TotalBotWar

A New Pseudo Real-time Single-action Game Challenge and Competition for AI

Sergi Fuster Durà

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

May 22, 2023

Supervised by: Raúl Montoliu, PhD.

Acknowledgments

First of all, I would like to thank my Final Degree Work supervisor, Raúl Montoliu Colás, for being the guide and director of this project, and for solving all the doubts related to the development that I have had during these months.

I would also like to thank the help and company of two good friends Anatoliy Myndresku and Llorenç Lavernia for giving me their ideas and opinions about this project.

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

Abstract

This work presents TotalBotWar, a new pseudo real-time single-action challenge for game **AI** for **mobile devices**, as well as some initial experiments that benchmark the framework with different agents. The game is based on the real-time battles of the popular TotalWar games series where players manage an army to defeat the opponents one. In the proposed game, a turn consists of an order to control one of your units. One interesting feature of the game is that if a particular unit does not receive an order in a turn, it will continue performing the action specified in a previous turn. The turn-wise branching factor becomes overwhelming for traditional algorithms and the partial observability of the game state makes the proposed game an interesting platform to test modern AI algorithms.

It should be added that it is not necessary to know about programming to play, also the manual game mechanics have been implemented in which you can control your troops with the mouse.

Finally, for reasons that will be explained in the following chapters, the structure of the developed system is not the conventional one, but a Cloud Gaming [26] style structure has been necessary.

Contents

Contents vii

List of Figures ix

List of Tables xi

1 Introduction 1
- 1.1 Work Motivation . . . 2
- 1.2 Objectives . . . 3
  - 1.2.1 Main Goal . . . 3
  - 1.2.2 Specific Goals . . . 4
- 1.3 Environment and Initial State . . . 4
- 1.4 Game Overview . . . 6
  - 1.4.1 Main Characteristics . . . 7

2 Planning and resources evaluation 9
- 2.1 Planning . . . 9
- 2.2 Resource Evaluation . . . 10

3 System Analysis and Design 13
- 3.1 Requirement Analysis . . . 13
  - 3.1.1 Functional Requirements . . . 13
  - 3.1.2 Non-functional Requirements . . . 15
- 3.2 System Design . . . 17
  - 3.2.1 Security . . . 17
  - 3.2.2 Multi-User . . . 18
  - 3.2.3 Documentation . . . 19
  - 3.2.4 Readability . . . 20
- 3.3 System Architecture . . . 20
  - 3.3.1 Hardware Requirements . . . 20
  - 3.3.2 Other Requirements . . . 20
- 3.4 System Analysis . . . 21
  - 3.4.1 Action Space . . . 21
  - 3.4.2 State Representation . . . 21

3.4.3 Game Complexity 22
3.4.4 Baseline Agents 22
3.5 Interface Design 24
3.6 Videogame Art 25

4 Work Development and Results 29
4.1 Work Development 29
4.1.1 Initial Conception 30
4.1.2 Game Core 30
4.1.3 Server 31
4.1.4 Database 31
4.1.5 Unity 32
4.1.6 Firebase Database 33
4.1.7 Socket Implementation 34
4.2 Results 38
4.2.1 Goals Achieved 38
4.2.2 Comparison between Planning and Final Work Accomplished . . . 42
4.2.3 Applications of the Work Performed 42

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

Bibliography 49

A Source code 51
A.0.1 GitHub 51
A.0.2 Game State in JSON 52

List of Figures

1.1 Bonus Type Diagram . . . 8

2.1 Planning Table Start . . . 10
2.2 Gantt Chart . . . 11

3.1 Authentication and Career Conditions Security . . . 18
3.2 Use of Pipelines for parent-child Communication . . . 19
3.3 TotalBotWar Menu Interface . . . 24
3.4 TotalBotWar Positioning State Interface . . . 24
3.5 TotalBotWar Battle RND vs OSLA . . . 25
3.6 Knight Running Animation . . . 25
3.7 Archers Attacking Animation . . . 25
3.8 Swordsmen Death Animation . . . 26
3.9 Background N.1 . . . 26
3.10 Background N.2 . . . 27
3.11 Background N.3 . . . 27

4.1 Main Loop Abstract . . . 36
4.2 Game Step 1/2 . . . 36
4.3 Manage Intersection . . . 37
4.4 Game Step 2/2 . . . 37
4.5 Update of Data and Rendering Loop . . . 38
4.6 Random vs OSLA Ingame Background N.2 . . . 39
4.7 Random vs OSLA Ingame Background N.3 . . . 39
4.8 Server and Game Working While Positioning . . . 40
4.9 Server and Game Working While Battle . . . 42
4.10 Planning Table Finish . . . 43

List of Tables

1.1 Subjects Related to the Project 4
1.2 Attributes of Military Units 7
1.3 Available Resolutions on TotalBotWar Game 7

3.1 Functional requirement «SETUP1. Game Teams» 14
3.2 Functional requirement «SETUP2. Units Positioning» 14
3.3 Functional requirement «ACTION1. Unit Selection» 14
3.4 Functional requirement «ACTION2. Destination Setting» 15
3.5 Non-Functional requirement «SECURITY1. Malicious Third-Party Person» . 15
3.6 Non-Functional requirement «SECURITY2. Career Conditions» 15
3.7 Non-Functional requirement «MULTIUSER. Concurrent Programming» . . . 16
3.8 Non-Functional requirement «DOCUMENTATION. Docstring» 16
3.9 Non-Functional requirement «READABILITY1. Clean Code» 16
3.10 Non-Functional requirement «READABILITY2. PEP 8 Python» 16
3.11 Number of possible actions for 8 units depending on the battlefield size. . . . 22

Chapter 1

Introduction

Contents

1.1 Work Motivation **2**
1.2 Objectives **3**
1.3 Environment and Initial State **4**
1.4 Game Overview **6**

In recent years, games have proven to be important testbeds for Artificial Intelligence (AI). For instance, deep reinforcement learning has enabled computers to learn how to play games such as Chess [22], Go [22], Atari games [13], and many other games [9]. Despite these important advances, there are still games that pose important challenges for state-of-the-art AI agents. Some examples are Blood Bowl [10], Legend of Code and Magic [11], MicroRTS [14], FightingICE [8], Hanabi [24], Splendor [2], StarCraft [3], and the General Video Game AI framework [18], among others.

In this document, we propose TotalBotWar, a new pseudo realtime challenge for game AI. The game is inspired by the real-time battles of the popular TotalWar game series[1], where two players control respective armies with the objective of defeating each other. On each turn, the agent must decide where the unit must move to. When two opposite units collide, they will start to fight. The result of the combat depends on the type of units and their attributes. If during a turn a unit does not receive any order, it will continue its movement following the previous one, or it will stand still if none was given. This introduces unknown information on the state: it is possible to know that an enemy unit is moving, but not its destination. The game has a high number of possible

[1]Creative Assembly, https://www.totalwar.com/

actions in a turn ($\approx 6.7 \times 10^7$) and also a huge number of possible states ($\approx 3.3 \times 10^{29}$), which provides a significant challenge for AI agents. An initial set of experiments are also presented, where four different agents are benchmarked to give a baseline to future researchers. Three of them are primary agents where a) units never move (but can fight), b) always move forward, or c) move to a random localisation. The one remaining is more sophisticated, applies human knowledge by using a heuristic function. It has also been implemented the possibility to play as a human and control the troops yourself in real time. It is important to mention that this work complements and builds on the base of the game developed by Alejandro Estaben [4] (alumnus of the degree), keeping the theme this time the game is available for mobile devices, it is possible to play in human mode and you can choose which AI algorithms will be confronted (future work will add even more changes).

This chapter aims to provide an insight into the motivations that drove the selection of this project, along with its objectives and starting point. It is crucial to clarify that the project idea originated from Raúl Montoliu, and it is part of a collaborative grant that I am undertaking with him. To enhance your understanding of the upcoming discussions, I kindly invite you to watch a brief video showcasing the gameplay of the video game. Several videos can be saw by clicking in the following links:

- Video: TotalBotWar - Human vs OSLA
- Video: TotalBotWar - Random vs OSLA
- Video: TotalBotWar - DEBUG

By doing so, you will have a better grasp of the topics we will be addressing throughout the chapter.

1.1 Work Motivation

> “ *Artificial intelligence (AI) has advanced significantly in recent decades, but there are still many challenges that remain to be solved.* “

The motivation for this project is primarily the intrinsic motivation that exists in all work related to computer technology, discovery and research. CHAT GPT [15], DALL-E [16], Bard [7]... All these are AI technologies that have revolutionized or will revolutionize the world, and we have reached this point of development thanks to thousands and thousands of researchers who, more or less, did their bit to develop all the knowledge necessary for this goal. Like them, Raul and I want to do our bit in this field because we believe it has a promising future and there is still a lot to discover.

Think about it, even if this project is not a direct help to the world of artificial intelligence research, it can be one of the dominoes that will trigger important future results. Interest in this world may not only become stronger in me after the development of this project, it may also be born and grow in all those people who play the game and make bots for it.

Also, in the world of video games there are many unmet or unfinished goals related to AI. We would all like to be able to play games and face human-like enemies that test our capabilities and help us improve. Even bots that would discover new ways of playing games never seen before. It would also be very desirable not to have to configure the bots ourselves, but for them to be able to adapt themselves to the level of the player and create an ideal learning curve.

1.2 Objectives

This section sets out the objectives prior to the completion of the project, their resolution can be consulted in the Section 4.2.

1.2.1 Main Goal

The objectives are to develop a functional project, that really is a realistic option to learn AI focused on programming bots that play video games and are able to win. Not only can it be a source of fun, but it can be a real tool to learn something that a game developer needs.

The fact that the game is playable and entertaining can lead to many benefits. All video game companies need workers specialized in AI to develop bots for their games, bots that always lose, bots that always win, bots that are interesting, if you are good in this field you will always have a job. On a larger scale, the world needs a lot of AI experts to open new frontiers and solve problems. Like a virus that spreads exponentially, the taste for AI that is generated in each person will be a potential source of contagion for many others, with the consequence of helping the development of this field.

Another important long-term goal we have is to implement the possibility of online (multiplayer) play. This will allow us to implement our own Turing test. The Turing Test is a method proposed by the British mathematician Alan Turing in 1950 to determine whether a machine can demonstrate human intelligence. The basic idea is that if a machine can carry on a conversation indistinguishable from that of a human, then that machine can be considered to have artificial intelligence. At the end of a game the user must answer the question "Have you played against a human?". The answers and their result will be stored to generate data that may be useful in the future.

1.2.2 Specific Goals

“ To build a good wall, you should put one brick a day as best you can do it, after 1 year you will have a wall formed by 365 bricks. It will probably be the best wall in the world, and your effort to achieve it has been very little. “

To achieve the main objective explained in the previous section of this same chapter, we established a series of more specific objectives.

- G1. Make a game easily accessible to everyone... How many people do you know who don’t have a cell phone? That’s what I mean, the most effective way to make a software product accessible today, is to make a smartphone application. So we set as an important goal to develop the game for mobile devices, both Android and iOS.
- G2. Make it easy to add new algorithms to the game. That is, to have a flexible and scalable game.
- G3. Have bots that used the most advanced AI techniques as well as the most complex ones that were able to challenge humans.
- G4. We wanted the user to be able to choose from all available algorithms.
- G5. To put into practice the knowledge acquired during the course of study.

In the Table 1.1 the related subjects to the project can be find.

Closely Related Subjects	**Related Subjects**
Programming 1	Databases
Programming 2	Operating Systems
Networks and Multiplayer Systems	Game Engines
Algorithms and Data Structures	Mathematics I

Table 1.1: Subjects Related to the Project

1.3 Environment and Initial State

The initial project requirements were:

- The basic logic of the game had to be programmed in Python.
- The game had to be for mobile devices.

As previously mentioned, this game is based on a preliminary version developed by Alejandro Estaben (Programmer) and Cesar Diaz (Artist) for the *CodinGame* online platform [2]. The project was entirely developed in Java using the *CodinGame* SDK [3]. The game does not feature a human mode and instead operates on a league system, where the participating bots are pre-defined, resulting in limited flexibility and restricted access to specific bots for certain segments of the public. Furthermore, the game was not officially released for public use and could only be accessed through the provided link. As a result, the game remained highly inaccessible to the general public.

At this point, we knew few things, among them that we had to:

- Make the core in Python: I was able to take advantage of some things from Alejandro Estaben's code but most of them had to be replaced to adapt them to the project or because I thought it was appropriate to change them.
- Make the game for mobile devices.
- Allow the user to choose the bots that were going to play the game.
- Implement the human mode.

And left to my choice the rest decisions like the engine we were going to use to develop the mobile app, the way to communicate the app with the game in Python, and all the other less weighty decisions involving all the main tasks.

My initial decisions were different from the ones that have ended up being implemented due to complications in the development of the initial ideas. Initially the idea was to communicate the game and the app through a REST API, the game would store the states in a database and a server would be in charge of extracting the data and sending it to the clients, i.e., the mobile application, through http requests, would obtain the game states. We also contemplated the idea of using Flutter to develop the client but we ended up changing it for Unity since I know how to use it much better.

My way of organizing myself for this project was based on the use of Trello, here I wrote down the tasks I had to do, the ones I was doing, the ones that were done, the bugs and the decisions that were to be taken. Once a week Raul and I had a meeting where he saw the status of the project and I presented him my doubts and decisions, he gave me his opinion and clarified my ideas.

