Praktikumsbericht

Development of an Autonomous Mobile Robot

Valentin Niewada

DLR

Institut für Robotik und Mechatronik	BJ.: 2015
	IB.Nr.: 572-2015/24

Freigabe:	Die Bearbeiter:	Unterschriften
	Valentin Niewada	_____
	Sebastian Brunner	_____
		_____
	Der Abteilungsleiter	_____
	Der Institusdirektor	_____

Dieser Bericht enthält 60 Blatt, davon 55 Diagramme

Ort: Oberpfaffenhofen	Datum:	Bearbeiter:	Zeichen:

NIEWADA Valentin
Promotion 2015
University year 2014-2015

Master's Degree in Sciences "Imaging, Robotics and Sciences for the Living"
Robotics and Automation

Final Year Internship Report

« Development of an autonomous mobile robot »

DLR Oberpfaffenhofen
Münchener Straße 20
82234 Weßling
Germany

Internship tutor :
Sebastian BRUNNER
Sebastian.Brunner@dlr.de
03/02/2015 – 08/31/2015

Summary

I. Schemes Table

II. Acknowledgments

First of all, I would like to thank Dr. Michael SUPPA and Dr. Alin ALBU-SCHAFFER for allowing me to work and learn in their institute during those six months.

Then I would like to especially thank Sebastian BRUNNER, Peter LEHNER and Andreas DOMEL for introducing me to the DLR EUROC Team and for their support which made this internship a great professional experience.

To finish, I would like to thank my teammates for their good mood and support which provided to my internship a great work environment.

III. Introduction

Although the robotics community did a lot of research in the field of autonomous mobile robotics, there are still many unsolved challenges. With this dynamic, the **European Robotics Challenges** (**EUROC**) aim at enhancing mobile robotics research by building concrete projects with industrial applications.

During my final year internship for the ***Master's Degree in Sciences "Imaging, Robotics and Sciences for the Living"*** delivered by the **Strasbourg University** which has taken place in the **Robotics and Mechatronics Institute** at the **DLR Oberpfaffenhofen** (Germany), I had the opportunity to actively participate to the making of one of those challenges. In fact, the EUROC DLR team in currently developing, in association with KUKA, the autonomous mobile robot **"Miiwa"** which will be a work support for in lab and on field experiments for the challengers.

My teammates and I goal during this internship was to implement, to test and to validate a scenario using all the parts of the robot in order to be able to assist the challenger teams during the current EUROC stage. I was in charge of the "environment modeling" topic.

This report constitutes a detailed overview of my work from simulation to real system integration as well as the situation of the EUROC and its link to open sourced libraries and software into the autonomous mobile robotics and environment modeling research.

IV. Glossary

Those abbreviations and acronyms will be often used toward this report:

- **DLR**: Deutsches Zentrum für Luft- und Raumfahrt (German Aerospace Center)
- **DOF**: Degrees of Freedom
- **EUROC**: European Robotics Challenges
- **PCL**: Point Cloud Library (C++ library)
- **RGB**: Red Blue Green, to designate color cameras
- **RGB-D** (sensor): for "Red Green Blue Depth", said for a vision sensor which can provide colour and depth images
- **ROS**: Robot Operating System (programming software)
- **TCP** : Tool Center Point
- **Voxel** : Cubic subdivision of volume

1. The project's environment

1.1 EUROC

1.1.1 A European contest

The **EUROC** (for **EUropean RObotics Challenges**) have been created and are supported by a consortium of European institutes and companies working in the field of robotics. Officialised by the **European Union**, it aims at bringing new dynamics and increase competitiveness in the European manufacturing industry by developing new state of the art robotics applications. Those applications are based of autonomous systems and human-robot cooperation. Opened to research teams all over Europe, those challenges are based on three categories all of them supported by well-known companies which bring support and technologies. The three challenges are:

- Challenge 1: **Reconfigurable Interactive Manufacturing Cell (RIMC)**. Based on the human-robot cooperation, this challenge proposes to use one (or more) robotics arm(s) with customizable manipulator to solve an assembly task with teamwork.
- Challenge 2: **Shop Floor Logistics and Manipulation (SFLM)**. Here the goal is to realize an autonomous pick and place task in a partially unknown environment with a mobile robot. This challenge is also focused on security for both humans and machines.
- Challenge 3: **Plant Servicing and Inspection (PSI)**. With a 6 rotors drone, the challengers will have to inspect hazardous and hardly accessible parts of a factory plant then send useful data to the operator who will do a diagnostic.

Launched in 2014 first semester, the **EUROC** timeline is divided into three stages. At the end of those stages, a selection will occur and some teams will be eliminated. The three stages are:

- Stage 1: **Simulation Contest (4 months)**. Teams have to solve a set of tasks on a dedicated simulator. The simulator is different for each challenge and has a similar behaviour than the real system.
- Stage 2: **Benchmarking, free-style and showcase (15 months, 5 teams)**. Teams have the opportunity to develop on the real system associated to the challenge.
- Stage 3: **Pilot Experiments (9 months, 2 teams).** Teams will experiment their solutions on the field and be welcomed by a support company.

We have entered into **Stage 2** and the number of remaining teams has decreased from 102 to only 15. Each challenge will bring a finalist team which will have to compete against the two others to win the **EUROC**. But this competition is also created to detect talented researchers and engineers all over Europe and gather ideas to think the manufacturing industry for the years to come.

Figure 1 - EUROC logo

1.1.2 The DLR's involvement in the EUROC

In this major project, the **DLR** has teamed up with the German company **KUKA** to support the second challenge: **"Shop Floor Logistics and Manipulation"**. In fact, KUKA has already conducted researches on autonomous mobile robotics with its range **"Omnirob"**. As a consequence, the newly developed **"Miiwa"** will be used as target system for **Challenge 2**. This robot is designed to perform autonomous pick and place actions in unknown environment. Its shape and dimensions make it capable to work in an environment first designed for humans. It is equipped with a 7 DOF arm and two pairs of RGB cameras.

Notice: A more detail description of the Miiwa will be given starting page 17

KUKA has developed the electrical and mechanical hardware (except for the pan tilt mast) as the same time as drivers for the robot's base movements and the ultrasounds sensors as well as the system integrator. The DLR has added the cameras, the gripper and the pan tilt's support and was in charge of developing the software in order to make the robot **ROS-compatible** (see **Software development tool** page 15) to allow challenger teams to work on the very same software without integration issues. This preparation included to develop a set of functions to have access to all the system's sensors and actuators but also to establish a SSH connection for remote control. As a consequence, the number of computers in the robots passed from two to four which required some hardware customization. Furthermore, the DLR has developed the simulator related to **Stage 1**.

In Weßling, the EUROC development team is divided in **two groups**. The first one is composed with **titular researchers** and works on the system preparation (hardware and software) while an international student team develops a test scenario like a challenger team.

The **DLR Oberpfaffenhofen** has also welcomed for one week each of the selected teams for **Stage 2** during the months of July and August 2015 and will continue to receive them occasionally until the end of this Stage (first semester 2017). This first visit allowed the teams to receive information about the **Miiwa** use delivered as well by KUKA as by the titular team. Challengers also performed measurements and set ups in order to be able to work by themselves before the next code camp session.

As a consequence, the most part of this internship is dedicated to the development and the integration of a scenario which will test every part of the **Miiwa** to assure its functionalities before the next workshops.

1.2 Work overview

In this chapter, I will introduce the tasks I was asked to accomplish. The main goal of the team was to develop a scenario in which every part of the system will be tested, so we agreed on a "pick and place" scenario with unknown obstacles avoidance. Here are the main requirements of this scenario:

- **Pick and place objects**
  - Deal with objects with unknown color and shape
  - Pick the objects with the right torque
  - Recognize shape and color
- **Be able to move in space without collision**
  - Use path planning to grab and drop object in know and dedicated target zones
  - Map the environment

Environment modeling composes with object recognition the **"Vision"** part of the development. It aims at mapping the close environment of the robot to detect a set of obstacles which will have to be avoided during path planning.

It uses point clouds generated by the two pairs of cameras and creates octrees to model the visible obstacle, also keeps track of the previous ones in their known positions. More theoretical details and a description of the realization, pre and post processing of those octrees will be given in **Octree Building** page 13 and **Realisation on simulator** page 29.

The first part of the development was done on simulator. After some efficient results, a switch to the real system was performed with the same kind of scenario.

2. Introduction to environment modelling

3D modeling approximates the shapes of objects or globally of a seeable environment. The source of this process can be a set of images or a point cloud. It is used for special representation task like physics simulation (for example in the medical field for organs or bones simulations), video games, and 3D printers or for **environment modeling**. In this context, it will be an input of path planning for a robotic platform.

We must make the distinction between two kinds of 3D modeling. One approach only represents **surfaces** (or shell) of the targeted object or environment. This method is used when volume information is not necessary as in video game environments. Moreover it is easier to work with shells when rendering effects (like texturing or light effects) are computed. The other approach can work with **volume representation** which allows building solid environment. This approach is used in our work. In fact, the robot must accomplish its tasks in a real 3D environment. As a consequence, this constraint requires knowing the shape and size of the objects which compose it from every point of view to be able to plan a safe trajectory.

The model can be **polygonal**, which means that the objects are approximated by a set of polygons with or without the same shape and size. An inconvenient will be that the curves will not be smooth. This effect can be countered using **curve modeling**. In fact, curve modeling assigns each point on a surface a weight which will pull or push the curve trying to recreate this surface. Beyond that, new techniques in software processing allow engineers but also computer artists to directly **sculpt a 3D object**, create complex shapes and even stick picture or textures in it to always be closer the realism.

Recently, 3D modeling became more accessible to the general public with the apparition of sensors like **Microsoft Kinect** or **ASUS Xtion** and their dedicated software. As a consequence, the ROS community started to develop dedicated libraries and integrate environment modeling in bigger projects. With this dynamic, the Octomap (**[Hornung13]**) library was release in 2013 and provides a useful set of tools for octree building.

3D modeling is used in this project to **create a map** of the robot's close environment. As a consequence, the position of obstacles will be known in a global frame. We will be able to correctly use path planning algorithms and avoid collisions. Knowing the fact that the Miiwa will be used in industrial plan and will work with humans nearby, security must be a priority during robot's runs.

The octree generation requires pre-processing in order to only keep the important information from the cameras. In terms of software, the **Octomap** library will be used. For more details about the realization in simulation, see "**Realisation on simulator**" page 29 or for the real system integration, see "**Realization on real system**" page 41.

3. Octree Building

Introduced by Meagher in 1981 (**[Meagher81]**), the octree concept is a powerful way to model unknown 3D objects. It is based on the object geometry but is not perturbed by holes or even convex or concave shapes which makes is the main difference to constructive solid geometry method. Moreover, an octree is not based on colour or texture. We will use this method to create a map of the robot's environment and keep track on obstacles. In fact, the octree approximates the environment and returns a volume and location information useful for 3D path planning.

Octrees main feature is the approximation of the object's shape by a **set of voxels** containing binary information: "**full**" or "**empty**". Those voxels can be divided **into eight children** (Fig. 2) who will carry the same kind of data but with a higher resolution.

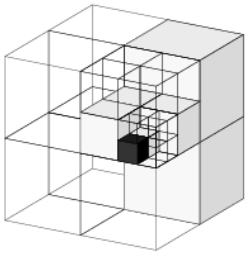


Figure 2 - Octree organisation ([Hornung13])

As a consequence, data are stored in a **hierarchical tree** structure where each entry leads to eight others. The size of the cubes will so decrease exponentially until they reach the desired (or the maximum possible) resolution (Fig. 3), the one which will allow us to get a model. These smaller cubes will be called "**terminal nodes**" or "**leafs**".

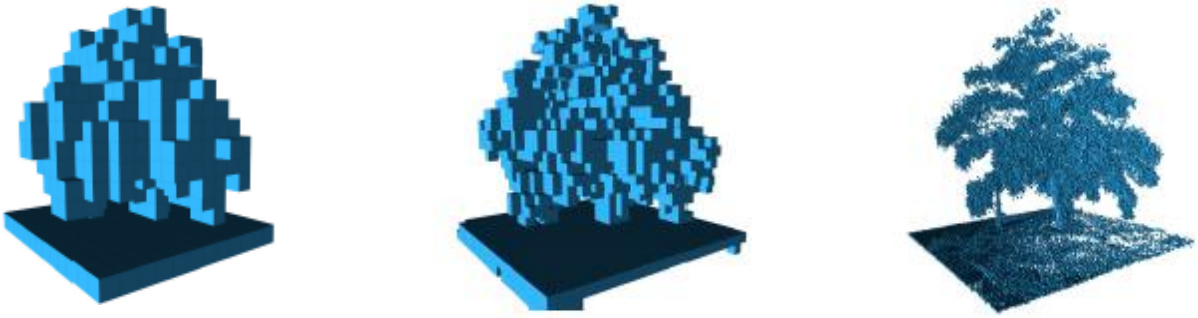


Figure 3 - Increasing the octree resolution ([Hornung13])

We can distinguish **four major octree behaviours**. First of all, octrees can be (**[Hornung13]**) used without **entering any size parameters** to characterize the environment. In fact, as each voxel may be a child node of a bigger one, we are always capable of increasing the tree. We must also get a **full 3D model**; consequently empty voxels are not forgotten and constitute a part of the data set. During the application and as we will use vision sensors, the map will be **updatable**. Finally, octrees will have to be **compact** in order to save computer resources.

The two states of a node simplify the decision if it is empty of full. But as sensors can be subjects to **aberration** and **noise**, we cannot allow the process to create a cube, and so in your case detect an obstacle, each time a portion of the environment seems to be full. This is even more important if we want to work with small leaf resolution. As a consequence, a **probabilistic approach** is used (**[Moravec85]**). The 2D occupancy map method can be used for every octree depth level to build the model. Here the *logOdds* of the probability $P(n)$ of a node n to be occupied at the time t knowing the previous measures

$z_{1:t}$ is given by:

$$L(n|z_{1:t}) = L(n|z_{1:t-1}) + L(n|z_t) \quad (3.1)$$

Where:

$$L(P(n)) = \log\left(\frac{P(n)}{1 - P(n)}\right) \quad (3.2)$$

Notice: the logOdds is used for more user-friendly notations.

As we can see, we passed from an occupancy probability **of 0 or 1 exclusively** to a value in the **segment** $[0; 1]$ (see annex **Details on LogOdds function** page 50) which represents the sum of the previous measures and the current one. We need now to give a value to $P(n)$. Three constants are defined:

- At $t = 0$, the absence of measure gives $P(n) = 0.5$
- If the sensor provides enough local data to fill a leaf, we have $P(n) = hit$
- In the contrary to our previous affirmation, we have $P(n) = miss$

The values hit and $miss$ are defined **by the user**. They are linked to the **trust** we have in the sensor, in fact if we increase the gap between them then the received data will have a **bigger impact** on detecting obstacles. We can also note than the condition $(0 < miss < 0.5 \ and \ 0.5 < hit < 1)$ must be true on the one hand because of the log function definition and also to avoid a convergence to a fully occupied or empty environment.

In the case of a moving obstacle, it will have an influence on the octree leading to change the voxels state until they stick to its new localization. But with the constant $P(n)$ values we defined earlier, we will need the **same amount of measures** to make a cube empty that we required making it full. This aspect may reduce the processing time and if the obstacle is moving fast, we may keep unnecessary full cubes in our map. This is called "**overconfidence**". Consequently, an update policy is needed (**[Yguel07]**).

We will introduce two new user customizable constants: l_{max} and l_{min} which will define respectively the upper and lower probability bound required to declare a cube full or empty. The definition of $L(n|z_{1:t})$ is:

$$L(n|z_{1:t}) = \max(\min(L(n|z_{1:t-1}) + L(n|z_t), l_{max}),\ l_{min}) \quad (3.3)$$

This new formula is keeping $L(n|z_{1:t})$ between l_{max} and l_{min} **limiting the confidence** we give to the map. Moreover, we can see that reducing the gap provides more frequent updates. The state of a voxel will be given as soon as one of the two bounds is reached. Knowing the system transformation and the distance between the sensor and the obstacle (here given by a depth image) we can situate leafs in the 3D global frame and reconstitute the obstacles.

By this way of thought, we understood how an octree is built in theory. From a practical point of view, the **Octomap** library (**[Hornung13]**) is chosen because of its accessibility and its open source aspect to dynamically map the environment (see **Realisation on simulator** page 29).

4. Software development tool

For the software development, both on simulator and on the **Miiwa**, the software middleware **ROS** (**Robot Operating System**) was chosen. Released in 2007 by the **Stanford Artificial Intelligence Laboratory**, ROS is now well-known by the robotics community. In fact, its open-source (Unix based) aspect allows engineers and researchers to provide new libraries and tools for the community. ROS is downloadable with a set of useful basic libraries and software like **RViz Visualizer**. Choosing ROS provides also the guarantee for every challenger team to work with the same open-source software and so provides equality in this competition. As we speak, ROS has been used for famous robots like **Aldebaran's NAO of Willow Garage's PR2**. Here we will introduce the basic ROS terms which will be used along this report.

Figure 4 - ROS Logo

ROS is also based on cooperation and teamwork. The modules, called **"nodes"** composing the on-going software can communicate and share information with no regard on their language (which can be **C++** and **Python**), so developers have certain flexibility. Furthermore, nodes can be grouped in **packages**. As a consequence, every part of a system can be independently represented clarifying a lot the whole project organisation. ROS also provide a set of tools and terminal command line to focus on a particular node, viewing its outputs, its connections or if it is well executed.

