FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Visualizador 3D de Ambiente de Simulação para Múltiplos Robôs

Nuno Barreira Pereira

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Pedro Gomes da Costa

Second Supervisor: José Lima

July 31, 2024

Resumo

Esta dissertação apresenta todo o desenvolvimento por trás da programação de um algoritmo que permite a um utilizador executar simulações com uma frota de robôs móveis autónomos em grande escala. Um outro sistema, desenvolvido pelo CRIIS, já se encontra disponível, mas, embora permita traçar rotas, localizar robôs móveis e processar todo o seu mapeamento, não suporta a simulação de forma simultânea de muitos dispositivos. Assim, os objetivos principais serão conectar a uma outra plataforma que suporte um grande número de robôs o Gestor de Frota, de forma a controlar uma frota extensa e, ao mesmo tempo, comprovar que o novo visualizador é escalável, permitindo simulação com a mais diversa quantidade de equipamentos. Desta forma, todas as trajetórias dos robôs e o seu comportamento durante a simulação terão de ser programados e visionados. O framework ROS (Robotic Operating System) permite que se leiam grafos a partir de ficheiros YAML. Em seguida, sendo cada aresta e vértice lidos, todos os caminhos possíveis são delineados. Posteriormente, são fornecidos os robôs que se pretende simular, juntamente com as arestas que se pretende que cada um percorra.

Alguns dos resultados foram bastante eficazes e positivos, visto que agora é possível simular e visualizar qualquer número de robôs com sucesso, mas, ao mesmo tempo, um dos objetivos não foi concluído.

Abstract

This thesis presents the entire development behind the programming of an algorithm that allows a user to carry out simulations with a large fleet of autonomous mobile robots. Another system, developed by CRIIS, is already available, but although it allows routes to be plotted, mobile robots to be located and all their maps to be processed, it does not support the simultaneous simulation of many devices. Therefore, the main objectives will be to connect to another platform that supports a large number of robots, in order to control a large fleet and, at the same time, to prove that the new visualiser supports a scalable fleet, allowing simulation with the most varied number of devices. All the robot trajectories and their behaviour during the simulation had to be programmed and visualised. The ROS (Robotic Operating System) framework allows graphs to be read from YAML files. Once each edge and vertex has been read, all possible paths are outlined. Then the robots to be simulated are provided, along with the edges along which each is to travel.

Some of the results have been quite effective and positive, as it is now possible to successfully simulate and visualise any number of robots, but at the same time one of the objectives has not been achieved.

Acknowledgements

I'd like to thank my supervisors, Pedro Gomes da Costa and José Lima, for all their amazing help and support during the development of my dissertation. They were always available and ready to help, and I'm so grateful for all their guidance and encouragement! I'd also like to thank Diogo Matos, who really helped me out so much during my time at iilab.

I'd also like to thank all my family, especially my parents, who were absolute superstars! They were so patient and understanding, always there to help me solve my problems and cheer me up when things seemed to be going wrong.

I'd also like to thank my friends in Gaia who were there for me every day, giving me a boost when I needed it most and encouraging me to keep pushing until I reached the final goal.

I'd like to thank all my friends from university who have been with me every step of the way, helping me through every stage of my higher education, and they were there for me again during the dissertation.

I'd also like to thank my best friend, Maria Donga, who was an absolute crucial person when it came to giving me advice and cheering me up in the most difficult moments. I'd like to thank her a lot for her patience and concern.

I want to thank all the friends from Coimbra that I made on the last two years for all the support that gave me everyday.

I would like to thank all the staff from iilab for the great treatment that they gave me when I was there working on my thesis.

Finally, I'd like to say a very special thank you to my grandmothers. I'm dedicating this dissertation to them because I know that my success will bring them both the greatest happiness.

Nuno Pereira

"Don't compare yourself with anyone in this world.
If you do so, you are insulting yourself."

Bill Gates

Contents

List of Figures

List of Tables

Abreviatures and Symbols

API	Application Programming Interface
AMR	Autonomous Mobile Robot
IDE	Integrated Development Environment
IP	Internet Protocol
LCM	lightweight Communications and Marshalling
MORSE	Modular Open Robots Simulation Engine
ROS	Robot Operating System
TCP	Transmission Control Protocol
TEA*	Time Enhanced A*
UDP	User Datagram Protocol
YAML	Yet Another Markup Language
YARP	Yet Another Robot Platform

Chapter 1

Introduction

This chapter will first present the background and motivations for this dissertation. Next, it will detail the biggest challenges of the research. Finally, it will provide a concise overview of the document's structure.

1.1 Context

The rapid progress of automation technology has prompted companies to seek innovative solutions to meet increasing production demands. In the industrial sector, there is a growing interest in utilizing a fleet of multi-robots to efficiently execute numerous tasks within a minimal time frame. As a result, companies are increasingly investing in simulators to plan the potential utilization of big fleets of mobile robots. This approach allows companies to optimize their operational and financial performance by enhancing efficiency.

However, it's important to recognize that while there are benefits to this approach, there are also some negative points about it. For instance, real-world scenarios can subject a robot to situations that a virtual environment would not, and simulated robots only replicate what they have been programmed to without accounting for certain exceptions or unforeseen events. Given the importance of ensuring safety in industrial settings, it is critical to integrate a visualization system in conjunction with the robot fleet to enhance monitoring capabilities. This system enables the programming of robot paths, provides supervision to prevent deadlocks or collisions, offers visibility into the equipment's locations, and facilitates robot mapping.

1.2 Challenges

There are two main objectives: firstly, to develop a program that allows to visualise a large number of mobile robots simultaneously in the new visualiser and secondly, to ensure that the fleet manager developed by CRIIS is fully integrated with the new visualizer. The existing multi-robot system, implemented using the ROS (Robot Operating System) framework with C++ as the primary programming language, currently supports a fleet of around ten devices, a relatively small

number for industrial applications, so it is important to increase it in order to be possible to control an industrial number of AGV's. It is also very important that you have a Fleet Manager that is able to create collision free paths for each of the robots, which makes the second point extremely important.

1.3 Dissertation Structure

This document consists of an introductory chapter, this one, and five other chapters.

In the next chapter, 2, all the background research for the dissertation will be presented. It discusses what Mobile Robots are and the foundations of Mobile Robot Systems and Mobile Robot Simulators. Regarding the latter, there is an overview of the most commonly used simulators in robotics, highlighting their pros and cons, and ending with a table that presents all of them more intuitively. Also in 2, the importance of good Middleware for robotics simulation is discussed, introducing the most frequently used ones. Finally, the chapter addresses the importance of good trajectory planning, distinguishing some similar concepts that are often confused.

In Chapter 3, the implementation of the algorithm is described. There is a brief review of how everything was implemented and planned, including the trajectory planning for the robots.

The Chapter 4 presents all the features of the Visualizer used. It shows how it was implemented, highlighting the libraries used for its development. It also discusses each of the objects in the scenario individually, explaining their roles in the simulation.

Chapter 5 presents all the tests conducted and the results obtained. Then, it provides an analysis, explaining the purpose of each test.

In 6, a conclusion is drawn from the results, stating whether the goals of the dissertation were met or not. It also explains how the developed project can be improved in future work.

Chapter 2

State of Art

This chapter will cover all the previous research that has been carried out to deepen the subject of the project and explain some of the decisions that have been made.

A brief introduction to mobile robotics will be given, mentioning its importance in the industrial sector. Next, still related to the same topic, robotic simulators will be discussed, highlighting the most popular ones today and making a comparison between them. The most widely used middlewares in the world of robotics will also be mentioned, as well as the reason why the ROS framework was used. Finally, there will be a short presentation of what Path Planning is based on and some definitions of terms often misused or confused.

2.1 Mobile Robots

The evolution and innovation in the world of robotics have been particularly emphasized by society, highlighting the progress in mobile robotics. Previously, industrial robots and mechanical arms were favored. Nowadays, due to production demands, the focus has shifted to machines capable of moving and acting in their environment autonomously, known as Autonomous Mobile Robots (AMRs). By investing in this market, companies can reduce operational costs [38] and make processes more efficient and intelligent.

Mobile robots have unique aspects and competencies. Using sensors, they can perceive their surroundings [21], allowing them to map the entire space and plot their own routes to move autonomously and safely. To interact, they also have actuators capable of performing various tasks, such as loading and unloading materials. Moreover, AMRs are very flexible, as they can be adjusted to perform prioritized work at precise moments.

The integration of mobile robots into various industries signifies a paradigm shift towards intelligent, adaptable, and efficient automation. This transforming technology not only meets the demands of modern production [3] but also represents a strategic investment for companies looking to enhance their competitiveness in an ever-evolving market.

Besides the utilization of Mobile Robots in industry, the future passes by using them in daily life [19]. It means that the robots must be enabled to interact with humans not putting their safety

in danger. [36] states that humans bring new challenges in terms of motion planning and control. In this way, a human-aware motion planner must be created taking into account the social acceptance of it and providing legible paths [36].

Autonomous Mobile Robots (AMRs) are designed to operate continuously, 24/7, but they are not immune to failures [3]. A significant cause of these failures is the robots' interaction with varying environmental conditions. Therefore, regular maintenance is essential to ensure the robots remain functional and efficient. Many AMR manufacturers recommend and provide guidelines for different levels of maintenance, helping users to keep their equipment in optimal working condition, like [31].

2.1.1 Multi-Robot Systems (MRS)

Multi-Robot Systems is the name given to the groups that are formed by two or more robots that navigate and share the same workspace.

In robotics research, various terms such as Multi-Robot Systems, Multi-Agent Systems, Robotic Swarms [26], and Sensor Networks [26] are commonly used interchangeably to describe groups of robots operating together in a coordinated fashion within the same environment [20]. Although these terms generally refer to the concept of multiple robots working collaboratively, each one has distinct characteristics, as stated in [33]:

- **Multi-Robot Systems**: Represent all the systems that are formed by two or more Robots.
- **Mutlti-Agent System**: This term is not exclusive to robotics and generally refers to a system consisting of multiple intelligent agents that can interact with each other. That's the reason why some other areas like biology, psychology and economics also do some research in this area.
- **Swarms**: This is the term used for the coordination of multiple robots. In this context, the collective value is significantly more highlighted than the individual, as the group demonstrates greater effectiveness in accomplishing tasks. However, certain aspects must be considered and met for swarms to be efficient: robustness, scalability, and flexibility
- **Sensor Networks**: This term refers to a collection of mobile sensors that communicate with one another. Thus, any of the aforementioned groups could essentially function as a sensor network.

2.2 Mobile Robots Simulators

A mobile robot simulator is a computer program that makes it possible to create an environment in which the robot can move around indoors or in the open without the need to use the robot itself, saving physical and financial resources [4]. Robot simulators emulate the behavior of robots in a virtual world where they can interact with other robots and/or rigid bodies. Simulators have

become essential tools for industry. They make it possible to test concepts and algorithms [37], and design and plan trajectories.

As previously stated, one of the key advantages of this approach is that it offers an economically beneficial solution [38]. Conducting virtual testing prior to implementing it on a physical machine [1] enables us to ascertain the absence of any potential risk factors that could inflict damage upon our robots. Conversely, as previously stated, the robot can be redesigned and rebuilt at any stage, with each component undergoing individual testing. The most renowned and widely used in business environments are Gazebo, Stage, and MORSE.

For simulations to yield realistic outcomes, it is crucial to accurately model sensors such as cameras, lidars, infrared, and ultrasonic devices. These sensors play a vital role in detecting and interacting with the robot's environment. Precise sensor models in simulators are essential for the effective development and testing of robots' perception, navigation, and control algorithms [38].

These platforms are generally compatible with ROS. In [37], a project was developed to simulate a mobile robot in Gazebo, the simulator mentioned above, in a ROS environment. As can be seen in [37], ROS is an open-sourced robot operating system. In this same article, it's possible to note that due to research into various frameworks, numerous robotic software systems have emerged both in industry and in educational institutes. Derived from all these different development software for robots, ROS was created, which is a collection of many of these software systems. Still in [37] it is stated that the use of this system is promoted by its flexibility since it supports various programming languages, its stability, and its simplicity in creating robust robot behaviour on different platforms. This means that, now, there is no more effective and credible operating system than ROS.

2.2.1 Choosing the right 3D Simulator

Choosing the right 3D simulator for a project is a crucial decision that can significantly impact its success. Whether the user are developing simulations for robotics, autonomous vehicles, virtual environments, or any other application requiring 3D visualization and interaction, several factors need careful consideration [2]. The following aspects are the ones that are seen as the most relevant when choosing a Simulator:

- **Community** - Integrating a simulator into an established framework necessitates selecting a tool with a robust and active user base. Tools with a thriving community not only ensure reliability but also provide extensive support through various Q&A platforms. This vibrant user community facilitates easier learning and usage, making these tools more accessible for developers and users.

- **Integration Capabilty** - The simulator should be cross-platform, which must be compatible with various operating systems, including Windows, macOS, and Linux. This ensures that users can work with the tool regardless of their preferred or available system. Additionally,

the simulator needs to be portable, allowing for easy download and deployment without requiring significant modifications. This portability ensures seamless usability across different environments and facilitates straightforward integration into diverse workflows.

- **Being Open-source** - It is important to choose an open-source simulator, meaning it should be freely accessible to everyone without any licensing restrictions. Open-source simulators offer several advantages. They often have active communities that contribute to continuous the development and improvement of the applications, ensuring that the software stays updated.

- **Human Modeling Capability** - Ensuring the human safety in interactions with robots is one of the biggest concerns in the field of robotics. Consequently, it is crucial for simulators to replicate human behavior, given its inherent unpredictability. Such simulations are essential for a comprehensive understanding of potential risks, thereby facilitating the development of robust safety environments. By modeling human actions, these simulators enable the design of robotic systems that can operate safely and seamlessly alongside humans, accommodating the dynamic and often unforeseen nature of human behavior. This capability is crucial for advancing the field and enhancing the integration of robotic systems in environments shared with humans.

2.2.2 Gazebo

Gazebo was created in 2002 to help develop interaction environments between mobile robots [25]. This simulator is precisely designed to replicate dynamic scenarios that a mobile robot might face.

Gazebo provides a comprehensive platform for the rapid development and testing of multi-robot systems in innovative and engaging manners. It is a powerful, scalable, and user-friendly tool with the potential to broaden the scope of robotics research to a larger audience. Consequently, Gazebo is being explored for integration into undergraduate education [15].

In the world of robotics simulation, there are some key aspects that should not be overlooked. Gazebo is no exception and fulfils them, making them advantages of using this platform.

Positive Aspects

- It must be able to show solutions to collision and contact problems. In this way, the simulator has multiple physics engines.

- Gazebo maintains a simple API for addition of different objects like actuators, sensors, new robots and arbitrary objects.

- Gazebo has an extensive library of robot sensors such as cameras, lasers, sonars, GPS, as well as noise models that can be parameterised as the user wishes.

- It allows users to interact with the simulation via programming interfaces, such as the Robotic Operating System (ROS).
- Gazebo has a graphical interface that allows the manipulation of a 3D world.
- It can be simulated both indoors and outdoors.