I have used GitHub as a code management and control tool.

[2] *CodinGame*, https://www.codingame.com/
[3] *CodinGame* SDK, https://www.codingame.com/contribute/view/486222077fe22e3aa6bcdc0f729dd46223bb

1.4 Game Overview

TotalBotWar is a 1 vs 1, pseudo real-time, single-action game, partially inspired in the real-time battles of the Total War games series. In our game, both players start with the same number of military units and their objective is to defeat the other player. The winner is the player who first destroys all the opponents units or the one with more units alive on the battlefield when the maximum time is reached, which is set to 180 seconds by default.

There are five different types of units: Swordsmen, Spearmen, Archers, Knights and General. The game uses a rock-paper-scissors-lizard-spock [4] (see figure 1.1), swords beat spears, spears beat horses and horses beat swords, in addition, there are 2 special units that have other utilities, the archers who are weak in melee against all other units but can attack at a distance and in area, and the general who is like a soldier with a sword but also powers all units that are close, the powered units will have an extra in some of its attributes: damage, range, defense, speed, resistances...

The directions in which the units collide and their movement (see Figure 4.3), if your units are stationary and another unit collides with them from behind your units will instantly suffer a lot of damage while the enemy will not suffer damage, we call this "charge damage". The charge damage is only executed when the attacking unit is moving, besides this damage a bonus is added depending on the side from which it hits, if it hits from the front there will be no bonus but if it hits from behind there will be a bonus. The damage done by this charge will not only depend on the direction bonus and the type bonus, but also on the specific resistances of the affected person and the damage per charge of the unit that executes it. These are the attributes that define the damage per charge done and resisted by the units.

Each unit has an attribute vector modelling its behaviour. The attributes are Health Points, Attacking Strength, Defence, Charge Power, Charge Resistance, Moving Speed and defence against Arrows. Besides, archers also have Throwing Distance and Arrow Damage. Table 1.2 shows the values assigned to each attribute for each unit type.

Units can move to any place of the battlefield. Two units from the same agent can overlap, while they will fight if belonging to different armies. If an unit reaches the limits of the battlefield, it stops. Archers always shoot arrows to enemy troops into the attacking range. Troops suffer friendly-fire if they are close to an opponent unit receiving arrows.

[4] The game 'Rock, Paper, Scissors, Lizard, Spock' became popular due to its inclusion in the television series 'The Big Bang Theory,' created by Chuck Lorre and Bill Prady

Table 1.2: Attributes of Military Units

Attribute	**Swordsmen**	**Spearmen**	**Knight**	**Archer**
Health Points	250	250	200	100
Attacking Strength	20	15	12	10
Defense	10	20	12	5
Charge Power	5	10	100	5
Charge Resistance	25	125	15	0
Moving Speed	15	10	40	15
Defense against Arrows	10	30	30	10
Throwing Distance	-	-	-	450
Arrow Damage	-	-	-	20

The size of the battlefield depends on the size chosen by the user in the menu screen, in this screen a series of predefined resolutions are given, in Table 1.3 you can consult a table with some of these resolutions.

Table 1.3: Available Resolutions on TotalBotWar Game

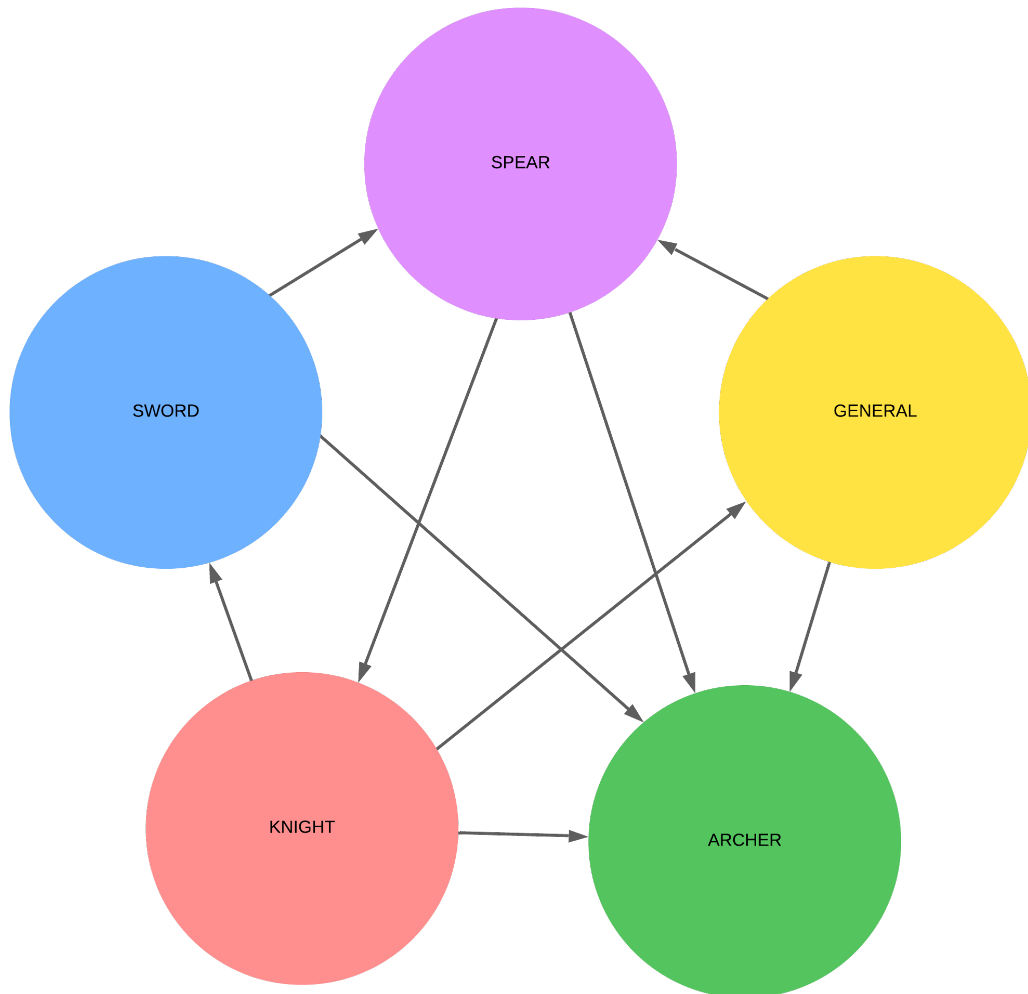
Index	**Horizontal Size**	**Vertical Size**
1	1000 px	500 px
2	2000 px	1000 px
3	750 px	750 px
4	1500 px	1500 px

1.4.1 Main Characteristics

The main characteristics for game AI are as follows:

- It is a 1 vs 1 game.
- It is (pseudo) real-time. Although the game engine performs actions in the order indicated in the turn, the effect of this order is practically negligible. Similarly, the effect of which player performs first the actions is minimal.
- Not all information is known in the state. The state contains information about the actual position of the enemy units and if they are moving or not, but it does not provide information about the final target where they are moving.
- It is single-action since in the same turn just one action can be performer, one unit can be assigned to just one destination.
- The agents have just 100ms to decide the actions to be executed on each the turn.
- It has a very large number of possible actions in a turn ($\approx$ 6.7E7) and possible states ($\approx$ 3.3E29).

Figure 1.1: Bonus Type Diagram



- Human mode is available, if you want to play in real time and manage the troops yourself you can do it.

CHAPTER 2

Planning and resources evaluation

Contents

2.1 Planning 9
2.2 Resource Evaluation 10

In this chapter we will focus on how to distribute the tasks as well as the time dedicated to each one. In addition, we are going to make an analysis and evaluation of the resources of the resources used for the elaboration of the project, both those that were available at the beginning and those that have been obtained during the development of the project. development.

2.1 Planning

For the preliminary planning of the project we used a Google Excel (presented in the Game Design Document) in which the expected tasks and their cost were exposed. Although the final result has been quite close, there have been some changes, both in the tasks and in the time spent for their completion. For example, in the original plan some tasks were shown as "DB setup", referring to the implementation of a database, which was finally discarded and replaced by another communication methodology. Between this method and the finally implemented one, other alternatives were tested which were also discarded but were time consuming, these tasks are not shown in the table **??** but are explained in the section 4.1 of this document. In addition, a gantt chart is also presented where the time spent on that task in days can be consulted, as well as its start relative to the other tasks in figure 2.2.

Task	Start Date	End Date	Status	Aproximate hours employed	Hours used	February	March	April	May	June
Documents	**10.01.23**		Open	**40h about**	**10h**					
Technical Proposal	10.01.23	11.1.23	Finish	2h about	2h					
GDD	14.02.23	27.02.23	Open	8h about	8h					
Memory			Pause	30h about						
Basic Logic			Pause	**60h about**						
Base Game			Pause	40h about						
Actions Requesting			Pause	5h about						
Observation tools			Pause	5h about						
Bots			Pause	10h about						
Server			Pause	**50h about**						
Investigation			Pause	10h about						
Server Setup			Pause	40h about						
Database			Pause	**50h about**						
Investigation			Pause	10h about						
DB setup			Pause	40h about						
Unity			Pause	**60h about**						
Setup structure program			Pause	20h about						
Read GS			Pause	20h about						
Setup Units			Pause	10h about						
Setup Interface			Pause	10h about						
Art and design			Pause	**20h about**						
Sprites			Pause	10h about						
Animations			Pause	10h about						
Prepare presentation			Pause	**20h about**						
Fix bugs			Pause	5h about						
Prepare demostration			Pause	10h about						
Power Point			Pause	5h about						
Total			Pause	**300**						

Figure 2.1: Planning Table Start

2.2 Resource Evaluation

For this project we have only made use of free software tools such as:

- **Pycharm**: It is a Python IDE and was used to implement the base game logic and part of the server.
- **Anaconda**: It is a package manager that was used in conjunction with Pycharm.
- **VsCode**: It is the IDE used to implement most of the server.
- **Unity 2D**: It is the videogame engine used to implement the client part.
- **Trello**: It is a task manager that has been used to plan tasks and manage personal work.
- **Github**: It has been used to store and manage the code of the logical base of the game as well as the server.
- **Discord**: It has been used to communicate with my TFG tutor Raúl Montoliu.

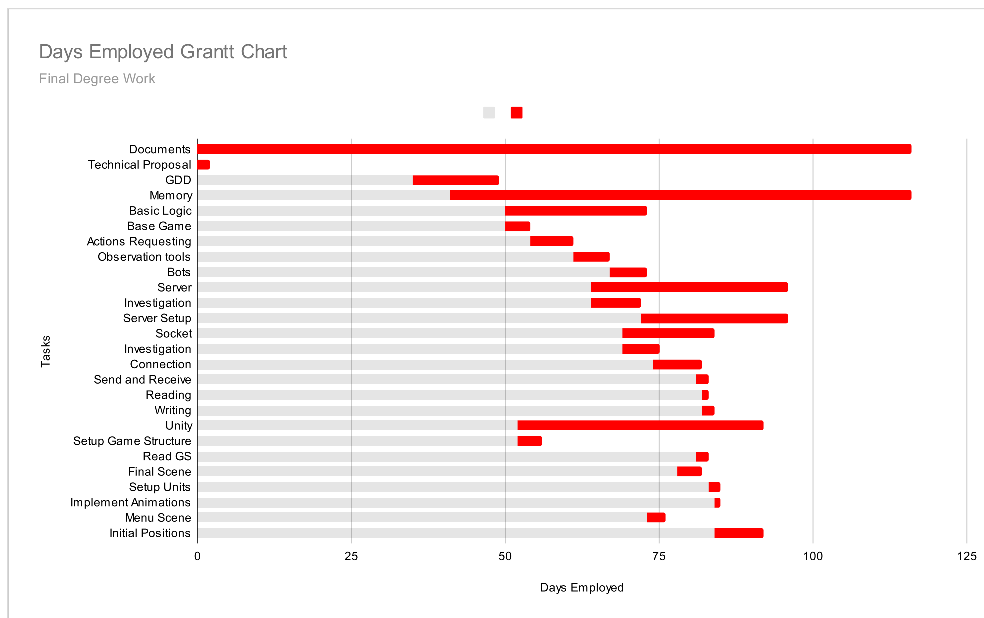


Figure 2.2: Gantt Chart

As for the hardware tools needed I have only needed a computer, I have made use of my personal laptop which has the following specifications:

- CPU: AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx
  - Maximum frequency: 2.1GHz
  - Cores: 4
  - Logical processors: 8
- RAM: 8,00 GB (6,94 GB usable)
- GPU: AMD Radeon(TM) Vega 8 Graphics (integrated)

CHAPTER 3

System Analysis and Design

Contents

3.1 Requirement Analysis **13**
3.2 System Design **17**
3.3 System Architecture **20**
3.4 System Analysis **21**
3.5 Interface Design **24**
3.6 Videogame Art **25**

This chapter presents the requirements analysis, design and architecture of the proposed work, as well as, where appropriate, its interface design. This chapter will present the analysis of the system requirements, both functional and non-functional, and finally their design. In addition, a more timid mention will be made for other sections with less weight, such as the interface design and some artistic sections.

3.1 Requirement Analysis

All those system requirements, both functional and non-functional will be presented superficially in the first two sections. After this, in the section 3.2 these functionalities will be explained in more detail, even providing UML diagrams [19] to facilitate their understanding.