Nodes have two major features to set up the links and inter-nodes communication. Depending on the developer choices, a node can:

- **"Subscribe"** to another node with a background process always receiving its output and waiting is there is not
- **"Publish"** information continuously

Consequently, a subscriber has to listen to a publisher. More than one subscribers or publishers can be declared in the same node but they always carry only one information, chains can so be made:

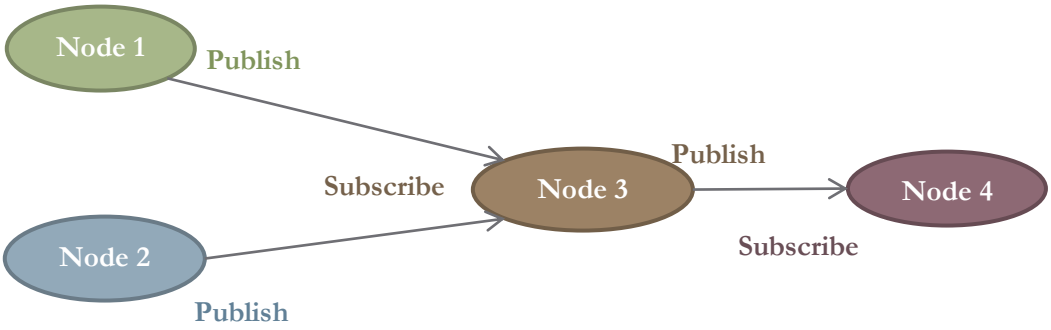


Figure 5 - Basic ROS communication

We can also make the distinction between topics and **"services"**. In fact, a service is a node which includes a particular function used occasionally ("**remote procedure calls**"); a service is so called each time we need it and do not returns information otherwise. A **service** can have multiple and various inputs and outputs. However, the **service** function has to be of the type **boolean** to return a value representing its well execution.

Both publishers and subscribers work with **"messages"**. Messages represent various types of data. All based on basics variable types (integer, float, string…) messages can be combined to form structures

with their own purposes and be declared as object in C++ and Python. For example, a set of three float can form a 3D point, moreover a set of 3D point forms a point cloud. The most common messages are included in libraries installed with ROS but custom messages can also be made.

In a nutshell, a ROS-based system makes a development more flexible and easy to organise. Our work space has been divided in four packages for each major part of the scenario: **Object Recognition**, **Environment Modelling**, **Path Planning** and a package called "**Main**" including the starter executable, general nodes and the state machine. To learn ROS to an advanced level was the first requested task we has to accomplished and took most of the first month.

The **Stage 1** simulator was realized by the DLR EUROC team with the open source software **Gazebo**. It allows creating our own simulation environment by providing many features like 3D models, kinematics, dynamics, collision detection and sensors. Consequently, **Gazebo** was chosen to build multiple variants of the requested task of **Stage 1**, and those very simulations will serve as dummies before integrating the scenario into the real system (Fig 6 and 7).

The simulator contains a model of the **KUKA 7 DOF arm** and the **Scene cameras mast** (with real dimensions) on **a 2 meters square table**. The two pairs of cameras are capable to provide RGB images and point clouds. Furthermore, the base of the arm can move on the plane represented by the table, which is the only difference regarding to the Miiwa. To sum up, every part of the real system is testable with it. Developments first results will be shown in the simulator.

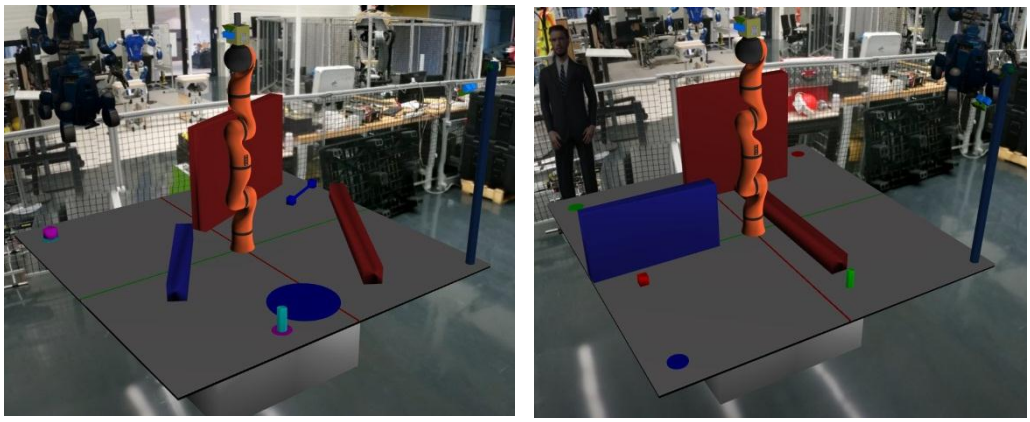


Figure 7 - Two variants of pick and place with obstacles

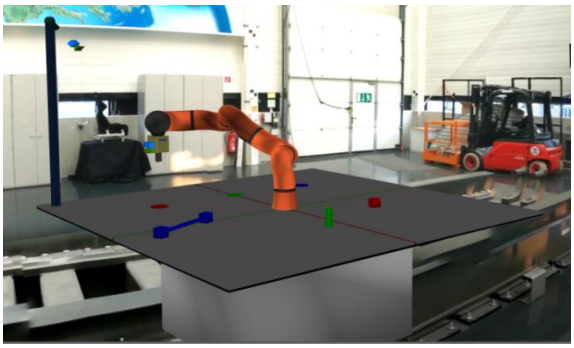


Figure 6 - Pick and place without obstacle

5. Miiwa

The **KUKA Miiwa** is the new KUKA's autonomous mobile robot. Its applications are dedicated to an **industrial environment** and some of its specifications may be **customized**. Its personalization makes it a great candidate for the EUROC challenges. Here it is customized for and by the DLR, is composed of three major parts:

- The **base** is mounted on four **mecanum wheels** (see "**Mecanum wheels**" page 19). Its height originally of 70 cm can be customized; here the system is 96 cm tall. At the top of it we have got desktop for storing and manipulating objects. Its shape is a pseudo rectangle of 0.61x1.8 m^2:

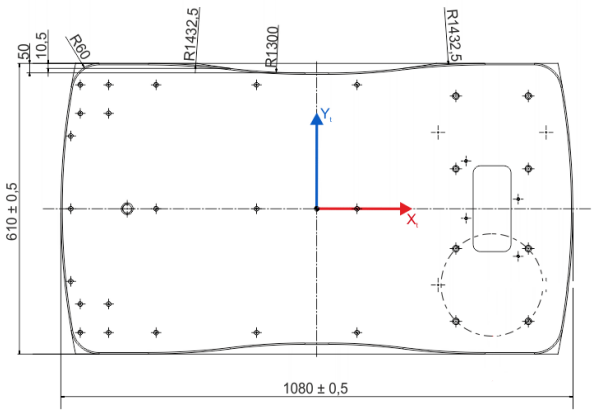


Figure 8 - Miiwa desktop scheme (KUKA)

- A **7 DOF KUKA arm** can be set up optionally and can be of two types:

	Payload	Reach	Repeatability
LRB IIWA 14 R800	7 kg	800 mm	±0.1 mm
LRB IIWA 14 R820*	14 kg	820 mm	±0.1 mm

** Our system will use this reference*

- Two pairs of **RGB cameras** have been installed by the EUROC DLR team to provide images and point clouds. One of them is said **"eye in hand"** and fixed to the arm's end effector: a **SCHUNK 1 DOF gripper**. The other is independent and mounted on a **2 DOF pan tilt** mast on the base table.

While moving, the base can act toward three distinct mode if there is or not obstacles in the working area: **Normal**, **Reduced Speed**, or **Positioning**. More details will be given page 20. Those modes and the sensors data are managed by four integrated computers, two original and two added by the DLR. The purposes and names of the computers are:

- **DLR Miiwa sensors**: for controlling the base and the arm as well as gain access to all sensors and actuators
- **DLR Miiwa server**: sets up the ROS core, the nodes and the services
- **KUKA Control Computer**: access all hardware components and provide control command for all joints and effectors
- **Navigation PC**: for global navigation and localisation

The following picture represents the final hardware we will use for the development (more pictures can be found in annex **Miiwa's gripper and pan tilt** page 49):

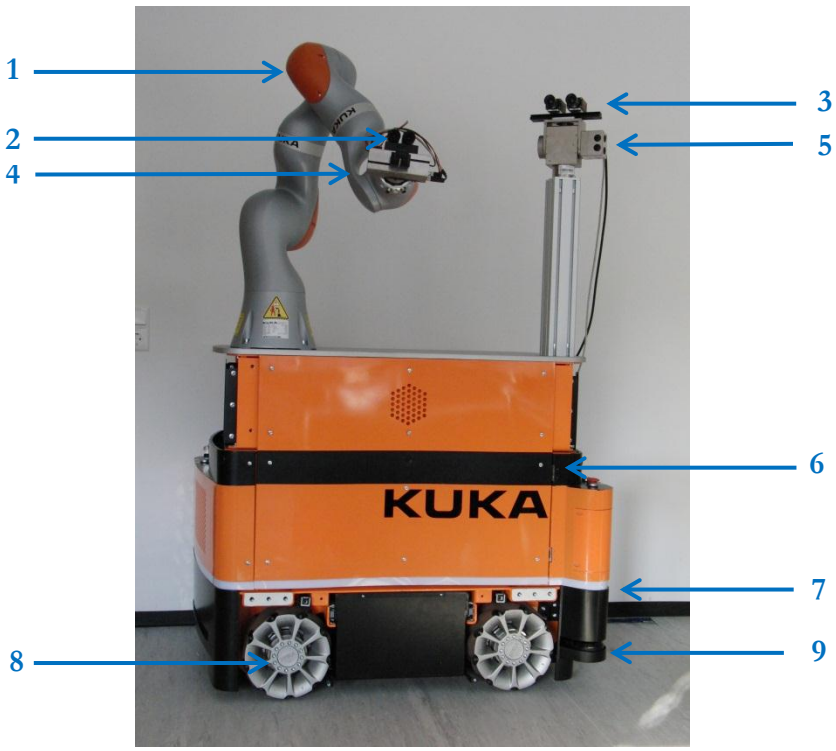


Figure 9 - KUKA Miiwa

Main parts of the robot:

1. 7 DOF arm (KUKA)
2. 2 RGB cameras designated as "Hand", resolution 1624x1234 (Allied Vision Technology)
3. 2 RGB cameras designated as "Scene", resolution 1624x1234 (Allied Vision Technology)
4. Gripper (SCHUNK)
5. 2 DOF pan/tilt base (SCHUNK)
6. Ultrasound sensors (x8)
7. Colored LED belt
8. 4 mecanum and independent wheels
9. Laser scanners (x2)

The robot is first thought to be **secure**. On the one hand, the two pairs of cameras allow using two points of view and may be useful for environment modeling and obstacles avoidance. On the other hand, the colored **LED belt** provides visual information to close by humans, especially when they enter to the danger zone scanned by **lasers** as well as **ultrasounds sensors**. We use both lasers and ultrasounds to process both 2D and 3D scans at two levels.

This system is also **adapted to industrial working environment**. In fact, the adjustable height of the table makes it suitable for many work areas and may be in a human reachable one. In the same aspect, the payload of its arm has been chosen to be able to pick and place objects until 14 kg which is appropriate for a replenishment tasks which involves metal pieces.

In the following chapters, we will give more details about Miiwa's hardware characteristics.

5.1 Technical data

Here are some useful **Miiwa** technical data:

Designation	Value
Height (until table)	0.7 m (minimum) 0.96 m (DLR custom)
Weight	500 kg
Max speed	0.83 $m.s^{-1}$
Max breaking distance	0.51 m
Autonomy	8 h
Numbers of motors	4
Nominal drive power per wheel	192 W
Peak drive power per wheel	576 W

5.2 Conception details

5.2.1 Mecanum wheels

The choice of four **mecanum wheels** for the Miiwa's is based on the wish to move freely, limit blockings in narrow spaces and execute tasks without having do a repositioning of the robot. In fact, this kind of hardware allows the base to move in every direction without changing its orientation but also rotate on itself. It is handy when we want to simplify the robot's movements in factories where travelling areas have well defined boundaries and where U-turn is prohibited. Here we can note that the robot **does not have steering mechanism** and its direction is only given by the **rotation of the wheel relatively** from one to another.

This independence **facilitates maintenance** as well as the separated rollers which can be replaced one by one.

Here is a 3D drawn of one of the mecanum wheel use on the Miiwa:

Figure 10 - A Miiwa's mecanum wheel

5.2.2 Lasers and ultrasound sensors

Two laser scanners and eight ultrasound sensors are set up around the Miiwa. Both create a fence on two levels around the robot in automatic mode and so prevent it to move if a moving obstacle appears to be too close. We can distinguish two reactions areas where an obstacle can enter: on the one hand the warning area is wider and if penetrated switches the system to **"reduced speed"** mode, on the other hand the protected area will stop it and will wait for the obstacles to disappear. While accomplishing tasks on automatic mode, the base can also enter into **"positioning"** mode.

Here is the data differentiating the three modes:

	Max speed	Stopping distance	Protected area radius*	Warning area radius*
Normal Mode	0.83 $m.s^{-1}$	0.51 m	0.76 m	0.56 m
Reduced speed mode	0.28 $m.s^{-1}$	0.1 m	0.35 m	0.97 m
Positioning mode	0.1 $m.s^{-1}$	0.03 m	0.28 m	1.04 m

**From the base centre*

If the basic use of lasers and ultrasound sensors are the same, both are still necessary here. In fact, the lasers sensors are known as more reliable than the ultrasound ones. Furthermore they can be used for laser-based positioning. The ultrasound sensors allow creating a 3D fence and are a second security in complex environments.

5.2.3 LED belt

In order to simply communication with humans, a **LED belt** displays **coloured light information** regarding to its current state. For examples we can note:

- A **white l**ocated permanent light indicates the moving direction of the robot
- A flashing or permanent **green** light indicates the charging progression (flashing for "on going")
- A **blue** light indicates that an obstacle is located in the robot's warning area, switching it into "Reduced speed" mode.
- A **red** light indicates that an obstacle is located in the robot's protected area or if an error has occurred, stopping the current task.

5.2.4 DLR's hardware customizations

The vision hardware as well as the gripper has been added by the DLR EUROC team in order to stick to the Stage 1 simulator. We have got **two pairs of RGB cameras** that give us two points of view. The use of two side by side cameras provides also point clouds using Semi Global Matching algorithm (see "**Point clouds** " page 32).

Figure 11 - Used Allied RGB camera

Here are their major specifications:

Designation	Value / Type
Constructor	Allied Vision Technology
Model	Manta G201C
Power supply	Power over Ethernet
Resolution	1624x1234
Sensor type	CCD progressive
Frame rate	14 fps
Global dimensions	74x29x44 (mm)

The first pair of cameras is **"eye in hand"** and will be designated as **"TCP"** since this point. It is attached to a **SCHUNK WSG-50 gripper** which can open its jaw until 110 mm and has a maximum grasping force of 80N.

Figure 12 - Used SCHUNK gripper (without fingers)

The second one is separated and fixed on top on a static mast. The support is a **SCHUNK pan tilt actuator** with a ±120° range in axis 1 and a ±360° range for axis 2:

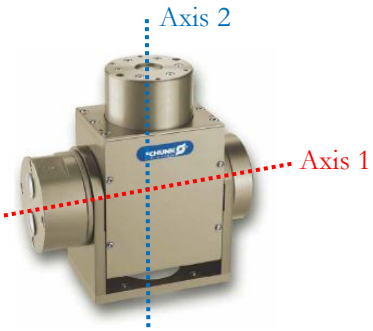


Figure 13 - Used SCHUNK pan tilt actuator

Those three products have been chosen for their high quality and their excellent repeatability (0.04° for the pan tilt for both axes).

6. Enrolment in research

6.1 State of the Art: Autonomous Mobile Robotics

To understand what the **EUROC Challenge 2** brings to autonomous mobile robotics research, we take a look on already existing systems used in laboratories or companies. Here we present three autonomous mobile robots by giving a short introduction and compare their characteristics to our current system. The following systems have been chosen for their purposes which can be compared to **Miiwa** requested tasks. Their development is finished or enough achieved for research goals and so their specifications may not change a lot.

6.1.1 PR2

First of all, one major example of an autonomous mobile robot with manipulator is **Willow Garage's PR2** (Fig. 14). It is equipped with two 4 DOF arms with a 3 DOF wrist attached on them and its RGBD cameras allow it to detect and manipulate various objects for high skilled tasks. Furthermore, PR2's torso can translate to change the robot's maximum height to 1.6 m which makes it adaptable to its environment.

PR2's software is **ROS-based**, but since Willow Garage is deeply involved in the developments of **ROS** and **Open CV** (image processing library), the PR2 is dedicated to research. As its high price (400 000$) makes it not accessible for everybody, the system is also accessible by internet using the Remote Lab (**[Pitzer12]**).

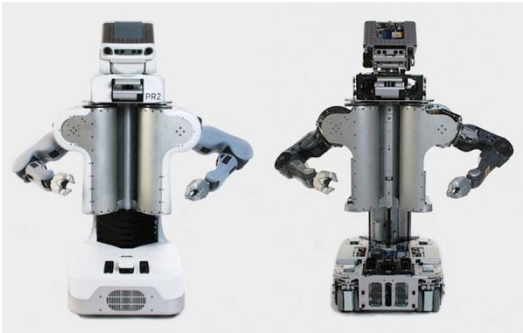


Figure 14 - Willow Garage's PR2

6.1.2 UBR-1

Developed by **Unbounded Robotics**, the **UBR-1** is a direct concurrent to Willow Garage's **PR2**. In fact, the URB-1 is also a **ROS-based** manipulator system using **RGBD** sensors organised in the same shape. But unlike the PR2, URB-1 is smaller, lighter, and only uses **one arm and gripper** (instead of two for the PR2). This makes the robot more agile in indoor environment. It also uses paired to RGBD sensors and a **2D laser scanner** for obstacle avoidance. Due to its size and tools, URB-1 designed for home (or office) applications.