Negative Aspects

- Due to its ability to display an entire 3D environment, a computer with a much higher level of performance is required to support this simulator.
- The simulated sensors tend to be too perfect for those available to us on a physical level.

As it only allows to simulate in three dimensions, to use Gazebo its needed a computer with high performance in order to allow the user to simulate a big fleet of robots. On the following figure 2.1, it is possible to verify this simulator's feature.

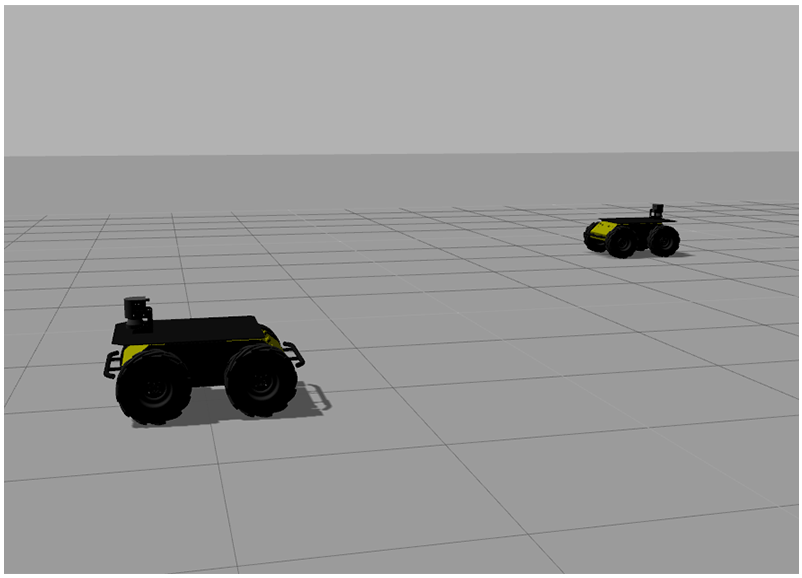


Figure 2.1: Gazebo Simulator's Environment

2.2.3 Stage/Player

Stage is a platform that allows the user to simulate a population of mobile robots in a two-dimensional bitmapped environment. Instead of prioritising being very reliable, this simulator prioritises being simpler and more efficient than others. On this platform, two distinct components must be differentiated, although they interact with each other: Player and Stage [16]. On one hand, Player provides an interface for the robot's sensors and actuators over a network [18]. Using the TCP protocol for communication, it allows the client to read from the sensors and issue commands to the actuators. On the other hand, Stage provides the robots and sensors themselves, with these

devices being accessed through Player. There are two original purposes of this simulator [10]. The first one is to enable a fast development of the drivers that are going to drive, eventually, the real robots. Second, is to allow the users to do some robot experiments without using real hardware and environments. Therefore, it's possible to list some of the advantages of using this platform:

Positive Aspects

- The fact that simulations are carried out in 2D means that users benefit in terms of efficiency. It also makes it a suitable choice for systems with more modest computational requirements.
- Stage is known for its simplicity. As such, it is very accessible to beginners.
- Stage can be easily integrated with other ROS libraries, allowing for easy integration into projects that use this framework.

Negative Aspects

- The simulator is quite limited, as it only allows simulations in 2D environments.
- It only enables the user to simulate in indoor environments [10].
- Compared to other simulators, Stage has a much less extensive library, which ends up limiting the variety of robots and sensors available for simulation.

It is possible to conclude that Stage was created with the specific purpose of facilitating research in multi-robot systems. In scenarios where programming and making experiments involve numerous robots, the advantages of a fast development are significantly amplified [10]. Stage allows for conducting experiments with large numbers of robots, which would be extremely costly to purchase. Various elements of Stage's design contribute to its effectiveness and suitability for multi-robot systems. The following figure 2.2 shows the environment of this simulator presenting a simulation with mobile robots.

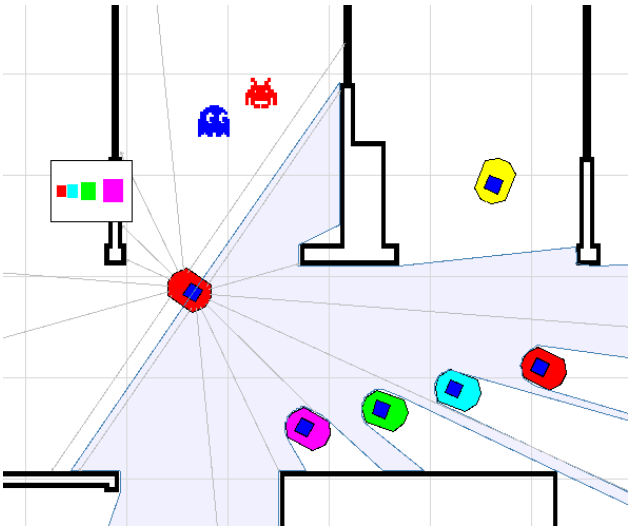


Figure 2.2: Player/Stage simulator environment

2.2.4 MORSE

Modular open robots simulation engine (MORSE) is a simulator that can be used in a wide variety of contexts to test interaction between various robots. It was developed on Blender Game Engine and Bullet physics engine [8], [25]. It allows the user to achieve both high and low levels of complexity, which makes it a very complete simulator. This platform belongs to the latest breed of robot simulators, the most flexible, versatile and reusable [7]. In figure 2.3 it's shown how the MORSE environment is with an example of a simulation that was made on this platform.

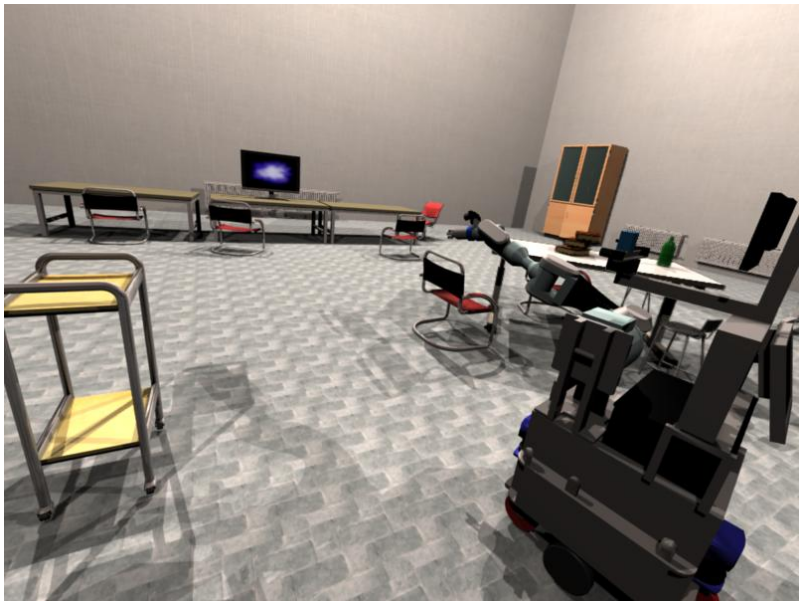


Figure 2.3: MORSE Simulator environment

This last feature is highly valued by engineers as it allows them to make different projects

with equipment with different requirements. Even so, all these projects are based on co-operation and communication between different robots, as well as their interaction with humans. It offers a whole library of configurable components, which allows the user to create all kinds of robots according to the users' needs. A major highlight of this application is its graphic capabilities. This is due to the fact that it runs on a game engine (Blender) [8]. This engine offers the necessary tools to achieve a high level of graphic detail such as textures, shadows and lights. Another great interest is its precision in terms of physics. This is possible because this application uses the Bullet engine.A list of the many advantages offered by this simulator can be drawn up:

Positive Aspects

- It supports graphically realistic 3D simulations, which contributes to a closer approximation to reality.
- As Morse is modular, it allows the user to customise the simulator according to their needs.
- It has a library full of robot and sensor models, thus promoting the application's versatility.
- It integrates with ROS, facilitating collaboration with other ROS libraries. This is extremely useful for projects that use this system.

With all these positive aspects, it seems to be the best simulator presented so far, but the fact that it has such a high level of complexity does bring some cons:

Negative Aspects

- It is not compatible with many computers due to its highly developed graphics level, which limits users considerably. Requires high computing resources.
- Low number of users. This leads to a weaker community, with less sharing of experiences and projects carried out on this simulator.
- Low number of programming languages. It only supports the Python language.
- In terms of functionalities, it only allows the user to visualise what is happening in the simulation environment.

2.2.5 CoppeliaSim (V-REP)

CoppeliaSim, formerly known as V-REP, is a versatile robotic simulator that looks a bit like Gazebo. However, it has a few different features. One of them is that it can simulate in different dimensions. It enables the user to choose between a 2D and 3D simulation. Although there are

many similarities, many engineers prefer to use this simulator because it allows for faster development of motion planning algorithms. It also has a much more extensive library of components and sensors than Gazebo.

This simulator has two great features. The first is that it is possible to programme in multiple programming languages such as Matlab, Python, Java, Lua and C++ /C. The second is that CoppeliaSim supports different physics engines such as ODE, Bullet, Vortex, Newton [5].

Stated in [29], V-REP is based on modular and distributed architecture as it is capable of simulate control, the actuators, the sensors and the monitoring. As can be seen in [32], there are various ways of programming to control the simulation. Even so, these are not completely independent and depend on each other, as can be visualized in 2.4:

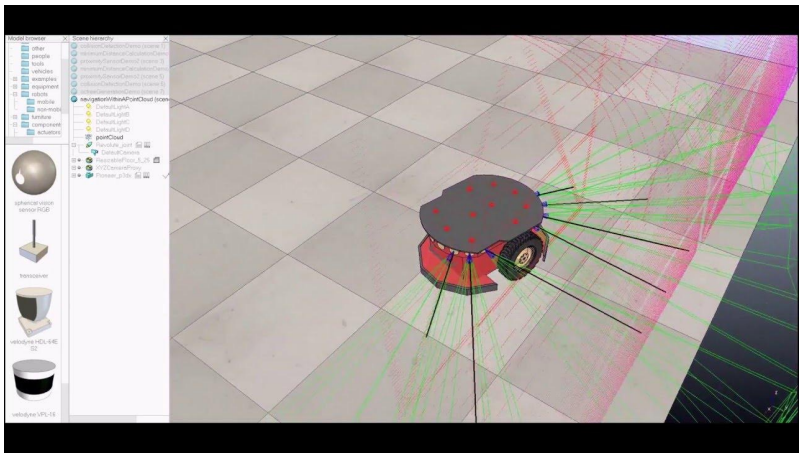


Figure 2.4: CoppeliaSim Simulation Environment

In this way, it is possible to make a list of the pros and cons of using this application in projects. Among the pros, the following points can be highlighted:

Positive Aspects

- It is multi-dimensional in that it allows both 2D and 3D simulations.
- It has functionalities that allow for the rapid development of motion planning algorithms.
- Due to its large community, a vast amount of documentation and tutorials are available.
- CoppeliaSim has a very good physics engine, which promotes the realism of its simulations and makes it a more reliable option than Gazebo.
- Unlike Gazebo, this simulator allows the user to edit the simulation scenarios.
- CoppeliaSim supports different physics engines such as ODE, Bullet, Vortex, Newton.
- It supports Windows, Mac OS and Linux.

- Using a remote API it is possible to program in Matlab, Python, Java, Lua and C++ /C.

Talking now about the cons, it must bear in mind the following points:

Negative Aspects

- Despite being a fairly stable simulator, users with older versions complain a little about this factor.
- For users who are just starting, CoppeliaSim is complicated to work with due to its complexity and robustness.
- Although there is a free version of CoppeliaSim, if it is pretended to use it for industrial purposes, it will be needed to buy a licence.

Compared to other simulators, CoppeliaSim seems to be the most complete and versatile, but it does require the user to already have a certain level of experience. A whole community has therefore been created allowing beginners to have access to tutorials and documents, which shows a high level of mutual support within the community.

2.2.6 Webots

Webots is simulation software that provides an environment for programming, modelling and simulating mobile robots [23]. This platform allows the user to define parameters and modify them on robots. The library provided by the application allows users to transfer their programs to a real mobile robot. As well as being able to modify the robot's parameters, certain aspects of the objects can be changed, such as their colour, texture, shape and even their mass. Webots also has many components such as sensors and actuators [5], which can be associated with the robots to program and simulate. Its architecture is based on nodes with a tree structure, where the nodes represent various parts of the robots, elements and modelling and control functionalities.

The investigation of social behaviour in Webots simulations to date has been mainly concerned with the behaviour of social insects, other Artificial Life experiments, or testing action-selection architectures [24]. Rarely has this simulator been used for modelling aspects characteristic of human-like agents. It is possible to see one of its applications on 2.5.

Figure 2.5: Webots Environment

According to, Webots has several features that make the simulator better and that are essential to it. These are as follows:

Positive Aspects

- Simulates any type of mobile robot, whether it has wheels, legs or is even a flying robot.
- It has an extensive library of sensors and actuators ready to be associated with mobile robots.
- It can be programmed in C/C++, Python, Java, MATLAB or from third party software via TCP/IP.
- Programs on real robots including Lego, Mindstorms, Aibo, Koala and Hemisson can be used.
- Uses the ODE (Open Dynamics Engine) library for more accurate simulation of physics components.
- It has several examples with code and models of commercial robots.
- Facilitates the simulation of multi-agent systems by offering resources that enable global and local communication between agents.
- Webots uses a customised ODE physics engine.
- Extensive documentation.

Even so, it does have some limitations. If it is wanted to access this simulator for a project that requires more complex resources, it won't be possible to use the free version. It would be then

necessary to buy the commercial version of Webots. Amongst these less positive aspects, a few others can be highlighted:

Negative Aspects

- Licence for commercial use will have to be paid for.
- Detailed 3D simulations can require high computing resources, making it more expensive.
- Even though it has substantial customisation capacity, which is an advantage, it ends up becoming a major challenge for a user with a low level of experience.
- Compared to other simulators, Webots is not as susceptible to and prepared for emerging technologies in the field of artificial intelligence.

2.2.7 Simulators Comparison

After analyzing the most popular simulators on the market, highlighting each virtue and each negative aspect, it is possible to summarise them in a table. In the following table, 2.1, those aspects that are most valued when a particular user wants to choose a simulator to carry out a project were chosen.