3.1.1 Functional Requirements

This section will have little weight in this document due to the nature of the game. As it is a bot game, player participation is very low, however, all aspects related to the

functional requirements will be explained in great detail.

The proposed system should allow the user to choose which teams to engage as well as be able to allow the user to choose which types of troops to use and assign their starting positions.

To allow the user to choose which types of troops to use and assign their starting positions, these functional requirements are shown in the tables 3.1 and 3.2.

In addition, the proposed system should be able to execute the actions of the Judges when they play in human mode, i.e. control the units. The functional requirements of selecting the desired unit and assigning a destination to it are shown in the tables 3.3 and 3.4.

Table 3.1: Functional requirement «SETUP1. Game Teams»

Input:	A Pair of String
Output:	Game with teams selected
The user can select the teams that will play the game in the menu window. Here will be presented a series of checkboxes for each team where the user will select one for each of them and the system will send the selections in String form to the server to process them and create the pertinent game instance.	

Table 3.2: Functional requirement «SETUP2. Units Positioning»

Input:	JSON Object List
Output:	Units positioned as desired
The user will have an intermediate state between the menu and the game where he will be able to choose the type of unit he wants using buttons and then position that unit on the battlefield by right clicking the mouse. The only restrictions will be the maximum and minimum number of troops to position and the obligation that one of them (maximum and minimum) is a general.	

Table 3.3: Functional requirement «ACTION1. Unit Selection»

Input:	Mouse left click position (Vector2)
Output:	Unit selected attribute change
When a user is playing as a human, he can select units by clicking on them.	

Table 3.4: Functional requirement «ACTION2. Destination Setting»

Input:	Mouse left click position (Vector2)
Output:	Unit destination change
The user, after having selected a unit, can assign it a new destination by selecting a new position within the playing field with the left mouse click.	

3.1.2 Non-functional Requirements

The system has to be secure, multi-user, scalable, efficient, readable and well documented. All these non-functional requirements are explained in more detail in the tables 3.5, 3.6, 3.7, 3.8, 3.9 and 3.10.

Table 3.5: Non-Functional requirement «SECURITY1. Malicious Third-Party Person»

Requirement:	Protecting user's games
Although the security of this system is not critical because no sensitive data can be leaked and no important user data can be lost, it is important to put certain barriers in place to ensure that the user's gaming experience is as good as possible. These barriers are explained in the section 3.2.1	

Table 3.6: Non-Functional requirement «SECURITY2. Career Conditions»

Requirement:	Avoid server failures
It is very important to take this section into account. When a game request is made to the server, the server searches for an available port and, after a certain time, returns the information to the user so that he can connect. If during this time, another process identifies that same port as free, before the previous user has connected, there would be a connection error for the last user trying to connect to the socket and, in addition, the connecting user could be connecting to another user's game, with a different configuration. To avoid this, thread management techniques such as locks have been used. This technique is explained in the section 3.2.1	

Table 3.7: Non-Functional requirement «MULTIUSER. Concurrent Programming»

Requirement:	Allowing multiple simultaneous users to play the game
If the game were not running on the server, this section would not be necessary, but since this is not the case, it has been necessary to develop a concurrent programming technique to be able to technique to be able to run several instances of the game at the same time. This technique is explained in the section 3.2.2	

Table 3.8: Non-Functional requirement «DOCUMENTATION. Docstring»

Requirement:	Facilitating the use of internal methods by users
As those who want to program bots for the game will need, firstly, a good understanding of the inner workings of the game logic and, secondly, to know the methods offered by the game to calculate results, as well as the basic heuristics provided by the administrators. It is necessary that all this is well documented to facilitate understanding and search efficiency, this is achieved using Python Docstrings[1] that allow us to document the classes, modules and methods to subsequently automate the extraction of these in a single well-structured and organized pdf document. This technique is explained in more detail in section 3.2.3	

Table 3.9: Non-Functional requirement «READABILITY1. Clean Code»

Requirement:	Facilitate understanding of the code for users and administrators
In order to facilitate the understanding of the code, both for administrators and users, a design guideline extracted from the book Clean Code [12] has been followed. This design pattern is explained in more detail in section 3.2.4.	

Table 3.10: Non-Functional requirement «READABILITY2. PEP 8 Python»

Requirement:	Facilitate understanding of the code for users and administrators
Another design pattern that has been followed for the realization of the project has been to follow Python's PEP 8, this is a style guide that establishes conventions for the syntax when writing Python code. This style guide is explained in more detail in section 3.2.4.	

3.2 System Design

This section proceeds to explain in more detail some of the requirements presented in the previous sections. The section is divided into the different types: Security 3.2.1, Multi-user 3.2.2, Documentation 3.2.3 and Readability 3.2.4. In addition to a detailed explanation of each system, UML diagrams [19] are provided in some sections to support the text.

3.2.1 Security

In this project, security has been partially necessary to protect the user's games as well as the integrity of the server. Since users are not only authenticated at the moment of accessing the server but also at the moment of connecting to the socket that allows them to obtain the game states. This is called double-entry security system, since our system is composed of two marked phases that have to be protected independently. In addition, security measures have also been added to avoid certain errors within the system, as we have mentioned above, it has been necessary to use locks to prevent previously occupied ports from being assigned.

Malicious Third-Party Attack

As we have explained above, it is necessary to protect our server endpoint from creating instances of the game for anyone accessing it. We have achieved this by parsing the HTTP request headers received from the client. The server will only accept requests that have a specific header with a specific value, these values can only be obtained from the source code of one of the parties to maintain the confidentiality of these, in addition, to prevent anyone from analyzing the source code of the application we have applied an obfuscation technique in the client code.

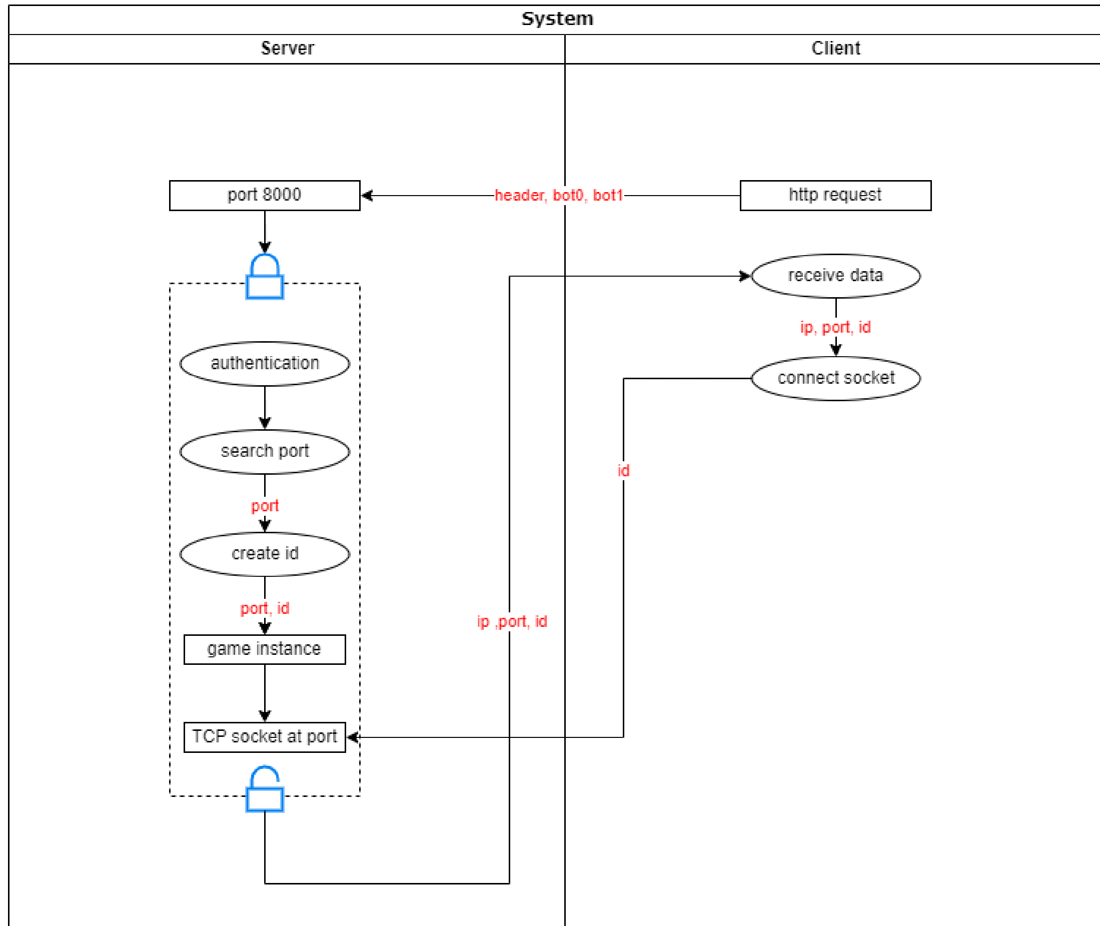
On the other hand it is also necessary to protect the socket in which the game instance is connected, since it remains listening waiting for the client connection and anyone could access it if we do not put a barrier. The solution to this problem is to provide the client with an identifier, this identifier is generated by a Python library called uuid that generates **Universal Unique Identifiers** [25], the client must forward this information to the socket, the socket will only accept the connection if the id matches the id of the client that is waiting. The socket will wait for 10 seconds for the connection, if there is no valid connection during this time, the game will terminate and release the socket. A summary of what happens in the system can be seen in Fig. 3.1.

Career Conditions

To implement the server we used the Python library called Flask [17], after some testing and research, we determined that Flask uses threads [1] to handle multiple requests at

[1] It is important to keep in mind that in Python threads work asynchronously and not in parallel.

Figure 3.1: Authentication and Career Conditions Security

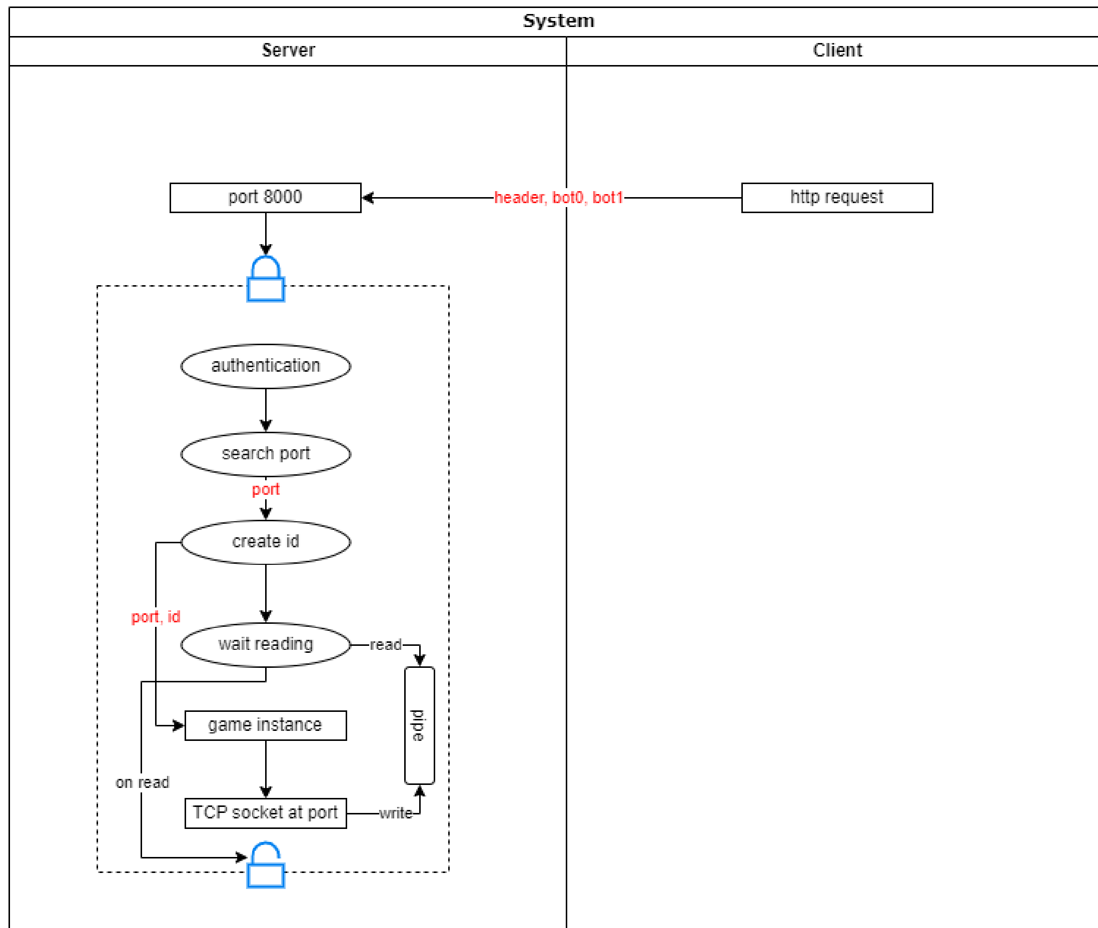


the same time (asynchronous). This gave rise to a possible error, and that is that if two requests are handled at the same time two clients can be assigned the same port with the errors that this would trigger. To prevent this from happening we have chosen to use locks (see Figure 3.1), so a thread is blocked until the previous thread has connected to the socket, thus avoiding this problem.

