocup soccer: A multiagent reinforcement learning case study. In *Robot Soccer World Cup*, pages 72–85. Springer, 2006.

[11] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. Robocup: The robot world cup initiative. In *Proceedings of the first international conference on Autonomous agents*, pages 340–347, 1997.

[12] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[13] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE, 2004.

[14] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[15] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław 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, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019.

[16] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8026–8037. Curran Associates, Inc., 2019.

[17] Lenka Pitonakova, Manuel Giuliani, Anthony Pipe, and Alan Winfield. Feature and performance comparison of the v-rep, gazebo and argos robot simulators. In *Annual Conference Towards Autonomous Robotic Systems*, pages 357–368. Springer, 2018.

[18] E. Rohmer, S. P. N. Singh, and M. Freese. Coppeliasim (formerly v-rep): a versatile and scalable robot simulation framework. In *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013. www.coppeliarobotics.com.

[19] Francisco Romero-Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. Speeded up detection of squared fiducial markers. *Image and Vision Computing*, 76, 06 2018.

[20] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[21] Stanford Artificial Intelligence Laboratory et al. Robotic operating system, 2018.

[22] Peter Stone, Richard S Sutton, and Gregory Kuhlmann. Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

[23] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.