Faculdade de Engenharia da Universidade do Porto

Automatic Inconsistency detection in a Logistic World Model (European Project STAMINA)

Rafael Lirio Arrais

Dissertação realizada no âmbito do
Mestrado Integrado em Engenharia Electrotécnica e de Computadores
Major Automação

Orientador: Prof. Dr. Mário de Sousa
Co-orientador: Eng. César Toscano

Junho 2015

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES | **2014/2015**

A Dissertação intitulada

"Automatic Inconsistency Detection in a Logistic World Model (European Project STAMINA)"

foi aprovada em provas realizadas em 17-07-2015

o júri

Presidente Professor Doutor Armando Jorge Miranda de Sousa
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

Professor Doutor José Luís Magalhães Lima
Professor Adjunto do Departamento de Eletrotécnia da Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Bragança

Professor Doutor Mário Jorge Rodrigues de Sousa
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.

Autor - Rafael Lirio Arrais

Faculdade de Engenharia da Universidade do Porto

Resumo

O aparecimento de novas soluções mais baratas e eficazes tornou a perceção tridimensional na nova referência na área dos sistemas de visão robóticos. As novas câmaras com registo de profundidade contribuíram para uma representação espacial dinâmica mais robusta e de maior qualidade.

Este projeto de dissertação apresenta um software para a deteção e processamento de inconsistências espaciais, permitindo atualizações automáticas à estrutura de dados na qual o projeto europeu STAMINA apoia a sua navegação.

Tal como é demonstrado ao longo do projeto de dissertação, verifica-se que é possível detetar alterações espaciais utilizando os algoritmos propostos, que contemplam diversas técnicas que permitem filtrar os resultados obtidos a fim de apresentar um resultado final mais satisfatório. Diversos parâmetros configuráveis permitem que o sistema desenvolvido seja facilmente integrado em diversos cenários.

O desenvolvimento tecnológico inerente a este projeto não só potencializa o aumento dos níveis de automação na linha de montagem do caso de uso do projeto STAMINA, mas também contribui significativamente para o desenvolvimento de novas soluções em pequenas e médias empresas.

Palavras-Chave: Automatic Inconsistency Detection, Dynamic Mapping, Octree, OctoMap, Point Cloud, ROS, Three-Dimensional Perception.

Abstract

The new sensing hardware and its associated perception have slowly become the new standard in robotics vision system. The advent of new depth camera systems made three-dimensional perception the cheaper and most effective solution to create reliable and high-quality three-dimensional dynamical representations of space.

This dissertation project presents a software for detecting and dealing with spatial inconsistencies in order to automatically update the data structure which supports navigation on the European project STAMINA.

As it is demonstrated, as a result of this dissertation project, it is possible to successfully detect spatial changes using the proposed algorithms, which present several techniques in order to filter the results as to provide a better overall result. Several customizable parameters make a seamless integration and reusability in diverse scenarios.

The development of such technology will not only potentiate the increase of automation levels in the assembly stage of the STAMINA use case but it will also result in a significant contribute to the rise of new solutions to small and medium-sized enterprises.

Keywords: Automatic Inconsistency Detection, Dynamic Mapping, Octree, OctoMap, Point Cloud, ROS, Three-Dimensional Perception.

Table of Contents

Figure List

Table List

Abreviations

AAU	Aalborg University
AGV	Automated Guided Vehicles
ALU-FR	Albert-Ludwigs-Universitat Freiburg
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
EIS	Enterprise Information System
ERP	Enterprise Resource Planning
FAC	*Fiche Accompagnement*
FN	False Negative
FP	False Positive
FPR	False Positive Rate
INESC-TEC	Instituto de Engenharia de Sistemas e Computadores – Tecnologia e Ciência
JSON	JavaScripit Object Notation
JSON	JavaScript Object Notation
LDP	Location Determination Problem
MES	Manufacturing Execution System
NaN	Not a Number
PBJ	Pixel Bender Bytecode File
PCD	Point Cloud Data
PCL	Point Cloud Library
PLY	Polygon File Format
R&D	Research and Development
RANSAC	Random Sample Consensus
ROC	Receiving Operating Characteristics
ROS	Robot Operating System
SDF	Simulation Description Format
SkiROS	Skill Based System for ROS
SME	Small and Medium-sized Enterprises
STAMINA	Sustainable and Reliable Robotics for Part Handling in Manufacturing Automation

STL	STereoLithography
TN	True Negative
TP	True Positive
TPR	True Positive Rate
UBO	Rheinische Friedrich-Wilhelms-Universität Bonn
UEDIN	University of Edinburgh
URDF	Unified Robot Description Format
URI	Uniform Resource Identifier
X3D	Xara 3D Project File
XML	Extensible Markup Language
YAML	YAML Ain’t Markup Language

Chapter 1

Introduction

The automotive industry is a key industry in the European Community and has one of the biggest automation levels in production.

Due to the production variability and to the large number of suppliers and components in the automotive industry, the handling of individual parts in the assembly stage constitutes a task with levels of automation lower than 30%. The lack of automation in the processes in this stage of production results in the execution of repetitive and physically exigent tasks by human operators.

The repetitive and successive handling of thousands of parts with considerable weight by human operators results in an accumulation of several tons of components transportation by an individual in a single work shift [1]. There are several accidents, errors and work injuries that are the result of fatigue caused by efforts that the human operators are subjected [2]. Those could be avoided by the automation of tasks that are dangerous to the health of human operators. Besides the health problematic, there is a change in the demographics of the European industries, where the age of the industry work force has increased considerably in the last years.

The development of technologies that could potentiate the increase of automation levels in the assembly stage will result in a significant contribute, not only to the automotive industry, but also to the development of new solutions in the assembly stage of small and medium enterprises (SME). The full automation of such tasks is possible by building on previous R&D to develop a fleet of advanced mobile robotic manipulators, capable of dealing with unstructured environments [3].

In order to function in different scenarios, the robotic solution developed has to be able to execute the same tasks as a human operator would. This is accomplished by developing a fleet of autonomous and mobile industrial robots with different sensory, planning and physical capabilities for jointly solving the challenges of the assembly stage [3].

This dissertation project is part of the European project STAMINA, Sustainable and Reliable Robotics for Part Handling in Manufacturing Automation, which has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration (FP7). From 2007 to 2013, the programme aimed to stimulate and improve links between industry and research within a European framework, consolidating Europe's leadership in key research areas [4].

STAMINA project will build on previous R&D, partnering with expert partners in each necessary key field, aiming at developing a fleet of autonomous and mobile industrial robots equipped with broad physical capabilities and several sensorial systems able to co-exist and work with humans at an automotive industry scenario. STAMINA project will focus on developing a sustainable and scalable robotic system to ensure a clear path for the future exploitation of the developed technologies [3], and to allow its easy integration in other SME scenarios.

STAMINA is the result of collaboration between PSA Peugeot Citroen, a French multinational manufacturer of automobiles and motorcycles and BA Systèmes, French market leader in the production of Automated Guided Vehicles (AGV), as industrial partners, and several European universities and research centers as research partners. The STAMINA project started on October 2013 and it will last for 42 months, finishing on March 2017.

1.1 - Motivation

As an answer to the recent changes in consumer's habits, represented by the growing need of product personalization, the automotive industry has been forced to adapt its strategy from a mass production to a production focused on personalization of products and services, in order to cope with the changes in consumer needs and to stay competitive. As a result of the changes, in order to satisfy the needs of consumers, the automotive industry has suffered a change in its paradigms of production, which resulted on a reorganization of the production and changed the distribution of tasks at the production line.

Changes in the paradigms of production and the need to stay competitive resulted on the implementation of a Lean production ideology, especially in the processes that lead to the feeding of parts to the assembly line.

There are three main material supply patterns for feeding parts to the assembly line that are currently spread in the automotive industry: continuous feeding, sequential feeding and kitting.

Continuous feeding provides the delivering of parts directly to the production line. In this pattern, all parts variation required for producing an automobile are available at the assembly line at all times.

Sequential feeding refers to the deliverance of parts in the sequence defined by production management systems to the assembly line. This pattern is usually applied to parts that have a large number of variants. Although sequential feeding usually requires no space in the production line to store parts variants, it requires upstream sequential operations.

Kitting consists in the process where parts are grouped and supplied together in one container called kit. Each kit contains parts for one assembly object (in this situation, a car). The main idea of kitting type distribution is to concentrate the value added on the production line.

Kitting operations are often performed by human operators, responsible for selecting parts, packing and supplying them to the assembly line, following an order issued by the production management system, in order to maintain the synchronization with the cars in production.

Kitting operations are related to the Lean manufacturing principle, since once the kit with the right parts has been delivered to the assembly line, the assembly operator has all the components within reach, being able to concentrate on the assembly operation. It also improves control over the flow of components, and requires less storage for parts, resulting in a reduction of floor space, since there is a decentralized storage in the kitting area.

It has been verified that kitting operations are also responsible for more flexible production lines, since it is easier to do production changeover and to address fluctuation of production rates. Early identification of defects and less damaged parts results in quality improvements while using the kitting type distribution.

In the present day, kitting operations are performed by groups of human operators, employed with the goal of making the kits of parts to be delivered to the production line. The process of making a kit with the right parts variations is a complex operation, due to the large variations in the products and the diversity of suppliers and parts. The automation level in this task, in the automotive industry, is below 30 percent.

As previously stated, there are several health risks associated with the kitting tasks being performed by human operators. Lifting very heavy car parts, weighting up to 10 kilograms, can lead to a handling by a single worker of up to 12 tons per shift. Besides the health problem associated with such efforts, the fatigue cause by the work can lead to dangerous mistakes to the workers health. This constitutes a harmful job to humans and so it should be given to robots. At the same time, the use of robots can help to solve the demographic challenges of Europeans ageing workforce.

The R&D needed to implement a robotic solution capable of rising the automation level of the kitting task in the automotive industry can not only solve the health hazards to an ageing workforce, but it could also potentiate the research and development of such technologies to other industries, allowing European industries to have the necessary leverage to pursue the path of innovation.

1.2 - Objectives

The aim of the STAMINA project is to develop a fleet of mobile robotic manipulators to do kitting tasks in a kitting supermarket, in order to feed the production line of a PSA Peugeot Citroen factory in Rennes, France [5]. The use case selected by PSA Peugeot Citroen for STAMINA represents the use of a fleet of mobile manipulators in a logistic supermarket for kitting tasks with the goal of feeding a sub-assembly production line.

Some of the biggest challenges of developing a fleet of mobile autonomous manipulators to perform complex operations such as kitting tasks while co-existing in an unstructured environment with human operators has to do with its localization, navigation, acknowledgment of obstacles and the ability to notify errors and incoherencies not only to other robots of the fleet, but also to the human operators.

Navigating on a dynamic environment is a complex task for an automatic mobile manipulator. There is an implicit need to support the navigation and manipulation of objects on a data structure which holds references to the position of the several objects that compose the universe of action of the robotic fleet. Keeping a structure holding a reference to every physical object on a factory scenario updated with the latest changes in real time is a challenging problem.

The end result of STAMINA is a robotic system that is able to drive autonomously to where it is needed and has cameras that allows it to have a perception of its surroundings. The autonomous behavior of the robotic fleet is supported by a complete integration with the factory production and management systems and the ability to dynamically update it in real time.

Most of the current approaches to mobile robot mapping assumes that the world is static, not taking into consideration the several processes and mechanisms that continuously change the state of the modeled universe. This simplification approach is generally not justified in real-world applications, where often there are several dynamic changes to the environment.

Three-dimensional perception is a field of research where, in the last years, a continued effort has been made in order to significantly improve its importance in robotics and in many other fields. This newfound impulse to develop the technology associated with three-dimensional perception is justified with the advent of new low-cost three-dimensional sensing hardware, as is an example the Microsoft Kinect, developed for the Microsoft XBox 360 game console. The possibility of applying such systems on SME robotics applications is a solution to

sensing the state of the world in real time, being able to dynamically update the world representation.

The main objective of this dissertation project is to develop a vision system to the STAMINA robotic system that is able to make the correspondence between a representation of the logistic supermarket and the images collected by the robots camera system, in order to check for spatial inconsistencies and report them in real time. This dissertation project is expected to focus on developing the mechanism to detect inconsistencies based on the detection of containing devices misplaced. The developed solution will be able to be easily integrated with the already defined STAMINA architecture.

1.3 - Document Organization

This document is divided into five chapters. The first chapter constitutes the introduction chapter, where a briefly presentation of the context, motivation and objectives of the dissertation project has been made.

The second chapter is aimed at making an introduction to the STAMINA project, contextualizing the motives that lead to the development of this dissertation project. At first, a briefly system architecture overview is presented, followed by a characterization of the several partners that compose the STAMINA project and their roles on the project. The characterization of some key components of the STAMINA project, as the automotive parts, the storage containers, and the layout of the kitting supermarket, as well as its normal operation flow are important to a better understanding of the whole project and to justify some assumptions made during development. The Information Model, Software Integration and World Model subchapters provide an understanding of the foundation in which the mechanisms developed during the dissertation project are expected to collaborate.

The third chapter provides an overview on some of the key aspects associated with the state of the art of several technologies used during this project. Its subchapters reflect a theoretical approach on some of the concepts that will be further discussed or are the fundamental principles to support some of the mechanisms developed.

The fourth chapter discusses the integration of the mechanisms developed with some of the already existing STAMINA interfaces and presents the developed solution. In this chapter, it is also identified some of the spatial changes that are the objectives of detection within

this dissertation project. The last two subchapters addresses the development of the tools created in order to accomplish the requirements defined, being the first one a presentation of the prototype software developed and the last one an analysis of the final solution, with its respective tests and results.

Finally, the fifth and last chapter presents the conclusion to this document, as well as some insights on what might be done in order to assure a continuity on the R&D established by this dissertation project.

Chapter 2

STAMINA Overview

2.1 – System Architectures Overview

The STAMINA autonomous robot fleet are mobile manipulators with different sensors, planning and physical capabilities for jointly solving logistic and handling challenges, such as de-palletizing, bin-picking and kitting. As previously stated, the STAMINA robot will be able to autonomously navigate and construct kits based on the requirements of the production plant.

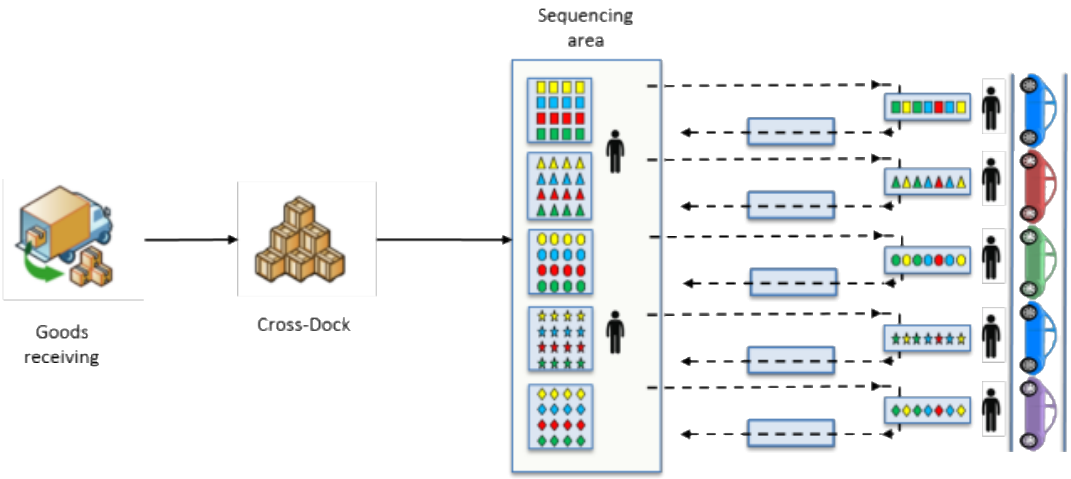


Figure 1 - Kitting Task [5].

In order to perform its objective, the STAMINA project is composed by several areas which constitutes the skill set that the robotic fleet is capable of executing: **navigation**, **picking parts**, **placing parts**, **planning** and **robot coordination** [6].

Each area is attributed to one of more research groups and it is expected that the research and development within each area and within each research group should be relatively autonomous. In order to coordinate the work done in each area and to check that all components are able to be assembled in a coherent system, there is a functional description of the whole system, without consideration of implementation. The system integration describes the hardware and software interfaces [6].

On the hardware level, the STAMINA robot is composed by a mobile platform, which receives velocity commands and provides odometry measurements. Mounted on the mobile platform, there are the robotic arm and the conveyor for the kitting boxes. In the extremety of the robotic arm, there is a gripper (hand). The STAMINA robot is also composed by vision sensors for picking and placing, navigation sensors and a Wi-Fi supervision and communication system, so that the robot can be integrated in a fleet and receive and send information to the control system [6].

On the software level, the role of the partners and the interactions between them is shown on the next diagram.

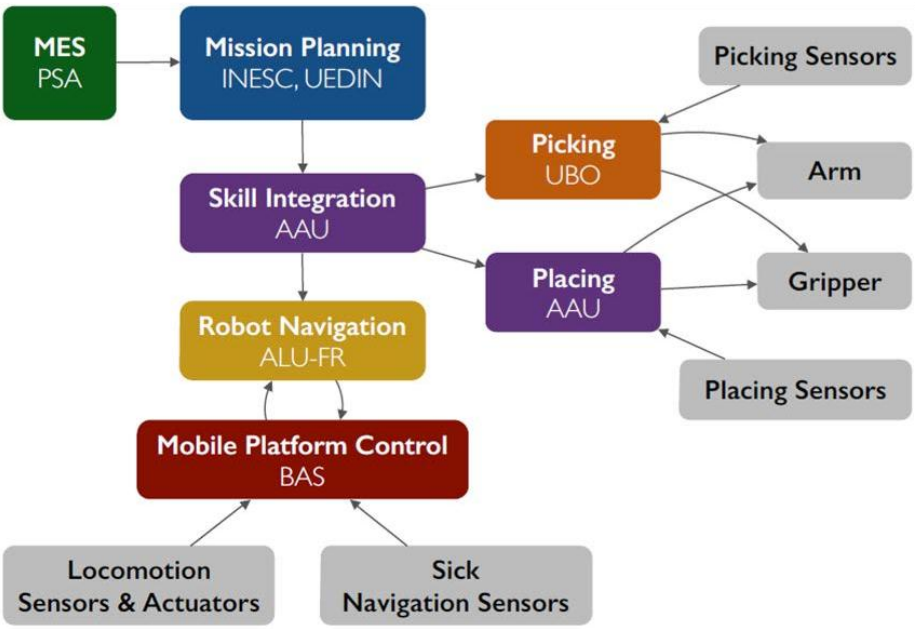


Figure 2 - STAMINA Software Architecture: Partners role. Adapted from [6].

The diagram is composed by the several areas of development (colored boxes) with indications of the partners responsible for its research and development. The diagram also symbolizes the components which interact with the different development areas (grey boxes). The role of each research group and industrial partner will be further analyzed in the following subchapter.

2.2 – Partners

PSA Peugeot Citroen group is the second largest car maker in Europe, with 11,8% of the market. In 2013, the group sold more than 2,8 millions of cars in all over the world, 58% of them in Europe, making the group European leader on the light commercial vehicles, with 20,3% of the market. In 2014, the PSA Peugeot Citroen group employed almost 185 thousand workers all over the world, and generated a revenue that exceeded 53 million Euros [7].

The group has a double role on the STAMINA project: On one had, it is the client of the project, *i.e.*, the end user of the technologies developed by the partners of the project. Because of that, PSA will be responsible for providing specifications to reflect industrial needs and constraints and also it will be in charge of continuously assessment of the technology, as to make it suitable to be placed in a serial production.

On the other hand, due to the experience of the R&D robotic team of PSA, critical issues regarding reusability, reliability, robustness, maintainability, safety and usability will be carefully addressed [8].

Founded in 1975, BA Systèmes SAS is an SME specialized in mobile robotics and in handling and storage solutions enabled by autonomous guided vehicles that is showing a constant growth for the last 8 years. As the market leader in France for AGB-based intralogistic solutions, BA Systèmes SAS has more than 250 customer sites across Europe, originating 20 million euros turnover in 2013 [9].

Within the STAMINA project, BA Systèmes SAS is expected to contribute with its expertise in safety requirements analysis for a mobile robot, R&D of solutions for integration of safety and vision systems for localization in a dynamic environment with humans, design of Supervisor Control Systems architectures and solutions, R&D for demonstrations and tests, business development and technology transfer plan.

Since there is huge commercial potential regarding the development of robotics in industry, BA Systèmes SAS will explore the results of the STAMINA project by opening up opportunities in new markets and countries.

Aalborg University (AAU) has the role of scientific and general management of the project, acting as the STAMINA project coordinator, AAU will be responsible for guaranteeing that the technologies developed by the partners can be connected and work together as its supposed to be.

Research-wise, AAU will provide the expertise, the required development work the implementation and the testing of the robot skill modelling paradigm, on the human robot interface, the placing skill and the inspection skill [8].

The experience of the Albert-Ludwigs-Universitat Freiburg (ALU-FR) on the field of mobile platform navigation will be used on the STAMINA project to solve the localization and navigation problem on dynamic environments. Furthermore, ALU-FR addresses motion planning and motion execution within STAMINA [8].

The participation of the Rheinische Friedrich-Wilhelms-Universität Bonn (UBO) is related with the task of grasping of parts from transport containers for depalletising and bin picking. The bin picking task is composed by the detection and pose estimation of parts as well as the grasp and arm motion planning for a robot manipulator [8].

The Instituto de Engenharia de Sistemas e Computadores – Tecnologia e Ciência (INESC-TEC) is responsible for vertical integration and for the cooperative behavior of the robots in terms of mapping and motion coordination, *i.e.*, traffic management. INESC-TEC will also be responsible for dissemination of the project, specifically the coordination of knowledge transfer and cooperation with the other STAMINA partners to promote appropriate visibility and sustainable exploitation paths [8].

The University of Edinburgh (UEDIN) is responsible for developing and implementing the high-level decision-making components that will be used to control individual robots in STAMINA, due to its expertise in automated planning, knowledge representation, cognitive robotics and planned execution monitoring. UEDIN is also in charge of orchestrating individual robots into fleets in order to produce more complex group behavior [8].

2.3 – Parts

A car is composed by thousands of individual parts, and there are several ways of selecting the parts to assemble a car, in order not only to produce different car models but also to satisfy the costumers needs of customization. Since there are unlimited configurations, it is crucial to select a small number of parts which will be representative of the STAMINA use case. The selection of a small number of realistic and representative parts will not only demonstrate the universality of the project, but will also allow cooperation between all partners.

The parts selected for the STAMINA use case were chosen based on factory observation of the various parts handled in the Logistics and Assembly workshop at PSA [5]. The parts were classified according to its properties (fragile, shiny, transparent, unbalanced), packaging properties (plastic wraps, cardboard separator, bulk storage, stacking) and task constraints (authorized grasps, part entanglement, placing constraints).

Following the criteria specified, six parts taken from the engine assembly line were selected for the STAMINA use-case. The parts have been chosen since they represent a wide variety of properties and constraints which will have an impact on the hardware and software choices concerning object recognition and grasping. It was also important to choose parts which are packaged in small boxes and in large volumes, since it has an impact on robot reachability [5].

The parts chosen are the **engine pipe**, **engine support**, **thermal shield**, **alternator**, **starter** and **air conditioning compressor**. The first 3 parts are packaged in small boxes while the last 3 are contained in large volume type packaging.

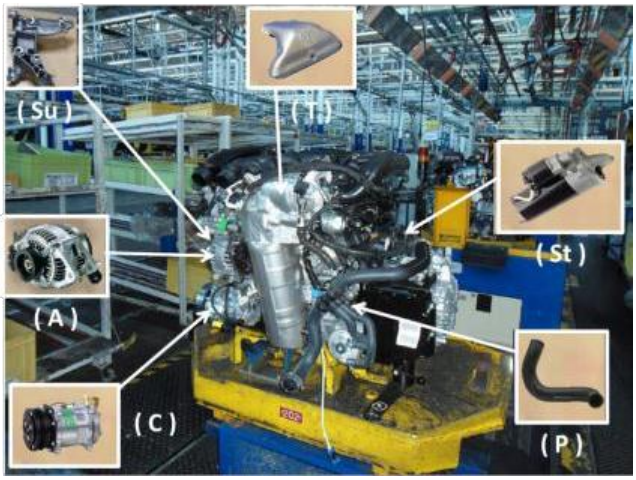


Figure 3 - Selected parts for the use case on the sub-assembly (engine) [5].

Figure 4 - Engine pipe, engine support, thermal shield, alternator, starter and air conditioning compressor, respectively [5].

There are, in total, 32 different parts variants for the 6 parts chosen: 3 variants of engine pipes, 7 variants of engine supports, 3 variants of thermal shields, 8 variants of alternators, 6 variants of starter engines and 5 variants of air conditioning compressors [5]. On **Annex A** it is gathered useful information on the part variants and their properties, such as the variant designation, PSA reference, container box type, weight, number of units per box and average daily consumption.

It was decided to chose parts belonging to the fabrication process of the same sub-assembly (engine), as it is possible to gather them within a kit that is coherent with the real factory needs. In figure 3, it is possible to observe the approximate location of every part on the sub-assembly chosen.

There should be noted that the small boxes containing engine pipes, engine supports and thermal shields are stored in racks placed on the logistic supermarket. Since the parts contained in the large volume boxes are heavier than those contained in the small boxes, the large volume boxes storing alternators, starters and air conditioning compressors are placed directly on the floor of the logistic supermarket.

2.4 - Layout and Storage

The proposed layout of the kitting supermarket has the aim of being an efficient arrangement of the different containers storing the part variants with the view of optimizing the motions of the STAMINA robotic fleet.

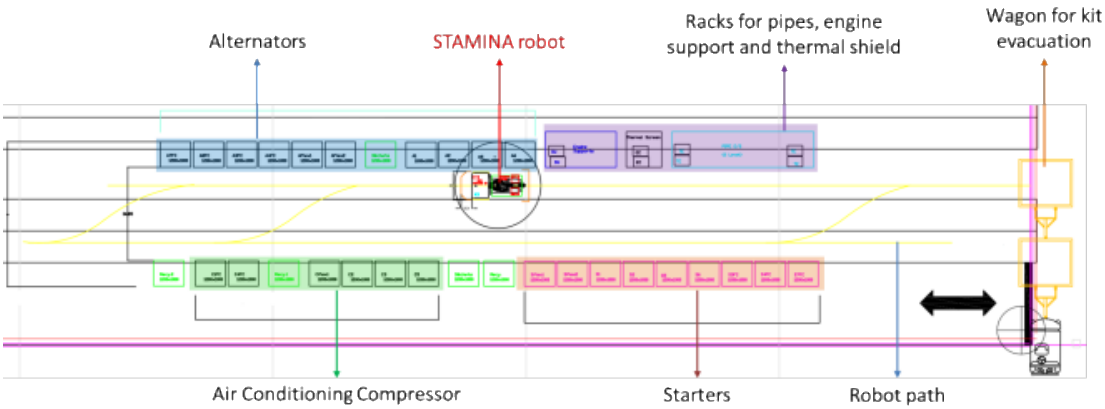


Figure 5 - Proposed linear topology for the kitting supermarket [5].

The proposed kitting supermarket will be organized as a straight line arrangement of the containers, grouping them according to part variants and families. The less-consumed parts are placed further away, to optimize motions. The STAMINA robotic fleet will navigate on the aisle performing loops, allowing for efficient path management of the fleet, as there will be fewer maneuvers to perform. The proposed layout will also allow multi-robot coordination with reduced interferences.

On the proposed layout, particular attention should be given to preserve the accessibility and ergonomics for logistic operations, with special focus on loading and unloading the racks or pallets. It is also necessary to take into consideration possible human intervention in the logistic supermarket. The proposed logistic supermarket layout should be able to adapt to different scenarios, so some flexibility to modifications will be provided.

As previously stated, the different parts for the use case are stored in different containers. Heavy parts are stored in large boxes, called pallets on the official STAMINA documentation [5]. Those pallets are stored directly on the kitting shopping floor, on an area suited for the effect. Smaller and lighter parts are stored on small boxes, which are stored in racks, distributed across the kitting shopping.

It is possible to find the dimensions of the different containers used on the use case on the following tables. The dimensions and formats of the containers are essential in order to properly model the system.

Table 1 - STAMINA use case small boxes characteristics [5].

Reference	Stores	Length	Width	Height	Weight (empty)
06432	Engine pipe & Thermal shield	594 mm	396 mm	314 mm	2,96 kg
06432	Engine support & Thermal shield	396 mm	297 mm	213 mm	1,44 kg
06422	Engine support	594 mm	396 mm	214 mm	2,38 kg
BNA20	Engine support	600 mm	400 mm	200 mm	0,80 kg
ANC20	Engine support	400 mm	300 mm	200 mm	0,50 kg

Table 2 - STAMINA use case pallets characteristics [5].

