

3D Mapping for Search and Rescue Operations

Luís Filipe Gonçalves Lemos¹

A Thesis presented for the degree of Master in Informatics

Supervisor: Luís Paulo Reis^{1,2}

¹DEI-FEUP – Departamento de Engenharia Informática da Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias, s/n, 4200-465 Porto PORTUGAL

²LIACC – Laboratório de Inteligência Artificial e Ciência de Computadores da Universidade do Porto (NIAD&R – Núcleo de Inteligência Artificial Distribuída e Robótica)
{luis.lemos, lpreis}@fe.up.pt



Universidade do Porto

Faculdade de Engenharia
FEUP

December 2008

Abstract

Robots have been a part of our lives for many years now, mainly in industry, where they perform repetitive tasks, but many more distinct uses can be thought for them. Autonomous robots, a subset of the robotics universe, have many applications but one of the most interesting of them might be their application to search and rescue in disaster environments.

Accurate construction of the map of a disaster area is the most critical step in allowing an autonomous robot to perform any task in an initially unknown scenario. Without an accurate map, the robot will not be able to successfully execute some elementary tasks like navigating. So, it cannot be expected to perform more complicated tasks like trying to identify victims based on the mapping. Some of the equipments currently used to produce maps are expensive, big and heavy while others may require a substantial processing overhead to find distances.

The work presented throughout this document describes a simple 3D mapping methodology to achieve accurate 3D maps of disaster scenarios using a robot equipped with a combination of a few inexpensive sensors and computationally light algorithms. The readings of those sensors are analysed to extract some information about the environment and, for increased accuracy, are combined using a sensor fusion methodology relying on probabilistic models based on a Bayesian method. To prove that the Bayesian method provides good results with few measurements of the same point, another more crude method of fusing the sensors measurements was implemented in which the fusion is performed by simply applying the mean of the measured values.

The prototype of the robot and corresponding algorithms, which are an integral part of the 3D mapping methodology, were tested in a simulated environment using the USARSim platform and a modified cut-down version of the P2AT simulated robot. Since the robot was simulated, the software robotic agent part, the client part of the simulator, also emulates the hardware architecture of the robot, namely the microprocessor, RAM and peripherals, allowing to demonstrate that the algorithms used are light. The demonstration is performed by finding the minimum required frequency of operation of the emulated architecture, for the computationally worst case scenario, and by performing a statistical comparison of the measures obtained with this methodology with reference measures. The software robotic agent's architecture also allows to seamlessly get the algorithms to work in a real robot with minimal modifications.

The results obtained show that it is possible to achieve accurate mapping with the simple setup developed in this work. It is also shown that the algorithms used to generate the 3D map are indeed light and can be executed in a fairly simple microprocessor running at few MHz. The results enable to conclude that the sensor fusion methodology and the robotic platform presented may be a promising solution for future inexpensive search and rescue robots.

Keywords: Robotics, Simulation, USARSim, 3D Mapping, Search and Rescue, Sensor Fusion, Bayesian method, Cloud Points.

Acknowledgements

I would like to thank all people who helped me with this work and without whom it would not be possible.

Particularly, I want to thank my supervisor, Prof. Luís Paulo Reis, for his patience and help to bring this thesis to a successful completion.

Contents

Abstract.....	3
Acknowledgements.....	5
Figures list.....	8
Tables list.....	10
Formulas List.....	11
1 Introduction.....	12
1.1 Objectives.....	13
1.2 Proposal.....	14
1.3 Document Organization.....	15
1.4 Summary.....	16
2 3D Mapping Sensor Technology.....	17
2.1 State of the Art Range Sensors.....	17
2.1.1 SICK LMS200.....	17
2.1.2 HOKUYO URG-04LX.....	19
2.1.3 Mesa Imaging SR4000.....	20
2.2 State of the Art in Vision Sensors.....	21
2.2.1 Stereo Pair of Cameras.....	21
2.2.2 Omnidirectional Camera.....	24
2.2.3 Fish-eye Camera Lens.....	25
2.3 Additional Range Sensors.....	27
2.4 Summary.....	28
3 Project and Implementation.....	29
3.1 Mapping Strategy.....	29
3.2 Constraints.....	30
3.3 Robot Hardware Architecture.....	31
3.3.1 Desired Architecture.....	31
3.3.2 Implemented Architecture for Testing.....	31
3.3.3 Implementation of the Testing Robot.....	32
3.4 Sensors and Sensor Models.....	35
3.4.1 In USARSim.....	35
3.4.2. In Robotic Software Agent.....	36
3.5 Robot Software Architecture.....	38
3.5.1 Hardware Emulation.....	38
3.5.2 Software Emulation.....	42
3.6 Software Robotic Agent.....	45
3.6.1 Main Program.....	45
3.6.2 Building the Cone.....	48
3.6.3 Managing Points.....	49
3.6.4 Speeding Things Up.....	53
3.6.1 Configuring.....	54
3.7 Map Viewer.....	56

3.8 Post-Process for Result Analysis.....	60
3.9 Calculating Results Accuracy.....	61
3.9.1 Statistical Comparison.....	61
3.9.2 Reference Point Clouds.....	62
3.10 Summary.....	65
4 Results.....	66
4.1 Perfect sensors.....	66
4.2 Non-ideal sensors.....	68
4.3 Results Analysis.....	72
4.4 Minimum Working Frequency.....	73
4.5 Summary.....	74
5 Conclusions and Future Work.....	75
References.....	77
Annexes.....	79
Annex A Detailed Objectives.....	80
Annex B CPU Instructions.....	82
Annex C Instructions Equivalence.....	85
Annex D Map Viewer Options.....	89
Annex E Simulation Environment.....	91
Unreal Tournament 2004.....	91
USARSim.....	92
Installation.....	94
Starting the Simulator.....	95

Figures list

Figure 1. - SICK LMS200 Laser Measurement Sensor.....	13
Figure 2. - Scanning area of the SICK LMS200 (the blue shapes) indicated by the grey lines.	17
Figure 3. - Kurt3D with the SICK Laser Measurement Sensor.....	18
Figure 4. - Pygmalion.....	18
Figure 5. - 3D map of the yellow arena returned by Kurt3D (left) and a corresponding view from UT2004 (right).....	19
Figure 6. - HOKUYO URG-04LX.....	19
Figure 7. - Mesa Imaging SR4000 sensor.....	20
Figure 8. - An office chair (left) and as viewed by a SR4000 sensor (right).....	21
Figure 9. - View from a stereo pair of cameras.....	22
Figure 10. - Object match (in red lining) on a stereo pair of images.....	22
Figure 11. - Position of an object (in green) inside a stereo pair of images.....	23
Figure 12. - Office reconstructed after the real one had been captured by a stereo pair of cameras and processing had occurred.....	23
Figure 13. - An omnidirectional camera sensor.....	24
Figure 14. - Illustration of the operation of an omnidirectional camera.....	24
Figure 15. - View of an office from an omnidirectional camera.....	25
Figure 16. - Fish-eye lenses.....	26
Figure 17. - Schematic of a fish-eye camera setup.....	26
Figure 18. - View from a fish-eye lenses.....	27
Figure 19. - Sonar sensor (left) and Infra-red sensor (right).....	28
Figure 20. - Schematic of the desired implementation of the robot.....	31
Figure 21. - Image of the original P2AT in USARSim, the robot modified to be used for testing.....	32
Figure 22. - The modified P2AT robot and its sensors relative placement.....	32
Figure 23. - Representation in 2D (left) and 3D (right) of the sensor model of uncertainty removed from (10).....	36
Figure 24. - Example of a sensor model for the error.....	37
Figure 25. - Emulated hardware architecture.....	39
Figure 26. - Example of two arithmetic instruction type.....	40
Figure 27. - Usage example of an if instruction of the conditional instruction type.....	41
Figure 28. - Usage example of a while instruction of the conditional instruction type.....	42
Figure 29. - Usage example of the memory addressing instruction type.....	42
Figure 30. - Example of a linked list implemented using two arrays in a C like language.....	44
Figure 31. - Example in C++ of the sum of the elements of a linked list implemented using two arrays.....	44
Figure 32. - Example in emulated CPU assembly instructions of the sum of the elements of a linked list implemented using two arrays.....	44
Figure 33. - Main program thread execution block diagram.....	46
Figure 34. - Cone generation function execution block diagram.....	48

Figure 35. - Structure of a two-way linked list.....	49
Figure 36. - Structure of the indexed ordered two-way linked list.....	50
Figure 37. - Example of an indexed ordered two-way linked list implemented in arrays.....	50
Figure 38. - Binary search algorithm block diagram.....	51
Figure 39. - Point search function execution block diagram.....	52
Figure 40. - Pre-defined position 1 of the robot.....	55
Figure 41. - Pre-defined position 2 of the robot.....	56
Figure 42. - Pre-defined position 3 of the robot.....	56
Figure 43. - Processing programming graphical interface with an example.....	57
Figure 44. - Map Viewer interface with a sonar cone printed. The left one shows the help menu and the right one the values of the internal state.....	58
Figure 45. - Block diagram of the post-processing algorithm.....	61
Figure 46. - Map generated from measurements of the IR sensor.....	63
Figure 47. - Map generated from measurements of the sonar sensor.....	64
Figure 48. - Reference set of points for location 1.....	65
Figure 49. - Reference set of points for position 2 of the robot.....	67
Figure 50. - Reference set of points for position 3 of the robot.....	68
Figure 51. - IR cloud points for position 1 obtained using the Bayesian method.....	69
Figure 52. - IR cloud points for position 2 obtained using the Bayesian method.....	70
Figure 53. - IR cloud points for position 3 obtained using the Bayesian method.....	71
Figure 54. - Detail of a cone with accessibility on and colour scheme set to four colour.....	89
Figure 55. - Detail of a cone with accessibility off and colour scheme set to four colour.....	89
Figure 56. - Detail of a cone with accessibility on and colour scheme set to grey scale.....	90
Figure 57. - Detail of a cone cuted using the window function with accessibility on and colour scheme set to four colour.....	90
Figure 58. - Detail of a cone filtered using the probability function with accessibility on and colour scheme set to grey scale.....	90
Figure 59. - UT2004 client-server architecture for multi player games (left) and single player games against the AI bots (right).....	91
Figure 60. - UT2004 screen shots.....	92
Figure 61. - USARSim architecture. The boxes in grey belong to UT2004.....	93
Figure 62. - USARSim's coordinate system.....	93
Figure 63. - Compile shortcut of USARSim under Windows® OS.....	95
Figure 64. - Running USARSim in server mode.....	95
Figure 65. - Image Server interface.....	96

Tables list

Table 1. - Some reasons why mapping used to be performed in 2D.....	12
Table 2. - List of objectives.....	14
Table 3. - List of tasks performed by the robot's artificial intelligence.....	14
Table 4. - Organization of this document.....	16
Table 5. - List of some desirable characteristics in a search and rescue robot.....	29
Table 6. - Constraints for the robot and the reasons why they are required.....	30
Table 7. - Text replacement in "Mapper.uc" robot definition file.....	33
Table 8. - P2AT robot configuration.....	33
Table 9. - Mapper robot configuration.....	34
Table 10. - List of sensors, their properties, property descriptions and the values they assume for this work.....	35
Table 11. - Different types of instructions of the emulated CPU.....	40
Table 12. - Some examples of conversions between the emulated CPU instructions and C++.....	43
Table 13. - Detailed list of tasks executed by the main thread of the robotic agent.....	47
Table 14. - Detailed list of tasks executed by the cone generation function.....	49
Table 15. - Detailed binary search algorithm.....	51
Table 16. - Detailed list of tasks performed by the point search function.....	53
Table 17. - Configuration parameters of the software robotic agent.....	54
Table 18. - List of tasks executed by the "setup" function of the Map Viewer.....	57
Table 19. - List of tasks executed by the "draw" function of the Map Viewer.....	58
Table 20. - Functionalities of the Map Viewer and their respective descriptions.....	59
Table 21. - Statistics for position 1.....	72
Table 22. - Statistics for position 2.....	72
Table 23. - Statistics for position 3.....	72
Table 24. - Difference between the values of Dumb and Bayesian methods.....	73
Table 25. - Worst case scenario conditions.....	73
Table 26. - List of future work.....	75
Table 27. - Detailed list of objectives, sub-objectives and their corresponding descriptions...	80
Table 28. - List of instructions of the emulated CPU, their syntax, usage and descriptions...	82
Table 29. - List of equivalence between the emulated CPU instructions and the C++ instructions used to program.....	85
Table 30. - Main steps to install the USARSim platform.....	94

Formulas List

(1) - Equation for the probability of Region I in the Bayesian method.....	37
(2) - Equation for the probability of Region II in the Bayesian method.....	37
(3) - Equation for the probability of Region III in the Bayesian method.....	37
(4) - Equation for the probability update in the Bayesian method.....	38
(5) - Equation of the weighted mean.....	60
(6) - Equation used to calculate the weight attributed to a point.....	60

1 Introduction

In an age where robots are each day more part of our lives, there are tasks where the robots are regarded as having more importance than others. Some of those tasks are the ones that can cause injuries to humans, like performing repetitive tasks on an assembly plant, or where there is considerable risk of death, like in disarming bombs. Whatever is the purpose of an existing specific robot, the focus of this work will be on ground robots for search and rescue.

In search and rescue, specially when performed by autonomous robots, the quality of the map used by the robot is of most importance. Not only does it serve the purpose of navigation, but can also be used to find victims and aid the rescue teams orientate themselves in the disaster area, among other things. If the map generated by the robot has poor detail or inaccurate measures, it will most likely increase the difficulty of the work of whomever (human or robot) is using it, whether it is being used to navigate around the scenario or find victims. Indeed, even a small error in a measure can propagate in several different manners and can, as an example, affect the algorithm used by the robot to locate itself in the scenario, making it follow the wrong path and, ultimately, getting lost and unable to fulfil its mission.

Initially, practically all maps were 2D for some reason or another. Some of the reasons for it are presented in Table 1. In fact, countless algorithms exist for planning in 2D. In some current applications, there are still 2D maps being used because the height is not necessary, like in an automated vacuum cleaner.

Table 1. - Some reasons why mapping used to be performed in 2D.

Reasons for 2D mapping
The scenarios of the disaster simulation were intentionally not too complex, most of them consisting of a building with all walls intact and some rumble spread around, like the yellow arena in USARSim. That is, most simulated scenarios where kept simple so that the robot would not have many difficulties succeeding.
It is easier to write algorithms and code for navigation if the height is disregarded.
Robots are simpler and cheaper if they don't have to climb or move on irregular surfaces.
The robot does not require a processing unit as powerful as it would need if it was mapping in 3D, neither would it require as much memory because there are considerably less points to process.
Processors were less powerful than they are today.

For the majority of the real life applications, namely disaster areas, 2D might not hold enough information neither for the rescue crew nor the robot itself. As an example, if a robot performs measurements of its sensors in the three dimensions of space but translates them into a 2D flat map, without height information, before planning its route, it is possible that there may be a passage between debris that doesn't show on the map as a viable passage due to the robot cataloguing it as being too narrow for him to fit through. This is one of the main reasons

why 3D mapping was used in this work. Other reason is that, with 3D mapping, potential victims can be identified right from the mapping with little processing overhead involved. In reality, most of the algorithms used in 3D mapping, planning and navigation are adapted versions of algorithms created for 2D.

Nowadays, there has been an increased usage of 3D maps but some of them are produced using big, heavy, energy consuming sensors like the SICK LMS200 Laser Measurement Sensor (Figure 1) (1) when compared to a simple IR sensor. Other sensors, like cameras, may require considerably more powerful processing units to process the measurements for distances in real time. These characteristics impose restrictions on the robot's size, autonomy and reach. Thus, in this work, there was an attempt to reduce the size of the robot and increase its autonomy and reach by using simple small sized sensors.



Figure 1. - SICK LMS200 Laser Measurement Sensor.

The disaster areas are, in reality, dynamic places and, to have maps as consistent with the reality as possible, the mapping should be done in 4D (the 3 coordinates of space plus time). This would allow the detection and tracking of moving objects which could be later identified as conscientious victims moving. This could significantly decrease the amount of time required for their rescuing. It could also increase the accuracy of the mapping because those moving objects could be removed from the generated map and reduce the probability of false static objects appearing. Due to the fact that 4D is too complex for this work, the maps will be just in 3D.

To ease the implementation of this work, it was decided to implement it under a simulated environment named USARSim. A description of USARSim, how it works and how install and work with it is presented in Annex E.

The remainder of this chapter presents a compact list of the objectives, states the purpose of this work and describes the organization of this document.

1.1 Objectives

The main purpose of this work is to provide a simple 3D mapping methodology, including a prototype of a robotic platform equipped with small sensors, which is able, at the same time, to produce rigorous 3D maps of a disaster scenario and use computationally light algorithms. To achieve this, several objectives, presented in Table 2, can be identified. A more extensive and detailed list of the objectives and their corresponding sub-objectives is presented in Annex A.

Table 2. - List of objectives.

Objective number	Description
1	Gather information about the USARSim simulation platform experimenting pre-built robots, to better understand the simulator and its potentialities.
2	Create a complete robot that conforms to the intended characteristics defined for it in this work.
3	Define a software architecture suitable for implementing the software of the robotic agent and implement this architecture in the context of the robot created, developing a complete autonomous robot.
4	Study distinct sensor fusion techniques and implement a suitable sensor fusion technique for the problem of mapping a rescue arena using inexpensive sensors.
5	Perform tests with several combinations of sensor models and sensor fusion algorithms, concluding which technique is better for each specific mapping problem.

1.2 Proposal

As said before, in this document, a 3D mapping methodology is proposed. It includes the definition, implementation and testing of a robotic platform. The 3D mapping methodology must be able to generate accurate 3D maps of a disaster area and will be described in detail later in this document, explaining all the motives that led to that specific choice, which in turn impose restrictions on the chosen robotic platform. The robotic platform must use few simple sensors and the lightest computational algorithms possible. To demonstrate it, a robot will be created in the simulation platform USARSim. Also, a software robotic agent will emulate the robot's hardware architecture, implement the robot's artificial intelligence and control the robot's body in USARSim.

In this work, the artificial intelligence of the robot will be limited to algorithms that perform the tasks listed in Table 3.

Table 3. - List of tasks performed by the robot's artificial intelligence.

Calculate what position the PTZ must have for each sensor measure and order it to move.
Translate each sensor measure into a cone of points, in case of the sonar, or into a line of points, in case of the IR sensor, and calculate the probability of occupancy of each of the points.
Perform changes of referential of the points calculated from robot part referential to absolute world referential.
Perform sensor fusion.
Save or update points with the most up to date probability of occupancy.
Manage points in memory.

The algorithms that control the robot's body will also be limited to those that control the PTZ. Besides that, the robot can be moved from its original position on the scenario by intervention of a human acting on the robot's wheels by means of the robot's interface program. This was planned that way because there are already many planning and navigation algorithms that could be easily implemented into the robot and, as such, there was no need to further complicate this work.

The map generated by the robot will be a 3D point cloud and will provide a probability of occupancy for each of its points. To maximize accuracy, the probability of occupancy of each point will be a combination of the measures of the sensors using a sensor fusion methodology. There will be two sensor fusion methodologies used. In the first methodology, a simple dumb method will be used in which the points are only marked as occupied or free, respectively 100% or 0%, and the update of an existing point is the mean value of the old value with the new one. In the other sensor fusion methodology used, for each new measurement of a sensor, the points on the map will be updated using probabilistic models based in the Bayesian method (10) discussed later in this document.

In order to prove the accuracy of the methodologies, the maps generated using realistic sensors' models will have the probabilities of all its points statistically compared against maps generated using ideal sensors' models by calculating the absolute error mean and the standard deviation of the absolute error. The most accurate map will have all these values closer to zero since the difference between probabilities for each point will be smaller.

1.3 Document Organization

Before continuing with the description of the matter of this document, a brief description of its organization will be presented.

This document is partitioned into several chapters which are enumerated in Table 4. Most of the chapters correspond to a phase of the work performed during the course of this Master Thesis and are temporally arranged in this document.

Table 4. - Organization of this document.

Chapter	Description
Introduction	The current chapter. Introductory chapter that describes the organization of the document, enumerates the objectives and introduces the work performed.
3D Mapping Sensor Technology	Chapter dedicated to the description of the current state of the art in 3D mapping of environments, particularly those performed in disaster areas.
Project and Implementation	This is one of the most important chapters of this document since it corresponds to the biggest part of the work. It describes the mapping strategy used, presents the constraints that the chosen strategy imposed on the work, describes the implementation of the robot's body in the simulator and its sensors. Also presents the software robot agent as well as the software that allows to display the map built by the robot.
Results	Presents several experiments performed in this work, presents their corresponding results and compares all the results to draw conclusions.
Conclusions and Future Work	The conclusions of the work.
References	References to documentation consulted throughout the duration of this work.
Annexes	Extra information not relevant for this document but relevant to the work or topics that would be too extensive to fit in this document.

1.4 Summary

This chapter presented an introduction to the work realized in the thesis, described the work proposal, listed the main objectives to fulfil and described the organization of this document.

2 3D Mapping Sensor Technology

In a work discussing a topic often associated with the term hi-tech such as search and rescue robotics is, it would make no sense not to spend some time describing the state of the art technology related to the specific topic of this work: mapping. So, in the following sub-chapters of this chapter, the state of the art technology in sensors used in robots for the purpose of mapping will be presented.

2.1 State of the Art Range Sensors

2.1.1 SICK LMS200

Nowadays, robots exist that are capable of generating maps in 3D, and many more are being created since many researchers are turning to 3D. Most robots use the well known, and still in use, SICK LMS200 presented in Figure 1, or similar sensor, to perform measurements and produce a 3D map of the world. The SICK LMS200 is a laser range sensor that performs measurements in an 180° angle in a horizontal plane parallel to its base as presented in Figure 2. To enable measurements to be performed in the third dimension, the sensor must be mounted in a tilt unit. One of the best known robots that uses this sensor for 3D mapping is Kurt3D (14), presented in Figure 3, but there are others like the Pygmalion, presented in Figure 4.

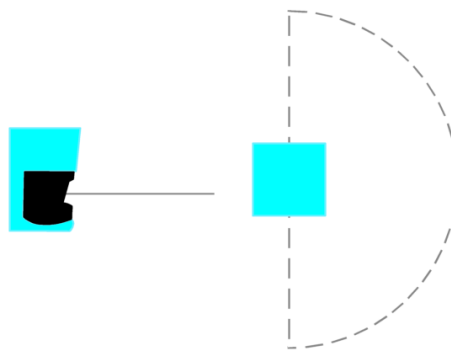


Figure 2. - Scanning area of the SICK LMS200 (the blue shapes) indicated by the grey lines.



Figure 3. - Kurt3D with the SICK Laser Measurement Sensor.

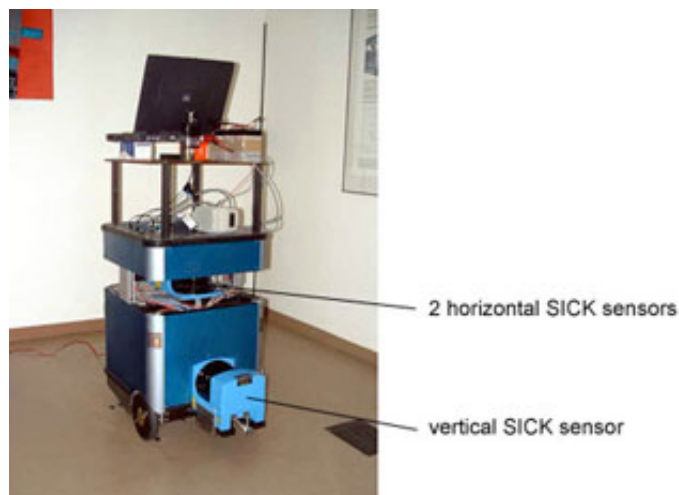


Figure 4. - Pygmalion.

