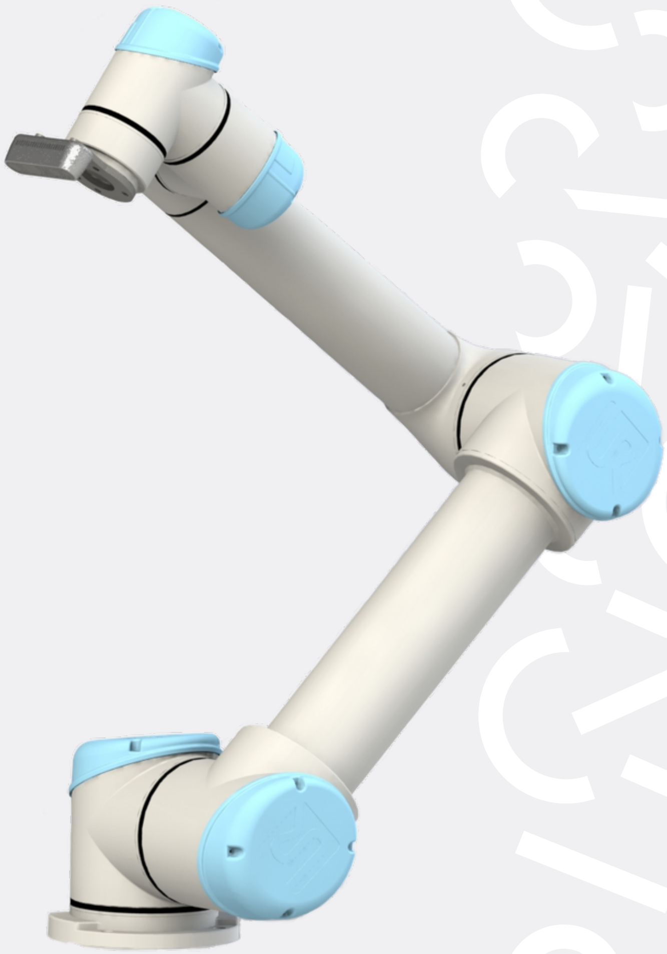
Model-free object grasping

Model-free object grasping with a learning-free approach.

TOM ERIK VANGE



SUPERVISORS
Jing Zhou, professor, UiA.
Ilya Tyapin, associate professor, UiA.

University of Agder, 2021
Faculty of Engineering and Science
Department of Engineering and Sciences

Abstract

The industry standards and capability are constantly advancing and pushing forward to increase data collection, efficiency, profit, and quality as well as decrease downtime, injuries, and hazards as much as possible. In recent years, robot systems have received more attention in the context of a large number of industrial applications, such as automotive manufacturing, additive manufacturing, assembly, quality inspection, and co-packing. The collaboration between multiple robots and human operators is considered to be the most prominent strategy in Industry 4.0 and future Industry 5.0, sharing the same space and collaborating on tasks according to their complementary capabilities. With the use of robots and their abilities could efficiency, profit, safety, and quality be further increased, potentially revolutionizing the industry and production.

This project was supported in part by DEEPCOBOT Project. DEEPCOBOT, Collective Efficient Deep Learning and Networked Control for Multiple Collaborative Robot Systems, are a research project funded by IKTPLUSS under Grant 306640/O70 from the Research Council of Norway. The project will investigate the design of a new generation of decentralized data-driven Deep Learning based controllers for multiple coexisting collaborative robots, which interact both between themselves and with human operators in order to collectively learn from each other's experiences and perform cooperatively different complex tasks in large-scale industrial environments. This is motivated by the increasing demand of automation in industry, especially the demand of a safer and more efficient collaboration between multiple robots and human operators to integrate the best of human abilities and robotic automation.

This project has looked into the problem of grasping an unknown and previously unseen object with a learning-free approach. By implementing a model-free picking algorithm onto a robot arm with a gripper and robot vision could it be able to pick up a vast variety of objects.

A virtual environment has been created with a robot arm and depth-camera during this project. The result from this project is a setup that is able to scan objects placed on a workbench and create a point cloud representation of these objects. The point cloud is since used to calculate the curvature of the objects, creating a foundation for further use in a learning-free setup for grasping previously unseen objects.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There has been a vast focus on efficiency and automation within industries and manufacturing over the last couple of decades. By improving the production, one could profit more while increasing the quality of the products, decreasing downtime, and enhancing the safety of workers at the plant. With the increasing efficiency in a plant, are often robots introduced within the production line for repetitive work. These solutions often have a more static implementation, meaning that if there are any changes in the work environment outside of the given scenario, either the robot environment must be changed back to the original setup or the robot re-programmed to account for these changes.

Interaction with unknown objects in an open-world environment is a challenging problem. A robot could, for instance, have to navigate to a given location for task execution, see and understand the working environment and what object to grasp and interact with. The robot would need to be equipped with a vision system, where it needs to be programmed to find out what it is seeing and how it should act accordingly with it.

With upgrading and implementation to new standards within industries and manufacturing, like Industry 4.0 and collaborating robots (also called cobots), a more dynamic, multi-purpose production line can be achieved. With a more dynamic and adaptive setup, might a robot solve several tasks and establish a safe work environment with the interaction between robots and humans.

This project will look into a learning-free approach, meaning no previous recorded data or training would be needed to execute a task to observe and interact with a previously unseen object for a robot. This project will set up a virtual test environment for a robot arm to observe and create point clouds of previously unseen objects, creating a model-free setup as a foundation for further projects.

1.1 Motivation

Most robots today are programmed to execute specific, fixed tasks in a defined area. These could do repetitive pre-defined tasks. If something in the environment changes, the robot might not solve the task anymore and would either need a correction in the working environment or reprogramming of the robot to suit the changes. This is one of the drawbacks with a fully static setup, that if any change comes up out of line with the designed setup, the task might not be fulfilled to a suitable level.

By taking a more dynamic approach, a robot could position itself or adjust to some of the changes, thereby lowering downtime where a stop of the process and correction earlier would have taken place. To make a robot more dynamic, must cameras, lasers, or other sensors that could give vital information about the work environment be implemented so that the robot could act accordingly to its' work environment. However, only implementing the sensor is not enough since the data must be treated, analyzed, and interpreted before appropriate actions can take place accordingly.

The motivation for this project comes from the increasing demand for automation in the industry. Increasing efficiency, accuracy, reliability, and safety to constantly new levels is an exciting and impressive achievement. During the last couple of decades, there has been a vast development where the industries, companies, and academia must adapt to new changes and requirements more and more rapidly. Looking forward, will the demand only increase further for creating a safer work environment, more refined products, availability, and affordability of equipment. In the pursuit of this will the use of robots increase to execute tasks with speed, high precision, and inexhaustible undertaking execution capabilities in various tasks and environments.

1.2 Goals for Project

This project aims to develop a foundation for learning-free and model-free object grasping in a virtual environment. The setup should simulate a working environment in ROS, a software framework used to program and control robots, that could later be tested and implemented in a physical setup. A robot created and controlled in the virtual environment should see objects as a partial point cloud. The system should execute the object's point cloud construction autonomously, not relying on pre-defined models or physical parameters from the object or any learning-based input for its construction or calculation. A learning-free algorithm purposed to evaluate the objects are Local Contact Moment (LoCoMo) [3]. LoCoMo does not need any prior knowledge of an object other than a view of it as a partial point cloud. The project should look into and lay a foundation for applying a LoCoMo application into the environment. This setup could provide a foundation for further testing and development for students or lecturers, creating a dynamic robot setup able to execute grasps of previously unseen objects.

1.3 Project Limitations

The scope for this project is limited due to several factors. This project will look into the LoCoMo algorithm but not develop an equivalent LoCoMo algorithm because of its size of scope. Instead, the goal will be to implement and set up a test environment in ROS for virtual testing with the possibility for further physical tests. The project will further apply and test a Zero Moment Shift (ZMS) calculation used to describe a point cloud's curvature in the virtual environment. This could since a LoCoMo algorithm use to find a feasible grasp on the objects surface.

An additional factor affecting the project was the uncertainty of what equipment to use and availability due to the demand from other students and the ongoing pandemic. The limitation was chosen to create a more durable, stable environment and application rather than an unstable test setup which could be hard to use since one of the criteria in the project is to develop and set up an environment that could be used and further developed.

1.4 State-of-the-Art

With the ever-increasing demand for efficiency, profit, and data as the industry of automation and robotics pushes forward are new methods and research developed. This chapter briefly overviews some of the related work and research towards grasping objects and their approach to solving this problem. There are two main approaches towards grasping techniques, learning-free and learning-based, also known as analytic and data-based approaches. These two approaches try to solve the challenging problem of grasping and interacting with an object in two different ways.

Learning-based Approach

A learning-based approach is using one form of Artificial Intelligence (AI), for instance, Machine Learning (ML) or Convolutional Neural Network (CNN), to solve a task. The learning-based approach and use have lately gained more interest from academia and industry for its potential. ML consists of several different algorithms which rely on previously recorded data. It can through experience or training data improve to execute a specific task. With CNN are data fed through a network consisting of several layers of nodes, input-layer, hidden-layer(s), and output-layer. The links between each node are weighted, or tuned, correspondingly for the network to achieve the performance wanted.
Applying a learning-based approach to a grasping robot could start by learning how to pick up an object. As the robot collects more data about grasping objects, are the performance and accuracy increased over time as its algorithm is tuned. The positive part of a learning-based approach is the adaptability and the potentially increasing accuracy over time with tuning. As the robot gets more training and an increasing data collection, its result could potentially increase to a certain success level over time. This approach's drawback is the potential amount of data and time needed during data collection and training before reaching a certain level of success rate.

One approach by [11] uses CNN to enhance the choice of grasp on a given point cloud. They first segment and simplifies the point cloud to a more basic shape, like a cylinder or sphere, before proposing grasp strategies dependent on the simplified shapes used. These are since fed through a CNN with up to 150k dataset. They claim the learning time for such a dataset size would take up to 40 minutes. Their approach has a low need for data and learning compared to others with similar approaches.

[14] are proposing the use of ML for finding grasp poses in a point cloud of objects presented in a clutter. Their approach starts by identifying a set of conditions needed for generating a set of grasp hypotheses, which focuses the grasp detection into more feasible regions in the point cloud. Since followed by creating a training set labeled automatically using grasp geometry. Because of this approach, could it label a vast amount of training data without the aid of a user, automating the training process.

The methods and approaches used are also dependent on equipment and strategies. [10] has opted for using a RGB camera with a static view over the grasping work area. Their self-supervised reinforcement framework is based on the vision-data training a deep neural network to perform closed-loop real-world grasps. This framework could cope with over 580k real-world grasp attempts. With their closed-loop approach, have they achieved a more dynamic grasp process able to evaluate the grasping process during its execution. The robot would try to adjust or re-grasp until the object is grasped. If the available grasping points are granted a low score, could it push or move an object to find a more feasible grasping point.

Learning-free Approach

A learning-free approach is analyzing the data given from a set of sensors or equipment, calculating how to interact with the object of interest. A learning-free, or analytic, setup could execute a task without being learned or told how to grasp an object but mainly act on the given data registered regarding the object's structure and work environment. The advantage of a learning-free system is that it can potentially be placed and integrated quickly into an environment, primarily relying on input data to execute tasks. This approach calculations might be computationally heavy, depending on the data collection analyzed and how many grasps or solutions are evaluated.

A former approach for grasping objects could consider both the object's surface as well as the grippers and their interaction to compute force-closure [12, 2, 26, 18]. This would rely on given data and known factors in advance to increase the likelihood or to be able to execute a successful grasp of the given object. These data could consist of the object's friction coefficients, mass distribution, and weight. Such an algorithm could from this calculate the force needed from the gripper and its fingers to grasp and hold the object. A similar approach is also used with form-closure, which forms the gripper's fingers to hold the object [25, 1]. Both of these approaches would require that the system knows some data of the object. In an application grasping previously unseen objects, are there little or no knowledge about the object at hand and would need a model-free approach for adaptive execution.

By analyzing the surface characteristics and curvature given as a point cloud, cloud the gripper's fingers be projected onto the surface of the object to calculate and find a suitable grasp. An algorithm doing this, called Local Contact Moment (LoCoMo) [3], would only need a predefined model of its' gripper and the surface of the object given as a point cloud. Their setup can grasp various shapes without previously recorded data of the object mass or attributes. With a wrist-mounted depth camera, they could scan the object, creating a point cloud to analyze and find a feasible grasping location.

Compared to LoCoMo, are the approach by [16, 4] instead using a single view point cloud to gather data from the objects. This creates a more straightforward approach since they do not rely on a complete 3D representation, thereby not needing to scan the complete work area to analyze and develop a grasp proposal. Their grasps are generated by creating a cutting plane perpendicular to the object's main axis and center. By this approach are most homogeneous objects center-of-mass close to the center of the grasping area, thereby increasing the control of the object during lifting and interaction. Though with a single viewpoint, they experience some vulnerability regarding exposure to noise and view angle, affecting the result.