Features / Simulators	**Gazebo**	**Stage/Player**	**Morse**	**CoppeliaSim**	**Webots**
Free to Use	Yes	Yes	Yes	Yes	Yes
Open-Source	Yes	Yes	Yes	No	Yes
Community	Big Community	Medium sized	Small Community	Medium sized	Small Community
ROS Compatibility	Yes	Yes	Yes	Yes	Yes
Graphical Interface	3D Graphics	2D Graphics	3D Graphics	3D Graphics	3D Graphics
Indoor/Outdoor	Outdoor and Indoor	Indoor	Outdoor and Indoor	Indoor	Outdoor and Indoor
Performance Level of Computer	High	Low	High	High	High
Physics Engine	Bullet, ODE, DART, Simbody	None	Bullet	Bullet, ODE, Vortex, Newton	ODE
Programming Languages	C/C++, Python	C++, Tcl, JAVA, and Python	Python	Python, Lua, C/C++, Matlab, Urbi, Octave or Java.	C/C++, Python, Java, MATLAB

Table 2.1: Simulators Comparison

Analyzing 2.1, it is possible to conclude that Gazebo has, by far, the biggest community among all the simulators making it possible to be always updated. CoppeliaSim, despite having a great Graphical Interface, isn't Open Source. It can lead to major costs, particularly in terms of licenses. For those who don't have a high-performance computer and don't need a detailed simulation, Stage/Player is the perfect choice as it only simulates 2D components. Gazebo, MORSE, and Webots are pretty similar but Morse is too limited in terms of programming languages, only allowing the user to program in Python. The only problem with Webots is that it has a small community, so it is not as prepared as Gazebo for new emerging technologies. Saying that, for users that have a big demanding project and need a graphically precise simulator, Gazebo should be its main choice. On the other hand, if the user has a low-level performance computer and the project demands a simpler simulation, Stage is a great choice.

One of the main objectives of this dissertation is to simulate as many robots as possible and to visualize their behavior. Consequently, the graphics component is accorded a somewhat lesser value, with greater emphasis on the program's scalability. The simulator selected for this purpose was, previously, Stage/Player. Although the gazebo has excellent capabilities, it would require excessive performance for the intended purpose, as it would necessitate the simulation of all ele-

ments in three dimensions. However, despite the capabilities of Stage/Player, the number of robots that could be simulated was limited, necessitating the development of a visualization tool capable of meeting the required specifications.

2.3 Middleware for Robotics Development

The robotic middleware facilitates faster, collaborative, and efficient development of robotic systems [14]. Most of the simulators mentioned in the previous section use the ROS (Robot Operating System) framework. Although ROS remains the most widely used, there are several other middlewares that should be noted and considered.

2.3.1 ROS (Robotic Operating System)

ROS is an open-source framework designed to facilitate the development of software for robotics applications. When it comes to programming, individuals have their own preferences regarding programming languages. Therefore, ROS was developed to offer multiple options so that each person can choose according to their preference. Therefore it supports some different programming languages such as Python, C++, Octave and LISP [13], [30]. It is structured around a modular design approach that is well-suited for peer-to-peer systems where multiple processes, running on different types of hardware, can communicate effectively. The architecture of ROS has four key elements that are going to be presented: nodes, topics, messages, and services.

- **Nodes** are individual software processes within ROS that are designed to perform specific tasks. In a ROS-based system, there are typically numerous specialized nodes, each responsible for distinct functionalities.
- **Topics**: Nodes communicate with each other by publishing and subscribing to data through channels known as topics. Topics act as communication channels where nodes can send messages to one another.
- **Messages**: Messages are the data structures used to transmit information over topics. Nodes publish messages to topics when they have data to share, and they subscribe to topics to receive messages from other nodes.
- **Services**: Services allow nodes to request and receive specific computations or actions from other nodes. Unlike topics, which involve ongoing data exchange, services are used for more direct, request-response interactions between nodes.

Additionaly ROS is often used in conjunction of a 3D simulator like Gazebo and it supports visualization on a tool named rviz, however the first one offers a better quality of simulation. The huge active community that this framework presents, provides new contributes and help the current and new users [14].

2.3.2 Lightweight Communications and Marshalling (LCM)

LCM is a library that enables message passing and marsalling. The main goal of this Middleware is to allow the users to develop low-latency message passing systems [12], especially for real-time applications like robotics [12]. Like ROS, LCM (Lightweight Communications and Marshalling) also provides data serialization and a publish-subscribe based message transport mechanism for processes running on multiple network nodes [35]. This middleware solution optimizes the efficient encoding and transmission of data across distributed systems while supporting dynamic communication patterns through asynchronous message publishing and subscription. Its main goal is to make common tasks of sending messages easily and to make many other tasks possible. LCM also catches many mistakes while the program is running, like data that's not right and types that don't match [12]. Finally, it should be emphasized that LCM, unlike ROS, does not function entirely as middleware. It works more like a library that can eventually be integrated with an existing software framework [14].

2.3.3 ZeroMQ

ZeroMQ [17] is an open-source library designed for building scalable and distributed systems. While it appears to be a library, it also functions as a framework. This middleware provides a socket API that can be utilized for in-process, inter-process, or inter-host communication via the TCP protocol [17]. Furthermore, its sockets are capable of addressing issues caused by asynchrony.

ZeroMQ supports various messaging models for message delivery. These include a centralized approach with a central hub managing message flow between different processes [14], peer-to-peer communication, a model where a central hub acts as a directory server, and a distributed system that can incorporate one or more central hubs. Additionally, it offers multiple communication patterns such as request/reply, producer/consumer, and publisher/subscriber [11].

2.3.4 Yet Another Robot Platform (YARP)

YARP (Yet Another Robot Platform) is an open-source middlweare. As stated in [22], YARP aims to reduce the time and effort spent on infrastructure-level software development by promoting code reuse and modular design, thereby enhancing the focus on research and fostering greater collaboration. One of the key points of YARP is its communication mechanism, which simplifies writing and running processes [22].

YARP supports inter-process communication, image processing, and provides a class hierarchy for easy adaptation across different hardware platforms. Compatible with Windows, Linux, and QNX6, YARP enhances research and collaboration in the dynamic field of humanoid robotics [9].

2.3.5 Choosing the right Middleware

The most widely used middleware for robotics is the Robot Operating System (ROS). Its flexibility in creating robotics applications is one of the main reasons for its popularity. Additionally, its large and highly interactive community makes it easier for beginners to get started with this middleware. Compatibility with a wide variety of simulators is crucial for robotics simulations, and ROS excels in this aspect by supporting numerous simulators.

2.4 Path Planning

Path planning is a critical component in the field of mobile robotics. Numerous methods exist for tracing paths that meet specific requirements. To enable Autonomous Mobile Robots (AMRs) to navigate a given space autonomously and safely, it is essential to select an algorithm that prioritizes both minimizing the distance traveled and avoiding collisions with obstacles. Additionally, the chosen algorithm should be efficient in terms of computational resources, adaptable to dynamic environments, and capable of handling various types of terrain and obstacles. Ensuring robust path planning not only enhances the operational efficiency of AMRs but also significantly contributes to their reliability and safety in real-world applications.

It is therefore important to identify and explain some of the concepts that are often confused in this subject, given the similarity between them and given that the wrong term is sometimes used in common parlance [6]:

- Path Planning: describes in geometrical or Mathematical terms what is the path from a certain initial point to a destination point, avoiding the collision with obstacles.
- Trajectory Planning: represents the path previously obtained from path planning, as a function of time. In other words, it tells us exactly where the robot should be for a given time.
- Motion Planning refers to the process of determining a sequence of movements or actions that a mobile robot should take to get to a specific goal while it is navigating in a certain workspace. This involves several key aspects, such as Dynamic Constraints (forces and torques acting on the robot) and Kinematic Constraints (ensuring the planned path respects the robot's kinematic limitations).

2.5 Conclusions

In conclusion, this dissertation explores the importance of having an extended fleet of Mobile Autonomous Robots (AMRs), on an industrial environment. They make it possible to speed up and save financial resources, focusing on the opportunities presented by the integration of simulation and visualization tools. The adoption of simulators is emerging as a strategy for companies aiming to optimize their operational and economic efficiencies. Simulating the deployment of multiple robots facilitates strategic planning, path programming, and operational supervision. It must be

borne in mind that these simulators have limitations, particularly the fact that they are unable to replicate certain scenarios that could happen in the physical world. The primary objective is to make it possible to link the ROS-based multi-robot system, which is supplied, with the simulation environment visualizer via the ROS framework. The aim is to make it possible to control a much larger fleet than was previously possible.

Chapter 3

System Implementation

The primary aims of this dissertation were to simulate an extensive fleet of robots moving simultaneously on the new visualizer and, at the same time, to test the Fleet manager associated to the visualizer. In industrial environments, engineers require a platform capable of supporting a large scale of Autonomous Mobile Robots (AMRs) to effectively control the entire fleet simultaneously. This chapter details the steps taken to achieve the final product and presents what the TEA* is.

Initially, it was necessary to establish a communication protocol (UDP) between two different pieces of equipment: one running the developed program and the other functioning as the visualizer. To trace each robot's trajectory, the Robot Operating System (ROS) was employed, allowing the reading of graphs that define the movement of each robot.

3.1 Fleet Manager

This fleet manager, developed by CRIIS, is able to assign tasks to each robot in the fleet, plan their trajectories, prioritise their actions and manage traffic to avoid collisions. The architecture of this manager is centralised so that it can control the entire fleet and plan trajectories taking into account those of the other robots. This fleet manager is based on the TEA* algorithm and its approach is modular based in order to allow the system to have a greater scalability and adaptability to different types of tasks. In order to optimize the system and its efficiency, those are received via MES (Manufacturing Executing System) and organized by the Task Scheduling module. This module, associated to the TEA* is responsible to minimize the quantity of conflicts of the paths that are planned during the execution of the tasks and, at the same time, to minimize the deadlock situations. There is also a Supervisor that is capable of change the previous planned paths to new ones in order to evict any collision. So when two different robots are more likely to collide, this system will rethink the trajectory making the environment way safer.

3.2 TEA*

The Time Enhanced algorithm is a search algorithm based on the search method A* [34]. The latter uses weighted graphs. Given the start node, this algorithm makes calculations using (Euclidean and Manhattan distances) to find the cheapest path to the end point. The TEA* algorithm thus adds a new time layer to the A* algorithm. It is designed to handle multiple AGV's systems to avoid collisions and deadlocks and to ensure the safety and efficiency of the system [34]. This new algorithm calculates the minimum path between two nodes over the time planes from k=0 to TMAX. It is important to note that the paths are always computed in a loop to account for collisions [34]. In TEA*, the neighbouring cells that will be occupied belong to the next temporal layers, so that collisions can be avoided and the robot's path can be planned with greater safety [27]. It is also important to note that when robots plan their paths using this algorithm, some have priority over others. So, when the robot with the higher priority is planning its path, the second one will already have taken into account some obstacles on the temporal layers that are related to the first one [28].

3.3 Communication Protocol

Since the visualizer is a separate application from the developed program, there needed to be a method for communication between the two to allow data exchange. In this dissertation, the communication was achieved using the UDP protocol.

To ensure successful communication, certain parameters had to be defined. First, a *socket* was created, which acts as an interface that enables communication between two devices over a network. Next, the server port was set to listen for the incoming data that would be sent. Finally, in the code, the IP address of the device running the visualizer was specified, and on the visualizer, the IP address of the computer running the program was set.

To verify that MRViz was correctly receiving the data and to provide feedback, a socket was also created in the program to function as a server. Consequently, the visualizer operates as a client. The procedure is the same for both platforms: the IP and Port fields are filled in, and communication is successfully established.

The following figure 3.1 provides a succinct overview of the operational principles involved in the project's UDP communication between the developed program and the visualizer.

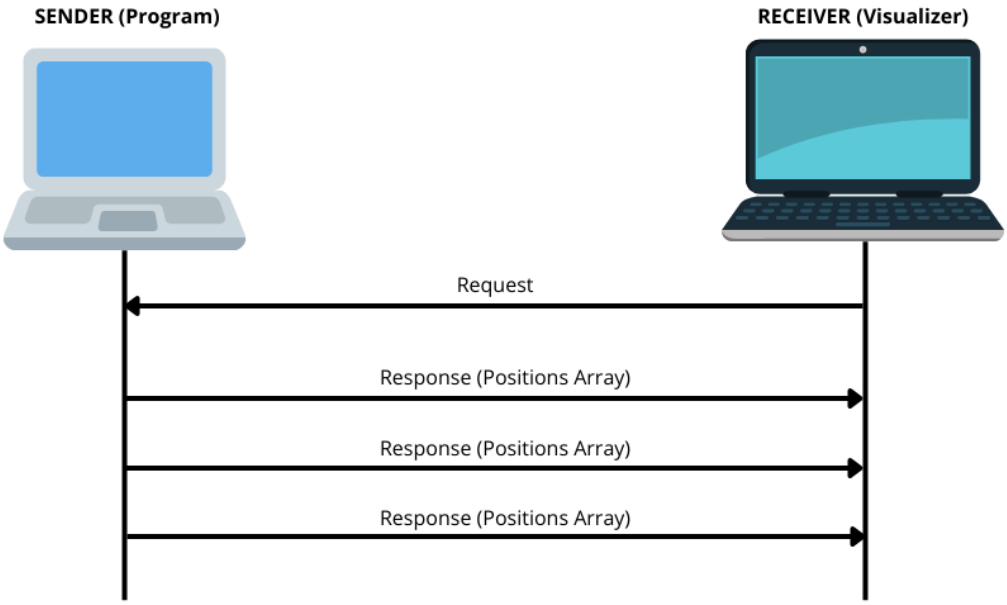


Figure 3.1: Project's UDP communication between the Program and Visualizer

3.4 Graph Reading

It was required that this dissertation utilize the Robot Operating System (ROS) framework. In this case, YAML (Yet Another Markup Language) files define the graphs which describe the movement and behavior of the robots. These files are published as topics within the ROS environment. To integrate in the developed program, a ROS node was created. This node serves as a communication point within the ROS network. A subscriber was then created within this node to listen to the specified topic. When messages containing graph data are published to the topic, the subscriber receives these messages and processes the graph information accordingly.

3.4.1 Graph Composition

As mentioned earlier, the graphs are defined in the YAML language by edges and vertices, each containing essential information for tracing the robots' paths. It is possible now to explain each of these parameters.

Parameters that Define Edges

- **CurveType** - Defines the type of curve of the trajectory that is pretended. In this project is mainly "Spline" but it can also be "Linear".
- **Destination_ID** - Defines what's the ID of the destination node.

- **ID** - ID of the Edge.
- **Origin_ID** - ID of the origin Node.
- **ParamB** - Defines the Backwards Parameter used to trace the Bézier Curves (Spline). Explained in 3.2.
- **ParamF** - Defines the Forward Parameter used to trace the Bézier Curves (Spline). Explained in 3.2.
- **VelocityForward** - In the case of this project, it defines the velocity of the Robot from one Node to another.

Parameters that Define Nodes

- **FameID** - The ID of the frame of the graph.
- **ID** - Represents the ID of the Node.
- **Theta** - The angle orientation of the robot on this specific Node.
- **X** - Defines the coordinate of this Node on the X-Axis
- **Y** - Defines the coordinate of this Node on the Y-Axis

With all those parameters it is now possible to calculate and plan the trajectory of the robots. When the graphs are read, all the fields are stored on a dynamic two-dimensional array. When the values then need to be used to plan the trajectory, the vector is followed until what is required is found.