3.2.2 Multi-User

When the execution of the logic and rendering of a game falls entirely on the client side, making a multi-user application is trivial. But in the case of cloud gaming [26], where the game is executed on a remote server, this becomes a problem to be solved since the server has to create independent instances of the game for each user. The implemented solution goes through the **concurrent programming**. The server, at the time of assigning a free port, creates a new process that receives the port to which it has to connect and, once connected, sends, through a **pipe**, a message to the parent (server)

Figure 3.2: Use of Pipelines for parent-child Communication



so that it can release the lock and give way to the next request. In Figure 3.1 this step has been omitted so as not to overload the diagram, but in Figure 3.2 a modified diagram to see this communication between processes can be seen.

3.2.3 Documentation

The Python docstring [20] has been used for code documentation. The docstring is a Python documentation string used to describe the purpose and functionality of a module, class, function or method. Subsequently, you can make use of a Python library such as Pdoc [1]. It is a library that allows you to generate Python documentation in various formats, including PDF.

3.2.4 Readability

To improve the readability of the project, the Python style guide PEP8 [21] has been followed in conjunction with rules from the book Clean Code by Martin, Robert C. [12].

3.3 System Architecture

As the heaviest load of the game falls on the server, the client will only have to process a light load related to communication and graphics renderization.

3.3.1 Hardware Requirements

As for the hardware required for graphics rendering and communication with the server, a low-end cell phone will suffice.

Minimum requirements

- Operating system: Android 5.0 or higher, iOS 9.0 or higher.
- Processor: 1.4 GHz quad-core processor or higher.
- RAM memory: 2 GB or more.
- Storage: At least 100 MB of storage available for game installation.
- Graphics card: Support for OpenGL ES 2.0 or higher.

Recommended requirements

- Operating system: Android 7.0 or higher, iOS 12.0 or higher.
- Processor: Eight-core processor at 2.2 GHz or higher.
- RAM memory: 4 GB or more.
- Storage: At least 500 MB of available storage for game installation.
- Graphics card: Support for OpenGL ES 3.2 or higher.

3.3.2 Other Requirements

On this kind of game, as the vast majority of the computational load falls on the server, the important thing is to maintain a good quality of communication. That is why it is important that the client has a good internet connection, broadband and stable connection.

Internet Requirements

- Minimum download speed: 10Mbps
- Minimum upload speed: 5Mbps
- Connection stability: Very stable

3.4 System Analysis

3.4.1 Action Space

On each turn, the current player can provide an action for one of their units. An action consists of assign a unit a particular destination defined by a vector of 2 coordinates (x and y), and the movement normally takes several turns to be completed. If, in a turn, the player does not indicate an action for a particular unit, it continues the movement following the previous action performed on this unit.

An action has the following format: “ID δx δy" where:

- ID is the unique ID of the unit.
- δx is the x-coordinate where we want the unit to be directed to.
- δy is the y-coordinate where we want the unit to be directed to.

For instance, some actions that can be played are:

- Action(1, 100, 50): Unit with ID 1 will move to position $\mathbf{v} = (100, 50)$.
- Action(3, 800, 0): Unit with ID 3 will move to position $\mathbf{v} = (800, 0)$.
- Action(5, 0, 0): Unit with ID 5 will move to to position $\mathbf{v} = (0, 0)$.

Notice that if destination position is equal to unit position it will stop.

In each turn the agent will be able to execute only 1 action, this action is created by a class designed for this purpose, this class has a constructor that receives (ID, δx, δy) as an example would be: *return Action(0, 500, 500)*, this indicates that unit with ID 0 must go to position (500, 500).

3.4.2 State Representation

The system provides information about the player and opponents units. First, the game indicates the total number of units for each player's army. Then, the system provides the following information for each one of the player and opponent's units:

- ID: Unique ID of the unit.

- Location: (x, y) vector indicating the actual position of the unit on the battlefield.
- Direction: (x, y) normalized vector.
- Life: Amount of health points (See Table 1.2). The unit is dead when its life reaches 0.
- Type: Unit type for swordsmen ("SWORD"), spearmen ("SPEAR"), knight ("KNIGHT"), general ("GENERAL") and archers ("ARCHER").
- Moving: Indicates if the unit is moving (1) or not (0).
- Destination: (x, y) vector indicating where the unit is going to stop, only for friendly units (for opponent units, random destination is provided).

Therefore, the state has $1 + 9n + 7n$ elements, where n is the number of units for each player's army.

3.4.3 Game Complexity

The number of possible actions that can be played on each turn is huge, due to the large battlefield size (1000×500). One possibility to handle its complexity is to artificially reduce the places where the units can be moved. According to the size of the units, we suggest defining two grids, the first one of 7×4 ($1000/150 \approx 7$, $500/150 \approx 4$) and the second one of 14×8 ($1000/75 \approx 26$, $500/75 \approx 8$). Note that the units can always be moved to any place on the battlefield. The use of the grid is just for reducing the complexity of the game. It is suggested to be used in the first stages of the implementation of the agent or for beginners. The number of possible actions where $W \times H$ is the size of the battlefield and n is the number of units is determined by the formula:

$$n \times (W \times H) \tag{3.1}$$

On Table 3.11, you can see an example with 8 units for different battlefield sizes.

Table 3.11: Number of possible actions for 8 units depending on the battlefield size.

Battlefield size	# Actions
1000×500	4×10^6
14×8	8.96×10^2
7×4	2.24×10^2

3.4.4 Baseline Agents

The positioning of the units is up to the player, who must always choose the armies to be used and in which positions they will start. However, in order to speed up the work, a series of pre-established training courses are provided.

1. StayStatic (SS): This agent does not execute any action, it only keeps the troops in the same place during the whole time. This agent is mostly used to test with other agents and see how they behave against this behavior.

2. AlwaysForward (AF): This agent makes his units always advance towards the front, if they reach the end of the battlefield they turn around and go straight to the opposite side.

3. Random (RND): In each turn this agent selects a random unit that does not have a fixed destination and sets a random destination among all the possible ones. This is the ideal agent when you want to test a new algorithm because the different movement patterns it can offer are infinite.

4. One Step Looking Ahead (OSLA): The One Step Looking Ahead (OSLA) agent is designed to simulate future steps in a game, selecting actions that lead to optimal results. In order to achieve this behavior in complex games with a multitude of possibilities, it is crucial to employ advanced tools and techniques. One fundamental aspect involves evaluating and comparing different game states, which is accomplished through the use of heuristics. Although our implementation utilizes a simple heuristic that yields satisfactory results, more sophisticated heuristics could be developed to further enhance performance.

   The heuristic employed by OSLA is based on a straightforward approach of summing the health of allied troops while subtracting the health of enemy troops. The resulting value represents the overall state evaluation.

   To handle the enormous number of potential actions, which can reach up to 1.66×10^7, a reduction strategy was employed. By limiting the number of actions tested for each unit to just eight, the total number of possible actions was reduced to $n \times 8$, where n represents the number of units. These actions involve movement in the four main cardinal directions (north, south, east, and west) as well as their combinations (northeast, northwest, southeast, and southwest), with the distance being a variable factor.

   Furthermore, to simulate forward steps effectively, a modified loop was developed to supplement the main game loop, known as "step" (refer to Figures 4.2 and 4.4 for a visual representation). This modified loop, referred to as "pseudo step," operates independently of time dependencies. The original loop normally adjusts unit movement based on the time elapsed since the last frame (δ time). However, to maximize the simulation of frames within the limited decision-making time of 100 ms for the agents, the pseudo step loop omits such time normalization. Its purpose is to progress the state by a specified number of frames and provide the corresponding outcome.

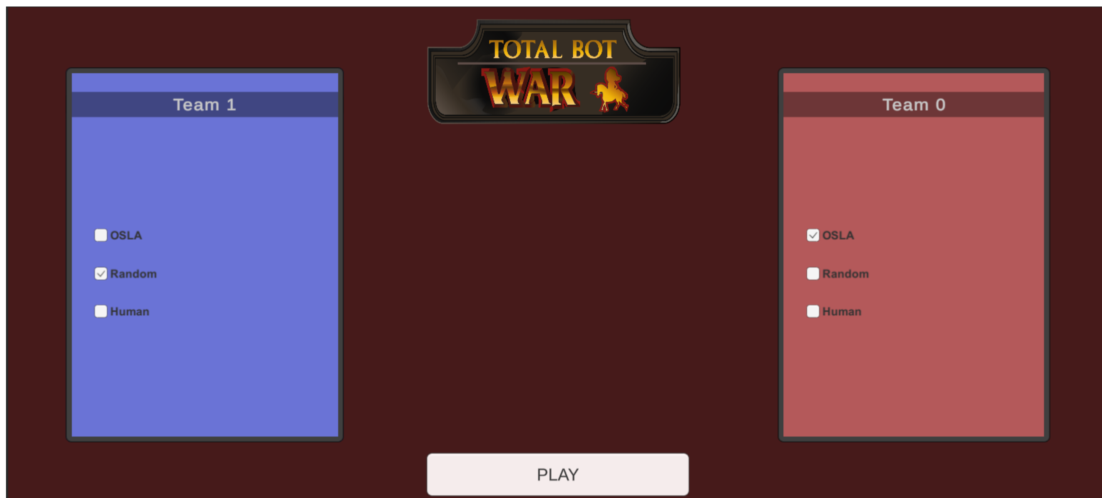


Figure 3.3: TotalBotWar Menu Interface

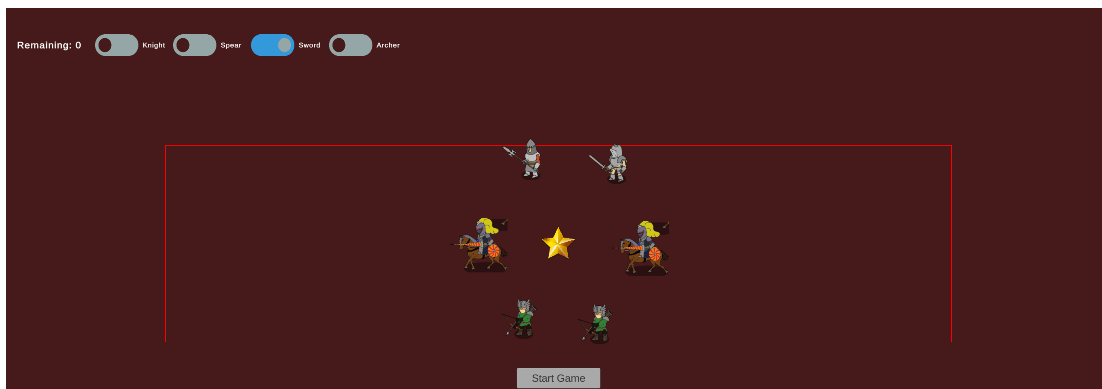


Figure 3.4: TotalBotWar Positioning State Interface

By employing the OSLA agent and implementing these techniques, the ability to project future game states accurately is greatly enhanced, enabling more informed and effective decision-making.

3.5 Interface Design

As this project has focused on the programmatic and engineering part, the interface design has been left in the background. For the game interface we have used basic assets from the Unity UI package and assets from the Final Degree Project of the former student Alejandro Estaben [4]. In the Figures 3.3, 3.4 and 3.5 interfaces in the 3 available stages of the game have been shown.

Figure 3.5: TotalBotWar Battle RND vs OSLA

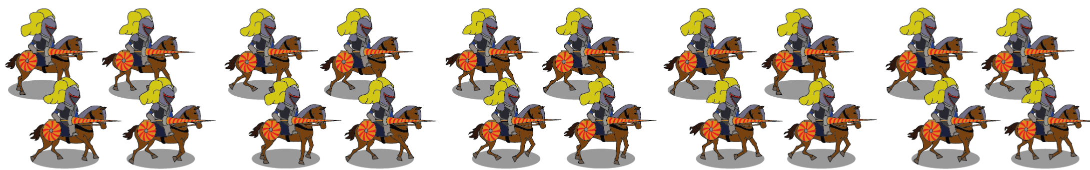


Figure 3.6: Knight Running Animation

Figure 3.7: Archers Attacking Animation

3.6 Videogame Art

As mentioned previously, we have made the decision to reuse the sprites created by César Díaz, a former student of the program, for Alejandro Estaben's TotalBotWar project. We acknowledge that this choice was influenced by time constraints and a lack of available resources.Images of some units can be seen at Figures 3.7, 3.6 and 3.8. And the images of the 3 available backgrounds at this moment can be seen at 3.9, 3.10 and 3.11.