An approach proposed by [17] uses a framework for finding grasps in a 2.5D point cloud by combining the use of laser range and stereo data. The grasping points are calculated based on the convex hull points obtained from a parallel plane to the top surface in the height of the object's visible center. This approach gave a more stable and sustainable success rate, but [17] experienced a problem of picking up, for instance, a bowl larger than the gripper since their approach considers the hole object, instead of picking it up by its rim.

Proposed Method

The task of grasping previously unseen objects is a challenging problem. Several approaches and research are made towards solving and improving this, though no method stands out clearly as a preferred approach since each method has its advantages and disadvantages.
This project will look into an analytic approach that could be complementary to LoCoMo. The contribution aims to address the limitation for this algorithm and purpose improvements, which could increase the results. The aim is to develop a virtual environment for testing and developing point cloud extraction and preparation for an algorithm such as LoCoMo. This environment could provide a foundation for further development to build upon in applying the proposed approach.

1.5 Preview of Chapters

The thesis is composed of eight chapters. The chapters and their contents for the rest of the thesis are as follows:

Chapter 2: Theory

This chapter presents the theory used for calculating and setting up the environment's functions.

Chapter 3: Method

The method covers the equipment used, its setup, and what the virtual environment consists of.

Chapter 4: Program Execution Structure

This chapter goes through the scripts created and explains their execution. This illustrates the flow of the project and how the virtual environment works.

Chapter 5: Test Setup

This chapter covers how the tests has been performed.

Chapter 6: Results

The result of the test conducted in this project is presented.

Chapter 7: Discussion and Future Work

Discussion elaborates both the result and the project in general and the future work for this project.

Chapter 8: Conclusion

Chapter eight includes the conclusion for the project.

Chapter 2

Theory

This chapter contains theory to give insight into the field touched upon in this project. This will cover the vision for the robot, algorithm's calculations, functionalities, and kinematic for the robot. This gives a deeper insight into applications of relevance toward this project and for consideration for further work.

2.1 Vision

The use of sensors can give a robot vital input regarding its environment, affecting execution, operations, assessment, or navigation. By implementing vision into a machine or robot, can information regarding a task be extracted for solving a task and enabling the interaction for the local environment [9]. With a form of vision, can a robot use the input of the given image to evaluate and execute a given movement or task, interacting with the environment more dynamically.

Point Cloud

A point cloud consists of several data-points in a 3D coordinate system $P_n = [x_n, y_n, z_n]$. These points could be created either with the aid of a software program, converting a model into a distributed cloud, or using equipment like LiDAR (Light Detecting and Ranging) or depth-camera, which create a cloud of points dependent on the detected surfaces in an environment. Additional information regarding each point can also be stored depending on the sensors used, like color value or surface temperature. With one or several point clouds taken, a model of either an object or area can be mapped and used to recreate a 3D representation.

Filtering

Since a raw, untreated point cloud does not contain defined information of, for instance, what points belong to an object or which are defined as the floor, must the data from the cloud be treated depending on its features to extract information or regions of interest. To reduce computational loads, reduce noise and extract the information of interest, could filters be used. By applying filters to separate and extract points from a cloud, could different processes analyze the relevant data for their specific operation.

For a defined range of interest or a static work environment, could the cloud be filtered by coordinates or depth. This would return only the points within the given threshold. This filtering could also apply to the point's additional attributes, like temperature or color. Even extracting a limited region could still result in a vast amount of points and data to analyze and could consist of an uneven distribution of points dependent on the detection of surfaces. The uneven distribution could affect further analyzes and calculation of the cloud. One method to create a more evenly distributed point cloud is by using a Voxel Grid filter. This filter applies a grid of boxes in the point cloud. Within each box or voxel, one point is represented as an approximated center of that group of points. This will create a more evenly distributed point cloud.

2.2 Search Trees

A search tree is a tree data structure. With the tree structure, can the data collection be divided into several layers of nodes and sub-nodes. These nodes can represent a data point or field, and their sub-nodes are the values either higher or lower than the given node. An example of this is shown in Figure 2.1.

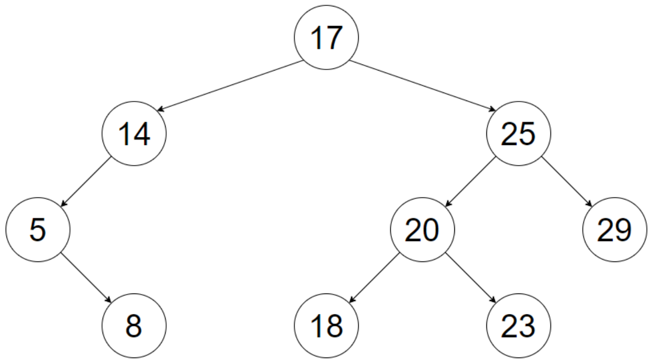


Figure 2.1: Search three

By dividing the data into subsections, can the search time for a given node or information be decreased. Instead of searching through the whole data set for some given information, can the search go through the branches towards the data of interest.

One form of such a search tree is an octree. An octree divides the search field into cubes, or octants, similar to a search tree divides the data within nodes and sub-nodes. Where there are no data, are the square not divided. Where there are data are the cubes divided into eight sub-nodes. This continues similarly to the search tree. This method can also decrease the data storage necessary to represent the structure or picture at hand since the nodes with no sub-nodes could represent a large area and decrease data needed to describe it compared to dividing the whole area with a small fixed size for its representation. An example of how an octree divides are shown in Figure 2.2. Here a cube with nodes and sub-nodes are illustrated on the left side, and a search tree representing the cube on the right.

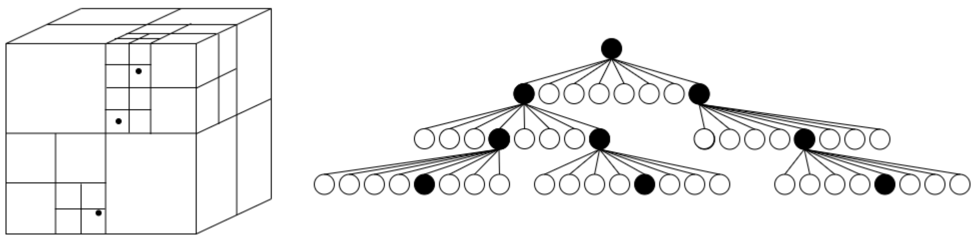


Figure 2.2: Octree example.

2.3 Robot Kinematics

To navigate, coordinate, or move a robot is a coordinate system needed as a reference frame to act from. Often are a system defined by several coordinate systems with equipment and task execution reliant on these.
A robot arm has most often one static coordinate system, defined at its' base. An example of how a robot arm could consist and be defined by several coordinate systems can be seen in Figure 2.3. Here is the coordinate for the robot's base (B), joints, camera (C), and gripper (G) shown. The view from the camera is defined relative to the camera's coordinate system, C. To act out from the given information, could the data from the camera be transformed into a global coordinate system, like B. In this case, could the data be used to describe a location of an object. This data could be used for the robot and gripper to interact with that given object.

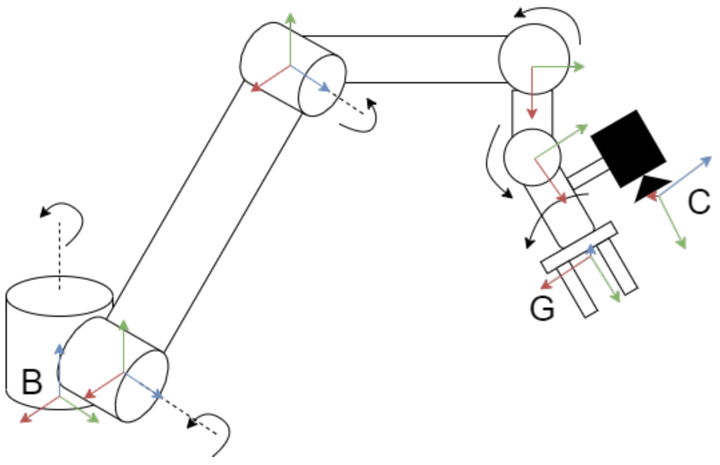


Figure 2.3: Eye-in-hand kinematics

To transform between two coordinate systems are two matrices used, one for translation and one for rotation. The translational matrix (**T**) is a vector used to describe the distance between the origins of each coordinate system in a $3x1$ matrix. The rotational matrix (**R**) is used for rotating a coordinate system to alight its axis accordingly and consists of a $3x3$ matrix. These two matrices can be combined into a rotational-translational matrix, **RT**, as shown in Equation 2.1.

$$\mathbf{RT} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

With a RT-matrix, could a point seen from the camera frame, C, be described relative to the base frame, B. This is shown in Equation 2.2. Here the point registered in the camera frame, p_c, is translated into the base frame B as p_b.

$$p_b = \mathbf{RT} \cdot p_c = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} \tag{2.2}$$

2.4 Zero Moment Shift

Zero Moment Shift (ZMS) is used to calculate the surface characteristics, in this case, the curvature of the surface of an object. The point cloud of an object, χ, is analyzed by each point, X, and a defined sphere with radius, ρ, around the given point X. This sphere is represented as $B_\rho(X)$ and the set of points it contains as ξ. This gives the expression $\xi = \chi \cap B_\rho(X)$ [3].

The Zero Moment $M_\rho^0(\xi)$ is obtained by calculating the average coordinates of the set ξ, where N is the total number of points in the set and X_i are one sampled point from ξ. This is shown in Equation 2.3.

$$M_\rho^0(\xi) = \frac{1}{N}\sum_{n=1}^{N} X_i \tag{2.3}$$

The ZMS is then defined as the distance vector n_ρ between the Zero Moment $M_\rho^0(\xi)$ and the given point X.

$$n_\rho = M_\rho^0(\xi) - X \tag{2.4}$$

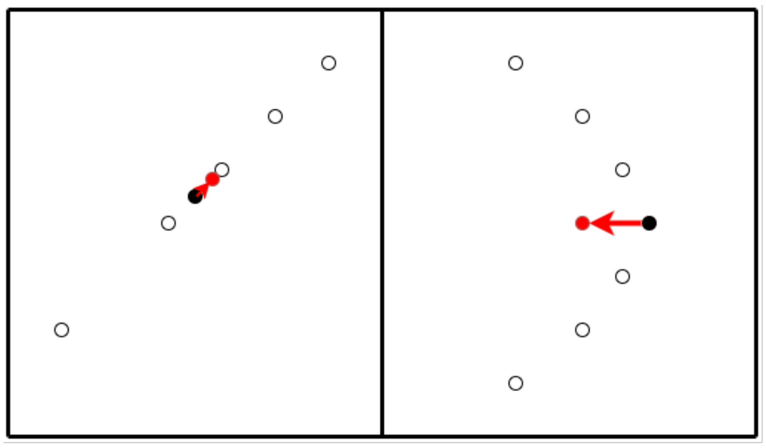


Figure 2.4: ZMS illustration

These vectors define the surface curvature of the point cloud. If the given n_ρ in an area contains high values, i.e. long vectors, the surface would be interpreted as more curved. This is because the average coordinate in a given $M_\rho^0(\xi)$ are further away from the current point X than in a more flatter surface that would result in a shorter vector n_ρ, assuming the distribution of points on the surface is even. An example of this can be seen in Figure 2.4. The two squares illustrate a set ξ, with the search point X as the black point, and $M_\rho^0(\xi)$ as the red point. The vector n_ρ is illustrated as the arrow between $M_\rho^0(\xi)$ and X.
These data are affected by the radius, ρ, of the sphere. Dependent on how big the radius ρ is from the center point X, will include more or fewer points in the calculation.

2.5 Local Contact Moment

Local Contact Moment (LoCoMo) uses ZMS to analyze the surface structure and score dependent on the likelihood of grasping success. A guiding factor for this are L1 $= |n_\rho|$, which gives a characteristics view of the given surface [3]. By using L1, can the curves and surface characteristics of the object be distinguished. Further, could this be used to compare the surface of the object to the surface of a gripper or its' contact patches. These surface probability of fitting each other are further ranked through a probability function, Equation 2.5, first introduced and purposed in [3].

$$C_\rho = 1 - \frac{max(x, \phi(x, \overrightarrow{0}, \sum)) - \phi(\varepsilon, \overrightarrow{0}, \sum)}{max(x, \phi(x, \overrightarrow{0}, \sum))} \tag{2.5}$$

Where ϕ represents the multivariate Gaussian density function, shown in Equation 2.6, which contains $X, \mu \in R^n$, n as the space dimension and $\sum$ as the covariance matrix. The multivariate Gaussian density calculates the statistical distribution in n-dimension.

$$\phi(x, \mu, \sum) = \frac{1}{\sqrt{(2\pi)^n |\sum|}} \exp\left(-\frac{1}{2}(X-\mu)^T \sum^{-1}(X-\mu)\right) \tag{2.6}$$

$$\varepsilon = n_\rho^1 - n_\rho^2 \tag{2.7}$$

ε represents the deviation or surface difference between the gripper, n_ρ^2, and the object, n_ρ^1, given in the same reference frame. This deviation are calculated using Equation 2.7. If both surfaces have similar shapes, would their vector be similar, resulting in a small error value after the subtraction. If the surfaces differ more from each other, will the result be a higher value ε. An illustration of this is shown in Figure 2.5. By ranking and checking the differences between the gripper and object surface, can a feasible grasp location be found and increase the likelihood of a successful pick.