Reference	Stores	Length	Width	Height	Separator
00081	Alternator & Starter	1150 mm	800 mm	730 mm	Cardboard
63396	Alternator	1200 mm	1000 mm	910 mm	Plastic (1 or 6 kg)
CCF75	Alternator	1200 mm	1000 mm	900 mm	Cardboard
CAF75	Starter	1200 mm	1000 mm	900 mm	Cardboard
CCF60	Starter	1200 mm	1000 mm	900 mm	Cardboard
16396	Air conditioning compressor	1130 mm	970 mm	860 mm	Plastic (5 or 9 kg)
17306	Air conditioning compressor	1130 mm	970 mm	700 mm	Plastic (5 or 9 kg)

The STAMINA robotic fleet will be able to handle tasks which guarantee a continuous work load, such as supermarket organization and waste disposal. Regarding the large volume boxes, the robot will have to discard separators between levels of parts, disposing them in specific garbage containers located on the logistic supermarket. There is also the need to gather all plastic separators in a designated container for recycling [5].

Small boxes serve the double purpose of containers for light parts and are also used as a Kanban[1] for the factory management system. As illustrated on the next figure, the rack is generally composed by two layers of full boxes fed by gravity. The layer on the top comprises the evacuation slope for empty boxes, and the STAMINA robot will be able to transfer an empty box from the bottom layer to the top layer.

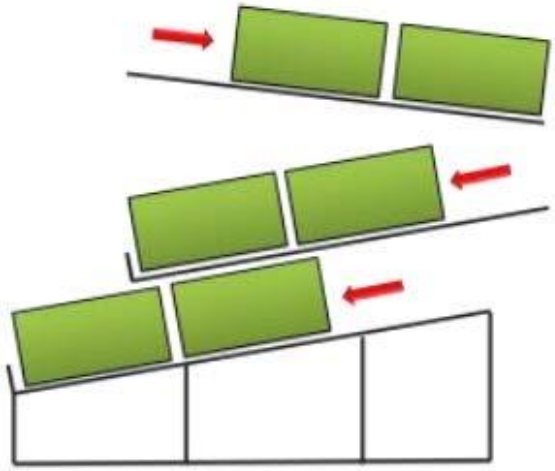


Figure 6 - Racks configuration [5].

On **Annex B** there are references for the several types of racks used on the logistic supermarket of the use case. It is possible not only to consult the diagram of the physical constitution of the set of racks, but also its dimensions.

[1] Kanban is a system to control the logistical chain from a production point of view, and is an inventory control system. Kanban was developed by Taiichi Ohno, an industrial engineer at Toyota, as a system to improve and maintain a high level of production. Kanban is one method to achieve Just in Time production.

2.5 – Operations

The objective of the STAMINA robotic fleet is to collect the 6 parts which compose the use case previously described in the logistic supermarket according to production needs and constitute a kit that will be inspected and delivered to the factory's logistics transportation device. The communication between the STAMINA robot with the PSA Peugeot Citroen's Enterprise Resource Planning (ERP) and Manufacturing Execution System (MES) allows an autonomous check of the production needs of the factory in real time. This allows the robotic fleet to allocate resources in order to fulfil the kitting task needs. The integration between the STAMINA robotic system and the PSA's ERP/MES will be further addressed in a subsequent subchapter.

The STAMINA robot fleet will operate in an environment where human operators work, mainly doing the reposition containers. As safety is one of the major concerns of the project, the robotic fleet should impose no threat to human operators. The robot will be able to navigate, operate safely and predict human motions. Besides, the workers must feel comfortable when working in close proximity with the STAMINA robot.

The Takt Time[2] of the production line is 65 seconds, and, as such, ideally, a complete kit should be delivered in that time. Due to the short time available for composing each kit, it is essential that all tasks that lead to it are optimized. The evaluation of the cycle time of the STAMINA robot to complete the mission of building a kit will allow an extrapolation of how many robots are needed for the fleet.

The first task of the mission delivering the mission to a specific robot in the fleet. The allocation is based on a "shopping list" composed by the different part variations which constitute the kit. In the current situation, the PSA shopping list is called *fiche accompagnement* (FAC), and it is generated by the factory's enterprise information system (ERP/MES). In the present solution, the list is printed and delivered to a human operator, which has the task of making a kit based on the needs expressed on the list. On the STAMINA solution, the robot will be able to communicate with the interface layer of the Enterprise Information System (EIS) and get the 6 part variants which compose the kit, creating a robot mission plan accordingly. In case of anomaly, as a wrong mission number or an unknown part reference, the robot should be able to alert the production manager.

---

[2] Takt time derives from the German word *Taktzeit*, and it represents the ratio between the available time for production and the amount of time that a system takes to manufacture a product.

The second task of the kitting mission is travelling to a designated area where the robot will load an empty kitting box to the platform mounted on the base of the robot. The empty kitting box will be transported by the robot and it will allow the construction of the kit by the robotic arm. The kitting box is composed by many separators which allow a better containment of the parts. The separators were designed aiming to reduce the number of accidental collisions between different parts and to reduce the probability of parts being mixed.

After getting an empty kitting box, a sequence of successive execution takes place. The sequence is composed by 5 steps and it is repeated once for each part that composes the kit.

The first iterative task is to navigate towards the location of the box or pallet that contains the reference of the part according to the shopping list. Due to the short cycle time, it is crucial to optimize the robot path, promoting a shorter itinerary with less maneuvers. The position of the STAMINA robot in front of the rack or pallet should be precise enough to allow the robotic arm to reach the part to be picked.

Assuming that the robot is positioned in front of the designated box or pallet, the sensory system of the robot scans the container with the objective of recognizing and locating the part needed. The robotic system should be smart enough to select the best part to be picked according to its optimization criteria.

The next step on the iterative process is to inspect the handled part and check its compliances with the information stored in the system. By comparing the scanned part with the information stored in memory, the inspection system will deliver valuable information on the quality and traceability of the part. One of the main objectives of the part inspection is to recognize and alert possible mistakes made by the logistics delivery system during the delivery of the container. If such a mistake is detected, the part is marked defective and it should be discarded to a specific container.

The following step on the iterative process is picking the part, done by the robotic arm. Based on the coordinates of the part in the robot frame, a grasp and motion planning will be generated and executed by the robot and gripper.

Finally, the last iterative step is to place the part in the kitting box. Following the picking task, the robot will generate a collision-free motion plan to place the part in the kitting box on-board. As previously stated, the kitting box will be composed by 6 designated compartments divided by separators, in order to avoid risk of collision and mixing between parts while moving the robot.

The kitting box is considered completed when the iterative process described is repeated 6 times, one for each part (engine pipe, engine support, thermal shield, alternator, starter and air conditioning compressor). Once the kit is marked as finished, it is necessary to inspect the complete kit, in order to check if the parts are contained in the appropriate compartment. Once the kit has been inspected and meets the quality requirements, it can be sent to the assembly line. In the event of an incorrect placement of one or more parts, a human operator should be notified by an alert.

The last step of the mission is the transportation of the kitting box to an appropriate area. After navigating towards the location of delivery, the robot will evacuate a complete kit and load an empty kitting box for its future mission.

In the next figure it is illustrated the flow of a normal operation of a STAMINA robot.

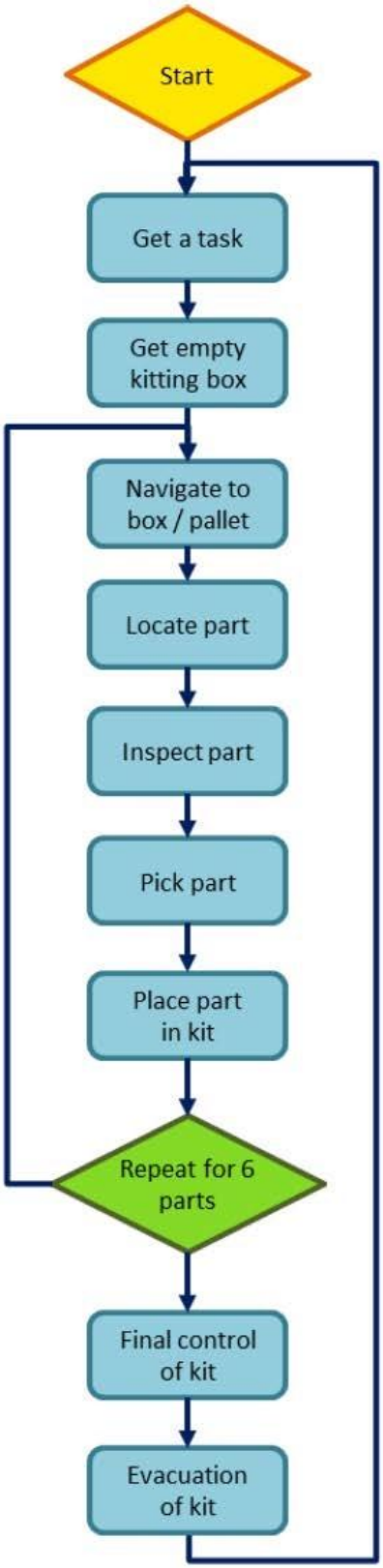


Figure 7 - STAMINA normal operation flow [5].

2.6 - Information Model

The connection between the PSA Peugeot Citroen Enterprise Information System (EIS) and the STAMINA robotic fleet system is made through a structure named **Logistic Planner** [10].

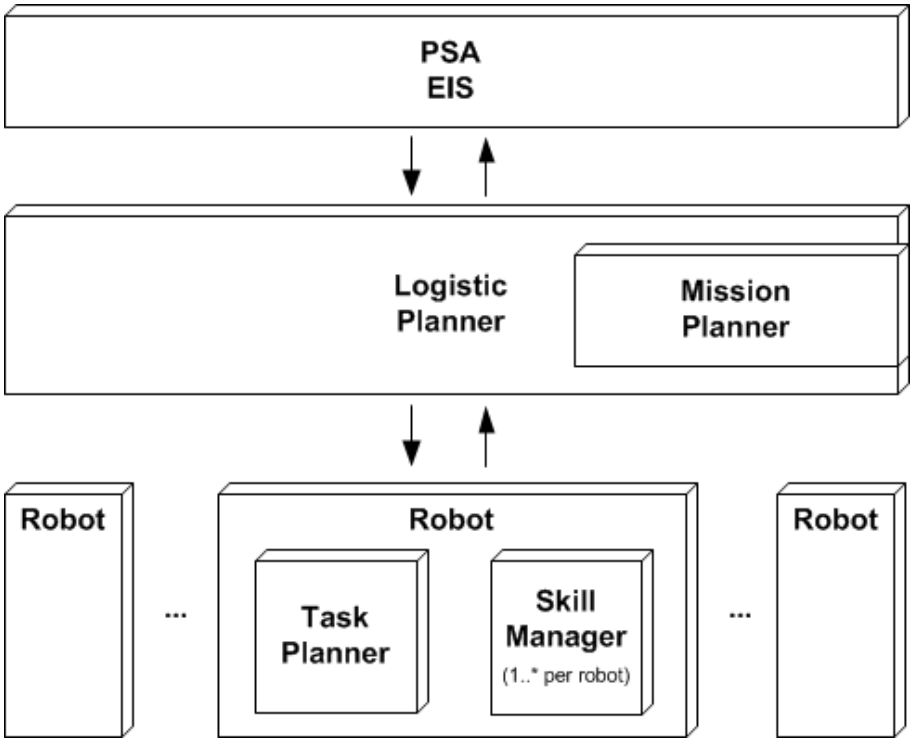


Figure 8 - STAMINA Abstract Architecture [10].

The Logistic Planner, shown in the last figure behaves as a bridge between the STAMINA system and the EIS by converting the semantics and the types of data structure transferred between systems. As such, the Logistic Planner is responsible for encapsulating the integration of the STAMINA system components with the EIS.

Besides the task of coordinating data transfers with the EIS, the Logistic Planner is also responsible for creation and coordination of robotics missions and tasks to the STAMINA robotic fleet, through an auxiliary structure, the **Mission Planner**. The Logistic Planner receives inputs and propagates the results of its actions to the PSA Peugeot Citroen EIS.

In order to successfully accomplish the goals of the kitting process, the Logistic Planner receives from the EIS a large set of information about the parts, containers, kitting supermarket layout and kitting necessities. It is expected that the Logistic Planner can retrieve all the necessary information to a successful accomplishment of its objectives prior

to its execution, organizing the information as it is best suited for the purposes of the project.

Although a large amount of data should be extracted from the EIS prior to execution, a frequent interaction between the Logistic Planner and the EIS is expected, since the kitting orders are produced on the higher enterprise levels, based on the factory necessities, and then transmitted to the Logistic Planner, responsible for coordinating the robotic fleet movements. The Logistic Planner, as previously stated, will also have the responsibility of receiving from the robotic fleet information about the success or failure of the kitting missions and propagate the information to the EIS.

2.7 - Software Integration

Software integration represents the connection between different computing systems and algorithms, providing different functionalities, so that they act as a coordinated whole. In a project like STAMINA, software integration is a crucial tool for cooperation between partners, since it makes possible that different partners can contribute with small parts. Once grouped, the small batches of work can constitute the final complete system.

By creating a structured platform, the software integration aims to simplify the process of creation of new algorithms and also provides the necessary tools for code reuse. With a structured platform, the import of external algorithms not specifically coded for the system should be straightforward and flexible.

On the STAMINA project, the functionalities which composes the robotic fleet are implemented based on a task oriented ideology. The fundamental element associated with the task oriented programming is the **skill**. The core idea of robot skills is that they are fundamental software building blocks, concatenated to form a complex task [11]. A skill is developed, in a general sense, to mimic a human action, such as *"pick"*, *"place"* or *"move"*.

The structure of a skill allows for the creation of a level of abstraction which simplifies the task of commanding or programming the robot by a human operator, allowing an autonomous planning based in a semantic model of the world, where exists a group of skills available to the programmer. In order for the skills to be useful for task-level programming, they must both change the world state through sensing or action, and be self-sustained. The

concept of self-sustainability of a skill implies that each skill should be parametric in its execution, able to identify if it can be executed in a specific state of the world, and able to verify whether or not it was executed successfully [12]. In order to apply the concept of self-sustainability, each skill is equipped with a set of pre-conditions that verify the practicability of the skill execution, and with a set of post-conditions that test the successful execution of the skill.

Unlike traditional robot programming, based on an object location and pose, the skill concept is object-centric, in the sense that physical objects are provided as input parameters, rather than coordinates. This makes possible the use of the same skill on different kinds of objects in different scenarios.

On the STAMINA project, an architecture named SkiROS (*Skill Based System for ROS*) was implemented. SkiROS is a repository completely coded in C++, which implements a layered architecture where each layer is developed as a stand-alone package, which may share a few dependencies with other layers [12]. This architecture allows the creation of a group of parameters and interfaces that aims to normalize the creation of skills and consequently the integration of software developed by different partners.

On this layered structure, the task layer deserves special attention, since it is where the skills are concatenated by human programming or by a mission issued by the enterprise systems, in order to generate complex robotic behaviors. The task layer interacts with the skills layer, allowing an easy software integration between partners, without dependencies on low-level SkiROS layers [12].

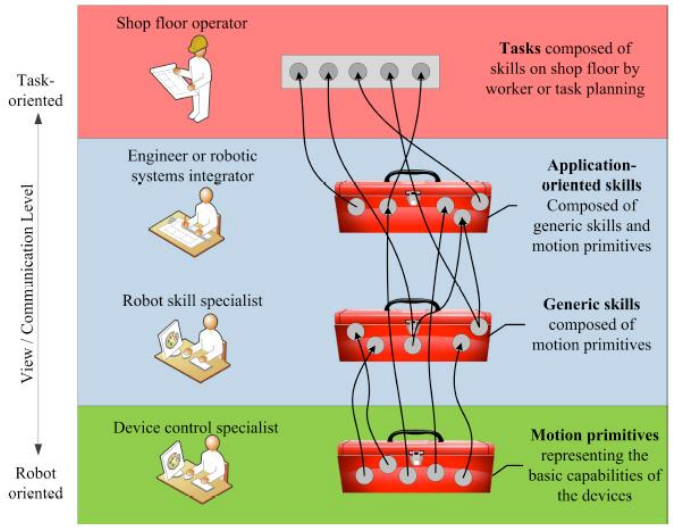


Figure 9 - Task Oriented Programming Structure [12].

All available skills are coordinated through a skill manager, which is implemented by processes running on board the robot. The skill manager is responsible for loading the available skills, checking if the robot has the resources necessary for execution and communicating the information over the network [12].

On the final solution, there will be several skill managers on the network. Each component such as a manipulator or a navigation platform will have its own skill manager, that will list the available skill set for the respective component. Every skill manager node in the network will offer an Application Programming Interface (API) composed by a service, responsible not only for querying the current list of available skills and their semantics, but also to activate a skill execution block [12]. The service described has a behavior similar to a function, as it can receive inputs and send back outputs.

The concept of task associated with this paradigm is based on the object-centric execution of skills, which results in a modification of the world, either by action or sensing. As such, in order to secure the coherent behavior of the task execution by the robotic fleet, it is necessary to implement a coordination structure which works as a data base for the location of the objects that constitute the kitting supermarket.

This database, which models the world where the robotic fleet will execute its activities, is called **World Model**. There is a strong connection between the skills and the World Model. Since the skills are the action performed by the robotic fleet in order to complete its missions, it will need input information about location of the parts, as well as information about the location of racks, boxes, other STAMINA robots and obstacles present on the kitting shopping. The skill execution result in a change in the state of world, as such, it is also required to update the World Model with information derived from the result of an action or sensing skill execution.

This constant update of the World Model makes it a dynamic structure, resulting in potential coherence problems between different robots of the fleet. The concept of World Model, on an implementation level will be further analyzed on the following subchapter, leaving a theoretical analysis for the respective state of the art chapter. The dynamic and coherence question will also be addressed in a subsequent section of this document.

2.8 - World Model

Within the scope of the STAMINA project, the World Model is a database composed by the objects which populate the universe of action of the robotic fleet. Those objects are all the racks, pallets, boxes and parts which are present on the kitting supermarket, as well as other robots of the fleet and obstacles.

The STAMINA robots should use the World Model as a reference to the position of the racks and pallets which contain the parts used to make a kit. The fleet will also have the ability of updating the World Model with information derived from its sensors. Each STAMINA robot is equipped with a camera system, allowing the robot to have a perception of its surroundings. This information will be used not only to navigate, but also to update the World Model so that other robots on the fleet can have a sense of its changes. The update process makes the World Model a dynamic structure.

Although the notion of representing the World Model as a dynamic structure is considered a necessity within the STAMINA project and the majority of the mechanisms for real time updates were already implemented, the World Model was, in the initial stages of development, modeled as static structure, in order to comply with the first phases of testing.

The World Model, as a concept, is a database which organizes all physical objects on the scene in a graph, specifying each object location, orientation and dimension in a 3D environment that constitutes a graphical representation of the scene to model. The World Model implementation was motivated by the scene graphs used in computer graphics. The World Model is defined as a graph, where nodes are world elements, *i.e.*, objects of the physical universe, and edges define a relationship *contain*, which evaluates to true or false depending on whether a parent element contains the child element. The analysis of the World Model as a structure modeled by a scene graph will be further analyzed at the State of the Art chapter.

In order to successfully implement the World Model concept, there are a number of preliminary information that needs to be either sensed by the robot or defined beforehand. The location of each rack and box must be mapped prior to the execution of the operation. Pallets and objects should be detected from the vision system of the robot, although some prior knowledge about the area where they are located is necessary for planning.

Chapter 3

State of the Art

This chapter has the objectives of providing a theoretical approach to some of the technological concepts that are the bedrock to support some of the mechanisms developed during the project. Besides focusing on presenting the reader with an overview on some of the concepts used, this chapter also addresses the state of the art of some of the technologies used.

The State of the Art chapter is divided in six subchapters, each reflecting a key technological area.

The first subchapter addresses the World Model as a data structure, analyzing the concept used to model it. The second subchapter presents the key concept on spatial change detection, the point cloud. Subchapters three, four and five analyze some of the technologies used to manipulate and process point clouds and to construct dynamic representations of spatial data, such as the PCL library, Octrees and the OctoMap framework. Finally, the last subchapter addresses the base framework for most of the development made during this project, the Robot Operating System (ROS).

3.1 – World Model and Scene Graphs

On the STAMINA project, the **World Model** is the data structure that aims to model the state of the logistic supermarket of the use case. This representation should include racks, pallets, parts, robots and obstacles, in order to create a reliable database of the location of objects which compose the work area of the STAMINA robotic fleet. The update mechanism of the World Model makes it a dynamic representation of the current state of the logistic supermarket.

The definition of World Model, on the scope of the STAMINA project is based on the concept of scene graph from computer graphics. This subchapter is aimed at making a small theoretical approach to the scene graph concept.

A **Scene Graph** is a data structure used to hierarchically represent relationships between transformations applied to a set of objects on a tridimensional scenario. In the most general form, a scene graph can be used to store information that have some degree of hierarchical relationship. It also provides a convenient way of representing logical groups of objects formed using their spatial positions or attributes [13].

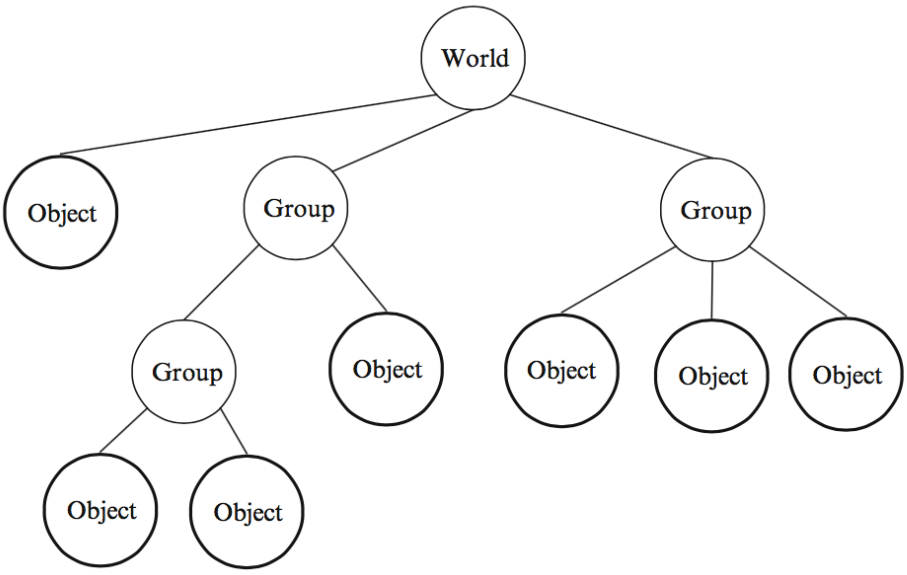


Figure 10 - Scene Graph structure [13].

The structure of a Scene Graph depends of the information it stores, even though there are several concepts which are transversal to every representation. A Scene Graph, as a hierarchical structure is based on the hierarchical relationship between its nodes. As such, there are four different kinds of nodes:

1. **Root Node**. Represents all the set of objects on the tridimensional space modeled. It is the origin of all structure and, as such, it is usually called *"World"* or *"Universe"*. The root node constitutes a special type of a group node.

2. **Group Node**. It is defined as an internal node of the tree, *i.e.*, it descends from another node and it contains a number of descendant nodes. This kind of node represents a logical grouping of objects on the modeled universe. This kind of node does not store any specific information about objects which derive from it, but it can store attributes that apply to the whole set, as is an example the visibility.

3. **Transformation Group Node**. This kind of node is a special case of the Group Nodes, used to represent geometrical transformations on all the descendant objects. This kind of transformations could be specified within the Group Nodes, and, as such, sometimes there is no need to create a new set of nodes specifically to represent geometrical transformations.

4. **Leaf Node**. This kind of node is used to represent an object or a component of an object. The "leaf" name is based on the fact that these nodes are located on the extremities of the tree structure. They usually specify the geometrical information and can sometimes store some attributes that complement the model. Cameras or light sources can also be represented by this kind of nodes.

The tree structure of a scene graph allows a property associated with a group node to be inherited by all of its child nodes. As such, a transformation applied to a group node can be considered as also applied to all its children [13].

A Scene Graph is especially useful to model composite objects, *i.e.*, objects that are made of physical connections of different components, since it is easy to represent translations and rotations of composite objects. The modelling of a table can be used as an example: Initially, it is necessary to define all the elements which compose the table (legs and base), according to the local referential of the composed objects, using geometrical transformations of simple geometrical figures (for example, a cube). On a next stage, the elements need to be placed on the right positions according to themselves, in order to properly model a table. On this stage, the hierarchical relationship between the parts that compose the table are defined.

After the hierarchical structure is successfully defined, if the table changes position on the scene, the geometrical transformation is propagated to the elements that compose the table, in order to maintain the integrity of its elements.

On the previous example, the main concept of the Scene Graphs is exposed: the hierarchical relationship between nodes that compose objects. On the previous example, instead of thinking of each element that composes the table as an independent instance, a Scene Graph groups them in a hierarchical structure, in order to preserve their absolute position between themselves, on the event of a geometrical transformation being applied to the object. As such, a transformation applied to the transformation group of the table will automatically be applied to all the components in the transformation group [14]. A transformation applied to one part of an object often cascades with the transformation applied to the adjacent interconnected parts [13].

Geometric transformations are described by matrices, composed by a rotation, scale and a translation component. The transformation of one node relative to another can be easily obtained from a scene graph [13]. The **Model Transformation Matrix** of an object gives the composite transformation which converts points from the local coordinate space of the object to the world coordinate space defined by the Root Node [14].

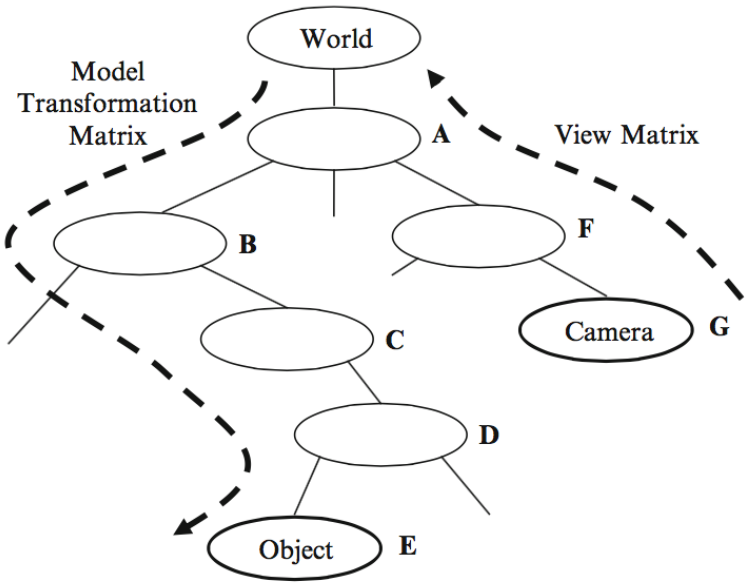


Figure 11 - Transformation Matrix determination of an object represented by a leaf node on a scene graph [13].

The Model Transformation Matrix of an object can be obtained by collecting all matrices along the path from the root node to the leaf node representing the object. Along the path, at each node, the matrix is post-multiplied by the transformation matrix at that node. The process is illustrated on the previous figure, where node transformation matrices are denoted by letters A to G. The Model Transformation Matrix of the object on figure 11 is given by $A \cdot B \cdot C \cdot D \cdot E$ [13].

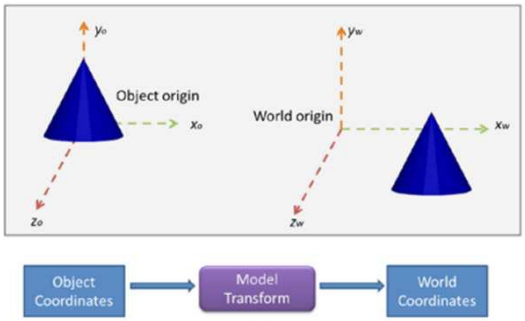


Figure 12 - Model Transform [14].

The transformation given by the Model Transformation Matrix are applied from the leaf node upward to the root node of the scene graph, by applying translation, rotation and scaling operations. As such, the Model Transformation Matrix is used to place and orientate the object on the global coordinate system.

Leaf nodes can also be used to represent fictitious objects such as light sources and cameras [13]. In figure 11, the matrix $A \cdot F \cdot G$ represents the transformation from the coordinate system of the camera to world coordinates. The inverse matrix, $(A \cdot F \cdot G)^{-1}$ is used to transform a point from world coordinates to camera coordinates. This matrix is called the **View Matrix** [13].

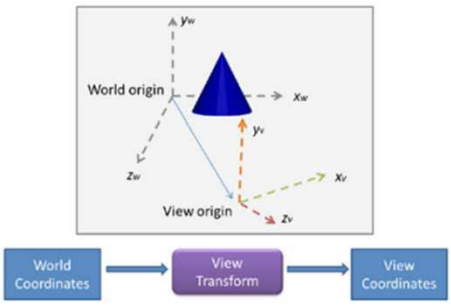


Figure 13 - View Transform [14].

The transformation given by the View Matrix moves the origin of the world coordinate system to the position of the scene observer, *i.e.*, to the camera position. As such, the View Matrix is used to place and orientate an imaginary camera, which is used to render only the objects that can be seen through the camera. In this transformation, only translation and rotation operations are used.