3.4.2 Trajectory Planning

As discussed in the previous subsection, graphs play a crucial role in enabling the tracing of trajectories. They specify not only the type of trajectory that needs to be followed but also the coordinates of the start and end points. For a Linear *CurveType*, calculating the trajectory is straightforward since it involves using the Euclidean distance formula between two points.

However, when it comes to a Spline, Bézier curves are used. To effectively plan the trajectory using Bézier curves, it is essential to have not only the start and end points and their respective angles but also two additional parameters: "paramB" and "paramF". These parameters help in calculating two additional control points that define the curve's shape.

In the following figure (Figure 3.2), it is possible to better understand how the trajectory is plotted using this method. The figure provides a visual representation of the control points and how they influence the resulting curve, ensuring a smooth and accurate path between the start and end points.

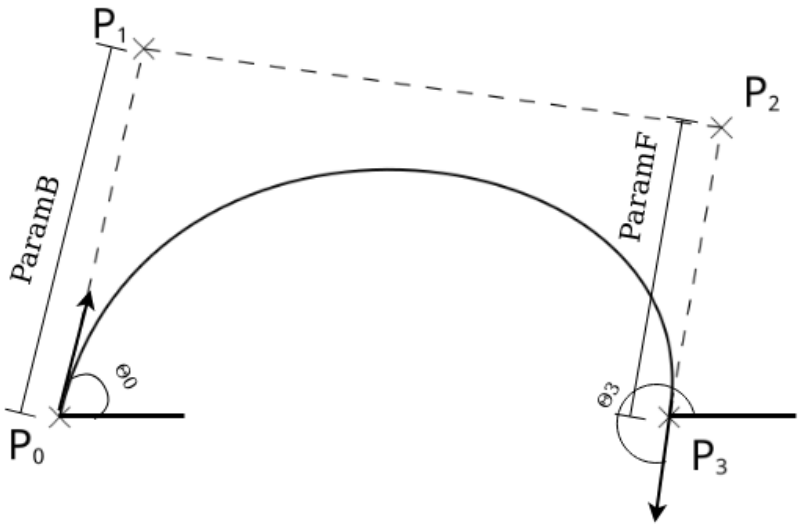


Figure 3.2: Bézier Curve description

With all this data, Trigonometry can be used to calculate P_1 and P_2. The following 3.3 and equation 4.1 show how the calculations were made and how the angle in P_3 was manipulated in order to calculate P_2. Firstly, two lines were drawn which will be the hypotenuses of the two triangles formed, 3.3.

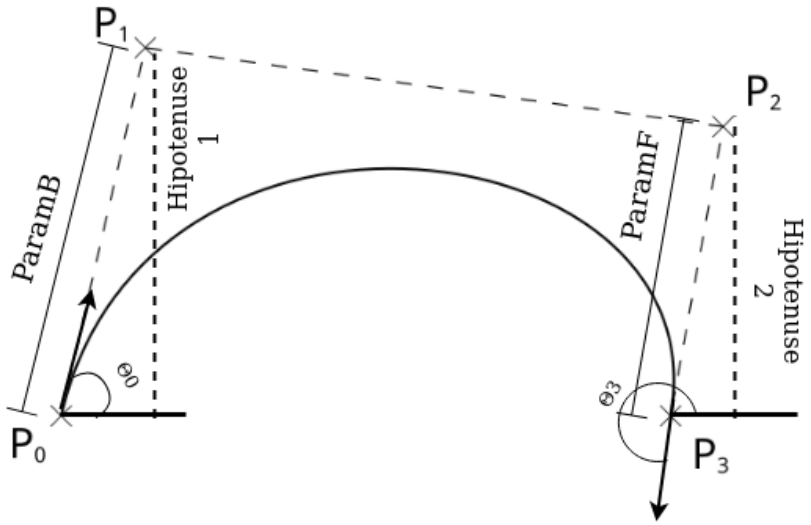


Figure 3.3: Trace the triangles hypotenuse

The following section outlines the methodology employed to derive the formulae utilized in calculating the coordinates of both P1 and P2. Initially, the angle formed at the endpoint between the line representing "paramF" and the origin was determined. Through a series of calculations, it was established that the value of θ_3 could be expressed as follows:

$$\theta_3 = \theta_3 + PI \tag{3.1}$$

With this calculation, it would now be possible to calculate P_2, since the value of the angle is now correct for trigonometric calculations. This brings us to the following equations:

$$P_1(X) = x_0 + (cos(\theta_0) * ParamB) \tag{3.2}$$

$$P_1(Y) = y_0 + (sin(\theta_0) * ParamB) \tag{3.3}$$

$$P_2(X) = x_3 + (cos(\theta_3) * ParamF) \tag{3.4}$$

$$P_2(Y) = y_3 + (sin(\theta_3) * ParamF) \tag{3.5}$$

Having already obtained the four points that define the Bézier curve, these values can be used in the 3.6 and 3.7 equations, which are responsible for plotting the Bézier curves. Using $P_0(x_0, y_0)$, $P_1(x_1, y_1)$, $P_2(x_2, y_2)$ and $P_3(x_3, y_3)$ the following calculations are made:

$$\begin{aligned} c_x &= 3 \times (x_1 - x_0) \\ b_x &= 3 \times (x_2 - x_1) - c_x \\ a_x &= x_3 - x_0 - c_x - b_x \\ c_y &= 3 \times (y_1 - y_0) \\ b_y &= 3 \times (y_2 - y_1) - c_y \\ a_y &= y_3 - y_0 - c_y - b_y \end{aligned} \tag{3.6}$$

Having obtained these values, it's possible now to calculate each point on the robot's trajectory. The following equations are used:

$$\begin{aligned} x(\lambda) &= a_x\lambda^3 + b_x\lambda^2 + c_x\lambda + x_0, \quad 0 \leq \lambda \leq 1 \\ y(\lambda) &= a_y\lambda^3 + b_y\lambda^2 + c_y\lambda + y_0, \quad 0 \leq \lambda \leq 1 \end{aligned} \tag{3.7}$$

The curve is parameterized by λ, ranging from 0 to 1. In this project, it was implemented one hundred points along this parameter range. These points were evenly distributed along the curve and stored in a vector. Each point represents a specific position along the trajectory. After obtaining the one hundred points that define the curve, the function to control the movement of each robot can be implemented. To achieve this, it is necessary to consider the Velocity Forward point provided by the graphs. When this function is called, it will receive the X and Y coordinates

of the robot's current position, the X and Y coordinates of the next Bézier point, the Velocity Forward value, and the time value (t) as parameters. Initially, the X and Y velocity components are calculated as demonstrated in the 3.8, being that θ_c represents the current angle of the robot.

$$\begin{aligned} v_x &= v \cdot \cos(\theta_c) \\ v_y &= v \cdot \sin(\theta_c) \end{aligned} \tag{3.8}$$

In the given scenario, the provided values furnish the necessary conditions for computing the subsequent position of the robot. This task can be accomplished using the equations referenced as 3.9, which essentially constitute the fundamental position equations.

$$\begin{aligned} x_1 &= x_0 + v_x \cdot t \\ y_1 &= y_0 + v_y \cdot t \end{aligned} \tag{3.9}$$

Given that calculations can often lead to errors, an algorithm capable of correcting the position of the robots if they stray too far from their planned route was also implemented into this function.

Having now all the robots' next positions, all that remains is to associate them with each of them individually. This function returns the X and Y coordinates obtained, which are then equated to the X and Y coordinates of each of the robots and then sent to the visualizer.

3.4.3 Angle Definition

To ensure the robot follows the Bézier curve correctly, it is crucial that it maintains the correct angle at each point along the path. To achieve this, a method that calculates the tangent at each point of the trajectory automatically was developed. This approach allows the robot to adjust its orientation seamlessly as it travels along the curve.

To calculate the tangent of a curve, it's first needed to derive the equations that define it. The equations for x(λ) and y(λ), which are given in Equation 3.7, were differentiated to obtain the following derivative equations:

$$\begin{aligned} \frac{dx}{d\lambda} &= 3a_x\lambda^2 + 2b_x\lambda + c_x \\ \frac{dy}{d\lambda} &= 3a_y\lambda^2 + 2b_y\lambda + c_y \end{aligned} \tag{3.10}$$

Once $\frac{dx}{d\lambda}$ and $\frac{dy}{d\lambda}$ are obtained, calculating the angle is straightforward by taking the arctangent of the ratio $\frac{\frac{dy}{d\lambda}}{\frac{dx}{d\lambda}}$. This calculation is performed using the function *atan2(dy, dx)*, resulting in one hundred angle values stored in a vector.

With these angle values at hand, the robot is equipped to move along the Bézier curve accurately and smoothly. During each iteration, the robot knows its designated point and the corresponding angle, ensuring precise navigation along the path.

This method not only ensures that the robot follows the curve correctly but also allows for dynamic adjustments in orientation, providing a seamless movement experience along the Bézier curve.

3.5 Overview

This chapter presents all the program development that was done to make a simulation with a large number of robots possible. Of particular note is the planning of the robots' trajectories, as well as the calculation of the angle along each point of the Bézier curve. These last two points are essential to be able to control a large-scale fleet of robots, making sure that each of the players fulfills their routes without influencing others.

Chapter 4

3D Visualizer

4.1 Introduction

In order to be able to effectively control a fleet of autonomous mobile robots, it is necessary to have a visualization platform that makes it possible to follow and check the behaviour of each piece of equipment as they make their respective trajectories. A 3D visualisation platform was used to facilitate this interaction and management, MRViz. Developed using Lazarus IDE and other important libraries, this tool's main objective is to make it possible to visualise an extensive group of AMRs.

This chapter explores this tool in detail, starting with a description of its main features, which include the visualisation of complex objects, interactive camera manipulation and the integration of realistic textures. It also discusses MRViz's integration with project-specific data, highlighting its ability to provide meaningful insights through the visual representation of certain relevant information.

Figure 4.1, provides a concise overview of the project's functioning, presented in the form of a block diagram.

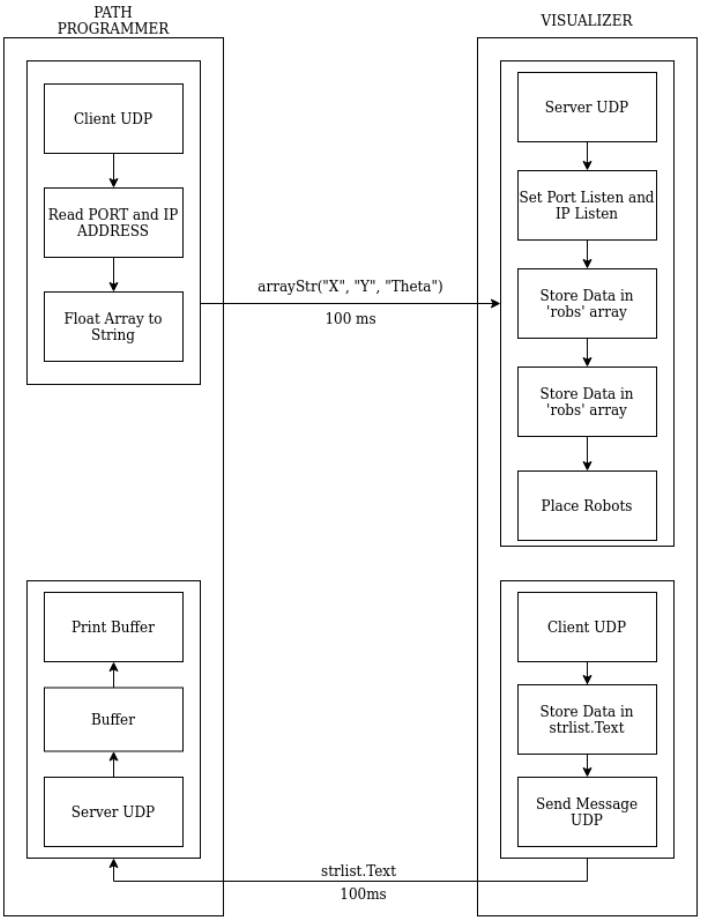


Figure 4.1: Block Diagram of the Project

4.2 Implementation's Technologies

As mentioned in the previous section, for the development of MRViz, some libraries associated with an IDE platform, Lazarus, were used. Through programming in Pascal, combined with the use of certain important packages, such as GLScene, it was possible to code a scenario for the navigation of the robots.

4.2.1 Lazarus

Lazarus is an IDE for software development based on a Free Pascal compiler. As such, it has and provides a wide range of functionalities. In this case, it's worth highlighting the fact that it has an entire graphics development environment, OpenGL. This programming interface allows users to create 2D and 3D graphics applications through its vast list of visualisation functions. Therefore,

this API was used to create a whole environment for navigating AMRs in a simplified and very intuitive way.

4.2.2 GLScene

GLScene is an open source 3D library based on OpenGL, designed for Delphi, C++ Builder and Lazarus. It uses the OpenGL API to render 3D scenes simply and efficiently. GLScene works as a framework to which 3D objects and effects provided by the library can be added. When using GLScene, it is usually created a scene component (TGLScene) and it is associated with a form (TForm). The TForm is a class that provides a window surface where the 3D graphics are displayed. In figure 4.2, it's possible to see the GLScene editor in which all the components that were to be obtained in the final scenario were placed:

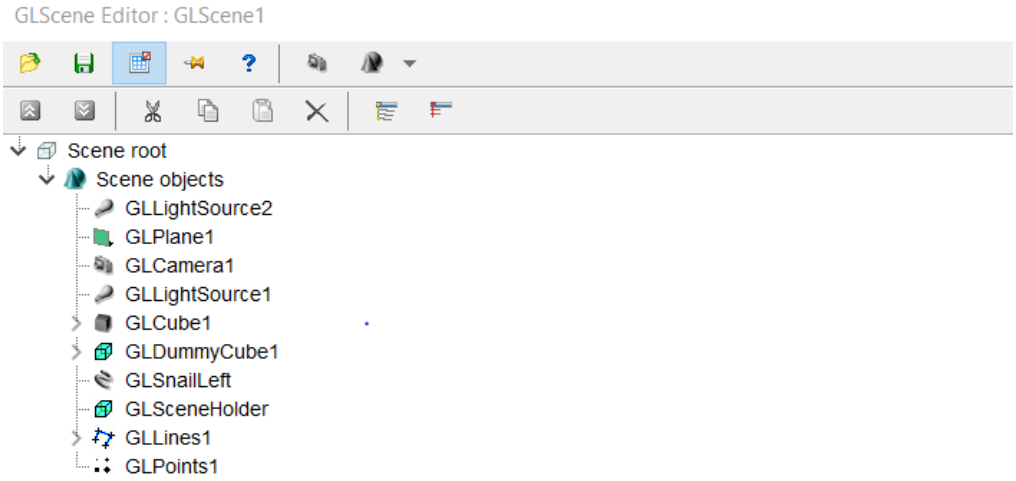


Figure 4.2: Project's GLScene Editor

As can be seen, this Scene root has several Scene Objects. Each one can be dealt with individually, explaining its role:

- **GLLightSource** - In GLScene, GLLightSource is a component that can be used to add and configure light sources on the 3D scene. This object belongs to the TGLightSource class that has parameters for the lights position, spot light direction, color, attentuation and light behaviour. The maximum of those on a scene is set in 8.