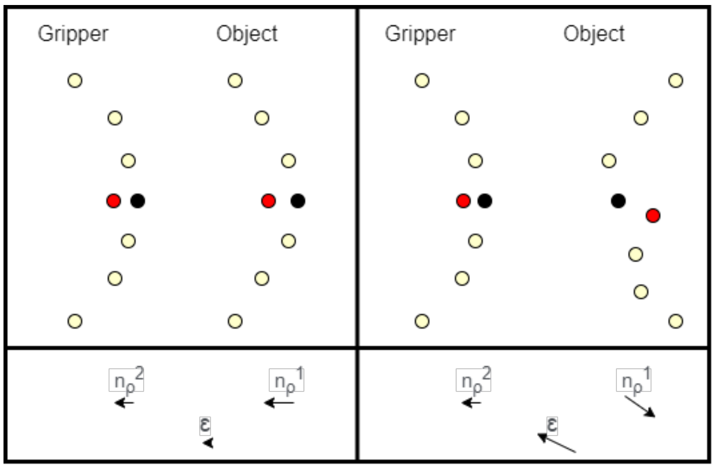


Figure 2.5: Surface Fit example

2.6 Grasp Handling

With the characteristics of the point clouds from both the gripper and the object can the process of finding a suitable grasp begin.
In Equation 2.8 the probability of contact between the surface of the object and each finger of the gripper is presented. Here N_S represents the given points in the point cloud near a finger of the gripper, and n is the number of points near a finger in the object point cloud. $C_\rho^{i,Xi}$ is the probability of local contact moment between a point, Xi, from the object projected perpendicularly onto the surface of the gripper. C_i gives the contact probability for each finger, found by using Equation 2.5 [3].

$$C_i = \frac{1}{N_S} \sum_{i=1}^{n} C_\rho^{i,X_i} \tag{2.8}$$

To select a grasp with high plausibility of success are the grasps ranked with the use of Equation 2.9. Here k is a normalizing term, n_f as the number of fingers on the gripper, ω_i as a weighting factor that satisfy $\sum_{i=1}^{n} \omega_i = 1$, giving the ranking value R.

$$R = k \prod_{i=1}^{n_f} C_i^{\omega_i} \tag{2.9}$$

With the grasps ranked from high to low plausibility for success, could the top grasps be analyzed further for final evaluation before execution of the most suitable grasp.

2.7 Safety

With the implementation of equipment that brings along potential dangers and hazards must barriers, cautions, or other safety measures be implemented and held along with it. This is to secure humans, equipment, and other assets or values that might be exposed to these dangers. When implementing and running a project in a physical environment, there is a potential risk where humans and the robot could work in the same environment simultaneously and handle objects or equipment. These cases should follow the safety standards that apply accordingly.

ISO 10218

The ISO 10218 is a standard regarding safety implementation for industrial robots and consists of two areas in focus: *Robots* and *Robot systems and integration* [7]. This standard was created when the risks with robots and their working environment were recognized and are meant as guidance and precautions for construction of both the work area for the robot and what measures are needed to be held to obtain a satisfactory level of safety.

ISO 10218:1 (Robots)
The first part goes into detail about industrial use, controller and manipulators used to control a robot, and what functions must be implemented. Before implementing the robot in an environment, there should be taken a risk analysis consisting of points like the robots planned operation and their scenarios (maintenance, adjustments, cleaning and similar), unexpected shutdowns or startups, the availability of the robot for people, the outcome of the failure of this robot fully or partially and other hazards regarding the robot type, its tasks and what material it is handling. Dangers should be limited or lowered to acceptable levels for the robot to be implemented.
During the robot operation, the standard also sets an upper limit of what force the robot could work or execute to limit the impact force if a situation should occur where a human might get into the robot's path. The robot should also have some measures or emergency stop function to stop it either from its execution standpoint or activated by a human when an operator sees a potential risk.

ISO 10218:2 (Robot systems and integration)
This part goes further into tools used, workpieces, periphery, and safeguarding in the environment of the robot. It specifies the significant dangers and hazards in different setups and safety requirements within the corresponding scenarios. These requirements to lower risk or hazard could be either reconstruct or change of work area or the implementation of, for instance, barriers, surveillance during operation, signs, and other visual safety measures. It also goes into environmental factors that can affect the robot.

ISO TS 15066

This standard builds upon the ISO 10218 standard and goes further into details for risk analysis and use of cobots where robots and humans will interact [8]. Its focus is to reduce risk during operation, power and efficiency output from the cobot that can cause harm and lowering potential hazard levels to an acceptable level, and implementing routines to aid the workers. This can secure the safety of the working environment around the cobot and aid with combining the speed, force, and precision from the cobot with the skills, flexibility, and problem solving from a human.

Chapter 3

Method

The programs and equipment used during this project are listed in this chapter.

3.1 Hardware

The equipment used in the project was provided by the university and the Mechatronics Innovation Lab (MIL). The equipment consisted of a computer, a depth camera, and access to a robot arm.

The computer provided and used in this project was fitted with an Intel I5 660 3.33 GHz dual-core CPU, AMD 5770 1GB 850 MHz GPU, 8 GB RAM, and an SSD. The operating system installed was Ubuntu 18.04.05, since this operating system had support with ROS melodic morenia, which could use several open-source packages available relevant to the equipment used or available.

3.1.1 Robot

The robot used in this project is produced by Universal Robots, which create collaborative robot arms to integrate and solve tasks in industries, packing, educational use, and more. Their product line consists of 4 robot arms with 6 DoF (Degrees of Freedom) in different sizes. The one used in the virtual environment in this project, UR5, is the second smallest arm in the series. With a range of 85 cm and capable of lifting payloads up to 5 kg, it is suited for a variety of tasks. The tools for the robot are mounted on its end, also known as its end effector, and can be changed depending on the task at hand [24]. This robot model was chosen because the university has access to this model, making it suitable when the project is continued or used as a foundation for related work.

3.1.2 Camera

The camera used in this project was an Intel RealSense D435i. It is an RGB-D camera able to create a point cloud in the range 0.2 m up to 10 m, dependent on light condition. It also has an IMU (Inertial Measurement Unit) to measure up to 6 DoF. The RealSense D4-series supports implementing and running with ROS, making it easy to integrate and use in such an environment. The mounting of the camera affects the results and approach for solving the task. A fixed-view scan position with a depth camera could result in shadows and blind spots in the point cloud, dependent on the object and its position. This could lower the success rate of grasping an object since features in the blind spots are not accounted for. To improve this, could a set of basic geometric shapes be used to describe the object in view [9]. Nevertheless, as mentioned, could additional features be in the blind spots, and with a model-free learning-free approach, would a static point of view not be as sought after because of this. By attaching the camera onto the robot's end effector, giving a dynamic viewpoint or eye-in-hand setup, an object can be scanned from several angles, thereby adding more information on its shape and decrease potential blind spots. This approach would be beneficial since additional data are gathered on the object shape and increase the likelihood of finding a feasible grasping point, and were therefore chosen.

3.2 Software

To control the equipment used and sensor data are several different software used and applied to control the robot safely, solve the task at hand, and aid in the project execution. The mainly used programming language for this project was python. This was because of the student's previous experience in python compared to the alternatives.

During the project was Visual Studio Code (VSC) used for coding and file structure overview. VSC has an intuitive graphical interface regarding file tree used to navigate in the development folders and the additional plugins installed to aid in coding, structures, troubleshooting, and functions. As a safety measure for not losing the developed files and code was GitLab used to take backups of the project folders, saving them online, and keeping track of the software development. The file structure included in the backup was the package repository in the ROS-environment, where most developed codes were stored. GitLab also gives an overview of use and changes, making it a tool for keeping track of development and progress when used in a team. Several developers could work on the same project in their branches to secure progress, keeping an overview of the structure and acting as a safety measure. One could also roll back to previously developed versions if needed. As an additional safety measure were several images created of the computer running the virtual environment.

3.2.1 Robot Operating System

The Robot Operating System (ROS) is an open-source developing environment to develop, simulate and program equipment or robots to control and adapt them to a given task [19]. Because of several developers' contributions are many different packages and programs available and can be downloaded and integrated to enhance a robot's function, such as use sensors or other data to execute a given task. This gives ROS a modular approach. The packages applicable could be downloaded and implemented instead of downloading an extensive program where one would only need a minority of its functions. One example could be integrating an IR-sensor to assist a mobile robot from hitting obstacles or walls while driving or through camera recognition to recognize objects. The mainly implemented and supported program languages in ROS are C++, Python, and Lisp, which makes this accessible for a wide range of developers [20].

ROS

Figure 3.1: Robot Operating System [19]

ROS are regularly creating new versions or distributions, implementing and adapting functionality towards the current distributions [20]. The version used during this project was ROS melodic morenia, which was the second newest version at the project start. This distribution was chosen because of the available packages for the Universal Robots and camera RealSense. The support time was to 2023 before EOL (End-Of-Life), and had more available information on forums in the community compared with the newest version at the time, Noetic Ninjemys. ROS builds upon some functions and uses that are covered in the following subsections.

Node

Processes in ROS are called *nodes* [20]. These nodes could be calculating and computing individual tasks that could since be used to control a robot. In this project, one node can be storing the data from the camera for use in another node, which could estimate the surface features. By splitting the process into several programs, or nodes, can more be done in parallel, thereby increasing efficiency. It also helps with the troubleshooting where a problem can be listed down to one or some specific nodes instead of in one big program. The nodes used in this project are shown in Table 3.1. An additional map of the communication between the nodes during execution can be seen in Appendix I.

Table 3.1: Node list

Node name	Function
/assemble_voxelgrid_applied_pointcloud	Second voxel grid filter package.
/gazebo	Gazebo application.
/gazebo_gui	Graphical User Interface for Gazebo.
/my_assembler	Assembler used for point cloud merging.
/pcl_manager	First voxel grid filter package.
/robot_state_publisher	Publisher for transformations for the robot.
/scan_trajectory_pub	Scanning procedure script.
/voxel_assembled_cloud	Second voxel grid filter for filtering assembled cloud.
/voxel_grid	First voxel grid filter extracting points above workbench.

Topic

A topic is the message containers used to transport data between nodes [20]. These nodes can subscribe or publish data to different topics, which could be used in a process or task. The topics are not bound by any node, and can therefore be used by several nodes, thereby increasing the streaming of communication. Topics used during this project are shown in Table 3.2. Additional overview of the topics traffic during execution are shown in Appendix J.

Table 3.2: Topic list

Topic name	Use
/arm_controller/command	Setting commands for joints and position for UR-robot
/assembled_pc_scan	Assembled point cloud from second voxel filter.
/camera/depth/points	Points published from camera.
/camera_PC_transposed	Transposed point cloud from camera- to base-coordinate system.
/tf	Transfer function for mobile coordinates.
/tf_static	Transfer function with static relation coordinates.
/voxel_grid/output	Filtered cloud from camera.

Master

The master, or roscore, keeps track of all active nodes, topics, services, and addresses.
For a node that publishes to a topic, it will give its address to the roscore. By doing this, will the roscore keep track of the active topics.
When communication for a node is needed, the node will first contact the master, which provides the address of the node publishing on the topic of interest. Since the subscribing node now has an address to the node publishing on the given topic, it will use this address and connect for the message subscription so that the communication stream will go directly between the nodes.

Packages

Packages in ROS contain programs, functions, nodes, libraries, or other sorts of data that can be implemented to solve or add value to a ROS environment [20]. With the created packages for distribution in the community, does it improve and make it easier for other users to include and expand their environment to solve their problems at hand. This modular approach combined with the open-source environment makes the ROS environment a popular and good alternative regarding making, building up, and using equipment and robots for projects and tasks, as long as the equipment is supported or capable.
The packages used during this project are listed in Table 3.3.