The combined **Model-View Matrix** is obtained by combining the two transformation matrixes previously exposed (Model Transformation Matrix and View Matrix). The Model-View Matrix transforms the object's local coordinates to camera coordinates. Based on the previous example, the Model-View Matrix is given by $(A \cdot F \cdot G)^{-1} \cdot A \cdot B \cdot C \cdot D \cdot E$, i.e., by multiplying the View Matrix with the Model Transformation Matrix [13].

Multiplying the vertices of an object with the Model-View Matrix will result on the direct transformation of the object coordinates, defined in the local coordinate system of the object to the camera coordinates [14].

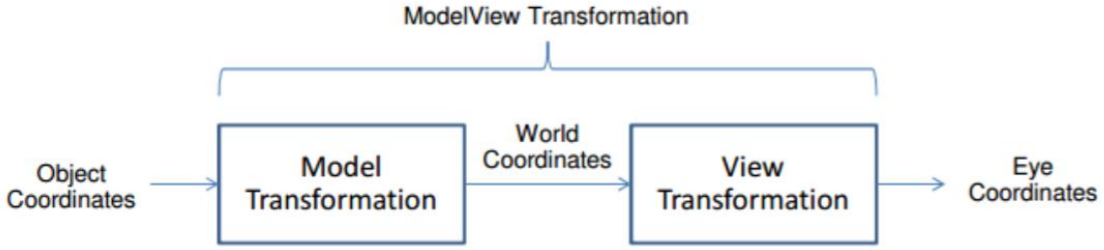


Figure 14 - Model-View Transformation [14].

The World Model, as a data structure is stored on **JSON** files. JSON is also used to transfer the World Model from the Logistic Planner to the other STAMINA components.

JSON (JavaScript Object Notation) is a lightweight data-interchange open standard format [15]. JSON, although language-independent, derived from the JavaScript scripting language, and was designed to be easily read by humans as well as easily parsed and used by machines [16]. JSON data objects mostly consist on attribute-value pairs, and it is used as an XML (Extensible Markup Language) alternative, primarily to transmit data between a server and a web application [15].

3.2 – Point Cloud Architectures

A Point Cloud is defined as a data structure commonly used to represent three-dimensional data [17]. Point Clouds represent the basic input data format for 3D perception systems, and provide discrete, but meaningful representations of the surrounding world [18].

By definition, a point cloud is a collection of 3D points in some coordinate system. In a three-dimensional coordinate system, these points usually represent the X, Y and Z geometric coordinates of an underlying sampled surface, often intended to represent the external surface of an object. The X, Y and Z coordinates of any point on the collection are given with respect to a fixed coordinate system, usually having its origin at the sensing device used to acquire the data [18].

Point clouds can be acquired from hardware sensors such as stereo cameras, 3D scanners or Time-of-Flight cameras [17]. These devices measure a large number of points on an object's surface, outputting the point cloud as a data file. It is also possible to generate a point cloud from a computer program synthetically. Each point of the data set represents the three-dimensional distance from the acquisition device to the surface that the point has been sampled on. The point cloud obtained represents the set of points that the acquisition device has measured of a three-dimensional space on a given time.

There are two most used approaches to measure distances and converting them to 3D points. The most common way are Time-of-Flight systems, where a sensor measures the delay until an emitted signal hits a surface and returns to the sending device, estimating the true distance from the sensor to the surface. Knowing the speed with which a ray propagates it is possible to estimate the distance it travelled by measuring the exact time when the ray was emitted and when the signal returned. It is also possible to use triangulation techniques, where two different sensors are used at the same time. In this situation, the distance is estimated by means of connecting correspondences seen by the two different sensor [18].

Examples of point cloud applications range from robotics applications, such as robotic navigation, to medical imaging and the creation of digital elevation models of the terrain, where point clouds are employed to generate a three-dimensional model of an urban environment.

In the past, systems for creating a high-quality three-dimensional representation of the world were expensive, with costs that exceed the budget of many robotics projects. With the advent of new, low-cost 3D sensing hardware, such as the Microsoft XBox 360 Kinect, three-dimensional perception suddenly became a cheaper solution to many robotics problems. The new sensing hardware provides real time point clouds as well as 2D images for a low-cost price, and, as such, it is expected that most robots in the future will be able to use three-dimensional perception to solve many problems [19].

Point cloud representations are able to store much more information than just the 3D positions of points as acquired from the input sensing device. One example is distances from the sensor to the surfaces in the world, which can be stored as a property of each resultant 3D point [18].

In order to understand and analyze the geometry around a certain point, most geometric processing algorithms need to discover a collection of neighboring points around the query point, representing the underlying scanned surface through sampling approximations. In order to allow the search of neighbor points in fast ways, the mapping systems needs to employ mechanisms for enabling searches of points without re-computing distances between each other every time [18].

A solution for the problem previously discussed is to partition the point cloud into several chunks, such that queries with respect to the location of neighboring points can be answered fast, by using spatial decomposition techniques. Most spatial decomposition techniques, although different in its implementation, are based on the construction of boxes with different widths that enclose points of the point cloud. Two of the most used techniques are the kd-trees and octrees, that will be further analyzed in this chapter.

3.3 – Point Cloud Library (PCL)

One of the most recognized and used initiatives in the area of point cloud perception is the **Point Cloud Library** (PCL). The Point Cloud Library is a free, large scale, cross-platform and open project for point cloud processing, being developed by a large number of engineers and scientists from many different organizations, geographically distributed all around the world. [17] PCL is currently supported by an international community of robotics and perception researchers [19].

PCL can be seen as a modular library, where its components are split into a series of smaller code libraries which can be compiled separately, resulting in a graph of code libraries. PCL is currently supported on Linux, MacOS, Windows and Android/iOS [17].

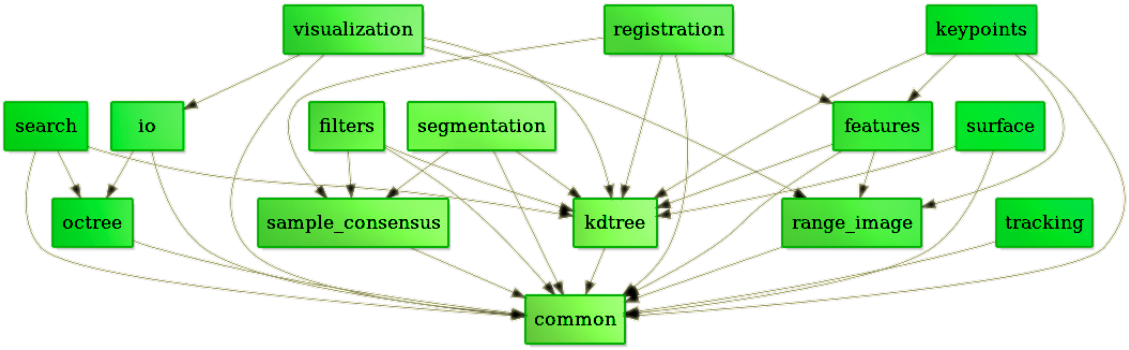


Figure 15 - PCL Architecture [17].

From an architecture perspective, PCL is a C++ library for point cloud processing [19]. PCL proposes a framework of numerous point cloud processing algorithms, such as **filtering**, **feature estimation**, **surface reconstruction**, **registration**, **model fitting** and **segmentation**. Implementation of the algorithms are compact and clean, since each set of algorithms is defined via base classes that attempt to integrate all the common functionality used throughout the entire pipeline [19].

3.3.1 - Basic Structure

The basic data type in PCL is a ***PointCloud***, A *PointCloud* is a C++ class which contains data fields for characterizing a stored point cloud. There are two different kinds of point cloud datasets, the organized and the unorganized point cloud dataset.

An **organized point cloud dataset** is a point cloud that resemble an organized structure like a matrix of an image. In this kind of organized structure, the data is split into rows and columns. The advantage of an organized point cloud dataset is that by knowing the relationship between adjacent points, nearest neighbor operations are much more efficient, thus speeding up the computation and lowering the costs of certain algorithms [20].

An **unorganized point cloud dataset** is characterized by non-existing point references between points from different point clouds due to a multitude of factors, such as varying size, resolution, density and point ordering [21]. Unorganized point cloud datasets obtained from three-dimensional shape acquisition devices usually are incomplete and noisy due to occlusions and physical limitations of the scanners. In the region that are invisible to the cameras, the surface of the scanned model is not covered by sample points, which might lead to an imperfect shape on the surface reconstructed by most reconstruction algorithms [22].

The main difference between the two different kinds of point cloud datasets has to do with the way the points are stored in memory. On the organized point cloud datasets, the memory layout closely related to the spatial layout as represented by the X, Y and Z value. It is expected that most algorithms that work on unorganized point cloud datasets also work on organized point cloud datasets, although the opposite is not always correct. As the point clouds used within this dissertation project are unorganized, special care was taken when applying some of the PCL algorithmic procedures.

The **width** attribute specifies the width of the point cloud dataset in the number of points, meaning that it can either specify the total number of points in the cloud for unorganized point cloud datasets or it can specify the total number of points in a row for organized point cloud datasets. The **height** data field is used to store the height of the point in the number of points, meaning that it can either specify the total number of rows of an organizes point cloud data set or it can be set to 1 for unorganized datasets [23].

The **points** data field contains the data array where all the points of type **PointT** are stored [23]. PCL comes with a variety of pre-defined point types, ranging from structures for XYZ to more complex n-dimensional representations [24]. The most basic *PointT* data type is

the *PointXYZ*, used to represent 3D xyz information only. As previously stated, point clouds representations are able to store much more information than just the 3D position of points acquired from the sensing device. As such, there are several other *PointT* data types adding several components, in order to cope with the additional information stored. Examples of such data types are *PointXYZI* (adds intensity), *PointXYZRGB* (adds RGB information) and *PointXYZRGBA* (adds RGB and alpha information). It should be noted that the point clouds worked within this project are expected to be modeled as ***PointXYZRGBA*** data types.

Another property of the *PointCloud* class is the boolean ***is_dense*** attribute, which specifies if all the data points are finite or whether the XYZ values of certain points might contain *Inf*/*NaN* values [23].

There are two other attributes, that are usually optional, and not used by the majority of the algorithms in PCL on its current implementation. These attributes are used to specify the sensor acquisition pose in terms of its origin and orientation [20].

3.3.2 - Point Cloud Data (PCD) File Format

The **Point Cloud Data** (PCD) file format is used for storing multi-dimensional point data. Each PCD file contains a header which identifies and declares certain properties of the point cloud data stored on the file [25]. The PCD header contains several fields which are very similar to the attributes of the *PointCloud* class previously analyzed. The PCD header is followed by the data in ASCII, with its points separated by lines, spaces or tabs, or binary, a memory copy of the points vector of the point cloud.

There are several other file formats to store three-dimensional point cloud data, such as PLY, STL, OBJ and X3D. All the file formats mentioned suffer from several shortcomings when compared to the PCD file format, as none of the files offers the flexibility and speed of PCD files. The main advantages of the PCD file format include the ability to store and process organized point cloud datasets and store different data types, allowing the point could data to be flexible and efficient with respect to storage and processing.

3.3.3 - Libraries

As previously stated, in order to simplify development, PCL is split in a number of modular libraries, which can be compiled separately. In this topic, a brief analysis of some of the available point cloud libraries will be made.

The **common** library holds the data structures and methods that are used by the majority of PCL libraries. It holds the *PointCloud* class previously analyzed as well as the large number of pre-defined point types which are used to represent points, surface normal, RGB color values and feature descriptors. The common library also contains a multitude of functions for basic operations, such as calculating distance, norms, means, covariance, angular conversions and geometrical transformations [26].

The **I/O** library contains classes and functions which implement input and output operations such as writing to a PCD file or reading from a PCD file. The library also includes classes and functions for capturing point clouds from a diverse number of sensing devices.

As mentioned previously, due to measurement errors and imprecisions, certain datasets might present a large number of shadow points or outliers. As such, the **filter** library implements data filters for downsampling, outlier removal, indices extraction and projections [19].

Three-dimensional features are representations at certain points or positions in space which describe geometrical patterns based on the information available around the point. The **feature** library contains data structures and mechanisms for three-dimensional feature estimation from point cloud data. Two of the most widely used geometric point features are the surface normal and curvatures [26].

Keypoints are points in a point cloud that are stable, distinctive and can be easily identified using a defined detection criterion [26]. Some features are expensive to compute, and it would take a long time to compute them at every point of the point cloud. As such, keypoints identify a small number of locations where computing feature descriptors is likely to be most effective [23]. The **keypoints** library implements different keypoints extraction methods, that can be used to decide where to extract feature descriptors [19].

The **segmentation** library contains algorithms and functions to implement cluster extraction. If a point cloud is composed by a number of spatially isolated regions, it is easier to break the point cloud into several clusters, which can be processed independently, optimizing the process.

The main objective of the **surface** library is to implement surface reconstruction techniques which allow reconstruction of the original surface from 3D scans. There are several functions within this library that implement meshing, convex hulls, smoothing and resampling algorithms [26].

Registration is a technique which identifies corresponding points between several data sets and finds a transformation that minimizes the alignment error between corresponding points, combining the datasets into a global consistent model. The **registration** library contains functions and methods to successfully execute a point cloud registration algorithm [26].

A range image or depth map is an image whose pixel values represent a distance or depth from the sensor's origin [26]. The **range_image** library implements support classes and functions to handle images created from point cloud datasets [23].

PCL comes with its own **visualization** library, offering, for example, methods for rendering and setting visual properties of point cloud datasets, drawing basic 3D shapes on screen either from sets of points or from parametric equations.

Nearest neighbor searches are a core operation when working with point cloud data and can be used to find correspondences between groups of points or features descriptors or to define the local neighborhood around a point [26]. A **Kd-Tree** is a data structure for organizing points in a space with *k* dimensions that enables efficient range searches and nearest neighbor searches. For operations with point cloud data, mainly in three dimensions, the Kd-Tree will be three-dimensional as well. A Kd-Tree is a binary search tree with some constraints imposed on it.

On an implementation point of view, the main idea is to split the point set alternatingly by a vertical line that has half of the points of the x-coordinate (left and right) and by a horizontal line that has half of the points of the y-coordinate (below and half) [23]. As such, each level of a Kd-Tree splits the children points along a specific dimension, using a hyperplane that is perpendicular to the corresponding axis.

The **kdtree** library provides the Kd-Tree data structure which makes fast nearest neighbor searches possible.

Another type of spatial decomposition technique is the **Octree**. The Octree is a hierarchical tree data structure created from point cloud data. The root node is composed by a cubic bounding box which encapsulates all points from the point cloud. At subsequent tree level, the space becomes subdivided by a factor of 2, resulting in an increased voxel resolution. The resulting leaf nodes provides functionalities such as spatial occupancy and point per voxel checks. The Octree automatically adjusts its dimensions to the point dataset [26]. One of the biggest advantages of octree data structures is that they are easy to update and support point insertion and deletion almost natively [18].

The Octree implementation also provides efficient mechanisms for nearest neighbor searches, such as Neighbors within a Voxel Search, K Nearest Neighbor Search and Neighbors within a Radius Search. As such, PCL provides the functions needed for generating and operating Octrees in the **octree** library.

As it is an important technology within the scope of this dissertation project, Octrees will be further analyzed in a subsequent chapter.

3.4 - Point Cloud Processing

Point cloud datasets are composed by a very large number of points nowadays, in particular for dense image matching applications, where the amount of images and its resolution increases [27]. In general, it is not convenient to process raw point cloud data, since there is no connectivity information to define local neighbors [28]. As such, in order to be able to perform operations that demand knowledge on neighboring points, such as filtering, surface reconstruction, segmentation or matching, spatial decomposition techniques are used, since quick data access is essential.

As a way of meeting requirements associated with data querying, spatial decomposition techniques are used as an access structure. Octrees and Kd-Trees are examples of such techniques. While Kd-Trees have a data-driven approach, requiring less memory for the tree structure, enabling a faster data access, Octrees are based on a space-driven partitioning approach, supporting quick data update, since the tree partitioning is not relying on the current data, and thus, does not require an update of the structure when adding or removing data [27].

On the purpose of real-time mapping of a dynamic environment, the usage of octree data structures as a way of spatially decomposing the environment presents a big advantage, since octrees are easy to update and are able to support dynamic point insertion and deletion.

Being the three-dimensional analog of quadtrees, an octree is a tree-based hierarchical data structure for managing sparse three-dimensional data [29]. Both quadtrees and octrees are usually referred to as space trees and are based on the principle of recursive decomposition of the space.

The root node of an octree describes a cubic bounding box which encapsulates all points. The construction of an octree is obtained by recursively subdividing the cubical volume represented by the root node into eight congruent disjoint cubes, called octants. As the name implies, every internal (non-leaf) node has eight children. Each node on the tree spans a particular space area, expressed as an axis-aligned bounding box. The recursive subdivision process is finished when blocks of uniform value are obtained, or when a predetermined level of decomposition is reached.

In the figure below, it is possible to see the octree block decomposition of a three-dimensional object. In this simple example, it is possible to observe that each node in the tree represents a cubic and only leaf nodes of the octree store information about its occupancy, while internal nodes needs to be recursively subdivided in eight children nodes, in order to determine its occupancy. As such, each node in the octree has one father and either eight children, if it is an internal node, or zero children, it is a leaf node.

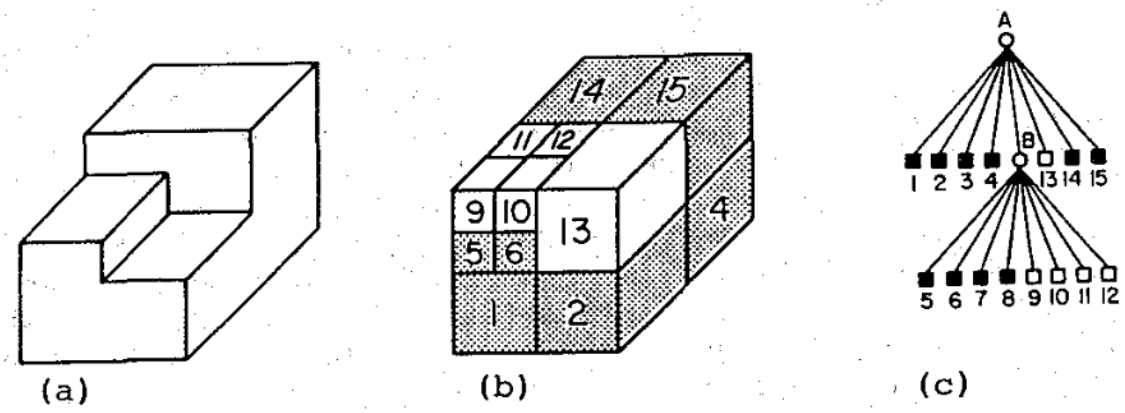


Figure 16 - (a) Example three-dimensional object; (b) Octree block decomposition; (c) tree representation [30].

3.5 - OctoMap

Three-dimensional mapping constitutes an essential component of a variety of robotic applications, ranging from underwater, flying and extra-terrestrial robots to autonomous mobile manipulators. Robotics applications often require a three-dimensional model of the environment in order to fulfill the requirements of the project.

The three-dimensional representation of the environment is a central component of most autonomous systems as it is a key component during action planning and execution. It is, therefore, essential that the mapping is as efficient as possible, not only with respect to access times, but also with respect to memory consumption. On most robotic applications, memory consumption is often the principal bottleneck of three-dimensional mapping systems [31]. In order to have an efficient representation of the environment it is important that the model is compact in memory, so that a robot can keep the representation of even large environments in its memory, perform access and query operations, and efficiently transmit it between multiple robots.

Raw point cloud processing on its own is not a memory efficient way of processing three-dimensional environment mappings, since the memory consumption increases with the amount of measurements, without an upper bound. The lack of connectivity information between points to define local neighbors [28] has also been mentioned as a disadvantage. Besides the previously analyzed drawbacks, the other major disadvantages of the method are that neither free space nor unknown areas are modeled, and dynamic objects cannot be handled directly [31]. These factors make the direct representation of an environment using point clouds only suited for high precision sensors in static environments and when the representation of unknown areas is not relevant.

Along the years, several alternative approaches to represent three-dimensional environments in robotics have been proposed, such as elevation maps, multi-level surface maps and OctoMaps. Although Elevation Maps [32] and Multi-Lever Surface Maps [33] are efficient in its memory handling, they are not able to represent unmapped areas either.

In robotics mapping, to model three-dimensional environments in a discrete way, a method using grid of cubic volumes of equal size, called voxels has been proposed by [34] and [35]. However, the fixed grid size method is still not memory efficient enough. In a situation where the mapping has to accommodate fine resolutions or when the environment is a large-scale outdoor scenario, a fixed grid often translates to an excessive use of memory space. In

this case, the grid map needs to be initialized so that it is at least as big as the bounding box of the mapped area, regardless of the actual distribution of map cells in the volume [31].

Several robotic applications require a probabilistic representation of the environment, modelling free, occupied and unmapped areas. The process of creating three-dimensional representation used by mobile robots often produce measures afflicted with uncertainty, due to the errors on the three-dimensional sensing devices used. On the scope of robotics applications, a probabilistic representation of the environment allows the handling of sensor noise and dynamic changes, as well as the integration of data from multiple sensors and of multiple robots.

The unknown areas representation is essential in autonomous navigation tasks, where a robot is expected to automatically plan collision-free and efficient paths to achieve its goals. Being able to represent unmapped areas is very important on a robotic application, since the information about unknown regions of the environment can potentiate either its autonomous mapping, or the need to be avoided by the robot.

OctoMap is an integrated framework based on octrees for the representation of three-dimensional environments which combines the advantages of previously discussed approaches to three-dimensional environment modeling, allowing for efficient and probabilistic updates of occupied and free space while keeping the memory consumption at a minimum. Compressions methods ensures the compactness of the resulting model [31].

OctoMap is fully documented, open source framework implemented as a self-contained C++ library for three-dimensional mapping. Since its implementation in 2010 [36], the framework has suffered various updates and was used in several robotics research projects. The library has been successfully integrated with the Robot Operating System (ROS) and it is supported in several platforms, such as Linux, Windows and MacOS.

The OctoMap framework uses a tree based representation, similar to octrees. In its nature, octrees can be used to model a true or false condition, usually representing the occupancy of a cell. In an octree, if a certain volume is measured as occupied by the sensing system, the corresponding node is initialized. All uninitialized nodes could be either free or unknown, in this Boolean scenario, making the model ambiguous and not suited for a three-dimensional representation of the environment.

OctoMap proposes a solution to resolve the ambiguity, by representing the three possible scenarios: occupied, free and unknown volume. An occupied space corresponds to the intersection point of a distance sensor such as a laser range finder or a depth camera, with the environment. The free volume is obtained by the observed space between the sensor and the end point. Areas that are not initialized implicitly model unknown space [31].

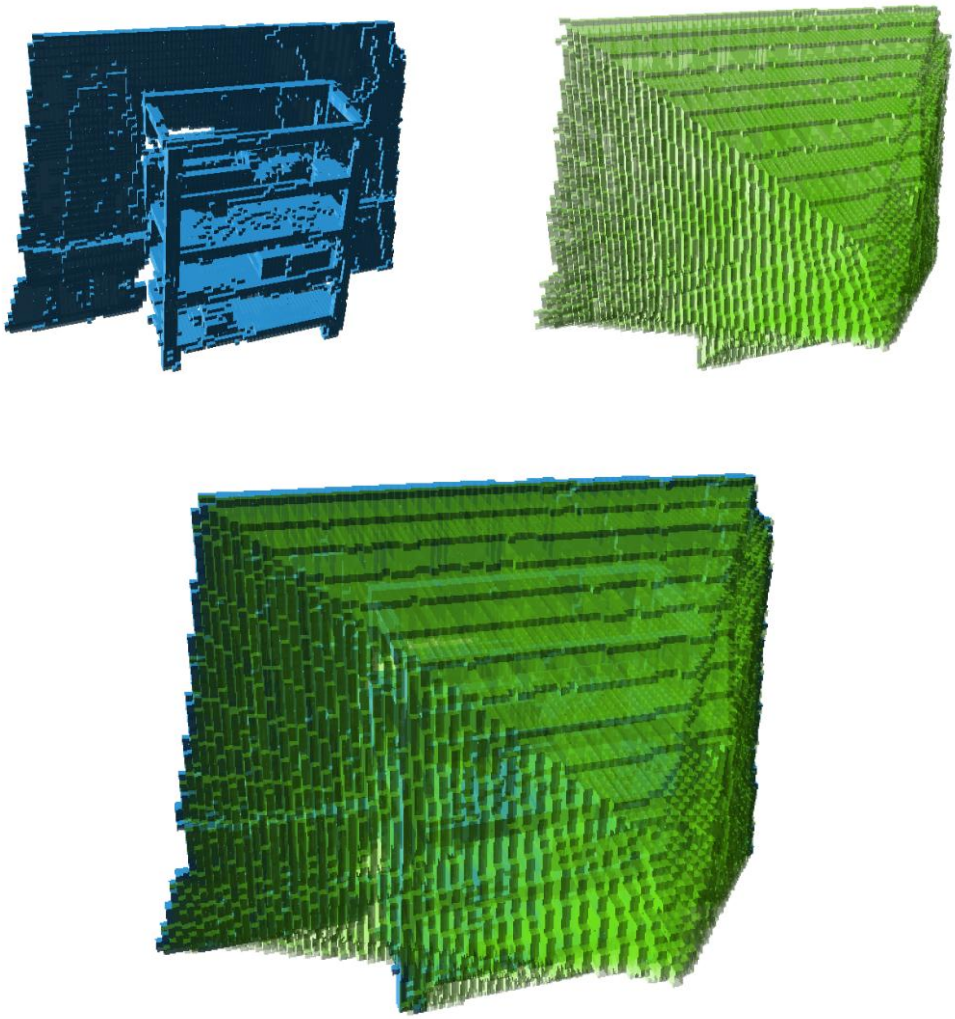


Figure 17 - Octree map generated using the OctoMap framework. The top left image represents the occupied voxels, the top right image represents the freevoxels, and the bottom image represents the free voxels (green) and the occupied voxels (blue).

By using the proposed algorithm for creating the nodes previously discussed, OctoMaps are able to successful trim the resulting tree, since if all children of a node have the same state, either occupied of free, they can be grouped. This allows for a compact representation of the tree, substantially reducing the number of nodes needed to maintain the tree.

The OctoMap framework integrates sensor readings using occupancy grid mapping modeled as a recursive binary Bayes filter [37]. The update formula for the probability $P(n \mid z_{1:t})$ of a leaf node n to be occupied given the sensors measurements $z_{1:t}$ is estimated according to equation 3.1 [31].

$$P(n \mid z_{1:t}) = \left[1 + \frac{1 - P(n \mid z_t)}{P(n \mid z_t)} \frac{1 - P(n \mid z_{1:t-1})}{P(n \mid z_{1:t-1})} \frac{P(n)}{1 - P(n)}\right]^{-1} \tag{3.1}$$

On the update equation, the parameter $P(n \mid z_t)$ is specific to the sensor and represents the probability of voxel n to be occupied given the measurement z_t by the sensor system. The update formula is influenced by the previous estimate, $P(n \mid z_{1:t-1})$, a prior probability $P(n)$ and the current measurement z_t [31].

Given the common assumption of a uniform prior probability leads to $P(n) = 0.5$ and using the log-odds notation, equation 3.1 can be rewritten as it is proposed in equation 3.2. This new representation of the equation of the update rule allows for faster updates since multiplications by additions [31].

$$L(n \mid z_{1:t}) = L(n \mid z_{1:t-1}) + L(n \mid z_t) \tag{3.2}$$

In equation 3.2, $L(n)$ is defined as shown in equation 3.3.

$$L(n) = \log\left[\frac{P(n)}{1 - P(n)}\right] \tag{3.3}$$

In the event of a pre-computed sensor model, the logarithm do not have to be computed during the update step.

While using the OctoMap in robotics applications, it is common practice to apply a threshold on the occupancy probability. The threshold has the function of defining two discrete states, by assuming that a certain voxel is occupied if the occupancy probability is above a defined threshold or is free otherwise [31].

Assuming that free and occupied measurements have equal probabilities in the sensor model, with the update equation previously presented, in order to change state of a voxel, it is necessary to integrate as many observations as have been integrated to define its current state. If a certain voxel was marked as free k times, then it has to be observer occupied at least k times before is considered occupied to the threshold [31]. This property leads to a potentially unwanted delay in the update of the state of the voxel, specially while dealing with robotic systems working in dynamic environments.

Equation 3.4 [38] proposes a clamping policy that ensures updatability by defining an upper and lower bound on the occupancy estimation. The modified update formula limits the number of updates that are needed to change the state of a voxel. Equation 3.2 is therefore transformed into equation 3.4, where l_{max} and l_{min} represent the upper and lower bound on the log-odds value.

$$L(n \mid z_{1:t}) = \max(\min(L(n \mid z_{1:t-1}) + L(n \mid z_t), l_{max}), l_{min}) \tag{3.4}$$

By applying the clamping update policy in the OctoMap update formula, a quick reaction to changes in the environment and the ability to compress neighboring voxels with pruning are secured [31].

3.6 – Robot Operating System (ROS)

The **Robot Operating System (ROS)** is a flexible framework for writing robot software. It is composed by a collection of tools, libraries and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms [39].

ROS is an open-source, meta-operating system for robots. It provides the services expected from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management [40].