Both robots are designed for lab or personal research and aim at gathering a development community behind them. Released in 2013, its major advantage compared to the PR2 is its price of **40 000$:** ten times cheaper than PR2's. We can see an description of the URB-1 provided by Unbounded Robotics on Figure 15:

Figure 15 - URB-1 description (Unbounded Robotics)

6.1.3 KIVA

We will now switch into a more specific system which is currently used storage facilities and so takes a part on a well knows company development. **Kiva Systems** has been founded in 2003 but is since 2012 a subsidiary on **Amazon**. The third generation of **KIVA robots** is known to be employed in Amazon's storage facilities.

KIVA (Fig. 16) is designed to transport shelves in a working area or bring it to an operator to allow him to pick or place an object on it. So KIVA is thought to save time and energy to Amazon's employees but also to save storage place as the lift is done from under the shelf. On the one hand, its major force comes from is **payload** allowing it to lift **three times its mass** and its possibility to be deployed in **group** and move into complex indoor environment with obstacles avoidance. On the other hand, its applications are limited. KIVA is not ROS-based and not destined to research purposes, so its enhancement depends on Amazon's economic plan.

Figure 16 - A KIVA robot

6.1.4 "Miiwa" innovations

First let's compare the major specifications of all previous systems to the Miiwa:

	PR2	URB-1	KIVA	Miiwa*
Constructor	Willow Garage	Unbounded Robotics	Kiva Systems	KUKA and DLR
Price	400 000$	40 000$	Not available for one unit	300 000€
Height	From 1.33 m to 1.645 m	0.96 m	0.40 m	0.96 m
Manipulator	Two 4 DOF arms with mounted 3 DOF wrist and grippers	One 7 DOF arm	None	One KUKA 7 DOF arm with mounted SCHUNK gripper
Arm length	0.921 m	0.75 m	None	1.823 m
Manipulator payload	1.8 kg	1.5 kg	450 kg	14 kg
Total Weight	200 kg	73 kg	150 kg	500 kg
Moveable base type	Omnidirectional, 4 pairs of steered wheels	Differential drive	Omnidirectional	4 mecanum wheels
Base size and shape	Square (0.66x0.66 m^2)	Circular (diameter: 0.49 m)	Square (0.6x0.6 m^2)	Rectangle (0.61x1.08 m^2)
Max speed	1 $m.s^{-1}$	1 $m.s^{-1}$	1 $m.s^{-1}$	0.83 $m.s^{-1}$
Visual sensors	5-megapixel colour camera Depth camera with wide angle	RGBD camera	None	2 pairs of RGB cameras. One eye in hand, one on a 2 DOF pan/tilt mast
Software	ROS	ROS	Proprietary	ROS

**Current available data in DLR*

First of all, that the **Miiwa** is bigger and heavier than our three others examples, which can limit its movements in small environments. Then its maximum speed is the lowest in every available mode.

But we can ask ourselves: are this size, weight and maximum speed negative points? In fact, the **Miiwa** is designed to work in an **industrial environment**, occupied and alive and as a consequence **close to humans**. In that case, security is the most important aspect of its conception so a high maximum speed may be useless because it is never reached. Moreover, the two pairs of cameras provide two points of view to avoid collisions.

To work close to human also means to adapt to their world. The **adaptable height** of the robot (arm excepted) is thought to allow humans to interact with objects on it in the most practical way. Another example of this adaptation is the **maximum arm payload** ten times higher than PR2's and URB-1's which by their design solve similar tasks. This payload is more adapted to working area replenishment.

In a nutshell, the innovation of the **Miiwa** is its design and characteristics thought for the industry. Furthermore, its ROS-based architecture will allow a simpler maintenance and update of the system in the years to come.

6.2 State of the Art: Environment Modelling

Over the past few years, research on environment modelling has been trying to find new approaches based on well-known techniques such as octree building. On the one hand, these new approaches generally aim at reducing calculation time, fit better to the real environment or use the model for path planning tasks. On the other hand, researchers try to find ways to adapt environment modeling to more complex and realistic areas outside labs which are closer to daily life applications.

In this section we will resume a selection of recent papers talking about environment modelling optimisation and applications in robotics. Then we will situation our project in the state of art and discuss about its innovations.

6.2.1 Octree based environment modeling

As we saw in section 3 "**Octree Building**", octree are cubic divisions of the environment based on occupancy. But in 2012, Ouyang and Zhang have proposed an octree building **based of spheres** (**[Ouyang12]**) aiming at proposing an adjustable balance between calculation time and collision threshold. Units of octrees are here spheres (Fig. 17) but respect the same organization as describe in section 3.

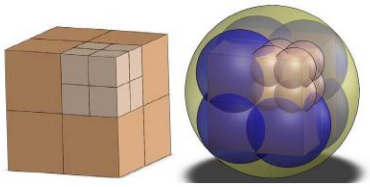


Figure 17 - Spherical octree ([Ouyang12])

The spheres dimensions are based on the cubes (Fig. 18). Here we pass from a binary "collision-no collision" system to a **three level based** checking. If we define a_1 and a_2 the size the two cubes from two objects and d_i the distance between the two centroids of those cubes, i aiming at the current decomposition level, we have got **three cases**:

- If $d_i > \sqrt{3} * \frac{(a_1+a_2)}{2^i}$ (6.1):
  - No collision
- If $\frac{(a_1+a_2)}{2^i} < d_i < \sqrt{3} * \frac{(a_1+a_2)}{2^i}$ (6.2) :
  - We need to check at the next lower octree level
- If $d_i \leq \frac{(a_1+a_2)}{2^i}$ (6.3):
  - A collision in detected

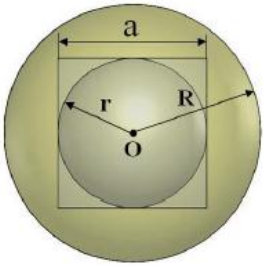


Figure 18 - Sphere construction ([Ouyang12])

So the number of operation to detect a collision may be **decreased** and the smooth surface may also be **better fitted** while keeping the octree advantages.

A mobile robotics related approach may be found in **[Jessup14].** In this paper, octrees are used by two autonomous mobile robots using **SLAM** (**Simultaneous Localization and Mapping**) to map a complex environment, but the originality of this approach is the use of algorithm to merge the octrees generated by both robots in order to create a **single common map**.

Using the voxels *logOdds* (see "**Octree Building**" page 13, the algorithm is able to determine is a voxels from octree O_1may or may not be merged with octree O_2 transformed into O_1's frame. Let's x_1, x_2 be the centres of the voxels n_1 and n_2 and respectively $n_1 \in O_1$ et $n_2 \in O_2$. We write x_2' for x_2

transformation in O_1 and x_2' is the centroid of the voxel $n_2' \in O_1$. If:

- The $n_2{}'$ occupied zone in O_1 is free, then:

$$L(n_1|z_{1:t}) = L(n_1|z_{1:t-1}) + L(n_2|z_{t-1}) \quad (6.4)$$

- The $n_2{}'$ occupied zone is already in O_1, then if :
  - This zone is at the same level that $n_2{}'$ Then :
    - Re-use of (6.4) with updated $P(n_1)$
  - This zone is at a superior level n_1, comparing to $n_2{}'$ then:
    - n_1 is divided into n_1^i $(1 < i < 8)$, and $L(n_1^i) = L(n_1)$ (Fig. 19)
    - Re-use of (6.4) on each n_1 children :

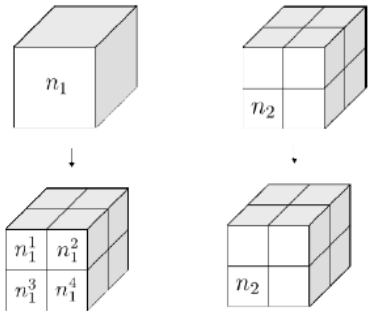


Figure 19 - n_1 division ([Jessup14])

$$L(n_1^i|z_{1:t}) = L(n_1^i|z_{1:t-1}) + L(n_2|z_{t-1}) \quad (6.5)$$

  - This zone is at a lower level $n_1$comparing to $n_2{}'$ then:
    - Same divisions but for $n_2{}'$ (Fig. 20), $L(n_2^{i\prime}) = L(n_2)$ and :

$$L(n_1^i|z_{1:t}) = L(n_1^i|z_{1:t-1}) + L(n_2^{i\prime}|z_{t-1}) \quad (6.6)$$

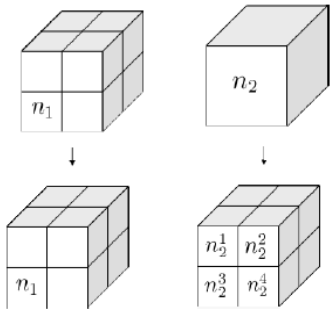


Figure 20 - n_2 division ([Jessup14])

With this approach, researchers are able to **merge both maps** generated by the mobile robots. As a consequence, both of them have knowledge of environment details that they never explored. In terms of applications, we can think about **cooperative exploration** in outdoor or industrial environments and real time updates.

To sum up, octree use optimisation is still an important field of research. In fact, to be able to save calculation or scanning time is important to react quickly if the applications require path planning of is this local environment in subject to often change.

6.2.2 Other recent environment modeling approaches

Path planning for biped robots could be hard to set up especially if the obstacles can be small enough for the robot **to step over them** and so reduce the planned path. With this idea described in **[Cupec05]**, environment modeling can be used to smooth the trajectory of a biped robot. First of all, the robot is defined as a **circle** and the detected obstacles are extended in order to draw an **occupied space** around them (Fig 21), defining in the same times a **free area**:

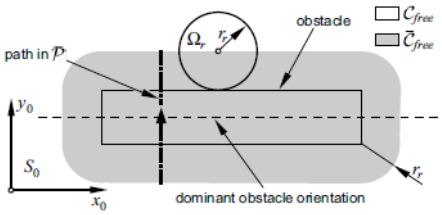


Figure 21 - Free and occupied areas definition ([Cupec05])

But as some obstacles can be passed over because of their small size, some conditions may be defined to classify them. After a conversion from a 3D point cloud to **a 2.5D map**, the authors build security areas and checking points to fulfil those tests (Fig. 22). To resume, an obstacle may be passed over only if the rectangle built from the x_i points and related to the occupied, warning and free areas valid size criteria.

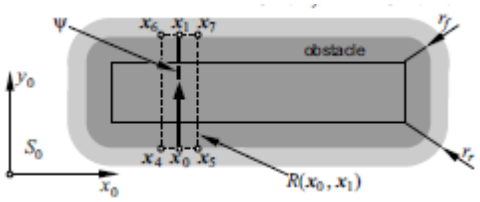


Figure 22 - Checking figures building ([Cupec05])

When experimenting, the authors were able to make a biped robot move between two points by cutting through an area full of small obstacle. But environment modeling can also be used in daily life application like building the map of a multi levels parking deck. That is what achieved the authors of **[Heigele12]** have achieved by concatenating tiles of occupancy grid generated by laser scanners.

In a nutshell, environment modelling applications do not necessary use octrees and can be a useful tool for path planning problems especially if they are linked to complex environments or original robots.

6.2.3 The project's approach

During this project, we will try to develop a complex application for environment modelling destined for the industrial world. Here the building of the map may allow the robot to avoid obstacle but will not have to disturb him during the pick and place phase. The major strength of our approach will be to apply environment modelling to an autonomous complex system.

6.3 Towards open source robotics

The EUROC is also a part of a new aspect of robotics research: **the open source accessibility**. As we have already describe the basic specifications of **ROS** (see "**Software development tool**" page 15), we must also introduce others open sourced libraries and software which were used in the project's development.

First of all, the **Point Cloud Library** ("PCL", **[Rusu10]**) provides an open sourced set of point cloud processing tools free to use for education, research or commercial solution. It release is linked to the creation of the **Open Perception Foundation**, a non-profit public organisation which aims at promoting its development. For the major part coded in C++, PCL is cross platform and encourage feedbacks and completion. It is financed by many commercial companies like NVidia, Google or Intel but is also helped by universities around the world. As we already talk about **Octomap** (**[Hornung13]**), we can say that the use of **PCL** will be a great help for octrees generations. In fact, its tools will allow us to filter the point clouds before generating octrees and so guarantee the quality of the environment modelling.

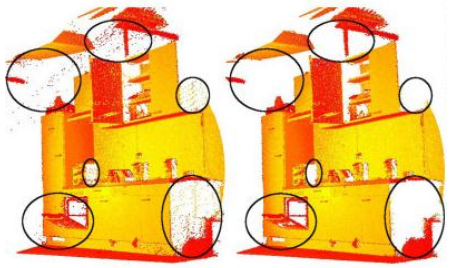


Figure 23 – Removing noise with PCL (PCL tutorials)

The simulator **Gazebo** is also a great example of open sourced tools in the robotics development community. Released in 2009, Gazebo allows building an indoor freely configurable environment with realistic physics to robot simulations purposes (Fig. 24). Gazebo is now financially supported and integrated into ROS by **Willow Garage** and became a reference in robotics simulation especially by being the official base software of the **DARPA Robotics Challenge 2013**.

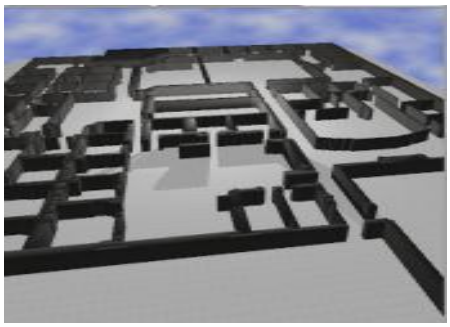


Figure 24 – The Willow Garage map (Gazebo tutorials)

In a nutshell, to base the EUROC model systems on ROS and encourage the use of open source development kits is obviously first to allow every challenger team to work with the same accessible tools, but also to promote cooperation between researchers without software compatibility issues or proprietary restrictions. As a consequence, benchmarking is easier and development costs are reduced. Furthermore, the close link between the EUROC and the European industry might lead to a new way to develop industrial robotics solutions based on an open sourced ground in the years to come.

7. Realisation on simulator

7.1.1 Frames transformations

Notice: *Code lines in this part refer to* Transformations broadcaster ROS node (C++) *page 52 and* System Parameters Publisher (Python) *page 54.*

First of all, to clarify the interface between the **"environment modeling"** and **"path planning"** parts, we have to agree on a global frame to represent in the point clouds and the octrees. This frame is called "map" and located in the simulator in the center of the table:

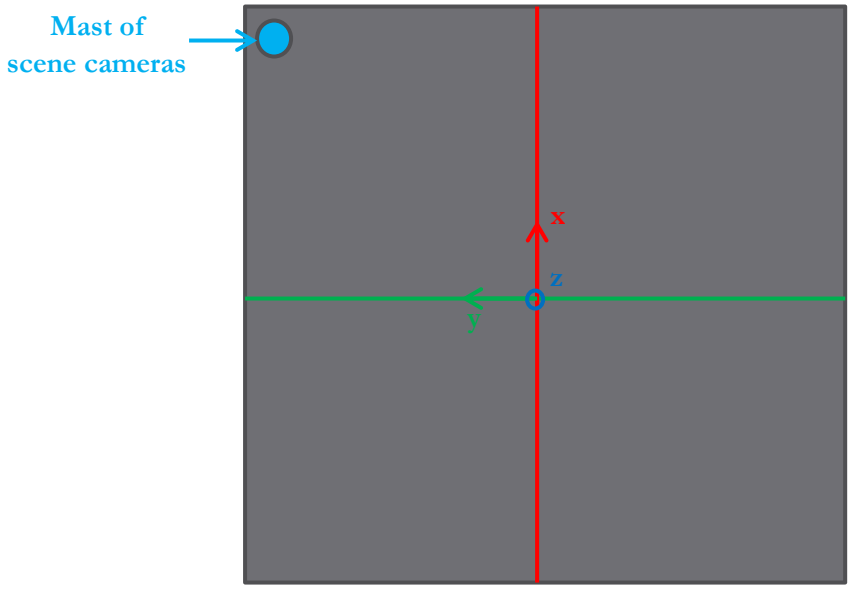


Figure 25 - "Map" frame localisation

*Notice: The color representation (**x** red, **y** green and **z** blue) will be kept as it is in the figures to come.*

Then the transformations between the arm's joints and mast's frames must be computed in order to express the point clouds in the right frame. Using the **TF Library for ROS** (**[Foote13]**), transformations calculus can be automatically done after been declared in a ROS node. **TF** allow building a **"TF tree"** where each frame is linked only to one parent (see **TF Tree** page 51). To declare a transformation, a `TransformBroadcaster` and a `Transform` objects are used:

```
static tf::TransformBroadcaster br;
tf::Transform transform;

//transform.setOrigin( tf::Vector3(msg->x, msg->y, 0.0) );
transform.setOrigin( tf::Vector3(transx, transy, 0.0) );
tf::Quaternion q;
q.setRPY(0.0, 0.0, 0.0);
transform.setRotation(q);
br.sendTransform(tf::StampedTransform(transform, time2, "map", "lwr_base"));
```

Here the frames `"map"` and `"lwr_base"` are linked by a 2D translation of `transx` toward x-axis and `transy` towards y-axis. The rotation is used as quaternion by the program but for more clarity we use "roll, pitch, yaw" angles. Here not rotation is necessary: `q.setRPY(0.0, 0.0, 0.0)`. Then the `TransformBroadcaster` makes sure that this transformation is always available when the program runs.