Kurt3D, as can be seen in Figure 3, is a big and heavy robot, although not as much as the Pygmalion. Kurt3D is 45cm long, 33cm wide, 47cm height and 22.6Kg in weight. The SICK LMS200 alone measures 15.6cm long, 15.5cm wide, 21cm height and weights 4.5Kg. In fact, most of the robots that perform 3D mapping possess similar characteristics, meaning that most have big dimensions and weight. Although the SICK LMS200 is big and heavy, it has three great advantages over most of the other sensors. It has very good precision (1cm in range and 0.25° in angle), very long range (80m maximum) and is considerably fast at returning measurements (few milliseconds). Its precision can be roughly appreciated by comparing the map on the left side of Figure 5, generated from measurements taken by the sensor, with the screen shot on the right side of Figure 5, which is a view of the same scenario, the NIST yellow arena, in UT2004 more or less from the same angle.

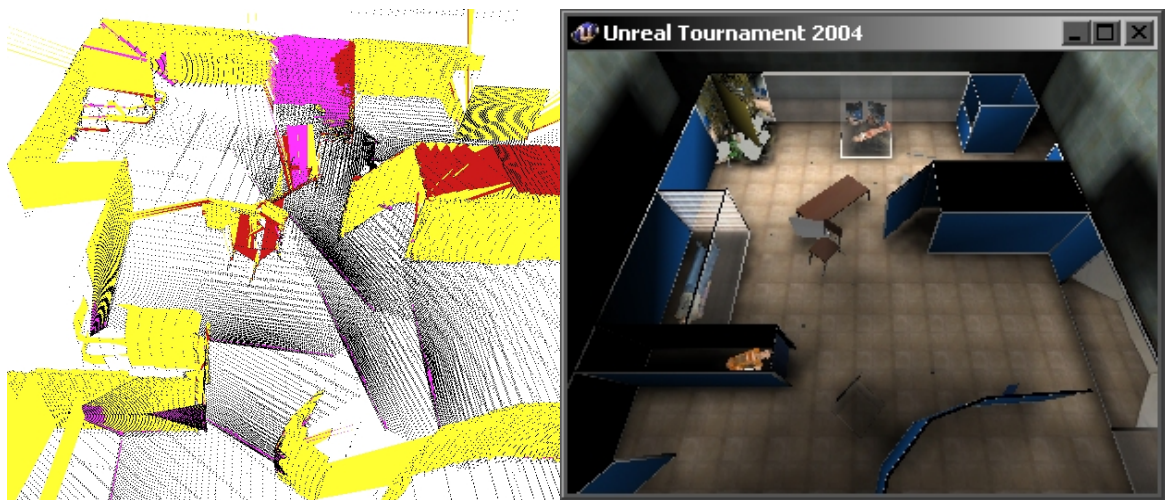


Figure 5. - 3D map of the yellow arena returned by Kurt3D (left) and a corresponding view from UT2004 (right).

2.1.2 HOKUYO URG-04LX

A more recent and more sophisticated counterpart sensor of the SICK LMS200, belonging to the group of range sensors, is the HOKUYO URG-04LX. A picture of this sensor is presented in Figure 6.



Figure 6. - HOKUYO URG-04LX

This sensor is much lighter and smaller than its older counterpart. Without an heat sink, it is 5cm long, 5cm wide, 7cm height and 160g in weight. It also has very good precision (1cm in range and 0.36° in angle), medium range (4m maximum) and is considerably fast at returning measurements (100 milliseconds). As can be noticed, this sensor as a range significantly smaller than the SICK LMS200 and its angular precision is a little worse but, the angular precision difference, is probably due to the difference in scanning angles, 240° for this sensor versus 180° for the SICK LMS200. The scanning area of this sensor is similar to the SICK LMS200 scanning area, with the respective differences in angles and range, since it too only measures in a horizontal plane parallel to its base.

2.1.3 Mesa Imaging SR4000

Still in the group of the range sensors, there is the Mesa Imaging SR4000 (18). This sensor, presented in Figure 7, is similar to a camera but instead of acquiring an optical image, it measures distances. It can be perceived as each of its pixels being an infra-red sensor, each taking a measure of the distance between themselves and the nearest non-transparent object. The result will be a map of distances from each pixel of the sensor to the object being measured like the one presented in Figure 8, where smaller distances correspond to lighter colours.



Figure 7. - Mesa Imaging SR4000 sensor



Figure 8. - An office chair (left) and as viewed by a SR4000 sensor (right).

This sensor is 6.5 cm wide, 6.5 high, 6.8 long and weights 470 g. Although smaller than the SICK LMS200, this sensor gains, to some extent, over some of the great advantages of the former sensor because despite having a range of only 5m, it has more or less the same resolution (1cm in range and 0.23° in angle) and it too is still fast at returning measures (up to 54fps with a 176×144 grid sensor which corresponds to a maximum of $176 \times 144 \times 54 = 1368576$ measures per second). Despite having only 43.6° in horizontal angle, it has the great advantage that it performs measurements directly in 3D without the need of a tilt unit due to its vertical angle of 34.6° (compared to 0° of the SICK LMS200).

2.2 State of the Art in Vision Sensors

2.2.1 Stereo Pair of Cameras

Another method of generating 3D maps is the usage of a stereo pair of cameras. This method is relatively complex and demands a great deal of computing power to be performed in real time with negligible delay. Figure 9 shows the view from a stereo pair of cameras.

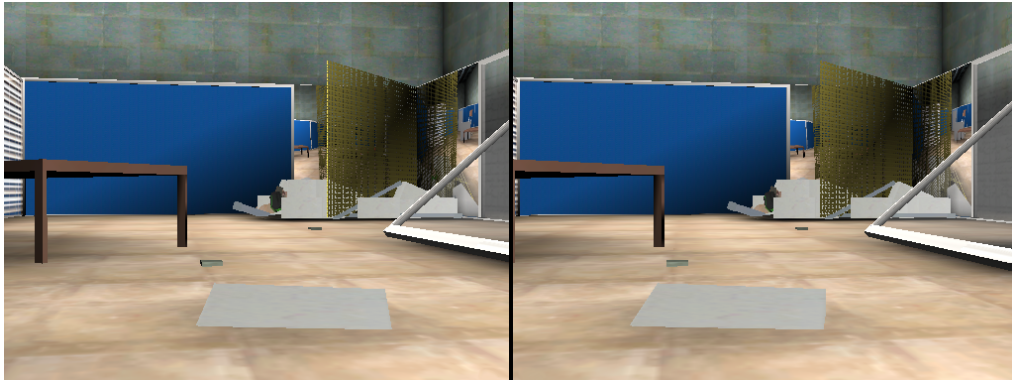


Figure 9. - View from a stereo pair of cameras.

Performing measurements from a pair of stereo images involves sophisticated algorithms of image processing to detect and match the objects from both images, like presented in Figure 10, and to extrapolate the distances of the objects relative to the robot (11), like presented in Figure 11. The distance between the robot and an object will depend on the distance that separates both cameras and on the position the object has inside each image.

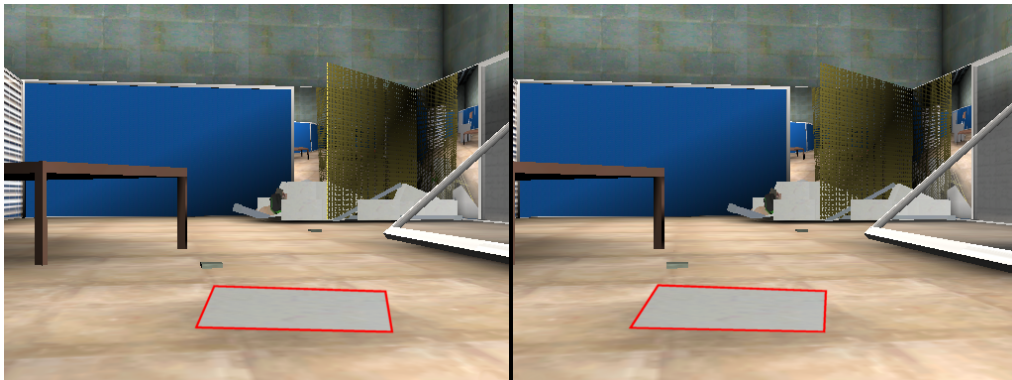


Figure 10. - Object match (in red lining) on a stereo pair of images.

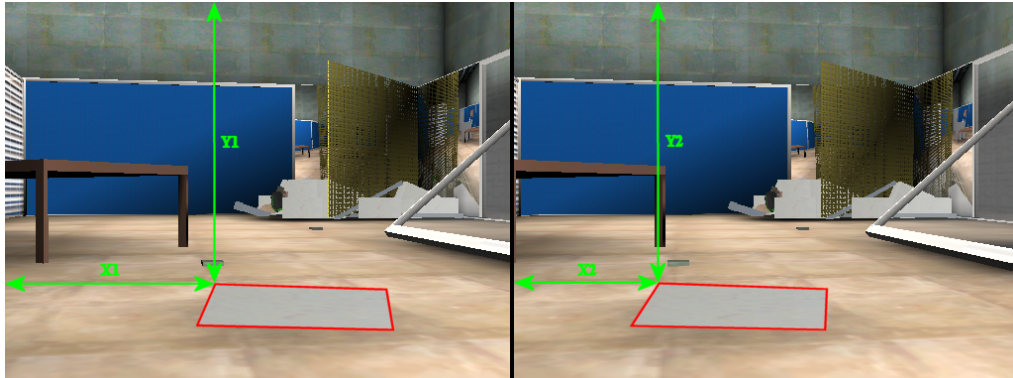


Figure 11. - Position of an object (in green) inside a stereo pair of images.

As can be noticed in Figure 10, the same object in both images has slightly different shapes due to the difference in perspective between the cameras, which is one of the difficulties encountered when trying to match objects in a stereo pair of images.

Measuring distances from a stereo pair of images as the added difficulty of having to deal with problems like the illusion of having different colours, even that an object has really only one, due to the fact that the illumination is not uniform all around every object and the cameras will capture that difference of illumination as being several colours, some darker than others. Figure 12 shows an office room that was captured with a pair of stereo cameras, was processed to build the map and finally was covered with textures to help humans visualize the environment.



Figure 12. - Office reconstructed after the real one had been captured by a stereo pair of cameras and processing had occurred.

2.2.2 Omnidirectional Camera

Currently, there are techniques of capturing the surroundings using just one camera. One of those techniques is by using an omnidirectional camera, like the one presented in Figure 13. It is basically a camera pointed at a mirror, usually hyperbolic, which increases the camera field of view and allows it to have a 360° horizontal by, almost, 180° vertical view of the surroundings, like demonstrated in Figure 14.



Figure 13. - An omnidirectional camera sensor.

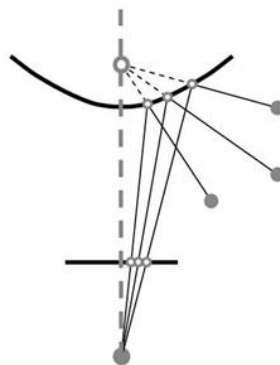


Figure 14. - Illustration of the operation of an omnidirectional camera.

In Figure 14, the upper black line represents the mirror and the bottom black line represents the camera sensor. The grey solid lines represent the paths of the light and the grey dashed line represents the middle of the camera-mirror setup with their corresponding focal points, a grey dot at the bottom and a small grey circle on top of the grey dashed line

respectively.

A view from an omnidirectional camera perspective is presented in Figure 15. Notice that the black circle at the centre of the image is the camera and that the objects at the edges of the image are stretched.

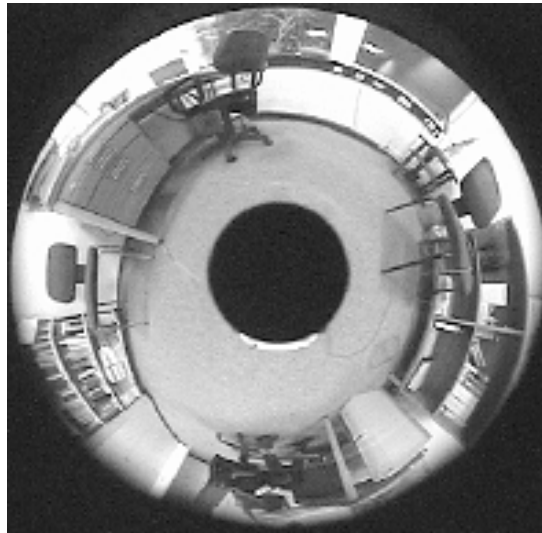


Figure 15. - View of an office from an omnidirectional camera.

As can be deduced from Figure 15, to process the image, it is required the application of an algorithm that performs the inverse of the distortion of the image, which can represent a significant overhead of processing, depending on the type of mirror and on the order of the equation chosen to approximate the effects of the mirror.

Although relatively more complex to process and calibrate than the stereo pair of cameras, mainly because of the distortion of the mirror, a single omnidirectional camera can be used to measure distances to objects (20).

2.2.3 Fish-eye Camera Lens

Still in the group of vision technology, and also using just one camera, is the fish-eye lens. This type of lens, shown in Figures 16, is in reality a group of lenses, as presented in Figure 17, that are combined in a manner that increases the field of view of a camera by compressing the scene being captured to fit the normal field of view of the camera, as presented in Figure 18. The photo of Figure 18 was taken with the lens of Figure 16.



Figure 16. - Fish-eye lenses.

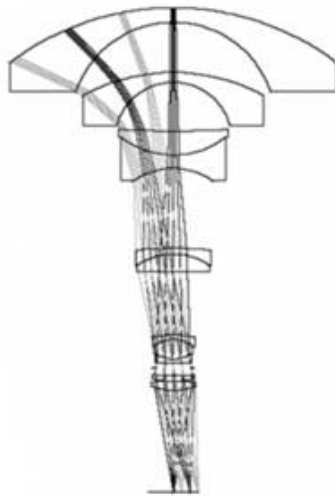


Figure 17. - Schematic of a fish-eye camera setup.



Figure 18. - View from a fish-eye lenses.

Much like the omnidirectional camera, images taken using this technique are also more complex to process than images taken with the stereo pair of cameras due to the distortion of the lenses which have to be counteracted by an algorithm that performs the inverse of the distortion of the lenses.

2.3 Additional Range Sensors

Returning to the range sensors, there are two basic sensors that, nowadays, are not usually used to measure distances but to detect if a collision is imminent. Those sensors, presented in Figure 19, are the sonar and the infra-red, respectively left and right. Small dimensions, lightweight and low power consumption are some of the characteristics these sensors possess.

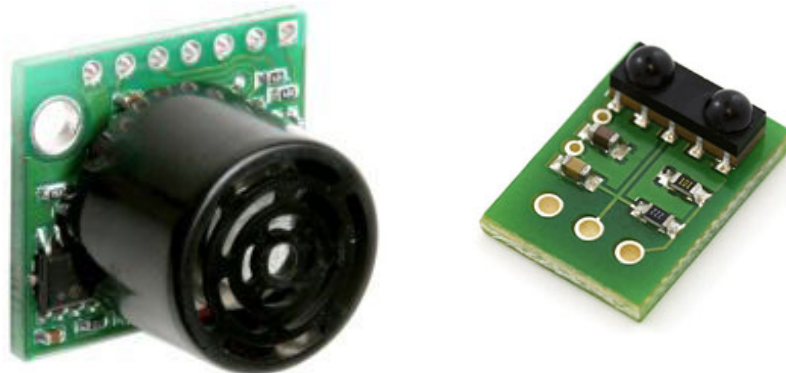


Figure 19. - Sonar sensor (left) and Infra-red sensor (right).

Although they lack precision for great distances, specially in harsh conditions, it is not too affected at close range. Due to the fact that these sensors are mainly used to detect the imminence of a collision, their presence in numbers is usually higher than any other sensor placed on a robot and usually cover the robot's body around a specified height.

2.4 Summary

In this chapter, the state of the art in sensor technology used in robotic search and rescue for mapping was presented along with a brief description of each sensor. The sensors described were the SICK LMS200, the HOKUYO URG-04LX, the Mesa Imaging SR4000, the stereo pair of cameras, the omnidirectional camera and the fish-eye lens. There were also presented two sensors, the infra-red and the sonar, that are not usually used for mapping.

3 Project and Implementation

This chapter presents the biggest part of the work performed throughout the thesis. It explains the 3D mapping methodology chosen, presents its implementation and justifies all the decisions taken about the robot's hardware and software architectures.

3.1 Mapping Strategy

There are some characteristics that are desirable in a search and rescue robot. Some of these characteristics are presented in Table 5.

Table 5. - List of some desirable characteristics in a search and rescue robot.

Characteristic number	Characteristic description
1	Small dimensions to allow the robot to reach tight places.
2	Light to allow easier movement and manoeuvring, and less disturbance of the surroundings.
3	Long operational range to map the entire area and, preferably, find all victims before the battery is depleted.
4	Produce accurate maps, both for its own navigation and the human rescue teams.
5	Detects 100% of the victims.
6	Flawless localization, meaning that it will always know exactly where it is and where it has been.
7	Flawless navigation, meaning that, if possible, it can always find a way to reach its destination that inflicts the least possible disturbance in the ambient.
8	Moves quickly but without compromising the stability of the environment, meaning that it does not hit objects in its surroundings neither it provokes collapses of objects (like debris).

The first three characteristics of the list can be achieved by using the least possible amount of reduced size low power sensors. The third characteristic will be augmented proportionally to the sensors decrease in power consumption. The partial solution for the last characteristic of the list is somewhat related with the solution for the first three because, if the sensor count is low and the sensors have small sizes, then the robot can have smaller dimensions, which will make it lighter and demand less binary from the engines, translating any spare binary in quickness of movement, which solves the quickness part of that characteristic of the list.

In order to have the fewest sensors as possible, the best way is for the sensors to have some mobility instead of being statically placed on the robot. By installing them in a moving

platform, one single sensor can replace several others and still return equivalent measurements. It is stated equivalent measures because the actual measure presented by the sensor will be different but the resulting computed measure will be equal. With an increased mobility allied to a sweeping tactic, it is possible to that single sensor to return even more measurements than all the others statically placed, making it possible to obtain a 3D map of part of the world.

By optimizing the placement of the sensor sweeping platform in the robot and carefully choosing the sensors that are installed in it, we can discard most of the other sensors that usually cover the robot's body like touch sensors, to detect impacts, or sonars, to detect proximity, which also contribute to the weight of the robot.

To produce the most accurate maps (characteristic four in the list), a sensor fusion algorithm should be used to combine all sensor measures in order to take all the individual errors of the sensors and minimize the final error.

3.2 Constraints

The proposed mapping strategy imposes some constraints on the shape of the robot's body, sensors used and their placement. Those constraints are also related to the intended characteristics of the robot. Table 6 lists the constraints and the reasons why they are required.

Table 6. - Constraints for the robot and the reasons why they are required.

Constraint	Reasons
Use just one sensor of each type, if possible.	To reduce the weight of the robot and increase the lifetime of the battery.
Sensors should be as small and light as possible	To further decrease the weight of the robot, the energy spent and, at the same time, improve its manoeuvrability, due to the decreased weight and size, and improve its range, due to the extended battery time.
Mount all sensors required in a moving platform.	This will allow the use of the same sensors to create a 3D map of part of the world without moving the robot's body.
The moving platform should have very small movement constraints.	This will allow the mapping of a bigger part of the world without moving the robot's body and, thus, without possibly introducing errors due to the change of position of the robot.
The moving platform should be placed in one of the extremities of the robot's body, away from any other part of the robot.	Without any other part of the robot in the way of the sensors, more of the world can be captured in one sweep of the moving platform. Also, there is no need to introduce algorithms to filter the parts of the robot from the sensors measurements.
Sensors should be able to scan the ground in front of the robot.	In order to detect holes or obstacles that prevent the robot's movement, or any other hazardous that can risk the integrity of the robot.
Sensors should be able to detect obstacles on top of the robot.	To be able to measure the height of the map on that position and allow an evaluation of the stability of the structure by the engineers of the search and rescue team.

3.3 Robot Hardware Architecture

3.3.1 Desired Architecture

A robot architecture, that was thought to comply with the constraints, is the one in Figure 20. It is composed of a body with four wheels setup as a differential steering wheel system, one GroundTruth sensor mounted in the robot's body, one Pan-Tilt-Zoom (PTZ) platform placed in the frontal part of the robot's body, and finally one infra-red (IR) sensor and one sonar sensor, both mounted on the PTZ to perform measurements of the environment in order to generate the map. The PTZ would be able to move at least 90° in each direction (up/down/left/right) from a central position, due to its base be mounted vertically, and the sensors field of view would not be obscured by any part of the robot.

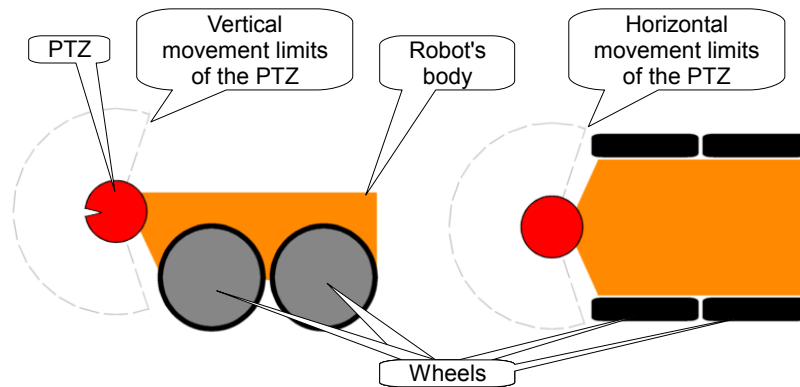


Figure 20. - Schematic of the desired implementation of the robot.

3.3.2 Implemented Architecture for Testing

Due to lack of knowledge in 3D image editing, the desired robot architecture was not implemented. Instead a modified cut-down version of the P2AT shown in Figure 21, which already exists in USARSim, was used for testing purposes.

The P2AT already possessed some of the most significant characteristics we expected. It was stripped of all its sensors except the ground truth, one sonar, one IR and the camera. The original P2AT has only one camera but a second one was added to allow the sensors to occupy the central position of the PTZ where the camera of the original P2AT was. The cameras were moved to the extremities of the PTZ and were left on the robot to enable us to see where the PTZ is pointing, and give an estimate to where the sensors are pointing, and did not contribute to the generation of the map. The ground truth sensor remained in the physical centre of the robot's body and the two sensors left (IR and sonar) were moved to the PTZ platform. Unfortunately, I have not been able to rotate the PTZ base into a vertical position, so it was still placed in the frontal part of the robot but in a lowered position that allowed the ground in front of the robot to stay in the sensors line of sight when the PTZ is pointing down.

The modified version of the P2AT is presented in Figure 22 along with the relative placement of its sensors and PTZ.



Figure 21. - Image of the original P2AT in USARSim, the robot modified to be used for testing.