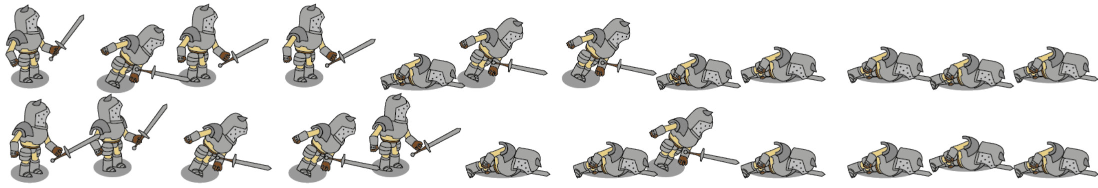


Figure 3.8: Swordsmen Death Animation

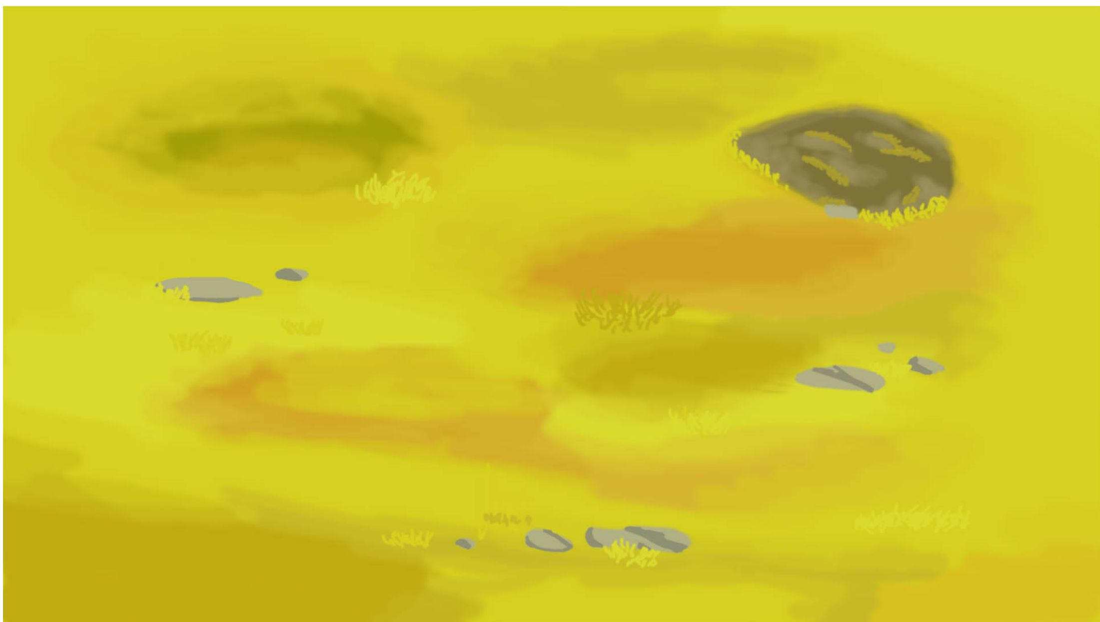


Figure 3.9: Background N.1

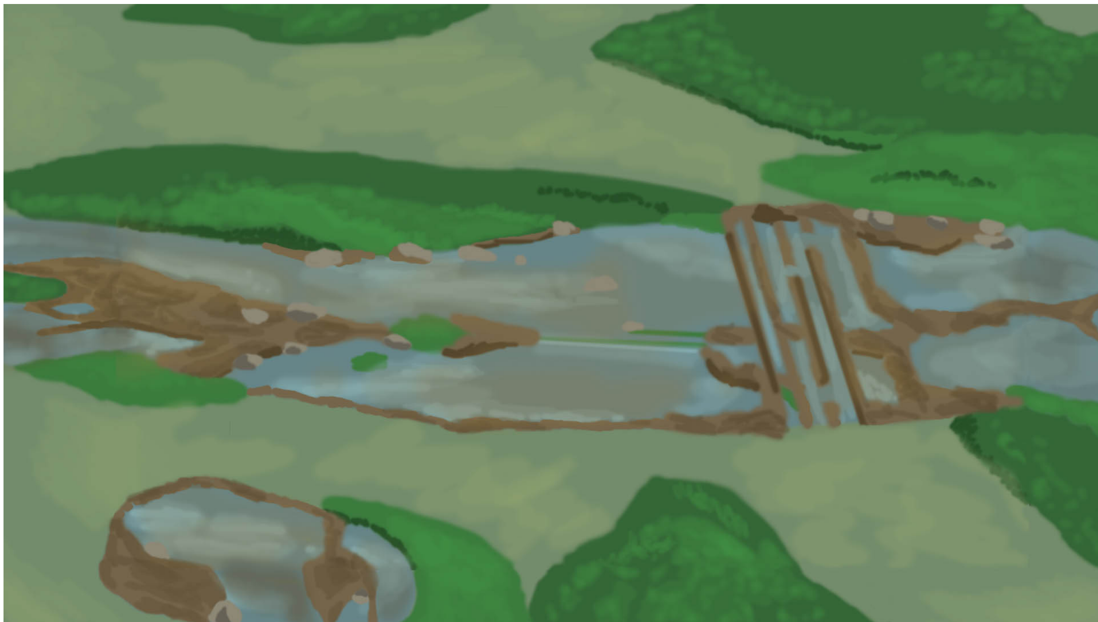


Figure 3.10: Background N.2

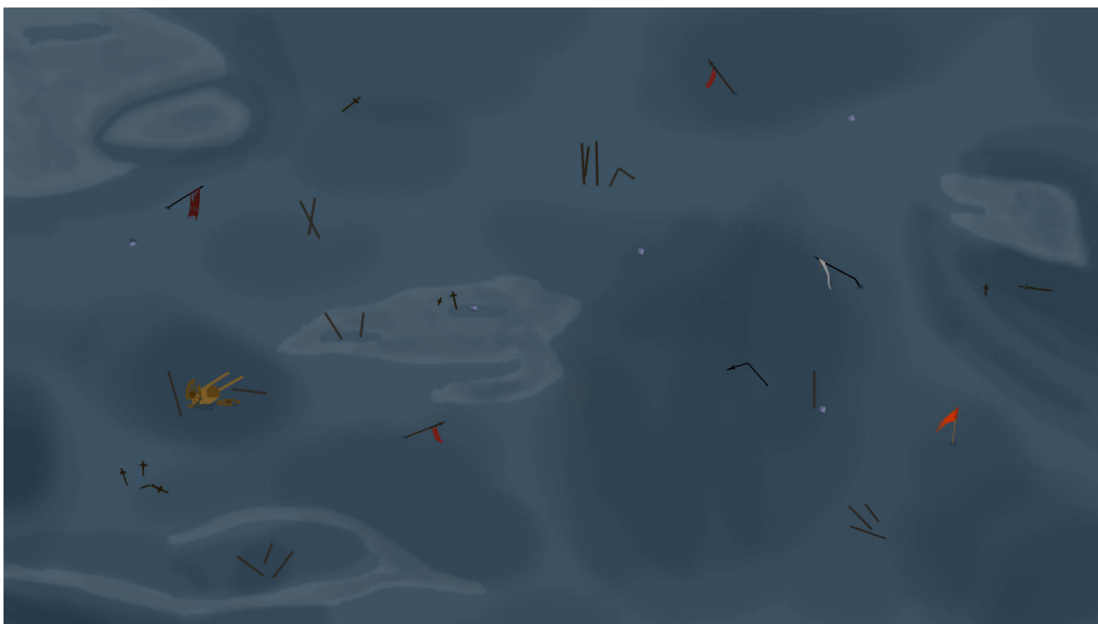


Figure 3.11: Background N.3

Chapter 4

Work Development and Results

Contents

4.1 Work Development **29**
4.2 Results **38**

Throughout the development of the project, several setbacks were encountered that required making decisions and adjusting certain aspects. Initially, a specific technology was planned for the project, but further research and unexpected issues revealed that it was not the most suitable choice. This necessitated a reevaluation of the approach and a decision to switch to a different technology. Additionally, unforeseen technical challenges arose during development, which required seeking alternative solutions and adjusting the approach to address them. Despite the difficulties faced, these setbacks fostered greater creativity and the discovery of more innovative solutions to achieve the project's goals. In the following section 4.1, the initial plans, encountered setbacks, and the chosen alternative will be discussed.

4.1 Work Development

The following is an outline of the work done in chronological order. As expected, during the course of the project, there have been problems to face and many decisions to make. We have had to make alternatives to the initial plan in order to move the work forward but, after much effort and time, we have achieved our goal.

4.1.1 Initial Conception

The first settled and realistic idea that settled in our heads was to build a Python server that would be in charge of creating instances of the game for each client. To these instances we would create an entry in a relational database locally where the game states would be stored. Likewise, the server would work as a REST API that by means of a player's identifier would access the area of the database corresponding to the player's game and extract the desired states, send them to the client and the client would render them on screen.

4.1.2 Game Core

The initial phase of the project involved programming the foundational logic of the game using Python. To visualize the progress of the project, we required a user-friendly graphical interface. Therefore, we opted for the pygame library [23], which is a Python library providing tools for utilizing 2D graphical resources and allowed us to preview the game.

To begin, we focused on implementing the essential classes that would serve as building blocks for the rest of the development. Among these, the Vector class played a crucial role. In this class, I partially implemented vector functionalities, such as vector normalization, calculating vector magnitude, determining the distance and direction between two points, scalar product calculation, angle measurement between vectors, and more...

Once the foundational classes were in place, we proceeded with the main classes, starting with the units. Initially, we created a generic class, Unit, which served as an abstract class for the rest of the unit types. It included general attributes like life, direction, position, ID, target, and so on, along with methods for movement, rotation, life reduction, attack boosts, attacking, and more. Subsequently, we implemented subclasses for each specific unit type, such as Archer, Sword, Spear, Knight, and General.

With the unit classes established, we could move on to implementing the main game loop. This loop was responsible for deploying troops and moving them in each iteration. The graphical representation of this loop is divided into parts shown in Figures: 4.2, 4.3, and 4.4.

In this phase it was necessary to start with the actions, the actions are objects of the Action class, this class is characterized by determining a unit and the location where it should go. We implemented the player class that has a method called *think* and returns an Action, so each iteration of the game will call this method of the players and will pass them a copy of the game state called Observation, the players must read this Observation and generate an Action for the game to execute it. So we implement a variation of the basic Player, the RandomPlayer (see RND), this player generates a random Action,

chooses a random unit and assigns a random position on the map. Facing two of these bots we could start implementing the rest of the mechanics.

The main mechanics to be implemented were collisions and charges (see Figure 4.3), i.e., every time two units collided the angle of impact was taken into account as well as the types of troops, since the game follows basic rock-paper-scissors (see Figure 1.1) style rules, horses beat swords, swords beat spears and spears beat horses.

After this, a bot still basic but more complex than the previous ones, the OSLA (see OSLA), was implemented. This bot is characterized by evaluating a certain number of actions and choosing the one that gives the best result in the short term. For this it was necessary to implement a basic heuristic that was based on the number of accumulated life of each team and in addition, we had to implement simulation mechanics that allowed us to simulate a given number of frames to be able to predict in a certain way the future of an action, since a single frame did not give any information. With all this we managed to implement a bot called OSLAPlayer that seems to make logical decisions and always beats RandomPlayer.

With all this working we could finish the first stage of the project and move on to the implementation of the database and the server.

4.1.3 Server

As the main idea was to implement a REST API to return information from a database, a server was built with Flask that had several endpoints to extract information according to the client's needs. One of the endpoints was designed to return all the information from the database while another was designed to return only the last stored state, the information was always returned in JSON format.

4.1.4 Database

The next step was to implement a database to store the game states in it. We decided to use a relational database using SQLite with a single table, so we did not need an ORM[1]. Each row corresponded to a state and the columns of this table were:

- ids: "0 1 2 3 4..."
- teams: "0 0 0 0..."
- healths: "200 100 0 65 23.4..."
- types: "SWORD ARCHER ARCHER KNIGHT GENERAL..."

[1] An ORM (Object-Relational Mapping) is a programming technique that allows developers to interact with a relational database using object-oriented paradigms. It acts as a bridge between the application code and the database, abstracting the underlying database operations.

- states: "IDLE MOVING ATTACKING..."
- positions: "(0,0) (254,865)..."
- directions: "(0,1) (0.45, 0.55)..."

In each column was the value of all units in string format separated by spaces, so when mapping it on an object we can easily separate the values of the strings.

4.1.5 Unity

On the client side we needed to make an http request from unity to the server and, after receiving the response, format the text strings separating them by spaces and finally map these results to objects. Once this was done, we could start with prefabs instances with the information of these objects, for example:

- Item 1
  - **id** = 0
  - **team** = 0
  - **health** = 100
  - **type** = "knight"
  - **state** = "idle"
  - **position** = (0) 0
  - **direction** = (1) 0

With this object we can instantiate a knight prefab for device 0 with id 0, at position (0,0) with address (1, 0) and idle state.

It should be noted that all these procedures are done by using coroutines asynchronously to ensure that the result is available before executing another method that will use that information. For example, network access is done asynchronously and a listener is set up and executed when the http request is finished.

Problems and Alternatives

Everything was flowing correctly until the database was implemented and a **performance test** was done, the operation was mechanically correct, everything worked, the game ran, the database was updated and the client could render the game but, the performance was not good enough. The game initially (before including the database) was running at 90-100 fps while now, with the database built in, it was running at 15-20 fps. I made an exhaustive analysis of the problem and determined that the problem was to insert the states 1 by 1, since the insertions consumed approximately 50 milliseconds so that in 1 second only gave time to perform approximately 20. Different options were

considered such as inserting packages of several states, but, it was not viable since the reading should be in real time since the user should be able to interact with the game since the mechanics was implemented to play in human mode. Another option was to change the database to a non-relational database, based on documents that stored json. After some research we determined that this could be a viable option since this type of databases show 2 advantages over the classic SQL databases:

1. **Scheme flexibility**: the documents or json of a nosql database do not need to maintain a fixed structure, this comes in handy in this case because there are data that we only need once at the beginning of the game as for example, the type of the troop.
2. **Velocity**: NoSQL databases are generally faster than SQL databases for bulk reads and writes due to their horizontal distribution and the elimination of the need to join tables.