Concerning the 7 DOF arm, the **Denavit-Hartenberg** representation gives us the values of the rotation parameters α (for **x** axis rotation) and θ (for **z** axis rotation) as the same time as the translation parameters **a** and **d**. The DH table for this arm is (with q_i actual value of joint i) the following:

α [rad]	a [m]	d [m]	θ [rad]
0	0.0	0.16	q_1
$\boldsymbol{\pi/2}$	0.0	0.15	q_2
$\boldsymbol{-\pi/2}$	0.0	0.4	q_3
$\boldsymbol{-\pi/2}$	0.0	0.0	q_4
$\boldsymbol{\pi/2}$	0.0	0.39	q_5
$\boldsymbol{\pi/2}$	0.0	0.0	q_6
$\boldsymbol{-\pi/2}$	0.0	0.0	q_7

To that we add a simple **2D translation** to locate the base of the arm on the table. With the end-effector and the two cameras come static transformations. Values have been measured on the real system:

Frames	Translations [m]			Rotation [rad]		
	X	Y	Z	Roll	Pitch	Yaw
Arm link 6 – Base of gripper	0	0	0.08	0	0	π
Base of gripper – Center of gripper	0	0	-0.093	$-\pi$	0	0
Base of gripper – TCP RGB camera	-0.02	-0.56	-0.063	$-\frac{\pi}{2}$	$\frac{\pi}{2}$	0
TCP RGB camera – TCP Depth camera	0	-0.04	0	0	0	0

Mast's transformations are given by the simulator building. They are static except for the 2 DOF cameras support. The transformations are the following:

Frames	Translations [m]			Rotation [rad]		
	X	Y	Z	Roll	Pitch	Yaw
Map - Base of mast	0.92	0.92	0	0	0	$\frac{3\pi}{2}$
Base of mast - Base of support	0	0	1.1	0	0	0
Base of support – Support plate	0	0	0	0	Tilt	Pan
Support plate – Scene RGB camera	0.2	0.2	0	0	0	0
Scene RGB camera – Scene Depth camera	0	-0.04	0	0	0	0

All dynamically changed parameters (q_i, arm's translations and pan/tilt angles) are broadcasted through a specific ROS node and listened by the **TF tree**:

```
    ros::Subscriber sub_stamp = node.subscribe("current_telemetry_time", 10,
stampCallback);
   ros::Subscriber sub_trans = node.subscribe("arm_translations", 10,
armtransCallback);
   ros::Subscriber sub_pantilt = node.subscribe("pan_tilt_angles", 10,
pantiltCallback);
   ros::Subscriber sub = node.subscribe("current_configuration", 10,
configurationCallback);
```

Here we create four `Subscriber` objects linked each to one publisher designed by its name (for example: `"arm_translations"`). We wait for `10` messages to be received before clamping the link. Then the local callback function (example: `armtransCallback`) processes the data.

To finish, the frames can be viewed dynamically using **RViz Visualizer**:

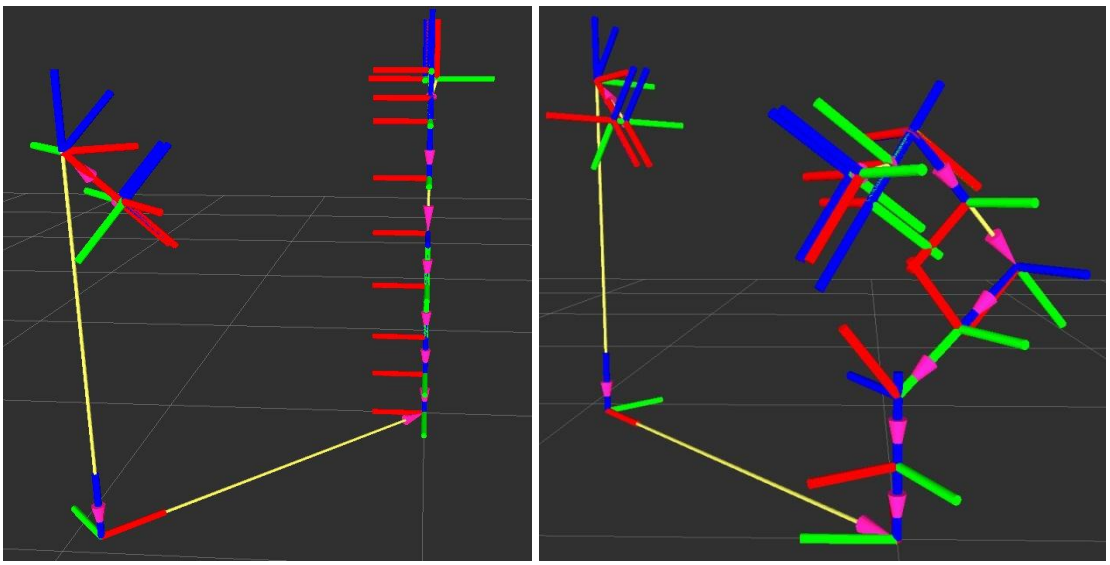


Figure 26 - Simulations frames

7.1.2 Point clouds creation

Here we detail the path to obtain octree from the simulator's two pairs of RGB cameras from point clouds generation to octree making. In fact, original point clouds cannot be used to generate octree. We must first select the information to keep.

Notice: *The following screenshots are generated by the* ***"scene" depth image****. The "TCP" depth image is used too during the simulation, but results* ***are similar*** *and will not been showed in this part.*

7.1.2.1 Point clouds generation using depth cameras

The created depth image contains for each pixel the depth information describing the distance between the reference camera (here it is always the left one while looking at the scene) and the viewed object. As we know its position into the reference camera frame, and as we have already implemented the **TF tree** with all system's frame transformations, we can express the position of each point in the global frame "**map**". Here, the point clouds are created by the use of a **Semi Global Matching algorithm** (**[Hirschmüller05]**) with as input both RGB images from one pair of cameras.

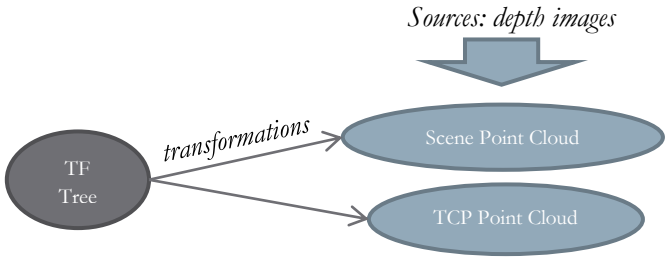


Figure 27 - Link between TF Tree and point clouds

The point cloud can be visualized in **RViz Visualizer** in the global frame:

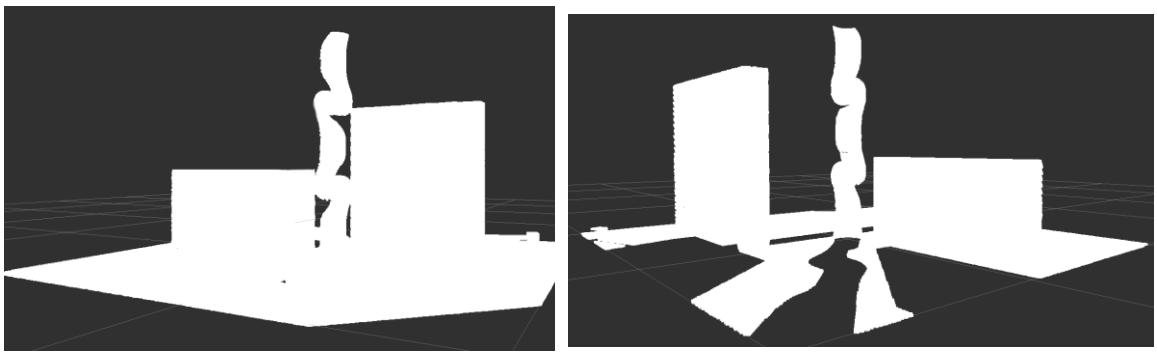


Figure 28 - Scene point cloud from two points of view

Here, the point cloud represents the whole scene viewed by the cameras. We will see in the next sections that some information have to be deleted. Furthermore, experiences have shown that the high number of points drastically increase the octree processing time. Before generating those octrees, we should first down sample the point cloud.

7.1.2.2 Down sampling

Notice: *Code lines in this part refer to* Down sampling Point Cloud Node (C++) *page 55*

In order to decrease the filtering and octree processing time, we sample the current point cloud:

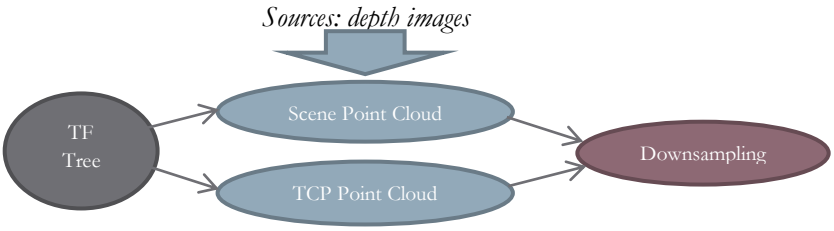
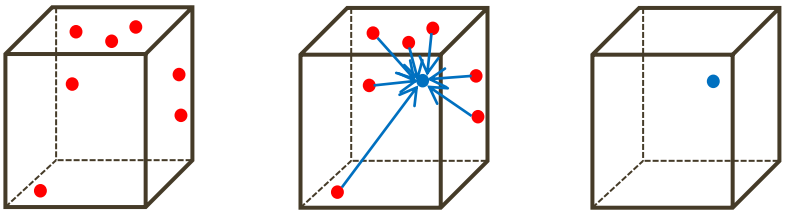


Figure 29 - Add the down sampler

The **Point Cloud Library** (**[Rusu10]**) is an open source C++ function library which provides an efficient number of tools for point cloud processing. For the down sampling, we will simply use a **VoxelGrid** filter. This filter divides the 3D space into voxels of a given resolution, and then calculates the centroid of every point in that voxel. Those very points are so replaced by the centroid:



1 - Selection of a set of points | 2 - Calculation of the centroid | 3 - Saving of the average point

Figure 30 - VoxelGrid filtering

The centroid's 3D coordinates are calculated. Be C the centroid of N points $P_i = \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}$ in a voxel subdivision and $C = \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix}$ in map, we have:

$$c_x = \frac{\sum_i x_i}{N} \; ; \; c_y = \frac{\sum_i y_i}{N} \; ; \; c_z = \frac{\sum_i z_i}{N} \quad (7.1)$$

We can see that the average point is not necessarily the center of the voxel. This approach takes more time and calculations than keeping this center but grants a better precision on edges and surfaces. As the simulator's point clouds represent a small environment (2x2x2 meters) and do not contain significant noise, we can afford to keep a high resolution after down sampling. Using PCL, experiences have shown that a 1 cm resolution gives good results.

After a complete scan of the "**Scene**" camera, without any more filtering, we pass from a number of **1 843 200** points to **59 667**, in other words **30.89 times less**. Code lines using the down sampling can be seen below:

```
pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
    pcl_conversions::toPCL(req.cloud_in, *cloud_pcl);
    sor.setInputCloud (cloud_pcl);
    sor.setLeafSize (0.01f, 0.01f, 0.01f);
    sor.filter (*cloud_filtered_TCP);
```

Here the `VoxelGrid` object takes as input the point cloud `cloud_pcl` converted beforehand into PCL format (required). Then we define the leaf size using the protected function `setLeafSize`. The result of the filtering is saved in the variable `cloud_filtered_TCP` in this example.

We can now display the down sampled point cloud:

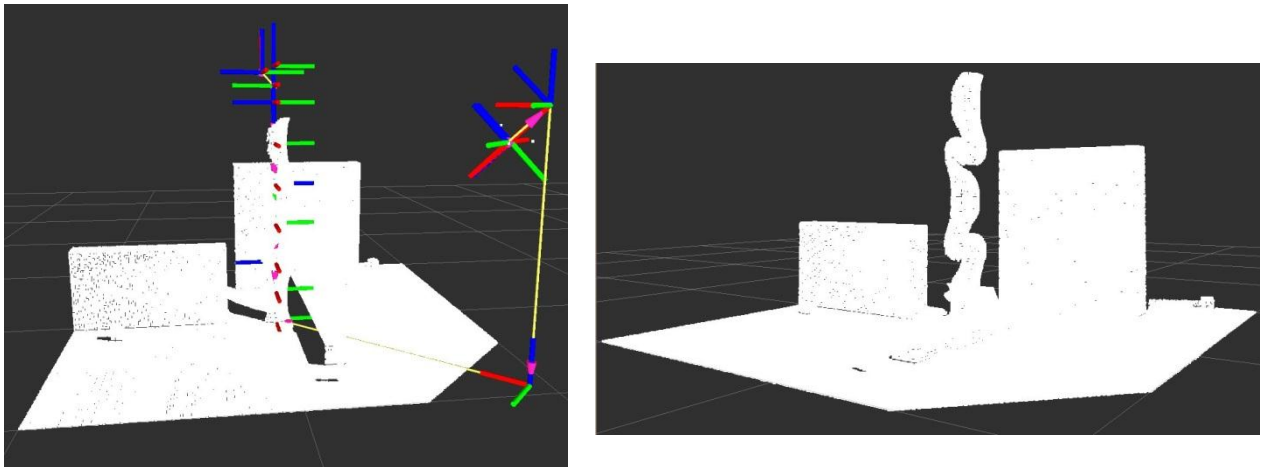


Figure 31 – Down sampled Scene point cloud

7.1.3 Octree generation with Octomap

After down sampling, the point clouds are ready to be used in order to build an octree. To do so, we use the ROS compatible open-source library Octomap (**[Hornung13]**) which provides us with tools create the octree as described in section 2 "**Introduction to environment modelling**". Combining those functions with the frame tree we built before, we are able to locate the voxels in the global frame and so match with the **TF Tree** (this provides an easier localization of obstacles for path planning).

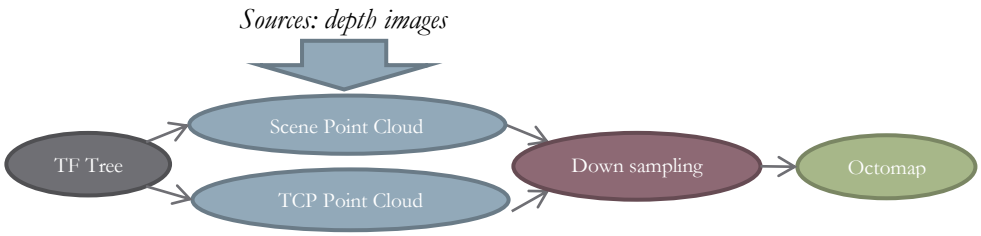


Figure 32 - Add Octomap Server

The simulator provides **trustful data** from the sensors; furthermore we know that the obstacle **will not move**. With that information, we can compute a high l_{max} and a low l_{min} and well as an acceptable gap between hit and $miss$. Experiences have shown good results for:

- $l_{max} = 0.97$
- $l_{min} = 0.12$
- $hit = 0.7$
- $miss = 0.3$

Simulation gives us the octree generated by the scene point cloud:

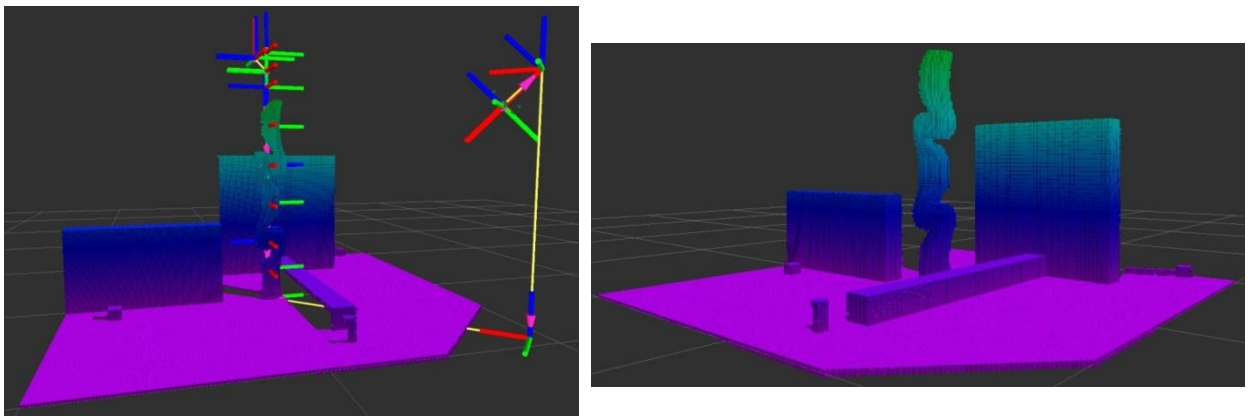


Figure 33 - Octree from the scene depth camera

But as the octree will represent the obstacles for path planning, the table and the arm should be erased from them. This task introduces a few intermediate states.