Table 3.3: ROS packages used

Package for	Package name	ref
UR5	universal_robot	[23]
Intel RealSense camera	realsense-ros	[21]
Camera Gazebo Plugin	realsense_gazebo_plugin	[5]
Created environment	ur5_env_description	

Filter

In this project are voxel grid filters used to treat the data from the cloud. This will create a more evenly distributed point cloud and lower the number of points in the processed cloud. Reducing the number of points to process could increase the speed of the program's overall performance further on in the process since less data will be handled. Although this will decrease the points to analyze and calculate, it will also remove some of the surface features, leading to loss of information. The data removed will depend on the grid size applied to the point cloud. One advantage for creating a more evenly distributed point cloud would be where the point cloud is further analyzed by its' surface distribution and characteristics. Having an uneven distribution might affect some areas' surface evaluation and weighting more than others because of a denser region of points.

A voxel grid filter was chosen because of its efficiency compared to other filters, as well as PCL and ROS already include support and availability for it. [20, 27]. Using the supported voxel grid filter in ROS, one could also define the cut-off area in the cloud. In a fixed working area, this could be used for extracting only the region of interest. The voxel grid filter could reduce noise and even out the distribution of points, leaving a more refined point cloud.

3.2.2 Gazebo

In Gazebo, a virtual work environment can be created for testing equipment and robots without a physical setup. Gazebo was chosen because of its compatibility with ROS and the ability to simulate sensor inputs and behaviors.
The advantage of using a virtual environment for development and testing is no risk of damaging any equipment or expose people to any danger. Bugs and faults could be caught early and corrected before implementing it onto a physical setup, lowering the risk. Another benefit is that several developers could use and develop in a local virtual environment in parallel, potentially lowering the development time. It is a good practice to use a virtual environment both for developing and testing. However, a virtual environment alone can not guarantee the behavior and performance of a physical system. Inputs from sensors and equipment, for instance, could have varying degrees of disturbance, deviations, or errors, as well as how accurate the simulation tool is to represent the work environment could affect the outcome. A good practice is to take advantage of both approaches by starting and developing the system in a virtual environment before implementing it onto a physical environment and fine-tuning the system and its parameter to satisfy the given requirements.

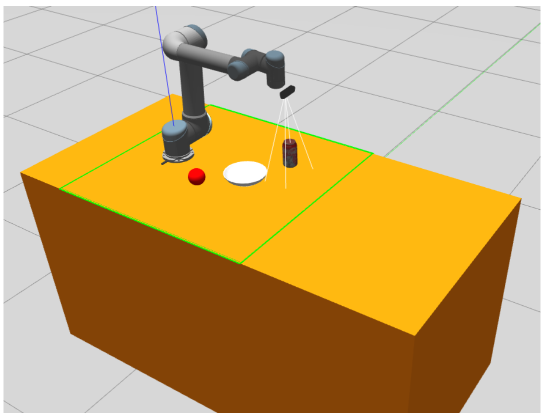


Figure 3.2: Gazebo environment

The gazebo environment was set up with a UR5 mounted on a workbench. The camera was mounted to the robot's end effector to collect data from objects placed in front of the robot. The object must be within a range for the camera to gather and merge the object's surface data. The environment is shown in Figure 3.2.

3.2.3 Point Cloud Library

The Point Cloud Library (PCL) is a collection of algorithms for processing point clouds [22]. The PCL contains functions for filtering and manipulating point clouds, recognizing or estimating cloud features, recognizing shapes of objects, and more. PCL is open-source software that is free to use and comes included in ROS. PCL is based and created in C++, although other versions exist of the libraries wrapped or converted to suit other program environments, such as python.
In this project was a library called *pclpy* used. This python library converts and wraps the original PCL-library from C++ to a python library with similar wrapping and functionality, by applying another library, pybind11. Though other python libraries deliver similar functionality and conversions, might their approach need other tools or programs that can increase the complexity and code needed for maintaining the library. pybind11, which pclpy uses, is a headers-only lightweight library used for conversion between python and C++ [15]. The implementation with a python environment was chosen because of the student has some earlier experience with python compared with C++.

3.3 Surface Observation

The robot moves in a predefined scanning pattern so the camera can collect the object's surface attributes and data from each given pose. Collecting data from several view angles gives more features and visible surfaces for analyzing, thereby decreasing the blind spots for the object. In this setup are the scanning poses predefined at six locations for gathering data. These six poses were chosen because of the combination of coverage and time of execution. A picture capture position every approximately 60^0 around an object resulted in good coverage and surface detection. An increasing number of poses for detecting surfaces would increase the time for execution and were therefore not increased further when good coverage was reached. The scan poses were chosen to give an overview of the workbench better to detect surfaces in case of a clutter of objects were to be analyzed. A view of the poses is shown in Figure 3.3.

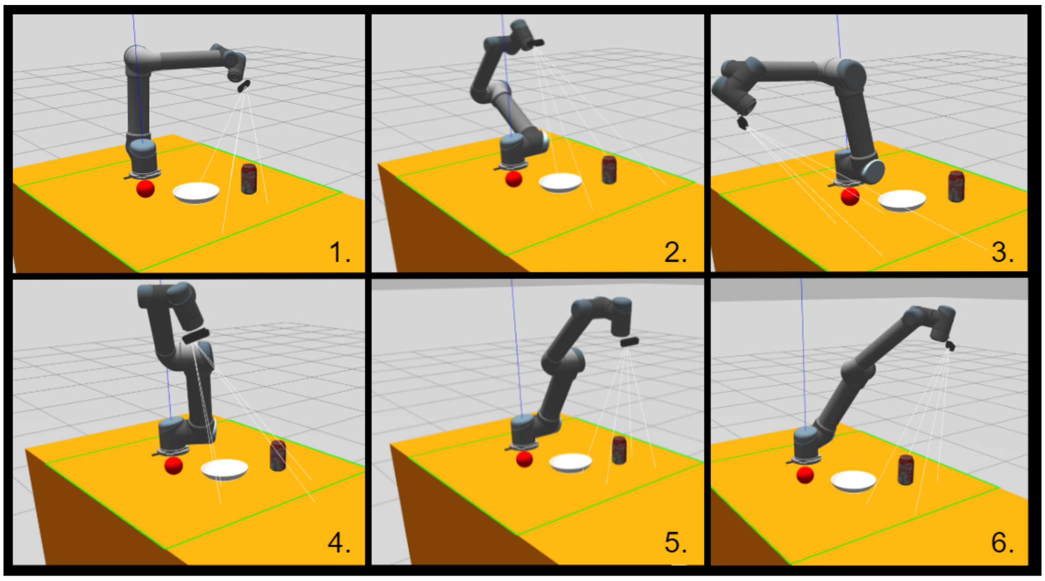


Figure 3.3: Scan pattern

The robot movement is set up for static analysis for scanning the workbench, meaning both the trajectory and poses are predefined for the task of scanning. This approach was taken since the object are assumed to be within a defined work area for scanning and further interaction. It is also assumed that the size of the objects would correspond to a feasible gripper to a UR5 and the robot's capabilities, which are up to 5 kg. The range of a gripper's maximal stroke are assumed to be between 85 to 155 mm, dependent on model used [24]. The poses were set to a generally high position as a precaution for decreasing risk of colliding with objects as the robot moves over the workbench, keeping the objects past the cut-off distance for the camera, as well as getting an overview of the objects.

3.4 Point Cloud Construction

The clouds from the scanning trajectory are merged and filtered to construct a point cloud of the object. The point cloud is registered relative to the camera's coordinate system, *camera_depth_optical_frame*. Changing the camera position would change the view in the point cloud. The points would change, and trying to assemble a point cloud with a reference in the camera frame would only overlap the points registered since they are observed in its dynamic frame and not translated to a global, static frame. By finding the rotational-translational matrix between the *camera_depth_optical_frame* and *base_link*, could the points registered by the camera be presented relative to the base frame. When the camera moves to a new position and registers new points, these points would not overlap but create a 3D representation of the object. The rotational-translational matrix is found with the use of the function *lookupTransform* in the pclpy-library. This could since be used to translate the points into the base_link frame.

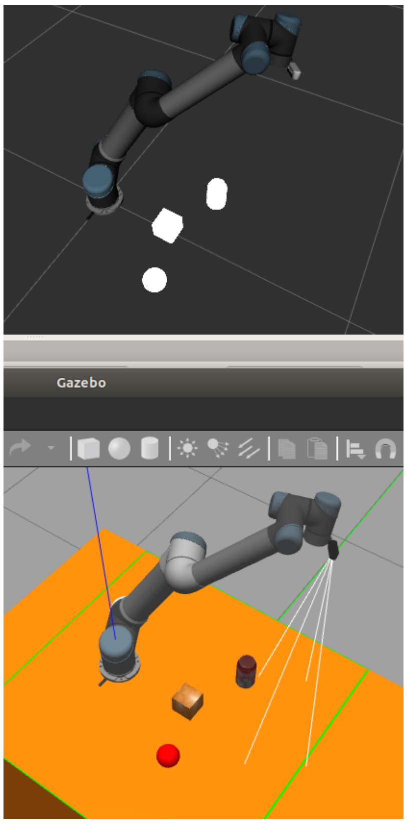


Figure 3.4: Gazebo (bottom) and Rviz (top) simulation

When the points are given relative to a static coordinate system, is it more evident where the object is placed in the work environment, which could be used to define where the robot should go when interacting with it. After the point cloud is transposed to *base* are a voxel grid filter applied to remove the points that are below or equal to the top of the workbench. The points left are for the object. To assemble the different clouds into one are a pre-installed library *laser_assembler* from ROS used. It subscribes to the topic from the voxel grid filter and collects the data it broadcasts. Since all the clouds are defined in the same coordinate system, base_link, will the assembler create a 3D representation of what is placed on the workbench, as shown in Figure 3.4. This point cloud is since published to a new topic, where a new voxel grid filter is subscribed and collects the cloud for a final adjustment before the cloud could be stored. By applying a voxel grid on the assembled cloud, could the overlapping points and scanned area be treated to make the point distribution more consistent. The second voxel filter is also removing the points belonging to the base of the robot, which is seen in some scanning poses during the cloud capturing.

A more unified and evenly distributed point cloud created would be more suited for calculation and assessment. A cloud that is uneven or consisting of clutters with a varying density of points could affect the calculations carried out of its surface, potentially influence the treatment, and result in interaction with the object.

This process ends with a point cloud representing the registered surfaces for the object placed on the workbench. These are since stored in a file structure, both for archiving and analyzing purposes. An example of such a cloud is shown in Figure 3.5.

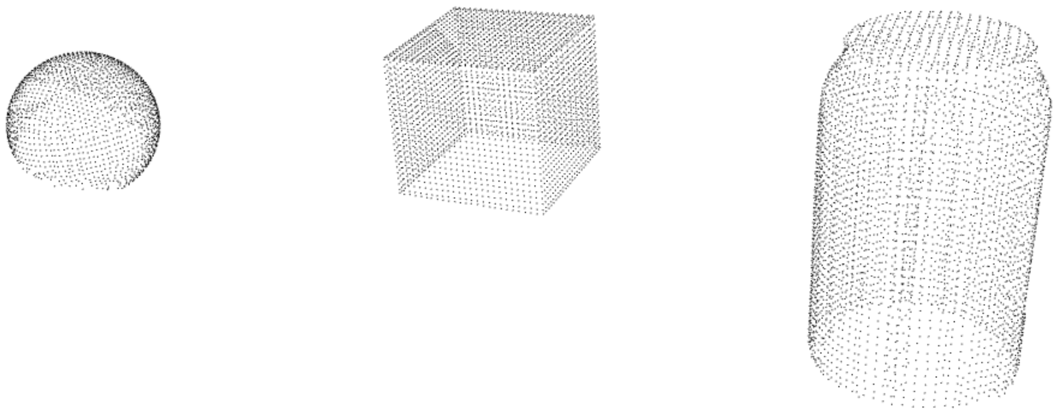


Figure 3.5: Constructed point cloud

3.5 Zero Moment Shift

The point cloud of the object is read from a pcd-file for calculations. A python script, *zms_algorithm.py*, is used to conduct the calculations. The python script uses a pclpy-library for handling and calculations. The calculations done to the specified cloud are stored in new files for further assessment. The ZMS-calculations are stored in text files, *.txt*, in $[x, y, z]$ coordinates. This could be used by a LoCoMo algorithm for surface curvature, which would get L_1 by the absolute value of ZMS's n_ρ. A second pcd-file is also created to illustrate the ZM of the surface better than an array with numbers. Here one could observe the curvature attributes calculated of the given point cloud. An additional pcd-file is stored to better visualize the effects of defining a set ξ has on the ZM, $M^0_\rho(\xi)$.

Chapter 4

Program Execution Structure

This chapter goes over the structure, execution, and scripts of the given project explained from start to finish. A flowchart showing the corresponding process is shown in Figure 4.1, as well as an overview of node and topic communication for the running environment are shown in Appendix I and J accordingly.