- **GLPlane** - It is an object that refers to the **TGLPlane** class. This class is usually used to create flat surfaces in 3D environments. It can be configured in terms of position, texture and dimensions. In order to the user to edit the parameters of the floor, buttons and labels were declared on the TForm and were lately associated to its function. In that way it is possible to choose the sizes (4.1) and the image that represents the surface (4.2).

```
1     GLPlane1.Height:=strtofloat(EditMapSizey.Text);
2     GLPlane1.Width:=strtofloat(EditMapSizex.Text);
```

Listing 4.1: Height and Width of the Plane in TForm

```
1       //GLPlane1.Material.Texture.Image.LoadFromFile('floor2.bmp');
2       GLPlane1.Material.Texture.Image.LoadFromFile(LabelFloorName.Caption);
```

Listing 4.2: Change Plane Code

In 4.1 and 4.2 it's shown the code that was used to program the visualizer that allows to edit the height, width of the surface and the texture of it. The sizes are chosen on two different labels that are present in MRViz. The user writes it and after that the values are converted to float and stored in the respective variables. On the other hand, to change the floor, it's possible to choose a file on the computer and turn it into the environment surface. As seen in 4.2, the user can change it on the Lazarus editor or on the visualizer. On the 3D App, there is a button that opens the computer files and then the user chooses it (4.3).

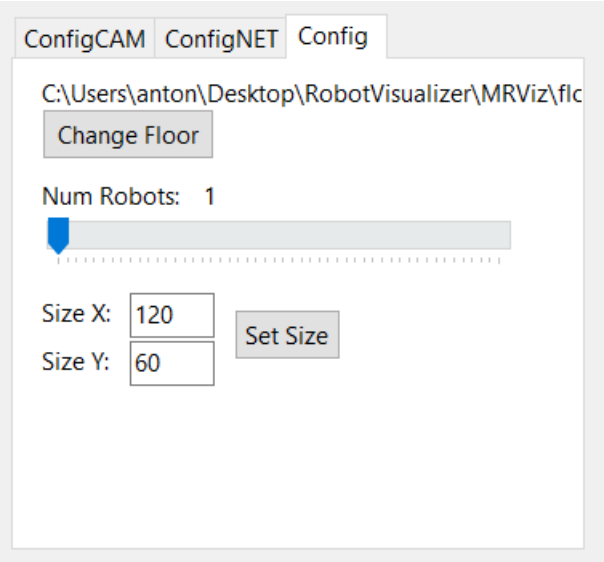


Figure 4.3: Visualizer's Change Floor Button

- **GLCamera1** - It is a component that belongs to the TGLCamera class. Represents a camera in a 3D scene and makes it possible to visualize the scene from its point of view. The class includes position, direction, focal length and depth of view. When the visualizer is running, the user can change the position by its coordinates (X,Y,Z) using the Mouse Movement and the Mouse Wheel. Now its going to be shown how it was all programmed on the TForm.

```
1     procedure TForm1.GLSceneViewer1MouseDown(Sender: TObject; Button:
          TMouseButton; Shift: TShiftState; X, Y: Integer);
```

```
2 begin
3   mx := X;
4   my := Y;
5   mx2 := X;
6   my2 := Y;
7 end;
```

Listing 4.3: Mouse Down Procedure

This procedure captures the initial position of the mouse when the left button is pressed. So, the mx, my, mx2 and my2 are storing the initial value of the coordinates X and Y of the mouse. After that is needed a procedure that is called when the mouse moves.

```
1     procedure TForm1.GLSceneViewer1MouseMove(Sender: TObject; Shift:
          TShiftState; X, Y: Integer);
2 begin
3   if ssLeft in Shift then begin
4     mx2 := X;
5     my2 := Y;
6     getcampos(); // Update the labels with the current camera position
7   end;
8 end;
```

Listing 4.4: Mouse Move Procedure

So, as seen in 4.4, when the mouse is moved this function is called.The mx2 and my2 values are updated to the new ones. Those are updated so another function can be called in order to do the final objective, move the camera. In 4.5 is displayed how it is done.

```
1     procedure TForm1.GLCadencer1Progress(Sender: TObject; const deltaTime,
          newTime: Double);
2 begin
3   if ((mx <> mx2) or (my <> my2)) then begin
4     GLCamera1.MoveAroundTarget(my - my2, mx - mx2);
5     mx := mx2;
6     my := my2;
7   end;
8   if (GLCamera1.Position.Z < 0.1) then GLCamera1.Position.Z := 0.1;
9 end;
```

Listing 4.5: Cadencer Progress Procedure

This function is called continuously to update the camera position. It operates whenever the value of mx2 is different from mx or when the value of my2 is different from my. When it is verified, it's called GLCamera1.MoveAroundTarget with the parameters (my-my2) and

(mx-mx2). That is a method that the GLScene library provides and it's responsible for the adjustment of the camera in order to orbit around a certified point. The both parameters represent the following:

$$VerticalAngle(Y-axismovement) = my - my2 \tag{4.1}$$

$$HorizontalAngle(X-axismovement) = mx - mx2 \tag{4.2}$$

In equation 4.1 it is being calculated the mouse position difference in Y axis. This value will determinate how much the camera has to rotate, up or down, around the point. Conversely, in 4.2 is referred to the X axis and it will determine how the camera has to rotate on the right or left way. The final product is showed in 4.4.

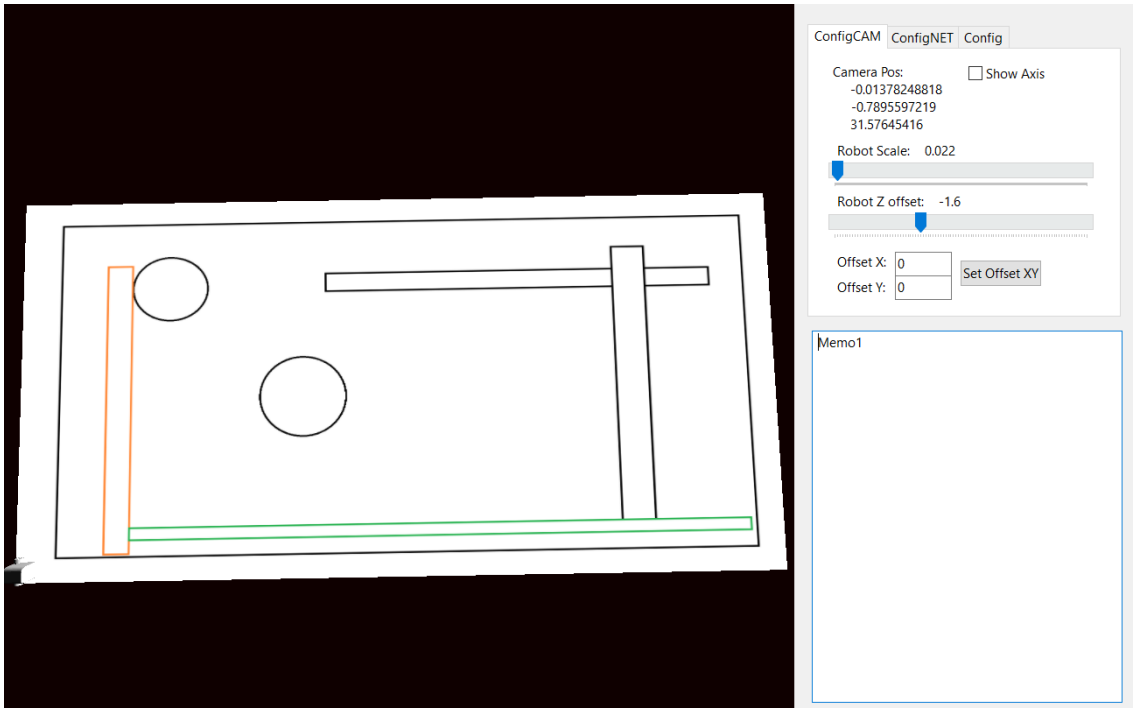


Figure 4.4: Camera View and Coordinates

- **GLCube** - Belonging to the TGLCube class, it represents a 3D cube within the GLScene framework. In terms of properties the width, height, position, orientation and the material can be highlighted. In this project, however the GLCube is declared on the TForm, it is not programmed to appear on the visualizer, so there are no interactions with it.
- **GLSceneHolder** - This object is initially declared as being a TGLDummyCube that is a type of object that in the GLScene library responsible for grouping other ones together. This class also allows to move, rotate and scale the entire group as it works as a parent of the other scene objects.

```
1   for i := 0 to nrobots - 1 do begin
2    freeforms[i] := TGLFreeForm(GLSceneHolder.AddNewChild(TGLFreeForm));
3    freeforms[i].LoadFromFile('box.stl');
4    freeforms[i].Position.X := (- GLPlane1.Width / 2) + i * 2;
5    freeforms[i].Position.y := - GLPlane1.Height / 2;
6    freeforms[i].Position.z := freeforms[i].Position.z + robotzoffset;
7
8    freeforms[i].Scale.X := scalestl;
9    freeforms[i].Scale.y := scalestl;
10   freeforms[i].Scale.z := scalestl;
11  end;
```

Listing 4.6: Declaring TGLFreeForm objects as children of GLSceneHolder

In 4.6, there is the part of the code that turns the TGLFreeForms obects into GLSceneHolder children. So, in the loop that is shown, TGLForm objects are being dynamically created, they are added as GLSceneHolder children and after that the position and scale are set. As it works with **Hierarchy**, when any transformation is applied to the GLSceneHolder, all its children is going to be affected too.

So, the big advantage of using the **"TGLDummyCube"** class, is that it's possible to manage a whole fleet of objects simultaneously instead of control each one individually.

4.2.3 Important Aspects of the TForm

The TForm contains all the visual and functional aspects of the Lazarus application. It integrates some GUI elements with 3D rendering capabilities in order to create a whole interactive scene for the users. The main functionalities of the TForm are the following:

- **Handling Events** - Contains procedures that react to interactions between the user and the system like Mouse movements (**GLSceneViewer1MouseMove**), Button clicks (like **ButtonChangefloornameClick**) and form creation (**FormCreate**).
- **Integration of Different Components** - As refered on the previous chapter, it has the capability to integrate some objects that belong to the GLSceen library and some others from Lazarus. As can be seen on the following code, two packages were mainly used, the **LCLBase** and the **GLSceneRunTime**:

```
1   TForm1 = class(TForm)
2    ButtonConnect: TButton;
3    ButtonSetOffsetXY: TButton;
4    ButtonChangefloorname: TButton;
5    GLCadencer1: TGLCadencer;
6    GLCamera1: TGLCamera;
7    GLCube1: TGLCube;
```

```
8     GLDummyCube1: TGLDummyCube;
```

Listing 4.7: Declaration of Components in TForm

- **Data Management** - It manages state and data related to the application's functionality, such as storing robot positions, managing the Network communications and handling the user input. The first two that were mentioned, are extremely important in the context of the project. The main objective is to place the robots on the 3D scene correctly. For that it is first needed to connect, using the UDP protocol, to the client. For that, it was first created a **'LUDPComponent1'** that belongs to the class **'TLUDPComponent'** class, enabling this protocol communication for the application. Following that, a procedure was created. This situation is shown in 4.8.

```
1   procedure TForm1.LUDPComponent1Receive(aSocket: TLSocket);
2 var
3   recv: string;
4   outlist: TStringDynArray;
5   i: integer;
6   parsestr: TStringList;
7
8 begin
9   parsestr := TStringList.Create;
10  LUDPComponent1.GetMessage(recv);
11  parsestr.Text := recv;
```

Listing 4.8: Creation of the TForm1.LUDPComponent1Receive procedure

The received message is stored in the string recv using **LUDPComponent1.GetMessage(recv)**. Subsequently, **parsestr.Text** is populated with the content of **recv**. As shown in 4.9, if **parsestr.Count** contains any data, the next step is to split this string by commas, filling the dynamic string array **outlist**. This array will have four elements: the position [0] is reserved for the robot's ID, [1] for the robot's X coordinate, [2] for the robot's Y coordinate, and [3] for its Theta angle. When the first position of **outlist** has a value of zero or greater, the robot positions are stored in the **robs** array. Since these values are initially strings, the **StrToFloat** function is used to convert them to floating-point numbers before storing.

```
1 if parsestr.Count >= 0 then begin
2     for i := 0 to parsestr.Count  -1 do begin
3       outlist := SplitString(parsestr[i],',');
4       if (StrToFloat(outlist[0])>= 0) then begin
5         Edit1.Text:= inttostr(parsestr.Count) +'--'+ outlist[0]+' '+outlist
              [1] +' '+outlist[2]+' '+outlist[3];
6         robs[trunc(StrToFloat(outlist[0]))].x:=StrToFloat(outlist[1]);
7         robs[trunc(StrToFloat(outlist[0]))].y:=StrToFloat(outlist[2]);
```

```
8           robs[trunc(StrToFloat(outlist[0]))].theta:=StrToFloat(outlist[3]);
9         end;
10      end;
11    end;
12    placerobots();
13    if CheckBoxSendUDP.Checked then feedbackposits();
14  end;
```

Listing 4.9: Robot positions storage

The procedure ends by calling the function **placerobots**. Equaling the freeforms positions and angle to the values that were stored in robs array, the robots are now ready to be placed on the 3D scene. 4.10 shows how it is done in the code editor.

```
1  procedure TForm1.placerobots();
2  var i: integer;
3    node: TGLLinesNode;
4  begin
5    memo1.Clear;
6    for i := 0 to nrobots -1 do begin
7       freeforms[i].Position.X := robs[i].x + offsetX;
8       freeforms[i].Position.y := robs[i].y + offsety;
9       freeforms[i].RollAngle := robs[i].theta;
10      node := GLLines1.Nodes.Add as TGLLinesNode;
11      node.AsVector := VectorMake(robs[i].x + offsetX, robs[i].y + offsetY,
            robotzoffset);
12
13      memo1.Lines.Add(Format('%d %.2f %.2f %.2f', [i, robs[i].x + offsetX, robs
            [i].y + offsetY, robs[i].theta]));
14    end;
15     GLSceneViewer1.Refresh;
16
17 end;
```