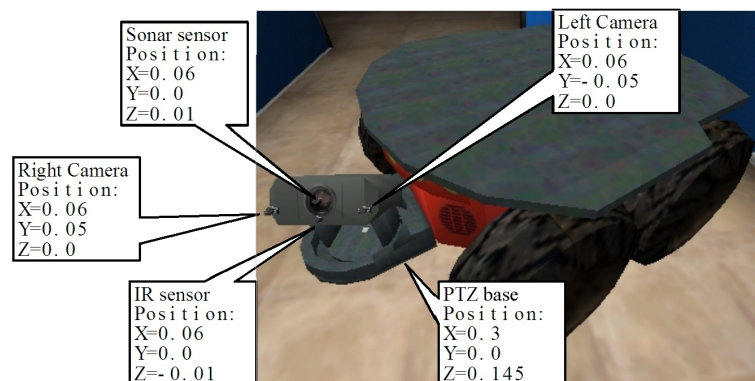


Figure 22. - The modified P2AT robot and its sensors relative placement.

The positions of the sensors shown in Figure 22 are relative to the $(x=0; y=0; z=0)$ point of the part of the robot they are attached to. The complete configuration of the robot, and how to perform it, is presented in a later point of this document.

3.3.3 Implementation of the Testing Robot

Since the robot used in this work is based in a pre-existing one, implementing it is very easy. USARSim installs two files in the "System" folder of UT2004 called "USARBot.ini" and "USARMisPkg.ini" that are used to provide the configuration of the robots and the configuration of the mission packages, respectively.

Starting with the mission package, the PTZ moving constraints must be altered to allow the PTZ to move as much as we desire for this work. Luckily, the physical design of the PTZ parts allows for this, otherwise, changing this settings would not have any effect since UT2004 uses the part design to simulate the rigid body. For this work the constraints were removed completely in the configuration file and are performed instead by the software robotic agent. To remove the constraints, locate the text "[USARMisPkg.CameraPan]" and "[USARMisPkg.CameraTilt]" and replace the corresponding MinRange values by 1 and the MaxRange values by 0. When MinRange is set to be greater than MaxRange, the corresponding part does not have any constraints.

Now, the robot must be created. To create the robot, it is only necessary to copy the "P2AT.uc" file in the "USARBot\Classes" folder of UT2004 and paste it with the name of our robot. The robot in this work was called "Mapper", so the pasted file's name must be changed to "Mapper.uc". Be aware that all text mentioned here is case sensitive, including the name of the file. Next, the file of the new robot ("Mapper.uc") must be opened in a simple text document, like notepad, and the text presented in Table 7 must be replaced from the original to the modified. After this step, USARSim must be compiled. To perform the compilation is just necessary to click the link "Compile" under the "Programs\USARSim" folder in the "Start menu" of the Windows® OS like it was presented earlier in this document. If no errors occur, a message of successful completion will be presented

Table 7. - Text replacement in "Mapper.uc" robot definition file.

Original	Modified
class P2AT extends SkidSteeredRobot config(USARBot);	class Mapper extends SkidSteeredRobot config(USARBot);
KParams=KarmaParamsRBFULL'USARBot.P2AT.KPa rams0'	KParams=KarmaParamsRBFULL'USARBot.Mapper.KP arams0'

Finally, the robot must be configured. To configure it, the file "USARBot.ini" must be edited and the configuration of the P2AT robot must be copied. That is a part of the file like the one in Table 8. Then it must be pasted and modified to remove unnecessary sensors. In the end, it must be like the configuration in Table 9.

Table 8. - P2AT robot configuration

P2AT robot configuration
[USARBot.P2AT] bDebug=False Weight=14 Payload=40 ChassisMass=1.000000 (...) Sensors=(ItemClass=class'USARBot.GroundTruth',ItemName="GroundTruth",Position=(X=0.0,Y=0.0,Z=-0.0),Direction=(Y=0.0,Z=0.0,X=0.0))

Table 9. - Mapper robot configuration

Mapper robot configuration
<pre> [USARBot.Mapper] bDebug=True Weight=14 Payload=40 ChassisMass=1.000000 MotorTorque=40.0 bMountByUU=False batteryLife=50000 JointParts=(PartName="RightFWheel",PartClass=class'USARModels.P2ATTire',DrawScale3D=(X=1.0,Y=0.55,Z=1.0),bSteeringLocked=True,bSuspensionLocked=true,Parent="",JointClass=class'KCarWheelJoint',ParentPos=(Y=0.20399979,X=0.13199987,Z=0.13104749),ParentAxis=(Z=1.0),ParentAxis2=(Y=1.0),SelfPos=(Z=-0.0),SelfAxis=(Z=1.0),SelfAxis2=(Y=1.0)) JointParts=(PartName="LeftFWheel",PartClass=class'USARModels.P2ATTire',DrawScale3D=(X=1.0,Y=0.55,Z=1.0),bSteeringLocked=True,bSuspensionLocked=true,Parent="",JointClass=class'KCarWheelJoint',ParentPos=(Y=-0.20399979,X=0.13199987,Z=0.13104749),ParentAxis=(Z=1.0),ParentAxis2=(Y=1.0),SelfPos=(Z=-0.0),SelfAxis=(Z=1.0),SelfAxis2=(Y=1.0)) JointParts=(PartName="RightRWheel",PartClass=class'USARModels.P2ATTire',DrawScale3D=(X=1.0,Y=0.55,Z=1.0),bSteeringLocked=True,bSuspensionLocked=true,Parent="",JointClass=class'KCarWheelJoint',ParentPos=(Y=0.20399979,X=-0.13199987,Z=0.13104749),ParentAxis=(Z=1.0),ParentAxis2=(Y=1.0),SelfPos=(Z=-0.0),SelfAxis=(Z=1.0),SelfAxis2=(Y=1.0)) JointParts=(PartName="LeftRWheel",PartClass=class'USARModels.P2ATTire',DrawScale3D=(X=1.0,Y=0.55,Z=1.0),bSteeringLocked=True,bSuspensionLocked=true,Parent="",JointClass=class'KCarWheelJoint',ParentPos=(Y=-0.20399979,X=-0.13199987,Z=0.13104749),ParentAxis=(Z=1.0),ParentAxis2=(Y=1.0),SelfPos=(Z=-0.0),SelfAxis=(Z=1.0),SelfAxis2=(Y=1.0)) MisPkgs=(PkgName="CameraPanTilt",Location=(Y=0.0,X=0.3,Z=0.145),PkgClass=Class'USARMisPkg.CameraPanTilt') Cameras=(ItemClass=class'USARBot.RobotCamera',ItemName="CameraLeft",Parent="CameraPanTilt_Link2",Position=(Y=-0.05,X=0.06,Z=0.0),Direction=(Y=0.0,Z=0.0,X=0.0)) Cameras=(ItemClass=class'USARBot.RobotCamera',ItemName="CameraRight",Parent="CameraPanTilt_Link2",Position=(Y=0.05,X=0.06,Z=0.0),Direction=(Y=0.0,Z=0.0,X=0.0)) HeadLight=(ItemClass=class'USARBot.USARHeadLight',ItemName="HeadlightLeft",Parent="CameraPanTilt_Link2",Position=(Y=-0.05,X=0.08,Z=-0.03),Direction=(Y=0.0,Z=0.0,X=0.0)) HeadLight=(ItemClass=class'USARBot.USARHeadLight',ItemName="HeadlightRight",Parent="CameraPanTilt_Link2",Position=(Y=0.05,X=0.08,Z=-0.03),Direction=(Y=0.0,Z=0.0,X=0.0)) Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="Sonar",Parent="CameraPanTilt_Link2",Position=(Y=0.0,X=0.06,Z=0.01),Direction=(Y=0.0,Z=0.0,X=0.0)) Sensors=(ItemClass=class'USARBot.IRSensor',ItemName="IR",Parent="CameraPanTilt_Link2",Position=(Y=0.0,X=0.06,Z=-0.01),Direction=(Y=0.0,Z=0.0,X=0.0)) Sensors=(ItemClass=class'USARBot.GroundTruth',ItemName="GroundTruth",Position=(X=0.0,Y=0.0,Z=-0.0),Direction=(Y=0.0,Z=0.0,X=0.0)) </pre>

The cameras and the headlights can be removed from the file because they are not used by the software robotic agent, but they may be handy if someone intends to watch the robot in action. Actually, if the robot had been implemented as a real robot, the cameras and headlights would not have been inserted.

From this point on, if the simulator is started and the "Mapper" robot invoked, it should appear on the scenario.

3.4 Sensors and Sensor Models

3.4.1 In USARSim

In USARSim, the IR sensor is modelled as a straight line that starts at the sensor and ends when it intersects the first non transparent object. It then returns the length of the line. The sensor has a maximum range and an error that increases linearly with the distance of the point measured. However, its characteristics can be modified in the “USARBot.ini” configuration file.

The sonar is modelled by many straight lines starting at the sensor, projecting themselves away from the sensor, inside its cone of action, and ending in the first object (transparent or not). After one pass of measurements, the length of smallest line is returned. Similarly to the IR sensor, the sonar sensor has a maximum range and an error that increases linearly with the distance of the point measured. It also has an angle that defines the sonar's cone of action.

The sensors with their properties, their descriptions and the values they were configured to in this work are listed in Table 10.

Table 10. - List of sensors, their properties, property descriptions and the values they assume for this work.

Sensor name	Sensor property	Property value	Property description
Sonar	Maximum range	5 m	The maximum range of the sensor in normal conditions
	Maximum error	0.05 m	The maximum error of the sensor in normal conditions. The error is a function of distance.
	Angle of cone	0.0873 rad	It is the line of sight angle of the sensor. Any object outside this angle is not sensed.
	Position	(X=0.06,Y=0.0,Z=0.01) m	The relative position of the sensor in regard to the centre of tilt unit of the PTZ.
IR	Maximum range	5 m	The maximum range of the sensor in normal conditions
	Maximum error	0.05 m	The maximum error of the sensor in normal conditions. The error is a function of distance.
	Position	(X=0.06,Y=0.0,Z=-0.01) m	The relative position of the sensor in regard to the centre of tilt unit of the PTZ.

In the experiments conducted, which are detailed in a later point of this document, for the particular case of the measurements without error that serve to get the comparison references, the error was set to zero.

3.4.2. In Robotic Software Agent

Each measurement from the IR or the sonar sensors will be converted into a line or cone of points, respectively, where each one of those points will have a probability associated with it. That probability will reflect the probability of that point being occupied and it depends on the sensor's model of uncertainty. Keep in mind that a line can be understood as a special case of a cone where the angle of the cone is zero. This approach allows the same model of uncertainty to be used with both types of sensors.

The model of uncertainty used in this work will be the one represented in Figure 23 (10) and it divides the sensor's action range in three regions. Region I is the region where the points are probably occupied. Region II is the region where the points are probably empty. Region III is the region where it is unknown if the points are occupied or empty.

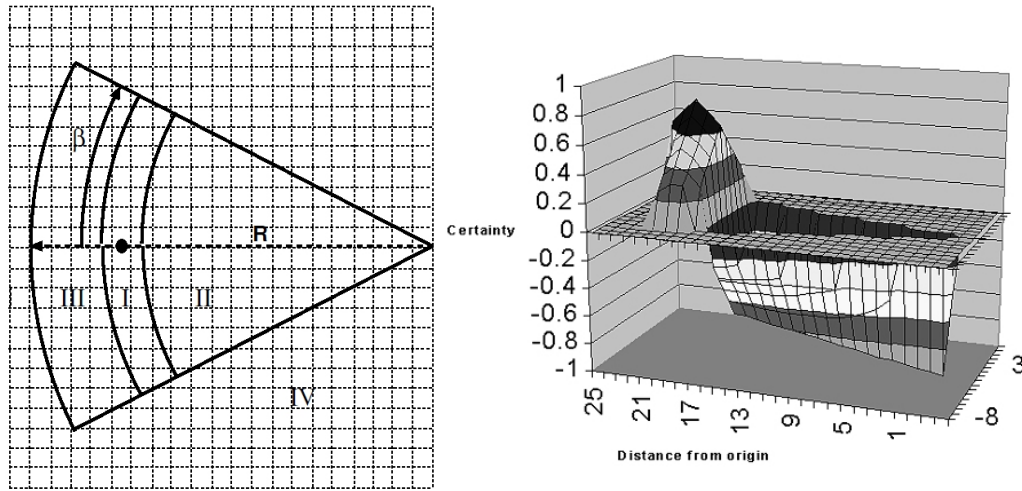


Figure 23. - Representation in 2D (left) and 3D (right) of the sensor model of uncertainty removed from (10).

The regions' limits are determined by using the maximum range of the sensor and the maximum error for the measured distance. If a sensor measures the distance A and the sensor model states that, for that distance, the maximum error is E , then the limit that separates Region I from Region II is at a distance of $A-E$ from the sensor, and the limit that separates Region I from Region III is $A+E$. In this work, the sensor model for the error over the distance presented in Figure 24 is used for both sensors. The curve presented is also the default for range sensors in USARSim but can easily be modified.

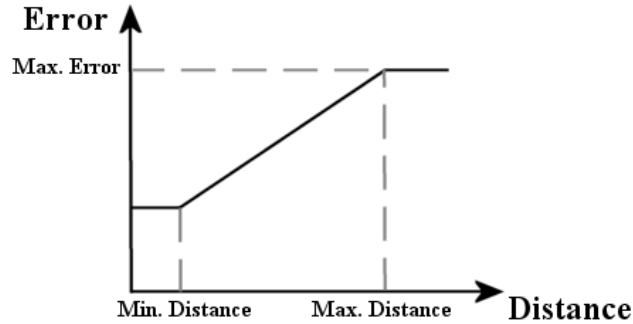


Figure 24. - Example of a sensor model for the error.

The probability of each point will be calculated using one of the two methods mentioned earlier in this document. The first is the dumb method that updates the probabilities using a mean and the other is the Bayesian method.

For the dumb method, the sensor model presented will be applied to each measure from the sensor but the error will always be regarded as zero, meaning that Region I will be constituted solely by the points that present themselves at the measured distance and will be given the probability of 100% occupied. All other points between the sensor and the measured distance will be given the probability of 0% occupied. If any of those points already exists in memory, their probabilities will be updated by performing the mean value of the currently given probability with their old value of probability of occupancy.

For the Bayesian method, the formulas used for the calculation of the probabilities are (1) to (3) and belong to regions I to III of Figure 23, respectively.

$$P(occupied) = \frac{\frac{R-r}{R} + \frac{\beta-\alpha}{\beta}}{2} \times max_{occupied} \quad (1)$$

$$P(occupied) = 1 - \frac{\frac{R-r}{R} + \frac{\beta-\alpha}{\beta}}{2} \quad (2)$$

$$P(occupied) = unknown ; do nothing \quad (3)$$

Where R is the sensor's maximum range, r is the measured distance from the sensor to the point being calculated, β is the beam half angle, α is the angle between the line that connects the centre of the sensor to the point being calculated with the line that defines the centre of the

cone, and $\max_{\text{occupied}}$ is a pre-defined value that defines the assumption that a point is already occupied before any measurements are made. In the special case of the IR sensor, since there is no β nor α , the term $\frac{\beta - \alpha}{\beta}$ in the above formulas is zero.

After calculating the probability of a measured point using formulas (1) to (3), the probability of each point in the world is updated using formula (4)

$$P_{\text{world}}(\text{occupied}) = \frac{P \times P_{-1}}{P \times P_{-1} + (1.0 - P) \times (1.0 - P_{-1})} \quad (4)$$

Where P is the probability of the measured point and P_{-1} is the previous probability of that point in the world. If there is no previous point (P_{-1}) then $\max_{\text{occupied}}$ is used instead.

3.5 Robot Software Architecture

3.5.1 Hardware Emulation

The robot's computational platform was built with two main purposes in mind. One was to allow to count the number of instructions used to implement the proposed algorithms and, the other, was to allow a quick conversion of the algorithms from the emulated platform to a real robot. With this in mind, the robotic agent architecture was developed to emulate the hardware of a real robot. It emulates a 32bit single threaded computer with separate instruction and data pointers, memories and buses, where all peripherals are regarded as memory positions. There is no floating point arithmetic, meaning that all operations are performed over integers including functions like sine, cosine, tangent, etc. Also, to keep the architecture as simple as possible, there are no advanced features like stack, internal registries (except for the instruction and data pointers), cache or memory management hardware and all memory positions correspond to a 32bit integer variable.

Figure 25 shows the emulated hardware architecture of the entire robot. The white blocks are the "real" hardware of the robot and the grey blocks are the classes that emulate the hardware. We can see that the "real" hardware is composed of a CPU, RAM and peripherals. The peripherals are sub-divided in input and output peripherals. Input peripherals contain Sonar, IR, robot position and posture, light intensity information and light toggle information. Output peripherals contain light intensity, light toggle, wheels speed, pan and tilt position.

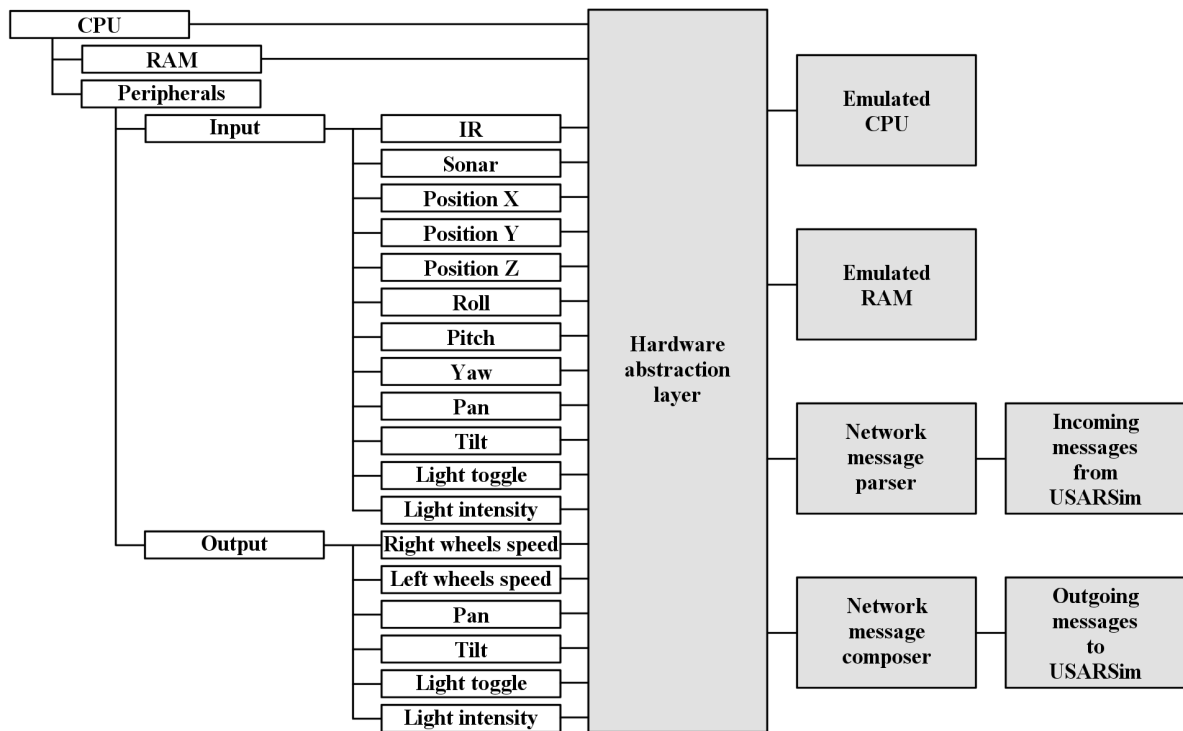


Figure 25. - Emulated hardware architecture.

The blocks that emulate the hardware do not translate to hardware in a real robot. Those blocks handle the communications with the simulator, emulate the RAM, emulate the sensors and actuators, and implement the abstraction layer that hides the actual implementation of the sensors, actuators and RAM, and treats them as memory positions.

For the emulation to be complete, a set of assembly like instructions was created for the emulated CPU. These instructions were loosely based on the 80C51 microprocessor instructions but were implemented in such a way that the programmer could benefit to some extent, in the assembly, from characteristics typical of high level languages like encapsulation. The different types of instructions and their purpose is presented in Table 11. Since the complete set of instructions is too extensive, it is presented only in Annex B along with the corresponding syntax, usage description and examples. The parameters received by an instruction are each separated by one space. It is assumed that all instructions take only one clock cycle to execute.

Table 11. - Different types of instructions of the emulated CPU.

Instruction type	Instruction type name	Description
A	Conditional function call	In this instruction type a function is called if, or while, the value of a certain variable is true or when the comparison of the values of two certain variables is true. Most of the instructions of the emulated CPU belong to this type of instructions.
B	Arithmetic	This instruction type includes all the arithmetic operations over the value of one or two variables (addition, sum, subtraction, multiplication and division) as well as all calculations performed over the value of a single variable like sine, cosine, tangent, etc.
C	Memory addressing	With only two instructions, this instruction type allows for indirect memory manipulation. A variable can have the address of another variable saved as its value and the value of a variable can be loaded into the memory pointer as an address.

As one can conclude from Table 11 and Annex B the set of instructions is very limited but, as will be demonstrated, contains all the necessary instructions to process all the algorithms required by this work.

Carefully analysing the instructions in Annex B, one can conclude that most instructions are somewhat easy to understand and use, and that the usage of the memory addressing instructions can be a little confusing. To try and clarify the usage of each instruction type, a detailed description of each follows.

The instruction type easiest to understand is the arithmetic. These are ordinary calculations over variables and are straightforward to use. Figure 26 shows an example of the calculation of the sine of the sum of two numbers. The variables used are named a, b, c and d where a and b are the numbers to be added, c will hold the result of the sum and d will hold the result of the sine.

```

...
add a b c
sin c d
...

```

Figure 26. - Example of two arithmetic instruction type.

The conditional function call type of instructions are used to perform comparisons and call functions. All comparisons were defined to call a function to impose the modularization of the code, making it easier to analyse and implement. Figure 27 shows an example of a conditional function call type of instructions where two numbers are compared and the smaller number is subtracted to the greater number. The variables used are named a, b and c where a and b are the numbers to be compared and c will hold the result of the subtraction. The function called when a is greater than or equal to b is named add1 and the function called when a is smaller than b is named add2.

```

...
ifge2 a b add1 add2
...
add1: sub a b c
ret
...
add2: sub b a c
ret
...

```

Figure 27. - Usage example of an if instruction of the conditional instruction type.

When a function returns, it will return to the line of code following the function call except if the function was called with an while instruction. In the later case, the function will return to the same line of code it was called from and will execute the next line of code after the while instruction only when the comparison becomes false. In the case of the example in Figure 27 either function add1 or add2 will return to the instruction following the if (ifge2) instruction. Figure 28 shows the example of the usage of the while instruction by recursively calculating the factorial of a number. The variables are named r, t and n where r will hold the result of the calculation, t is a temporary variable and n is the wanted number of the factorial.

```

...
mov r 1
mov t n
wg t 1 factorial
...
factorial: mul r t r
sub t 1 t
ret
...

```

Figure 28. - Usage example of a while instruction of the conditional instruction type.

The memory addressing instruction type is the most hard to comprehend. It was

implemented to allow the usage of pointers, arrays, linked lists, etc. The easiest way to demonstrate its usage might be with an example. Figure 29 shows an example usage of the memory addressing instruction type by calculating the sum of the values of an array addressed by a variable. Like in many languages an array is implemented as consecutive memory positions where position zero is also the base address of the array. So, to access position x of an array, one needs to sum x to the base address of the array. The variables are named $v_0, v_1, \dots, v_n, a, n, t_1, t_2, s$ and i where v_0 to v_n are the array positions stored in sequential memory positions, a will hold the base address of the array, n is the number of elements of the array, t_1 is a temporary variable that will hold the address of the current position of the array, t_2 is a temporary variable that will hold the value of the memory position addressed by t_1 , s will hold the sum of the elements of the array and i is a counter that is used to address an element of the array.

```

...
ma v0 a
mov i 0
mov s 0
wl i n sum1
...
sum1: add a i t1
mv t1 t2
add s t2 s
add i 1 i
ret
...
```

Figure 29. - Usage example of the memory addressing instruction type.