The virtual environment is defined in a urdf-file, shown in Appendix A. To start the virtual environment, one must execute a launch-file, *UR5env.launch* shown in Appendix B. This file launches the environment in Gazebo, imports the used model for the UR5 and camera, and launches the filters used for handling the point cloud. When the environment is fully launched in Gazebo can an object be placed on the workbench. Gazebo has a built-in library with a wide variety of objects which can be imported and placed in the environment by coordinates.

With an object placed on the workbench, can the scan procedure be started. This can be started with a *rosrun* command. The script controlling the scan procedure is *Scan_trajectory.py*, shown in Appendix C. To see the published data of the topics of interest, can a program called *Rviz* be launched additionally. Rviz is a program with support for ROS applications used as a visualization tool. [20]. In Rviz one can import the robot model used in Gazebo, coordinate systems, topics, sensor data, camera view, point clouds, and more for visualizing the data. Rviz has also been used to troubleshoot during the project since data can be visualized easily by importing and subscribing to their topics and analyzing the given data.

Scan_trajectory.py is a python script which are launched with the command
rosrun ur5_env_description scan_trajectory.py. This script defines the poses for the robot were to take a depth image from and schedule the execution of taking the depth image, the transformation, and assembly of the point cloud. The position is defined at the start of the script as the angle of each joint in radians. Following are a for-loop started where each stage is executed in order. The robot will first move to a home position where the end effector will be placed above the scanning area. A timer is since started before the robot moves to the first scanning pose. The use of a home position gives some predictability for the system, as the users could know where the robot could start and end its trajectory. When the robot reaches the position for the first depth-picture are a if-loop initiated for controlling the call of functions. The first function initiated in this loop are from another script, *camera_PC_transform.py* shown in Appendix E. Here it subscribes to the topic */camera/depth/points*, to collect the cloud streamed from the camera. Since are a transpose from the camera's coordinate system to the robot's base coordinate system checked and executed. From this is a transformation matrix calculated. With these matrices is the point cloud from the camera's coordinate system, */camera_depth_optical_frame*, transposed into the robot's base coordinate system, */base_link*. The transposed point cloud are since published to a new topic, */camera_PC_transposed*.

A voxel grid filter, shown in Appendix D, are subscribed to the transposed point cloud topic. It receives the point cloud and filters it. Since the filter are defined in relation to the base_link coordinate system, will the height in the point cloud be corresponding in the z-direction. The filter defines the range of interest in one of the axes, which are from the workbench's surface and upwards. The filter also distributes the points more evenly before publishing them on a new topic, */voxel/output*.

After the transform script has been initiated in *scan_trajectory.py* is a wait function used to hold the robot in position until a cloud is published from the transformation. This is done to prevent affecting the transformation between the coordinate systems. The processing of the clouds is dependent on the data registered in field-of-view (FoV).

When a message is received that the transformation is done, will the script *pc_assembler.py*, shown in Appendix F, be initiated. This script starts an assembler-function in ROS and passes the subscribed point cloud into a builder. Here are the clouds collected and built in a common coordinate system, which here is */base_link*. The assembled point cloud are since published to the topic */assembled_pc_scan*. A second voxel grid filter, shown in Appendix G, is subscribed to this topic and filters the cloud based on the base_link x-direction to remove the points corresponding to the UR5's base, which are detected in some scanning poses. Also by using this second voxel grid filter are the overlapping points merged since some surfaces can be registered from several view-angles and are placed on top of each other by the assembler. The filtered point cloud are since published on the topic */voxel_assembled_cloud*.

The *scan_trajectory* script is since starting the iterating for-loop again with the following position scheduled. As the system executes the given commands, is a point cloud built with data collected from several view angles, giving a more detailed view of the object placed on the workbench and decreasing the blind spots. When all the depth-pictures are taken from each position will the robot return to the home position. The script will then print a message with execution time for the scanning before it stops.

For storing the point cloud of the object to a pcd-file are a command executed manually. The command, "*rosrun pcl_ros pointcloud_to_pcd input* := */voxel_assembled_cloud*", are supported by the ros-library *pcl_ros*. Running this command will store the PointCloud2-message received from a given topic, in this case, */voxel_assembled_cloud*. The file would be stored in the file structure where the command window is currently active. This could be initiated by navigating to the folder for storing the pcd-files, right-click on it, and choose *Open in Terminal*. As the messages are published to the topic, will they be stored in the folder for further use and assessment.

After the point cloud is stored as a pcd-file can a separate python script be executed. The script *zms_algorithm.py*, shown in Appendix H, will calculate the ZMS for the cloud and create a new pcd-file to showcase the calculations. The script can be executed with the command *python*3 *zms_algorithm.py* from a command terminal. The reason for storing the point cloud through a command terminal and separating this calculation into an own script external from the ROS environment was for using a python library, *pclpy* [15], which relies on a python 3.6 to 3.8 version, while the python version supported with ROS melodic ordinarily is 2.7.
Before starting the script, one must edit the file path to the input file of interest for execution and the name for the new file to create with the results stored.

The script will since go through the point cloud, point by point. It starts by searching for the points within a distance ρ from the given search point, X, before calculating their average position of this set of points, ξ. This average will be the Zero Moment (ZM) corresponding to point X. The distance between is the Zero Moment Shift vector, n_ρ, which is calculated and stored in a list. An additional point cloud is created for demonstration purposes that stores all the points from one radius search, illustrating for one set ξ. The point is defined by defining the index number in the point cloud, which the user sets. The points within the defined sphere are colored and stored accordingly. The

ZMS for the cloud is stored in *zms_array.txt*, where they could be collected for assessment and use.
At the end of the script are a new for-loop initiated to store the two point clouds. The original point cloud and the ZM are applied in different colors to differentiate them. This is done to illustrate the function and its affect. The last point cloud containing the object and ZM calculation is since stored in a pcd-file.

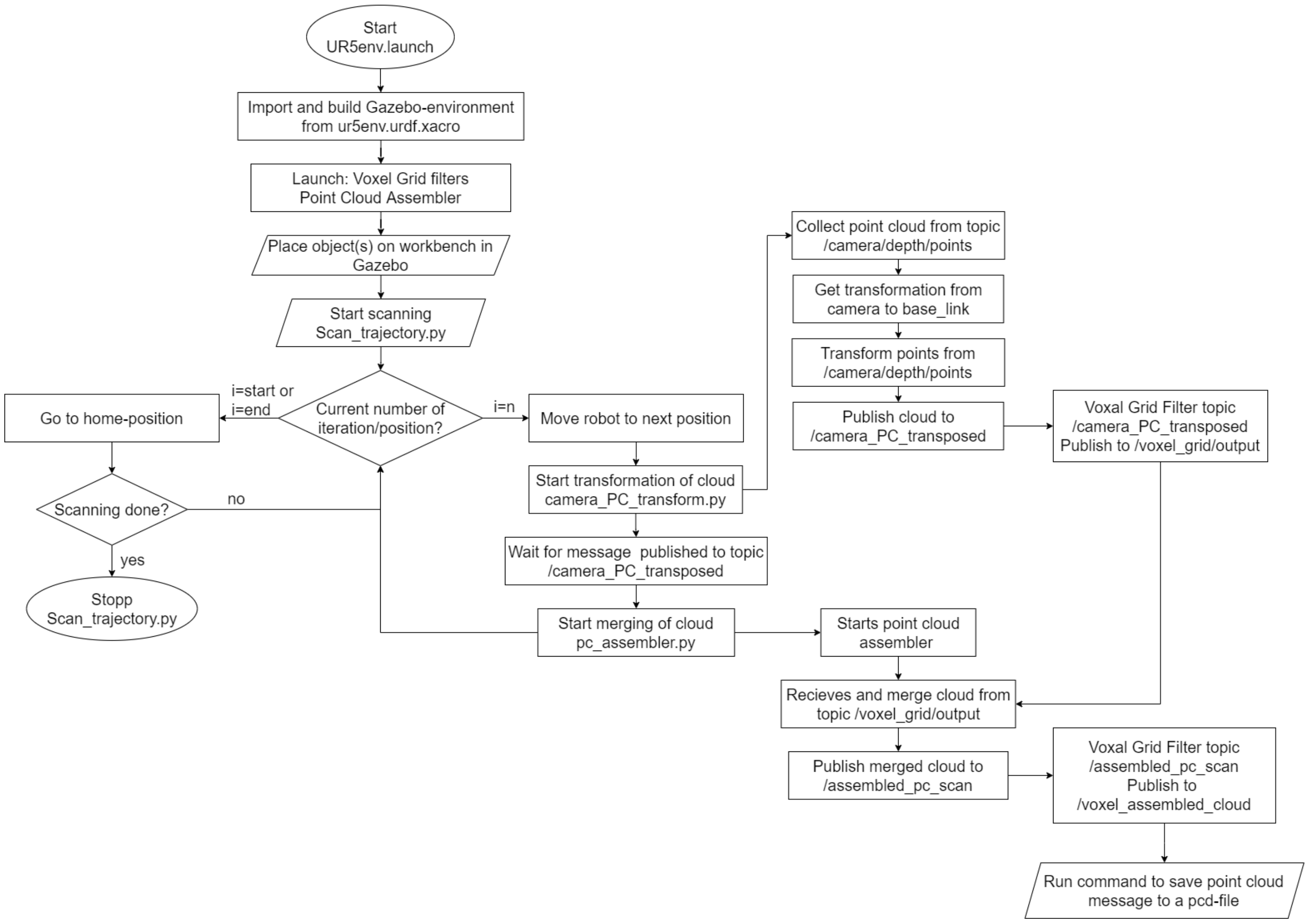

Start
UR5env.launch
Import and build Gazebo-environment from ur5env.urdf.xacro
Launch: Voxel Grid filters Point Cloud Assembler
Place object(s) on workbench in Gazebo
Start scanning Scan_trajectory.py
Current number of iteration/position?
i=start or i=end
Go to home-position
Scanning done?
no
yes
Stopp Scan_trajectory.py
i=n
Move robot to next position
Start transformation of cloud camera_PC_transform.py
Wait for message published to topic /camera_PC_transposed
Start merging of cloud pc_assembler.py
Collect point cloud from topic /camera/depth/points
Get transformation from camera to base_link
Transform points from /camera/depth/points
Publish cloud to /camera_PC_transposed
Voxal Grid Filter topic /camera_PC_transposed Publish to /voxel_grid/output
Starts point cloud assembler
Recieves and merge cloud from topic /voxel_grid/output
Publish merged cloud to /assembled_pc_scan
Voxal Grid Filter topic /assembled_pc_scan Publish to /voxel_assembled_cloud
Run command to save point cloud message to a pcd-file

Figure 4.1: Flowchart scripts

Chapter 5

Test Setup

This chapter contains the method of how the testing has been conducted for this project. This way, one can replicate the setup and result in a later or similar scenario for comparison or further development. The test environment is set up in Gazebo with the use of objects included in the Gazebo library. The objects used in the testing were chosen because of their variety of surface, size, curvature, and shapes. By testing with several different objects, one could see how the different aspects of the setup would perform accordingly, both regarding collecting and creating a point cloud of the model and calculating the ZMS for the cloud accordingly.

Single Object Capture

During capturing of single object point clouds are the objects placed in the center of the scanning area. This is done first by importing the object of interest into the environment. The pose of the object was since defined to [0.4, 0.0, 1.0], which are defined in meters relative to the global coordinate, *world*. This will place the object in front of the robot accordingly, as shown in Figure 5.1.
With the object in place was the script *scan_trajectory.py* executed to start the scanning procedure.

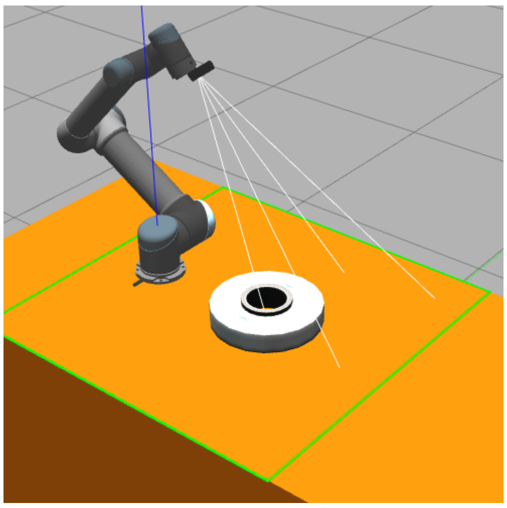


Figure 5.1: Single object scanning Placement

Object Clutter Capture

With testing of several objects placed on the workbench inline or in a clutter, have them been placed within [0.3 - 0.8,-0.4 - 0.4, 1]. Different numbers and shapes of objects have been tested to see both performances regarding accuracy and execution since more surfaces would need more data to analyze.

Zero Moment Shift Calculation

The testing for the ZMS calculation has been conducted by defining the pcd-file to which point cloud it should evaluate. During the testing have several values for ρ been conducted to both see and evaluate its effect on the calculation of the surface structure.