7.1.5 Point clouds filtering

7.1.5.1 Table filtering and RANSAC Algorithm

Notice: *Code lines in this part refer to* RANSAC filtering node (C++) *page 56*

The first process we use on the point cloud aim at **deleting the support table** without affecting the other obstacles or the objects. If the table appears in the octree then the path planning will consider it as obstacle and will not allow the arm to move. We implement a **RANSAC (Random Sample Consensus) algorithm** (**[Fischler81]**) to the down sampled point cloud:

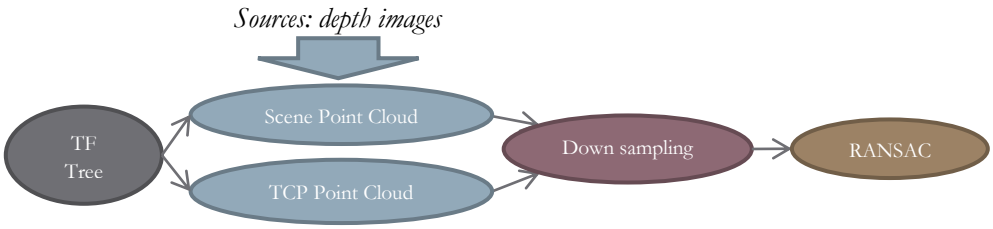


Figure 34 - Add RANSAC algorithm

The **Random Sample Consensus** algorithm is a **non-deterministic iterative method**. It aims at finding in random set of data taken from a root one a match with a chosen model. If the data set is well chosen, with a recognizable model and enough data to match it, then the learning aspect of the algorithm is able to find a better match for each iteration. Its greatest advantage is the **robustness** in the model matching, and the more iteration we process, the better the result. Furthermore, compared to classical model fitting algorithm like "**least squares**", RANSAC does not use the entire data set to run and so avoid deviations. On the downside, we do not have control on the **computation time** and have to **manually enter the fitting parameters** which require one function per application.

In this section we explain the algorithm as the same time as we discuss the implementation. First of all, the RANSAC filtering will be defined as a **ROS-service**; as a consequence we call it only when necessary and save computer resources:

```
    ros::init(argc, argv, "ransac_filtering");
    ros::NodeHandle n;
    ros::ServiceServer service_scene = n.advertiseService("ransac_scene",
chatterCallback_scene);
```

Then we define the model and its parameters. In our case, we want to find a planar surface containing all the points within a **distance threshold** of 5 *mm*. This threshold is a security against noise; we do not need a bigger value because, in the simulation, the cameras cannot see the down side of the table.

```
    pcl::SACSegmentation<pcl::PointXYZ> seg;
    seg.setOptimizeCoefficients (true);
    seg.setModelType (pcl::SACMODEL_PLANE);
    seg.setMethodType (pcl::SAC_RANSAC);
    seg.setMaxIterations (10);
    seg.setDistanceThreshold (0.005);
```

We start the algorithm, until the iterations are completed and if the number of data is sufficient we set as input the current down sampled point cloud and take randomly a set of points. Then the RANSAC algorithm defines a set of **inliers**:

```
    int nr_points = (int) cloud_converted_scene->points.size ();
    while (cloud_converted_scene->points.size () > 0.5 * nr_points)
    {
        seg.setInputCloud (cloud_converted_scene);
        seg.segment (*inliers, *coefficients);
        if (inliers->indices.size () == 0)
        {
            std::cout << "Could not estimate a planar model for the given dataset." <<
std::endl;
            break;
        }
```

Those **inliers** are then extracted from the current point cloud. As here we want to delete the table, we remove them from the tested data and use the rest as a new input:

```
        // Extract the planar inliers from the input cloud
        pcl::ExtractIndices<pcl::PointXYZ> extract;
        extract.setInputCloud (cloud_converted_scene);
        extract.setIndices (inliers);
        extract.setNegative (false);

        // Get the points associated with the planar surface
        extract.filter (*cloud_plane);

        // Remove the planar inliers, extract the rest
        extract.setNegative (true);
        extract.filter (*cloud_f);
        *cloud_converted_scene = *cloud_f;
```

When one of the two stopping conditions is validated, the latest filtered point cloud is sent back as **service response**. Here we see the point cloud after been filtered by the **RANSAC algorithm**:

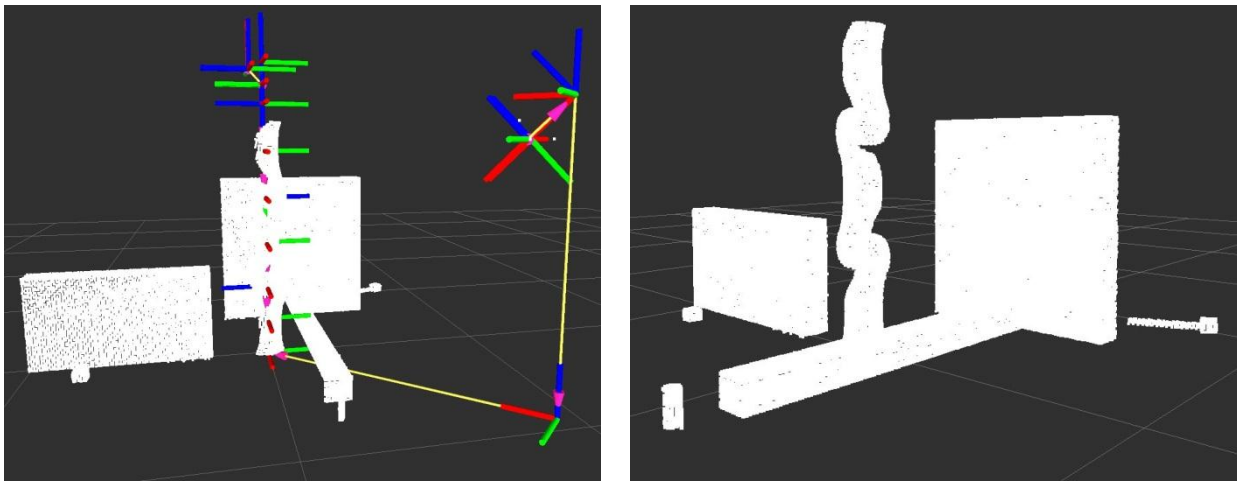


Figure 35 – Scene down sampled point cloud after RANSAC

7.1.5.2 Arm filtering

Notice: *Code lines in this part refer to* Filtering the arm from « Scene » point cloud (C++) *page 57*

A final part to delete from the "Scene" point cloud is the **7 DOF arm**. If we do not then the path planning will see an obstacle right in the same position as the arm and will fail. We work with the RANSAC filtered "Scene" point cloud:

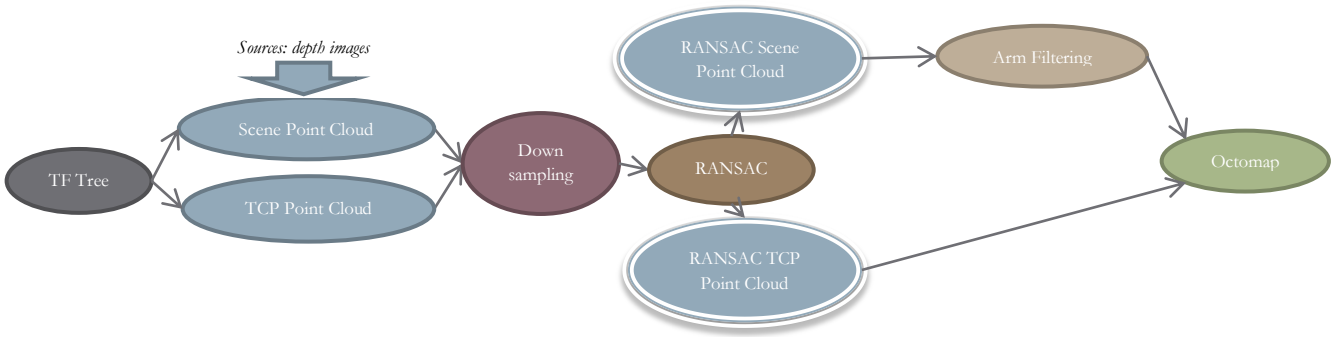


Figure 36 - Add arm filtering

We first locate the arm in the global frame and know its configuration. Also, using the TF Tree provides us with the current position of all arms' frames and so computes point transformation. As a consequence, we divide the arm into four parts representing the arm's major pieces:

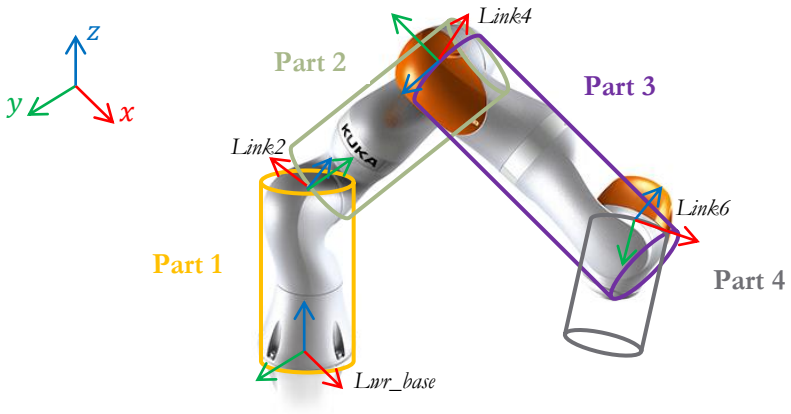


Figure 37 - Divisions of the LWR Arm

From here we approximate each selected part with a cylinder located at the base of each piece:

	Begin point (frame origin)	End point (frame origin)	Radius (m)	High (m)
Part 1	Lwr_base	Link2	0.11	0.31
Part 2	Link2	Link4	0.11	0.4
Part 3	Link4	Link6	0.11	0.39
Part 4	Link6	handmade point*	0.11	0.233

**Handmade point created by a translation of 0.223 cm of link6 origin toward y axis to include the gripper and the fingers*

For each point in the current point cloud, we check if it belongs to each previous part following this algorithm:

1 – Query **TF** to transform the current point from map to the selected frame. We wait for the transformation to be ready then execute it. Example:

```
listener.waitForTransform("lwr base",
dummy.header.frame_id, timestamp,
ros::Duration(10.0) );

listener.transformPoint("lwr_base",
timestamp, dummy, "map",dummy_resp);
```

2 – Generation of the current point cylindrical coordinates in the new frame. Radius is calculated regarding to the two axes generating the cylinder (according to the **DH representation**). For *Lwr_base*:

$$radius_{base} = \sqrt{x_{point}^2 + y_{point}^2} \quad (7.2)$$

3 – If the calculated radius is bigger than the cylinder **reference radius**, then the current point does not belong to this part of the arm. We increment the integer *part_verified*.

4 – The point belongs to an infinite cylinder generated by the previous frame. If its high (for *lwr_base* the z coordinate) is bigger than the **current part length** then the point does not belong to this very part. We increment *part_verified*. Example (for 3 and 4):

```
if (radius >
ref radius){part verified++;}
//Else if we are close from the axis,
but too high, this is still a candidate
else if (dummy_resp.point.z > 0.31){
part verified++;}
```

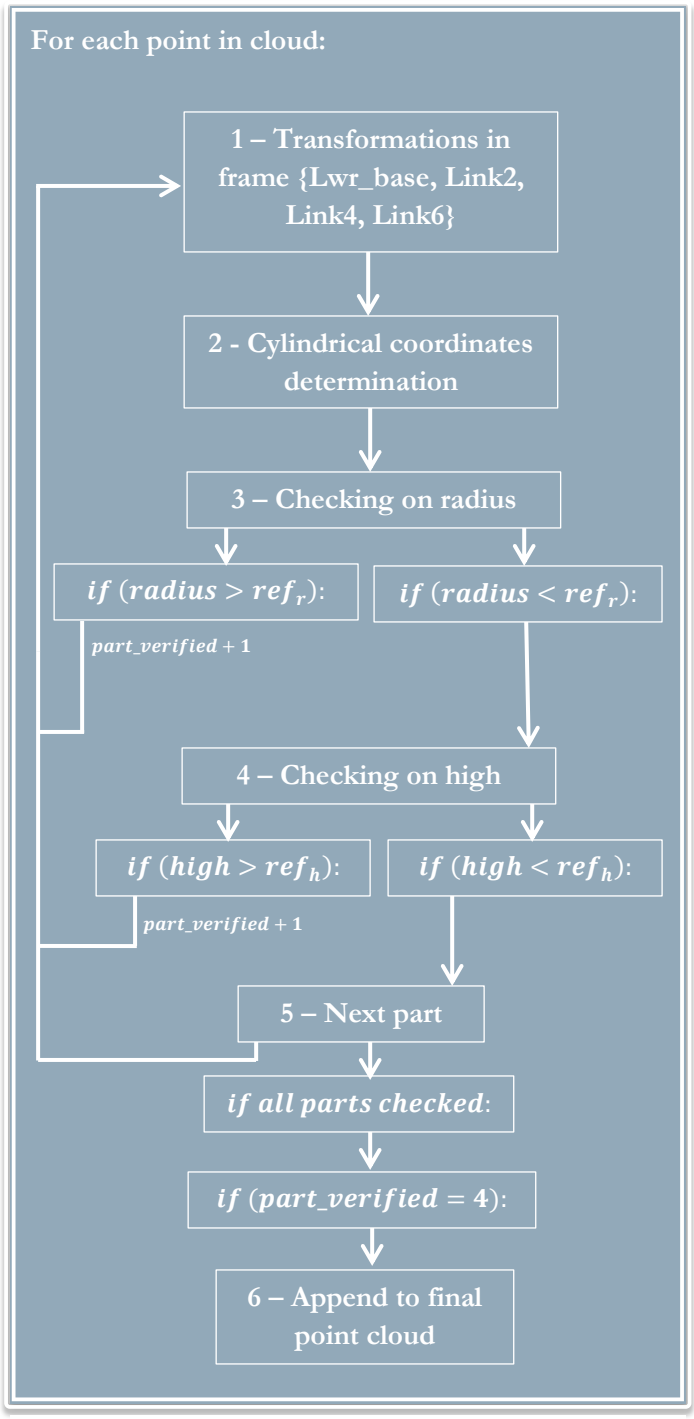


Figure 38 - Arm deletion algorithm

5 – Switch to next part.

6 – If the current point has been defined has outside all parts, than we append it into a final point cloud.

```
if (part_verified == 4){
            final_pointcloud.points.push_back(req.pointcloud_to_transform.points[i]);}
```

This previous algorithm is provided as a **ROS service** and called after the **RANSAC filtering** to reduce the number of points and reduce the processing time. The final point cloud now contains only the obstacles and the objects as we can see below:

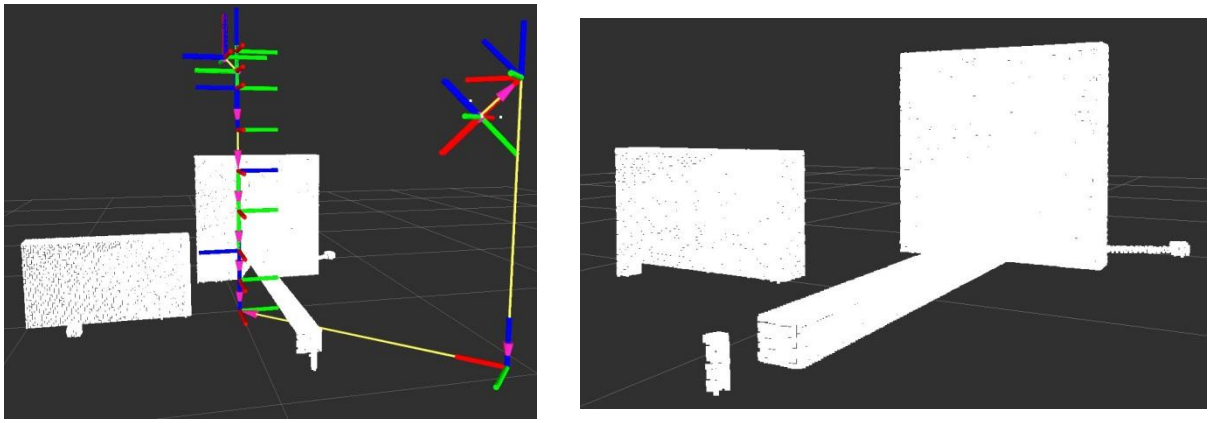


Figure 39 - Final scene point cloud

We can now build the octree using **Octomap** and use it for path planning into the simulator:

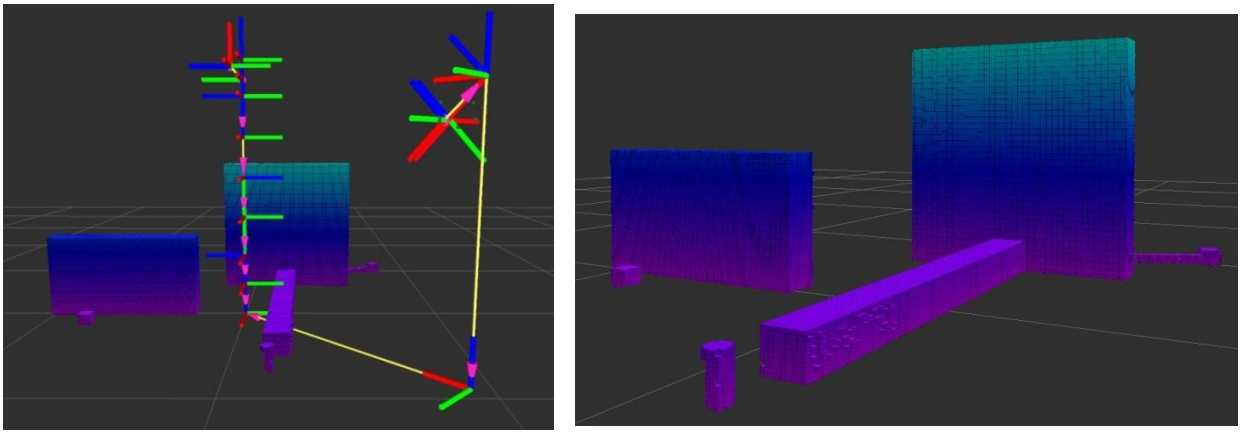


Figure 40 - Final filtered octree

8. Realization on real system

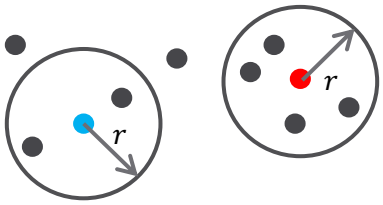
After the development on simulator, the scenario must be exported to the real system. Helped by a set of **topics** and **services** integrated in the robot's computers by the titular team, we are able to use the actuators and gather data from the sensors. Furthermore, this part of the development could detect **bugs**. Reporting those problems will make the system more robust. In this section, we describe the additional necessary work to adapt the previous scenario to the real system. First of all, we have to add processes in order to filter the **noise of the point clouds**. Then, we discuss the system's **state machine**.