3.5.2 Software Emulation

To improve the speed of the program that emulates the robot hardware architecture and software architecture, the actual instruction set for the CPU was not implemented but translated into equivalents in the language used to program. That language was C++ and Table 12 presents some example translations. Annex C presents the full list of equivalences between instructions as well as detailed descriptions of each instruction.

Table 12. - Some examples of conversions between the emulated CPU instructions and C++.

CPU Instruction	CPU Syntax/Example	C++ Syntax/Examples (in <i>italic</i>)
call function	call f0	<i>f0();</i>
if greater than or equal call function	ifge v0 v1 f0	<i>if(v0>=v1) f0();</i>
if less than or equal return	iflr v0 v1	<i>if(v0<=v1) return;</i>
while greater than call function	wg v0 v1 f0	<i>while(v0>v1) f0();</i>
move value to variable	mov v0 v1	<i>v1=v0;</i>
divide values and store result	div v0 v1 v2	<i>v2=v0/v1;</i> and v2 can be v0, v1 or another variable
divide values and store remainder	rem v0 v1 v2	<i>v2=v0%v1;</i> or <i>v2=v0-(v0/v1)*v1;</i> and v2 can be v0, v1 or another variable
store sine of value	sin v0 v1	<i>v1=sin(v0);</i> and v1 can be v0 or another variable
get memory address value	mv v0 v1	<i>v1=*v0;</i>

Looking closely at the list of all instruction equivalences in Annex C, one realizes that most conversions are easy to understand and straightforward to implement. But the instructions related to memory addressing can be a little difficult to realize their true potential. Figure 31 and Figure 32 show, in C++ and emulated CPU assembly respectively, an example of translation for an application using the memory addressing instructions. In that example, it is performed the sum of the elements of a linked list implemented in two arrays, one of values and other of indexes to the next element of the list. Assume that, if the next list position does not exist, the index will have a negative value. Also assume that the arrays are similar to those shown in Figure 31.

The variables are named first, list0, list1, ..., listn, index0, index1, ..., indexn, a, s and t where first is the position of the first element of the list, list0 to listn are the array positions of the list values stored in sequential memory positions, index0 to indexn are the array positions of the next elements of the list stored in sequential memory position, a will hold the value of the current element of the list, s will hold the result of the sum and t is a temporary variable to store addresses.

```
list[]={10,3,67,9};
index[]={2,3,-1,0};
first=1;
```

Figure 30. - Example of a linked list implemented using two arrays in a C like language.

```

...
s=0;
i=first;
while(i!=0){
a=list[i];
s+=a;
i=index[i];
}
...

```

Figure 31. - Example in C++ of the sum of the elements of a linked list implemented using two arrays.

```

...
mov s 0
ma list0 list
ma index0 index
mov i first
wge i 0 sum1
...
sum1: add list i t
mv t a
add s a s
add index i t
mv t i
ret
...

```

Figure 32. - Example in emulated CPU assembly instructions of the sum of the elements of a linked list implemented using two arrays.

In reality, to minimize the risk of errors occurring while running or compiling the program, the functions in C++ do receive parameters by reference instead of being passed through

global variables but this fact does not affect the translation from C++ to emulated CPU assembly as long as the names of the variables and functions are kept consistent.

3.6 Software Robotic Agent

With the hardware and software emulations defined, the software robotic agent can be programmed to order the robot in the simulator to execute actions, perform measurements and build the 3D map of the scenario. The robotic software agent is composed of several threads of execution, each with its functionality, and many functions that perform specific tasks like building the sensor cone for the model of uncertainty. One thread executes the hardware abstraction layer, other the parsing of the messages from the simulator and other executes the main program.

3.6.1 Main Program

The main program thread is responsible for managing and coordinating all other threads, and for imposing an order of execution of the tasks to be performed by the software robotic agent. The block diagram of the main program thread execution flow is provided in Figure 33. The grey block is the starting point. A more detailed list of the tasks of the main program thread and their corresponding execution flow is listed in Table 13 for better comprehension.

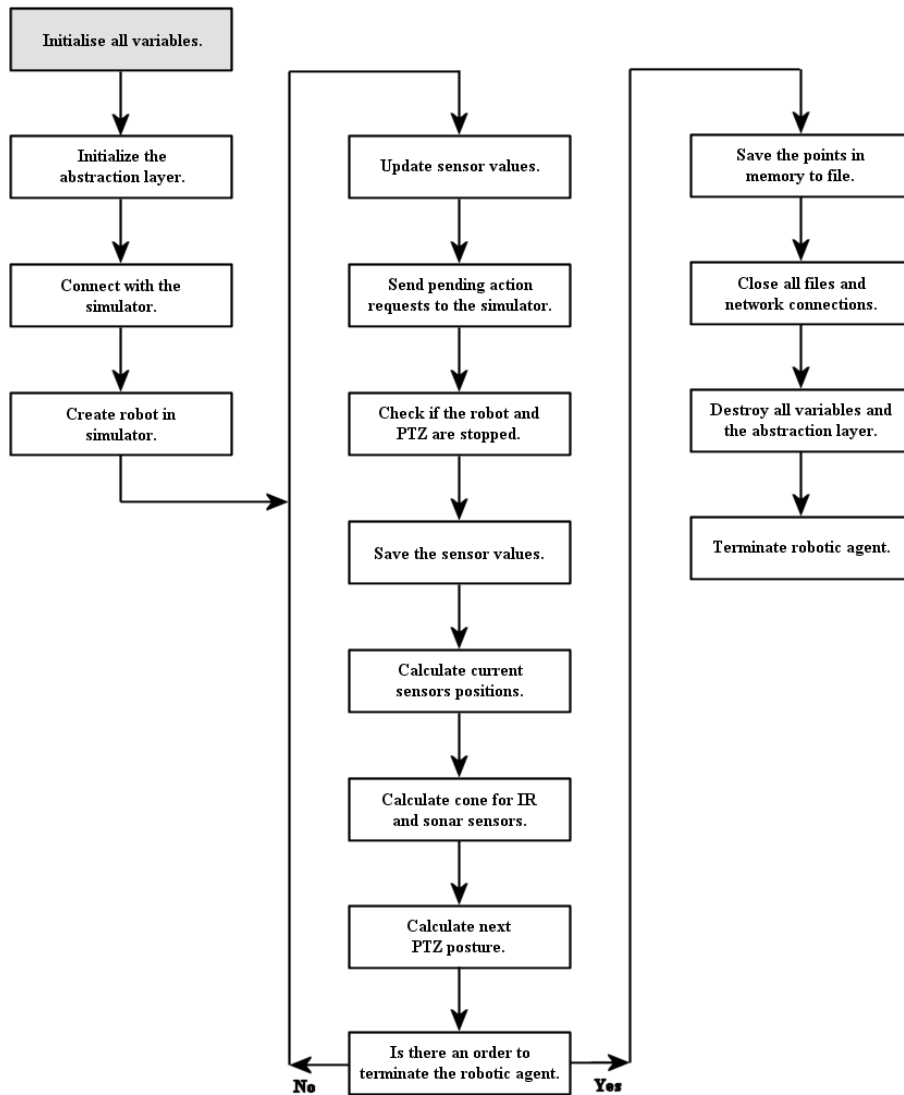


Figure 33. - Main program thread execution block diagram.

Table 13. - Detailed list of tasks executed by the main thread of the robotic agent.

Task number	Next task on success	Next task on failure	Task description
1	2	15	Initialise all variables.
2	3	14	Initialize the abstraction layer to emulate all peripherals, RAM, network message parser and network message composer on robot creation.
3	4	13	Connect with the simulator over the network.
4	5	12	Create robot in simulator.
5	6	12	Update sensor values and send pending action requests to the simulator.
6	7	11	Check if the robot and PTZ are stopped by comparing with previous values.
7	8	-	Temporarily stop updating all sensor values and save the values of the robot position (x,y,z), robot posture (roll, pitch, yaw), PTZ posture (pan, tilt), IR measured distance and sonar measured distance. Then restart updating the sensor values.
8	9	-	Calculate current sensors positions on the robot by using its current position and posture, and the PTZ posture. The positions are relative to the world referential.
9	10	-	Calculate cone for IR and sonar sensors.
10	11	-	Calculate next PTZ posture.
11	12	5	Check if an order to terminate the robotic agent has been received.
12	13	-	Save the points in memory to file.
13	14	-	Close all files and network connections.
14	15	15	Destroy all variables and the abstraction layer.
15	-	-	Terminate robotic agent.

Task 6 in Table 13 is to prevent the induction of errors in sensor measures due to the robot or the PTZ not being stopped. If the robot or the PTZ are moving when a sensor measurement is performed, the value measured would be higher or smaller than the real one due to, when the sensor signal was sent, the sensor position and posture changing to a position closer or farther to the object when the signal was received.

Task 7 on that same table is to prevent sensor values to get out of synchronization while each of the sensors is read due to the inability of the emulated CPU to read more than one memory position at once, since it is a single threaded processor. This task would not be required in a real robot since it would be possible to control when the measurements are performed.

3.6.2 Building the Cone

The most time and processor consuming task the emulated CPU must execute is the creation of the cones, specially for the sonar sensor due to its angle of view. Figure 34 shows a block diagram of the execution flow of the cone generation function, where the grey block is the starting point. Table 14 presents a more detailed list of tasks performed by the cone generation function.

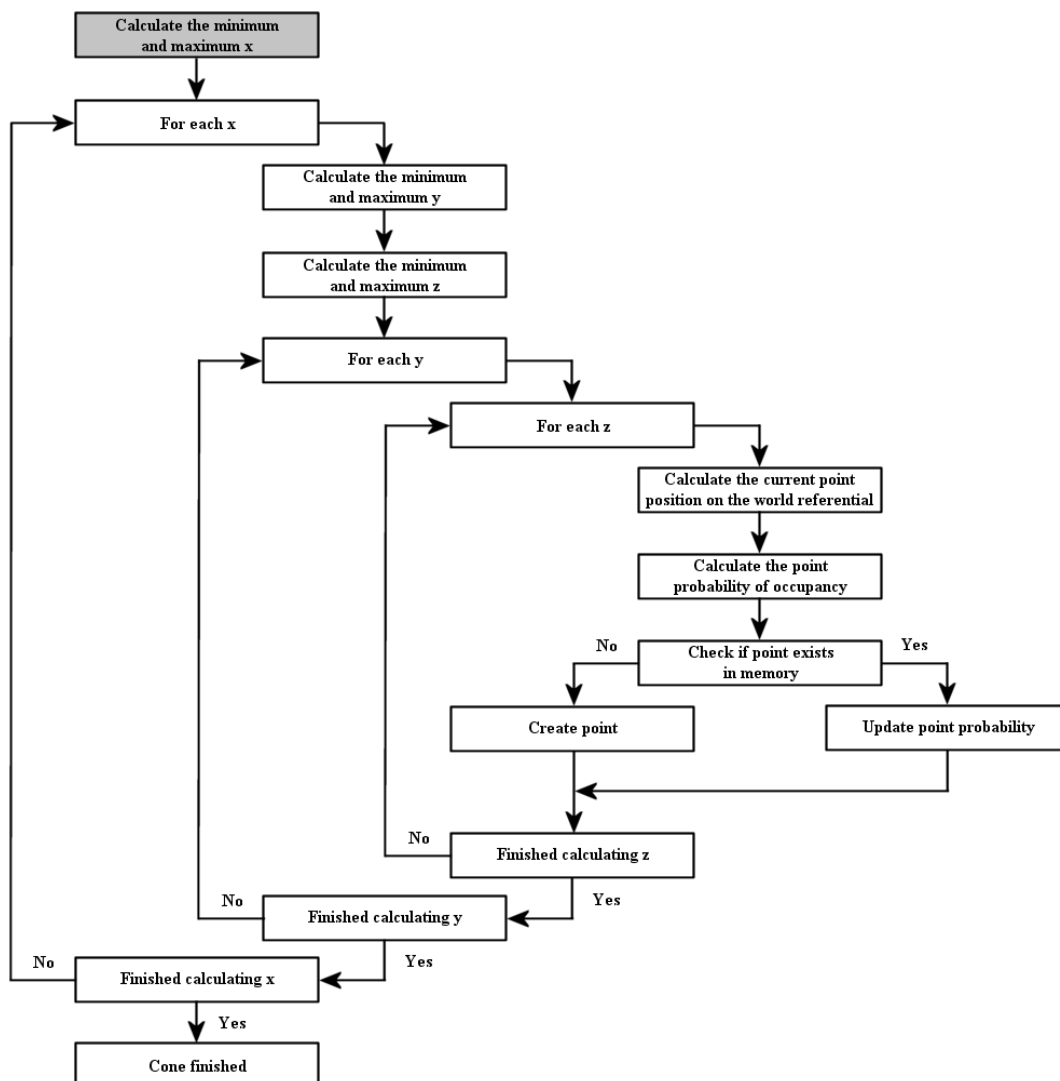


Figure 34. - Cone generation function execution block diagram.

Table 14. - Detailed list of tasks executed by the cone generation function.

Task number	Next task on success	Next task on failure	Task description
1	2	-	Calculate the minimum x (xmin) and the maximum x (xmax).
2	3	-	For each x between the xmin and xmax on the cone referential.
3	4	-	Calculate the minimum and maximum values of y and z (ymin, zmin, ymax and zmax, respectively) for the current x on the cone referential.
4	5	-	For each y between ymin and ymax on the cone referential.
5	6	-	For each z between zmin and zmax on the cone referential.
6	7	-	Calculate the current point position on the world referential.
7	8	-	Calculate the point probability of occupancy using the selected method.
8	9	10	Check if point already exists in memory.
9	11	-	Update the point probability using the selected method.
10	11	-	Create the point in memory.
11	12	5	Finished calculating all z.
12	13	4	Finished calculating all y.
13	14	2	Finished calculating all x.
14	-	-	Cone finished

Since, as stated earlier in this document, the IR line is a special case of the sonar cone, the points for the IR are generated by the same cone generation function with the differences stated earlier.

3.6.3 Managing Points

Due to the massive amount of points (millions) produced by the cone generation function, specially for sonar measures, a strategy to manage them had to be implemented to allow for a fast search of existing points and insertion of new points. The adopted strategy was to use an indexed ordered two-way linked list which is nothing more than an ordered two-way linked list to store the points and an index, implemented in an array, to index some of the points of the linked list, effectively turning a long linked list into blocks of small linked lists (cluster of points). Figure 35 presents the structure of the a two-way linked list and Figure 36 presents the resulting data structure of the indexed ordered two-way linked list.

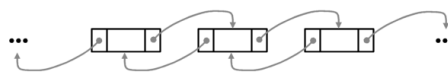


Figure 35. - Structure of a two-way linked list.

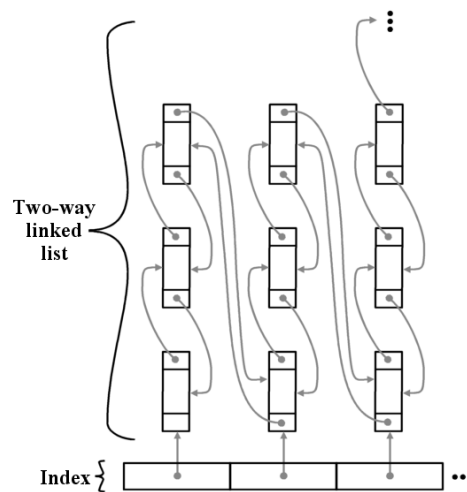


Figure 36. - Structure of the indexed ordered two-way linked list.

The linked list, similarly to the index, is implemented in a group of five arrays that store the point coordinates, the probability, the position of the previous node of the linked list and the next node of the linked list. The points are ordered by x, then by y and finally by z. The order of the points in the linked list is kept by setting the correct previous and next node positions much like presented in the example of Figure 37. The index in Figure 37 indexes every 4 nodes.

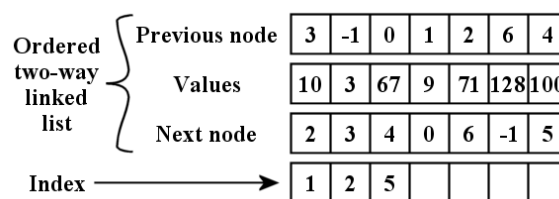


Figure 37. - Example of an indexed ordered two-way linked list implemented in arrays.

To find points, the index must first be searched. The search over the index is performed using a binary search algorithm and Figure 38 presents the block diagram of execution of the binary search algorithm implemented in this work to find points in the index. Table 15 provides a more detailed description of the algorithm. The letters F, L and M in Figure 38 and Table 15 represent variables.

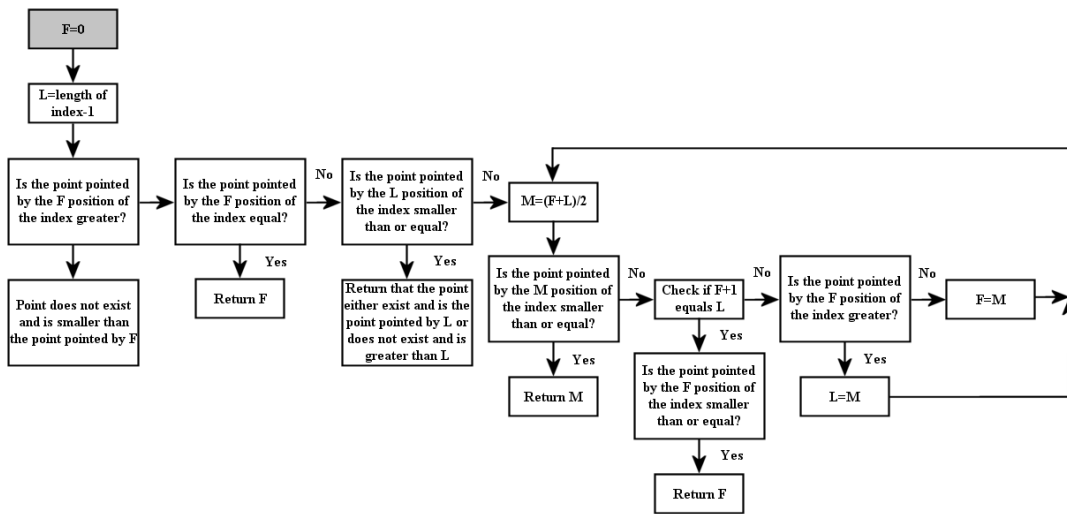


Figure 38. - Binary search algorithm block diagram.

Table 15. - Detailed binary search algorithm.

Task number	Next task on success	Next task on failure	Task description
1	2	-	Set F variable to zero.
2	3	-	Set L variable to length of index-1.
3	4	5	Check if the point pointed by the F position of the index is greater than the point being searched.
4	-	-	Return from function indicating that the point does not exist and that it is smaller than the currently pointed point.
5	6	7	Check if the point pointed by the F position of the index is equal to the point being searched.
6	-	-	Return from function indicating that the point is the currently pointed point.
7	8	9	Check if the point pointed by the L position of the index is smaller than or equal to the point being searched.
8	-	-	Return from function indicating that the point pointed either exist and it is the point pointed by the L position of the index, or does not exist and it is greater than the currently pointed point.
9	10	-	Set M variable to be the integer division of the sum of F and L by 2 ($M=(F+L)/2$).
10	11	12	Check if the point pointed by the M position of the index is smaller than or equal to the point being searched.
11	-	-	Return M.
12	13	15	Check if F+1 equals L.

13	14	18	Check if the point pointed by the F position of the index is smaller than or equal to the point being searched.
14	-	-	Return F.
15	16	17	Check if the point pointed by the F position of the index is greater than the point being searched.
16	9	-	Set L equal to M.
17	9	-	Set F equal to M.
18	-	-	Return error. The function should never get here because it always succeeds in one of the other earlier steps.

As one can notice by analysing the algorithm, the index search function either returns the position of the wanted point in the linked list, if it exists in the index, or it returns the position of the point indicated by the index that is immediately smaller than the wanted point.

Figure 39 presents the block diagram of execution of the algorithm implemented in this work to find points in the indexed ordered two-way linked list. Table 16 provides a more detailed description of the algorithm. The letter M in Table 16 represent a variable. Similarly to the binary search algorithm used to search the index, the algorithm used to find points either returns the position of the point, or it returns the position of the point immediately smaller than the wanted point.

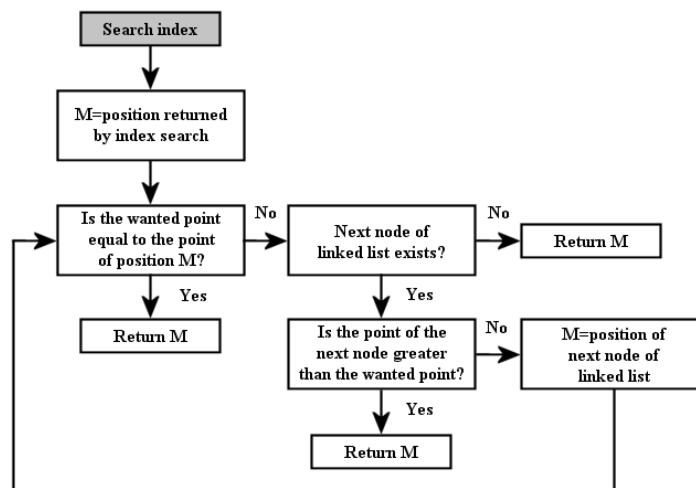


Figure 39. - Point search function execution block diagram.

Table 16. - Detailed list of tasks performed by the point search function.

Task number	Next task on success	Next task on failure	Task description
1	2	-	Search index for the wanted point.
2	3	-	Set M to the position of the linked list returned by the index search.
3	4	5	Check if the wanted point is equal to the point of the position M of the linked list.
4	-	-	Return the current position (M).
5	6	9	Check if there is a next node in the linked list.
6	7	8	Check if the point of the next node in the linked list is greater than the wanted point.
7	-	-	Return the current position (M) of current node (not the next node) of the linked list.
8	3	-	Set M to the position of the next node of the linked list.
9	-	-	Return the current position (M).

To insert a point, first a function similar to the point search function is executed that either returns the position of the point if it exists or returns the position where the point must be inserted if it does not exist. Then the point is either updated or inserted depending on the return of the point search function.

The size of the cluster of points inside each index entry of the indexed ordered two-way linked list has great influence in the speed of the algorithm. If this value is too small, finding and inserting points will be quick but the index array is resized too often. If this value is too big, finding and inserting points will take longer but the index array is less often resized. After some testing, 5120 was found to be the best value for the size of the cluster in the computer where the software robotic agent was executed.

3.6.4 Speeding Things Up

If the points have a small belief of being occupied, they can be regarded as being free. Since we are only saving those points that are not free, the points of small beliefs can be disregarded from being saved in memory. This saves memory and improves the overall speed of the program because, due to most points having small probabilities, there will be significantly less points to search when trying to determine if a certain point already exists. This tactic may pose a problem due to the fact that the unknown points, those that have not been measured yet, are also not in memory. How to distinguish the measured free points from the ones that are still not measured?