Chapter 6

Results

In this chapter the results from the project are presented. There are different parts and perspective regarding the point cloud generated, system performance and algorithm calculation presented. The results are from the execution in a virtual environment using ROS and Gazebo.

6.1 Point Cloud

The cloud created in the virtual environment is stored in a pcd-file for further use and analysis. Several cloud collections were conducted to test the amount on data gathered, quality and process compilation. The clouds created were evaluated by parameters and features which could affect the execution and further data treatment in the process.

6.1.1 Cloud Merging

When the robot moves through the predefined trajectory, point clouds are captured from each view angle, filtered, and merged to one cloud representing the object. A part of this process is shown in Figure 6.1 - 6.4. Here one can see how the cobot and camera in the Gazebo simulation are scanning some objects and how this data looks like. In Figure 6.2, the data posted from the different topics are shown. The point cloud published from the camera is shown on the left. Here are all the surfaces within the field of view registered. The data filtered and treated are shown on the right side. Here is the top of the workbench filtered out and the different angles captured, which are represented in one cloud. Here one can see how the soda can on the left casts a shadow over the bowl, while on the right are the points representing the bowl presented since they were seen and registered from another view angle.

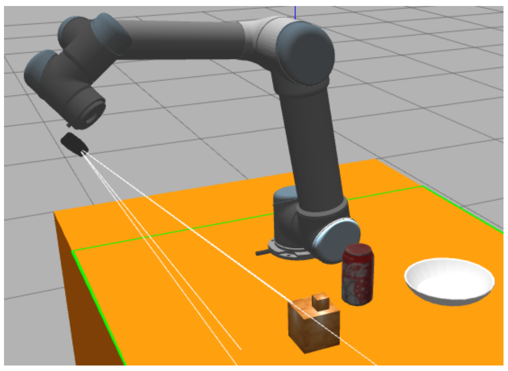


Figure 6.1: Gazebo object scan

All the surfaces registered from the workbench plane and upwards are posted to one topic, as shown in Figure 6.3 and 6.4. Here are two angles from the objects shown. The result of this is a point cloud representing the four objects consisting of 104 768 points, where the voxel grid filter was set to 1 mm grid size.
The alignments and merging of the clouds are crucial to create a point cloud that represents the object correspondingly. If the point cloud deviates from the object, could further analysis and interaction result in a fault execution. The merging of point clouds from different angles has resulted in a point cloud structure with the same shape as the object. One example of this could be seen for *Sodacan bottom* and *Sodacan* in Figure 6.12. Here one could see that the cloud constructed has a cylindrical shape. The surfaces in height are continuous with no sudden deviation or misinterpretation from the can shape. The result is a point cloud assembled representing the object seen, which corresponds to its shape and size.

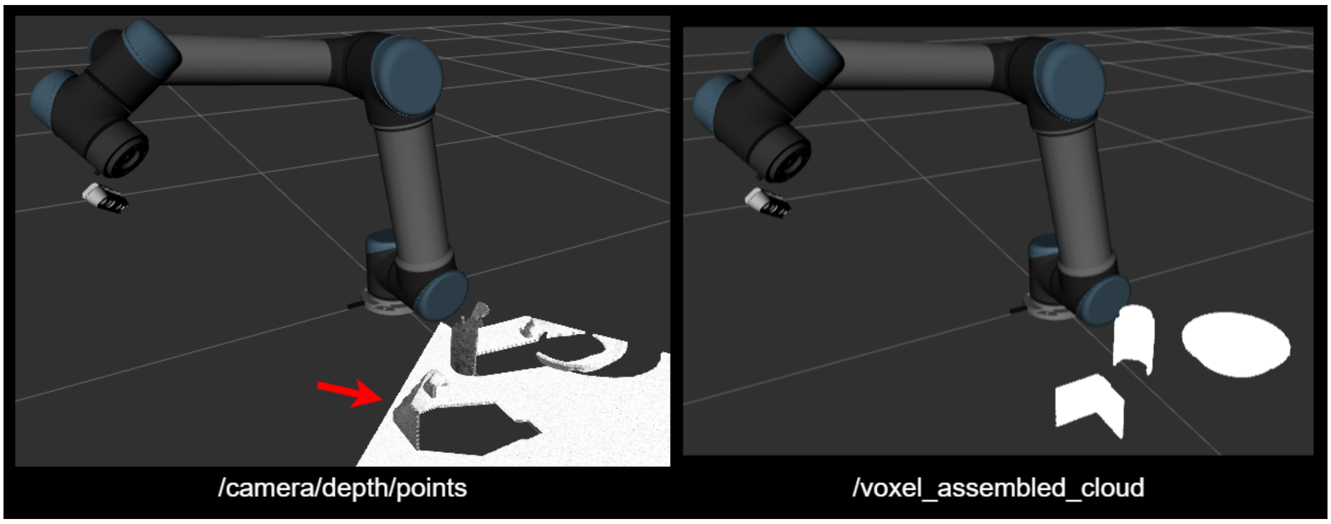


Figure 6.2: Point cloud extraction

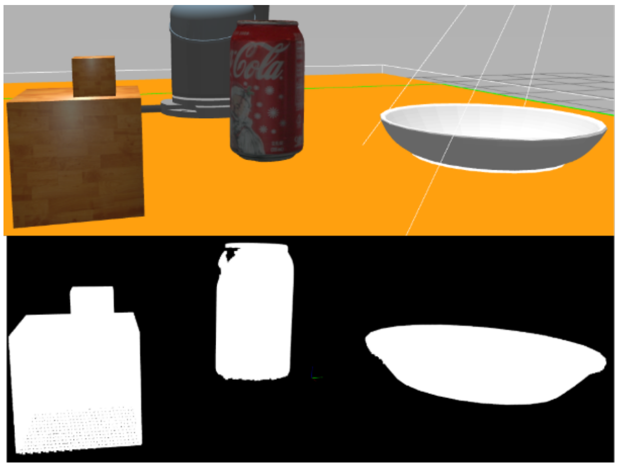


Figure 6.3: Objects point cloud view 1

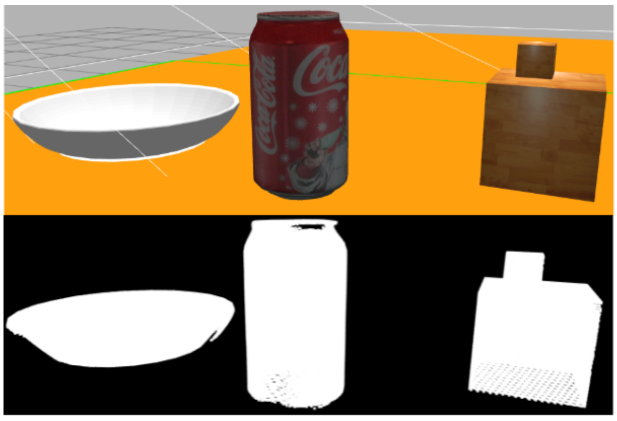


Figure 6.4: Objects point cloud view 2

6.1.2 Point Distribution and Density

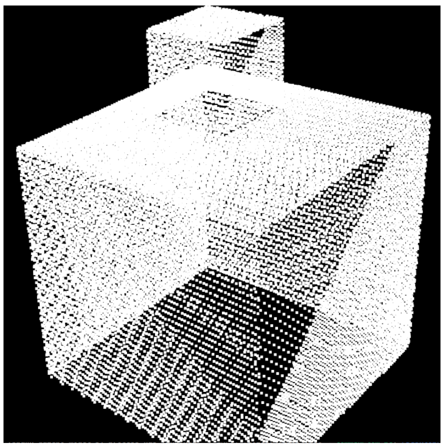


Figure 6.5: Cubes point cloud

The number of points included in the object representation depends on the grid size defined in the voxel grid filters and the surfaces registered by the camera.
Figure 6.5 illustrates how an uneven capture of an object could look like. Here two cubes have been placed in a location where parts of their surface have been closer than the camera's cut-off distance. This could be seen in Figure 6.2 marked by the red arrow. As a result, one corner of the cubes is defined by fewer points since these points registered here are from other view angles with less view of that particular surface. According to the data sheet for the camera, Intel Realsense D435i, should the close range limit for detection be 20 cm, compared to the simulation, which has a cut-off border at 50 cm [6]. The surfaces within the scanning range that are not obscured in several angles from the camera view get a good even distribution of points.

The density of points in the point cloud was in this project changed in the voxel grid filters. The raw point cloud published from the camera could consist of up to 921 600 data points in the virtual environment. Running and collecting untreated clouds from the scanning procedure could result in a merged cloud of up to 5 529 600 points. The analyzing and treatment of point clouds are computational heavy. A point cloud consisting of a vast amount of data points will increase the execution time on this given setup.
The raw point cloud was treated to extract the points of interest from the workbench plane and upwards.
The voxel grid filter for the cloud assembler has been set to different sizes and tested to check the resolution and effect on the cloud created.
The test results are presented in Table 6.1.

Table 6.1: Point cloud data density

Figure showing objects	Voxel grid size	Number of points in cloud
Figure 6.7	0.5 mm	163 108
	1 mm	99 390
	5 mm	5 704
Figure 6.6	0.5 mm	105 869
	1 mm	56 765
	5 mm	2 508
Figure 6.8	0.5 mm	687 331
	1 mm	432 219
	5 mm	19 547

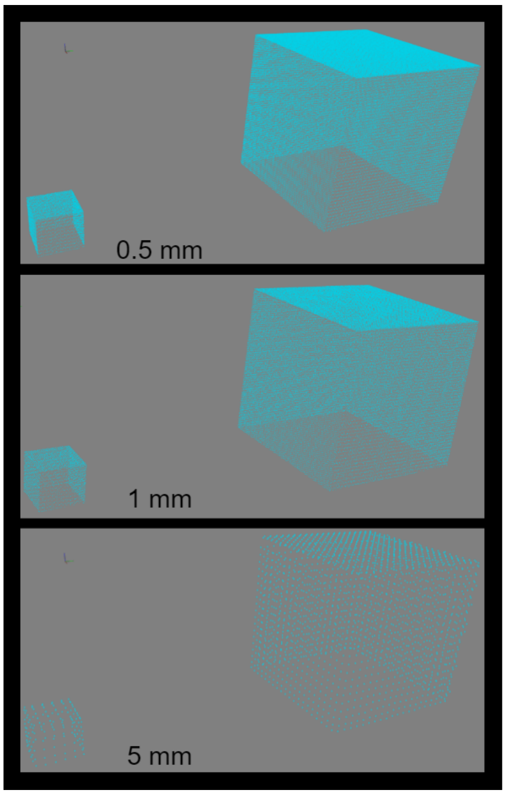


Figure 6.6: Voxel grid sizes cubes

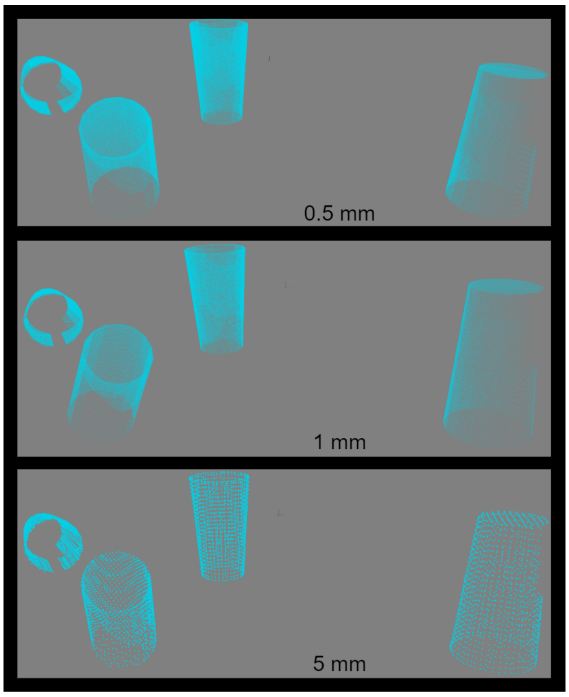


Figure 6.7: Voxel grid sizes cups

Table 6.2: Average execution performance

Single Object/Clutter	Voxel Grid Size	Number of Points	Time execution
Clutter	1 mm	594 800	2 min 3 sec
Single Object		47 390	2 min 17 sec
Clutter	2 mm	153 400	1 min 58 sec
Single Object		14 000	2 min 17 sec
Clutter	4 mm	36 680	2 min
Single Object		3 700	2 min 16 sec
Clutter	5 mm	21 300	1 min 59 sec
Single Object		2 420	2 min 17 sec
Clutter	7 mm	13 370	1 min 55 sec
Single Object		1 200	2 min 17 sec

By applying a voxel grid are some uneven distribution removed, which depends on the given filter size. Where the filter size is small are only the uneven distribution within that scale evened out. In Figure 6.6 - 6.8 one could see the effects of increasing the voxel grid size. In this showcase do they vary from 0.5 to 5 mm.