In addition my tutor, Raul Montoliu, proposed to use a Google service called Firebase Database [6] for this purpose. Firebase offers a real-time database in the cloud, which allows developers to store and synchronize real-time data between clients in real time. So I set out to implement this alternative.

4.1.6 Firebase Database

First it was necessary to investigate the possibilities offered by this Google service. After determining how to register the application in the platform and adding the firebase sdk in the application we proceeded to find a way to authenticate our users so that no one else could access the database without permission. The alternative to this was to use an authentication method built into firebase called AppCheck, but after some research we realized that to this day (May 22, 2023) AppCheck is not compatible with Unity. So the alternative to this was to use our server as an intermediary, instead of letting the users interact with the database we would first make the clients have to identify themselves to our server using a custom header as explained in the 3.2.1 section. This is quite secure since the server is the only one that can directly access the database thanks to the credential system used by Firebase. In addition, only the clients that have access to the custom header could access our server and could only execute the operations that it offers on the database.

Problems and Alternatives

After some time of work dealing with the functionality and tools offered by Firebase, as well as its limitations, I finished the implementation of the database in Firebase and everything worked correctly, but the same problem returned, although even more serious. Now the game was running at 2 fps. This was because now, as the database was in the cloud, the insertion was even slower. We had to look for an alternative that did not involve the use of databases because it had already been demonstrated that they were useless for streaming gameplay.

The new idea was to use **sockets**, the server will create child processes that will connect to available sockets and listen for a client connection. When the client connects the game will start in the child process that will send the game states over the socket. This ended up being the correct option, we will now proceed to explain its implementation in detail.

4.1.7 Socket Implementation

For the implementation of this methodology it was necessary to adapt both the client side and the server side, as well as the main program of the game.

Server

First it was necessary to implement a search algorithm for free ports. This was achieved by a simple loop that goes through all available ports and tries to connect a socket to each one, when it is able to connect a socket to a port this means that port is free so it has done its job.

Flask [17] works asynchronously to handle multiple requests in parallel. This can lead to race conditions when assigning available ports to clients, to the point where the same port is assigned to different clients, leading to obvious problems. The procedure to solve this problem has already been explained in the section 3.2.1 on career conditions security.

The next challenge was to generate child processes that connect to a given socket, for this purpose we made use of the subprocess library [5], it is a tool to work with external processes from a Python program. It allows to start a process in the operating system and control it from Python. With this tool, the creation of the child processes as well as the communication through pipes has been carried out. The pipes have been used to transmit a message from the child to the parent, when the child has been able to connect to the socket it sends a success message to the parent so that it unblocks and moves on to the next request, otherwise the child sends an error message and cuts the execution of the game.

Once we had defined the mechanism for creating processes and assigning them a socket, we needed to define the communication protocol that the socket would use to communicate with the client. Two options were presented:

- **UDP:** This is a connectionless transport protocol, i.e. messages are sent as datagrams over the socket without knowing if there is anyone listening on the other side. If the message is not intercepted by anyone, it is lost. Despite its unreliability, it is very useful in cases where transfer speed is more important than data integrity.

- **TCP:** This is a transport protocol with connection, that is, two processes have to be connected at both ends of the socket sending and receiving data consecutively, the receipt of a message is the sending signal for the next one. This protocol, although slower than UDP, ensures data integrity (reception order) and data arrival.

First of all, we thought that the UDP protocol would be the most appropriate because in cloud gaming [26], speed prevails over integrity, i.e. information losses are not critical, so certain losses could be tolerated as long as the transfer speed improves. In fact, an implementation of this method was made, but soon after we decided to change it to TCP protocol, since we forgot to take into account the other side of the coin. Data goes from the server to the client in the form of states (frames) but it also goes from the client to the server in the form of inputs to be processed by the server. In this case the integrity of the data is important because we can not allow the system to be unresponsive, since one of the most important things in video games is to give the user the feeling that he has total control over what happens, also the order of processing of inputs is also important because it is what differentiates between selection of troops and destinations, with the first click units are selected and with the second one destinations are assigned.

After determining the communication protocol it was necessary to determine a system to protect the socket from unwanted clients, since although the endpoint was protected by a custom header, the socket had no protection and, in the time interval from when a client makes a request to the server and receives the response, until it connects, another process could connect to the socket waiting to start the game. The mechanism used for this purpose is the one explained in the second paragraph of the section 3.2.1 on third-party attacks.

A real game state sent by server in JSON format can be seen at A.0.2.

Figure 4.1: Main Loop Abstract

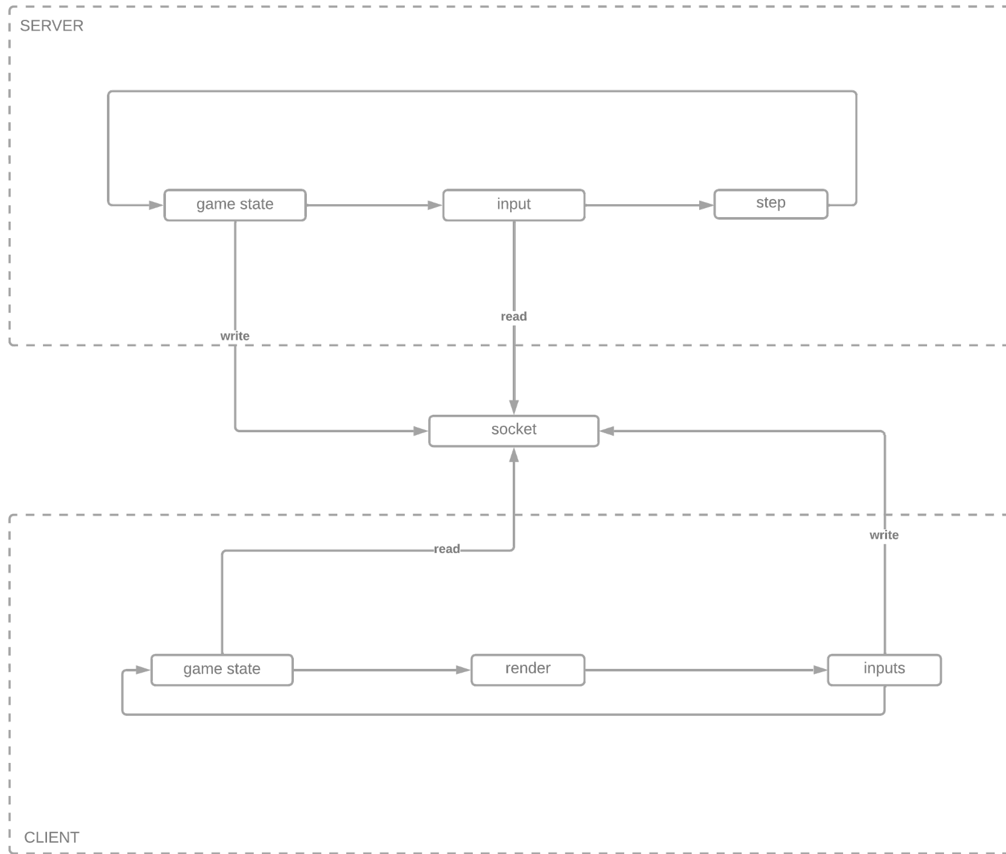


Figure 4.2: Game Step 1/2

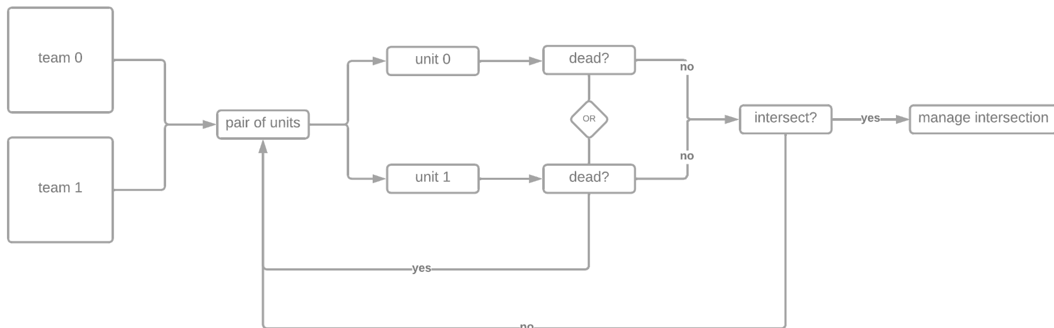


Figure 4.3: Manage Intersection

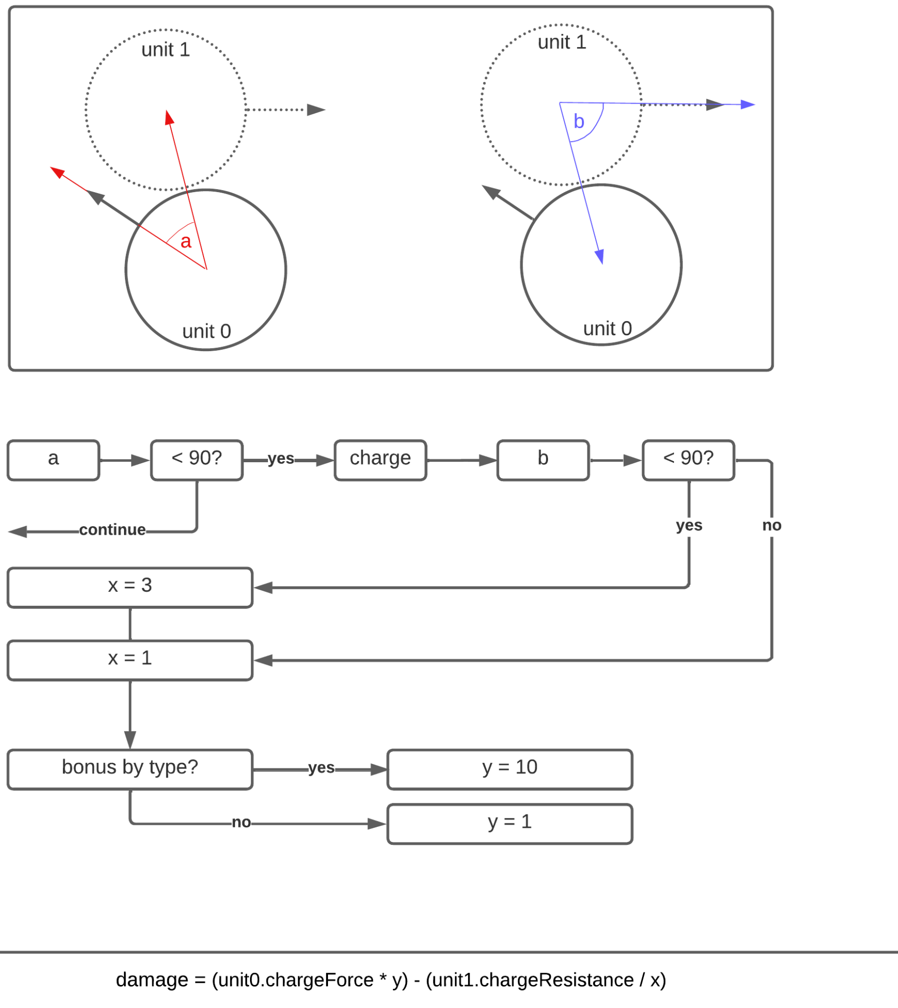


Figure 4.4: Game Step 2/2

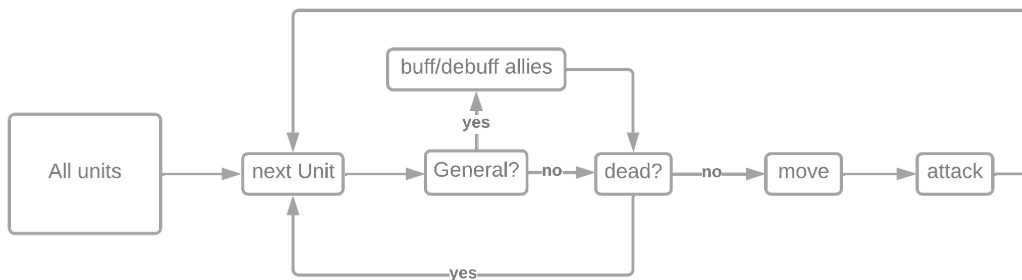
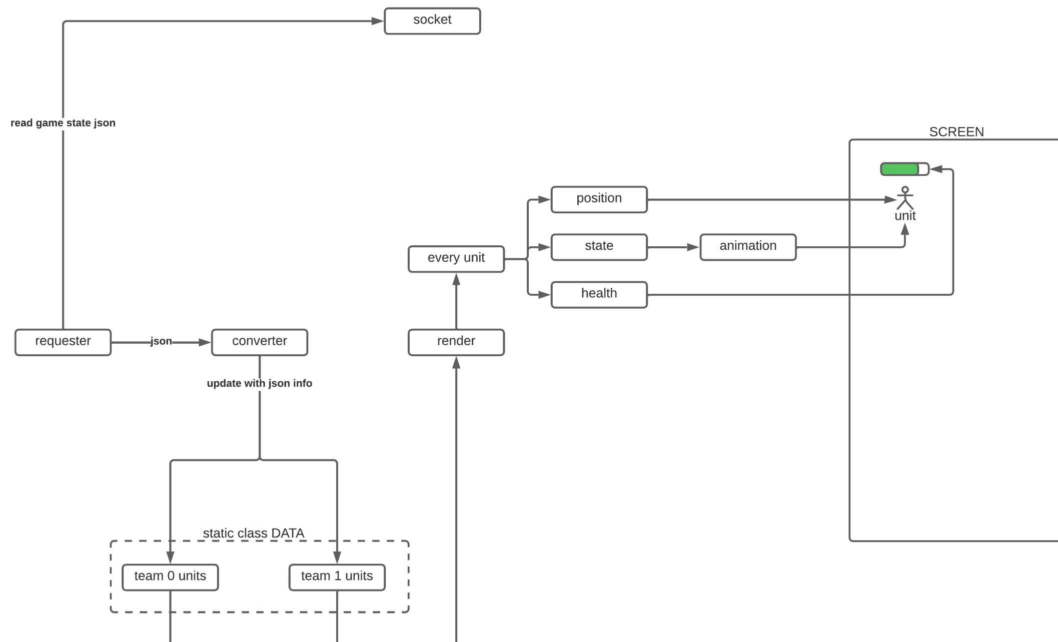


Figure 4.5: Update of Data and Rendering Loop



4.2 Results

In general we are very happy with the results, we believe that we have achieved all the proposed objectives and we are satisfied with the result, although it is true that there is still a lot of development to be done to reach the final goal. I think I have left a good base to start working with, it is going to be a very good starting point for other people to continue with the development of this and we think that this project has a lot of real potential. We think this is going to be a very good platform to develop agents and learn in the process. In the Figures 4.6 and 4.7 a 2 different game can be seen. Also at Figures 4.8 and 4.9 the server and the game can be seen working at the same time. In case you have not read it in section 1, I will link you again to the available videos. I encourage you to watch them.