The answer is simple. For each cluster of points, the group of points measured without moving the robot, the position and posture of the robot from which the points were measured can be saved. This way, it is possible to know that every point inside the PTZ cone of action from the robot's position till the measured points is free. This is always true if one considers to be true the assumption that most space inside any imaginary cube where the robot can perform measures is free. As an example, inside an intact office delimited by four walls and equipped

with chairs and desks, most space will be free space because the chairs and the desks occupy only a small percentage of all the space between the four walls.

This optimization is specially important for the Bayesian method of calculating the probability of occupancy since the other method mentioned in this work, the dumb algorithm that performs the means of the probabilities of the points, does not save so many points due to the majority of them belonging to free space, hence not saving them.

3.6.1 Configuring

Before running the software robotic agent, it is necessary to configure some parameters through the command line. The configuration parameters have a great influence in the behaviour of the software robotic agent and, some, are mandatory so, the simulation will not run without them properly set. Table 17 lists all the parameters necessary, explains them and lists their acceptable values, when required. The default values of most parameters were the values used for most of the experiments.

Table 17. - Configuration parameters of the software robotic agent.

Parameter	Description	Acceptable values
Simulator IP	The IP address of the computer where the simulator is running.	Any dot separated IP address. Default is 127.0.0.1
Simulator Port	The port where the simulator is listening for connections.	Integer number between 1 and 65535. Default is 3000.
IR Noise	Maximum IR error value. This is used in the sensor model of the error to determine the region limits of the sensor model of uncertainty.	Any float value. Default is 0.0
Sonar Noise	Maximum sonar error value. This is used in the sensor model of the error to determine the region limits of the sensor model of uncertainty.	Any float value. Default is 0.0
Point Increment	Spacing between points in millimetres. This is used to determine the points for the cones.	Any positive integer greater than zero. Default is 10 (1cm).
Pan Increment	Horizontal increment of the PTZ in one of a thousandth of radians (1/1000).	Any positive integer greater than zero. Default is 10.
Tilt Increment	Vertical increment of the PTZ in one of a thousandth of radians (1/1000).	Any positive integer greater than zero. Default is 10.
Grid Type	Indicates if the grid point increment will be static with the value provided before or if it will be dynamic with an increment that depends on distance measured by the sensor.	true or false. Default is false.
Initial Pan	The initial horizontal angle (pan) of the PTZ in radians. If this value is larger than the maximum horizontal angle of the PTZ then the initial angle is set to the maximum horizontal angle.	Any signed integer value. Default is 2000000000.
Initial Tilt	The initial vertical angle (pan) of the PTZ in radians. If this value is larger than the maximum vertical angle of the PTZ then the initial angle is set to the maximum vertical angle.	Any signed integer value. Default is 2000000000.

Point Cluster	Size of the group of points inside each index entry of the ordered indexed two-way linked list.	Any positive integer value greater than zero. Default is 5120.
Algorithm	Chooses between one of three algorithms to calculate points. Available choices are the dumb algorithm, the Bayesian method algorithm and the Bayesian method with minimum belief.	0, 1 or 2. Default is 2.
Initial Belief	The initial belief of percentage of occupancy for the algorithms using the Bayesian method. Value is in per mille (tenths of percentage).	Any positive integer value between 0 and 1000. Default is 900.
Minimum Belief	The minimum belief of occupancy below which the points are not saved in memory. Value is in per mille (tenths of percentage).	Any positive integer value between 0 and 1000. Default is 300.
Start Location	Starting location for the robot. There are three pre-defined locations.	0, 1 or 2. Default is 0.

As can be noticed by analysing Table 17, there are three pre-defined locations as starting points for the robot. This was hard coded into the program to guarantee the repeatability of the experiments since, if the robot had to be led to those positions, it would probably not be able to match them exactly. Figures 40 to 42 show the positions 1,2 and 3 respectively.



Figure 40. - Pre-defined position 1 of the robot.



Figure 41. - Pre-defined position 2 of the robot.



Figure 42. - Pre-defined position 3 of the robot.

3.7 Map Viewer

To enable the visualization of the map generated by the robot, a Map Viewer was implemented in a Java based graphics programming language called Processing (13). This language makes the implementation of a purely graphical interface easy by providing an

easier programming language for a newbie to work on. Basically, each function of the Processing programming language is part of an abstraction layer that calls Java graphical functions. Figure 43 shows the Processing programming interface with an example that prints a cube and allows it to be rotated and translated.

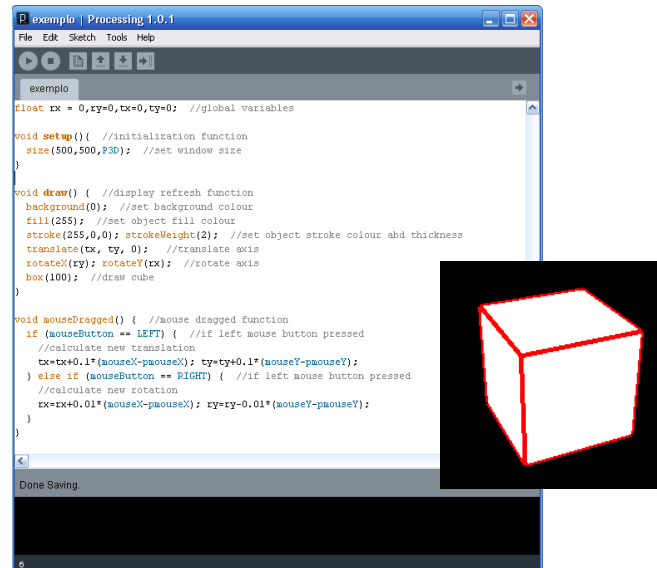


Figure 43. - Processing programming graphical interface with an example.

Processing has two main functions called "setup" and "draw". The "setup" function is used for preparing to display the graphics and involves tasks like setting the canvas size. The "draw" function is the equivalent to the main function in Java and is used to continually refresh the display. This last function has the particularity that it executes continually and does not end when its last instruction is executed. The tasks executed by the Map Viewer to display a point cloud are listed in Table 18 and 19 for the "setup" and "draw" functions respectively.

Table 18. - List of tasks executed by the "setup" function of the Map Viewer.

Task number	Description
1	Define canvas size and graphics engine as P3D.
2	Define background colour
3	Load font type and define the text mode.
4	Open file with points to display and load them to memory.
5	Calculate current translation of the point cloud to display it on canvas since the first time it is drawn.
6	Calculate maximum and minimum values of the window that constraints the points to display to prevent the refresh rate to become too slow.

Table 19. - List of tasks executed by the "draw" function of the Map Viewer.

Task number	Next task on success	Next task on failure	Description
1	2	-	Set the background colour to erase the previous points displayed
2	3	-	Set currently selected translations and rotations of the axys
3	4	-	For all points in memory
4	5	3	Check if the point is inside the window of points to display
5	6	3	Check if the probability of the point corresponds to an unfiltered probability.
6	7	8	Check if colours are set to gray scale.
7	9	-	Define the probability of the point as the colour to display.
8	9	-	Define one of the four colours to display depending on the probability.
9	10	11	Check if the accessibility flag is true
10	12	-	Display the point as a square with half centimetre of side.
11	12	-	Display the point as a point
12	13	3	Check if all points where displayed.
13	1	-	Display menu.

The Map Viewer has an interface like the one in Figure 44 and possesses the functionalities listed in Table 20. The map is printed on the interface in 3D with the colour of each point corresponding to its probability of being occupied. The darker the colour the lower is the probability, when the colour scheme is grey scale. If the colour scheme is four colour ranges, the probability of occupancy is equally divided by four and increases from blue (cold) to red (warm) colours. The order is blue<green<yellow<red. The background black colour is an indication of unoccupied or unmapped points.

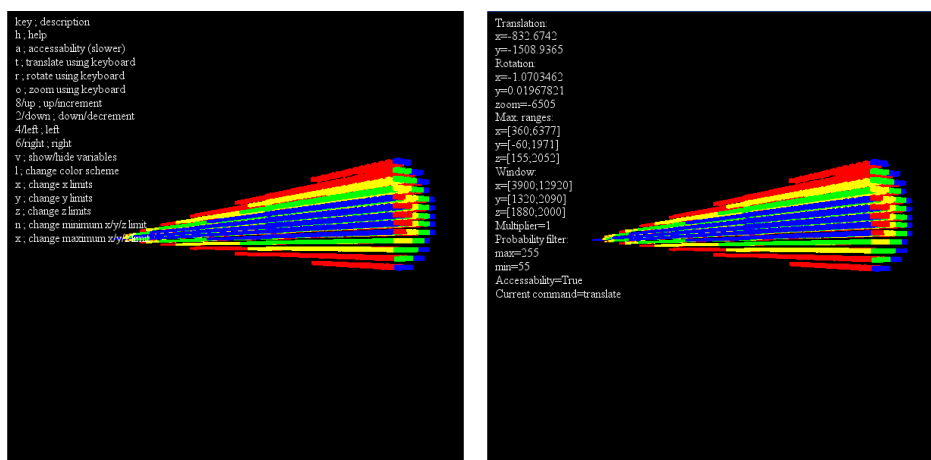


Figure 44. - Map Viewer interface with a sonar cone printed. The left one shows the help menu and the right one the values of the internal state.

Table 20. - Functionalities of the Map Viewer and their respective descriptions.

Function	Key sequence	Description
Help	h	Shows or hides the help.
Accessibility	a	When off all points in the interface are printed as dots and when on all points are printed as squares.
Up/increment	UP or 8	Up command. Rotates or translates up, or increments the currently selected value.
Down/decrement	DOWN or 2	Down command. Rotates or translates down, or decrements the currently selected value.
Left	LEFT or 4	Left command. Rotates or translates left.
Right	RIGHT or 6	Right command. Rotates or translates right.
Maximum	x	Commands that the maximum of the currently selected command is to be changed.
Minimum	n	Commands that the minimum of the currently selected command is to be changed.
Translate	t	Translates all points. It must be followed by an up/down/left/right command to execute the intended action.
Rotate	r	Rotates all points. It must be followed by an up/down/left/right command to execute the intended action.
Zoom	o	Zooms in or out. It must be followed by an up/down command to execute the intended action.
Variables	v	Shows or hides the current variables values.
Colour Scheme	l	Changes the colours of the points between continuous grey scale and four colour ranges.
x/y/z limits: maximum limit	x/y/z x	Change the maximum x/y/z value range of the points to print. Points with values greater than this maximum are not printed. This is useful to limit the points to print to a determinate range of values. Must be followed by an up/down command to set a new value.
x/y/z limits: minimum limit	x/y/z n	Change the minimum x/y/z value range of the points to print. Points with values less than this minimum are not printed. This is useful to limit the points to print to a determinate range of values. Must be followed by an up/down command to set a new value.
Probability filter: maximum probability	p x	Change the maximum probability range of the points to print. Points with probabilities greater than this maximum are not printed. This is useful to limit the points to print to a determinate range of probabilities. Must be followed by an up/down command to set a new value.
Probability filter: minimum probability	p n	Change the minimum probability range of the points to print. Points with probabilities less than this minimum are not printed. This is useful to limit the points to print to a determinate range of probabilities. Must be followed by an up/down command to set a new value.

Annex D presents figures that visually display the effects of some of the functionalities of the Map Viewer's interface.

3.8 Post-Process for Result Analysis

Because most points will not match, due to errors in measurements using non-ideal sensors, a method was devised to create the missing points in a manner that the calculated points would not diverge much from the values of probability they would have if they had been measured. That method is to calculate the value of a missing point by performing a weighted mean of the closest points surrounding him. Formula 5 presents the equation used to calculate the weighted mean.

$$M = \frac{\sum_{i=1}^n p_i * v_i}{\sum_{i=1}^n p_i} \quad (5)$$

Where M is the mean value resulting from the calculation, n is the number of points, v_i is the value of the current point and p_i is the weight attributed to the current point. In this formula, the weight attributed to a point was calculated as being the distance of the current point to the missing point, subtracted to the maximum distance of all points being used to the missing point, as presented in Formula 6.

$$p_i = \max(d_1, d_2, \dots, d_n) - d_i \quad (6)$$

Performing the subtraction to the maximum ensures that points further away from the missing point have smaller influence on the result. Figure 45 presents the block diagram of the post-processing algorithm that performs the creation of the missing points required for the statistical analysis.

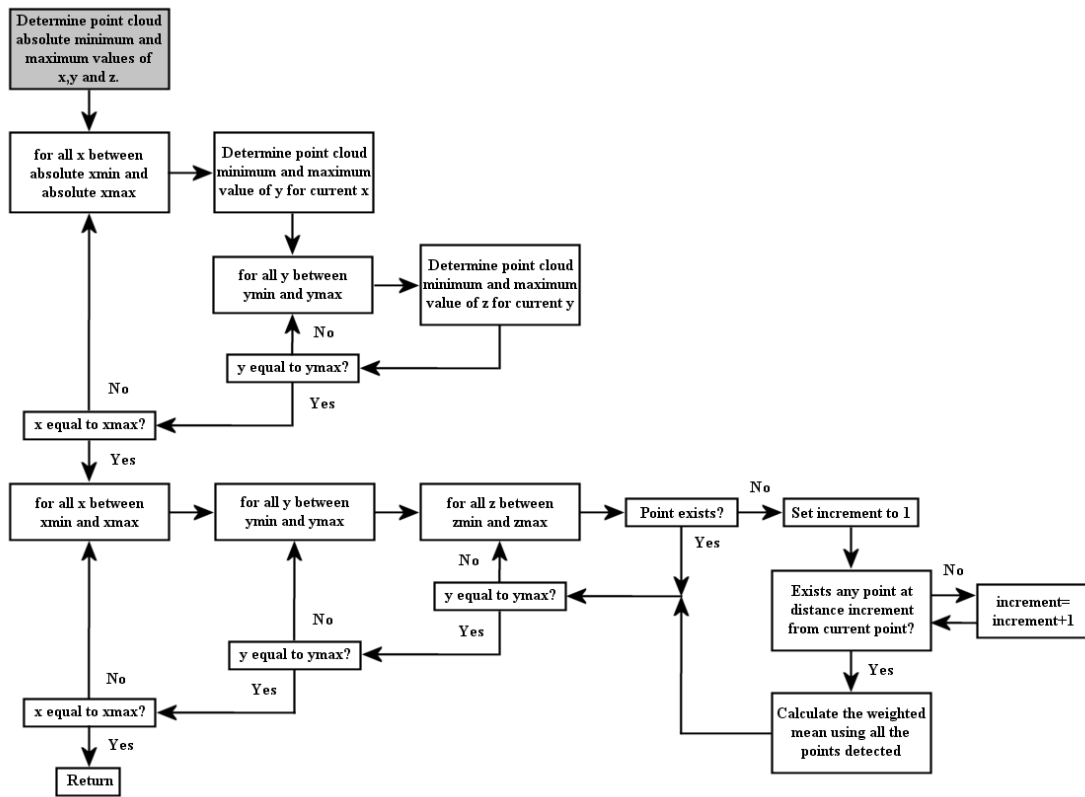


Figure 45. - Block diagram of the post-processing algorithm.

3.9 Calculating Results Accuracy

3.9.1 Statistical Comparison

In order to test the accuracy of the measures, a set of points is statistically compared with a reference set of points through the comparison of some statistical values which are calculated. The statistical values chosen to be used as comparison were the mean of the absolute error of the probability of the entire set of points and the standard deviation of the absolute error of the probability of the entire set. The closer to zero these values are, the more accurate the measured set of points is.

For each point in the target point cloud, the same point is retrieved from the reference point cloud and the absolute value of the difference of the probabilities is calculated, obtaining the absolute error of the probability. After calculating the absolute error, the mean value and the standard deviation value of the absolute error can be calculated using the usual formulas for the mean and standard deviation.

By performing these calculations for all the intended point clouds, one can compare the values of the different means of the absolute error and the different standard deviations of the absolute error to have an idea of how accurate is each one of the point clouds relative to the other.

3.9.2 Reference Point Clouds

The required reference set of points was generated by performing measures of the scenario with ideal sensors, meaning sensors without noise, and using the Bayesian method of building the cones. But this is not enough since the generation of the cones creates lots of points that are known to be free if a visual observation of the map is performed. For this reason, after being measured, the reference set of points was filtered to keep only those points that had a probability of occupancy greater than 90%. Figures 46 and 47 respectively presents the set of points resulting from the IR and sonar measures for position 1 of the robot.

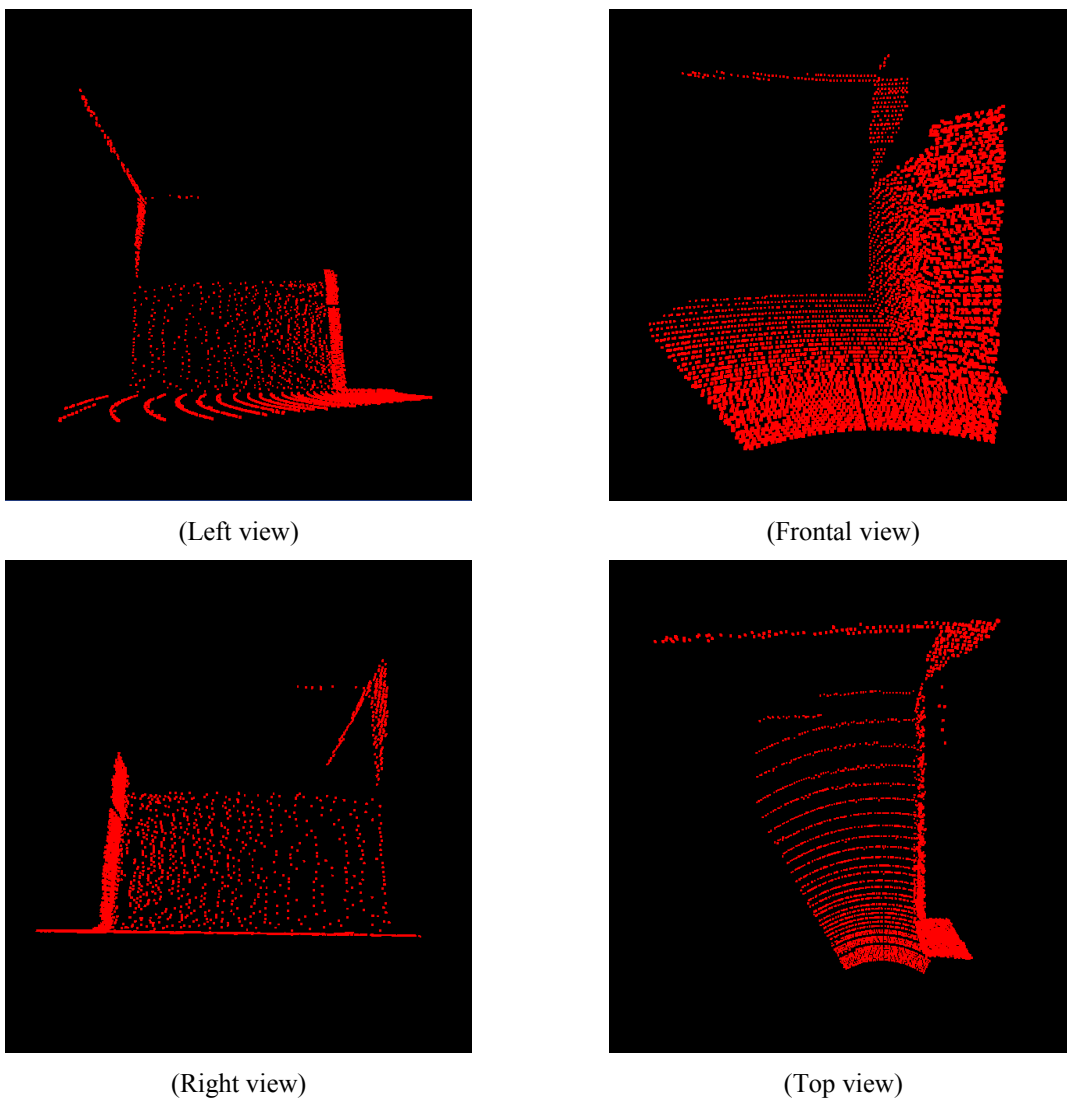


Figure 46. - Map generated from measurements of the IR sensor.

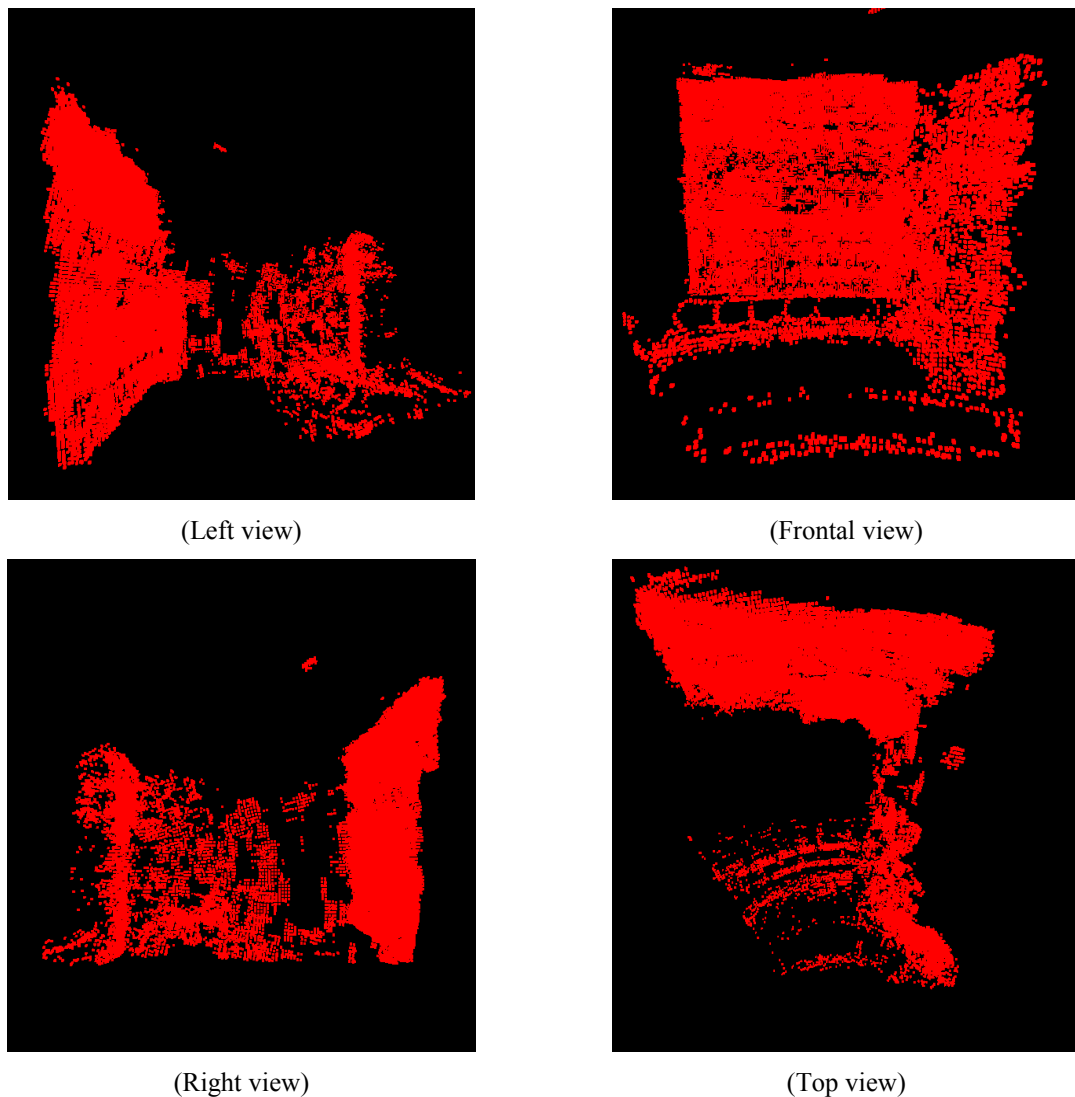


Figure 47. - Map generated from measurements of the sonar sensor.