Many modern robot systems rely on software that is divided into many different processes, running across several different computers. There is an implicit need for collaboration and communication between several processes, as it is a good practice to divide the robotic software into small, stand-alone parts that cooperate to achieve the overall goal. It is also advisable to implement collaboration and communication mechanisms when there are multiple computers controlling subsets of the robot's sensors or actuators, and when various robots attempt to cooperate on a shared task [41].

The scope and scale of robotic applications continues to grow, resulting in different robotic system which perform with some degree of similarity. Common tasks such as navigation, motion planning, and mapping could be reused in different robotic applications and scenarios.

Most robotics software projects contain drivers or algorithms which could be reusable outside the project but, due to a variety of reasons, much of this code has become entangled with the middleware, being very difficult to extract its functionality outside of its original context [42].

ROS is composed by a distributed framework of processes, which allow executables to be individually designed and loosely coupled at runtime. The corresponding processes can be grouped into Packages and Stacks, which can be easily shared and distributed. The ROS architecture also supports a system of code repositories, enabling distributed cooperation. As such, the primary goal of ROS is to support code reuse in robotics research and development.

ROS has three levels of concepts [43]. The **ROS Fileystem Level** concepts mainly cover ROS resources that can be stored on disk, such as Packages. The **ROS Computation Graph Level** is the peer-to-peer network of ROS processes that are processing data together. The **ROS Community Level** concepts are ROS resources that enable communities to exchange software and knowledge, such as Distributions, Repositories, Wiki, and Mailing Lists.

A **Package** is the main unit for organizing software in ROS. A package may contain ROS runtime processes (nodes), ROS-dependent libraries, datasets, configuration files, third-party pieces of software, or anything else that is usefully organized together [43]. A ROS package is composed by a directory which contains an XML file describing the package and stating any dependencies [42]. The package structure allows collaborative development between partners. A collection of ROS packages is a directory tree with ROS packages at the leaves, and as such, a ROS package repository may contain various subdirectories.

The fundamental concepts of the ROS implementation are nodes, messages, topics, and services. **Nodes** are processes that perform computation. A ROS system is meant to operate in a distributed way, and as such, each node is usually responsible for a component of the robot, being the overall system comprised of many nodes. As an example, one node can perform tasks associated with localization, as other node performs path planning operations.

The use of nodes in ROS has advantages to the overall system functionality, as there is additional error tolerance, since crashes are isolated to individual nodes. Nodes also constitute an abstraction to other layers of implementation, resulting in a code complexity reduction.

Nodes communicate with each other by publishing messages to topics. A **Message** is a strictly typed data structure. Standard primitive such as integer, floating point, and boolean, are supported, as are arrays of primitive types and constants. Messages can include arbitrarily nested structures and arrays.

A node sends messages to another nodes by publishing them to a given **Topic**. First the topic is initialized as an advertisement via a topic name, which is simply a string such as *"camera"* or *"octomap"*. A topic is an identification of a message within the runtime structure of ROS [44]. In other words, topics are named buses over which nodes exchange messages.

In general, nodes are not aware of who they are communicating with, as topics are anonymously published and subscribed. The mechanisms allow for multiple concurrent

publishers and subscribers to a topic, as such, nodes interested in a specific data subscribe to the relevant topic, while nodes that generate data publish to the relevant topic. Analogously, a single node may publish and subscribe to multiple topics.

Although the topic-based publish-subscribe model is a very flexible communication paradigm, its many-to-many one-way transport is not appropriate synchronous transactions that require request and reply interactions, which are often an essential component of a distributed system. The request and reply mechanism is made possible via a **Service**, which is defined by a string name and a pair of strictly typed messages, one for the request and one for the reply.

Analogous to web services, which are defined by a unique URIs and have request and response documents of well-defined types, a providing ROS node offers a service under a unique string name, and a client calls the service by sending the request message and awaiting the reply.

Besides being able to support software reuse and distributed computation, ROS also makes algorithm testing and development easier and quicker. ROS well-designed systems separate the low-level hardware control and high-level algorithm processing and decision making into different programs, making it possible to simulate the low-level hardware in order to test the high-level part of the system.

Since physical robots may not always be available to work with, ROS provides a simple way to record and play back sensor data and other kinds of messages. As the publish and subscribe message system is anonymous and asynchronous, as previously stated, the data can be easily captured and replayed without any changes to the code. In ROS, the recording of data are stored in *"bags"* and there is a tool called *rosbag* used to record and replay the robot's sensor data.

Visualization is an important part of debugging and development, since it allows a graphical representation of the datasets being worked by the ROS framework. The most well-known tool in ROS for visualization is RVIZ. **RVIZ** provides general purpose, three-dimensional visualization of several sensor data types and any robot described using robot description files, as well as the capacity of creating custom shapes like cubes, spheres and points. RVIZ is integrated into ROS and it can visualize the most common message types provided in ROS, such as laser scans, three-dimensional point clouds and camera images. RVIZ also uses information from the *tf* library, to show all the sensor data in a common coordinate frame, together with a three-dimensional rendering of the robot.

Besides RVIZ, **Gazebo** is also a tool within the ROS universe to provide graphical representation of robots, focusing on its simulation. Gazebo is an open-source robot simulator that makes it possible to rapidly test algorithms, design robots and perform regression testing using realistic and complex indoor and outdoor scenarios Gazebo is maintained by the Open Source Robotics Foundation and only available for Linux. [45] Using Gazebo, it is possible to define the characteristics of not only the robot but also the environment. It is also possible to interact with that robot, via ROS, in the same way that the interaction occurs in real life [41].

As previously mentioned, in order to represent or simulate robots in RVIZ or Gazebo, there are several description formats being the most used and important within this dissertation project the Unified Robot Description Format (URDF) and the Simulation Description Format (SDF).

The **Unified Robot Description Format (URDF)** is an XML format for representing a robot model [46]. URDF is a useful and standardized format in ROS, specifying the kinematic and dynamic properties of a single robot [47]. The URDF of a robot can be plugged into ROS as a global parameter, being then possible to access the description of the robot by any node. URDF is characterized for having a standard description of the robot, which is used both for runtime execution, simulation and visualization, which constitutes the main advantage of URDF.

As URDF can only represent the properties of a single robot in isolation, not being able to specify the pose of the robot itself within a world. URDF cannot also specify joint loops, lacks friction and other properties, and it cannot specify objects that are not robots, such as obstacles and light sources [47].

In order to solve the shortcoming of URDF within the needs of a simulation engine, the **Simulator Description Format (SDF)** was created for use in Gazebo. SDF is an XML format that describes objects and environments for robot simulators, visualization, and control [48]. SDF is scalable and makes it easy to add and modify element such as robots, static and dynamic objects, lighting, and terrain.

3.6.1 - Coordinate Systems and the tf library

Robots are designed to operate in the physical world, interacting with objects and other robots. As such, it is natural and necessary to use coordinates to describe the positions of various parts of the robot along with objects the robot should avoid or interact with [41].

In order to properly work in a real environment, robotics systems need to track several spatial relationships, such as the relationship between a mobile robot and a fixed frame of reference for localization, the relationship between the various sensor frames and manipulator frames, and the relationship between the robot and placed frames on target objects, for control purposes [42].

The spatial relationships previously mentioned are tracked by keeping a careful record of the coordinate frames in which the coordinates on both sides of the spatial relationship are described. The coordinate frame in which a message is expressed is usually identified by a message type field named *frame_id* [41].

Since sensors and actuators within a robot can have different coordinate frames, data may be captured with respect to a certain coordinate frame, but an analysis on a different coordinate frame can be more desired [44]. To analyze data coming from different coordinate frames, it is essential to know how they are related to one another. The relationship is obtained using geometric transformations, which can convert coordinates from one frame to another.

The transformation matrix is able to transform a point or vector from one coordinate system to another. It is common practice to assemble geometric transformations to generate global transformations. As an example [49], in order to compute the transforms between two frames, a and c, with b in between, knowing the transformations that connect a with b, T_b^a, and b with c, T_c^b, it is possible to calculate the global transform from coordinate frame a to c with the following equation:

$$T_c^a = T_b^a \times T_c^b \tag{3.5}$$

As a robotic system gets more complex, it is essential for the developer to be able to focus on developing a precise task within the reference frame of its preference. As such, ROS supply a standard library called ***tf***, designed to provide a standard way to keep track of coordinate frames and transform data within the entire system [49]. By using the *tf* library within a robotic project, the developer can be sure that the data is in the coordinate frame that they want, without requiring it to have knowledge of all the coordinate frames in the system.

From an implementation point of view, transforms and coordinates frames can be expressed as a graph with the transforms as edges and the coordinate frames as nodes. The task of computing the net transform is very simple in this representation, as it is simply the product of the edges connecting any two nodes. The graph can have one or more disconnected subgraphs, and the transforms between nodes can be computed within connected subgraphs, but not between disconnected subgraphs. Two nodes may have multiple paths between them, resulting in an ambiguous query, where two or more potential net transforms are possible. To solve this potential problem, the graph must be acyclic [49].

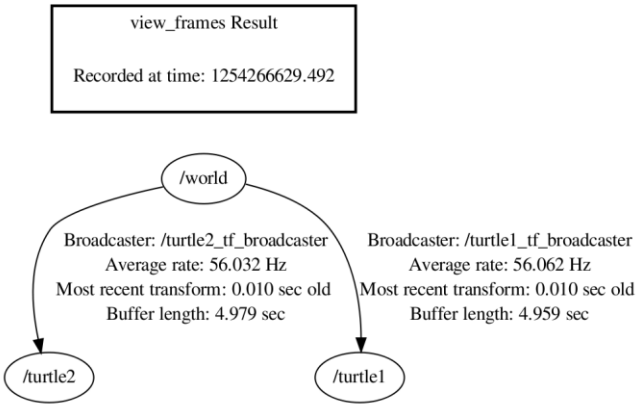


Figure 18 - Example tf tree between two simple virtual robots in one of the ROS tutorials [49].

In a dynamic environment where it is expected that objects such as robots with moving parts or obstacles change over time, another problem arises when the transformations that makes the correspondence between different coordinate frames changes over time. The *tf* library provides mechanisms for seamlessly solving potential time-influenced incoherencies. In order to successfully compute transformations where changes over time occur, all data which is going to be transformed by the *tf* library must contain two pieces of information, referred to as a Stamp. A **Stamp** is composed by the coordinate frame in which it is represented and the time at which it is valid [49].

Chapter 4

Automatic Inconsistency Detection

This aim of this chapter is to describe the work done in the context of this dissertation project. The chapter will be divided into three subchapters, as a way of representing the different areas of development.

The first subchapter addresses the changes proposed to the system architecture of the STAMINA project. It should be noted that some of the assumptions made are based on the current agreement between the different partners of the STAMINA project, expressed on several project documents. The information about the current state of the project was always taken into account while incorporating the changes proposed as the result of this dissertation project. The proposed alterations should be capable to be seamlessly incorporated into the STAMINA project framework.

The second subchapter analyzes the work done on generating a preliminary prototype software for spatial changes detection based on point cloud comparison made possible by the octree structure. The prototype developed was important not only because it allowed a familiarity with the PCL library and the octree structure, but also because it was a starting point for the final solution software, as some of the algorithms used were an inspiration for the final software-based solution.

Finally, the third chapter analyses the final solution, responsible for detecting spatial inconsistencies, graphically representing them, and reporting them to the logistic level. It should be noted that the final solution derives from the work done on previous stages of the project.

4.1 – World Model Dynamics and System Architecture

In chapter 2, the actual implementation of the World Model was presented as a static structure containing the objects considered important to be present on the model of the physical environment. The World Model, as a static structure, stores information about the main components of the kitting supermarket, such as racks, pallets, boxes and parts. On this implementation, the World Model was used as a navigation map, describing the three-dimensional location and orientation of the objects that compose the kitting supermarket. Chapter 3 analyzed the concept of scene graph behind the definition of World Model, within the scope of the STAMINA project.

Based on the previous analysis, it was already mentioned that the final solution for the World Model should be dynamic, in the sense that it should allow updates from the STAMINA robotic fleet. This subchapter aims at explaining the need for a dynamic representation of the World Model. The need for a dynamic representation will be justified by presenting several potential inconsistency scenarios that were identified for the project's use case.

It was previously stated that the World Model will perform as a database for queries from the robotic fleet about the location of parts, racks, boxes, pallets, objects and other robots. Specifically, it is expected that the robotic fleet will consult the World Model in order to retrieve information to construct the navigation path to move to the appropriate location, in order to perform the kitting operation. As such, the World Model will be an auxiliary tool to the localization and navigation of the STAMINA robot. It is also expected that the World Model will be the key element linking the STAMINA robot to the Logistic Planner.

The robotic fleet will also be capable of updating the information stored on the World Model, with the objective of making it a realistic representation of reality. A constant update, made possible by the robotic fleet camera system, will ensure that the World Model will always be a close representation of the state of the world at a certain instant in time. In short, the World Model will be an input for the robotic fleet, on the scope of mission planning, and its state will be changed during execution.

The first World Model instance, developed for the STAMINA project, is not dynamic, *i.e.*, it does not allow updates by the robot fleet or by the human operators. This first World Model implementation, proposed to be assessed during the first test sprint, is presented next.

The preliminary organization of the World Model proposed for the use case is presented on the next figure. As previously stated, each node represents a model of a real object as a two dimensional object. By the process of extrusion, occurs the transformation to a 3D object.

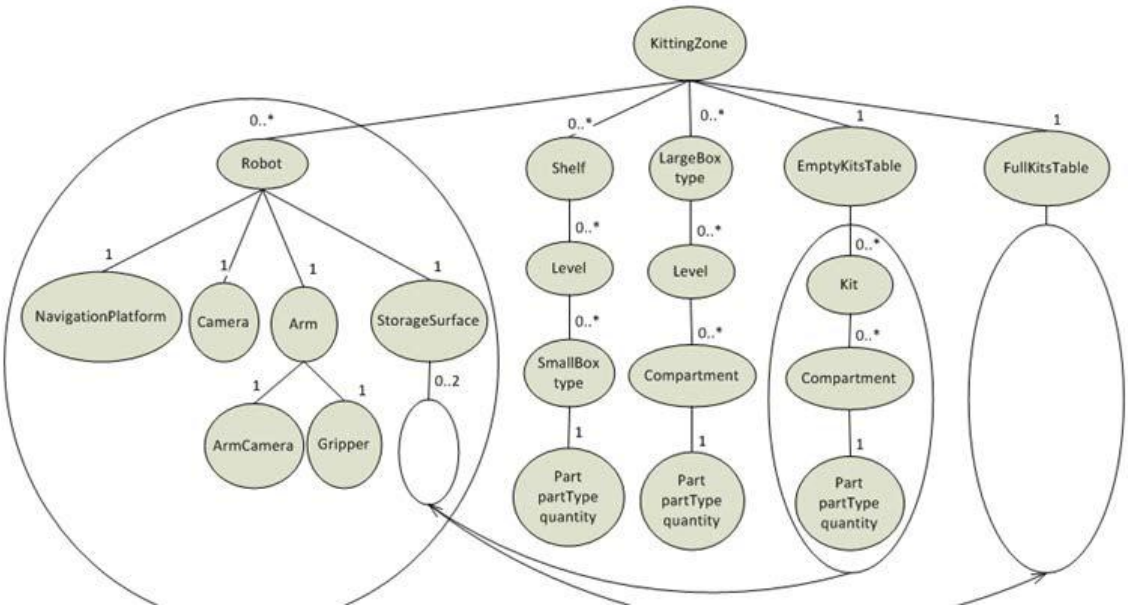


Figure 19 - Preliminary World Model.

The following characteristics are shared between every node of the graph:

Table 3 - Parameters description of the Preliminary World Model.

Parameter	Description
Parent	The parent node (root node has no parent node)
Children	The set of children nodes
Name	Identifies the type of physical object represented by the node
id	Unique identifier of the physical object
Origin.rpy	The orientation of the local coordinate system (roll, pitch and yaw angles in radians) in relation to the coordinate system of the parent node
Origin.xyz	The position of the local coordinate system in relation to the coordinate system of the parent node
boundingVolume	The 3D volume occupied by the physical object, represented as a 2D closed polilyne extruded by height units along the z-axis and centered at the origin of the local coordinate system

As it is necessary to incorporate dynamic and consistency check into the World Model, the final implementation will have to be able to be actualized by the robotic fleet and by the human operators, in order to assure that the information stored on the World Model is as updated as possible.

The work done on this dissertation project is expected to be used to generate signals to notify the worker responsible for the logistic supermarket on incoherencies between reality and the World Model. The notification will allow a verification of the anomaly by the responsible for the logistic supermarket or by his crew.

It is also expected that the information gathered from the inconsistency mechanism will be used for automatic updates of the world model, based on the inconsistency detection between the physical reality, sensed by the STAMINA robotic fleet sensorial system, and the information stored on the World Model structure.

The World Model is a hierarchical data structure, where the different objects that constitute the universe are stored in non-relational database. The World Model, as a dynamic structure, will suffer continuous updates. As such, it is essential that the robotic fleet receives the complete current state of the World Model from the Logistic Planner on the start of each mission.

In order to make a dynamic system possible, the robotic fleet should have the capability of updating the World Model using the data extracted from its camera system. Contrasting the initial situation, where the robot receives the complete current state of the World Model, during the update stage, there is no need to send all the World Model information back to the Logistic Planner as a robot will only be able to update a small fraction of the World Model at a given time, corresponding to its surroundings. As a result, it is important to define that only a small amount of information will be exchanged between the robot and the Logistic Stage, during the update stage. Analogously, if the World Model is updated by an external entity as another robot from the fleet, only a small information package is propagated to the robot by the Logistic Planner, corresponding to the updated region.

The information sent by the robot is the point cloud sensed by the robot camera system, as well as the location and position of the robot's camera on its own referential. The information sent by the robot is then processed at the level of the Logistic Planner, resulting on the detection of inconsistencies between the reality and the World Model. Based on the comparison made, the system can signalize incoherent situations.

The incoherencies detected are classified based on their volume, position and time. The spatial change detection software is capable of detecting the volume and the geometric center of the incoherency, making it possible not only to detect it in space, but also to estimate its volume.

It is important to define the situations where incoherencies between reality and the World Model are possible, in order to identify potential problematic cases. On the next section, the situations where incoherencies are possible to detect will be further analyzed.

- **Pallet out of position.** The many parts variants that can compose a kitting box are stored on containers that vary in size and shape. The big parts are stored in large boxes (named pallets on the project documentation), while the small parts are contained in small boxes, stored on racks. The pallets, as previously stated, are stored directly on the logistic supermarket floor, on a designated area. Due to not having a well defined position for storage, the position of the pallets on the floor can vary not only on location within some centimeters, and within orientation. The STAMINA robot should be able to locate the pallets on the logistic supermarket and report situations where the boxes are outside the designated area.

  Due to the lack of a defined space for the storage of the pallets, they can be stored on different positions and locations, including outside of the designated area, by mistake of the human operator responsible for reposition. As such, the system should be able to continuously update the World Model with the current position of the pallets located on the kitting supermarket. On the event of a box being outside of the designated area, the system should be able to generate a notification in order to signalize the need of repositioning.

  Another possible scenario is the absence of the pallet, *i.e.*, the listing of a pallet on the World Model that does not exist on the real world. The system should be able to identify this situation based on the sensor system of the STAMINA robot, and generate the appropriate notification to the human operators.

- **Rack out of position.** The small boxes, containing the small and lighter parts are stored on racks. Contrary from pallets, racks are stored on a fixed and known position on the kitting supermarket and, as such, their positions are known at all times. The task of locating the racks is based on checking if they are on the position stated on the World Model, within a tolerance. An incoherent location or orientation of the rack should be notified to the human operators.

There are several situations which can lead to a small movement of the rack, as is example the case where the rack is accidentally nudged by a human operator or by the robotic fleet.

It is also possible that the rack is listed on the World Model but does not exist in reality, similarly to what can happen with pallets, described above. Due to its nature, pallets are replaced frequently, contrary to racks, which are expected to remain fixed during the mission execution. As such, the situation where a rack does not exist on reality, but exists on the World Model should be rarer than the pallets case.

The reflective materials that compose the racks, as well as its small volume can present a challenge on its detection using point clouds.

- **Box out of position.** Racks currently used at the PSA Peugeot Citroen's factory are generally composed by two layers of small boxes, stored inclined to allow full boxes to slide to the exterior position by gravity. As such, it is possible that the final solution will have incoherence problems based on the dislocation of boxes generated by the gravity slide. On the current state of the World Model, racks are modeled as horizontal storage layers, so this incoherence should not occur.

  Similarly to the situations previously analyzed, it is possible that the box is present on the World Model but it is not present on reality. The contrary situation, of a box being present on reality but not on the World Model is also possible, and should be addressed by the mechanism developed. There is also the case where a box exists both on reality and on World Model but is slightly outside of position. All these situations should be very frequent on the normal execution of the mission, and, as such, were chosen to be the main focus of this dissertation project. The incoherencies should generate an alarm to notify the human operator responsible for the logistic supermarket to take appropriate measures.

  The STAMINA robot is equipped with a sensor system which allows it to detect the model of the box it is analyzing. As such, the system is able to compare this information with the one stored on the World Model, generating a notification on the event of an inconsistency.

- **Robot detection.** The final STAMINA system will be composed by several robots navigating and working at the logistic supermarket. As such, it important that the robots are aware of the location of other robots of the fleet, in order to plan and optimize the

navigation and avoid contact. Since there are no planed interactions between robots, the detection of another robot should be seen as the detection of an obstacle that the robot must transpose.

Since the information about a robot location is stored on the World Model structure, it is possible do extrapolate if an obstacle detected on a certain position is, in fact, a robot of the fleet. The detection of a robot by the camera system of another robot could also be used to evaluate if the other robot odometry is successfully working.

- **Human detection.** Human operators will also work at the logistic supermarket alongside the robotic fleet. As such, it is important that the mechanism developed should be able to properly differentiate between the detection of a rack, a pallet, a robot, a human or an obstacle, in order to generate the appropriate alarms when needed.

  Due to the frequent movement of a human operator, the inclusion of the information about its location on the World Model is not justified, since the rate of movement of the human is faster than the rate of update of the World Model. This situation could lead to an always outdated World Model database.

- **Obstacle detection.** The distinction between an obstacle and a robot or a human is made based on the static nature of an obstacle, *i.e.*, contrary to a robot or a human, which move frequently on the logistic supermarket, an obstacle is usually at the same position over a long period of time, until being removed by a human operator. An obstacle can easily disturb the normal operation of the logistic supermarket so the generation of a notification is important, as to inform the workers on the necessity of removing it.

- **Wrong part or part out of position.** The STAMINA robot have skills that allow it to verify if a part is stored at a wrong container of if they are stored on a position different from what it is stated at the World Model. There should also be the case where the robot is expecting to find a part that does not exist in reality. Within the scope of this dissertation work, it should be possible to detect if a box is empty or occupied by components, but it is outside the scope of this dissertation project to verify if the part is of a wrong type.

- **Number of components on a container device.** Besides being able to identify the kind of part it is inspecting, the STAMINA robot can also detect the number of components stored on a container. The information of the number of components stored on a container can be useful to update the World Model, in order to allow an optimization of navigation of

the future missions. As previously stated, it is expected that this dissertation project can verify if a box is empty or occupied by components, but it is the responsibility of other partners of the project to address the detection of the number of components stored on a container.

According to the incoherence scenarios previously analyzed, the main focus of this dissertation project is to create the mechanism which analyses the point clouds obtained from the robot's camera system, in order to make possible to compare the information extracted from a point cloud with the information stored at the World Model database. The mechanism created should be able to handle all the scenarios previously discussed, although it is given special attention to the boxes, racks, components, human and obstacle detection, due to the possibility of testing them on a laboratory scenario.

The proposed system plans to add an extra node on the Robot Level, *i.e.*, working integrated with the STAMINA robot. As to perform its duty, the Spatial Change Detection node needs to receive the definition of the regions signalized as free or occupied, either automatically defined or previously defined by the user or during operational runtime. As it will be future analyzed on this document, the proposed node is also prepared to work comparing two octree representations, one being generated during runtime and other previously generated by a prior passage of the robot by the scene, representing the model scenario. This information is expected to be transferred to the robot by the Logistic Planner.

On the following figure is presented the proposed changes to the STAMINA abstract architecture, integrating the Spatial Change Detection node developed by this dissertation project.

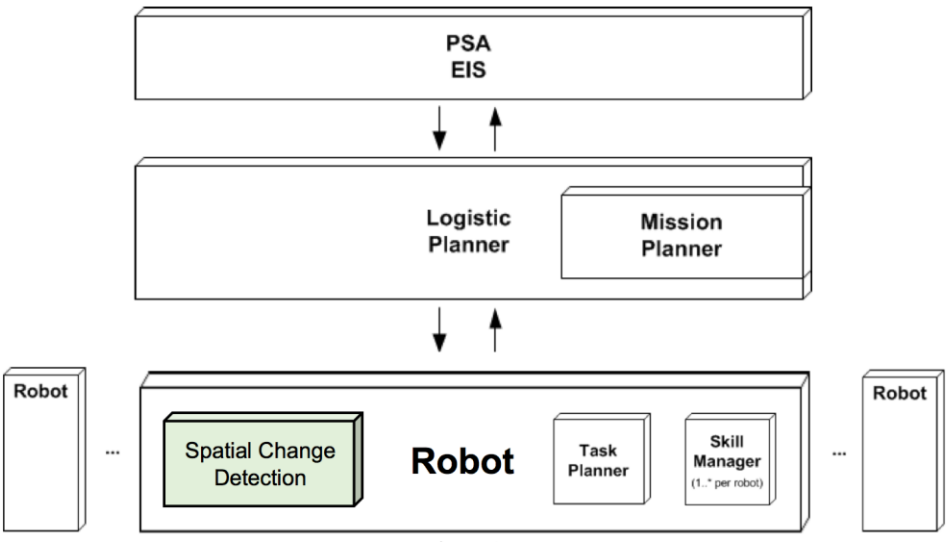


Figure 20 - Proposed Changes to the STAMINA Abstract Architecture.

4.2 – Prototype Software: Spatial Change Detection based on Point Cloud comparison using Octrees.

This second subchapter will address the work done in order to develop a preliminary prototype software for spatial changes detection based on point cloud comparison, made possible by using the octree structure to analyze sets of point clouds.

As previously analyzed in chapter 3, a point cloud is a data structure used to represent three-dimensional data [17]. A point cloud is a collection of 3D points in a three-dimensional coordinate system which represents the surrounding world. Point clouds can be acquired from many kinds of hardware, such as stereo cameras, 3D scanners or Time-of-Flight cameras [17].

In this use case, the point clouds were acquired by an RGB-D sensor, were *D* stands for depth. As previously stated, those kinds of sensors used to cost tens of thousands of euros each, but, nowadays, it is possible to buy such technology for a few hundred euros. The depth data is provided by an infrared sensor and then correlated to a calibrated RGB camera, which yields an RGB image with a depth associated with each pixel [50]. As previously discussed, a point cloud is a collection of points in three dimensional space, where each point can have additional features associated with it, such as the color of each point, as is the case of RGB-D sensors.

The most popular consumer RGB-D sensor is the **Microsoft Kinect**, an Xbox 360 game console accessory. Kinect was released in November 2010, based on a sensor design developed by PrimeSense Ltd. The main characteristics of the original Microsoft Kinect sensor are exposed on the next table.

Table 4 - Microsoft Kinect technical specifications [51].

Field of View	43º Vertical by 57º Horizontal
Frame Rate (depth and color stream)	30 frames per second (fps)
Resolution for depth stream	VGA (640x480)
Resolution for color stream	VGA (640x480)

ASUS produces a RGB-D sensor that is a cheaper and better alternative to the Microsoft Kinect, named **ASUS Xtion PRO LIVE**. This was the sensor used during this dissertation project, mostly because it was the one available at the laboratory while the development was taking place. However, if the experiments were to be replicated using a similar RGB-D sensor, like the Microsoft Kinect, the results would be very similar and the change of sensor would be seamlessly made, from a coding perspective. The technical specifications of the ASUS alternative sensor are expressed on the next table.

Table 5 - ASUS Xtion PRO LIVE technical specifications [52].

Field of View	58° Horizontal, 45° Vertical, 70° Diagonal
Frame Rate (depth and color stream)	30 fps for VGA (640x480) or 60 fps for QVGA (320x240)
Resolution for depth stream	VGA (640x480) (with 30 fps) or QVGA (320x240) (with 60 fps)
Resolution for color stream	SXGA (1280x1024)

There are several libraries that enable the access of the data from RGB-D sensors such as the ASUS Axion Pro Live, as are examples the OpenKinect, OpenNI and CL NUI [53]. OpenNI was selected over the other technologies because it is the framework better supported and documented by ROS and PCL. OpenNI is an open-source framework capable of reading sensor data not only from Kinect, but also from other RGB-D sensors. It is designed and maintained by its member companies to be an industry standard, which implies a sufficient degree of stability and maturity [53].

As previously discussed in chapter 3, point cloud datasets acquired by RGB-D sensors are composed by a very large number of points, which are not convenient to process, as previously explained. As such, in order to perform tasks such as spatial change detection, it is essential to use spatial decomposition techniques, in order to reduce the amount of points and therefore, the overall complexity of the method.