8.1 Octree realization on real system

8.1.1 Noise reduction

To switch to the real system brings a new aspect in point clouds: **noise**. If octree generation can handle a part of the noise reduction, the amount of data and the frequency of updates can add in the octree false obstacles and disturb path planning. As a consequence, a **noise reducing filter** must be designed. Noise could disturb two processes. First, the presence of isolated points in the environment **adds obstacles** to avoid during path planning. Too many of those obstacles make the planning fails, especially if we plan for a big element such as the base of the **Miiwa**. Then, we can observe noise around plane surfaces like the robot's desktop. This noise may make harder the **object recognition** and localization, and be able to distinguish objects to manipulate is also a part of environment modeling.

We use a **radius filter** to delete noise. This filter checks for each point its number of neighbors inside a sphere with a given radius. If this point has fewer neighbors than a threshold, then it is designated as noise and deleted:



In this example, if the threshold is 3, then the blue point will be considered as noise.

Figure 41 - Principle of radius filter

To find parameters which fit our case, we proceed to **measurements** on down sampled point clouds in the testing room (the down sampling stays the same as in section 7.1.2.1). For each point, we calculate the mean distance between it and its k-nearest neighbors. Here $k = 100$, this choice is arbitrary but aims at keeping precision in the noise reduction as the number of points in a point cloud after down sampling is more than ten times this value of k. As a first example, we observe the calculated mean distances for a down sampled point cloud made by looking at the Miiwa's desktop with the "scene" cameras, moreover an object to pick (a screw) is added:

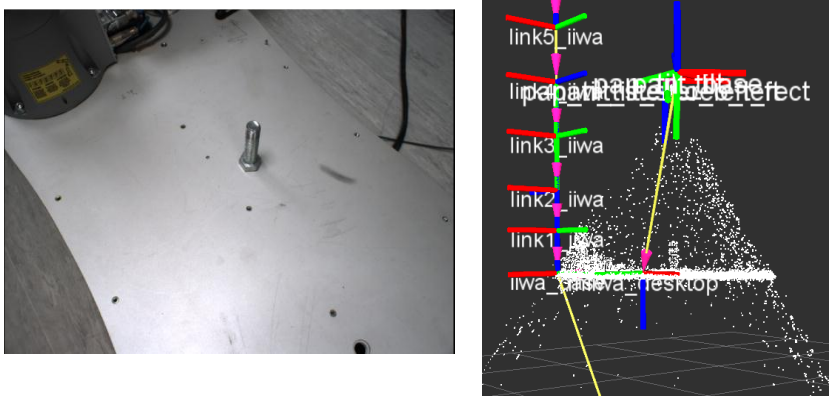


Figure 42 - Desktop point cloud and associated RGB image

The following scheme represents the mean distance between each point (designated by their position in the point cloud) and its 100-nearest neighbors (in m):

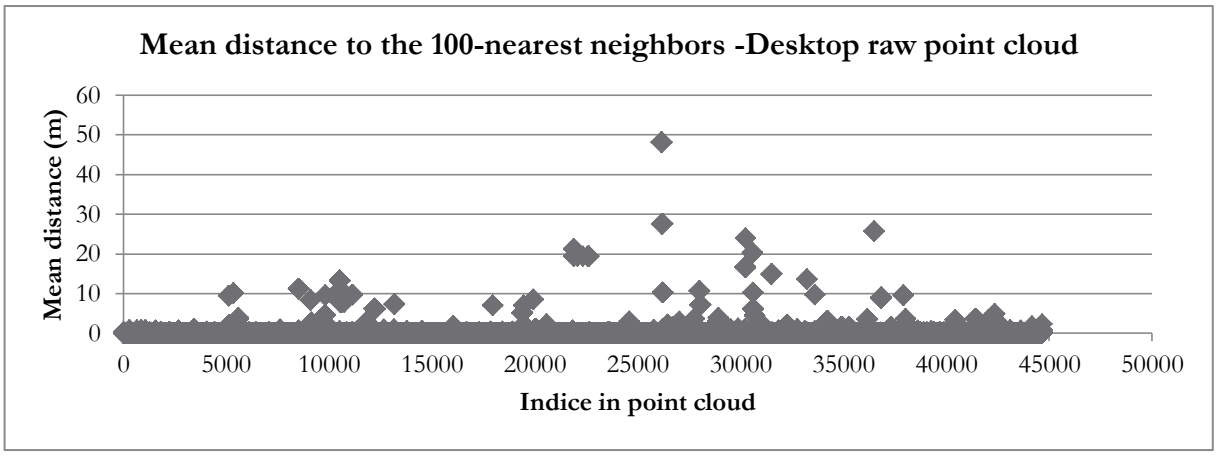


Figure 43 - Measures for desktop raw point cloud

From those measures, results are:

- **Number of points** in the point cloud: 44669
- **Mean distance** to the 100-nearest neighbors: 0.13 m
- **Ratio** of points away from their 100-nearest neighbors by a mean distance:
    - Between 0 m and 0.05 m : 93%
    - Between 0.05 m and 0.1 m : 2.97%
    - Higher than 0.1 m : 4.03%

As a consequence, a **radius of 0.06 m** is chosen to filter the noise. A ROS-Service is designed to apply the filter to the down sampled point cloud using PCL. The number of neighbors to contain in the sphere is set up to 100. Then we measure again the mean distance between each point and its 100-nearest neighbors:

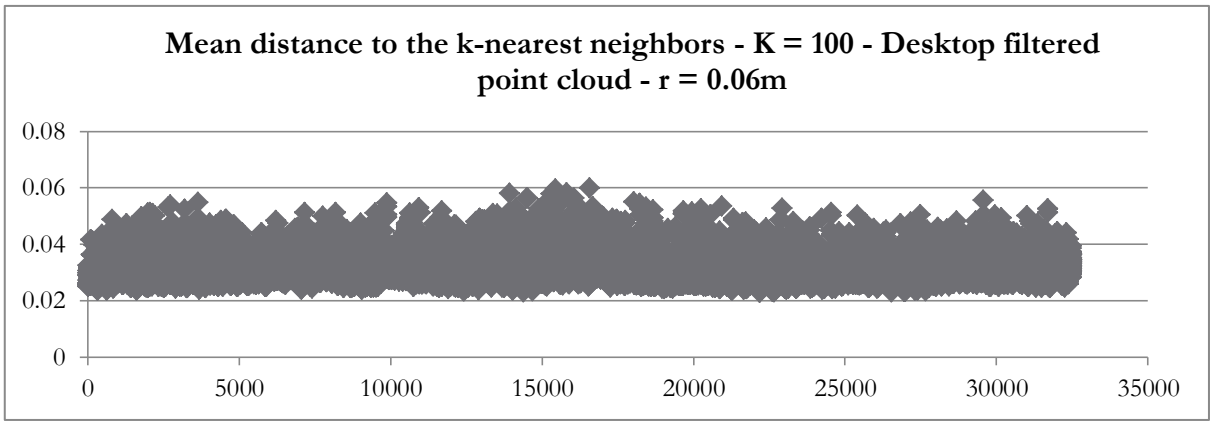


Figure 44 - Measures for filtered desktop point cloud

From those measures, results are:

- **Number of points** in the point cloud: 32482
- **Mean distance** to the 100-nearest neighbors: 0.03 m
- **Ratio** of points away from their 100-nearest neighbors by a mean distance:
    - Between 0 m and 0.05 m : 99.71%

- Between 0.05 m and 0.1 m : 0.29%
- Higher than 0.1 m : 0%

***NB**: The radius value of 0.06 is chosen because a value of 0.05 decreases the quality of the screw representation in the point cloud.*

***NB 2**: Another example which confirms the chosen parameters is given annex* Another example to confirm noise filter's parameters *page 58*.

The designed filter gives good results for the chosen parameters. Which **RViz**, we visualize the filter's effect on the same point cloud:

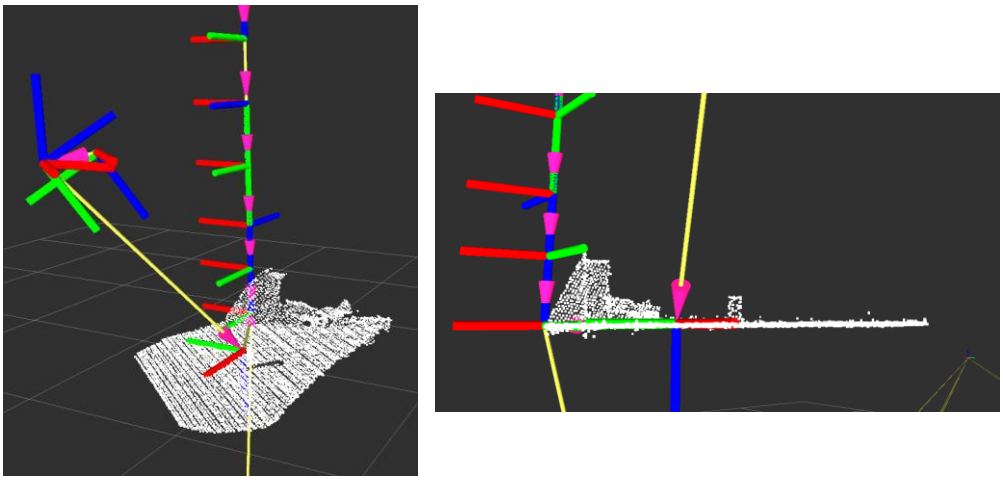


Figure 45 - Noise filtered desktop point cloud

8.1.2 Concatenation of point clouds

With the previous filters details in section 7.1.2.2 and 8.1.1, we can now build a general point cloud of the environment before making a first octree. Using the "scene" cameras, we take a set of pictures from the robot's starting position. The generated point cloud is filtered and concatenated with the previous ones (with an empty one at first). Doing that, a 360° map of the room is built and so can be used by the path planning as input.

More pictures of the octree can be found in **Final octree screenshots** page 60.

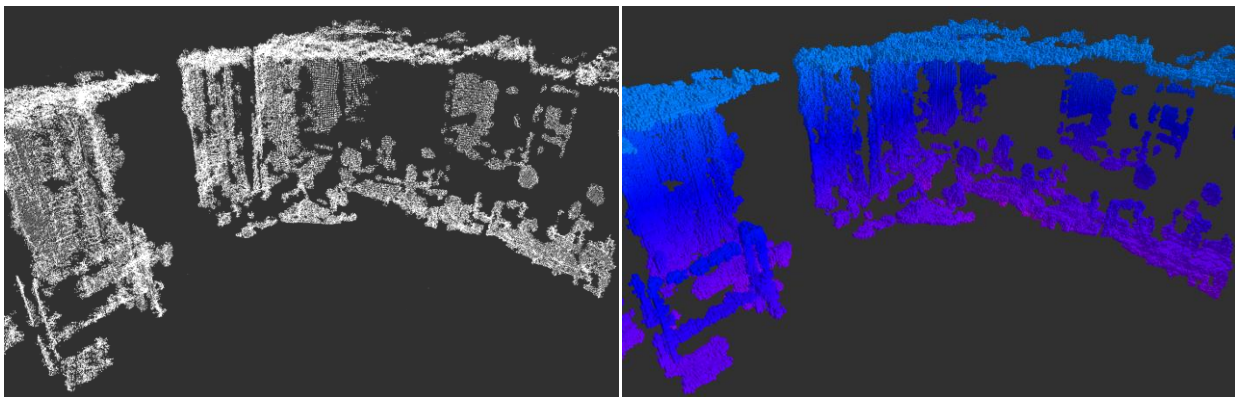


Figure 46 - Fraction of the final octree

9. Critics and expectations

The work done on environment modeling and tasks management has given results good enough to allow the team to achieve a pick and place scenario with the KUKA Miiwa. However, a few issues have been discovered during the development. One example which has affected environment modeling is the too long processing time to create point clouds. As a consequence, only one map is made in the beginning of the scenario and updates take time. But as current constraints do not include moving obstacles, this has not disturbed the octree creation. Another problem appeared in the robot's software and forbid the "hand" cameras to produce point clouds and participate to the mapping like during simulations. But this does not question the progress of the scenario.

Those issues and others have been reported to the titular team by the student team during the development and have been or will be taken care of before the second challengers teams visit on November 2015.

As soon as they are fixed, we can expect a better modeling of obstacles especially for complex ones such as shelves or if the picking area is crowded. The state machine could also been completed with additional tasks and backups depending on the scenario.

The next step of the student team development would be to achieve a small assembly task during this pick and place scenario (for example: to screw a nut on a bolt). The filters designed for environment modeling, as well as the state machine, constitute the base of this future development. Furthermore, this report has been uploaded on the Robotics and Mechatronics Center wiki page with more information about the project in order to keep track of this work and transmit it to the next student team.

10.Conclusion

To conclude, a demonstration scenario has been developed to help the DLR EUROC team to find and fix bugs and misbehaviors on the new KUKA Miiwa. As a consequence, EUROC challenger teams can now develop their own solution on the system.

To do so, a map of the environment has been created using an octree generated by point clouds. Those point clouds have been filtered in order to only keep their interesting information. In fact, this map was used as one of the path planning's input and helped to avoid obstacles while base moving as well as while manipulating an object with the arm. Furthermore, to keep the system autonomous, a state machine has been developed. It commanded the robot's actions to achieve a pick and place task with object detection and obstacle avoidance.

After developing on simulator, to switch on the real system allowed the team to detect and fix a few issues but also to validate the actuators and sensors running. Now the DLR is planning to make the robot achieve more complicated tasks like small assembly, and the work done constitutes the base of a future development.

With this project, the EUROC hope to open new perspectives on applied autonomous mobile robotics in the European industrial field. But more than specific applications, they also promote the use of open-source development in robotics research and engineering.

From a personal point of view, I consider this internship as a success. During those six months at DLR, I was given the chance to apply my skills to an international and multidisciplinary engineering project. The tasks I was assigned made me learn a lot about computer vision and tasks management but also about teamwork and project management. That professional experience brought me an education which will be essential in the years to come.

11. Bibliography

Publications

[Butterfasse01] ***DLR-Hand II : Next Generation of a Dexterous Robot Hand,*** J.Butterfasse; M.Grebenstein, H.Liu and G.Hirzinger, *IEEE International Conference on Robotics & Automation*, 2001

[Cupec05] ***An approach to environment modeling for biped walking robots,*** R.Cupec, G.Schmidt, *Intelligent Robots and Systems IEEE International Conference,* 2005

[Fischler81] ***Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography,*** A.Fischler and R.Bolles, *Comm. of the ACM,* 1981

[Foote13] ***TF : The Transform Library,*** T.Foote, *IEEE International Conference on Technologies for Practical Robot Applications*, 2013

[Fuchs09] ***Rollin' Justin – Design considerations and realization of a mobile platform for a humanoid upper body,*** M. Fuchs, Ch. Borst, P. Robuffo Giordano, A. Baumann, E. Kraemer, J. Langwald, R. Gruber, N. Seitz, G. Plank, K. Kunze, R. Burger, F. Schmidt, T. Wimboeck and G. Hirzinger,*ICRA'09: Proceedings of the 2009 IEEE international conference on Robotics and Automation*, 2009

[Heigele12] ***GridTiles: A Method for Modelling a Spatial Unconstrained Environment,*** C. Heigele, H. Mielenz, J. Heckel and D.Schramm, *Intelligent Computer Communication and Processing IEEE International Conference,* 2012

[Hirschmüller05] ***Accurate and Efficient Stereo Processing by Semi-Global Matching and Mutual Information,*** H.Hirshmüller, *IEEE Conference on Computer Vision and Pattern Recognition, 2005*

[Hornung13] ***OctoMap : A Probabilistic, Flexible, and Compact 3D Map Representation for Robotic Systems***, A.Hornung, K.M. Wurm, M. Bennewitz, C. Stachniss, W. Burgard, *Autonomous Robots Volume 34*, 2013

[Jessup14] ***Merging of Octree Based on 3D Occupancy Grip Map,*** J.Jessup, S.N.Givigi, A.Beaulieu, *System Conference (SysCon), 8th Annual IEEE*, 2014

[Meagher81] ***Octree Encoding: A new technique for the Representation, Manipulation and Display of Arbitrary 3D Object by Computer,*** D.Meagher, *Computer Graphics and Image Processing 19*, 1981

[Ouyang12] ***Octree-based Spherical hierarchical model for Collision detection,*** F.Ouyang and T.Zhang, *IEEE World Congress on Intelligent Control and Automation*, 2012

[Pitzer12] ***PR2 Remote Lab: An environment for remote development and experimentation,*** B.Pitzer, S.Osentoski, G.Jay, C.Crick, O.Jenkins, *IEEE International Conference on Robotics and Automation*, 2012

[Rusu10] ***3D is here : Point Cloud Library (PCL)***, R.B. Rusu and S. Cousins, *ICRA Communications*, 2011

Websites

Official DLR website: http://www.dlr.de

Official EUROC website: http://www.euroc-project.eu

PR2 online manual provided by Willow Garage:
https://pr2s.clearpathrobotics.com/wiki/PR2%20Manual

Official Unbounded Robotics website: http://unboundedrobotics.com

Official KIVA Robotics website: http://www.kivasystems.com/

Online documentations and tutorials for:

- Robot Operating System (ROS) : http://www.ros.org
- Point Cloud Library (PCL) : http://www.pointclouds.org
- Octomap : https://octomap.github.io

12.Annexes

12.1 DLR

The DLR (***Deutsches Zentrum für Luft- und Raumfahrt***) is the national research center for aerospace, energy and transportation of Germany. It also has the role of the **German Space Agency**. Founded in 1907, the DLR has come through many names and expanded a lot before it became what we know today. From plane modeling, it included in 1948 the first German space research center. As we speak, its research activities are obviously aerospace and aircraft but are also directed to nowadays issues like energies, green tech, transports, robotics, and more. With its headquarter in **Cologne**, the DLR has approximately 7400 employees divided in 13 German centers and 3 Europeans and Americans centers in Brussels, Paris and Washington DC. It is also engaged in education. In fact, the **DLR School Labs** take place in 10 DLR centers and aim at developing the scientific curiosity of teenagers with weekly courses and experiments and promote research studies. Furthermore, it is opened to train international interns and student-workers.