As can be observed in Figure 47, the sonar does not return good results to be usable as a reference so only the measurements obtained through the IR were used to generate the reference set of points.

After having the set of points, the probabilities of the remaining, unfiltered, points were updated to read 100% occupancy and the processing described in chapter 3.8 is applied. Since the cones, with measures taken by ideal sensors, present very abrupt transitions in probabilities between occupied and free space, this method will produce a very accurate map to be used as reference. Figure 48 presents the final appearance of the reference set of points for position 1 of the robot.

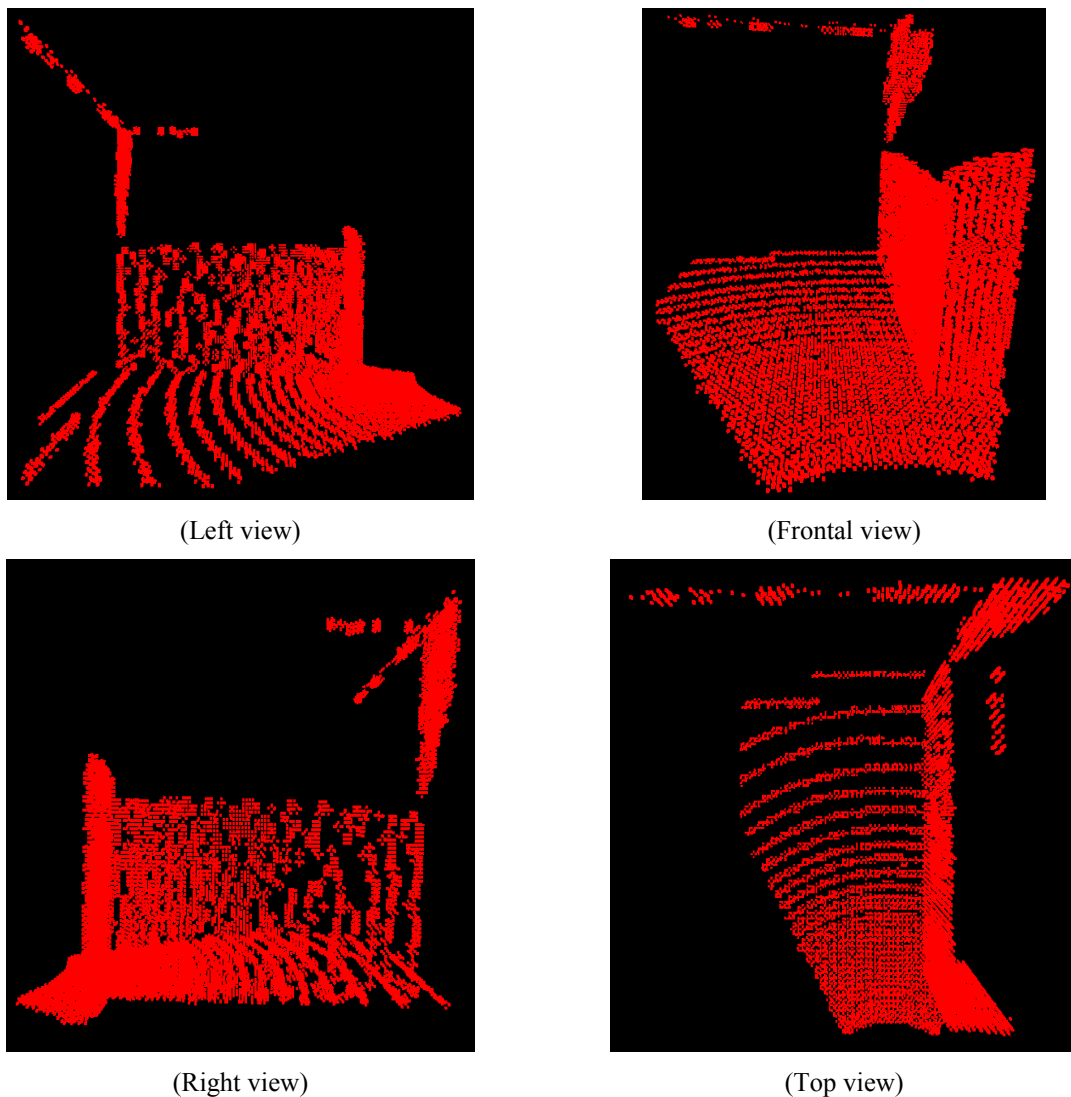


Figure 48. - Reference set of points for location 1.

3.10 Summary

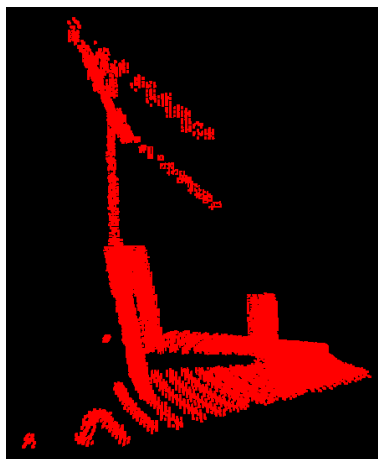
This chapter described in detail the implementation of the work. It covers the initial definition of constraints that the robot's design must comply with for the successful accomplishment of the proposed objectives for this work. Also covers the desired and the effectively used robot physical construction in USARSim and how to create and configure it, the robot hardware and software architectures, including how the emulation of its hardware and software were performed. Also describes the sensors and sensor models, the software robotic agent, an auxiliary post-processing function to create missing points and the Map Viewer, which is the program built to display the point clouds.

4 Results

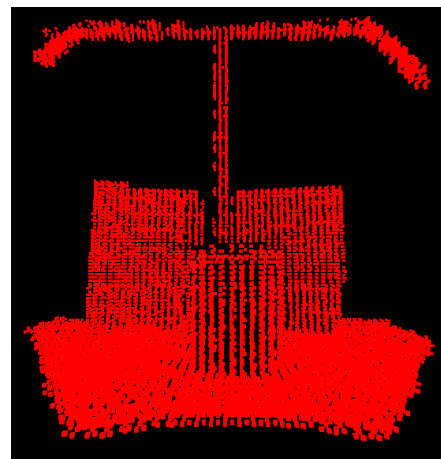
This chapter is devoted to exposing the results of each set of measurements, the processing they suffered and to the analysis of the results obtained by the statistical comparison of each of those measurements.

4.1 Perfect sensors

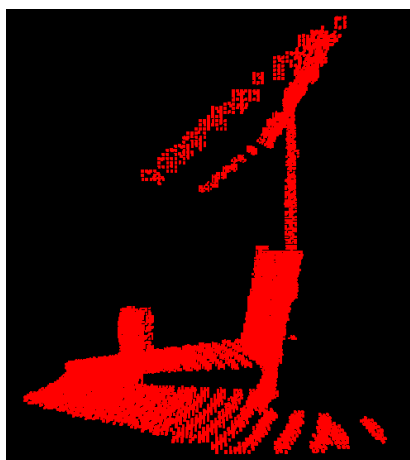
Similarly to what is described in chapter 3.8 for position 1 of the robot, the point clouds of positions 2 and 3 are submitted to the same processing resulting in the reference sets of points of Figures 49 and 50.



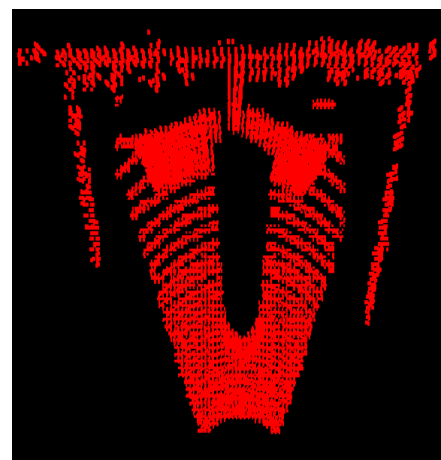
(Left view)



(Frontal view)



(Right view)



(Top view)

Figure 49. - Reference set of points for position 2 of the robot.

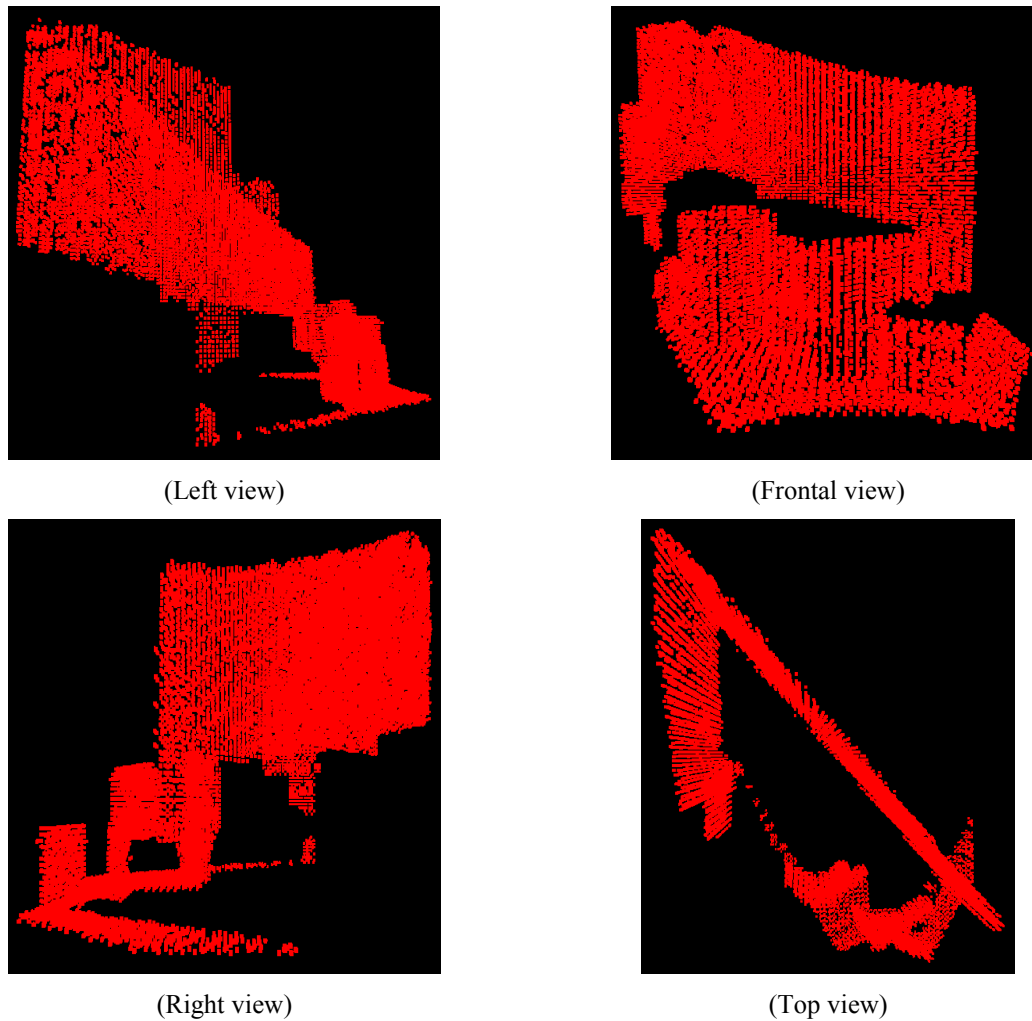


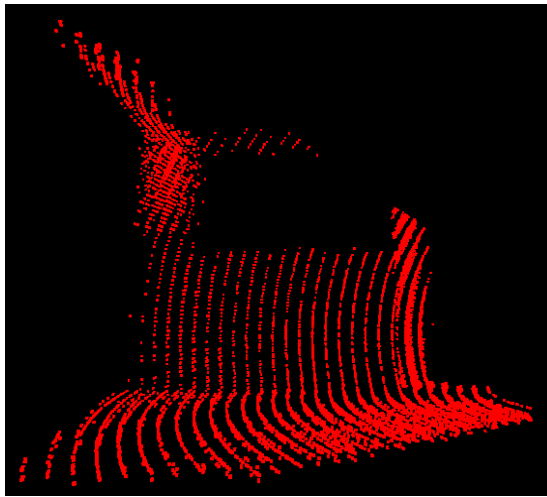
Figure 50. - Reference set of points for position 3 of the robot.

If one recalls the figures that presented the positions of the robot (Figures 40 to 42) and compares them with the point clouds presented in Figures 47, 49 and 50, one can conclude that something is wrong. The person that shows in Figures 40 to 42 does not show in Figures 47, 49 and 50. It appears that the simulator, for some reason, only displays people to cameras but, for the rest of the sensors, they are only cylinders.

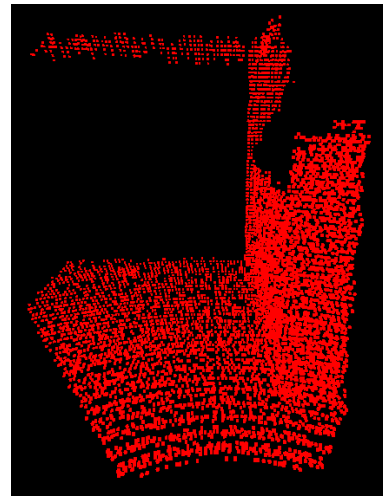
4.2 Non-ideal sensors

Now that there are a set of references, the actual experiments with non-ideal sensors can be performed. After obtaining the measurements, the point clouds were filtered to remove all probabilities below 90% because the remaining 10% of the points are the most interesting ones since they represent the actual obstacles as can be observed in Figures 51 to 53. In order not to clutter this document only the figures with the IR measures of each position are presented, since that with the sonar is a little harder to distinguish the objects. Also, since visually the difference between both fusion methods is imperceptible in small static figures,

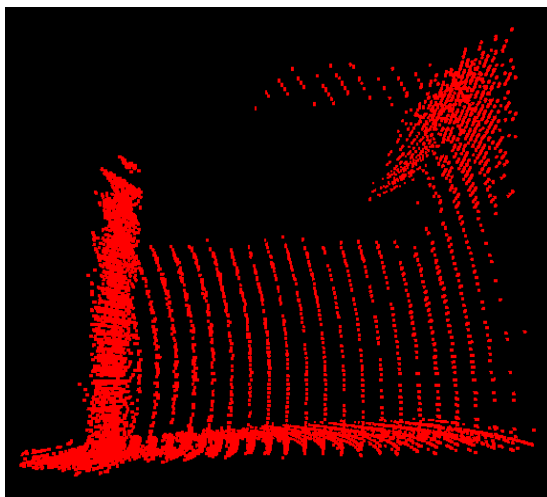
only the figures belonging to the Bayesian method are presented.



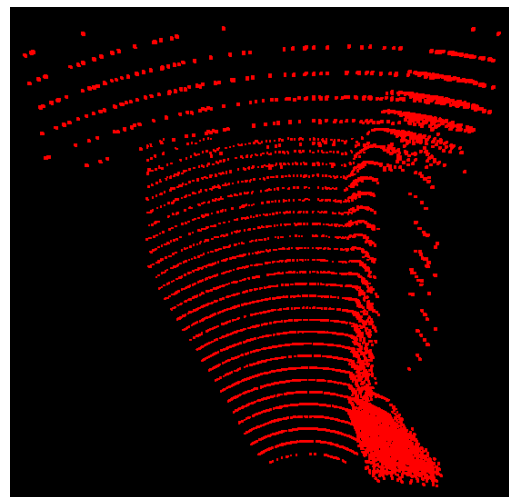
(Left view)



(Frontal view)

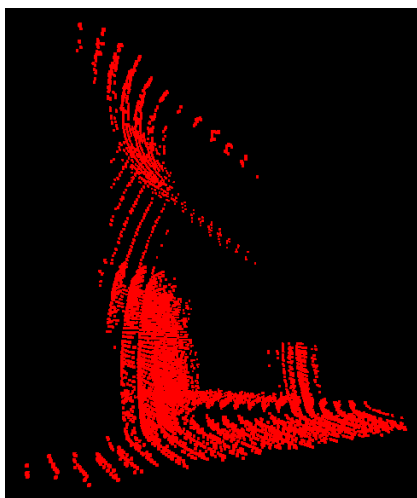


(Right view)

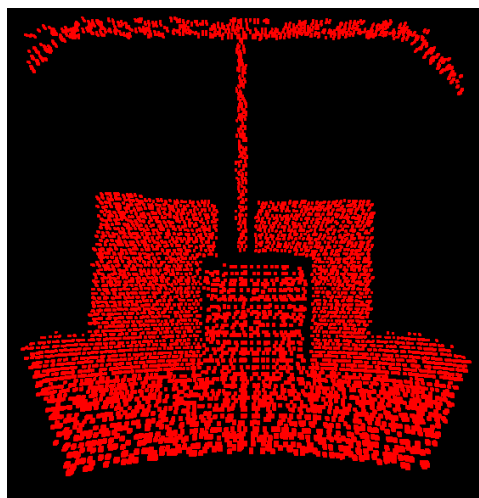


(Top view)

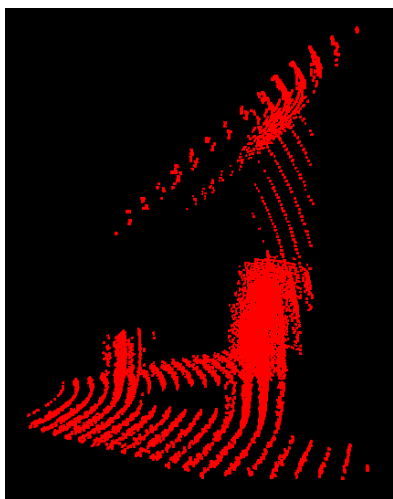
Figure 51. - IR cloud points for position 1 obtained using the Bayesian method.



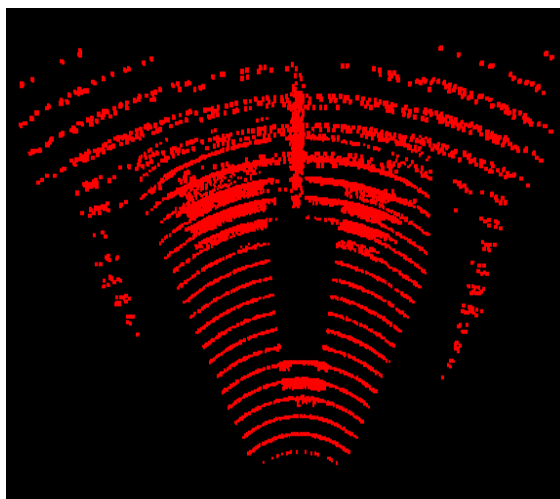
(Left view)



(Frontal view)



(Right view)



(Top view)

Figure 52. - IR cloud points for position 2 obtained using the Bayesian method.

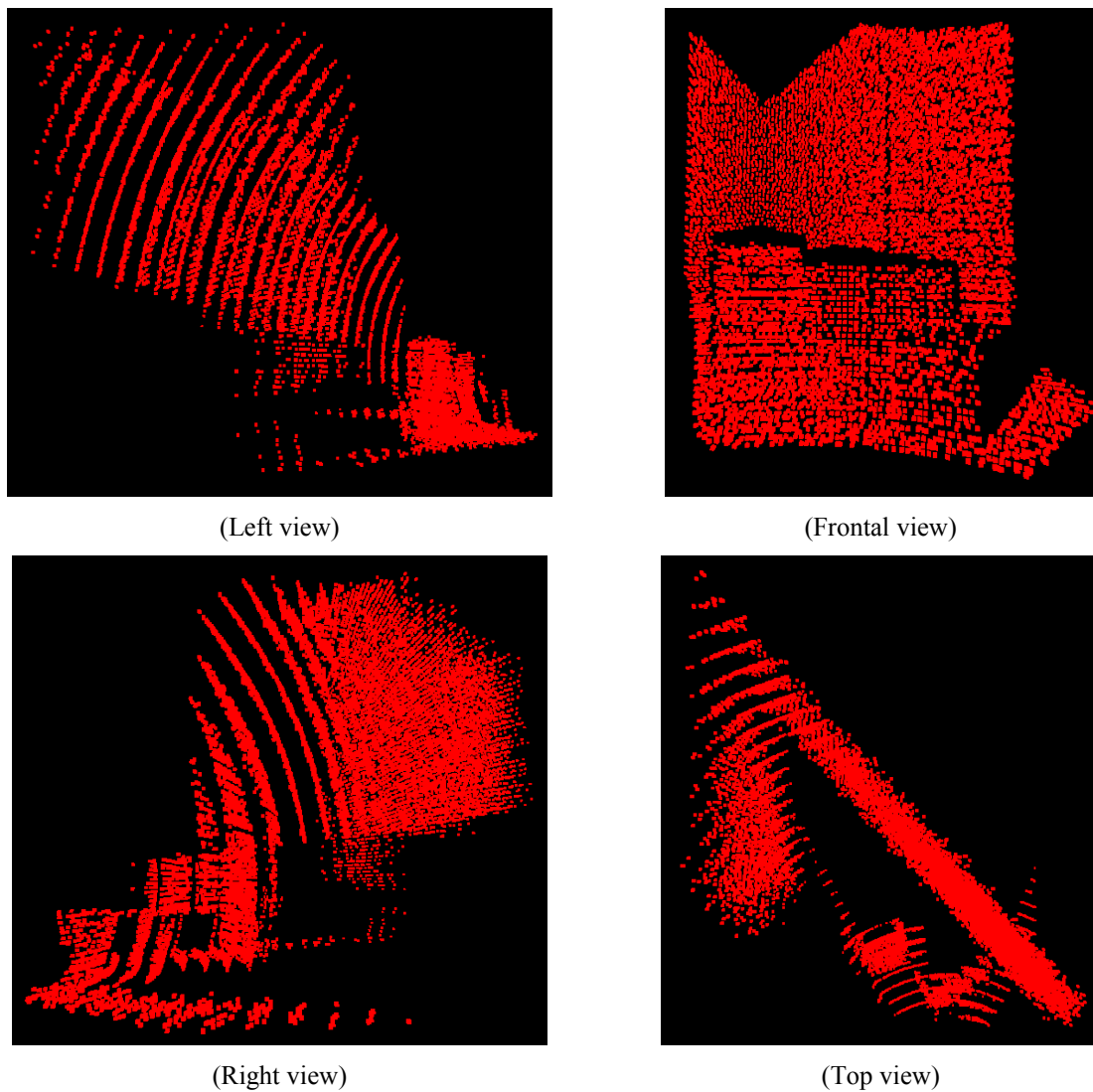


Figure 53. - IR cloud points for position 3 obtained using the Bayesian method.

Following the filtering, the processing described in chapter 3.8 was applied to each of the point clouds. The resulting sets were then statistically compared with the reference point clouds using the algorithm described in chapter 3.9.1 and their corresponding statistics are presented in Tables 21 to 23 in percentage.

Table 21. - Statistics for position 1.

Algorithms	Absolute error mean			Absolute error standard deviation		
	IR	Sonar	Fusion	IR	Sonar	Fusion
Dumb	15.28	30.65	26.78	19.27	26.27	23.73
Bayesian	14.45	28.19	24.36	19.05	25.35	22.63

Table 22. - Statistics for position 2.