- Video: TotalBotWar - Human vs OSLA
- Video: TotalBotWar - Random vs OSLA
- Video: TotalBotWar - DEBUG

4.2.1 Goals Achieved

This section focuses on the objectives achieved within the scope of this undergraduate project. Throughout the development of this work, a set of objectives was established

Figure 4.6: Random vs OSLA Ingame Background N.2

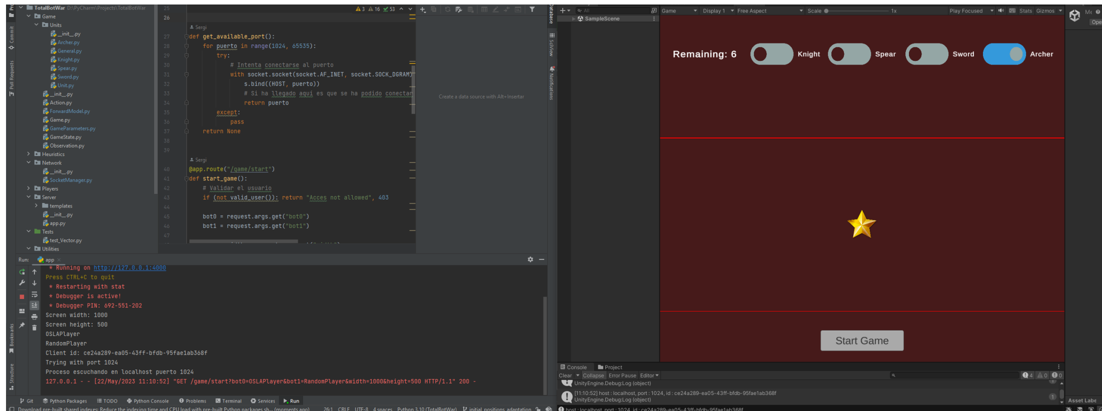
Figure 4.7: Random vs OSLA Ingame Background N.3

(see Section 1.2) to address and overcome the challenges at hand. In this section, the initial objectives will be presented and a detailed discussion on how they have been achieved will be provided, offering a clear overview of the progress made and the results obtained.

While the results will be described in this section, it is important to note that detailed technical aspects will not be included. To avoid unnecessary repetition and maintain a concise approach, specific technical details will be addressed in the corresponding sections of the document where methods, techniques, and analyses are thoroughly explained. References to these sections will be made to guide readers to where they can find more detailed information on the technical aspects.

By adopting this approach, we aim to provide a clear overall understanding of the accomplishments in relation to the set objectives while also encouraging interested readers

Figure 4.8: Server and Game Working While Positioning



to consult the relevant sections for a more in-depth and technical comprehension of the specific processes and outcomes.

- ◇ G1: As observed throughout the document, the game has been made accessible for mobile devices. Through the implementation of the server (see Subsec. 4.1.3) and the TCP socket (see Subsec. 4.1.7), the game is now capable of running on mobile devices with minimal computational burden. This achievement maintains the feasibility of all other goals. However, it is important to note that the Cloud Gaming derived structure implemented in the project has imposed constraints on fully meeting this objective without compromising others.

- ◇ G2: An architecture has been implemented that leverages the game's central server to directly load algorithms, eliminating the need for users to download updates. The server acts as a REST API, notifying the client about available bots. The client interface has been designed with flexibility in mind, displaying the bots on the screen and providing checkboxes for selection.

  Additionally, an endpoint has been automated to retrieve the names of available bots by formatting the file names in the "Players" folder. This automation eliminates the need for additional time investment. For security reasons, a personal review of user-uploaded bots has been implemented, following a more austere methodology. Once the code review is completed, the process of uploading a bot is as simple as dragging and dropping a file (or folder) onto the server.

  In summary, this approach ensures efficiency and security in managing user bots within the game. It removes the requirement for users to download updates and provides a seamless process for selecting and uploading bots.

- ◇ G3: As of the time of writing this document, the most advanced implemented bot is OSLA (see OSLA). Despite being a relatively simple agent, it consistently

delivers impressive results and poses a considerable challenge. However, the true significance lies in the opportunity to develop more sophisticated bots. The game's core has been entirely developed in Python, making it the recommended language for programming these agents. Python stands out as the language with the greatest potential for artificial intelligence, thanks to the extensive range of high-quality libraries it provides for this purpose. Some of the notable libraries for AI in Python include:

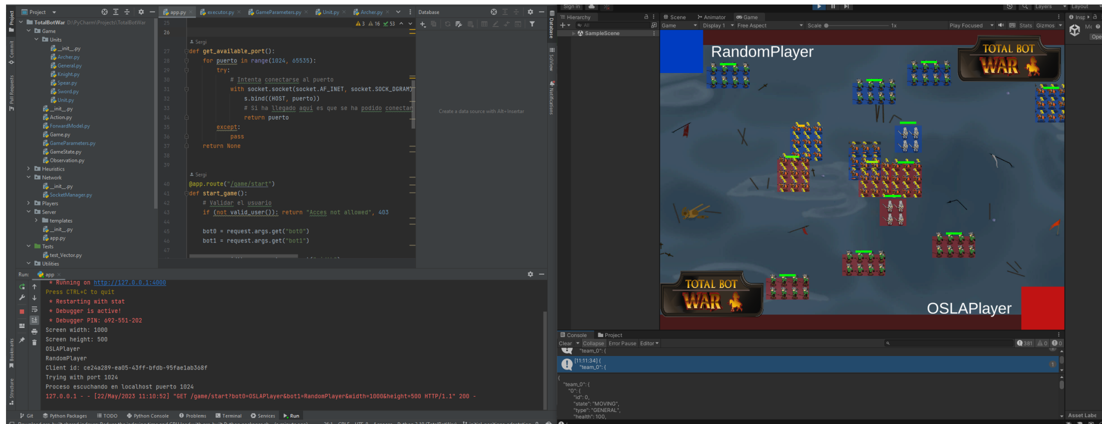
- **TensorFlow**
- **PyTorch**
- **OpenCV**
- **Gensim**
- **SciPy**

These libraries offer a wide range of functionalities, such as machine learning, deep learning, natural language processing, computer vision, and scientific computing, empowering developers to explore and implement advanced AI techniques in their projects.

◇ G4: As we have explained in section G2 above, the implementation of the server and the flexibility that this offers us has also allowed us, collaterally, to meet this objective. Since, when the game is started, the client asks the server for a list of all the available bots, so it can choose among all of them. Once chosen, it communicates with the server and sends 2 strings with the identifier *bot0* and *bot1*, so the game can be configured so that these are the bots that play the game. In both figures 3.1 and 3.2 you can see how the client sends the relevant information to the server to configure the game parameters.

◇ G5: In any professional endeavor, the pursuit of learning should be a central objective, and this project is no exception. I take great pride in the accomplishments we have achieved thus far. The challenges and obstacles we encountered along the way provided valuable opportunities for growth, problem-solving, and personal development. Through this journey, my passion for programming and system design has deepened significantly. Moreover, I have acquired a wealth of knowledge spanning diverse subjects, including Servers (4.1.3), Networking, Python (4.1.2), C#, Unity (4.1.5), GitHub, Concurrent Programming, Cybersecurity (3.2.1), REST APIs, Databases (4.1.4), SQL, JSON, Firebase (4.1.6), and more. Embracing these multifaceted areas of study has not only broadened my skillset, but also enhanced my understanding and expertise in various aspects of technology and software development.

Figure 4.9: Server and Game Working While Battle



4.2.2 Comparison between Planning and Final Work Accomplished

When we started this project, we knew it was very ambitious. It has been a task that involves many areas of development and has presented numerous challenges that have allowed us to grow but have also consumed a lot of time. Despite being generous in our project planning (see Fig. 2.1), the time spent has exceeded the estimated time (see Fig. 4.10).

4.2.3 Applications of the Work Performed

The project at date May 22, 2023, is fully playable and functional. However, it is currently not playable on mobile phones, and we need to host the server and adapt the client to access the server's IP on the new host before putting it on the app store. Additionally, we would like to improve the graphics and add some extra functionalities, as well as incorporate new bots before making it public. Once these tasks are completed, the game can be applied for real algorithmic AI learning purposes, as well as for testing and research in this vast field, as mentioned in Section 1.2.

Tasks	Start Date	Days Employed	End Date	Status	Aproximate hours employed	Hours used	February	March	April	May
Documents	**10.01.23**	**115**	**05.05.23**	**Open**	**40h about**	**40h**				
Technical Proposal	10.01.23	1	11.1.23	Finish	2h about	2h				
GDD	14.02.23	13	27.02.23	Finish	8h about	8h				
Memory	20.02.23	74	05.05.23	Open	30h about	30h				
Basic Logic	**01.03.23**	**22**	**23.03.23**	**Finish**	**60h about**	**77h**				
Base Game	01.03.23	3	04.03.23	Finish	40h about	30h				
Actions Requesting	05.03.23	6	11.03.23	Finish	5h about	7h				
Observation tools	12.03.23	5	17.03.23	Finish	5h about	20h				
Bots	18.03.23	5	23.03.23	Finish	10h about	20h				
Server	**15.03.23**	**31**	**15.04.23**	**Finish**	**50h about**	**40h**				
Investigation	15.03.23	7	22.03.23	Finish	10h about	10h				
Server Setup	23.03.23	23	15.04.23	Finish	40h about	30h				
Socket	**20.03.23**	**14**	**03.04.23**	**Finish**	**55h about**	**38h**				
Investigation	20.03.23	5	25.03.23	Finish	20h about	10h				
Connection	25.03.23	7	01.04.23	Finish	5h about	3h				
Send and Receive	01.04.23	1	02.04.23	Finish	20h about	15h				
Reading	02.04.23	0	02.04.23	Finish	5h about	5h				
Writing	02.04.23	1	03.04.23	Finish	5h about	5h				
Unity	**03.03.23**	**39**	**11.04.23**	**Finish**	**85h about**	**128h**				
Setup Game Structure	03.03.23	3	06.03.23	Finish	20h about	40h				
Read GS	01.04.23	1	02.04.23	Finish	20h about	15h				
Final Scene	29.03.23	3	01.04.23	Finish	5h about	10h				
Setup Units	03.04.23	1	04.04.23	Finish	10h about	20h				
Implement Animations	04.04.23	0	04.04.23	Finish	5h about	3h				
Menu Scene	24.03.23	2	26.03.23	Finish	5h about	10h				
Initial Positions	04.04.23	7	11.04.23	Finish	20h about	30h				
Prepare presentation	**30.12.99**	0	**30.12.99**	**Pause**	**20h about**	**0h**				
Fix bugs		0		Pause	5h about	0h				
Prepare demostration		0		Pause	10h about	0h				
Power Point		0		Pause	5h about	0h				
Total		0			310	323h				

Figure 4.10: Planning Table Finish

Chapter 5

Conclusions and Future Work

Contents

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

After many months of development, the first functional version of the project has been obtained and with it many lessons and skills acquired during the development. In the following sections I will comment my personal opinion about what this project has given me and can give me as well as what I have in mind to implement in the future, either by myself or by other students who adopt this project.

5.1 Conclusions

With regard to the project, all the objectives established at the outset have been met. This project has helped me learn many new things, including how to manage large projects and how to organize myself. I have learned that it is very important to have a very clear idea of the project and that it is worth spending more time designing the system before starting to work, although it seems that you lose time it ends up being the opposite. I have also learned that it is worth spending time to have a clean and structured code, this ends up saving a lot of time when the project is growing and leaves the door open to grow the project in the future.