During the past decades, the DLR has been involved in many researches and development projects like the **Mir Space Station** (1997) and the development of the **Boeing 747** aircraft. It has also participated in recent projects like the **Rosetta** space probe which has transported the robot **Philae**. In terms of robotics, the humanoid robot **Justin** used in research labs (**[Fuchs09]**) equipped by the **DLR-Hand II** (**[Butterfasse01]**) can be taken as an example of DLR's great achievements.

Figure 47 - DLR's Justin and Hand

The Weßling DLR's site "**DLR Oberpfaffenhofen**" is one of the biggest DLR site in Germany. It currently employs around 1700 researchers, engineers and human resources workers in a dozen of institutes and facilities. Its fields of studies are:

- Microwaves and radars
- Remote Sensing Technology
- Communication and navigation
- System dynamics and control
- Robotics and mechatronics

Furthermore, the Weßling site is able to train pilots and astronauts due to a set of simulators and training facilities like landing areas. It includes also an **Earth observation center**. The **Robotics and Mechatronics Center (RMC)** is considered as a world leading research centers. Its strength comes from the gathering of three institutes working in the **DLR Oberpfaffenhofen** such as the **Institute for System Dynamics and Control**, the **Institute for Optical Sensor Systems** and the **Institute for Robotics and Mechatronics** where my internship has taken place. If the main areas of studies were DLR's main topics like aeronautics, transport and robotics applications in space, the **RMC** activities has been diversified. In fact, the **RMC** in involved in appliances of robotics in various nowadays issues like medical robotics, factory of the future and human assistance.

12.2 Miiwa's gripper and pan tilt

Figure 48 - Miiwa's gripper

Figure 49 - Miiwa's pan tilt

12.3 Details on LogOdds function

If $p \in]0;1[$, the LogOdds function is defined by:

$$logOdds(p) = \log\left(\frac{p}{1-p}\right) = \log(p) - \log(1-p) \quad (N)$$

The function graph on is definition segment is the following:

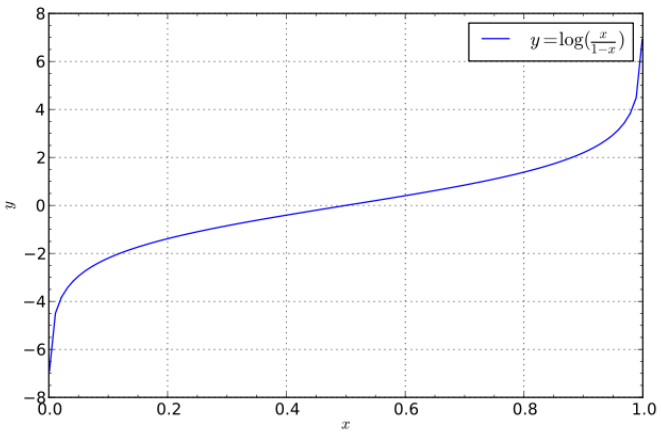
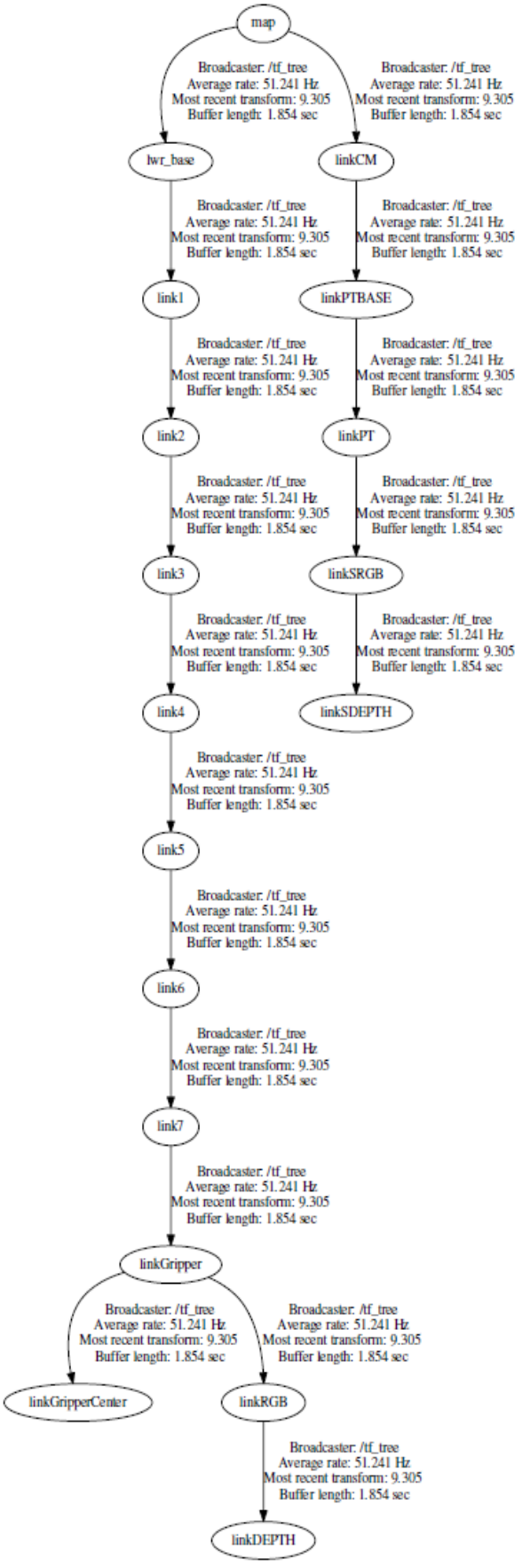


Figure 50 - logOdds graph

Regarding to this previous graph, we can confirm chooses of setting up $P(n) = 0.5$ when $t = 0$ and the importance of the condition ($0 < miss < 0.5$ *and* $0.5 < hit < 1$) made in section 3.

12.4 TF Tree


map
Broadcaster: /tf_tree
Average rate: 51.241 Hz
Most recent transform: 9.305
Buffer length: 1.854 sec
lwr_base
linkCM
link1
linkPTBASE
link2
linkPT
link3
linkSRGB
link4
linkSDEPTH
link5
link6
link7
linkGripper
linkGripperCenter
linkRGB
linkDEPTH

12.5 Transformations broadcaster ROS node (C++)

Notice: Due to the redundancy of this source code, only the 7 DOF arm frames transformations are here represented.

```
#include <ros/ros.h>
#include <time.h>
#include <tf/transform_broadcaster.h>
#include <euroc_c2_msgs/SearchIkSolution.h>
#include <std_msgs/Header.h>

float transx, transy, pan, tilt ;
u_int32_t secondes, nsecondes;
ros::Time time2 ;

void stampCallback(const std_msgs::Header &header){
 secondes = header.stamp.sec;
 nsecondes = header.stamp.nsec;
 ros::Time time(secondes, nsecondes);
 time2 = time;
}

void armtransCallback(const euroc_c2_msgs::Configuration &config){
  transx=config.q[0];
  transy=config.q[1];
}

void pantiltCallback(const euroc_c2_msgs::Configuration &config){
  pan=config.q[0];
  tilt=config.q[1];
}

void configurationCallback(const euroc_c2_msgs::Configuration &config){

  std::cout << "config q.[0] contains: " << config.q[0] << std::endl;

  static tf::TransformBroadcaster br;
  tf::Transform transform;

  tf::TransformBroadcaster br1;
  tf::Transform transform1;

  tf::TransformBroadcaster br2;
  tf::Transform transform2;

  tf::TransformBroadcaster br3;
  tf::Transform transform3;

  tf::TransformBroadcaster br4;
  tf::Transform transform4;

  tf::TransformBroadcaster br5;
  tf::Transform transform5;

  tf::TransformBroadcaster br6;
  tf::Transform transform6;

  tf::TransformBroadcaster br7;
  tf::Transform transform7;

// ------------------------------------------------------- ARM TRANFORMATIONS TREE

// ------------------------     VARIABLE TRANFORMATIONS
  //transform.setOrigin( tf::Vector3(msg->x, msg->y, 0.0) );
  transform.setOrigin( tf::Vector3(transx, transy, 0.0) );
  tf::Quaternion q;
  q.setRPY(0.0, 0.0, 0.0);
  transform.setRotation(q);
  br.sendTransform(tf::StampedTransform(transform, time2, "map", "lwr_base"));

  //frame 1
  transform1.setOrigin( tf::Vector3(0.0, 0.0, 0.16) );
  tf::Quaternion q1;
  q1.setRPY(0.0, 0.0, config.q[0]);
  transform1.setRotation(q1);
  br1.sendTransform(tf::StampedTransform(transform1, time2, "lwr_base", "link1"));

  //frame 2
```

```
  transform2.setOrigin( tf::Vector3(0.0, 0.0, 0.15) );
  tf::Quaternion q2;
  q2.setRPY(1.5708, 0.0, config.q[1]);
  transform2.setRotation(q2);
  br2.sendTransform(tf::StampedTransform(transform2, time2, "link1", "link2"));

  //frame 3
  transform3.setOrigin( tf::Vector3(0.0, 0.20, 0.0) );
  tf::Quaternion q3;
  q3.setRPY(-1.5708, 0.0, config.q[2]);
  transform3.setRotation(q3);
  br3.sendTransform(tf::StampedTransform(transform3, time2, "link2", "link3"));

  //frame 4
  transform4.setOrigin( tf::Vector3(0.0, 0.0, 0.20) );
  tf::Quaternion q4;
  q4.setRPY(-1.5708, 0.0, config.q[3]);
  transform4.setRotation(q4);
  br4.sendTransform(tf::StampedTransform(transform4, time2, "link3", "link4"));

  //frame 5
  transform5.setOrigin( tf::Vector3(0.0, -0.20, 0.0) );
  tf::Quaternion q5;
  q5.setRPY(1.5708, 0.0, config.q[4]);
  transform5.setRotation(q5);
  br5.sendTransform(tf::StampedTransform(transform5, time2, "link4", "link5"));

  //frame 6
  transform6.setOrigin( tf::Vector3(0.0, 0.0, 0.20) );
  tf::Quaternion q6;
  q6.setRPY(1.5708, 0.0, config.q[5]);
  transform6.setRotation(q6);
  br6.sendTransform(tf::StampedTransform(transform6, time2, "link5", "link6"));

  //frame 7
  transform7.setOrigin( tf::Vector3(0.0, 0.0, 0.0) );
  tf::Quaternion q7;
  q7.setRPY(-1.5708, 0.0, config.q[6]);
  transform7.setRotation(q7);
  br7.sendTransform(tf::StampedTransform(transform7, time2, "link6", "link7"));

}


int main(int argc, char** argv){
  ros::init(argc, argv, "tf_broadcaster");

  ros::NodeHandle node;
  ros::Subscriber sub_stamp = node.subscribe("current_telemetry_time", 10, stampCallback);
  ros::Subscriber sub_trans = ode.subscribe("arm_translations", 10, armtransCallback);
  ros::Subscriber sub_pantilt = node.subscribe("pan_tilt_angles", 10, pantiltCallback);
  ros::Subscriber sub = node.subscribe("current_configuration", 10, configurationCallback);

  ros::spin();
  return 0;
}
```

12.6 System Parameters Publisher (Python)

```
#!/usr/bin/env python

# Publishes system dynamic parameters like arm joints values, base translations and pan tilt
angles.

import rospy

from euroc_c2_msgs.msg import *
from std_msgs.msg import *

from euroc_c2_system import euroc_c2_system

system = euroc_c2_system()


def euroc_c2_publisher():
    current_configuration = Configuration(q=[0.0] * len(system.lwr_joints))
    current axis configuration = Configuration(q=[0.0] * len(system.cam joints))
    current_cam_configuration = Configuration(q=[0.0] * len(system.axis_joints))

    rate = rospy.Rate(50) # 10hz

    while not rospy.is_shutdown():

        current_telemetry = system.get_telemetry()
        current_telemetry_time = current_telemetry.header

        for (i, joint) in enumerate(system.lwr_joints):
            telemetry_index = current_telemetry.joint_names.index(joint)
            current_configuration.q[i] = current_telemetry.measured.position[telemetry_index]

        # Get the latest arm translation values
        for (i, joint) in enumerate(system.axis_joints):
            telemetry_index = current_telemetry.joint_names.index(joint)
            current_axis_configuration.q[i] =
current_telemetry.measured.position[telemetry_index]

        # Get the latest Cam Pan/Tilt angle values
        for (i, joint) in enumerate(system.cam_joints):
            telemetry index = current telemetry.joint_names.index(joint)
            current cam configuration.q[i] =
current_telemetry.measured.position[telemetry_index]

        pub_curconf.publish(current_configuration)

        # current_axis_configuration.q[0] > x translation
        # current_axis_configuration.q[1] > y translation
        pub_trans.publish(current_axis_configuration)

        # current cam configuration.q[0] > cam pan
        # current cam configuration.q[1] > cam tilt
        pub_pantilt.publish(current_cam_configuration)

        pub_time.publish(current_telemetry_time)

        rate.sleep()


if __name__ == '__main__':
    pub_trans = rospy.Publisher('arm_translations', Configuration, queue_size=10)
    pub_pantilt = rospy.Publisher('pan_tilt_angles', Configuration, queue_size=10)
    pub_curconf = rospy.Publisher('current_configuration', Configuration, queue_size=10)
    pub_time = rospy.Publisher('current_telemetry_time', Header, queue_size=10)

    try:
        euroc_c2_publisher()
    except rospy.ROSInterruptException:
        pass
```

12.7 Down sampling Point Cloud Node (C++)

```
#include <iostream>
#include <ros/ros.h>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/filters/voxel grid.h>
#include <pcl/conversions.h>
#include <pcl_conversions/pcl_conversions.h>

#include <../../devel/include/environment_modeling_pkg/DownsamplePointcloud.h>

pcl::PCLPointCloud2::Ptr cloud_filtered_Scene (new pcl::PCLPointCloud2 ());
pcl::PCLPointCloud2::Ptr cloud filtered TCP (new pcl::PCLPointCloud2 ());
pcl::PCLPointCloud2::Ptr cloud_pcl (new pcl::PCLPointCloud2 ());

sensor_msgs::PointCloud2 final_pointcloud2;

bool downsample_tcp(environment_modeling_pkg::DownsamplePointcloud::Request &req,
environment_modeling_pkg::DownsamplePointcloud::Response &res)
{
    final_pointcloud2.data.clear();
    // Create the filtering object
    pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
    pcl_conversions::toPCL(req.cloud_in, *cloud_pcl);
    sor.setInputCloud (cloud_pcl);
    sor.setLeafSize (0.01f, 0.01f, 0.01f);
    sor.filter (*cloud_filtered_TCP);

    pcl_conversions::fromPCL(*cloud_filtered_TCP, final_pointcloud2);
    res.cloud_out = final_pointcloud2;

    return true;
}

bool downsample_scene(environment_modeling_pkg::DownsamplePointcloud::Request &req,
environment_modeling_pkg::DownsamplePointcloud::Response &res)
{
    final_pointcloud2.data.clear();
    // Create the filtering object
    pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
    pcl conversions::toPCL(req.cloud_in, *cloud_pcl);
    sor.setInputCloud (cloud pcl);
    sor.setLeafSize (0.01f, 0.01f, 0.01f);
    sor.filter (*cloud_filtered_Scene);

    pcl_conversions::fromPCL(*cloud_filtered_Scene, final_pointcloud2);
    res.cloud_out = final_pointcloud2;

    return true;
}

int main (int argc, char** argv)
{
    ros::init(argc, argv, "downsampling_pointcloud");
    ros::NodeHandle n;

    ros::ServiceServer service_tcp = n.advertiseService("downsample_tcp", downsample_tcp);
    ros::ServiceServer service_scene = n.advertiseService("downsample_scene",
downsample_scene);

    ros::spin();

    return (0);
}
```