An octree is a tree-based data structure for organizing sparse three-dimensional data, constituting a spatial decomposition technique. The goal of the prototype software was to implement an algorithm that could detect spatial changes between two unorganized point clouds, which could vary in size, resolution and point ordering.

The two input point clouds for this software are the **model point cloud**, which represents the environment in its initial conditions, and the **target point cloud**, representing changes made in the model environment, such as adding a large box or changing the position of an object.

The implemented algorithm recursively compares the tree structures of octrees, identifying spatial changes represented by differences in voxels between the model and the target octree. The code responsible for the spatial change detection was developed using manly the PCL library and it is inspired by the **Spatial change detection on unorganized point cloud data** tutorial on the PCL documentation [54].

As previously mentioned, the algorithm receives a **model point cloud** and a **target point cloud**, stored on a PCD file. On this prototype solution, for simplification purposes, it was assumed that both point clouds were acquired at the same reference frame, thus being exactly the same if there were no changes to the environment. The **model point cloud** served as the reference point cloud, being the octree structure a description of its spatial distribution.

In order to compare the two octrees, it is first necessary to instantiate the *OctreePointCloudChangeDetector* class and define its voxel resolution for the lowest octree level. The class *OctreePointCloudChangeDetector* inherits from class *Octree2BufBase* which enables to keep and manage two octrees in the memory at the same time. It also implements a memory pool that reuses already allocated node objects and therefore reduces expensive memory allocation and deallocation operations when generating octrees of multiple point clouds [54]. Those properties would be important if the comparison would have been made using target point clouds acquired in real time, demanding a fast response from the algorithm.

To get the points that are stored at voxels of the target octree structure which are non existent in the model octree structure, the method *getPointIndicesFromNewVoxels* is called, returning a vector of the result point indices. In other words, the method retrieves a vector with the indices of all points that are new in the target octree when compared with the ones that exist in the model octree.

The *getPointIndicesFromNewVoxels* is a method of the PCL octree class *OctreePointCloudChangeDetector*, responsible for detecting new leaf nodes when comparing two octrees, and serializing their point indices. The mentioned method gets the indices from all leaf nodes that did not exist in the previous buffer [55].

Using the vector with the indices of all new points returned from the previously described method, it is possible to construct a new point cloud, composed by just those points, that represents the points where changes occur.

As the points that are different between the target point cloud and the model point cloud can be results of measurement errors and imprecisions, the resulting difference point cloud might have a large number of shadow points or outliers. As such, it is advisable to develop a mechanism to filter unwanted points, keeping only the points that correspond to real inconsistencies.

In order to remove outliers, a Euclidean cluster extraction preceded by a plane model segmentation algorithm was used.

The plane model segmentation consists in finding all the points within a point cloud that fits a simplified geometric model. The complexity of the expected structure that is to be searched for and the noise levels present in the data influence the expected efficiency of the search methods for searching a certain structure or pattern in a point cloud dataset [18].

One of the most popular techniques of data segmentation consists in relating points with three-dimensional geometric primitives. A geometric primitive is a basic surface model that is easy to compute, and can be approximate the measured dataset within some error bounds. In the case of this application, since the main aim is to detect cubic boxes, the used geometric primitive was the plane [18].

As previously stated, the mechanism for relating the point cloud dataset with geometric models can get more complex as the complexity of the surface searched for and the noise levels present in the data set increases. In order to avoid long computational times, the segmentation mechanisms usually employ heuristic hypotheses generators which provide upper bounds on the number of iterations that need to be run to achieve a certain required probability of success.

One of the most popular robust estimators is the **Random Sample Consensus (RANSAC)** method. RANSAC is capable of interpreting and smoothing data containing a large noise level, fitting successfully the measured data to a geometric model. The iterative method aims to solve the **Location Determination Problem (LDP)**, where the goal is to determine the points in the space that project a dataset into a set of landmarks with known locations [56]. The simplicity of the RANSAC method was the key factor for using it in this preliminary software. In fact, there are several other robust estimators that use RANSAC as a base, adding additional and more complicated features.

That are methods within the PCL library that allow a simple implementation of the RANSAC method [57]. Besides defining the method type as RANSAC, the geometric model to what the dataset is approximated was set to a plane. RANSAC is a non-deterministic algorithm as it produces a result with a certain probability, that increases as more iterations are allowed. For this project, the maximum number of iterations was set to 100 as it was a reasonable number of iterations to reach the desired outcome without compromising the duration of calculation. The distance threshold for the comparison was set for 0.01 meters, as it was the best factor obtained from analysis of the results.

After running the plane model segmentation, it was verified that several outlier points were being flagged as part of a plane, resulting in a large number of planes, many of which did not corresponded to an inconsistency that the software tried to detect. As a solution to solve this problem, it was decided to take into account only the planes that were composed by at least 30% of the difference point cloud data set. The threshold value was decided after several testing procedures. If none of the candidate planes segmented on the previous step were able to verify this condition, the two point clouds were considered equal and the process ended.

The next step in order to obtain the point cloud representing only the points where inconsistencies occurred, based on the point cloud that resulted of the octree comparison, already filtered by the planar model segmentation was to extract the subset of points based on the output given by the segmentation method. In order to process multiple planar models, the indices extraction process runs in a loop, extracting only the points of the difference point cloud that were identified as being part of a plane, removing the others.

After this step, in order to continue the process with a Euclidean Cluster Extraction, it was verified that the point cloud had several points that were defined as NaN. As such, a method defined within the pcl_filters library for removing the NaN values from the point cloud was used [58], as the non defined values created a problem on the following procedures.

A cubic box constitutes the kind of object that this software aims to detect. As a box is composed by several planes (although only a limited number of planes is visible at a single time). Therefore, it is essential to group the planes that composes a box, in order to be possible its segmentation. The method proposed next is able to detect the presence of multiple boxes in a scene, by grouping the planar surfaces into a group of points that constitutes the box and then segmenting it.

In order to group the planes, it is essential to determine the neighborhood points of points that compose a specific plane, in order to check if there is another plane present in its neighborhood.

Using a brute-force process, computing the neighborhood means that in order to determine the k closest points of a query point $p \in P$, all distances from p to all the points in P must be estimated and ordered, with the first smallest k ones corresponding to the set of point P^k. This solution, however, is extremely slow, making little sense to use it for the application being developed [18].

In order to achieve that goal, the Euclidean Cluster Extraction was used. As previously mentioned, a point cloud is not capable of storing neighborhood information, and, as such, it is a good practice to make use of a three-dimensional subdivision of the space using fixed width boxes.

The Euclidean Cluster Extraction algorithm is based on the property of determining all the k neighbors of the query point up to a radius r from it, grouping them in a cluster. Supposing there is a set of points $p_i \in P$ and another set of points $p_j \in P$, if the minimum distance between them is larger than an imposed distance threshold value d_{th}, then the set of points p_i are set to belong to a point cluster O_i and the set of points p_j is set to belong to a different point cluster Q_j. Mathematically, this definition can be defined as follows [18]:

Let $O_i = \{p_i \in P\}$ be a distinct point cluster from $O_j = \{p_j \in P\}$ if:

$$\min \left\| p_i - p_j \right\|_2 \geq d_{th} \tag{4.1}$$

From an implementation point of view, the imposed distance threshold value was set to be 0.1 meters on this software. Any set of points that are separated by more than 0.1 meters should be considered different clusters. Those values are the result of several testing procedures conducted, in order to reach the best possible configuration for the tested scenario.

The PCL library has a class [59] capable of implementing the cluster extraction in a Euclidean sense. The algorithm implemented by the PCL library was proposed by [18] and is very similar to a flood fill algorithm and its based on the previous described definition of a cluster. The algorithmic steps for its implementation would be:

1. Create a kd-tree representation for the input point cloud dataset P;
2. Set up an empty list of clusters C, and a queue of the points that need to be checked Q;
3. For every point $p_i \in P$, perform the following iterative steps:
   3.1. Add the point p_i to the current queue of points that need to be checked Q.
   3.2. For every point p_i on the current queue of points that need to be checked Q:
      3.2.1. Search for the set P_i^k of points neighbors of p_i in a sphere with radius $r < d_{th}$, where d_{th} represents the maximum imposed distance threshold;
      3.2.2. For every neighbor $p_i^k \in P_i^k$, check if the point has already been processed, and if not, add it to Q;
   3.3. When the list of all points in Q has been processed, add Q to the list of clusters C, and reset Q to an empty list;
4. The algorithm terminates when all points $p_i \in P$ have been processed and are now part of the list of point clusters C.

The PCL class for implementing the Euclidean Cluster Extraction defines two additional parameters that allow setting a minimum and a maximum number of points that a cluster must have in order to be considerer as such. It was determined that for the purposes of the test done, the clusters should have between 1000 and 25000 points.

After this process is finished, it is possible to extract several clusters of points, each representing an incoherent object, *i.e.*, a box that was not present on the *model point cloud*, but is present on the *target point cloud*. The software saves the clusters found into several different PCD files, that can be visualized later.

4.3 - ROS based Spatial Change Detection using Octrees

The prototype software previously presented was capable of detecting spatial changes by comparing point clouds using the octree structure to analyze sets of point clouds, representing the model and the target scenario. As it was verified that different points between the target point cloud and the model point cloud could be the results of measurement errors and imprecisions on the acquisition system, it was necessary to develop a mechanism to remove the outliers, allowing the detection of real inconsistencies only.

To remove unwanted points, a plane model segmentation algorithm followed by a Euclidean cluster extraction was used. As previously explained, the plane model segmentation consists in finding all the points within a point cloud that fits a simplified geometric model that resembles a plane. This approach, although suited for the purpose of the preliminary software, was only capable of properly detecting objects that are composed by planes, like boxes.

A more flexible approach was needed, since the paradigm of spatial changes detection should be applied to objects that are not composed by strictly planar surfaces, as is the case of the detection of a human or another robot from the fleet. As such, the approach used on the final implementation removed the filtering by a plane model segmentation and replaced it with another set of techniques that will be further addressed on a later section.

This subchapter is aimed at explaining the final solution proposed by this dissertation project. The final solution is a software capable of spatial changes detection, signalizing incoherencies in real time, by comparing a previously defined model scenario with an octree constructed with the OctoMap framework, based on the point cloud data acquired in real time by a camera installed on the robot.

The comparison software is composed by **two modes** of spatial change detection, that differ mostly on the form of specification of the model scenario. On the **first mode** of operation, the model scenario is defined as an octree built based on point cloud data previously acquired, while on the **second mode**, the model scenario is composed by a set of regions, specified trough a graphical tool by the user, that define which spaces should be free and which spaces should be occupied by objects.

The two modes of operation, from an implementation point of view, share most of their concepts and functions. However, due to some important differences to signalize, it was decided to present the algorithmic processes on different sections on this document. As such, first the comparison between the two octrees is presented, and on the following section, the

comparison using a defined region is used. The analysis of the transformations between different coordinate frames and camera calibration is an important component of the project, and, as such, it will be analyzed on a dedicated section. Before presenting the tests and respective results about the implementation of this stage of the project, this subchapter will address some utilitarian tools developed in order to simplify the execution of all the involved components.

4.3.1 - ROS Integration

The final STAMINA robotic system will be implemented on a ROS environment. Developing the software for spatial detection using the ROS framework has several advantages besides the simple integration with the STAMINA framework. As ROS is composed by a collection of tools, libraries and conventions, the task of creating the complex algorithms needed for successful reaching the objective is simplified.

ROS is an open-source operating system for robots, providing all services expected from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionalities, message-passing between processes and package management [60]. As such, the developed solution can easily be packaged and used on different robotics hardware, being the transition simplified, since ROS deals with the integration and low-level control between different platforms.

Nowadays, most modern robotic systems rely on software that is divided into many different processes, potentially running on more than one computer. ROS ideology states that is good practice to divide the robotic software into small, stand-alone parts which cooperate to achieve the overall goal. In order to cope with this need for segmented robotic software, the components were developed with collaboration and communication tools that enable connectivity with each other and with the several subsets that control the robotic system.

One of the most important features of ROS is the set of characteristics which allow ROS executables, called nodes, to be individually designed and loosely coupled at runtime. Nodes, can be grouped into ROS Packages, enabling them to be easily shared and distributed between partners. This organization supports code sharing and reusing between the several partners of the project.

As previously stated, ROS is composed by a collection of tools, libraries and conventions which simplify the process of developing complex algorithms, thus making it easy to successfully reach the objectives of a certain project. Within this dissertation project, one of the most important tools used was RVIZ. RVIZ is a tool integrated into ROS, used to visualize the most common message types provided by the operating system. RVIZ can easily render point clouds and octrees generated within a specified reference frame and represent them using a graphical interface. One of RVIZ most used feature during this project development was the marker representation, as it allowed graphical representations of the different stages of the algorithmic process. Another important tool used during the development stages of this project was *rqt_reconfigure* [61], which allows parameters to be dynamically altered during runtime. ROS also offers several tools for listening to messages, listing topics being published, among many others. The integration with the PCL and OctoMap libraries is also a key advantage on working with ROS.

Besides offering tools for visualization, dynamically altering parameters during runtime and checking the status of the several components of the ROS universe, the main advantage of using the ROS framework on this project was the ROS *tf* library.

As previously explained, the ROS *tf* library is responsible for providing a standard and seamless way to keep track of coordinate frames and transforms data within the entire system. The *tf* library is capable of handling all the computations needed to transform data between several coordinate frames, having into consideration potential time-influenced incoherencies.

Since the developed software is expected to be applied to a camera installed first on fixed robotic arm, and on a future stage on a mobile robot, the ability to seamlessly handle coordinate frames transformations in order to secure that the data being captured can be successfully processed is a crucial aspect of the project. As such, the usage of ROS *tf* library is a key component on this project and will be further addressed on a later section.

4.3.2 – Spatial Change Detection

The development of the final solution was composed by a set of iterative steps, where functionalities were added between each iteration. It started as an implementation of the algorithms used on the prototype software as a ROS node, evolving to the final solution with **two modes of operation**, that will be further explained. This section will first present the steps taken until a final implementation was reached and then it will explain the algorithmic process of the two modes of operation.

The first step of the iterative development was to create a ROS node which implemented a callback structure that was responsible for receiving point cloud messages being published on a predefined topic. This solution received a point cloud message for the model scenario and the target scenario. As it is not possible to acquire in real-time both a target and a model point cloud with just one camera, one had to be played back from a bag file. In order to implement a primary spatial detection algorithm, point clouds were used to construct an octree through the OctoMap framework that would be updated as the simulation proceeded. At this point, there was no solution for filtering the unwanted points originated from the false-positive detection of differences between the octrees.

The method of receiving two point clouds and converting them into octrees within the code execution was not only slow but also ineffective. As the OctoMap framework needs some time to construct the octree, and its probability of occupation takes into account the amount of time that a measurement occurred, the live computation of both the model and the target octrees was flagged as an incorrect approach, as its response time was not ideal.

On the purpose of real-time mapping of a dynamic environment, the usage of octree data structures as a way of spatially decomposing the environment presents a big advantage, since octrees are easy to update and are able to support dynamic point insertion and deletion. As such, it was decided to decouple the process of generating an octree from the node that computes the spatial changes detection.

On the final solution, the **first mode of operation**, based on the **comparison of a target octree with a model octree** receives the two octrees being published on specific topics. The octree from the model is read from a file, while the octree target can either be generated from a file or in real-time based on the information being captured by the depth camera. By reading from a file the octree of the model scenario previously captured, it is possible not only to speed the processes, as the octree has a considerably smaller size than a sequence of point clouds, but also increase the consistency of the model, since the octree stored on file is

the result of several seconds of updates, obtained trough the OctoMap framework. Using this new approach, the software can be executed in real time, with cycle times of less then half a second, for the set of points tested.

A major drawback was detected on the function used to compute the spatial change detection on the prototype version, related with the detection of inconsistencies of objects that existed on the model but were removed on the real time observation (target scene). Although the set of functions from the Point Cloud Library (PCL) allowed a correct detection of added objects on the target scene in comparison with the model scene, it lacked the proper representation of inconsistencies when it was necessary to detect missing objects. The set of PCL functions were capable of somehow signalizing a spatial change, but they were unable to correctly represent the octree cells that went missing. As such, it was necessary to develop an algorithm to solve this problem, abandoning the PCL library functions.

The algorithmic process for detecting inconsistencies when **comparing two octrees** (first mode of operation) is composed by two sets of iterations, one for detecting the inconsistencies characterized by objects being introduced on the target scene, referred for now on as **exceeding inconsistencies**, and another iteration for detecting objects from the model scene that went missing on the target scene, referred on the remainder of this document as **missing inconsistencies**.

On detecting **inconsistencies**, the algorithmic solution used is composed by a pair of nested iterations. One pair of nested iterations is responsible for finding the **exceeding inconsistencies** and the other pair is responsible for finding the **missing inconsistencies**.

The first step is to define a searchable area, corresponding to the observable part of the rack where the automatic search mechanism should look for inconsistencies. This prior step allows a faster and more responsive software, since the search is only done on the area previous defined by the user where inconsistencies are expected to happen.

In order to easily define the region, a graphical interface tool was created which allows the user to signalize on the environment where the algorithm should look for inconsistencies. This tool allows an easy selection of a region on the map by manipulating two dynamical markers presented in RVIZ. The dynamical markers appear at the diagonal extremities of the region to be defined and are easily manipulated by the colored arrows shown on the next figure. The tool allows the user to save the region defined on a file, and thus, allowing the comparison algorithm to take place without the need to redefine the region every time that the spatial change detection software is executed.

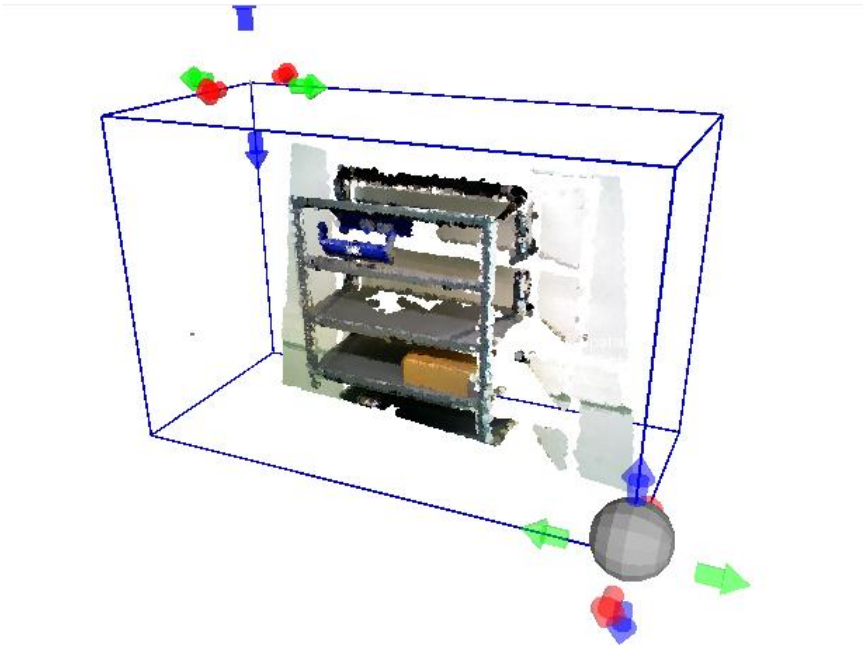


Figure 21 – Searchable Space Definition Tool.

On the following figure it is possible to observe the octree within the region defined as searchable. It should be noted that points outside of this region will not be used to check for inconsistencies.

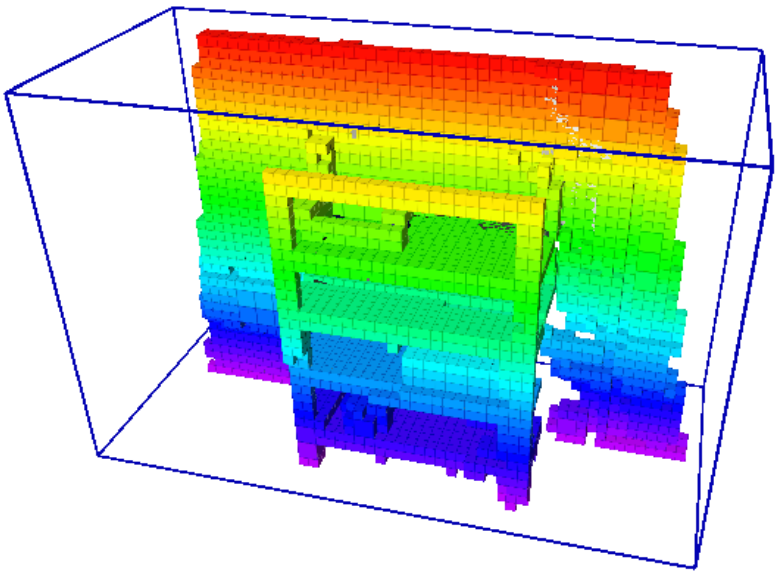


Figure 22 - Octree representation of the space withing the defined searchable region.

It should be noted, however, that this previous step is an optional optimization process, since the algorithm was capable of searching the complete space tested without a significant loss of responsiveness. It is expected, however, that if the comparison would take place using a very large octree the responsiveness of the code could potentially drop, justifying on those situations the use of the tool that defines a zone where the algorithm should check for inconsistencies, ignoring the remainder scenario. This region definition tool is supported by the fact that the use case chosen by PSA Peugeot Citron is composed by a linear kitting supermarket, and, as such, it is possible do define a region where the robotic fleet navigates and where it is important to check for inconsistencies.

The concept that supports the algorithm responsible for searching **exceeding inconsistencies** is the process of checking which cells of the target octree are **new** in comparison with the model octree. The algorithmic steps for finding **exceeding inconsistencies** succeeds as follows:

1. For every cell, defined as a node on the Target octree representation, within the volume flagged as searchable:
    1.1. Verify if the node exists, *i.e.*, if it is occupied or if it is free. If the node exists:
        1.1.1. Define a Bounding Box with the size of the cell corresponding to the node.
        1.1.2. Publish a marker with the dimensions of the defined Bounding Box, allowing a visualization of the cells where the algorithm will check for inconsistencies on RVIZ.
        1.1.3. Initialize counters for the number of neighbors and the number of occupied cells of the Model octree.
2. For every existing cell on the Target octree within the volume defined as searchable, iterate over the Model octree within the volume defined by the Bounding Box generated previously, and:
    2.1. Verify if the corresponding node on the Model octree exists, *i.e.*, if it is occupied of if it is free. If the corresponding node exists:
        2.1.1. Increment the number of neighbors counter.
        2.1.2. Check if the corresponding node on the Model octree is occupied. If it is occupied, increment the number of occupied cells counter.
3. If the previous iteration detected neighbors within the Bounding Box defined by the first iteration, *i.e.*, if the number stored on the neighbor's counter is greater than zero, then:
    3.1. Define the Occupation Ratio as the division between the number of occupied cells and the number of neighbors found.
4. If the Occupation Ratio is less then the defined exceeding threshold, an exceeding inconsistency has been detected, and so:

4.1. Push back the Target Bounding Box where the inconsistency was found to a vector that holds all the exceeding type inconsistencies
4.2. Publish a marker with the dimensions of the inconsistency cell, allowing its visualization on RVIZ.

As it was defined on the algorithm, the process does not iterate over cells which are flagged as an unknown region, being those areas ignored, since a scenario of inconsistency is inconclusive as there is no information about the state of occupation on one or both octree representations. As it is also visible, the algorithm only checks cells on the octree that are within the volume flagged as searchable by the user, according to the proceedings previously discussed.

The Bounding Box is an object of a developed class named *ClassBoundingBox* which defines a region in space. The objects of the class are constructed by passing the three-dimensional size of the space to be modeled. The class has several methods which allow data manipulation and queries on data stored, allowing for simple calculations on its volume, center point and neighborhood search functions.

The need to define an occupation ratio of a certain cell has to do with the fact that the octree representation can represent the leaf nodes with different sizes, depending on the actual occupation of a certain space. Supposing that a large region is space can be grouped as free or occupied, octree representation groups the region volume within a larger cell than it would if the space had an opposing occupation within its volume. As such, the same point in space could be grouped by a smaller or larger cell depending on its surroundings. Because of this reason, it is necessary to implement a parameter that can represent the rate of occupation of a certain cell on an octree when compared to another corresponding one. The occupation ratio is defined as the number of occupied cells divided by the number of neighbor cells found.

The change detection software was developed in such a way that the exceeding occupation ratio threshold and the missing occupation ratio threshold can be dynamically changed during runtime, using the ROS *rqt_reconfigure* tool. As such, for a certain scenario it is possible to alter the parameters and verify how the algorithm works best. In order to get a sense on how the occupation ratio influences the output of the software, several tests were made and the results are displayed on a later section of this subchapter.

In order to catch **missing inconsistencies**, the developed software follows the algorithmic steps previously exposed for the detection of exceeding inconsistencies, with the only difference that it iterates first over the model octree and then it iterates over the target octree, in order to check for cells that existed on the model octree that went missing on the target octree. With this double process of nested iterations, it is possible not only to check which objects were added to the scene, as it was previously possible using the prototype software, but it is also possible to have a reference to the cells that went missing in the scene.

After the double process to check inconsistencies, it was verified that the inconsistent cells had a disperse representation on the scene, being necessary not only to filter them, but also to group the neighboring cells into clusters. The algorithm to clustering is inspired by the previously presented Euclidean Cluster Extraction, and it also gets some influence from Flood Fill algorithms [62]. The clustering algorithmic steps succeeds as follows:

1. Starting with a vector of inconsistent cells, V, set up an empty queue of the cells that need to be checked, Q, and an empty vector of clusters, C.
2. Push back to the queue of cells that need to be checked, Q, every cell on the vector of inconsistencies, V.
3. While there still are cells that need to be checked on the queue Q, perform the following iterative steps:
   3.1. Initialize a seed cell, $s \in Q$, which corresponds to the first element of the queue of cells that need to be checked, and remove it from the queue.
   3.2. Create a new cluster C_i and add the seed cell, s, to it.
   3.3. Perform the expansion of the seed cell, by creating an auxiliary flood vector, F, and adding to it the seed cell, s.
   3.4. Iteratively check if the seed cell s is in the neighborhood of the remainder cells on the queue of cells that need to be checked, Q, and, if a neighbor cell is found:
      3.4.1. Push back the neighboring cell to the flood vector, F, and erase that cell from the queue of cells that need to be checked, Q, since it will be already attributed to a cluster.
   3.5. Pushback the auxiliary flood vector, F, to the cluster created C_i.
4. The algorithm finishes when all inconsistency cells have been attributed to a cluster and, therefore, the queue of cells that need to be checked, Q, is empty.

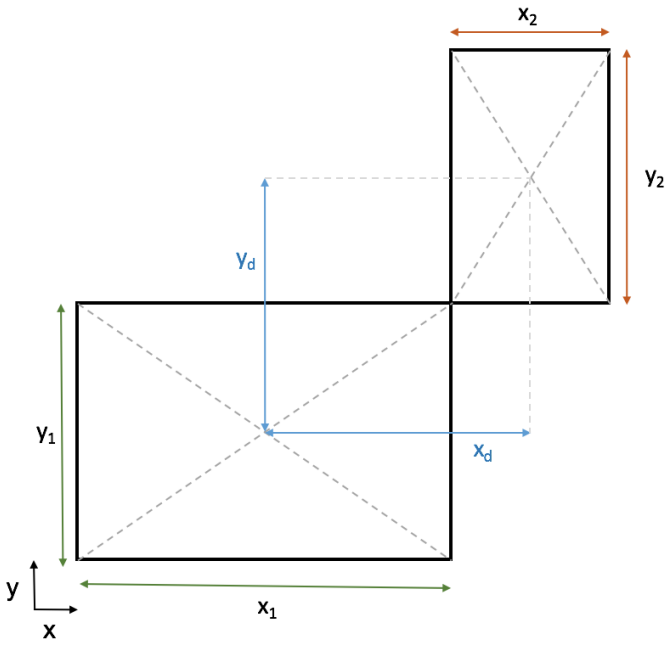


Figure 23 - Neighborhood criteria.

The criteria of considering a cell a neighbor can easily be explained in two dimensions, being the three-dimensional situation easily extrapolated. Defining two 2D cells with dimensions $(x_1,\ y_1)$ and $(x_2,\ y_2)$, and defining as (x_d, y_d) the distance between them, if:

$$x_1 + x_2 \geq x_d \tag{4.2}$$

and

$$y_1 + y_2 \geq y_d \tag{4.3}$$

then the two cells are neighbors. The two-dimensional example is used to explain the tree-dimensional scenario, being the only difference the addition of another dimension.

The cluster grouping allows to filter small clusters which represent measurement errors based on a variable volume threshold. As such, to be considered a valid inconsistency, the corresponding cluster must have a larger volume than the volume threshold specified. The process of computing the cluster volume within the spatial change detection software is very straight forward, as the cells which compose the cluster are bounding boxes defined by the class *classBoundingBox*. The developed class has a method that computes the volume of the box instantiated by multiplying the sizes that constitute the volume defined. The filter function computes the sum of the volume of every cell which composes the cluster and compares it with the volume threshold.