The time used for running through the positions and creating a point cloud depends on the number of points included. The average time needed for creating a point cloud consisting of a certain amount of points is shown in Table 6.2. The test was conducted on the objects shown, or with similar structural properties, as in the figures in this chapter with a similar environment setup. The clutter would consist of 15 objects placed randomly on the workbench, while the single object would consist of an average of the object shown in Figure 6.10-6.12.

As a result, point cloud creation with perspective to the distribution of points are the clouds mostly evenly distributed. The cloud creation is affected to some degree if the object is closer than the camera's cut-off limit. This could result in parts of the surface consisting of fewer points.
The same result could be seen in a clutter, where some objects are obscuring the camera view, casting shadows on other surfaces. In these cases could some blind spots occur. Regarding density in the cloud, it is dependent on the voxel grid size set. The points are evenly distributed over the surfaces registered by the camera.

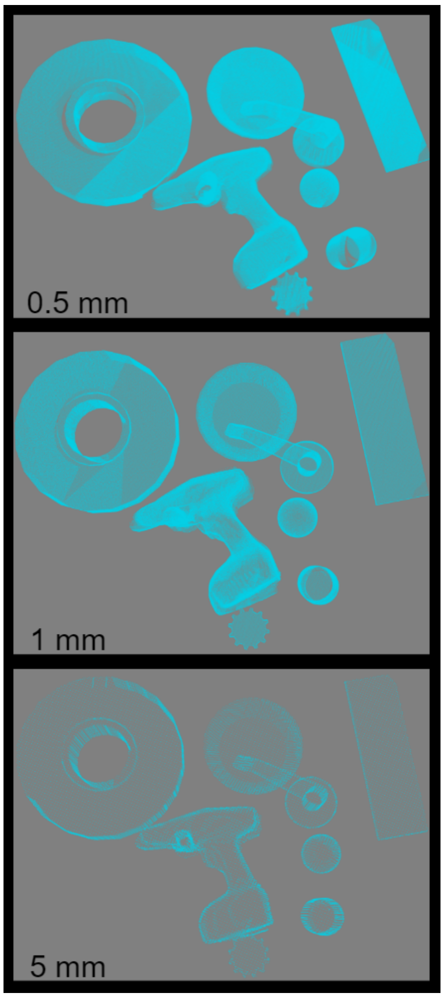


Figure 6.8: Voxel grid size clutter.

6.2 Zero Moment

The objects used for illustration the ZM and ZMS are shown in Figure 6.9. These objects have different size and curvature features, showing how these factors will lead to different results from the calculation.

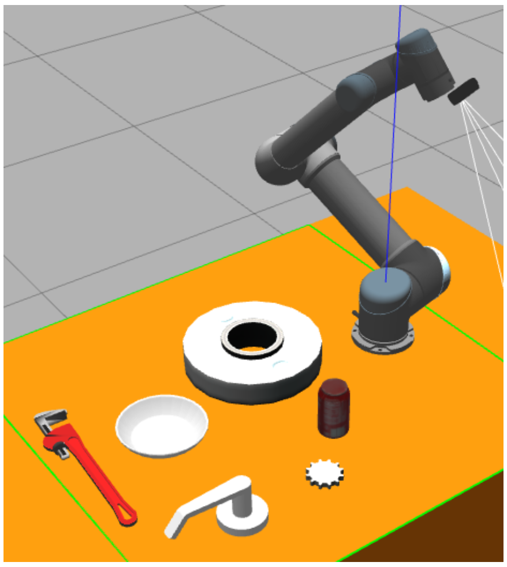


Figure 6.9: ZM object for scale.

The results from the ZM calculation are shown in Figure 6.10 - 6.12. Here six objects are shown with different radius, ρ, used for defining the set of points ξ. This is done to illustrate its effect on a point cloud graphically. The light blue points represent the object's registered surface, and the black points are the calculated Zero Moment. A small value ρ would result in the sphere $B_\rho(X)$ being small, meaning fewer points are included in each set ξ. A smaller set of points to calculate ZM would generally result in a smaller distance between the given setpoint X for the set ξ and the ZM $M^0_\rho(\xi)$. The opposite would occur with a larger value for ρ.

For the objects placed left in Figures 6.10 - 6.12, with $\rho = 2mm$, are the ZM more evenly distributed across the whole surface, compared with a greater ρ, where the curvature gets more defined. Looking at the bowl with $\rho = 8mm$ in Figure 6.10 one can see all the black ZM points in the bowl as a result of this. As illustrated here, the average point in a sphere $B_\rho(X)$ will tend to be in a concave surface.

Looking at the brim of the bowl, one can see a defined light blue edge. Here are the ZM points within the brim because of the high curvature at this part of the structure. The same tendencies and features can be seen on the other objects in varying degrees in relation to their structure curvature. At the areas with flat faces are there fewer average changes, resulting in a more evenly distributed pattern where the ZM point will lay within the plane of the surface. Closer to an edge or curvature will the difference result in the ZM point outside of the surface.

From the gear shown in Figure 6.12, one can see the effect of high local curvature at its teeth with an increasing ρ. At each tooth will the ZM points be placed more towards each tooth's center, while the ZM points in between each tooth will be placed just outside of the gear's main body. This shows how the size and local curvature affect the calculation of ZM since the average coordinate within the sphere $B_\rho(X)$ will be further away from the point X with a set of points ξ with high variance because of curvature. For a set ξ on a tooth, would the curvature be higher than between each tooth.