Algorithms	Absolute error mean			Absolute error standard deviation		
	IR	Sonar	Fusion	IR	Sonar	Fusion
Dumb	17.14	28.68	18.89	20.99	25.15	21.80
Bayesian	29.04	23.60	29.38	25.92	22.34	25.50

Table 23. - Statistics for position 3.

Algorithms	Absolute error mean			Absolute error standard deviation		
	IR	Sonar	Fusion	IR	Sonar	Fusion
Dumb	32.25	36.21	40.19	27.47	28.37	28.40
Bayesian	40.19	45.92	50.64	24.35	23.98	25.03

4.3 Results Analysis

As expected, by analysing Tables 21 to 23, the sonar yields worst accuracy than the IR, probably due to the angle of the cone being so large, when compared with the one of the IR, disallowing it from having good accuracy. As a consequence of the sonar's accuracy and the fusion methodology implemented, the fusion of both IR and sonar sensor measures also yields worst accuracy than the IR alone. This results should not discourage the usage of sensor fusion since, with it, it is possible to obtain extra information, like the presence of transparent surfaces, that is not possible to obtain with the IR alone. In fact, the analysis of those tables indicates that the sonar has a standard deviation smaller than the IR for scenarios where the majority of the objects is closer (do not forget that position 2 has a glass in front of the robot). This means that, although the mean of the absolute error is larger, the maximum deviation a measure can have is smaller.

Compiling the results of Tables 21 to 23 and determining the difference between the values obtained with Dumb and the Bayesian methods, Table 24 is obtained. In Table 24, a positive number indicates that the Bayesian method is more accurate than the Dumb method while a negative number represents the opposite.

Table 24. - Difference between the values of Dumb and Bayesian methods.

Position	Absolute error mean			Absolute error standard deviation		
	IR	Sonar	Fusion	IR	Sonar	Fusion
1	0.83	2.46	2.43	0.22	0.92	1.10
2	-11.89	5.08	-10.49	-4.92	2.81	-3.70
3	-7.93	-9.70	-10.44	3.12	4.39	3.38

Surprisingly, by analysing Table 24, the dumb method provided the best results for most of the IR measures. This could be due to the fact that the Bayesian method was originally intended for the sonar and the adaptation of it, performed in this work and explained in chapter 3.4.2, to apply this method to the IR is not the best.

Notice that the difference between the accuracy of the Bayesian method and the dumb method implemented in this work is not as significant as one might initially think. Even so, one must keep in mind that this is a statistical comparison that does not analyse the point clouds in detail and that, for the dumb method to reach the accuracy of the Bayesian method, more scans from the same position would be required because the mean function tends slowly towards the correct probabilities.

4.4 Minimum Working Frequency

Now that it has been demonstrated that the architecture presented in this document works, it is time to demonstrate that it is also fast enough for a relatively slow, and hence cheap, microprocessor. In order to do that, the instruction count is performed to get the minimum working frequency. Remember that it was previously defined that every instruction takes only one clock cycle to execute.

The instruction count was performed using two methods. The first method was to manually analyse the algorithm and count the number of instructions for the worst case scenario. The second method was to add a global variable to the program and increment it before each instruction in order to obtain a more accurate measure than the one obtained using the manual method.

Before presenting the results for both methods, the conditions for the worst case scenario must be presented. Those conditions and their respective descriptions are presented in Table 25.

Table 25. - Worst case scenario conditions.

Condition	Description
Insertion/searching the point list takes as long as possible.	All points are stored in an indexed ordered two-way linked list and, as such, the maximum number of positions of the list that must be read in order to find the right one is equal to the number of positions between two consecutive indexes. This is related to the pointsCluster parameter passed to the robot program.
All objects are detected at the sensor maximum range	The number of points in the cones obtained by applying the sensor model is maximum when the distance measured is equal to the maximum sensor range.
Points have maximum precision	Because the space is discrete in the robot's memory, the point precision is basically the spacing between adjacent points which has implications on the number of points inside each cone. This is accomplished by minimizing the horizontal (pan) and vertical (tilt) distances the PTZ has to move, and minimizing the depth distance between points (distance measured relative to the sensor). This is related with the pointInc, panInc and tiltInc parameters passed to the robot program.

Analysing the code and performing calculations bearing in mind the above conditions (the first method), one reaches the conclusion that the number of instructions to calculate one cone is 1073570686056 which, at one instruction per clock cycle would give a frequency of approximately 1074GHz. This number is obviously impossible since the simulation was run at almost real-time in a Core2Duo processor at 2GHZ which was far from being dedicated exclusively to the simulation.

Performing measurements using the second method, one gets the much more attractive number of 180864261 instructions which gives a frequency of approximately 181MHz. This huge difference in frequencies is mainly due to the fact that the worst case described in Table 25 does not happen in the calculation of every point. Another reason is that the distances between points inside a cone are not static but dynamic and are a function of the distance. They decrease for points closer to the distance measured by the sensor. The further one is from the distance measured by the sensor, the larger are the distances between points inside the cone. This implies that memory is spent mainly to store the most interesting points, which are those belonging to the actually occupied areas.

4.5 Summary

This chapter exposes and analyses the results of experiments performed with the robotic architecture and the 3D mapping methodology described in this document. Here, the point clouds for the ideal and non-ideal sensors are presented, as well as the results of the statistical comparison between fusion methods and the instruction count to find the minimum working frequency.

5 Conclusions and Future Work

Due to the generation of an accurate map being so important, investigators are always experimenting new sensors or new algorithms that are able to improve the accuracy of the generated maps. Sensors are getting smaller, lighter, more energy efficient and more accurate. Some sensors already return a 3D map of scenario in front of it without the intervention of any moving part, like the SR4000 described in this document.

There is also a concern to try and make the algorithms more light in order to require less computational power and save battery without sacrificing its purpose. Sometimes, completely new algorithms are designed to provide new functionality or the same functionality than another algorithm but more efficiently.

In this thesis, a methodology for mapping is described that generates accurate 3D maps of a disaster scenario. It is composed of a methodology to obtain measurements, a hardware platform and a group of algorithms that, in conjunction, complement each other. Steps are provided to guide through the construction of the robot's hardware and software. A list of constraints aids in the optimization of the robot to achieve its full potential.

The most basic range sensors are used and it is proven by the results that their performance and accuracy is good although the speed of generation of the map is a little slow because of the amount of points required to measure and the scanning method adopted. Anyway, this could be minimized by having more than one robot performing measurements in different parts of the scenario and sharing each others maps.

There is still some work to be done in the future to make this work usable in a real search and rescue scenario. A list of some of that work is presented in Table 26.

Table 26. - List of future work

Task number	List of future work
1	Optimize the algorithms used by the robotic agent.
2	Increase the speed of the point search algorithm by inserting another index level.
3	Implement a better fusion algorithm.
4	Implement the robot as described in chapter 3.3.1 Desired architecture.
5	Add a second PTZ to the back of the robot's body to perform 360° maps in one scan.
6	Add an object detection algorithm that enables to transform the point clouds into defined shapes.
7	Implement a real version of the robot to test it in real life.

One of the task of the future work suggested is the insertion of another index level (Task

number 2). The idea would be to further divide the effective amount of points indexed by each position of the currently implemented index in order to diminish the amount of time spent by the point search function so that the generation of a cone takes less time. Basically, a second index would be inserted between each position of the currently implemented index and the ordered linked list such that, each of the positions of this second index would point for an ordered linked list of, lest say, 16 points instead of the 5120 used in this work.

Another important future work task suggested is the implementation of an object detection algorithm (Task number 6) in order to enable the robot to store objects in memory instead of points, possibly saving memory since an object might be defined by a large set of points that might all fall over the same surface. The simplest method to implement the object detection is to follow the same reasoning that was applied to extract the reference point clouds. That is, filter out the points with a probability smaller than 90%, to use only the points with probabilities within the higher 10%, assume that those points have a probability of 100%, flatten the points to a plane and, finally, calculate the edges of the objects using any image edge detection algorithm like the one described in (19).

References

1. Stahn, R.; Heiserich, G.; Stopp, A., Laser Scanner-Based Navigation for Commercial Vehicles, Intelligent Vehicles Symposium, 2007 IEEE, 13-15 June 2007 Page(s):969 - 974
2. Virtual Robots Competition of the RoboCup Rescue , online, available at: <http://www.robocuprescue.org/virtualsim.html> [consulted on December 2008]
3. Epic Games , online, available at: <http://www.epicgames.com/> [consulted on December 2008]
4. Unreal Engine 2, online, available at: <http://www.unrealtechnology.com/html/technology/ue2.shtml> [consulted on December 2008]
5. MathEngine Karma User Guide , online, available at: udn.epicgames.com/Two/rsrc/Two/KarmaReference/KarmaUserGuide.pdf [consulted on December 2008]
6. Carpin, S., Lewis, M., Wang, J., Balakirsky, S., Scrapper, C.: USARSim: a robot simulator for research and education. IEEE International Conference on Robotics and Automation (2007)
7. Unreal Gamebots, online, available at: <http://gamebots.sourceforge.net/> [consulted on December 2008]
8. Unreal Tournament 2004 patch to version 3369 , online, available at: <http://www.beyondunreal.com/main/ut2004/ut2004essential.php> [consulted on December 2008]
9. USARSim project page, online, available at: http://sourceforge.net/project/showfiles.php?group_id=145394 [consulted on December 2008]
10. Murphy R. : Introduction to AI Robotics. The MIT Press, Massachusetts (2000)
11. Iocchi, L., Konolige, K., Bajracharya, M.: Visually Realistic Mapping of a Planar Environment with Stereo. Proc. of Seventh International Symposium on Experimental Robotics (2000)
12. Dellaert, F., Fox, D., Burgard, W., and Thrun, S.: Monte Carlo Localization for Mobile Robots, IEEE International Conference on Robotics and Automation (ICRA), 1999
13. Processing , online, available at: <http://www.processing.org/> [consulted on December 2008]
14. Kurt3D web page, online, available at: <http://www.inf.uos.de/kbs/KURT3D.html> [consulted on December 2008]
15. Jüngel, M., Risler, M.: Self-localization Using Odometry and Horizontal Bearings to Landmarks. RoboCup 2007: Robot Soccer World Cup XI, July 9-10, 2007, Page(s) 393-400
16. Sun, D., Kleiner, A., Wendt, T.: Multi-Robot Range-Only SLAM by Active Sensor Nodes for Urban Search and Rescue. Robocup 2008: Robot Soccer World Cup XII, 2008
17. Becker, D., Risler, M.: Mutual Localization in a Team of Autonomous Robots using Acoustic Robot Detection. RoboCup International Symposium 2008, Suzhou, China, July 15-18, 2008
18. SR4000 sensor, online, available at: <http://www.mesa-imaging.ch/prodview4k.php> [consulted on December 2008]

19. Wulf, O., Brenneke, C., Wagner, B.: Colored 2D Maps for Robot Navigation with 3D Sensor Data. Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, September 28 -October 2, 2004, Sendai, Japan
20. Millnert, O., Goedemé, T., Tuytelaars, T., Van Gool, L.: Range Determination For Mobile Robots Using One Omnidirectional Camera. International Conference on Informatics in Control, Automation and Robotics, ICINCO 2006, 1-5 August 2006, Setubal, Portugal.

Annexes

Annex A Detailed Objectives

In this annex, a much more detailed list of the objectives for this work is presented. Also, the dependencies between objectives and their sub-objectives is defined, meaning that to complete some objectives, their corresponding sub-objectives must first be completed (Example: To fulfil objective 3, all 3.x sub-objectives must be fulfilled). Table 27 presents the aforementioned list.

Table 27. - Detailed list of objectives, sub-objectives and their corresponding descriptions.

Objective number	Objective	Sub-objective	Description
1	x		Define the work to be realized during the Master Thesis.
2	x		Gather information about USARSim.
2.1		x	Read USARSim documentation to familiarize with the simulator.
2.2		x	Find additional information about USARSim on the web.
3	x		Install USARSim platform.
3.1		x	Install Unreal Tournament 2004.
3.2		x	Install Unreal Tournament 2004 patch for version 3369 if required.
3.3		x	Install USARSim files.
3.4		x	Install USARSim base map files.
3.5		x	Install USARSim arenas.
4	x		Try an example program, with a pre-built robot, that connects to USARSim to better understand the simulator.
5	x		Create a robot that conforms to the intended characteristics in this work.
6	x		Define the software architecture of the robotic software agent.
7	x		Create the robotic software agent.
7.1		x	Create a program that connects to the simulator.
7.2		x	Add functionality to create a pre-built robot and receive messages of its sensors.
7.3		x	Add functionality to send actuator messages to make the robot move.
7.4		x	Add a message parser to parse incoming messages to extract sensor and robot information and put it into variables.
7.5		x	Add an abstraction layer to hide sensors and actuators implementation and treat them as memory positions like in a real robot.
7.6		x	Add functions to change the referential from being relative to the part to being absolute to the map referential.
7.7		x	Add functions to change between cartesian and polar coordinates.
7.8		x	Add functionality to store coordinates in memory to save the points measured and calculated.

7.9		x	Add the sensors models and their corresponding probability functions.
7.10		x	Test the robot and calculate a valid set of points from the values returned by the sensors.
7.11		x	Add sensor fusion functionality.
7.12		x	Add functionality to scan automatically with the PTZ.
7.13		x	Add functionality to move the robot manually without interfering with the measurements.
8	x		Perform tests with several combinations of sensor models and sensor fusion algorithms.
9	x		Analyse results.
10	x		Draw conclusions.

Annex B CPU Instructions

This annex presents in Table 28 the full list of instructions created for the emulated CPU used in this work. It also presents syntax/examples and a detailed description of each instruction. In this work all function calls use global variables so there are no parameters when calling functions.

Table 28. - List of instructions of the emulated CPU, their syntax, usage and descriptions.

Instruction	Instruction type	Parameters	Syntax/Example	Description
function header	A		f0:	Indicates the starting point of the function.
call function	A	f0	call f0	Calls function f0.
return from function	A		return	Returns from the function it belongs to. Does not apply outside a function.
if zero call function	A	v0, f0	ifz1 v0 f0	If the value of the variable v0 is zero/false call function f0 else continue with program.
if not zero call function	A	v0, f0	ifnz1 v0 f0	If the value of the variable v0 is not zero/true call function f0 else continue with program.
while zero call function	A	v0, f0	wz1 v0 f0	While the value of the variable v0 is zero/false call function f0 else continue with program.
while not zero call function	A	v0, f0	wnz1 v0 f0	While the value of the variable v0 is not zero/true call function f0 else continue with program.
if zero call one function else call another	A	v0, f0, f1	ifz2 r f0 f1	If the value of the variable v0 is zero/false call function f0 else call function f1.
if not zero call one function else call another	A	v0, f0, f1	ifnz2 v0 f0 f1	If the value of the variable v0 is not zero/true call function f0 else call function f1.
if zero return	A	v0	ifzr v0	Used inside a function to return from it when the value of the variable v0 is zero/false.
if not zero return	A	v0	ifnzr v0	Used inside a function to return from it when the value of the variable v0 is not zero/true.
if greater than call function	A	v0, v1, f0	ifg1 v0 v1 f0	If the value of the variable v0 is greater than the value of the variable v1 then call function f0.
if less than call function	A	v0, v1, f0	ifl1 v0 v1 f0	If the value of the variable v0 is less than the value of the variable v1 then call function f0.
if greater than	A	v0, v1, f0	ifge1 v0 v1 f0	If the value of the variable v0 is greater than or equal

or equal call function				to the value of the variable v1 then call function f0.
if less than or equal call function	A	v0, v1, f0	ifle1 v0 v1 f0	If the value of the variable v0 is less than or equal to the value of the variable v1 then call function f0.
if greater than call function else call another	A	v0, v1, f0, f1	ifg2 v0 v1 f0 f1	If the value of the variable v0 is greater than the value of the variable v1 then call function f0 else call function f1.
if less than call function else call another	A	v0, v1, f0, f1	ifl2 v0 v1 f0 f1	If the value of the variable v0 is less than the value of the variable v1 then call function f0 else call function f1.
if greater than or equal call function else call another	A	v0, v1, f0, f1	ifge2 v0 v1 f0 f1	If the value of the variable v0 is greater than or equal to the value of the variable v1 then call function f0 else call function f1.
if less than or equal call function else call another	A	v0, v1, f0, f1	ifle2 v0 v1 f0 f1	If the value of the variable v0 is less than or equal to the value of the variable v1 then call function f0 else call function f1.
if greater than return	A	v0, v1	ifgr v0 v1	Used inside a function to return from it when the value of the variable v0 is greater than the value of the variable v1.
if less than return	A	v0, v1	iflr v0 v1	Used inside a function to return from it when the value of the variable v0 is less than the value of the variable v1.
if greater than or equal return	A	v0, v1	ifgr v0 v1	Used inside a function to return from it when the value of the variable v0 is greater than or equal to the value of the variable v1.
if less than or equal return	A	v0, v1	iflr v0 v1	Used inside a function to return from it when the value of the variable v0 is less than or equal to the value of the variable v1.
while greater than call function	A	v0, v1, f0	wg v0 v1 f0	While the value of the variable v0 is greater than the value of the variable v1 then call function f0.
while less than call function	A	v0, v1, f0	wl v0 v1 f0	While the value of the variable v0 is less than the value of the variable v1 then call function f0.
while greater than or equal call function	A	v0, v1, f0	wge v0 v1 f0	While the value of the variable v0 is greater than or equal to the value of the variable v1 then call function f0.
while less than or equal call function	A	v0, v1, f0	wle v0 v1 f0	While the value of the variable v0 is less than or equal to the value of the variable v1 then call function f0.
move value to variable	B	v0, v1	mov v0 v1	Make a copy of the value of the variable v0 to the variable v1.
add values and store result	B	v0, v1, v2	add v0 v1 v2	Add the value of the variable v0 to the value of the variable v1 and store the result in variable v2. In the end v2 will hold v0+v1.

subtract values and store result	B	v0, v1, v2	sub v0 v1 v2	Subtract to the value of the variable v0 the amount of the variable v1 and store the result in variable v2. In the end v2 will hold v0-v1.
multiply values and store result	B	v0, v1, v2	mul v0 v1 v2	Multiply the value of the variable v0 by the value of the variable v1 and store the result in variable v2. In the end v2 will hold v0*v1.
divide values and store result	B	v0, v1, v2	div v0 v1 v2	Perform the integer division of the value of the variable v0 by the value of the variable v1 and store the integer part of the result in variable v2. In the end v2 will hold v0/v1.
divide values and store remainder	B	v0, v1, v2	rem v0 v1 v2	Perform the integer division of the value of the variable v0 by the value of the variable v1 and store the remainder part of the result in variable v2. In the end v2 will hold v0-(v0/v1)*v1.
store sine of value	B	v0, v1	sin v0 v1	Store the sine of value of the variable v0 to the variable v1.
store cosine of value	B	v0, v1	cos v0 v1	Store the cosine of value of the variable v0 to the variable v1.
store tangent of value	B	v0, v1	tan v0 v1	Store the tangent of value of the variable v0 to the variable v1.
store arc-tangent of value	B	v0, v1	atan v0 v1	Store the arc-tangent of value of the variable v0 to the variable v1.
store square root of value	B	v0, v1	sqr v0 v1	Store the square root of value of the variable v0 to the variable v1.
get memory address value	C	v0, v1	mv v0 v1	Store the value of memory address in variable v0 as the value of the variable v1. This instruction was added to allow for the usage of pointers, arrays and linked lists (linked variables in this CPU architecture).
set variable memory address	C	v0, v1	ma v0 v1	Store the memory address of the variable v0 as the value of the variable v1. This instruction was added to allow for the usage of pointers, arrays and linked lists (linked variables in this CPU architecture).

Annex C Instructions Equivalence

This annex presents, in Table 29, the complete list of equivalences between instructions in C++, the language used to program the software robotic agent, and the emulated CPU instructions. The list also includes a description of what is expected the function in C++ supposed to be given the emulated CPU instruction.

Table 29. - List of equivalence between the emulated CPU instructions and the C++ instructions used to program.