It was found that the ideal volume threshold is specific to each scenario tested, and, as such, it is possible to fine tune the value during runtime, as it was specified as a ROS parameter. Similar to the occupation ratio previously discussed, also the volume threshold was tested on several values in order to better understand its influence on the spatial change detection. The results will also be further analyzed on a following section.

The clustering stage, besides allowing filtering small clusters based on their volume, as previously explained, also allows representing the found inconsistencies as a group of cells, and not as a disperse representation of individual cells. This feature not only translates into a better visualization, but it is also essential for computing the cluster volume and geometric center.

In order to generate a message that will allow updates on the world model based on the inconsistencies detected and reported by the spatial changes detection software using the camera systems installed on the STAMINA robot, it is essential to have some information not only about the type of inconsistency detected (exceeding inconsistency or missing inconsistency), but also about the position of the inconsistency and its volume. The set of attributes for a detected incoherence will be fundamental to achieve a successful execution of the system.

The geometric center of an object is the arithmetic mean position of all the points in all of the coordinate directions. The geometric center of a cuboid is the point of intersection of its diagonals, as can be seen on the following figure.

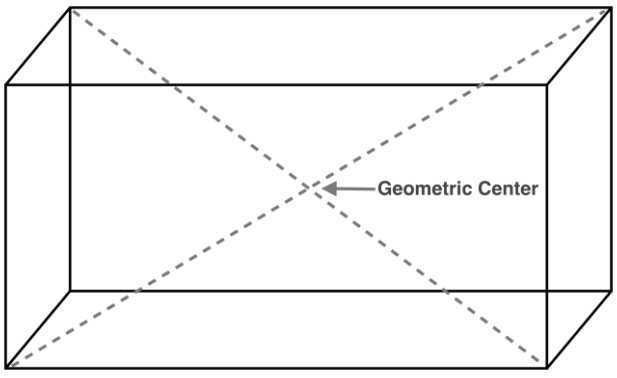


Figure 24 - Geometric Center of a Cuboid.

As such, the **geometric center of a cluster** is defined as the point which corresponds to the average value of the sum of the dimensions of each composing cluster in the three-dimensional coordinate directions.

As previously mentioned, the clusters of inconsistencies detected are filtered by its volume, and, as such, the software has already stored the **volume** of each of each cluster, computed by the process previously described.

The final feature of the spatial change detection software is to publish its results to allow a useful graphical representation of the inconsistencies and their corresponding geometrical center and volume. The ROS tool RVIZ was used during this dissertation project to graphically represent the information outputted by the developed comparison software.

Besides publishing ROS topics which can be visualized by RVIZ, the software also publishes ROS messages with the information on the inconsistencies detected and their attributes. The ROS messages are then received by a standalone ROS node that transforms the received information into JSON documents, in order to allow the World Model update mechanism previously exposed.

4.3.3 – Region Definition

The second mode of operation is characterized by the specification of which space of the scene should be free or occupied, and then using those spaces to check for inconsistencies on the target scenario. Contrasting with the first mode of operation previously presented, the region definition mode does not require an acquisition of point cloud data from the model scenario, as the comparison takes place with the spatial information that the user specifies.

The first development step of this mode of operation was to create the tool to allow the user to easily create regions on the scene where free and occupied cells are expected to be found. This tool was developed as a graphical interface very similar to the Searchable Space Definition Tool previously mentioned. Once again, RVIZ is used to represent dynamical markers which allow the user to create regions in space, as it is shown on the next figure.

The point cloud shown in the figure is the result of an auxiliary tool developed which can combine several point clouds of a bag file into just one large representation of the environment. In order to allow a successful composition of point clouds taken in different regions of the scenario, the *tf* library was used to handle the coordinate frames transformations. As a point cloud from an industrial scenario can ascend to several million points, the developed tool filtered the points, as to compose a point cloud only with the purpose of helping the placement of the boxes that define a region.

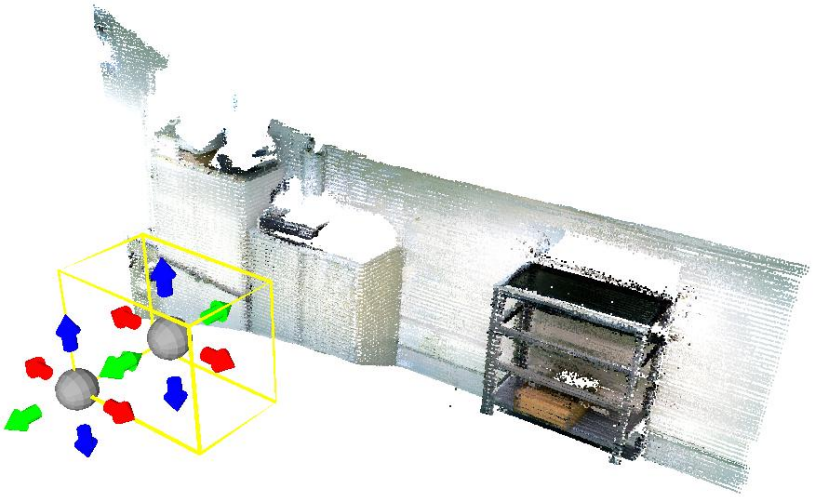


Figure 25 - Region Definition Tool.

As it is observable, the definition of the regions in this tool are slightly different from the previous presented solution. In this case, there are also two markers, with the difference that one is defined at the center of the cuboid while other is defined on the extremity of it. This difference is justified with the fact that in this case is easier to manipulate the cuboid by moving the central marker around the scenario, and then scaling the volume using the extremity marker.

After placing the markers at the appropriate position, the user can select from a dropdown menu if the region defined should be marked as a free or as an occupied space. On the next figure it is possible to see some sectors of the rack defined as free space (green) and the box stored on the rack defined as an occupied space (red).

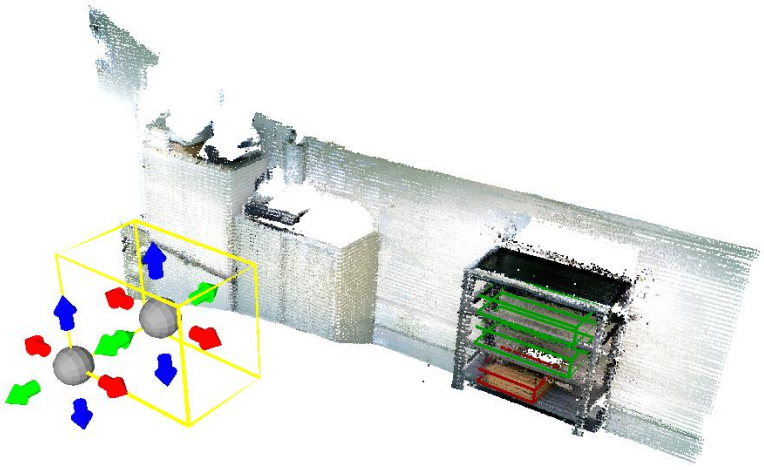


Figure 26 - Region Definition Tool. Green volumes represent free space and red volumes represent occupied space.

The tool creates a set of ROS parameters which can be saved by the user to a file, avoiding the definition a scenario several times. The tool also has an option to load the ROS parameters that define the spatial regions, allowing the user to update the region with new zones.

The second mode of operation of the spatial change detection algorithm developed implements a callback function that successively receives updates about the information concerning the location, dimension and type of regions stored on the ROS parameters. Since the ROS parameters can be dynamically altered during runtime and there is a callback mechanism to check for updates on ROS parameters defining the free and occupied regions, it is also possible to define new regions while the comparison algorithm is executing.

Contrasting the first mode of operation previously discussed, the information about the model scenario is not received by the spatial change detection software as an octree, but rather as a set of parameters that describe instances of the class *classBoundingBox*. It should be noted that the class, besides defining the region in space, also defines an attribute that signalizes if the region in space is defined as occupied of free. This parameter is later accessed by a function defined in the class to check for inconsistencies.

The main difference between the spatial change detection using two octrees or using a region defined model and a target octree, from an implementation point of view, is related with the algorithmic steps to detect exceeding and missing inconsistencies. As such, the process to detect incoherencies between a region defined model and an octree succeeds as follows:

1. For every space defined using the Region Definition Tool:
    1.1. Initialize the counters of the number of neighbors and the number of occupied cells.
2. Iterate over the Target octree within the spaces defined using the Region Definition Tool:
    2.1. Verify if the node on the Target octree exists, *i.e.*, if it is occupied or if it is free. If the node exists:
        2.1.1. Increment the number of neighbors found.
        2.1.2. Check if the node on the target octree is occupied. If it is, increment the number of occupied cells counter.
    2.2. If the previous iteration detected neighbors within the space defined by the user, *i.e.*, if the number stored on the neighbor's counter is greater than zero, then:
        2.2.1. Define the Occupation Ratio as the division between the number of occupied cells and the number of neighbors found.

2.3. If the space was defined by the user as a free space and if there is an Occupation Ratio larger than the exceeding threshold, an **exceeding type** of inconsistency has been detected, and so:

2.3.1. Push back the space where the inconsistency has been detected to a vector that holds all the exceeding type inconsistencies.

2.3.2. Publish a marker with the dimensions of the defined space, allowing its visualization on RVIZ.

2.4. If the space was define by the user as an occupied space and if there is an occupation ratio smaller than the missing threshold, a **missing type** of inconsistency has been detected, and so:

2.4.1. Push back the space where the inconsistency has been detected to a vector that holds all the missing type of inconsistencies.

2.4.2. Publish a marker with the dimensions of the defined space, allowing its visualization on RVIZ.

As previously stated, after constructing the vectors of inconsistencies, the process of clustering and computation of volume and geometric center follow a similar approach to the first mode of operation previously discussed.

Similarly to the first mode of operation, the output expected by this region-defined spatial change detection mode of operation is a set of topics that allow the visual representation of the inconsistencies using RVIZ, as well as a ROS message composed by the information about the location of the geometric center, the volume and the type of inconsistency detected. The message structure and the set of topics was developed to follow the same format as the ones mentioned on the first mode of operation.

4.3.4 – Coordinate Frames and Camera Calibration

The final solution software was developed and tested using a **ASUS Xtion PRO LIVE** camera mounted on the extremity of a **Universal Robot UR5**. The UR5 is a 6-axis robot arm with a working radius of 850 mm, capable of handling repetitive and precise tasks with a payload of up to 5 kg [63]. The intuitive programming framework, the overall easiness to use and the ability to handle lightweight and repetitive tasks constitute key advantages in using the UR5 robot during this dissertation project.

Figure 27 - Universal Robots UR5 [63].

Some of the technical specifications of the Universal Robots UR5 are presented on the following table.

Table 6 - Universal Robots UR5 Technical Specifications [64].

Weight	18.4 kg
Payload	5 kg
Reach	850 mm
Joint ranges	+/- 360º
Speed	180º/s
Repeatability	+/- 0.1 mm
Footprint	Ø 149 mm
Degrees of Freedom	6 rotating joints
Control Box Size	475 mm x 423 mm x 268 mm

The development and testing of the spatial changes detection software had into consideration the STAMINA use case, where the robotic fleet will navigate the logistic supermarket, in order to perform the kitting task. As such, it is necessary to compute the data acquired by the robotic camera system with reference to a global coordinate frame. Having all the data referencing a global coordinate frame allows a much easier comparison between data coming from several positions in space.

As a way of modeling this dynamic scenario, the camera system was fixed on the extremity of the presented robotic arm, which moved around during development and tests scenarios. The movement of the robotic arm allowed the acquisition of data by the camera system from several positions in space, simulating a real scenario, being the handling and processing of the acquired data done referencing a global coordinate frame. Correct calibration of the camera is important, as defining successfully the correct position of the camera in relation to the extremity of the robotic arm is a key component to acquire correctly placed spatial representations of reality.

As it was previously stated and analyzed in chapter 3, ROS provides a library that allows a standard and seamless way to keep track of coordinate frames within the entire system. The *tf* library is capable of handling all the computations needed to transform data between several coordinate frames, dealing with potential time-influenced incoherencies. The ability to seamlessly handle coordinate frames transformations in order to guarantee that the acquired data can be successfully processed and referenced to a common coordinate system constitutes an encapsulation of complexity that was crucial to the development of the spatial change detection software.

As previously mentioned in chapter 3, transforms and coordinate frames can be expressed as a graph with the transforms as edges and the coordinate frames as nodes. The transform tree representing the UR5 robot is shown on the next figure. This graphical representation was created by the *view_frames* tool, which is a component from the *tf* library.

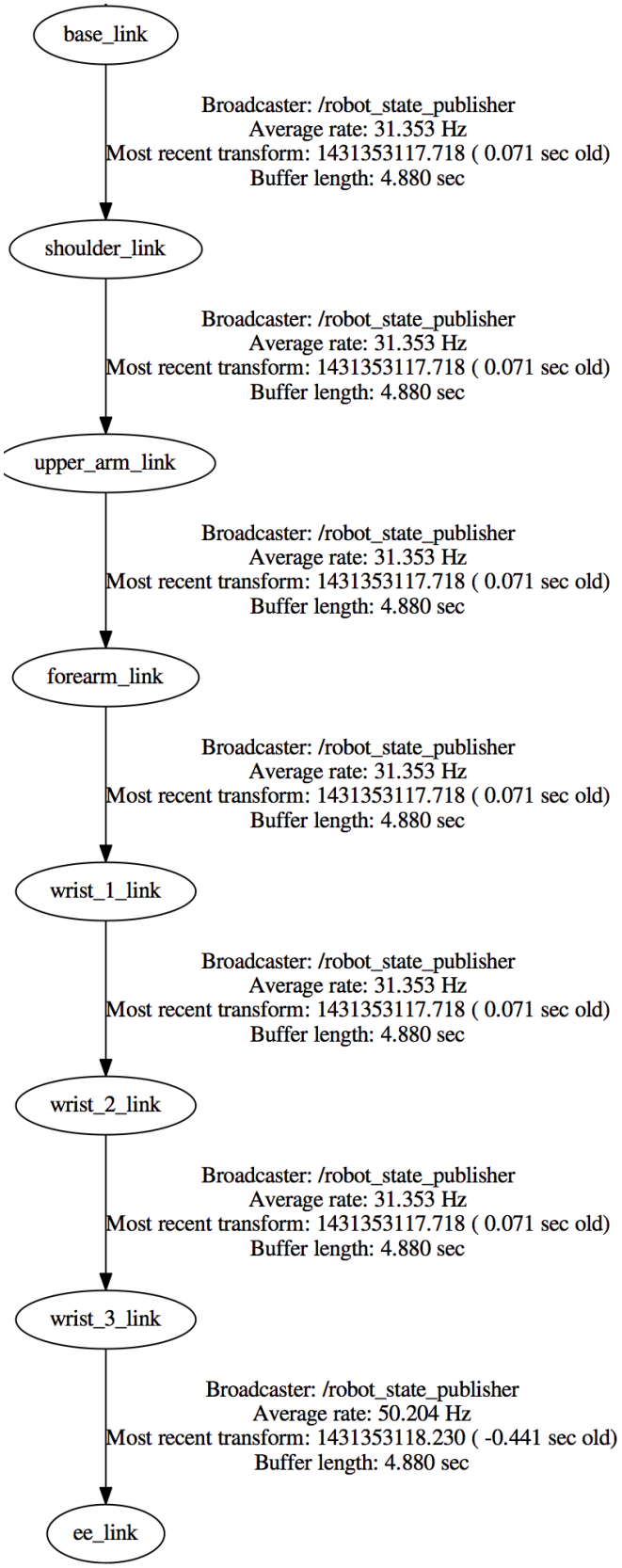


Figure 28 - Robotic Arm Transform Tree.

In order to properly model the robotic arm coordinate frames using the ROS *tf* library, an auxiliary package named *universal_robot* was used. The package was developed by several authors [65] and provides drivers, description files and utilities for Universal Robots arms. The package is compatible with the UR5 model and provides transformations between the several joints that can be handled by the *tf* library. The package is also composed by a set of description files which allow the robotic arm representation on RVIZ.

The camera system used during development and tests was also composed by a set of reference frames. The used ROS driver for OpenNI depth cameras used, *openni_camera*, is responsible for publishing the raw depth, RGB and infrared image streams. This package was also responsible for providing the appropriate coordinate frames used by the *tf* library. Similar to the robotic arm, it is also possible to render the representation of the camera in RVIZ, using the *hector_sensors_description* package, which contains URDF models of a generic camera sensor that can be used to generate a graphical representation.

Similarly to the robotic arm, the transform tree of the camera system can also be captured by the *view_frames* tool from the *tf* library, as is shown on the following figure.

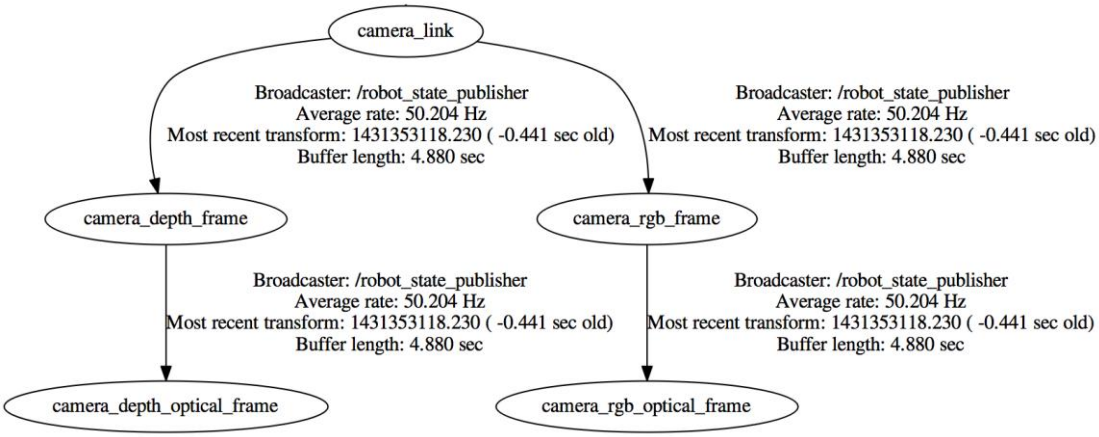


Figure 29 - Camera System Transform Tree.

In order to properly model the system composed by the robotic arm and the camera it is necessary to place the robotic arm in the scenario, referencing it to a global coordinate frame, and to connect the camera to the leaf node of the robotic arm transform tree, as shown in the following figure.

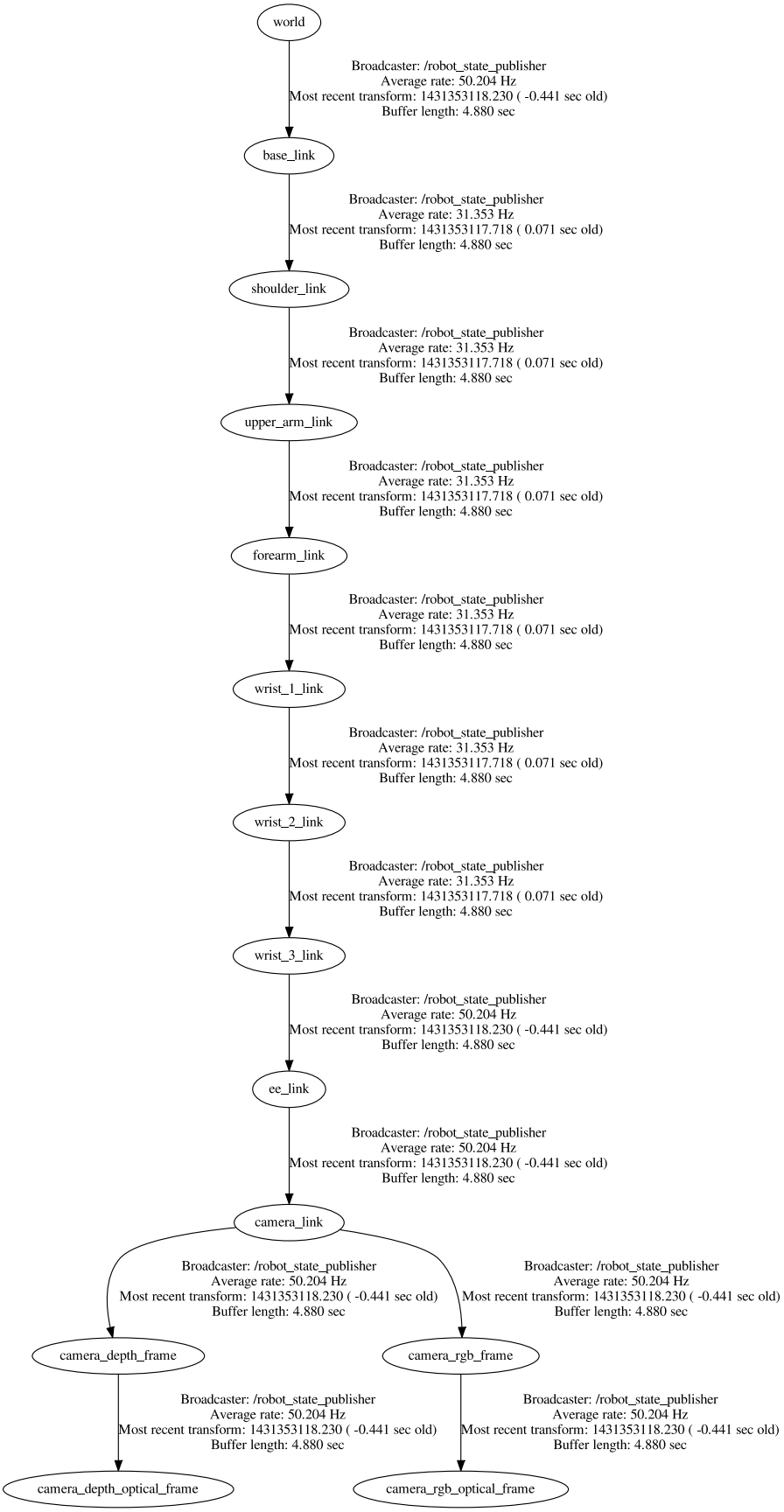


Figure 30 - Complete System Transform Tree.

Since the base of the robotic arm is expected to remain fixed on a given position in space, the transformation that connects the base of the robotic arm, *base_link*, to the global coordinate reference, *world*, is given by a simple translation matrix defined within the system configuration files. In this scenario, by simplification, the transformation from the base of the robotic arm to the world was defined by a simple elevation on the z axis, *i.e.*, the world reference is fixed immediately on the ground below the base of the robotic arm.

The connection between the extremity of the robotic arm, given by *ee_link*, and the camera system, *camera_link*, although also fixed in space during the execution of the process, is not as simple to configure as the first case described. Contrary to the first arbitrary transformation, a bad mapping between the camera frame and the tool frame of the robotic arm can result in several measurement errors. Although it was possible to physically measure the distances from the extremity of the robotic arm to the camera, it was not easy to have into consideration the inclination of the camera. As such, an automatized proceeding was developed in order to easily calibrate the camera in reference to the extremity of the robot.

The calibration process was developed as a standalone ROS package, which was composed by two phases. The first phase was developed as a Python ROS node which instructed the user to move the extremity of the robot to three non-collinear points, in order to define a plane in space. In order to simplify the process, a box surface was often used as a reference plane.

The position of the extremity of the robot, when it was touching each point, was recorded into a configuration YAML[3] file, referenced to the global coordinate frame. This first phase is responsible for storing the transformation from the extremity of the robot to the *world* reference frame.

[3] YAML is a human friendly data serialization standard for many programming languages. It was developed with the objective to represent data structures in a way that it is easily readable and edited by humans.

Figure 31 - Camera Calibration Process, phase 1. The robot is seen touching the plane defined by the surface of the box on three non-collinear points.

The second phase has the function of getting the transformation between the camera and the planar surface previously defined. This second step consisted in a C++ ROS node which presented a point cloud, instructing the user to signalize on the point cloud image displayed the three non-collinear points touched by the robot on the previous step. It required that the camera captured the planar surface previously touched by the robot, and, as such, the extremity of the robotic arm should be positioned as it is seen on the following figure.

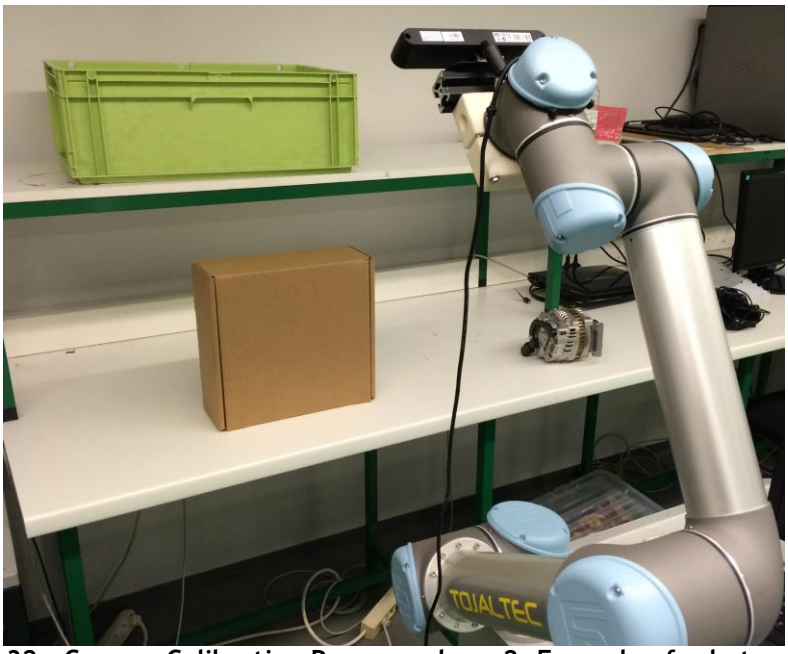


Figure 32 - Camera Calibration Process, phase 2. Example of robot position.

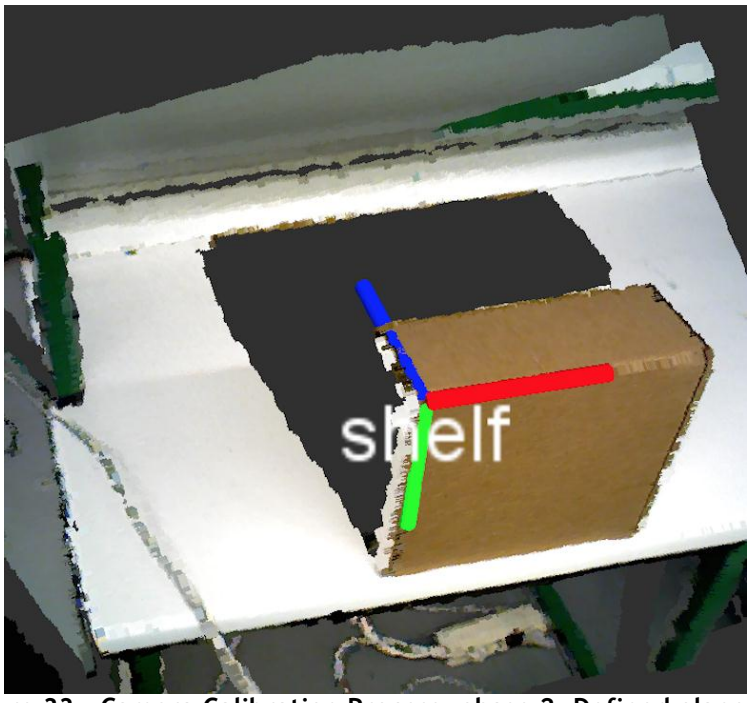


Figure 33 - Camera Calibration Process, phase 2. Defined plane.

By also defining a plane in space, it is then possible to compute not only the distance from the origin of the defined axis, but also the rotation, making it possible to compute a full transformation, from the origin of the defined axis within the plane to the camera reference frame.

The third phase in the calibration process is composed by a Python ROS node which makes the connection between the first and the second phase. As previously analyzed, the first phase was responsible for creating the transformation between the origin point of the plane defined and the world reference frame, while the second step aimed at getting the transformation between the same origin point and the camera reference frame. The third step closes the loop previously defined, connecting the camera frame to the extremity of the robotic arm, through the user defined plane. It is possible to store the transformations computed into system configuration files, in order to avoid a constant calibration every time that the robot is initialized. The graphical representation of the end result of the calibration process can be seen in the following figure.

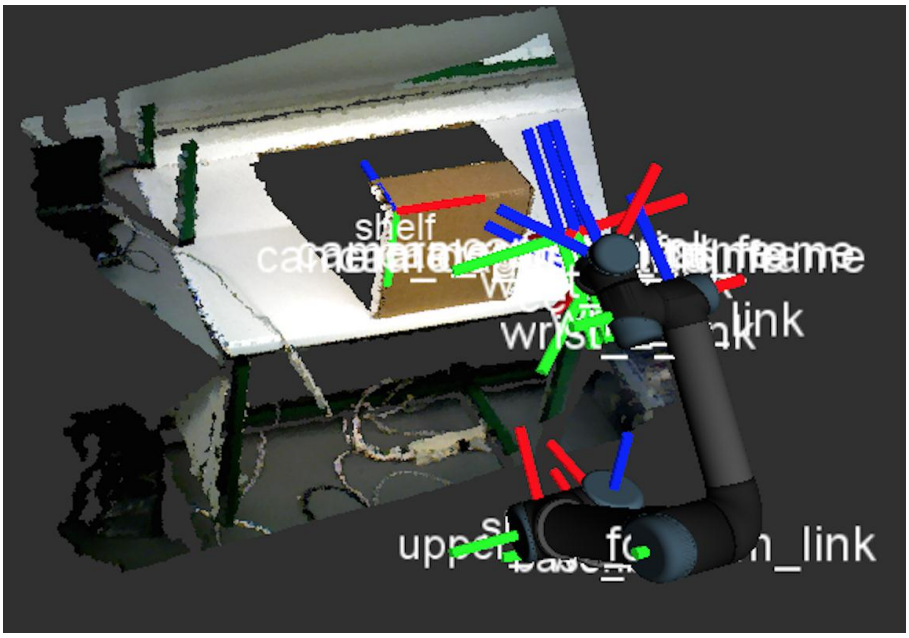


Figure 34 - Camera Calibration Process, phase 3.