Listing 4.10: Robots Placement

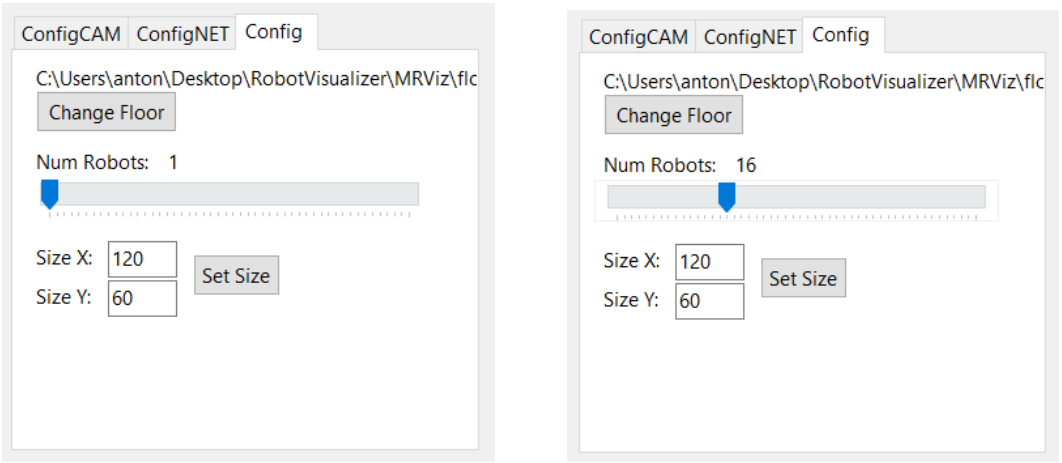
In figure 4.9, there is also a function that is called if, on the visualizer, the box destined to do the UDP communication back is checked. In case of it being true, the computer that was previously a server, becomes a client by calling the procedure **TForm1.feedbackposits()**. On this function, a string list is created and after that, robot by robot, the coordinates and angles are passed to it as strings. After the list is fully filled, **LUDPComponent1.SendMessage** function is called and "strlist" is sent to the other computer (now working as a server).

```
1  procedure TForm1.feedbackposits();
2  var
3    i: integer;
```

```
4    strlist: TStringList;
5    txt: String;
6  begin
7    strlist:= TStringList.Create;
8    for i := 0 to nrobots -1 do begin
9       txt := inttostr(i) + ' ';
10      txt := txt + floattostr(freeforms[i].Position.X) + ' ';
11      txt := txt + floattostr(freeforms[i].Position.Y) + ' ';
12      txt := txt + floattostr(freeforms[i].RollAngle);
13      strlist.Add(txt);
14   end;
15   LUDPComponent1.SendMessage(strlist.Text,EditIPout.Text+':'+EditPortout.Text
         );
16   strlist.Free;
17 end;
```

Listing 4.11: TForm1.feedbackposits() function

There are still another aspects that have to be refered and explained. One of the biggest goals of the project is to be able to control and visualize a big fleet of robots. Therefore, the platform has to allow the user to choose how many robots are pretended to be simulated and, when the number is chosen, this number of robots has to be created and displayed on the visualizer. In order to do it, a Trackbar is placed on the simulator. When it is pushed to the right, the value increases and to the left it decreases. On the TForm there's a function that stores the value of the position of the bar, as a string, on a label named *LabelNumRobots*. After that, when *FormShow* procedure is ran, the string is converted to an integer and stored in *nrobots*. It's then made a loop to create the number stored on *nrobots* as can be seen in 4.6. It is important to mention that while the program is running the number of robots can not be changed, it is a fixed and constant number. Below (figure 4.5), it's possible to see how the bar works and how it is represented on the visualizer.



(a) 1 Robot total

(b) 16 Robots total

Figure 4.5: Number of Robots Track Bar

4.3 Summary

This chapter presented how the visualizer works and how it was programmed. Lazarus with its libraries allows the users to program a scene using multiple objects that can interact with each others. GLScene plays a fundamental role providing a lot of different effects, components and 3D objects, that enables to create a more realistic scenario for the simulation. It is always associated to the TForm where it's possible to program every single object and giving it a special function.

Chapter 5

Tests and Results

In this chapter, all the results obtained during the completion of this dissertation will be presented. Detailed explanations of all tests conducted to verify the feasibility of the project will be provided. The results will be thoroughly analyzed and discussed to demonstrate the project's validity and effectiveness.

The visualizer will be demonstrated with the robots traveling along the pre-designed trajectory. The time evolution of their x and y positions, as well as the Theta angle, will also be shown.

5.1 Trajectory Planning Tests

As discussed in Section 3, the x and y coordinates of each robot, along with the corresponding theta angle of the Bezier curve, are stored in a vector. These values are then applied within each loop iteration to realize the robot's trajectory. To verify the accuracy of this method, time-based graphs were generated with the x and y coordinates on one graph and another graph plotting the theta angle against time as the robot moved. This approach confirmed that the splines were correctly drawn and the angles appropriately followed the robot's displacement.

5.1.1 Trajectory Planning Graphs

To facilitate analysis, lines of code were incorporated into the program to record all these values in an Excel sheet. Using these recorded values, graphs were plotted to inspect the trajectories visually. For example, in Figure 5.1, it's possible to observe the spline drawn for a robot starting at the position (X, Y): (13.0, 17.5) and ending at position (-5.0, -8.0). This visualization demonstrates the precision of the implemented method.

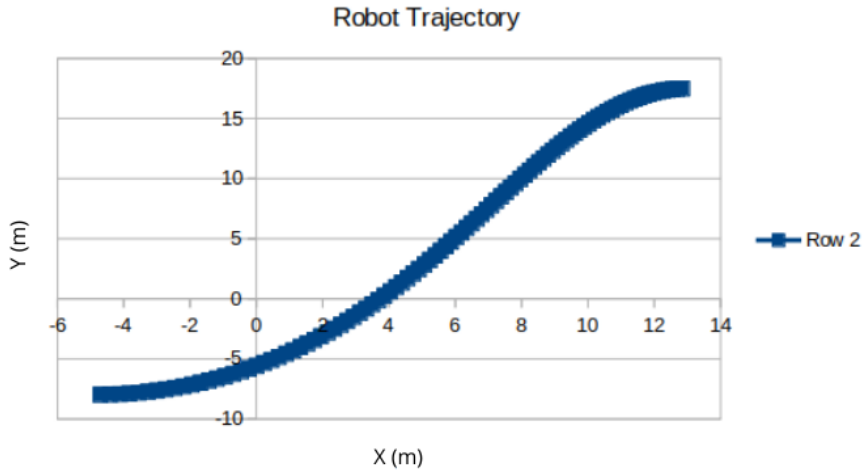


Figure 5.1: Bézier Curve between Two Points

In addition to analyze the robots' positions over time, it is crucial to examine the orientation angles (θ) to ensure that the robots' movements adhere precisely to the planned trajectories. Figure 5.2 illustrates a graph depicting the variation in the robots' orientation angles as a function of the Bézier Curve points.

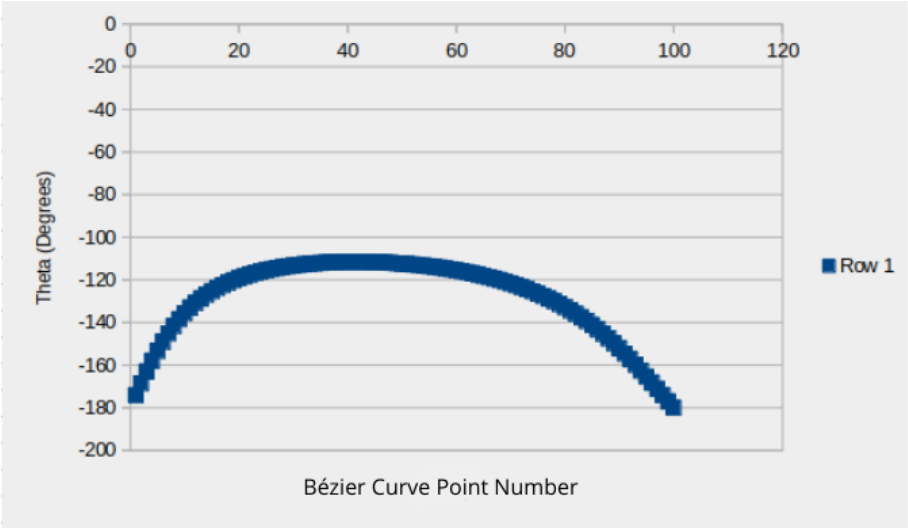


Figure 5.2: Angles of the Robot in Bézier Curve

5.1.2 3D Visualizer Tests

After confirming the accuracy of the trajectory computation method, which included verifying both the correct positioning of the robot at designated points and the accurate calculation of theta values, the focus shifted to ensuring seamless integration with the visualizer. The objective was twofold: first, to validate that the robots followed the computed path precisely as depicted in the

trajectory graph; second, to ascertain that the robots maintained the correct orientation angles throughout the movement.

To achieve this, an object was employed to trace a continuous line connecting the sequential points through which the robot passed. This visual representation served as a direct comparison against the trajectory graph, providing a visual confirmation of the path accuracy and the alignment of angle values.

5.1.3 Success Rate

In the context of simulations, it is crucial to evaluate the success rate when running the program. To this end, a robot was simulated traveling along multiple edges, to determine the error associated with each one. Initially, simulations were conducted on 10 different edges, recording the final X and Y coordinates and comparing them to the expected values at each node. Subsequently, routes comprising multiple edges were simulated to investigate whether increasing the number of edges would affect the error, either by increasing, decreasing, or maintaining it. Each simulation was repeated three times to ensure the reliability of the results.

VARIABLES / EDGES	X (Sim)	Y (Sim)	θ (Sim)	X (Real)	Y (Real)	θ (Real)
45	13.00	13.55	90.03°	13.00	13.5	90°
34	-5.01	-8.02	-180°	-5	-8	-180°
16	28.99	-15.54	-90°	29.00	-15.5	-90°
12	26.97	17.51	180°	27.00	17.50	180°
11	16.51	10.47	-180°	16.52	10.47	-180°
13	12.97	17.50	180°	12.99	17.50	180°
23	35.01	18.50	0.005°	35.00	18.50	0°
31	36.14	29.99	90.03°	36.14	29.94	90°
29	17.16	23.35	90.03°	17.16	23.34	90°
27	-5.02	-8.00	-180°	-5.00	-8.00	-180°

Table 5.1: Edges Simulation Results

Having obtained these values, the calculation of the success rate by comparing the experimental value and the real value can be done. Thus, in the table 5.2 the rates for each of the parameters,

X, Y and θ can be checked. The formula presented in 5.1.3 was used for the calculation.

$$\text{Success Rate (\%)} = \left(1 - \frac{|\text{Real Value} - \text{Experimental Value}|}{|\text{Real Value}|}\right) \times 100 \tag{5.1}$$

Where:

- Real Value is the value that is pretended.
- Experimental Value Is the value that was obtained on the simulation.

Percentage / EDGES	X Success (%)	Y Success (%)	θ Success (%)
45	100%	99.6%	99.97%
34	99.86%	99.99%	100%
16	99.97%	99.75%	100%
12	99.89%	99.99%	100%
11	99.93%	100%	100%
13	99.78%	100%	100%
23	99.99%	99.98%	99.95%
31	99.99%	99.83%	99.97%
29	100%	99.96%	99.97%
27	99.65%	99.99%	100%
AVERAGE	99.90%	99.91%	99.98%

Table 5.2: Success Rate

Based on the data analysis, it is evident that the algorithm performs exceptionally well for individual edges, achieving remarkable accuracy. Notably, the lowest average accuracy percentage is for the X coordinate, which still maintains an impressive value of 99.90%. Additionally, the angles are calculated with extraordinary precision, showing a minimal error of just 0.02%. The consistency of the program is further demonstrated by the uniformity of results across all test repetitions, as reflected in the tables. Next, the program's scalability will be evaluated by testing it with multiple edges for the robot. The subsequent results will be compared to determine whether the success rate remains stable, decreases, or potentially increases.

VALUES / EDGES	X (Sim)	Y (Sim)	θ (Sim)	X (Real)	Y (Real)	θ (Real)
34, 45	13.01	13.55	90.03°	13.00	13.50	90.03°
13, 34	-5.03	-8.01	-180°	-5.00	-8.00	-180°
23, 16	29.00	-15.51	-90°	29.00	-15.5	-90°
11, 13, 34	-5.033	-8.01	-180°	-5.00	-8.00	-180°
23, 16, 12, 11	16.49	10.47	-180°	16.52	10.47	-180°
29, 31	36.14	29.94	90.03°	36.14	29.94	90.03°
23, 16, 12, 11, 13	12.98	17.50	180°	12.99	17.50	180°
23, 16, 12, 11, 13, 34, 45	13.01	13.55	90.03°	13.00	13.5	90.03°

Table 5.3: Edges Simulation Results

As it is possible to see, the values considered here were those of the last edge. This allowed to check that the transition between each edge was correctly done, considering whether the final values were comparable to those obtained during the individual tests.

This being the case, it is viable to use the 5.1.3 equation to calculate the success percentage of these tests. It should also be remembered that to guarantee the viability of the results, the tests were each repeated five times, and in each case, the final values were the same since there was no variation.

Percentage / EDGES	X Success (%)	Y Success (%)	θ Success (%)
34, 45	99.99%	99.6%	99.97%
13, 34	99.34%	99.99%	100%
23, 16	100%	99.94%	99.99%
11, 13, 34	99.34%	99.99%	100%
23, 16, 12, 11	99.87%	100%	100%
29, 31	99.99%	99.72%	99.97%
23, 16, 12, 11, 13	99.82%	100%	100%
23, 16, 12, 11, 13, 34, 45	99.99%	99.83%	99.97%
AVERAGE	99.79%	99.87%	99.99%

Table 5.4: Success Rate

The results indicate a slight drop in averages, although the changes are minimal. Nonetheless, the overall efficiency remains quite high. The X coordinate, which previously had the greatest error, saw an increase to 0.21%. The Y coordinate, despite showing a difference of only 0.04%, also demonstrated a minor decrease in effectiveness. Conversely, the average value of θ experienced a slight increase, possibly due to more precise adjustments in angular calculations. These variations, particularly in the X coordinate, may be attributed to the transition from Edges and the increased complexity of processing multiple inputs.