12.8 RANSAC filtering node (C++)

```
sensor_msgs::PointCloud2 ransaced_scene_pointcloud, ransaced_tcp_pointcloud;
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_converted_scene (new
pcl::PointCloud<pcl::PointXYZ>);
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_converted_tcp (new pcl::PointCloud<pcl::PointXYZ>);

bool chatterCallback_scene(environment_modeling_pkg::RansacFiltering::Request &req,
environment_modeling_pkg::RansacFiltering::Response &res){

    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_f (new pcl::PointCloud<pcl::PointXYZ>);
    pcl::PCLPointCloud2::Ptr temp_cloud(new pcl::PCLPointCloud2);
    pcl_conversions::toPCL(req.cloud_in,*temp_cloud);
    pcl::fromPCLPointCloud2(*temp_cloud,*cloud_converted_scene);

    // Create the segmentation object for the planar model and set all the parameters
    pcl::SACSegmentation<pcl::PointXYZ> seg;
    pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
    pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_plane (new pcl::PointCloud<pcl::PointXYZ> ());

    seg.setOptimizeCoefficients (true);
    seg.setModelType (pcl::SACMODEL_PLANE);
    seg.setMethodType (pcl::SAC_RANSAC);
    seg.setMaxIterations (10);
    seg.setDistanceThreshold (0.005);

    int nr_points = (int) cloud_converted_scene->points.size ();
    while (cloud_converted_scene->points.size () > 0.5 * nr_points)
    {
        // Segment the largest planar component from the remaining cloud
        seg.setInputCloud (cloud_converted_scene);
        seg.segment (*inliers, *coefficients);
        if (inliers->indices.size () == 0)
        {
            std::cout << "Could not estimate a planar model for the given dataset." <<
std::endl;
            break;
        }

        // Extract the planar inliers from the input cloud
        pcl::ExtractIndices<pcl::PointXYZ> extract;
        extract.setInputCloud (cloud converted_scene);
        extract.setIndices (inliers);
        extract.setNegative (false);

        // Get the points associated with the planar surface
        extract.filter (*cloud_plane);

        // Remove the planar inliers, extract the rest
        extract.setNegative (true);
        extract.filter (*cloud_f);
        *cloud_converted_scene = *cloud_f;
    }

    // Convert the PCL::PointCloud into a sensor_msgs::PointCloud2 for exploitation
    pcl::toROSMsg(*cloud_converted_scene, ransaced_scene_pointcloud);
    res.cloud_out = ransaced_scene_pointcloud;

    return true;
}


int main (int argc, char** argv)
{
    ros::init(argc, argv, "ransac_filtering");
    ros::NodeHandle n;

    ros::ServiceServer service_scene = n.advertiseService("ransac_scene",
chatterCallback_scene);

    ros::spin();

    return (0);
}
```

12.9 Filtering the arm from « Scene » point cloud (C++)

```
sensor_msgs::PointCloud final_pointcloud;

float ref_radius = 0.11; // meters

bool filter_arm(environment_modeling_pkg::FilterArm::Request &req,
environment_modeling_pkg::FilterArm::Response &res)
{
    tf::TransformListener listener;
    final_pointcloud.points.clear();

    for (int i = 0; i < req.pointcloud_to_transform.points.size() ; i++){

        geometry_msgs::PointStamped dummy, dummy_resp;
        float radius; // meters
        int part_verified = 0;

        // We need to create a PointStamped to use tf
        dummy.header.frame id = "map";
        dummy.point.x = req.pointcloud_to_transform.points[i].x;
        dummy.point.y = req.pointcloud_to_transform.points[i].y;
        dummy.point.z = req.pointcloud_to_transform.points[i].z;

        ros::Time timestamp(req.time);

        /// Check between lwr_base and link1
        listener.waitForTransform("lwr_base", dummy.header.frame_id, timestamp,
ros::Duration(10.0) );
        listener.transformPoint("lwr_base", timestamp, dummy, "map",dummy_resp);

        radius = sqrt( pow(dummy_resp.point.x, 2) + pow(dummy_resp.point.y, 2));
        // Check the distance between the point and the axis
        if (radius > ref_radius){ part_verified++; /*Is a candidate to keep*/ }
        //Else if we are close from the axis, but too high, this is still a candidate
        else if (dummy_resp.point.z > 0.31){ part_verified++;  /* Is a candidate to keep */ }

        /// Check between link1 and link4
        listener.waitForTransform("link2", dummy.header.frame_id, timestamp,
ros::Duration(10.0) );
        listener.transformPoint("link2", timestamp, dummy, "map", dummy_resp);

        radius = sqrt( pow(dummy resp.point.x, 2) + pow(dummy_resp.point.z, 2));
        if (radius > ref_radius){ part_verified++; }
        else if (dummy_resp.point.y > 0.40){ part_verified++; }

        /// Check between link4 and link6
        listener.waitForTransform("link4", dummy.header.frame_id, timestamp,
ros::Duration(10.0) );
        listener.transformPoint("link4", timestamp, dummy, "map", dummy_resp);

        radius = sqrt( pow(dummy_resp.point.x, 2) + pow(dummy_resp.point.z, 2));
        if (radius > ref_radius){ part_verified++; }
        else if (dummy_resp.point.y < -0.39){ part_verified++; }

        /// Check between link6 and the end of the fingers
        listener.waitForTransform("link6", dummy.header.frame_id, timestamp,
ros::Duration(10.0) );
        listener.transformPoint("link6", timestamp, dummy, "map", dummy_resp);

        radius = sqrt( pow(dummy_resp.point.x, 2) + pow(dummy_resp.point.z, 2));
        if (radius > ref_radius){ part_verified++; }
        else if (dummy_resp.point.y > 0.093+0.08+0.06){ part_verified++; }

        /// If the point does not belong to any part
        if (part_verified == 4){
            final_pointcloud.points.push_back(req.pointcloud_to_transform.points[i]);}
    }
    res.transformed_pointcloud = final_pointcloud;
    return true;
}
int main(int argc, char** argv){
    ros::init(argc, argv,"FilterArm");
    ros::NodeHandle n;
    ros::ServiceServer service = n.advertiseService("filter_arm", filter_arm);
```

12.10 Another example to confirm noise filter's parameters

In this section, we apply the **same method** as in section 8.1.1 page 41 to design the **radius filter** aiming at reducing the **noise** in point clouds. Here we take a bigger point cloud representing a fraction of the experiment room:

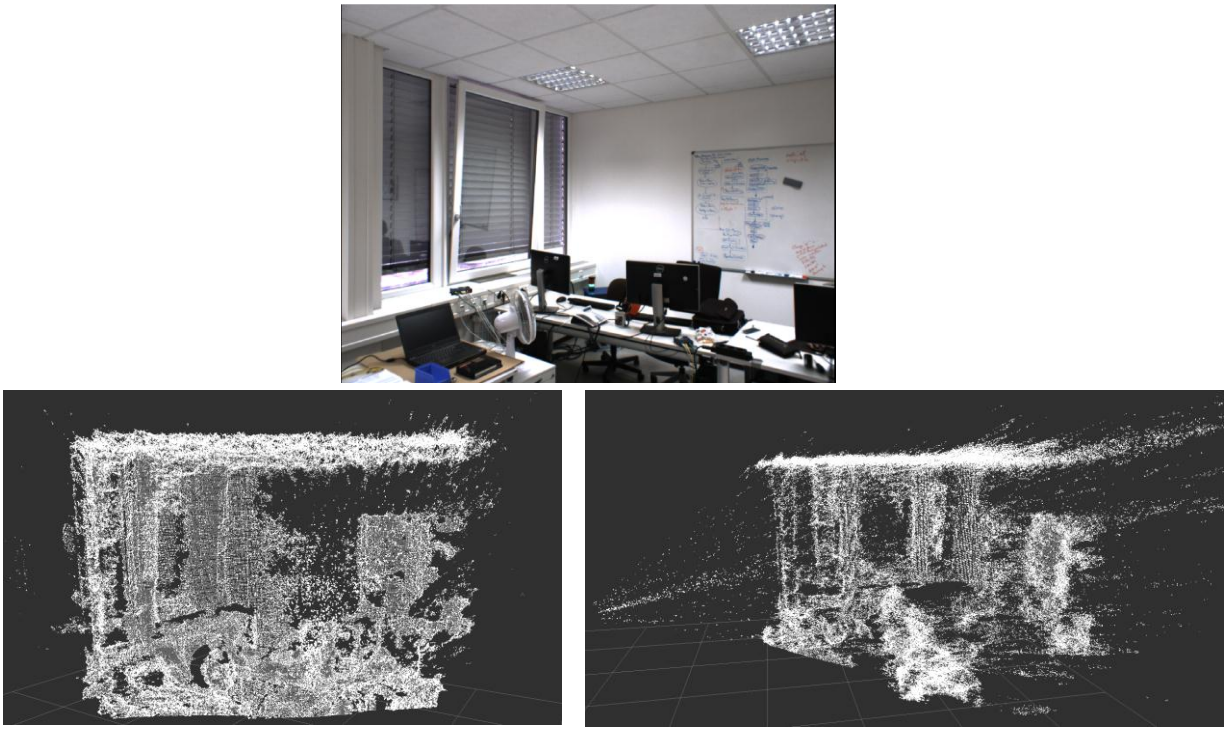


Figure 51 – Raw lab point cloud and associated RGB image

We calculate the mean distance between **each point and its 100-nearest neighbors**:

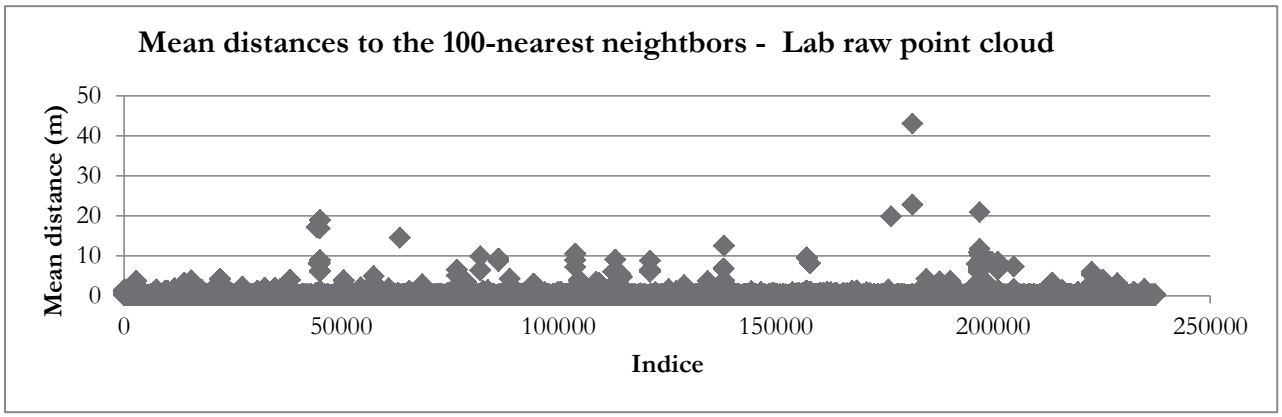


Figure 52 - Measured for raw lab point cloud

From those measures, results are:

- **Number of points** in the point cloud: 237296
- **Mean distance** to the 100-nearest neighbors: 0.08 m

- **Ratio** of points away from their 100-nearest neighbors by a mean distance:
  - Between 0 m and 0.05 m : 61.69%
  - Between 0.05 m and 0.1 m : 28.06%
  - Higher than 0.1 m: 10.25%

The noise reducing filter is applied with a radius of 0.06 m and $k = 100$:

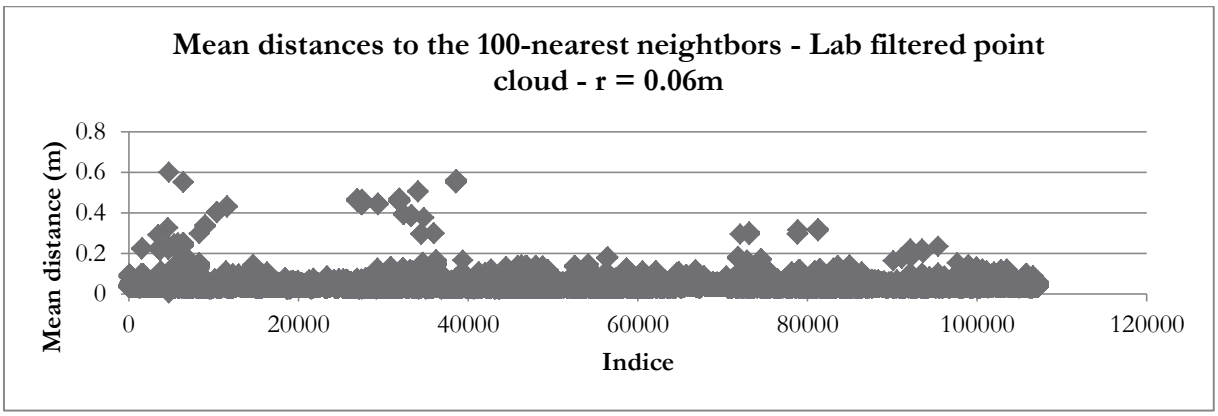


Figure 53 - Measures for filtered lab point cloud

From those measures, results are:

- **Number of points** in the point cloud: 107216
- **Mean distance** to the 100-nearest neighbors: 0.03 m
- **Ratio** of points away from their 100-nearest neighbors by a mean distance:
  - Between 0 m and 0.05 m : 92.85%
  - Between 0.05 m and 0.1 m : 6.53%
  - Higher than 0.1 m: 0.62%

The results **confirm** the chosen parameters. With RViz, we visualize the filtered point cloud:

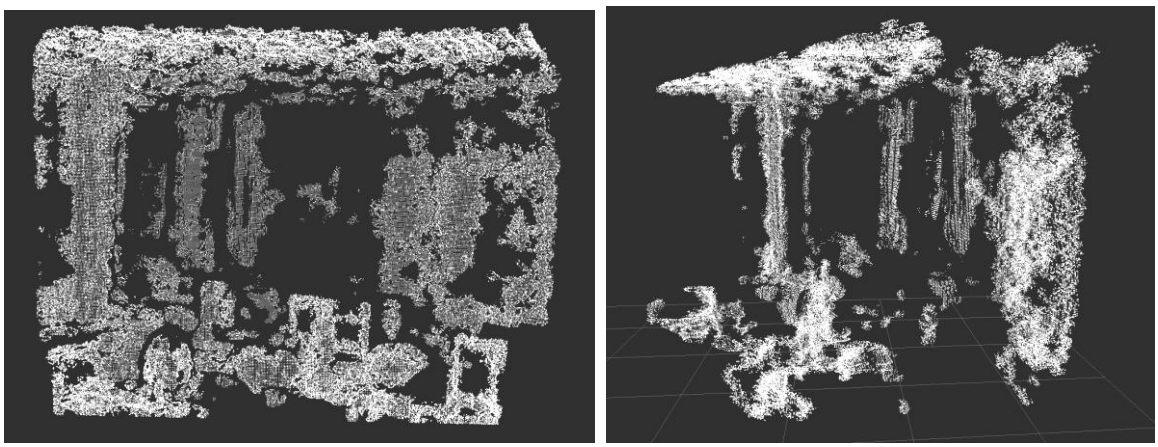


Figure 54 - Lab filtered point cloud

12.11 Final octree screenshots

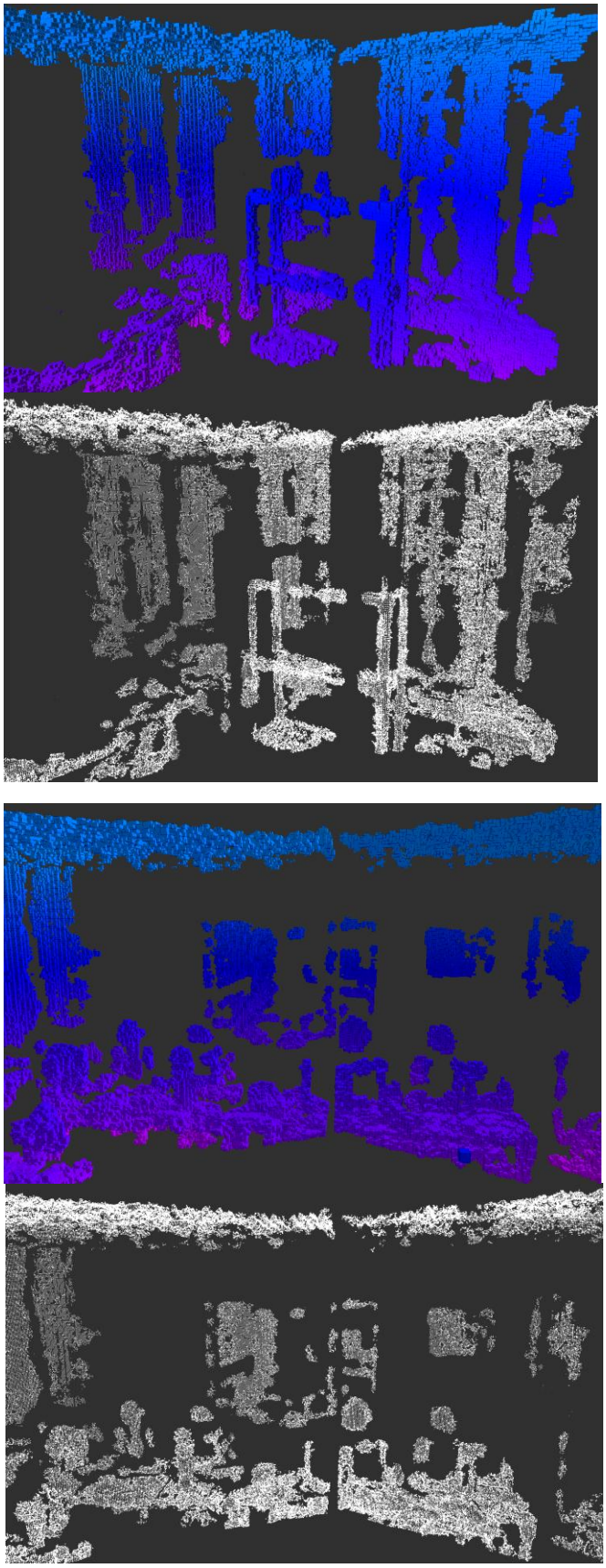


Figure 55 - Others example of final octrees