The simplest process to confirm if the calibration process succeeded was to initialize the point cloud acquisition and introduce an accumulation time in RVIZ, in order to allow an overlap of the recent acquired images. By moving the robot, the image should not leave a trail, or a ghost, of the previous acquired images in different positions. The expected behavior should be the display of the acquired images without a noticeable overlap of points representing different components of the captured scenario on the same position. A ROS launch file was created to easily initialize the system in order to verify the results of a calibration.

4.3.5 – Utilitarian Tools

One of the core properties of ROS is the division of software into stand-alone parts which are responsible for small tasks. Having that ideology in mind, several utilitarian components were created to perform frequent tasks, such as recording a bag file for posterior analysis, playing back the information stored on a previously recorded bag file, and generating octrees based on the octomap mechanism. Those utilitarian tools developed will be addressed in this section.

Since access to the robot is not always available and replication of the test conditions sometimes demands long setup times, it was important to develop a launch file to record

experiments in order to test several software configurations using them. As previously mentioned, since the publish and subscribe message system is anonymous and asynchronous, the data transmitted by ROS messages can be recorded and stored in files typically referred as bags.

The launch file responsible for recording a bag file for latter usage records raw depth image and RGB image from an OpenNI supported device, such as the ASUS Xtion PRO LIVE camera. The depth image is registered with OpenNI firmware, using calibration specifications provided for the specific camera used. The launch file also records the *tf* messages, responsible for keeping track of coordinate frames and transform data within the entire system.

In order to play the bag file recorded using the record launch file previously described, it was necessary to develop a playback launch file, which opens the bag file passed as an argument to the *roslaunch* command and plays back the raw depth image and RGB image recorded using the associated record launch file through an OpenNI device. The post-processing needed is executed in order to generate a stream of point clouds. The playback launch file executes all the needed processes initialized by the *openni* launch file, but communicates with it, in order to tell it to not get any data from a real camera, replacing it with the one supplied on the bag file.

There is an additional option embedded into the playback launch file, related with the playback simulation time. Usually, ROS uses the computer's system clock as the time source. Although, when running a simulation or playing back a bag file recorded, it might be desirable to use a simulated clock, in order to have control of the playback time, allowing the simulation to be accelerated or slowed [66].

Generating real time OctoMap based on the depth image being captured by the camera in real time is possible using the octomap_server launch file, which launches the octomap_server ROS node with the appropriate parameters for the use case. The node launched is responsible for building and distributing volumetric three-dimensional occupancy maps incrementally built from incoming range data [67].

Analogous to what was previously described, it was necessary to develop tools for recording octree files and playing back messages from previously recorded files. As such, a utilitarian ROS node was created in order to record octrees into *ot* files. The tool, named *octomap_msg2file*, receives as input the topic where octrees are being published, set to *"/octomap"* as a default, and as output the name of the file to be saved, set to *"tmp.ot"* as

default. The node *file2octomap_msg*, as the name suggests, was developed to play back the octomap message from a file previously recorded. It receives as an input the name of the file where the octomap is stored and as output the name of the topic to where it should publish.

In order to easily initiate the several operation modes that the software specifies, ranging from calibration to the actual spatial change detection executable, several launch files were created. Most of the launch files created allow the modification of several parameters and arguments, allowing a great degree of customization.

4.3.5 – Octree Comparison Tests and Results Analysis

The spatial change detection node, as previously stated, has two modes of operations. Both modes of operation aim at detecting spatial changes, either comparing two octrees or comparing an octree and a set of regions defined as free or occupied space. This subchapter aims at analyzing the spatial change detection between two octrees. Although the results presented were obtained using the octree comparison mode of operation, the region definition mode of operation render similar results, assuming that the regions in space are well defined.

In order to produce results which can be measured and are significant, it is necessary to create different scenarios in order to check the occurrences of different situations. However, as previously stated, there are several scenarios of inconsistencies, making it wise to choose the most common scenarios, in order to limit the number of experiments and make the analysis of results plausible. As such, the detection of incoherent objects on a rack was chosen as the test scenario. As the objects chosen are representative of the smallest objects present on the use case, it is expected that the comparison mechanism can scale up to work successfully with larger objects.

To produce the test results, five scenarios were constructed, each corresponding to a different arrangement of objects on different positions on the test rack. In order to acquire the point cloud data from the five scenarios in a consistent and uniform way, a movement procedure was created for the Universal Robots UR5 and then reproduced in each test scenario. The movement was a series of poses that allowed observing the rack from several positions, during approximately 90 seconds. The UR5 robot base was always fixed on the same place during execution, and, as such, the origin of the world node does not change between different scenarios.

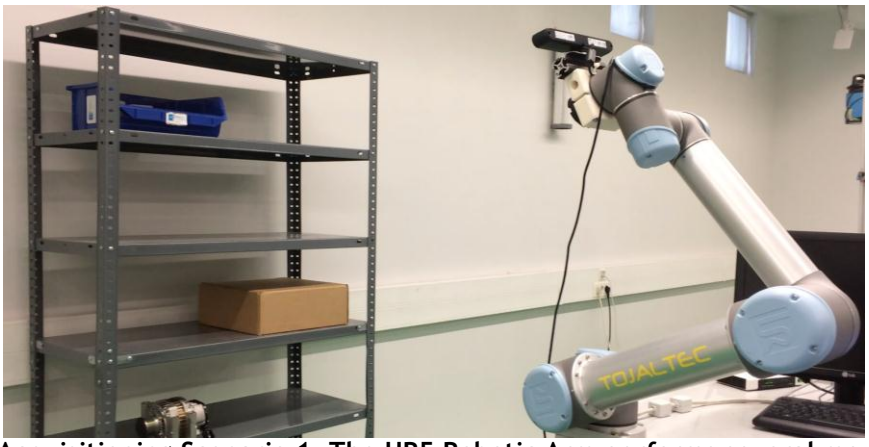


Figure 35 - Acquisitioning Scenario 1. The UR5 Robotic Arm performs several movements, always remaining fixed on an unmoving table.

Besides making the acquisition procedure uniform, the scenarios were captured sequentially in a short time interval and stored into bag files, in order to secure that the replication of testing occurs always on the same conditions. The produced scenario can be seen on the following group of pictures.

Figure 36 - Test Scenarios. From Bottom to Down and from Left to Right: Scenario 1, 2, 3, 4 and 5, respectively.

Using the Searchable Space Definition Tool previously mentioned, a region around the racks was created, and then, an also previously mentioned octree construction procedure was used to construct the octrees that would be used on the comparison algorithms. As the robotic arm moves during the movement, it was necessary to also capture the coordinate frames used by the *tf* library, in order to successfully construct the octrees using the OctoMap framework, having into consideration the movement of the robot while it was capturing the scenario.

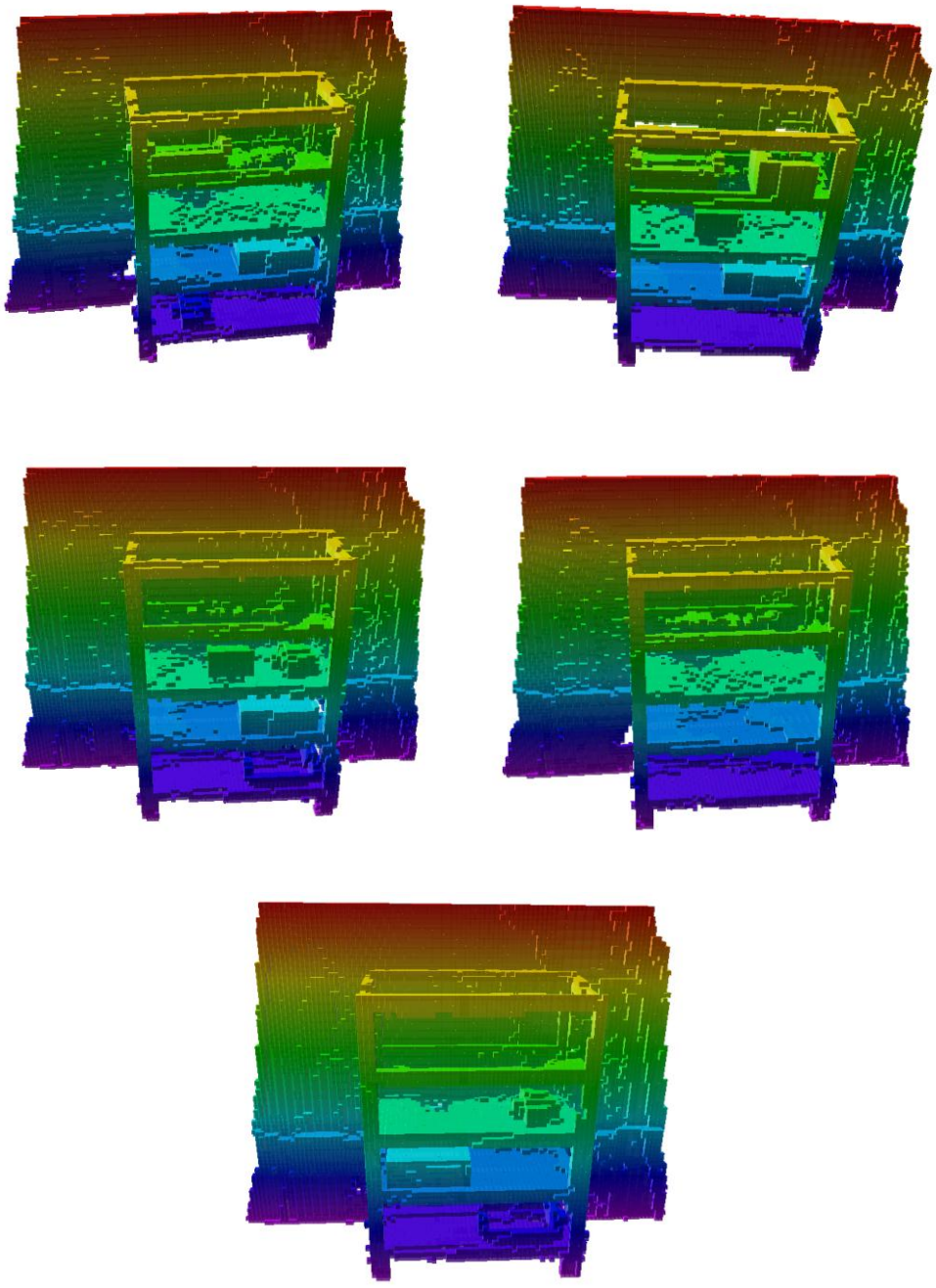


Figure 37 - Octree Representation. From Bottom to Down and from Left to Right: Scenario 1, 2, 3, 4 and 5, respectively.

By defining five different scenarios, and comparing each one with the others, it is possible to generate 20 different comparison scenarios. It should be noted that the tests were made considering a pair of scenarios as model and target separately, *i.e.*, for scenario 1 and 2, for example, both served as a model and as a target, so there were two comparisons made between them.

In order to verify the successfulness of the spatial change detection algorithm, a ground truth table was constructed, specifying the amount of inconsistencies of each kind, exceeding or missing, between two models. As previously mentioned, an exceeding inconsistency represents the detection of an addition of an object on the target scenario that is not present on the model scenario. The missing inconsistency represents the removal of an object from the target scenario, *i.e.*, an object that was present on the model scenario but has been removed on the target scenario. The developed table is presented following, with information on the 20 different combinations compared.

Table 7 - Comparison analysis between several combinations of scenarios.

Combination	Model Scenario	Target Scenario	Exceeding Inconsistencies	Missing Inconsistencies	Unmoving Objects
1	1	2	2	1	2
2	1	3	3	2	1
3	1	4	3	0	0
4	1	5	3	3	0
5	2	1	1	2	2
6	2	3	2	2	2
7	2	4	0	4	0
8	2	5	2	4	0
9	3	1	2	3	1
10	3	2	2	2	2
11	3	4	0	4	0
12	3	5	1	2	2
13	4	1	3	0	0
14	4	2	4	0	0
15	4	3	4	0	0
16	4	5	3	0	0
17	5	1	3	3	0
18	5	2	4	3	0
19	5	3	2	1	2
20	5	4	0	3	0

The test bench to be made has been divided into two different classes, the class corresponding to the exceeding inconsistencies and the class corresponding to the missing inconsistencies. By dividing the analysis into two different classes and by knowing the ground truth, expressed on the previously presented table, it is possible to determine the number of successful and unsuccessful inconsistencies detected. To each class, there are four different outcomes, expressed on the following table:

Table 8 - Contingency Table.

Total Population		Condition (Ground Truth)	
		Condition Positive	Condition Negative
Test Outcome	Test outcome Positive	True Positive (TP)	False Positive (FP)
	Teste outcome Negative	False Negative (FN)	True Negative (TN)

On the experiments produced, the **True Positive (TP)** signalizes a successful detection by the spatial change detection algorithm. Similarly, the **True Negative (TN)** represents the absence of a spatial change detection when there is, in fact, no change between scenarios. The TN occurrence was tested using the unmoving objects between two scenarios. If the algorithm did not signalize a constant object as an inconsistency between scenarios, then a TN occurrence is verified.

It is possible to detect two different kinds of errors on this experiment. The **False Positive (FP)** represents a detected inconsistency that is not verified in reality, being usually due to an error in acquisition or a set of permissive thresholds, that do not filter the inconsistencies successfully. Contrasting, the **False Negatives (FN)** represents the absence of an inconsistency when an incoherence situation is in reality verified.

The exceeding or missing occupation ratio threshold and the volume threshold are parameters that can be changed on runtime. There is, on this phase of development, the need to select the appropriate parameters in order to better detect the inconsistencies. The selection of parameters constitutes a trade-off between precise accuracy, where only true inconsistencies are detected, risking losing some detections that go under the defined threshold, or a more relaxed inconsistency detection, where it is almost guaranteed that all

inconsistencies will be detected, but with some incorrect occurrences also being flagged as an inconsistency.

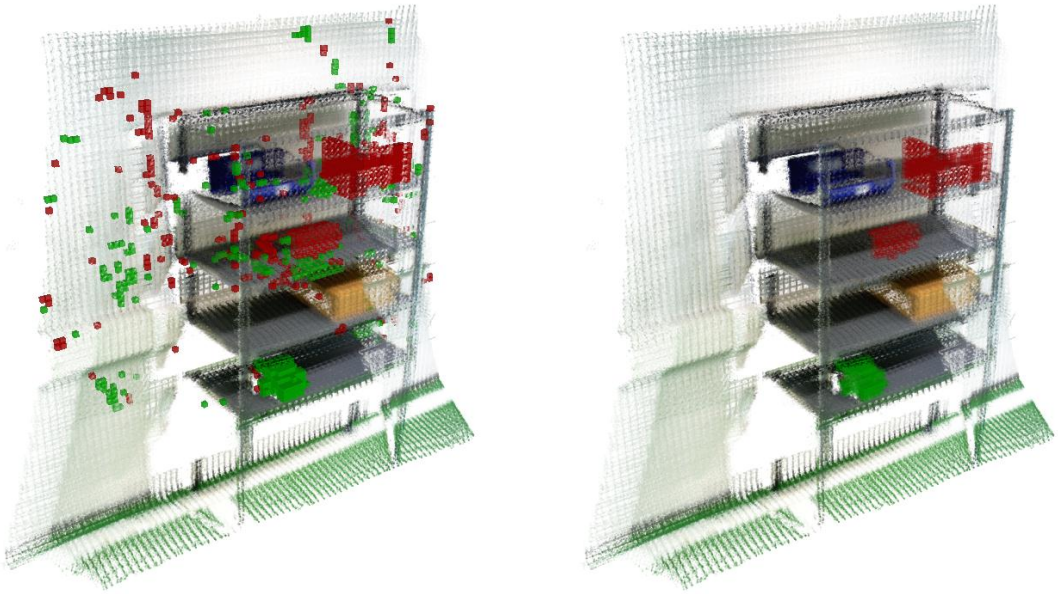
As there is the need to analyze several testing conditions, the threshold parameters were chosen in order to reflect real potential choices of parameters. The selection of threshold parameters to this testing process excluded values of parameters that produced an unrealistic combination or that were too strict or too relaxed in terms of filtering of inconsistencies. The chosen values for the testing procedure are represented on the next table.

Table 9 - Testing Parameters Selected.

Exceeding or Missing Occupation Threshold	0.0%	10.0%	20.0%		
Volume Threshold	0.0001 m^3	0.0005 m^3	0.001 m^3	0.002 m^3	0.004 m^3

Each test consisted on combining the several possible pairs of the chosen parameters values, and counting the number of TP, TN, FP and FN detected for each class: exceeding or missing. For a set of model and target scenarios, there were 15 combinations possible between the different parameters chosen.

On the following picture it is possible to see the visual result of a testing procedure. On this specific testing, the scenario 1 had the model role, while the target was scenario 2. The inconsistencies were signalized by several colors displayed using a color map, being the hot colors, ranging from red to yellow, the **exceeding inconsistencies**, and the cold colors, ranging from blue to green, the **missing inconsistencies**. As it is possible to verify, on this particular case, the algorithm successfully detects two exceeding inconsistencies (red and orange clusters) and one missing inconsistency (blue cluster). The model point cloud is shown as a visual aid to allow an easy visualization of the incoherencies.



(a) *(b)*

Figure 38 - Spatial Change Detection between Scenario 1 (model) and Scenario 2 (target): *(a)* unfiltered clusters, *(b)* filtered clusters.

After performing the experiment on the 20 combinations between possible scenarios pairs, the number of occurrences of the TP, FP, TN, FN detected for each set of thresholds chosen were summed. These several sums allowed the calculation of two evaluation metrics, which characterizes the rate of successful and unsuccessful inconsistency detection.

The **True Positive Rate (TPR)** is related with the sensitivity of the test, measuring the proportion of positives which are correctly identified as such, *i.e.*, the proportion of inconsistencies that are successfully signalized. The TPR is specified as follows:

$$True\ Postive\ Rate\ (TPR) = \frac{TP}{TP + FN} \tag{4.4}$$

The **False Positive Rate (FPR)** is an evaluation metric also known as the False Alarm Ratio, as it measures the proportion of falsely rejecting the null hypothesis for a particular test, *i.e.*, the ratio of inconsistencies that are signalized as an inconsistency wrongly. The FPR is defined as:

$$False\ Postive\ Rate\ (FPR) = \frac{FP}{FP + TN} \tag{4.5}$$

The **Receiver Operating Characteristics (ROC)** [68] graph was used in order to represent the tradeoff between the successful detection of inconsistencies and the false alarms for the several sets of parameters testes. The ROC graph for the exceeding and the missing classes are presented on the following figures.

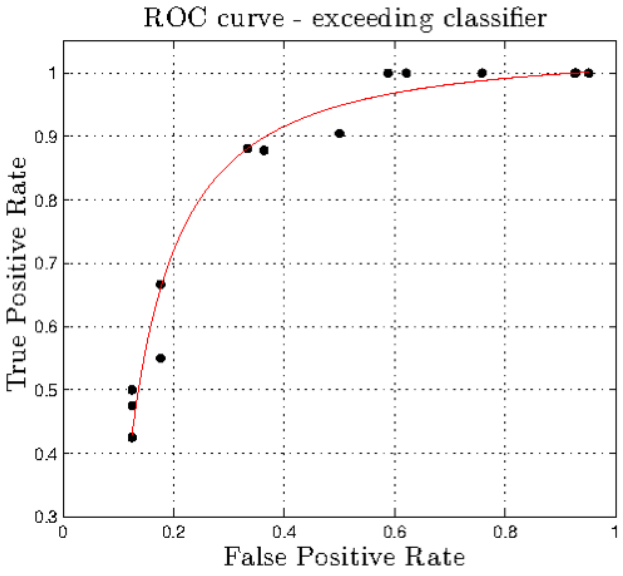


Figure 39 - ROC Graph - Exceeding Classifier

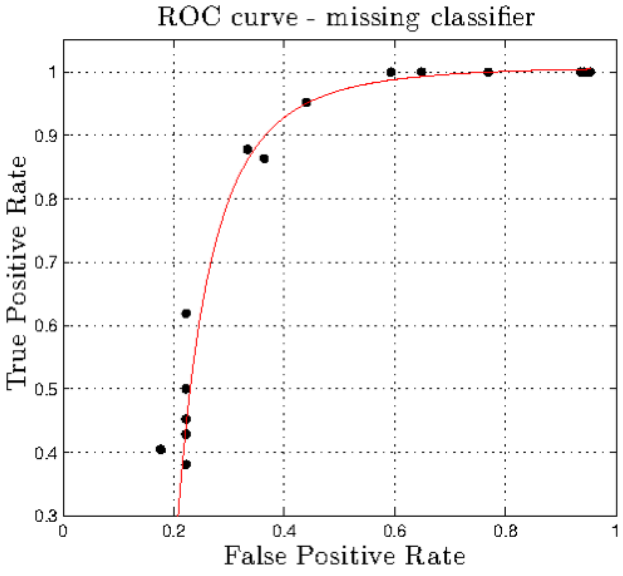


Figure 40 - ROC Graph - Missing Classifier

The analysis of both classifiers ROC Graphs allows to draw some conclusions to the testing process. As it was previously stated, choosing a particular set of parameters has a key importance on the behavior of the Spatial Change Detection algorithm. In order to reach a successful level of performance, the set of parameters should be chosen in order to try to maximize the True Positive Rate and to minimize the False Positive Rate.

It is also easy to verify that as the True Positive Rate grows to 100%, the False Positive Rate also grows significantly. This pattern is explained as a relaxation of the detection criteria: if the algorithm is capable of detecting all the incoherencies introduced on the system, the tradeoff is also detection an increased number of errors. Similarly, as the set of parameters get stricter, the True Positive Rate gets smaller, *i.e.*, few incoherencies are detected, but the False Positive Rate is closer to zero, *i.e.*, there are few errors signalizing incoherencies.

For the scenario tested, the best set of parameters were the 0.0% of Exceeding and Missing Occupation Ratio and the 0.001 m^3 of Volume Threshold. With this settings, the True Positive Rate and the False Positive Rate were approximately 88,1% and 33,3%, respectively for the Exceeding Classifier and 87,8% and 33,3% for the Missing Classifier.

4.3.6 – Region Definition and Mobile Robot Testing

It is also worth to mention that another testing procedure was executed, in order to evaluate the success of spatial change detection using the region definition mode of operation. This testing procedure had also the objective of assessing if the developed spatial change detection software could be successfully used on a mobile robot.

Figure 41 - Mobile Robot Acquiring Point Cloud data.

As it can be seen on the following picture, an ASUS Xtion PRO Live camera, the same model used on the previous analyzed testing procedure, was mounted and calibrated on the side of a mobile robot.

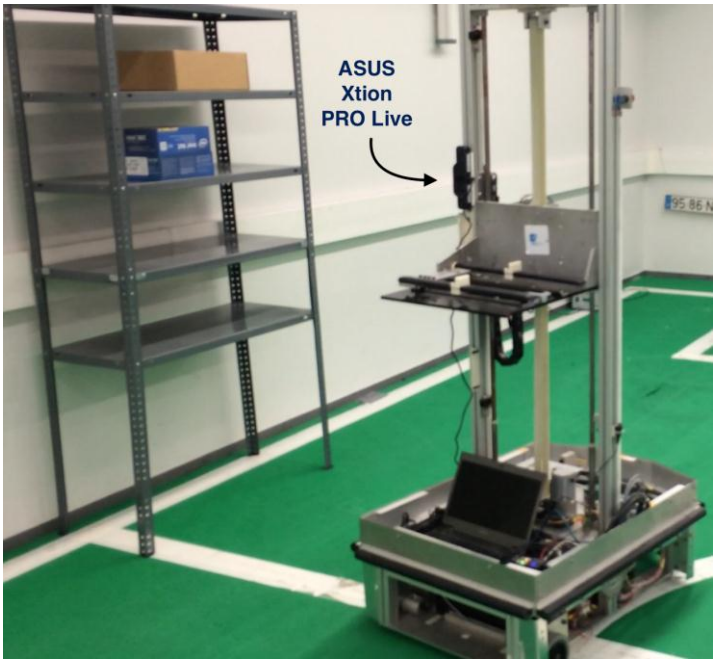


Figure 42 - ASUS Xtion PRO Live mounted on a mobile robot.

A trajectory for the robot was defined, as to simulate a movement along a linear logistic supermarket. The robot had the ability to automatically execute the defined trajectory, localizing itself on the test laboratory. As the robot system is implemented using a ROS framework, it is possible to access the coordinate frame transformations from the calibrated camera system to the robot system and then to the world system, similarly to what has previously analyzed for the robotic arm case.

In the following figure it is possible to observe the transformation tree of the system. It should be noted that in the presented tree, for visualization purposes, the camera and robotic system are presented as decoupled, being the coupling performed by the camera calibration procedure.

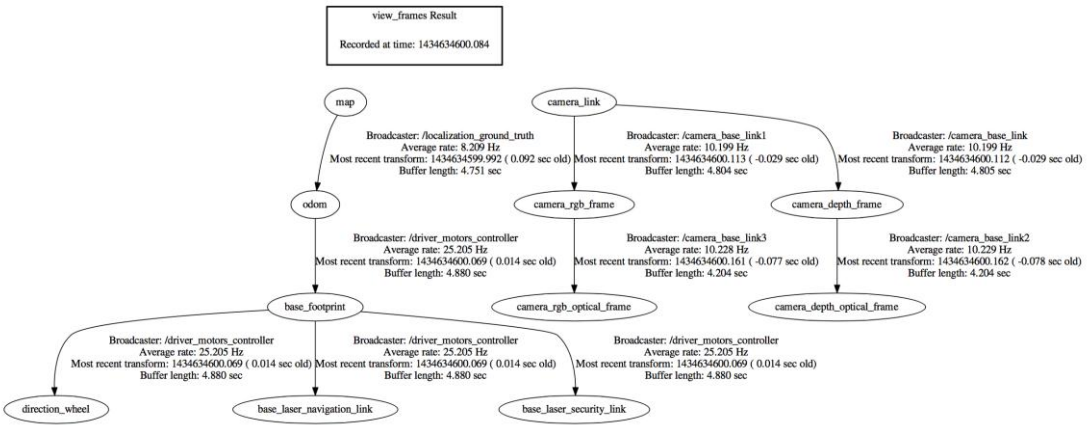


Figure 43 - Mobile Robot System and Camera System Transformation Tree.

The first stage of the testing procedure was composed by a complete movement of the robot along its defined trajectory, in order to capture a point cloud that would constitute the base to define the free and occupied regions using the previously presented Region Definition Tool.

The following figure presents the mobile robot defined trajectory as well as the testing laboratory walls detected by the robot laser scan. It is also possible to observe the robot's and camera system coordinate frames, as well as the approximate position of the testing rack.

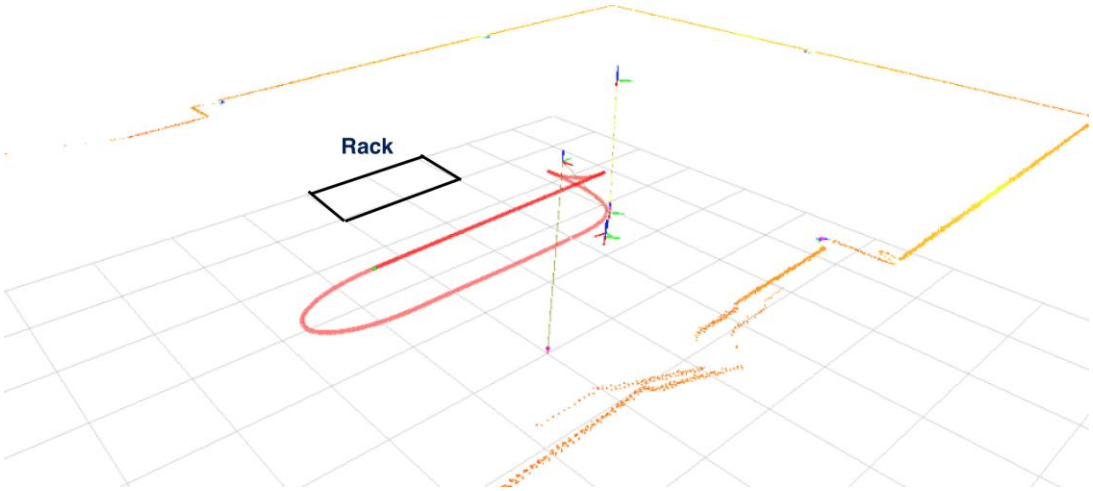


Figure 44 - Mobile Robot Trajectory (red). The yellow and orange points represent the laboratory walls, as detected by the mobile robot laser scan system.