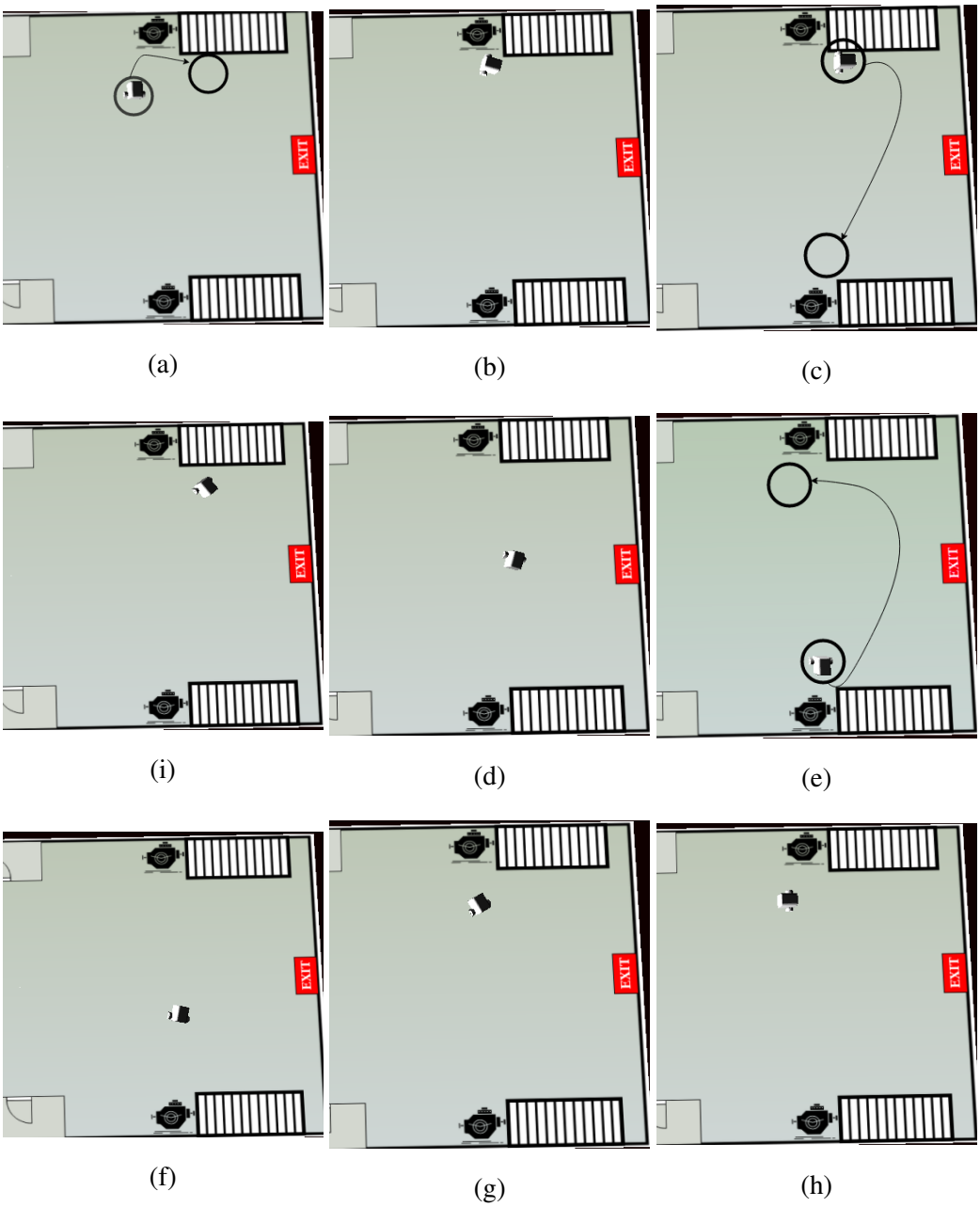
Despite these minor changes, the results remain extremely positive, with all values staying very close to 100%. The slight decrease in the average efficiency of the final coordinates does not significantly impact the robot's behavior, which remains consistent and reliable.

5.2 Scalability Tests

This section presents the scalability tests conducted for the simulation. Tests were performed with varying numbers of robots to evaluate the system's performance. The initial test involved a single robot. Subsequent tests incrementally increased the number of robots. A successful test case was defined as one in which all robots moved smoothly along their planned paths without disruptions.

One Robot Test

In this first test, the robot travels along the following Edges : {23, 16, 12}.



(a) (b) (c)

(i) (d) (e)

(f) (g) (h)

Figure 5.3: Scalability test with 1 Robot

Ten Robots' Test

In this test, 10 robots were placed in the scene, each with a different path. However, some of them had the same value as their end node, meaning some of the robots overlapped at the end. So, in

5.4, the evolution of each of the AMRs over time can be seen. First of all, it's important to note which edges each robot had:

- **Robot 1** = {64, 65};
- **Robot 2** = {100, 101, 102};
- **Robot 3** = {103,104};
- **Robot 4** = {23, 16};
- **Robot 5** = {55, 56};
- **Robot 6** = {22, 15};
- **Robot 7** = {11, 13};
- **Robot 8** = {53, 54};
- **Robot 9** = {51, 52};
- **Robot 10** = {57,99};

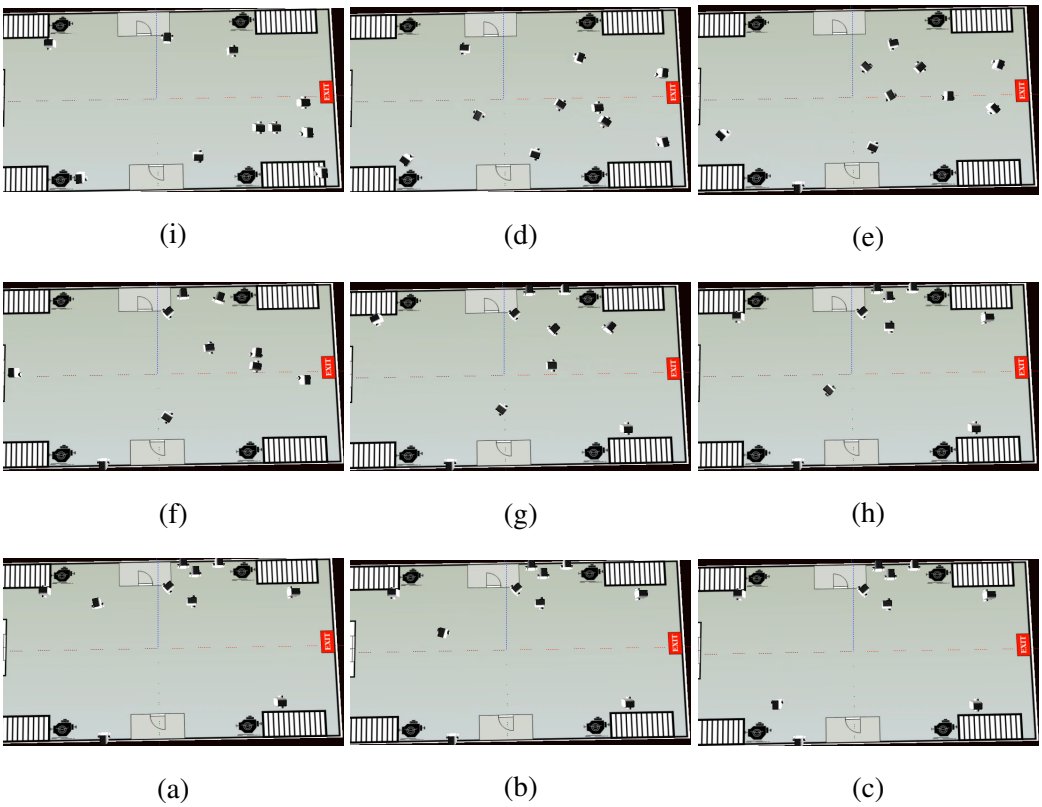
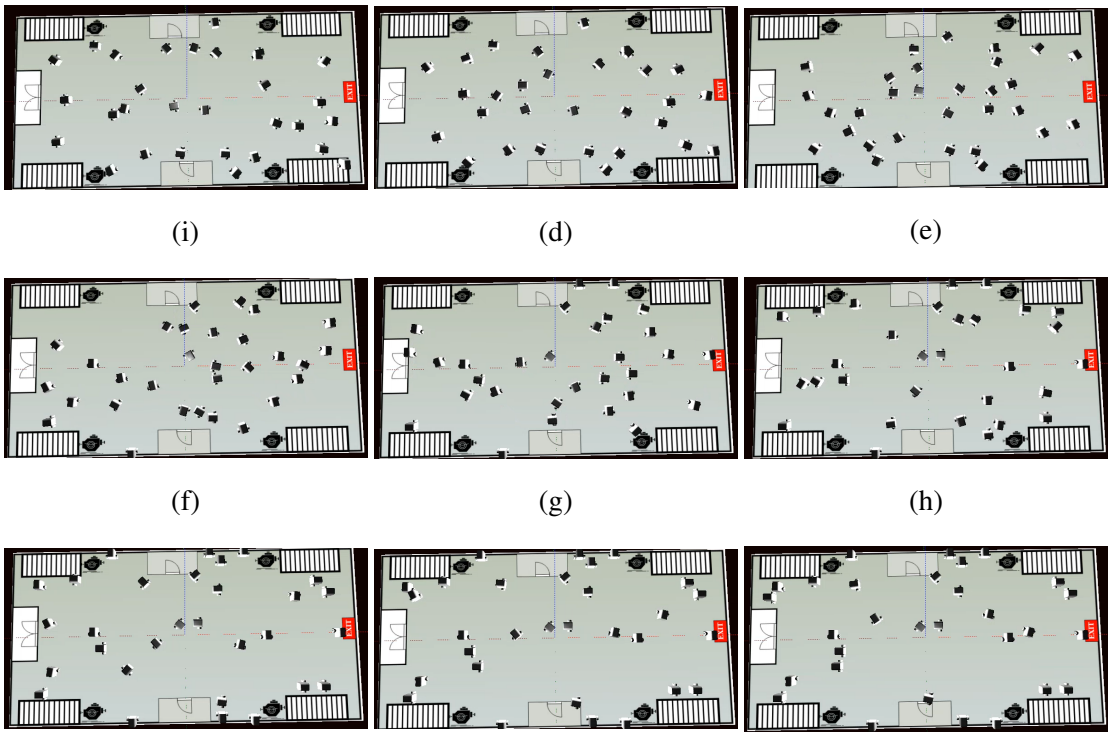


Figure 5.4: Scalability test with 10 Robots

Thirty Robots' Test

In this test, the number of robots was increased to thirty. By increasing the number of robots in the simulation to 20, the aim was to prove that the program was scalable, allowing the user to choose to simulate a fleet with a very high number of pieces of equipment. In 5.5, it's possible to see the evolution of the robots' positions over time, until they reach their final position. As in the simulation with 10 units, each one had a different path so that they could all be visualized simultaneously. Therefore, the routes associated with each of the robots were as follows:

- **Robot 1** = {64, 65};
- **Robot 2** = {100, 101, 102};
- **Robot 3** = {103,104};
- **Robot 4** = {23, 16};
- **Robot 5** = {55, 56};
- **Robot 6** = {22, 15};
- **Robot 7** = {11, 13};
- **Robot 8** = {53, 54};
- **Robot 9** = {51, 52};
- **Robot 10** = {57,99};
- **Robot 11** = {58, 59};
- **Robot 12**= {106};
- **Robot 13** = {62, 63};
- **Robot 14** = {105};
- **Robot 15** = {66, 67};
- **Robot 16** = {68};
- **Robot 17** = {69, 70};
- **Robot 18** = {71, 72};
- **Robot 19** = {73, 74};
- **Robot 20** = {76};
- **Robot 21** = {77,78};
- **Robot 22** = {79, 80};
- **Robot 23** = {81, 87};
- **Robot 24** = {82, 83};
- **Robot 25** = {84, 85};
- **Robot 26** = {86};
- **Robot 27** = {88, 89};
- **Robot 28** = {90};
- **Robot 29** = {98};
- **Robot 30** = {96, 97};



(i) (d) (e)

(f) (g) (h)

(a) (b) (c)

Figure 5.5: Scalability test with 30 Robots

5.2.1 Execution Time

This test aimed to see if, by changing the number of robots in the fleet, the execution time of a given would change. Tests were therefore carried out with 1, 10, 20, and 30 robots, with each robot traveling the same route as all the others. In Tables 5.5, 5.6, and 5.7, the results obtained can be seen. The simulations were repeated five times each and with three different routes. The routes are as follows: {23,16,12}, {84, 85} e {64, 65}.

First Trajectory

N Robots / N Test	1	10	20	30
1	15.67s	16.76s	31.60s	66.37s
2	15.64s	16.43s	31.40s	66.82s
3	15.42s	16.60s	31.66s	66.64s
4	15.58s	16.54s	31. 51s	66.71s
5	15.53s	16.58s	31.73s	66.27s
AVERAGE	15.57s	16.58s	31.58s	66.56s

Table 5.5: First Path for Execution Time Test

Second Trajectory

N Robots / N Test	1	10	20	30
1	10.54s	11.07s	19.70s	42.07s
2	10.28s	11.05s	19.44s	42.83s
3	10.14s	11.10s	20.03s	43.82s
4	10.35s	11.08s	19.88s	43.18s
5	10.23s	10.99s	19.76s	43.45s
AVERAGE	10.31s	11.06s	19.76s	43.07s

Table 5.6: Second Path for Execution Time Test

Third Trajectory

N Robots / N Test	1	10	20	30
1	10.02s	11.12s	20.14s	43.37s
2	10.07s	11.02s	20.12	43.63s
3	9.98s	11.13s	20.03s	43.52s
4	10.12s	11.08s	19.30s	43.49s
5	10.04s	11s	19.49s	43.45s
AVERAGE	10.05s	11.07s	19.82s	43.49s

Table 5.7: Third Path for Execution Time Test

The aforementioned results allow us to construct a graph demonstrating the evolution of execution time as the number of robots in the simulations increases. The resulting graphs are as follows:

First Trajectory

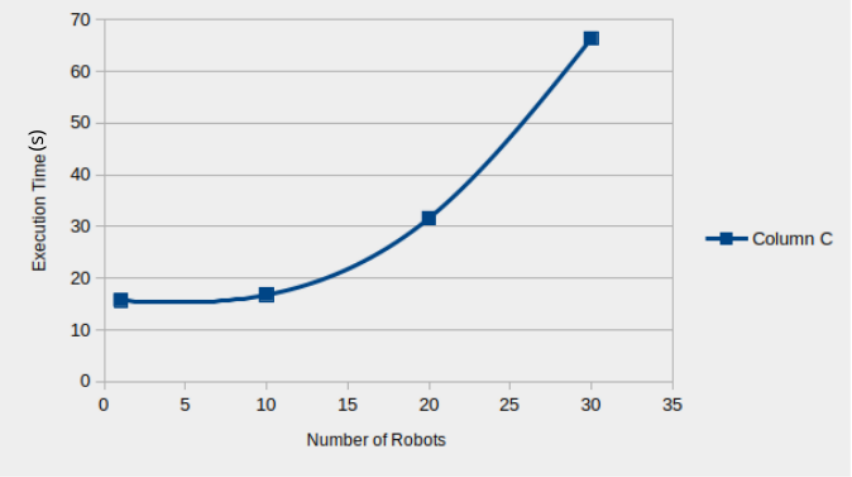


Figure 5.6: Evolution of Execution Time as the number of robots in the simulations increases: First Trajectory