Personally, I believe that what this project has given me will give me a lot of freedom from now on to carry out new projects on my own. I feel able to find solutions to many of the problems that I face every day and I am eager to put all this knowledge into

practice. I am very happy because I feel capable of developing very interesting projects as well as contributing to other people's projects. Thanks to this I see myself able to perform much better in the professional world from now on.

This knowledge not only helps me to solve problems but also to have more and better ideas. I am now much more aware of what can be achieved with the right tools and how to overcome adversity.

5.2 Future work

We have many ideas to implement in the future:

- Graphics: We want to create our own custom sprites tailored for the game. Due to time constraints, we had to use graphics created by César Díaz for the previous version of TotalBotWar. Since they were not designed specifically for this implementation, they don't fully meet our needs. Therefore, we would like to create our own sprites from scratch, taking into account our specific requirements.

  For example, since the sprite sizes are determined at runtime due to the flexible nature of the game, all animations should have the same proportions. Currently, if one animation has a 1:1 ratio while another has a 2:1 ratio, resizing the sprite to fit the first animation causes distortion in the second one. Although additional checks could be implemented to adjust the proportions depending on the active animation, this would require considerable effort for a minor issue. The correct solution would be to create new artwork that addresses this problem appropriately.

- Multiplayer: Currently, the game can only be played locally in human vs bot or bot vs bot modes. It is also possible to play human vs human, but it would require local play where the same user controls both teams. In the future, we aim to develop a multiplayer option that would allow two humans to play against each other from different geographical locations.

- Turing Test: We want to develop our own Turing test, building upon the previous objective. If the game allows playing with other people, we could create a game mode where players are unaware whether they are playing against another person or a human-controlled AI player. At the end of the game, we would present a form to the player asking them to indicate whether they believe they played against a human or an AI-controlled agent. This information would be stored in a database to draw conclusions in the future.

- Other Platforms: We would like to develop the game for other platforms, primarily for computers, but we are also considering the possibility of developing it for consoles. This is to reach as many people as possible and fulfill one of our main objectives (see G1).

- ◇ More Agents: We would like to have many more bots available in this video game. While we will develop some additional bots ourselves, the main idea is to encourage game users to create their own bots and upload them to our server.
- ◇ Info Saving: With the aim of conducting studies and training other AI algorithms, we would like to store information from game sessions. The idea is for each played game to save information in a database, such as the game ID, game states, playtime, player 1, player 2, winner, etc. This way, we can leverage this information to draw conclusions and train other AI algorithms that can analyze these states and learn from them.

Personally, I plan to continue with the development of this project because I enjoy it and I am interested in seeing it grow. I also feel a sense of responsibility to leave it in the best possible state so that future individuals working on it can do so in the most efficient manner, with well-structured and clean code.

I also want to create a couple more AI algorithms and implement data saving to make the most of the game sessions played by users when the game is published.

The next thing I want to do is polish the code and the functionality of the project, as well as improve the interface and aesthetics, in order to publish it as soon as possible. This way, users can contribute to the creation of bots at the earliest opportunity.

Bibliography

[1] Andrew Gallant. pdoc – api documentation generator for python. `https://pdoc.dev/`, 2021. [Online; accessed 3-May-2023].

[2] I. Bravi, D. Perez-Liebana, S. M. Lucas, and J. Liu. Rinascimento: Optimising statistical forward planning agents for playing splendor. In *2019 IEEE Conference on Games (CoG)*, pages 1–8, 2019.

[3] M. ertick, D. Churchill, K. Kim, M. ertick, and R. Kelly. Starcraft ai competitions, bots, and tournament manager software. *IEEE Transactions on Games*, 11(3):227–237, 2019.

[4] Alejandro Estaben. TotalBotWar: A New Pseudo Real-time Multi-action Game Challenge and Competition for AI. *arXiv preprint arXiv:2009.08696v1 [cs.AI]*, September 2020. `https://arxiv.org/abs/2009.08696v1`.

[5] Python Software Foundation. Python subprocess library documentation. `https://docs.python.org/3/library/subprocess.html`, 2021. Accessed on 2023-05-03.

[6] Google. Firebase. `https://firebase.google.com/?hl=es&authuser=0`, 2021. [Online; accessed 4-May-2023].

[7] Google. Google Bard. Google, n.d. `https://bard.google.com/?hl=en`.

[8] R. Ishii, S. Ito, R. Thawonmas, and T. Harada. A fighting game ai using highlight cues for generation of entertaining gameplay. In *1st IEEE Conference on Games (CoG'19)*, 2019.

[9] N. Justesen, P. Bontrager, J. Togelius, and S. Risi. Deep learning for video game playing. *IEEE Transactions on Games*, 12(1):1–20, 2017.

[10] N. Justesen, L. M. Uth, C. Jakobsen, P. D. Moore, J. Togelius, and S. Risi. Blood bowl: A new board game challenge and competition for ai. In *2019 IEEE Conference on Games (CoG)*, 2019.

[11] J. Kowalski and R. Miernik. Legends of code and magic. `https://jakubkowalski.tech/Projects/LOCM/`, 2019. [Online; accessed 8-April-2020].

[12] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, Upper Saddle River, NJ, 2008.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[14] S. Ontan, N. A. Barriga, C. R. Silva, R. O. Moraes, and L. H. S. Lelis. The first microrts artificial intelligence competition. *AI Magazine*, 39(1):75–83, 2018.

[15] OpenAI. ChatGPT: Language Models for Task-Oriented Conversations. OpenAI Blog, November 2021. https://openai.com/blog/chatgpt/.

[16] OpenAI. DALL · E: Creating Images from Text. OpenAI Blog, January 2021. https://openai.com/blog/dall-e/.

[17] Pallets Projects. Flask documentation. https://flask.palletsprojects.com/en/2.2.x/, 2023. Accessed: May 22, 2023.

[18] D. Perez-Liebana, S. M. Lucas, R. D. Gaina, J. Togelius, A. Khalifa, and J. Liu. *General Video Game Artificial Intelligence*. Morgan & Claypool Publishers, 2019.

[19] Dan Pilone and Neil Pitman. *UML 2.0 in a Nutshell*. O'Reilly Media, Inc., 2005.

[20] Python Software Foundation. PEP 257 – Docstring conventions. https://peps.python.org/pep-0257/, 2001. [Online; accessed 3-May-2023].

[21] Python Software Foundation. PEP 8 – Style Guide for Python Code. Website, 2001. Accessed April 21, 2023.

[22] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362:1140–1144, 12 2018.

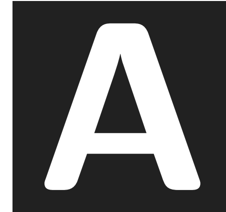
[23] Pygame Team. About pygame. https://www.pygame.org/wiki/about, 2021. Accessed on 2023-05-03.

[24] J. Walton-Rivers, P. R. Williams, and R. Bartle. The 2018 hanabi competition. In *2019 IEEE Conference on Games (CoG)*, pages 1–8, 2019.

[25] Wikipedia. Universally unique identifier. https://en.wikipedia.org/wiki/Universally_unique_identifier. Last accessed: 4 de mayo de 2023.

[26] Wikipedia. Cloud gaming — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Cloud_gaming, 2021. [Online; accessed 3-May-2023].

Appendix



Source code

As we have used GitHub for code management and version control, we believe it would be better to provide you with a direct link to access the repositories related to this project. This way, we can avoid lengthy code explanations and provide complete access to all the code that you may want to review.

A.0.1 GitHub

- Server: https://github.com/SergiFuster/TotalBotWar
- Unity: https://github.com/SergiFuster/TotalBotWar-Unity

A.0.2 Game State in JSON

What we will include directly in this section is the game state in JSON format. Obtaining this information would require downloading the appropriate versions of the project and running them on your computer, which can be cumbersome. We believe it is more convenient to provide the JSON object here directly, so you can observe an example of what the Unity application reads in each frame and how it updates the necessary information.

```
{
    "team_0": {
        "0": {
            "id": 0,
            "state": "MOVING",
            "type": "GENERAL",
            "health": 100,
            "position": {
                "x": 501.01,
                "y": 181.9
            },
            "orientation": {
                "x": 1.0,
                "y": -0.02
            },
            "width": 50.0,
            "height": 50.0,
            "archerTarget": false
        },
        "1": {
            "id": 1,
            "state": "ATTACKING",
            "type": "KNIGHT",
            "health": 67.0,
            "position": {
                "x": 175.59,
                "y": 228.44
            },
            "orientation": {
                "x": 0.36,
                "y": 0.93
            },
            "width": 60.0,
            "height": 60.0,
            "archerTarget": false
        },
        "2": {
            "id": 2,
            "state": "ATTACKING",
            "type": "KNIGHT",
            "health": 176.0,
            "position": {
                "x": 776.09,
                "y": 268.28
            },
            "orientation": {
```

```
47                 "x": -0.59,
48                 "y": 0.81
49             },
50             "width": 60.0,
51             "height": 60.0,
52             "archerTarget": false
53         },
54         "3": {
55             "id": 3,
56             "state": "MOVING",
57             "type": "SPEAR",
58             "health": 250,
59             "position": {
60                 "x": 380.79,
61                 "y": 258.72
62             },
63             "orientation": {
64                 "x": 0.0,
65                 "y": 1.0
66             },
67             "width": 70.0,
68             "height": 35.0,
69             "archerTarget": false
70         },
71         "4": {
72             "id": 4,
73             "state": "MOVING",
74             "type": "SPEAR",
75             "health": 250,
76             "position": {
77                 "x": 608.85,
78                 "y": 244.82
79             },
80             "orientation": {
81                 "x": -1.0,
82                 "y": -0.01
83             },
84             "width": 70.0,
85             "height": 35.0,
86             "archerTarget": false
87         },
88         "5": {
89             "id": 5,
90             "state": "ATTACKING",
91             "type": "SWORD",
92             "health": 195.0,
93             "position": {
94                 "x": 519.54,
95                 "y": 257.2
96             },
97             "orientation": {
98                 "x": 0.79,
99                 "y": 0.62
100            },
```

```
101             "width": 24.96,
102             "height": 12.64,
103             "archerTarget": false
104         },
105         "6": {
106             "id": 6,
107             "state": "MOVING",
108             "type": "ARCHER",
109             "health": 100,
110             "position": {
111                 "x": 501.95,
112                 "y": 144.17
113             },
114             "orientation": {
115                 "x": 0.0,
116                 "y": 1.0
117             },
118             "width": 80.0,
119             "height": 40.0,
120             "archerTarget": false
121         }
122     },
123     "team_1": {
124         "0": {
125             "id": 0,
126             "state": "MOVING",
127             "type": "GENERAL",
128             "health": 100,
129             "position": {
130                 "x": 576.56,
131                 "y": 335.41
132             },
133             "orientation": {
134                 "x": 0.89,
135                 "y": -0.46
136             },
137             "width": 50.0,
138             "height": 50.0,
139             "archerTarget": false
140         },
141         "1": {
142             "id": 1,
143             "state": "ATTACKING",
144             "type": "KNIGHT",
145             "health": 81.0,
146             "position": {
147                 "x": 735.72,
148                 "y": 323.94
149             },
150             "orientation": {
151                 "x": 0.59,
152                 "y": -0.81
153             },
154             "width": 60.0,
```

```
            "height": 60.0,
            "archerTarget": false
        },
        "2": {
            "id": 2,
            "state": "ATTACKING",
            "type": "KNIGHT",
            "health": 67.0,
            "position": {
                "x": 198.37,
                "y": 287.77
            },
            "orientation": {
                "x": -0.36,
                "y": -0.93
            },
            "width": 60.0,
            "height": 60.0,
            "archerTarget": false
        },
        "3": {
            "id": 3,
            "state": "MOVING",
            "type": "SPEAR",
            "health": 250,
            "position": {
                "x": 671.93,
                "y": 311.8
            },
            "orientation": {
                "x": 0.9,
                "y": 0.44
            },
            "width": 70.0,
            "height": 35.0,
            "archerTarget": false
        },
        "4": {
            "id": 4,
            "state": "MOVING",
            "type": "SPEAR",
            "health": 250,
            "position": {
                "x": 332.17,
                "y": 312.75
            },
            "orientation": {
                "x": -0.98,
                "y": 0.19
            },
            "width": 70.0,
            "height": 35.0,
            "archerTarget": false
        },
```

```
209         "5": {
210             "id": 5,
211             "state": "ATTACKING",
212             "type": "SWORD",
213             "health": 200.0,
214             "position": {
215                 "x": 537.46,
216                 "y": 271.27
217             },
218             "orientation": {
219                 "x": -0.79,
220                 "y": -0.62
221             },
222             "width": 45.0,
223             "height": 22.77,
224             "archerTarget": false
225         },
226         "6": {
227             "id": 6,
228             "state": "MOVING",
229             "type": "ARCHER",
230             "health": 100,
231             "position": {
232                 "x": 577.36,
233                 "y": 415.56
234             },
235             "orientation": {
236                 "x": 1.0,
237                 "y": -0.09
238             },
239             "width": 80.0,
240             "height": 40.0,
241             "archerTarget": false
242         }
243     }
244 }
```