The second stage was the spatial change detection phase, where the robot moved along the defined trajectory, acquiring point clouds through its camera system. The acquired point clouds were then used by the OctoMap framework to compose octrees modeling the free and occupied space of the environment. As the scenario captured was composed by a large area, it was necessary to only compute the octree on the space where inconsistencies should be checked.

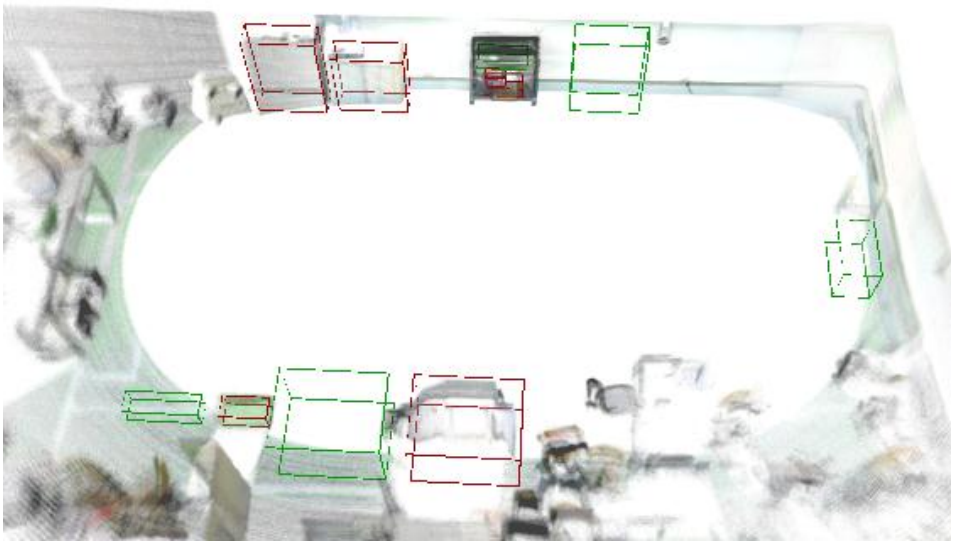


Figure 45 – Region definition. Green spaces represent space that should be free and red spaces represent space that should be occupied.

On the testing scenario, two regions were defined as free, while the lower region was defined as occupied. To test if the spatial change detection algorithm successfully worked, after calibrating the occupation ratio threshold and the volume threshold to the appropriate values, a box was placed on the top region and a box was removed from the lower space, as to produce an exceeding inconsistency and a missing inconsistency, respectively. Additionally, the middle region of the rack, defined as a free space was kept empty. The obtained results, with the correct inconsistency detections can be seen on the next figure.

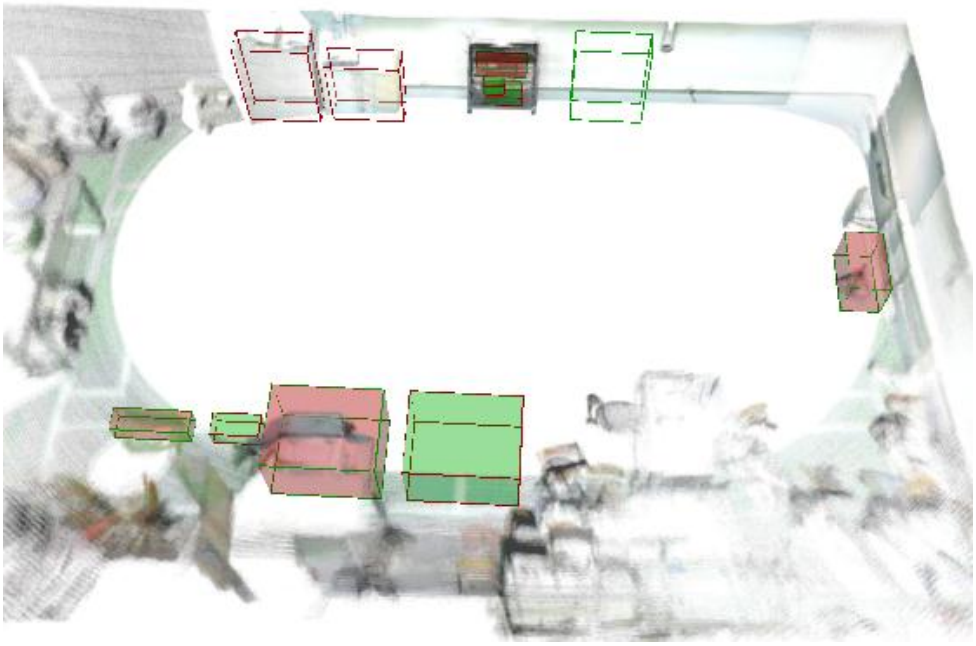


Figure 46 - Exceeding inconsistency (red) and Missing inconsistency (green).

Due to the short time available at the end of this dissertation project, further testing is required in order to validate the solution, although the results obtained from this preliminary assessment is completely satisfactory, as all the system components worked successfully.

Chapter 5

Conclusions and Future Work

The automotive industry has changed its strategy to a production focused on personalization of products. Nowadays, the major automotive brands offer several different automotive configurations, in order to cope with the needs of costumers and to stay competitive. This new approach to production lead to a new arrangement of tasks in the production line.

Kitting operations are now a reality in some production lines even outside of the automotive industry. As there is the need to reduce the health risks associated with the kitting tasks performed by human operators, several small and medium enterprises are more leaned into implementing robotic solutions to solve the health hazards to an ageing European workforce.

There are several challenges on developing a robotic system to perform complex operations such as kitting tasks while co-existing in a dynamic environment with human operators. One of the most challenging problem is related with the navigation of autonomous mobile manipulators, which must rely on a data structure in order to navigate and find key objects. As the kitting zone is a dynamic environment, characterized by a constant reposition of new containers and rearrangements of objects, it is necessary to develop a reliable mechanism to dynamically update the data structure in real time. The challenge gets further complex when there is also a dynamic and frequent update by the factory enterprise system.

As previously stated, in the past, depth camera systems for creating a reliable and high-quality three-dimensional representation of a logistic supermarket would be likely to exceed the budget of most of the robotics projects. However, nowadays, with the advent of system

like the Microsoft Kinect, three-dimensional perception has become the cheaper and most effective solution to a dynamical representation of a three-dimensional space.

The new sensing hardware and its associated perception have become, slowly, the new standard in robotics vision systems. As such, an algorithm for detecting and dealing with spatial inconsistencies as the one presented on this dissertation project constitute not only a step forward to the R&D on the field of robotics perception, but also can be seen as a case study for a real industrial robotics application, on a field as demanding as the automotive industry.

The solution proposed on this dissertation project was composed by two modes of operations. The tests and the respective results of the comparison between two octrees were previously analyzed, and it can be concluded that it is in fact possible to successfully detect spatial changes using the proposed algorithms. Several techniques were implemented in order to filter results and to provide a better overall result. As it was demonstrated, there are several parameters that allow a modification of the behavior of the algorithm, which allows its reusability in diverse scenarios. It is also important to notice that the tests conducted had the objective of showcasing the operation of the developed system according to a wide range of parameters. As such, it is possible to fine tune the final solution to achieve even higher success levels.

The region definition development and tests conducted using a mobile robot, although only briefly mentioned, compose a key component of this dissertation work. This mode of operation makes it possible to easily integrate the developed algorithm with a factory enterprise system, since the region definition can be either used to manually specify which areas should be free or occupied or it can even be replaced by an automatic and integrated system. Besides, the usage of the camera system on a moving robot demonstrates that the project can be successfully integrated into the STAMINA autonomous mobile manipulators successfully.

Even though the preliminary tests made with the camera system mounted on a mobile robot were very successful, further testing is necessary to validate the usability of this solution on mobile manipulators on a factory scenario. This is the evident next step on continuing the R&D initiated by this dissertation project.

It is also advisable to further testing the developed mechanism to detect inconsistencies such as humans and obstacles. It is expected that the proposed algorithm can successfully handle spatial change detection of objects larger than the small boxes tested, although further tastings are required.

The proposed solution was developed as independent ROS packages, that can easily be integrated with previous existing STAMINA components. The development process was always focused on allowing a simple integration with the processes being developed by the STAMINA partners. However, it was not possible to integrate the developed solution with the real STAMINA system, being that step a logical future proceeding. Although some of the architecture integration, update mechanisms and message protocols have been already defined, a further revision of the proposed structures should be addressed to allow a coherent integration.

Bibliography

[1] Aalborg University, “Stamina Summary Negotiation Meeting Presentation,” 2014.

[2] C. Nielsen, “Easy to Use Robots are Future Colleagues in Small Businesses,” 2014.

[3] STAMINA Project, “About Stamina,” [Online]. Available: http://stamina-robot.eu/about-stamina. [Acessed on 28 January 2015].

[4] European Union, “Seventh Framework Programme (2007 to 2013),” [Online]. Available: http://europa.eu/legislation_summaries/energy/european_energy_policy/i23022_en.htm . [Acessed on 17 June 2015].

[5] A. Lasnier e A. Chazoule, “D1.1.3 - First public report on use case and experiment specification including evaluation criteria,” STAMINA Project, 2014.

[6] N. Mabire e B. Jacq, “D1.2.1 - Report on Concept, Architecture and Integration of STAMINA demonstrator (version M11),” STAMINA Project, 2014.

[7] PSA Peugeot Citroën, “Revenue, sells, employees… Key financial datas,” [Online]. Available: http://www.psa-peugeot-citroen.com/en/automotive-group/overview/key-financial-data. [Acessed on 29 May 2015].

[8] STAMINA Project, “Partners - STAMINA,” [Online]. Available: http://stamina-robot.eu/partners. [Acessed on 28 February 2015].

[9] BA Systèmes, “Systèmes Logistiques par AGV - Chariot Automatique,” [Online]. Available: http://www.basystemes.com/about-us/basystemes/. [Acessed on 2 February 2015].

[10] A. Chazoule, C. Toscano, E. Fauré e G. Veiga, “D4.2.1 - Specification of the integration services with ERP systems and related sub-systems,” STAMINA Project, 2014.

[11] M. R. Pedersen, L. Nalpantidis, A. Bobick e V. Krüger, “On the Integration of Hardware-Abstracted Robot Skills for use in Industrial Scenarios,” em *2nd International IROS Workshop on Cognitive Robotics Systems: Replicating Human Actions and Activities*, 2013.

[12] F. Rovida, “D3.1 - Preliminary report on skill architecture,” STAMINA Project, 2014.

[13] R. Mukundan, Advanced Methods in Computer Graphics, Springer, 2012.

[14] C. Toscano, “3D Computer Graphics,” 2014.

[15] JSON, [Online]. Available: http://www.json.org. [Acessed on 23 June 2015].

[16] N. Nurseitov, M. Paulson, R. Reynolds e C. Izurieta, “Comparison of JSON and XML Data Interchange Formats: A Case Study”.

[17] Point Cloud Library (PCL), “About,” [Online]. Available: http://www.pointclouds.org/about/. [Acessed on 15 May 2015].

[18] R. Rusu, Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments, Institut für Informatik der Technischen Universität München, 2009.

[19] R. Rusu e S. Cousins, "3D is here: Point Cloud Library (PCL)," em *IEEE International Conference on Robotics and Automation (ICRA)*, 2011.

[20] Point Cloud Library (PCL), "Documentation: Getting Started / Basic Structures," [Online]. Available: http://pointclouds.org/documentation/tutorials/basic_structures.php. [Acessed on 1 June 2015].

[21] Point Cloud Library (PCL), "Spatial change detection on unorganized point cloud data," [Online]. Available: http://pointclouds.org/documentation/tutorials/octree_change.php. [Acessed on 2 June 2015].

[22] S. Liu, K.-C. Chan e C. C. Wang, "Iterative Consolidation of Unorganized Point Clouds," *IEEE Computer Graphics and Applications,* 2010.

[23] J. Delmerico, "PCL Tutorial: The Point Cloud Library by Example," 2013.

[24] Point Cloud Library (PCL), "Adding your own custom PointT type," [Online]. Available: http://pointclouds.org/documentation/tutorials/adding_custom_ptype.php#adding-custom-ptype. [Acessed on 2 June 2015].

[25] Point Cloud Library (PCL), "The PCD (Point Cloud Data) file format," [Online]. Available: http://pointclouds.org/documentation/tutorials/pcd_file_format.php. [Acessed on 2 June 2014].

[26] Point Cloud Library (PCL), "PCL Walkthrough," [Online]. Available: http://pointclouds.org/documentation/tutorials/walkthrough.php. [Acessed on 2 June 2015].

[27] K. Wenzel, M. Rothermel, D. Fritsch e N. Haala, "An Out-of-Core Octree for Massive Point Cloud Processing".

[28] J.-Y. Sim, S.-U. Lee e C.-S. Kim, "Construction of Regular 3D Point Clouds Using Octree Partitioning and Resampling".

[29] Point Cloud Library (PCL), "Spatial Partitioning and Search Operations with Octrees," [Online]. Available: http://pointclouds.org/documentation/tutorials/octree.php. [Acessed on 4 June 2015].

[30] H. Samet, "An Overview of Quadtrees, Octrees and related hierarchical Data Structures," em *Theoretical Foundations of Computer Graphics and CAD*, Springer Berlin Heidelberg, 1988, pp. 51-68.

[31] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss e W. Burgard, "OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees," *Autonomous Robots,* 2013.

[32] M. Hebert, C. Caillas, E. Krotkov, I. Kweon e T. Kanade, "Terrain mapping for a roving

planetary explorer," em *IEEE Int. Conf. on Robotics & Automation (ICRA)*, 1989.

[33] R. Triebel, P. Pfaff e W. Burgard, "Multi-level surface maps for outdoor terrain mapping and loop closing," em *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2006.

[34] Y. Roth-Tabak e R. Jain, "Building an environment model using depth information," *Computer 22,* nº 6, pp. 85-90, 1989.

[35] H. Moravec, "Robot spatial perception by stereoscopic vision and 3D evidence grids," 1996.

[36] K. Wurm, A. Hornung, M. Bennewitz, C. Stachniss e W. Burgard, "OctoMap: A probabilistic, flexible, and compact 3D map representation for robotic systems," em *ICRA 2010 Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipula- tion*, 2010.

[37] H. Moravec e A. Elfes, "High resolution maps from wide angle sonar," em *IEEE Int. Conf. on Robotics & Automation (ICRA)*, St. Louis, MO, USA, 1985.

[38] M. Yguel, O. Aycard e C. Laugier, "Update policy of dense maps: Efficient algorithms and sparse representation," em *Field and Service Robotics, Results of the Int. Conf., FSR 2007*, 2007.

[39] ROS, "ROS.org," [Online]. Available: http://www.ros.org/about-ros/. [Acessed on 7 June 2015].

[40] ROS, "ROS Wiki," [Online]. Available: http://wiki.ros.org/ROS/Introduction. [Acessed on 7 June 2015].

[41] J. M. O'Kane, A Gentle Introduction to ROS, Columbia, SC: University of South Carolina, 2014.

[42] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler e A. Ng, "ROS: an open-source Robot Operating System," em *ICRA workshop on open source software*, 2009.

[43] ROS, "ROS Wiki," [Online]. Available: http://wiki.ros.org/ROS/Concepts. [Acessed on 8 June 2015].

[44] M. Oliveira, "Automatic Information and Safety Systems for Driving Assistance," 2013.

[45] Gazebo, "Gazebo," [Online]. Available: http://gazebosim.org.

[46] ROS, "urdf," [Online]. Available: http://wiki.ros.org/urdf. [Acessed on 10 June 2015].

[47] Gazebo, "Tutorial : URDF in Gazebo," [Online]. Available: http://gazebosim.org/tutorials/?tut=ros_urdf. [Acessed on 10 June 2015].

[48] Open Source Robotics Foundation, [Online]. Available: http://sdformat.org. [Acessed on 10 June 2015].

[49] T. Foote, "tf: The Transform Library".

[50] K. Litomisky, “Consumer RGB-D Cameras and their Applications,” 2012.

[51] Microsoft, “Kinect for Windows Sensor,” [Online]. Available: https://msdn.microsoft.com/en-us/library/hh855355.aspx. [Acessed on 20 June 2015].

[52] ASUS, “Xtion PRO LIVE,” [Online]. Available: http://www.asus.com/Commercial_3D_Sensor/Xtion_PRO_LIVE/specifications/. [Acessed on 24 June 2015].

[53] N. Villaroman, D. Rowe e B. Swan, “Teaching Natural User Interaction Using OpenNI and the Microsoft Kinect Sensor,” em *SIGITE’11*, New York, NY, USA, 2011.

[54] PCL, “Spatial change detection on unorganized point cloud data,” [Online]. Available: http://pointclouds.org/documentation/tutorials/octree_change.php. [Acessed on 24 June 2015].

[55] Point Cloud Library (PCL), “Point Cloud Library (PCL): pcl::octree::OctreePointCloudChangeDetector< PointT, LeafT > Class Template Reference,” [Online]. Available: http://docs.pointclouds.org/1.1.1/classpcl_1_1octree_1_1_octree_point_cloud_change_detector.html. [Acessed on 24 June 2015].

[56] M. Fischler e R. Bolles, “Random Sample Consensus: A Paradigm for Model Fitting with Apphcatlons to Image Analysis and Automated Cartography,” nº 24.6, 1981.

[57] Point Cloud Library (PCL), “pcl::SACSegmentation< PointT > Class Template Reference,” [Online]. Available: http://docs.pointclouds.org/trunk/classpcl_1_1_s_a_c_segmentation.html. [Acessed on 24 June 2015].

[58] Point Cloud Library (PCL), “Module filters,” [Online]. Available: http://docs.pointclouds.org/trunk/group__filters.html. [Acessed on 24 June 2015].

[59] Point Cloud Library (PCL), “pcl::EuclideanClusterExtraction< PointT > Class Template Reference,” [Online]. Available: http://docs.pointclouds.org/1.7.1/classpcl_1_1_euclidean_cluster_extraction.html#details. [Acessed on 24 June 2015].

[60] ROS, “ROS Introduction,” [Online]. Available: http://wiki.ros.org/ROS/Introduction. [Acessed on 24 June 2015].

[61] ROS, “rqt_reconfigure,” [Online]. Available: http://wiki.ros.org/rqt_reconfigure. [Acedido em 18 June 2015].

[62] G. Law, “Quantitative Comparison of Flood Fill and Modified Flood Fill Algorithms,” *International Journal of Computer Theory and Engineering*, vol. 5, nº 3, June 2013.

[63] Universal Robots, “UR5 robots | Automate tasks up to 5 kgs,” [Online]. Available: http://www.universal-robots.com/en/products/ur5-robot/. [Acessed on 21 June 2015].

[64] Universal Robots, “UR5 Technical specifications,” [Online]. Available: http://www.universal-robots.com/media/50588/ur5_en.pdf. [Acessed on 21 June 2015].

[65] S. Edwards, S. Glaser, K. Hawkins, W. Meeussen e F. Messmer, “universal_robot,” [Online]. Available: http://wiki.ros.org/universal_robot. [Acessed on 21 June 2015].

[66] ROS, “Clock,” [Online]. Available: http://mirror.umd.edu/roswiki/Clock.html. [Acessed on 24 June 2015].

[67] ROS, “octomap_server,” [Online]. Available: http://wiki.ros.org/octomap_server. [Acessed on 24 June 2015].

[68] J. Davis e M. Goadrich, “The Relationship Between Precision-Recall and ROC Curves”.

[70] Point Cloud Library (PCL), “The PCD (Point Cloud Data) file format,” 2 June 2015. [Online]. Available: http://pointclouds.org/documentation/tutorials/pcd_file_format.php.

[71] T. Ono, Toyota production system: beyond large-scale production., Productivity press, 1988.

[72] Point Cloud Library (PCL), [Online].

Annex A

Table 10 - Engine pipe characteristics.

Variant	PSA Reference	Container	Weight	Units / Container	Average Daily Needs
P1	9684566480	06432	0,932 kg	5	66
P2	9686423080	06432	1,11 kg	4	33
P3	9673633180	06432	0,65 KG	5	26

Table 11 - Engine support characteristics.

Variant	PSA Reference	Container	Weight	Units / Container	Average Daily Needs
Su1	9672950980	04322	0,385 kg	22	35
Su2	9657457980	06422	1,437 kg	4	33
Su3	9674030280	06422	0,729 kg	12	103
Su4	9684613880	BNA20	0,781 kg	14	35
Su5	9675508280	06422	0,773 kg	12	103
Su6	9682776880	04322	1,5 kg	5	40
Su7	V758078180	ANC20	0,265 kg	34	40

Table 12 - Thermal shield characteristics.

Variant	PSA Reference	Container	Weight	Units / Container	Average Daily Needs
T1	9804088180	06422	0,395 kg	20	139
T2	V759560680	06432	0,748 kg	12	37
T3	V759560580	06432	0,55 kg	12	37

Table 13 - Alternator characteristics.

Variant	PSA Reference	Container	Weight	Units / Container	Average Daily Needs
A1	9803750980	00081	7,12 kg	3 layers of 20	324
A2	9666997980	00081	6,32 kg	3 layers of 20	60
A3	9678048880	00081	7,05 kg	3 layers of 20	216
A4	9678730980	00081	7,2 kg	3 layers of 20	10
A5	9674646180	63396	8,8 kg	3 layers of 20	180
A6	9675753680	63396	8,72 kg	3 layers of 20	12
A7	9809046080	63396	8,83 kg	3 layers of 20	24
A8	V757848880	CCF75	6,2 kg	4 layers of 20	24

Table 14 - Starter characteristics.

Variant	PSA Reference	Container	Weight	Units / Container	Average Daily Needs
St1	9654595680	00081	4,09 kg	4 layers of 21	185
St2	9646694080	00081	3,69 kg	4 layers of 24	39
St3	9647157980	00081	4,02 kg	4 layers of 21	165
St4	9670983080	CAF75	3,25 kg	5 layers of 30	270
St5	9664016880	CCF60	3,13 kg	4 layers of 35	224
St6	V764559080	CCF60	2,8 kg	4 layers of 42	68

Table 15 - Air conditioning compressor characteristics.

Variant	PSA Reference	Container	Weight	Units / Container	Average Daily Needs
C1	9677824580	16369	5,1 kg	4 layers of 20	32
C2	9678038980	16369	5,8 kg	4 layers of 20	16
C3	9671451180	17306	5,24 kg	4 layers of 20	288
C4	9800840380	17306	5,9 kg	4 layers of 20	256
C5	9678656080	17306	5,33 kg	4 layers of 20	336

Annex B

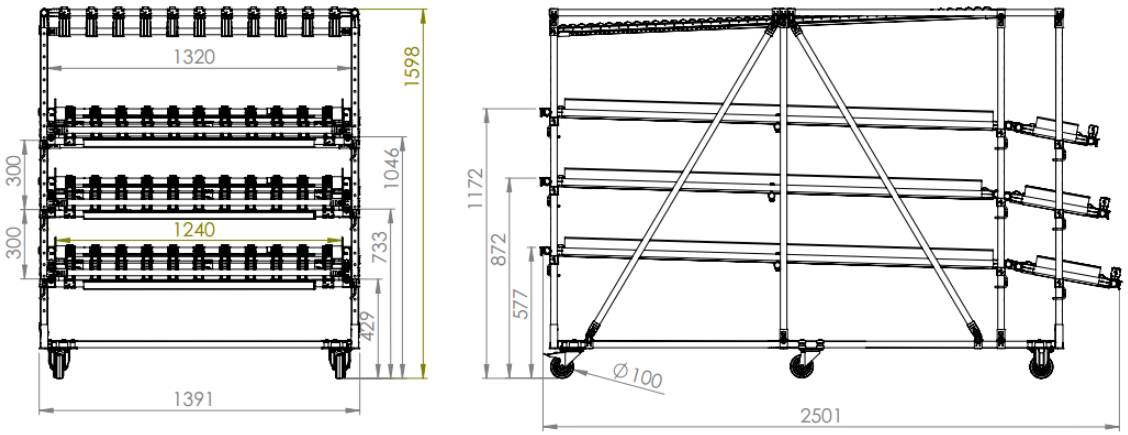


Figure 47 - Rack type 1.

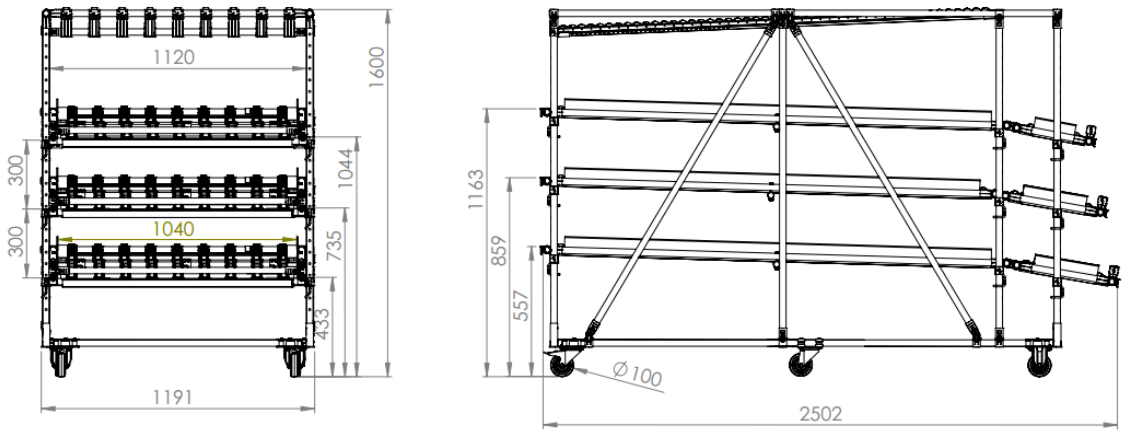


Figure 48 - Rack type 3.

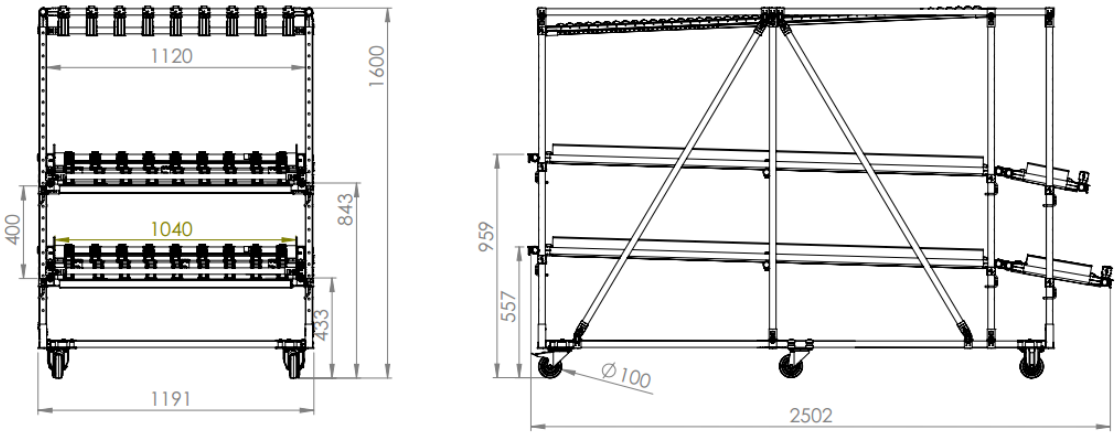


Figure 49 - Rack type 5.

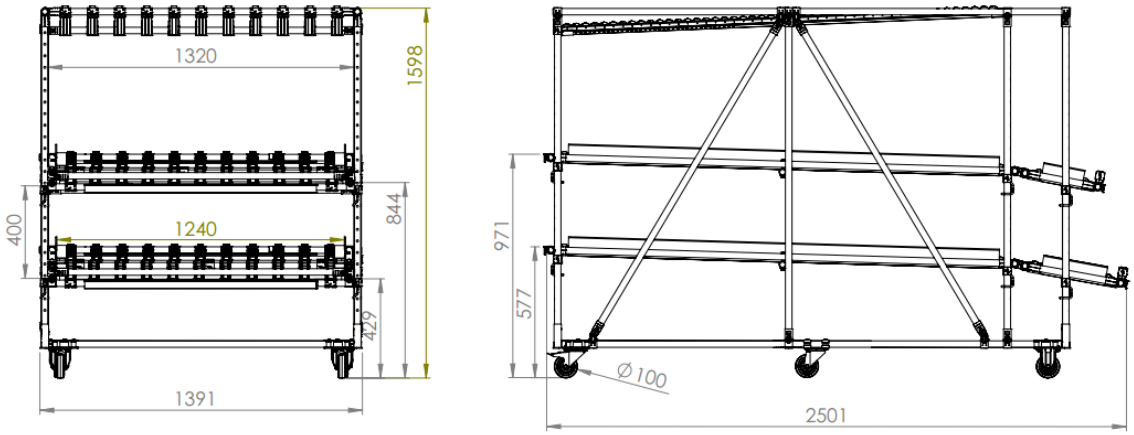


Figure 50 - Rack type 6.