CPU Instruction	CPU Syntax/ Example	Equivalence in C++	C++ Syntax/Examples (in <i>italic</i>)
function header	f0:	Indicates the starting point of the function.	<i>f0(){</i>
call function	call f0	Ordinary function call using global variables. Functions returns no results and receives no parameters since all variables are global.	<i>f0();</i>
return from function	ret	Ordinary return command in the middle of the function body to end it before the end of the function's body or simply the closing bracket } to end the function body.	<i>return;</i> or <i>}</i>
if zero call function	ifz1 v0 f0	Simple if instruction with only one comparison of a variable with zero/false.	<i>if(v0==0) f0();</i> or <i>if(v0==false) f0();</i>
if not zero call function	ifnz1 v0 f0	Simple if instruction with only one comparison of a variable with non-zero/true.	<i>if(v0!=0) f0();</i> or <i>if(v0==true) f0();</i> or <i>if(v0!=false) f0();</i>
while zero call function	wz1 v0 f0	Simple while instruction with only a function call inside its body, that will call the function while a variable is zero/false.	<i>while(v0==0) f0();</i> or <i>while(v0==false) f0();</i>
while not zero call function	wnz1 v0 f0	Simple while instruction with only a function call inside its body, that will call the function while a variable is non-zero/true.	<i>while(v0!=0) f0();</i> or <i>while(v0==true) f0();</i> or <i>while(v0!=false) f0();</i>
if zero call one function else call another	ifz2 r f0 f1	Simple if(){}else{} instruction set with only one comparison of a variable with zero/false. If the comparison yields true the function f0 is called else the function f1 is called.	<i>if(v0==0) f0();</i> <i>else f1();</i> or <i>if(v0==false) f0();</i> <i>else f1();</i>
if not zero call one function else call another	ifnz2 v0 f0 f1	Simple if(){}else{} instruction set with only one comparison of a variable with non-zero/true. If the comparison yields true the function f0 is called else the function f1 is called.	<i>if(v0!=0) f0();</i> <i>else f1();</i> or <i>if(v0==true) f0();</i> <i>else f1();</i>

			or <i>if(v0!=false) f0();</i> <i>else f1();</i>
if zero return	ifzr v0	Conditional return instruction that is only used inside a function. In C++ this is a combination of an if with a return inside its body. It compares a variable with zero/false and if the result yields true then the return is executed, else the next instruction in the function's body is executed.	<i>if(v0==0) return;</i> or <i>if(v0==false) return;</i>
if not zero return	ifnzs v0	Conditional return instruction that is only used inside a function. In C++ this is a combination of an if with a return inside its body. It compares a variable with non-zero/true and if the result yields true then the return is executed, else the next instruction in the function's body is executed.	<i>if(v0!=0) return;</i> or <i>if(v0==true) return;</i> or <i>if(v0!=false) return;</i>
if greater than call function	ifg1 v0 v1 f0	If the value of the variable v0 is greater than the value of the variable v1 then call function f0.	<i>if(v0>v1) f0();</i>
if less than call function	ifl1 v0 v1 f0	If the value of the variable v0 is less than the value of the variable v1 then call function f0.	<i>if(v0<v1) f0();</i>
if greater than or equal call function	ifge1 v0 v1 f0	If the value of the variable v0 is greater than or equal to the value of the variable v1 then call function f0.	<i>if(v0>=v1) f0();</i>
if less than or equal call function	ifle1 v0 v1 f0	If the value of the variable v0 is less than or equal to the value of the variable v1 then call function f0.	<i>if(v0<=v1) f0();</i>
if greater than call function else call another	ifg2 v0 v1 f0 f1	If the value of the variable v0 is greater than the value of the variable v1 then call function f0 else call function f1.	<i>if(v0>v1) f0();</i> <i>else f1();</i>
if less than call function else call another	ifl2 v0 v1 f0 f1	If the value of the variable v0 is less than the value of the variable v1 then call function f0 else call function f1.	<i>if(v0<v1) f0();</i> <i>else f1();</i>
if greater than or equal call function else call another	ifge2 v0 v1 f0 f1	If the value of the variable v0 is greater than or equal to the value of the variable v1 then call function f0 else call function f1.	<i>if(v0>=v1) f0();</i> <i>else f1();</i>
if less than or equal call function else call another	ifle2 v0 v1 f0 f1	If the value of the variable v0 is less than or equal to the value of the variable v1 then call function f0 else call function f1.	<i>if(v0<=v1) f0();</i> <i>else f1();</i>
if greater than return	ifgr v0 v1	Conditional return instruction that is only used inside a function. In C++ this is a combination of an if with a return inside its body. It compares two variables and if the first is greater than the second then the return is executed, else the next instruction in the function's body is executed.	<i>if(v0>v1) return;</i>
if less than return	iflr v0 v1	Conditional return instruction that is only used inside a function. In C++ this is a combination of an if with a return inside its body. It	<i>if(v0<v1) return;</i>

		compares two variables and if the first is less than the second then the return is executed, else the next instruction in the function's body is executed.	
if greater than or equal return	ifgr v0 v1	Conditional return instruction that is only used inside a function. In C++ this is a combination of an if with a return inside its body. It compares two variables and if the first is greater than or equal to the second then the return is executed, else the next instruction in the function's body is executed.	<i>if(v0>=v1) return;</i>
if less than or equal return	iflr v0 v1	Conditional return instruction that is only used inside a function. In C++ this is a combination of an if with a return inside its body. It compares two variables and if the first is less than or equal to the second then the return is executed, else the next instruction in the function's body is executed.	<i>if(v0<=v1) return;</i>
while greater than call function	wg v0 v1 f0	Simple while instruction with only a function call inside its body, that will call the function while the first variable of a comparison is greater than a second variable.	<i>while(v0>v1) f0();</i>
while less than call function	wl v0 v1 f0	Simple while instruction with only a function call inside its body, that will call the function while the first variable of a comparison is less than a second variable.	<i>while(v0<v1) f0();</i>
while greater than or equal call function	wge v0 v1 f0	Simple while instruction with only a function call inside its body, that will call the function while the first variable of a comparison is greater than or equal to a second variable.	<i>while(v0>=v1) f0();</i>
while less than or equal call function	wle v0 v1 f0	Simple while instruction with only a function call inside its body, that will call the function while the first variable of a comparison is less than or equal to a second variable.	<i>while(v0<=v1) f0();</i>
move value to variable	mov v0 v1	Simple attribution = in C++.	<i>v1=v0;</i>
add values and store result	add v0 v1 v2	Simple addition of two variables and the attribution of the result to one of the first two variables or to another different variable.	<i>v2=v0+v1;</i> and v2 can be v0, v1 or another variable
subtract values and store result	sub v0 v1 v2	Simple subtraction of two variables and the attribution of the result to one of the first two variables or to another different variable.	<i>v2=v0-v1;</i> and v2 can be v0, v1 or another variable
multiply values and store result	mul v0 v1 v2	Simple multiplication of two variables and the attribution of the result to one of the first two variables or to another different variable.	<i>v2=v0*v1;</i> and v2 can be v0, v1 or another variable
divide values and store result	div v0 v1 v2	Simple integer division of two variables and the attribution of the result to one of the first two variables or to another different variable.	<i>v2=v0/v1;</i> and v2 can be v0, v1 or another variable
divide values and store remainder	rem v0 v1 v2	Simple remainder (usually the % operator) of two variables and the attribution of the result to one of the first two variables or to another different variable.	<i>v2=v0%v1;</i> or <i>v2=v0-(v0/v1)*v1;</i> and v2 can be v0, v1 or another variable

store sine of value	sin v0 v1	Calculation of the sine of the value of the first variable and the attribution to the same or another variable	$v1 = \sin(v0);$ and v1 can be v0 or another variable
store cosine of value	cos v0 v1	Calculation of the cosine of the value of the first variable and the attribution to the same or another variable	$v1 = \cos(v0);$ and v1 can be v0 or another variable
store tangent of value	tan v0 v1	Calculation of the tangent of the value of the first variable and the attribution to the same or another variable	$v1 = \tan(v0);$ and v1 can be v0 or another variable
store arc-tangent of value	atan v0 v1	Calculation of the arc-tangent of the value of the first variable and the attribution to the same or another variable	$v1 = \text{atan}(v0);$ and v1 can be v0 or another variable
store square root of value	sqr v0 v1	Calculation of the square root of the value of the first variable and the attribution to the same or another variable	$v1 = \text{sqr}(v0);$ and v1 can be v0 or another variable
get memory address value	mv v0 v1	Store the value of the memory address stored in variable v0 as the value of the variable v1. This instruction was added to allow for the usage of pointers, arrays and linked lists (linked variables in this CPU architecture).	$v1 = *v0;$
set variable memory address	ma v0 v1	Store the memory address of the variable v0 as the value of the variable v1. This instruction was added to allow for the usage of pointers, arrays and linked lists (linked variables in this CPU architecture).	$v1 = \&v0;$

Annex D Map Viewer Options

The Figures 54 to 58 presented in this annex, visually displays the influence of some of the menu options of the graphical interface of the Map Viewer application, enumerating which options are activated or not.

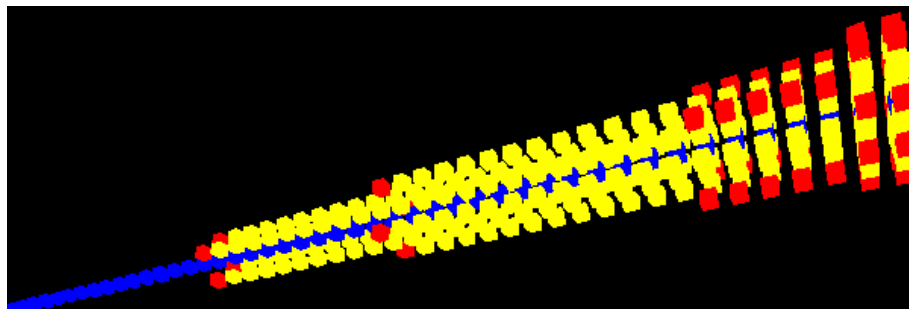


Figure 54. - Detail of a cone with accessibility on and colour scheme set to four colour.

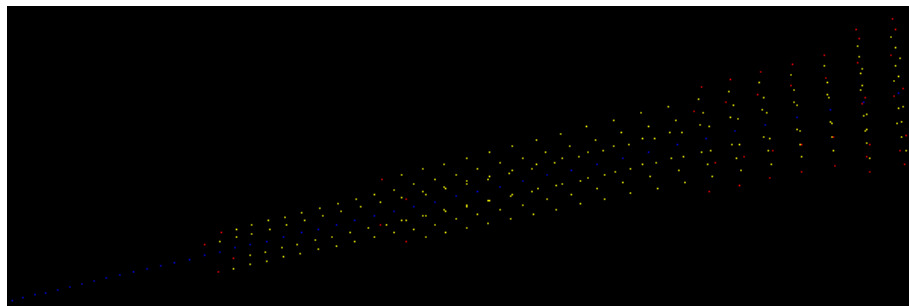


Figure 55. - Detail of a cone with accessibility off and colour scheme set to four colour.

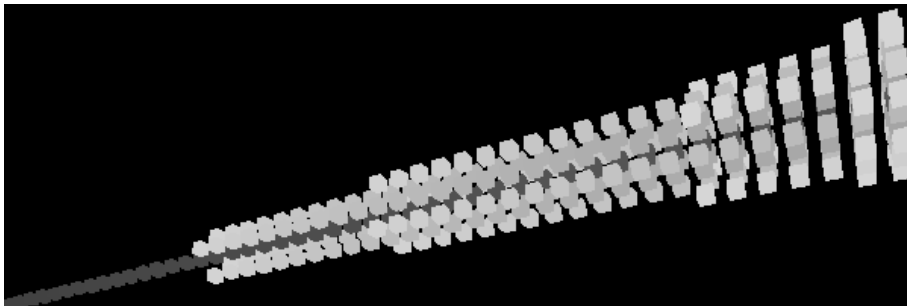


Figure 56. - Detail of a cone with accessibility on and colour scheme set to grey scale.

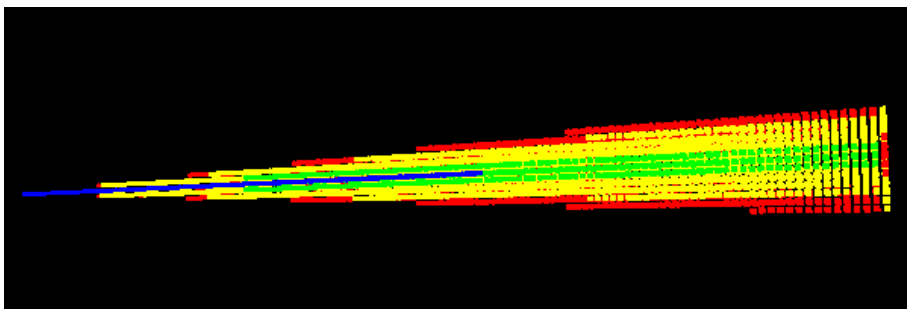


Figure 57. - Detail of a cone cuted in half using the window function with accessibility on and colour scheme set to four colour.

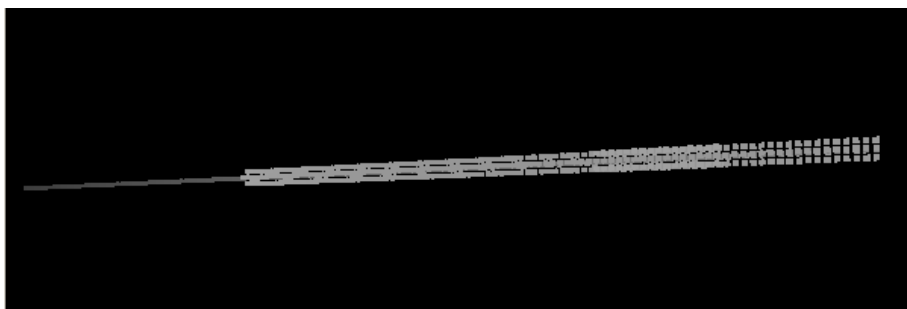


Figure 58. - Detail of a cone filtered using the probability function with accessibility on and colour scheme set to grey scale.

Annex E Simulation Environment

To ease the implementation of this work, it was decided to implement it under a simulated environment. Our choice fell on the USARSim simulation platform which is the simulation platform used in the Virtual Robots Competition of the RoboCup Rescue (2). The simulation platform is composed of a physics simulator, a graphics simulator and a robotics simulator.

Annex E.1 Unreal Tournament 2004

The physics and graphics simulation is performed by Unreal Tournament 2004 (UT2004) from EPIC Games (3), one popular multi player 3D first person shooter.

UT2004 uses a client-server architecture like the one shown in Figure 59. For each client there is a corresponding interface on the server side called bot. The client side is responsible for processing the graphics that each bot sees and for sending the player intended actions to the server.

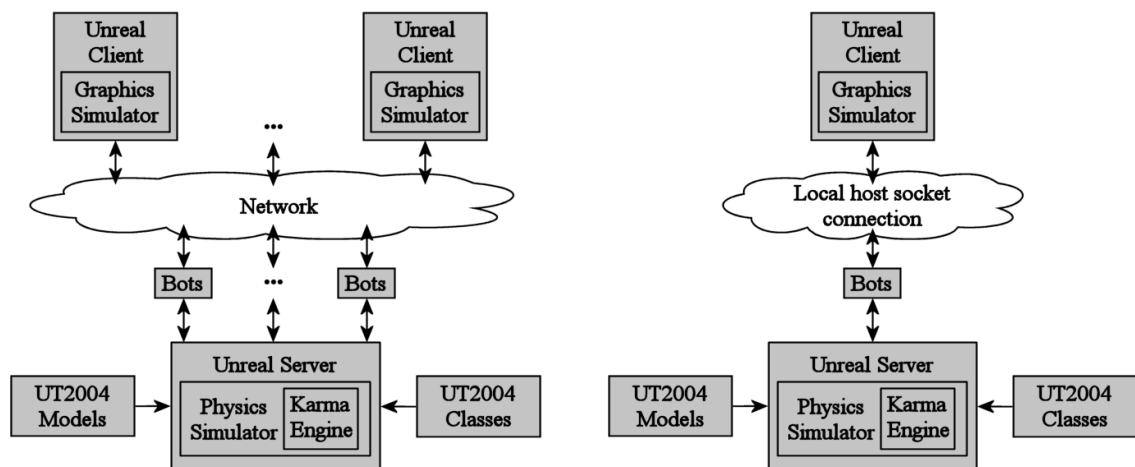


Figure 59. - UT2004 client-server architecture for multi player games (left) and single player games against the AI bots (right).

The server part of UT2004 is responsible for maintaining and updating the simulated world state, receiving from the bots the actions to execute and sending to each bot the corresponding part of the state of the world that affects it. The state of the world is calculated using a rigid body physics simulator called Karma (4)(5).

The UT2004 game engine also provides an Unreal Virtual Machine (6) which allows a user to create programs to interact with the game in a language called UnrealScript.

Figure 60 shows some screen shots of the graphics of the game. Notice the level of detail of the models present in those screen shots.



Figure 60. - UT2004 screen shots.

Annex E.2 USARSim

The robotics simulator is named USARSim and it is an UnrealScript application that runs on top of UT2004 to provide APIs, robot and scenario parts, sensors and actuators in order to facilitate the implementation and interaction with the robot, and the implementation of scenarios. Its combination with UT2004 results in a realistic simulator and the measurements returned by it are comparable in quality to those a real robot would return in a real environment under certain conditions like it is explained in (6).

USARSim also uses GameBots (7) which are a modified version of the game's bots to allow for network communication that is not controlled by UT2004's game engine.

Figure 61 presents the architecture of USARSim and its interaction with UT2004. The grey blocks of Figure 61, like the Unreal Engine and Unreal Client blocks belong to UT2004 as well as some of the models and classes respectively represented by the Models and Classes blocks. USARSim adds most of the white blocks, which are the models and classes of the robots, robot parts, scenarios, scenario parts, sensors, actuators and the GameBots. The controller application is the user's robotic software agent, like in the case of this work, or could be a third party robotic controller installed by the user.

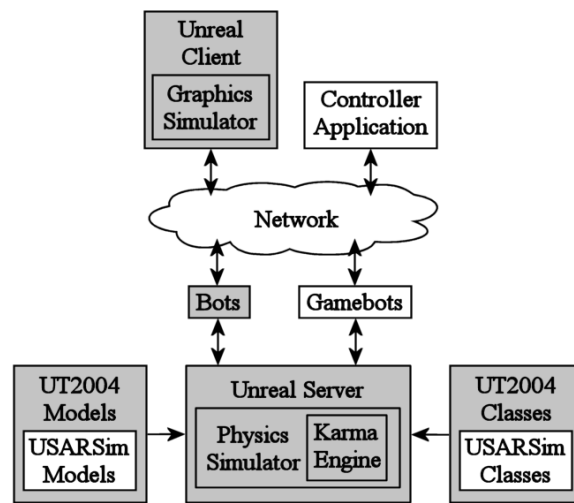


Figure 61. - USARSim architecture. The boxes in grey belong to UT2004.

USARSim uses the right hand SAE J670 Vehicle Coordinate System, presented in Figure 62, where the x axis points out the front of the robot, the y axis points to the right and the z axis points down. This results in a coordinate system that is similar to the left hand system used by UT2004, except that the height, the z axis, is inverted making heights grow toward increasingly negative numbers.

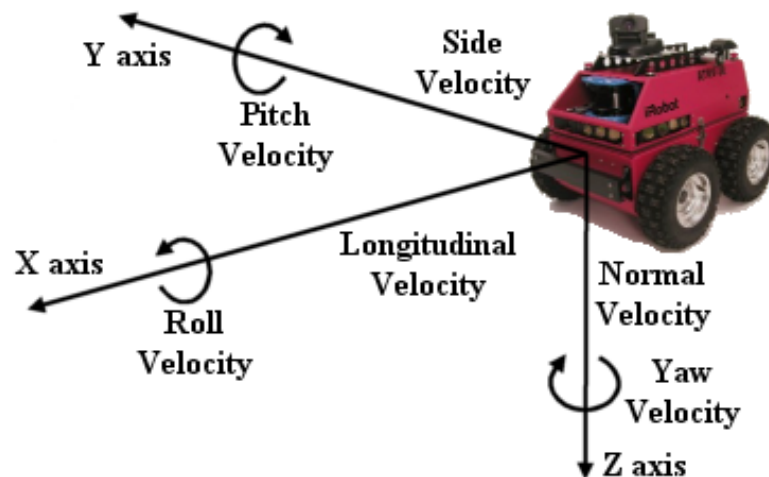


Figure 62. - USARSim's coordinate system.

Annex E.3 Installation

The installation of the simulation platform performed in the Windows® OS is very simple and will briefly be presented next. For a more detailed installation or to install in another OS, consult the USARSim manual (9). The main steps are listed in Table 30. First, the full commercial version of UT2004 needs to be installed. If the version of UT2004 that is being installed is different from v3369, a patch needs to be installed (ut2004-winpatch3369.exe)(8).

After installing UT2004 and the patch, USARSim can be installed (USARSimFull_3.31.exe)(9). It is then required to install the map base files (USARSimBaseFiles_3.30.exe)(9) to add to UT2004 the objects necessary to build the scenarios.

In case there is the need to work with robots that have more than one camera, or that require the simulation of a real camera, the easiest way is to install the image server application (ImageSrv_3.1.zip)(9).

The last step is to install the scenarios (called arenas) that will be used. In this work, the yellow arena scenario was used (DM-YellowArena_V3.1.exe)(9) for testing.

Table 30. - Main steps to install the USARSim platform.

Step	Description
1	Install the full commercial version of Unreal Tournament 2004.
2	Install Unreal Tournament 2004 patch for version 3369, if required. The file is named ut2004-winpatch3369.exe in (8).
3	Install USARSim files. The file is called USARSimFull_3.31.exe in (9).
4	Install USARSim base map files. The file is called USARSimBaseFiles_3.30.exe in (9).
5	Install USARSim scenarios. Several scenarios are available in (9).

Before starting to use the simulator, it is recommended that a compilation of the classes is performed. It should also be performed after modifying any class or model, like when building your own robot. The compilation is executed by selecting the Compile shortcut from the USARSim menu under the Windows® OS's Start menu/programs, like in Figure 63.

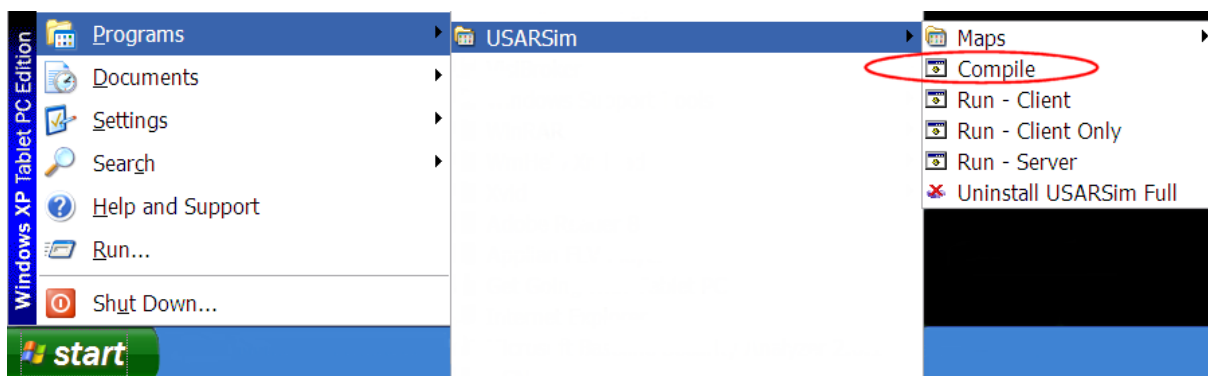


Figure 63. - Compile shortcut of USARSim under Windows® OS.

Annex E.4 Starting the Simulator

During the installation of the scenarios, each scenario adds a menu to the USARSim menu in Windows® OS's Start menu/programs. That menu contains links that allow launching the scenario in server or client only mode. The easiest way to start a simulation is to launch the intended scenario in server mode like shown in Figure 64.

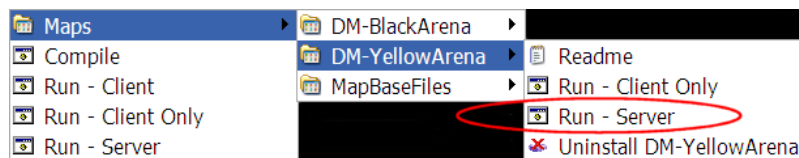


Figure 64. - Running USARSim in server mode.

After, if only one camera is required and/or no real camera is required by the robot to be simulated, both the robot controller and any number of Unreal Clients can be launched to be set with the camera views. The Unreal Client allows us to see the scenario from any perspective. The Unreal Client can also be embedded in the robot controller. For instructions refer to the USARSim manual (9).

If more than one camera is required or it is required to simulate a real camera, the image server application, presented in Figure 65, must be launched right after the launching of the scenario. Only after the image server has completely launched, the robot controller can be launched. There is no need to launch an unreal client because the image server already has a client built in that can be used to see the scenario or through the robot cameras.

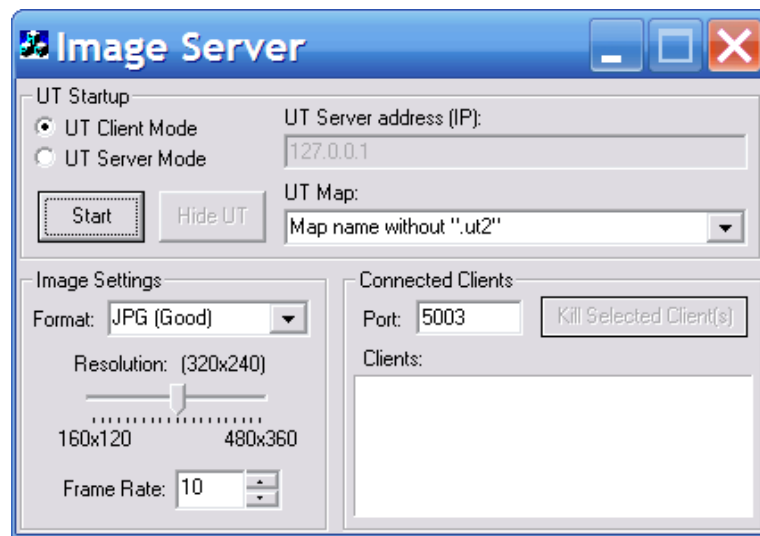


Figure 65. - Image Server interface.

The best way to use the Image Server is to configure it to run in "UT Server Mode", in which it connects to UT2004 using an IP address. For a more detailed explanation of how to work with the Image Server refer to USARSim manual (9).