Second Trajectory

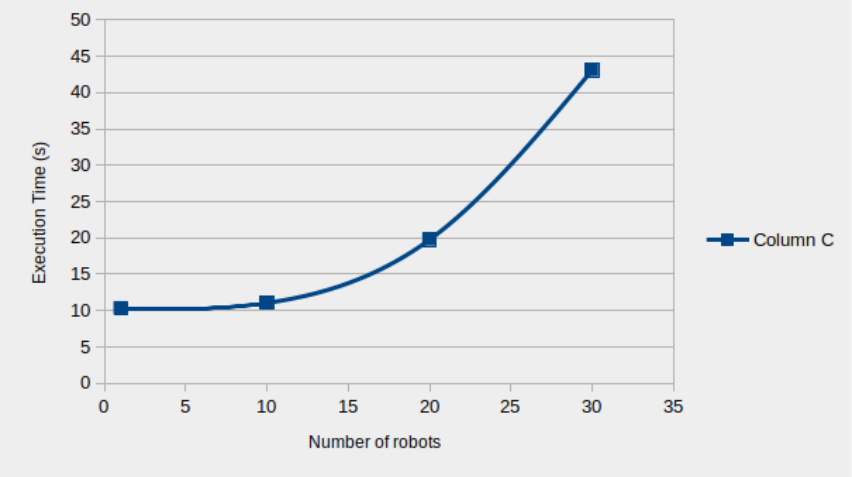


Figure 5.7: Evolution of Execution Time as the number of robots in the simulations increases: Second Trajectory

Third Trajectory

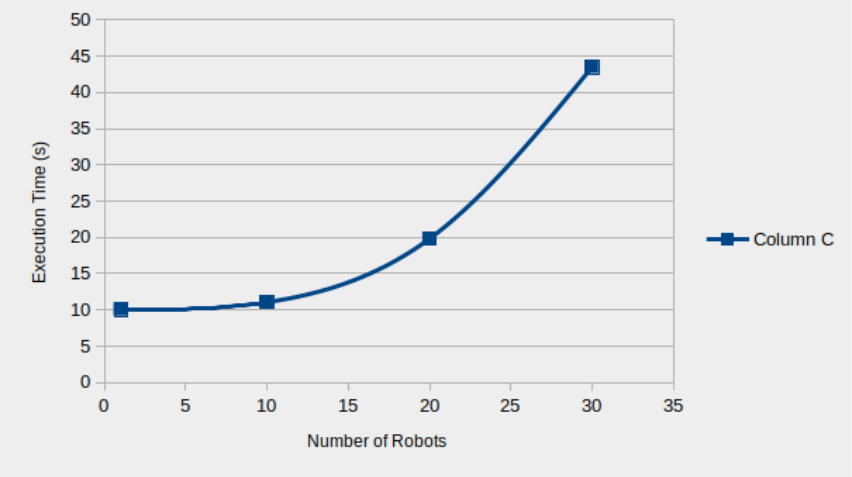


Figure 5.8: Evolution of Execution Time as the number of robots in the simulations increases: Third Trajectory

5.2.2 Overview of Scalability

These tests have demonstrated that the program is scalable, as it allows the selection of different numbers of robots and the simulation of them. However, it is evident that the visual quality and smoothness of movements are better maintained when simulating with a small number of robots. When simulating with thirty robots, the movements were slower and the frames per second decreased significantly. Nevertheless, each of the AMRs moved without encountering any issues, and at the end of the simulation, they were in the positions that would have been designated when their respective routes were associated with them. When simulating with 10 robots, the performance of the program saw minimal changes, and the operation was extremely similar to that of simulating just 1 robot.

The tests, in which the execution times were taken into account, prove what was said in the previous paragraph. It can be seen that between 1 and 10 robots, the difference in times is minimal, differing by a maximum of 1 second. When compared to simulations with 20 and 30 robots, it can be seen that the time taken for all robots to complete the trajectory increases significantly. To see whether or not these values depended on the trajectory, the same was repeated for two different trajectories, and the same results were obtained for each of them. The execution time appears to evolve exponentially, as evidenced by the trends observed in graphs 5.6, 5.7, and 5.8. This pattern suggests that as the input size or complexity increases, the time required for execution grows at an accelerating rate, rather than linearly. This exponential growth highlights the increasing computational demands of the process.

There are several reasons for this. One of them could be that for a larger number of robots, the amount of data is much greater, leading to a longer processing time. Additionally, the use of UDP communication can result in packet collisions, thus increasing the sending time. Another

factor could be the limitations of computing resources, such as the processor and RAM. However, to definitively determine the impact of computing resources, it would be necessary to test the visualizer on another computer, which wasn't possible.

5.3 Conclusions

After a careful analysis of all tests and their respective results, it is evident that the programmed algorithm was, in general, pretty successful. The simulations conducted with various fleet sizes demonstrated that the majority of results were positive, with trajectories being followed correctly. Notably, tests with large numbers of robots did not encounter any cases of deadlock or program bugs. However, it was observed that the algorithm's scalability posed minor issues. As the number of robots in the simulation increased, the execution time lengthened and the quality of the simulation diminished. Despite this, the robots' behavior remained visually efficient, mitigating the impact of these issues.

Chapter 6

Conclusions and Future Work

This chapter summarizes the results that were obtained during this dissertation and if the goals were all accomplished. It is also going to be presented the future work that can be done to improve all the algorithm that was made.

6.1 Results conclusions and Goals Accomplishments

The main goals of this dissertation was to make it possible to simulate a large-scale fleet of Autonomous Mobile Robots in a previously developed visualizer and, at the same time, test the Fleet Manager associated with the visualizer. To do the first, it would be necessary to create an algorithm capable of tracing the robots' trajectories and sending their positions. The project also required communication between the viewer and the program to be carried out using a UDP communication protocol. Another requirement was to use ROS to read the graphs, which provide the start and end points of each trajectory, and to receive instructions on which edges each robot should follow.

Therefore, with the results obtained, it is possible to draw some conclusions, paying particular attention to whether or not the objectives set have been met. Firstly, it can be said that one of the two main aims of the dissertation has been met and it is possible to carry out various simulations with as many robots as the user wants. As far as trajectory planning is concerned, it's possible to say that it was also a success in that the tests showed that the robots followed exactly the points calculated for the Bézier curve. Even so, sometimes, on paths where the difference between the coordinates was very low and the robot was subjected to significant rotation, its theta ended up changing in an abrupt way. It should also be noted that the UDP connection is also well implemented. The viewer correctly receives the positions of each robot and sends them back to the program, thus functioning as a client and server almost simultaneously. The second objective, unfortunately has not been met has the visualizer wasn't tested with the fleet manager that was developed by CRIIS.

In conclusion, it can be said that the implementation was successful and met one of the main objectives of the dissertation but that it wasn't fully conclude. The scalability of the program turns

out to be a focal point of the project, although the performance of the viewer decreases slightly when using fleets with a larger number of robots.

6.2 Future Work

In robotics, it is crucial to plan the behavior of robots while ensuring the safety of the environment and avoiding physical or material damage. For future work, it would be beneficial to develop a system that marks possible collisions between robots. Additionally, creating an interface to notify the user about the number of those possible collisions during simulations and which robots were involved could be valuable.

As mentioned earlier, the visualizer's performance worsens as the number of robots in the fleet increases. It would be worthwhile to address this issue by making changes to the program's methods. Testing the visualizer on a different computer should also be considered, as the drop in performance may be due to computing resources.something to consider since a possible reason for the drop in performance could be computing resources.

It is really important to test the fleet manager with the visualiser in the future, which was one of the objectives. It is crucial to be able to fully control the fleet while being able to visualise each of the robots and avoid some path errors such as collisions.

References

[1] Ilya Afanasyev, Artur Sagitov, and Evgeni Magid. Ros-based slam for a gazebo-simulated mobile robot in image-based 3d model of indoor environment. In Sebastiano Battiato, Jacques Blanc-Talon, Giovanni Gallo, Wilfried Philips, Dan Popescu, and Paul Scheunders, editors, *Advanced Concepts for Intelligent Vision Systems*, pages 273–283, Cham, 2015. Springer International Publishing.

[2] Mehrnoosh Askarpour, Matteo Rossi, and Omer Tiryakiler. Co-simulation of human-robot collaboration: From temporal logic to 3d simulation. In *arXiv*, 2020.

[3] Syeda Tanjila Atik, Akshar Shravan Chavan, Daniel Grosu, and Marco Brocanelli. A maintenance-aware approach for sustainable autonomous mobile robot fleet management. *IEEE Transactions on Mobile Computing*, 23:7394–7407, 2024.

[4] Lucas Favi Bocca, Jônatas Boas Leite, and Suely Cunha Amaro Mantovani. Simulator of mobile robots controlled by artificial neural networks to learning courses in robotics. In *2020 XIV Technologies Applied to Electronics Teaching Conference (TAEE)*, pages 1–7, 2020.

[5] Micael S. Couceiro, Patricia A. Vargas, and Rui P. Rocha. Bridging the reality gap between the webots simulator and e-puck robots. *Robotics and Autonomous Systems*, 62(10):1549–1567, 2014.

[6] Pedro Luís Cerqueira Gomes da Costa. *Planeamento Cooperativo de Tarefas e Trajectórias em Múltiplos Robôs*. PhD thesis, Faculdade de Engenharia da Universidade do Porto, 2011.

[7] Gilberto Echeverria, Nicolas Lassabe, Arnaud Degroote, and Severin Lemaignan. Modular open robots simulation engine: Morse. In *2011 IEEE International Conference on Robotics and Automation Shanghai International Conference Center*, 2011.

[8] Andrew Farley, Jie Wang, and Joshua A. Marshall. How to pick a mobile robot simulator: A quantitative comparison of coppeliasim, gazebo, morse and webots with a focus on accuracy of motion. *Simulation Modelling Practice and Theory*, 120:102629, 2022.

[9] Paul M. Fitzpatrick, Elena Ceseracciu, Daniele E. Domenichelli, Ali Paikan, Giorgio Metta, and Lorenzo Natale. A middle way for robotics middleware. 2014.

[10] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *In Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, pages 317–323, 2003.

[11] Sebastien Gougeaud, Soraya Zertal, Jacques-Charles Lafoucriere, and Philippe Deniel. Using zeromq as communication/synchronization mechanisms for io requests simulation. In *2017 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, pages 1–8, 2017.

[12] Albert S. Huang, Edwin Olson, and David C. Moore. Lcm: Lightweight communications and marshalling. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4057–4062, 2010.

[13] Lentin Joseph. *Robot Operating System (ROS) for Absolute Beginners*. APRESS, 2018.

[14] Prabhjot Kaur, Zichuan Liu, and Weisong Shi. Simulators for mobile social robots: State-of-the-art and challenges. In *2022 Fifth International Conference on Connected and Autonomous Driving (MetroCAD)*, pages 47–56, 2022.

[15] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Proceedings 01 2004 IEEE/RSJ International Conference on Intelligent Robots and System*, 2004.

[16] James Kramer and Matthias Scheutz. Development environments for autonomous mobile robots: A survey. *Autonomous Robots*, 22:101–132, 2007.

[17] J Lauener and Wojciech Sliwinski. How to design implement a modern communication middleware based on zeromq. 10 2017.

[18] Jun Won Lim, Sanghoon Lee, Il Hong Suh, and Kyung Jin Kim. Development of a roboid component for player/stage robot simulator. In *2009 IEEE International Conference on Network Infrastructure and Digital Content*, pages 968–972, 2009.

[19] Jovina Seau Ling Leong, Kenneth Tze Kin Teo, and Hou Pin Yoong. Four wheeled mobile robots: A review. In *2022 IEEE International Conference on Artificial Intelligence in Engineering and Technology (IICAIET)*, pages 1–6, 2022.

[20] Angel Madridano, Abdulla Al-Kaff, David Martín, and Arturo de la Escalera. Trajectory planning for multi-robot systems: Methods and applications. *Expert Systems with Applications*, 2021.

[21] Alexei A. Makarenko, Stefan B. Williams, Frederic Bourgault, and Hugh F. Durrant-Whyte. An experiment in integrated exploration. In *In Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 534–539, 2002.

[22] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. Yarp: Yet another robot platform. *International Journal of Advanced Robotic Systems*, 3, 03 2006.

[23] Olivier Michel. Cyberbotics ltd. webots™: Professional mobile robot simulation. In *International Journal of Advanced Robotic Systems*, volume 1, 2004.

[24] Ivica Mitrovic and Kerstin Dautenhahn. Social attitudes: Investigations with agent simulations using webots. *Journal of Artificial Societies and Social Simulation*, 6(4), 2003.

[25] Farzan M. Noori, David Portugal, Rui P. Rocha, and Micael S. Couceiro. On 3d simulators for multi-robot systems in ros: Morse or gazebo? In *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, pages 19–24, 2017.

[26] Lynne E. Parker, Daniela Rus, and Gaurav S. Sukhatme. *Multiple Mobile Robot Systems*, pages 1335–1384. Springer International Publishing, Cham, 2016.

[27] Diogo Pereira. Multi-agv coordination for heterogeneous robots. Technical report, Faculdade de Engenharia da Universidade do Porto, 2022.

[28] Diogo Pereira, Diogo Matos, Paulo Rebelo, Fillipe Ribeiro, Pedro Costa, and José Lima. Multi-robot coordination for a heterogeneous fleet of robots. In Danilo Tardioli, Vicente Matellán, Guillermo Heredia, Manuel F. Silva, and Lino Marques, editors, *ROBOT2022: Fifth Iberian Robotics Conference*, pages 229–240, Cham, 2023. Springer International Publishing.

[29] Ricardo Silva Peres, Andre Dionisio Rocha, and Jose Barata. Dynamic simulation for mas-based data acquisition and pre-processing in manufacturing using v-rep. In Luis M. Camarinha-Matos, Mafalda Parreira-Rocha, and Javaneh Ramezani, editors, *Technological Innovation for Smart Systems*, Cham, 2017. Springer International Publishing.

[30] Morgan Quigley, Brian Gerkey, Ken Conley, and Josh Faust. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[31] SMP Robotics. Mobile robot technical maintenance service. *Ieeeeeeee*, 2023.

[32] 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.

[33] Erol Sahin. Swarm robotics: From sources of inspiration to domains of application. In *Swarm Robotics: From Sources of Inspiration to Domains of Application*, 2004. https://doi.org/10.1007/978-3-540-30552-1_2.

[34] Joana Santos, Pedro Costa, Luís F. Rocha, A. Paulo Moreira, and Germano Veiga. Time enhanced a*: Towards the development of a new approach for multi-robot coordination. In *2015 IEEE International Conference on Industrial Technology (ICIT)*, pages 3314–3319, 2015.

[35] Tobias Simon, André Puschmann, Shah Nawaz Khan, and Andreas Mitschele-Thiel. A lightweight message-based inter-component communication infrastructure. In *2013 Fifth International Conference on Computational Intelligence, Communication Systems and Networks*, pages 145–152, 2013.

[36] E. A. Sisbot, L. F. Marin-Urias, R. Alami, and T. Simeon. A human aware mobile robot motion planner. *IEEE Trans. Robot*, 2002.

[37] Kenta Takaya, Toshinori Asai, Valeri Kroumov, and Florentin Smarandache. Simulation environment for mobile robots testing using ros and gazebo. In *2016 20th International Conference on System Theory, Control and Computing (ICSTCC)*, pages 96–101, 2016.

[38] Niez Yuldashev, Alexandra Dobrokvashina, and Roman Lavrenov. On sensor modeling in gazebo simulator. In *International Conference on Artificial Life and Robotics (ICAROB2024)*, 2024.

Appendix A

Videos

1 - https://www.youtube.com/watch?v=pTmojInyzak