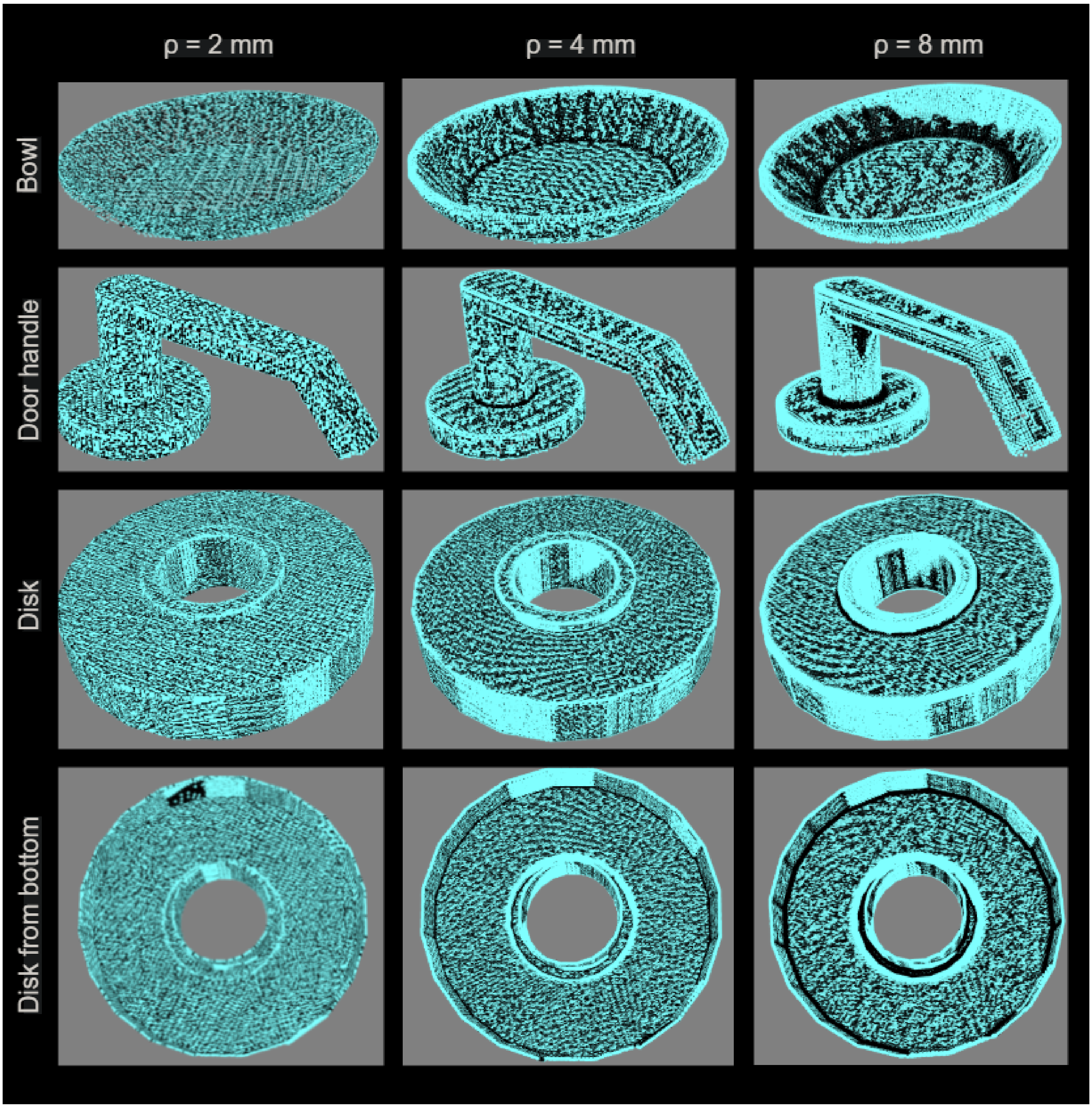


Figure 6.10: Zero Moment set one

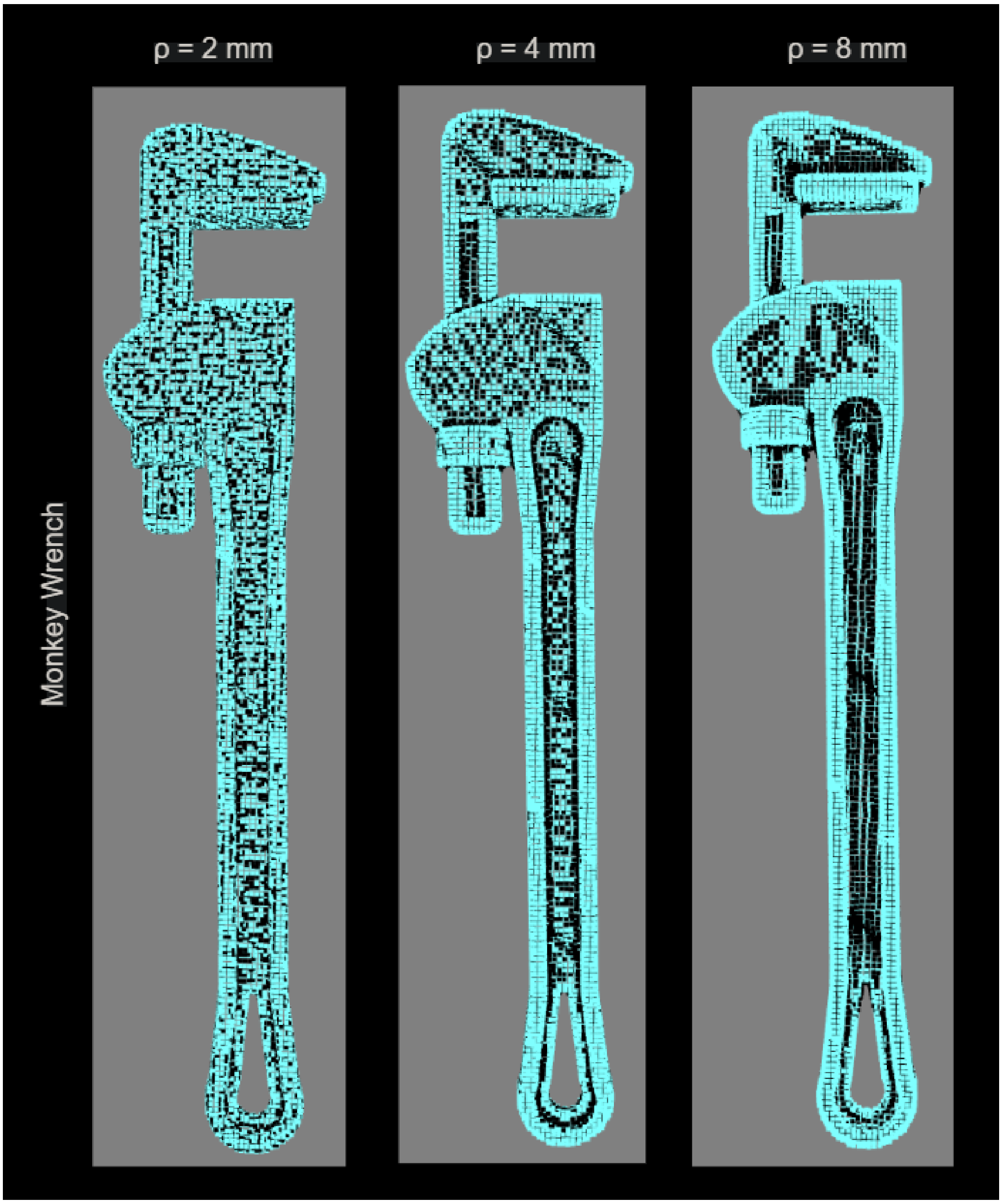


Figure 6.11: Zero Moment set two

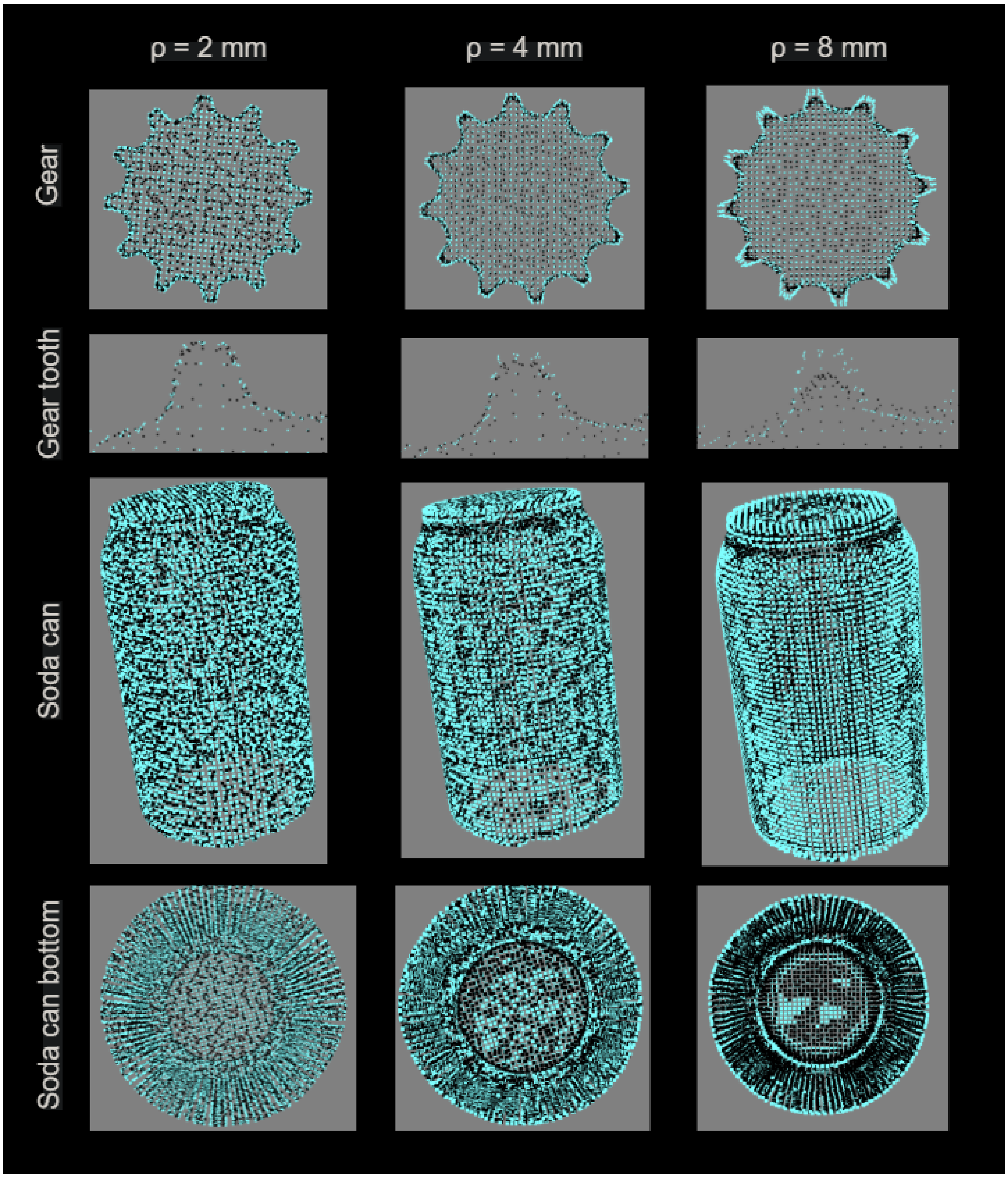


Figure 6.12: Zero Moment set three

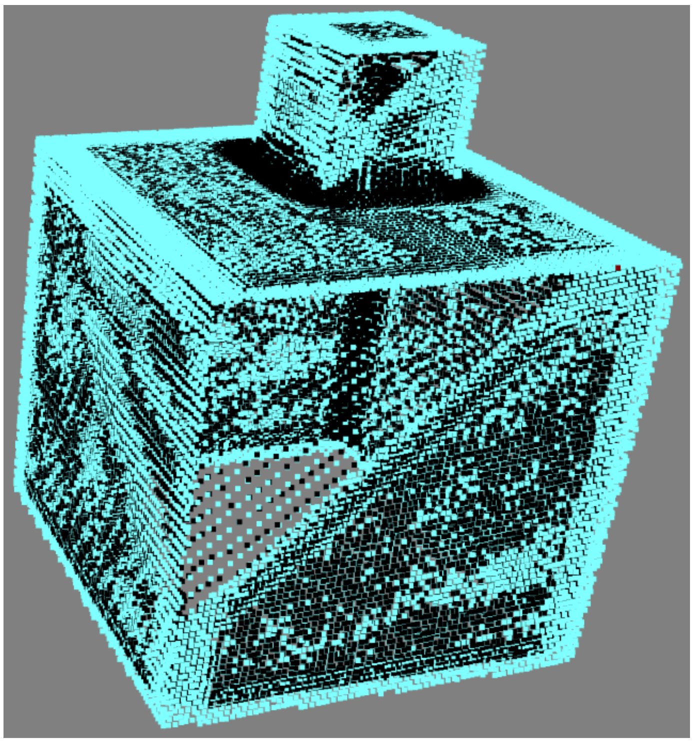


Figure 6.13: Zero Moment cubes

If the point cloud has an uneven distribution of points, could this affect the ZM and ZMS calculation, as shown in Figure 6.13. This section is from the point cloud created in Figure 6.1, where one of the corners wasn't successfully captured. The result was a line of ZM-points where the plane goes from a dense to a less dense representation of the surface, compared to the rest of the flat phase, which is more randomly distributed.

6.3 Zero Moment Shift

The ZMS vectors, which describe the point cloud curvatures, are calculated and stored during the creation of the ZM point cloud. The ZMS vectors are ordered accordingly to their representing point X in the point cloud.

The ZMS was conducted in further detail on the door handle for illustration. For the ZMS calculation on the door handle, shown in Figure 6.14 - 6.15 and Table 6.3, one can see how the vectors are affected by the curvature calculation for the object and how ρ is affecting this result.In these figures the current point X is shown as the red point, while the calculated ZM, $M_\rho^0(\xi)$, are a black point for the set ξ. The red arrow between them represents the ZMS vector. One can see the length and sign for the value in the X,Y, Z plane correspond to the coordinate system in Figure 6.15. The current point X was chosen to illustrate the effect on highly curved surfaces and edges.

The effect of varying the size of ρ can be seen in Table 6.3 under "Vector length". A larger ρ will include more points in the set ξ. This could potentially increase the distance between $M_\rho^0(\xi)$ and point X, which are defining the ZMS vector, n_ρ.

Table 6.3: ZMS results door handle

	X length	Y length	Z length	Vector length
$\rho = 2mm$	$-3.576 \cdot 10^{-6}$	$-4.480 \cdot 10^{-4}$	$-5.292 \cdot 10^{-4}$	$6.93 \cdot 10^{-4}$ mm
$\rho = 4mm$	$-6.076 \cdot 10^{-4}$	$-8.260 \cdot 10^{-4}$	$-7.018 \cdot 10^{-4}$	$1.243 \cdot 10^{-3}$ mm
$\rho = 8mm$	$-8.625 \cdot 10^{-5}$	$-1.606 \cdot 10^{-3}$	$-1.684 \cdot 10^{-3}$	$2.329 \cdot 10^{-3}$ mm

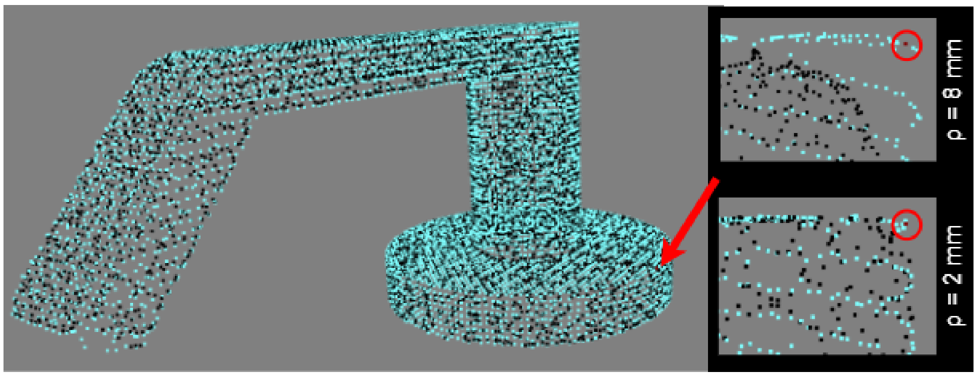


Figure 6.14: Door handle section area

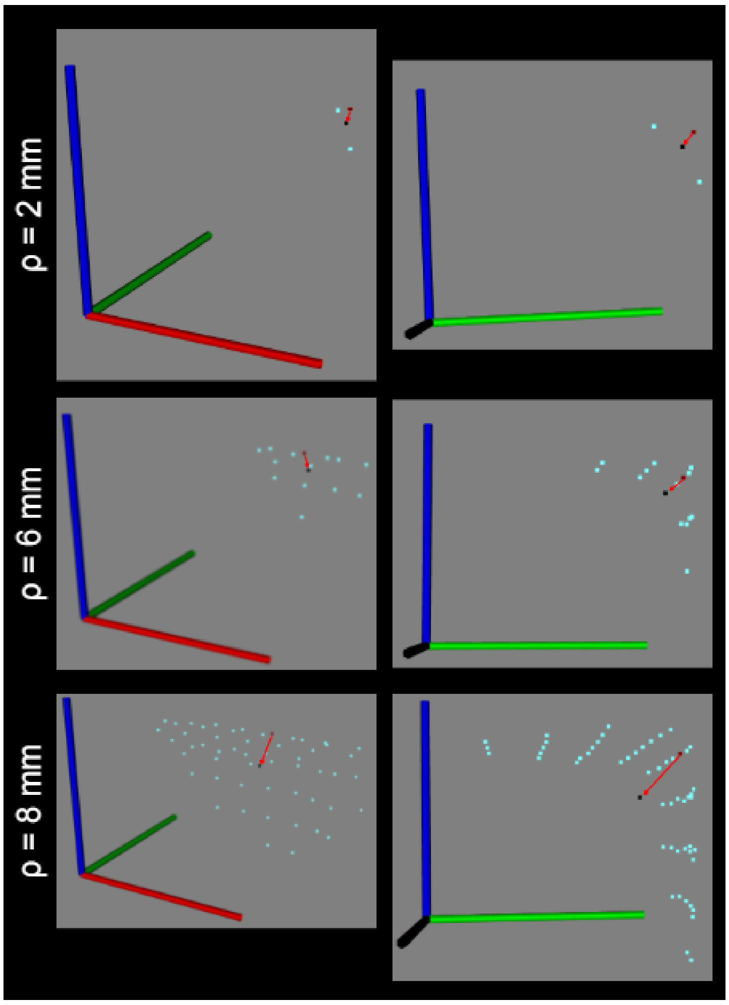


Figure 6.15: Door handle ZMS

The time execution for the ZMS-calculation are shown in Table 6.4. These tests have been conducted on two point clouds filtered with a voxel grid of 4 mm. The bowl was chosen since its cloud consisted of 3 698 points, which deviates by only two points from the average of 3 700 points, shown in Table 6.2. The time for execution shows the average execution time per task. The time was recorded both when the ZMS function would store and handle the point cloud, and when the function only would store a list representing the ZMS.

Table 6.4: Time execution ZMS script

Clutter/Bowl	ρ size	Time execution w/Point Cloud	Time execution w.o/Point Cloud
Clutter	1 mm	15.1 sec	11.3 sec
Bowl		1.6 sec	1.1 sec
Clutter	2 mm	15.5 sec	11.4 sec
Bowl		1.8 sec	1.2 sec
Clutter	4 mm	16.5 sec	12.8 sec
Bowl		1.9 sec	1.4 sec
Clutter	8 mm	23.2 sec	18.8 sec
Bowl		2.6 sec	1.9 sec
Clutter	16 mm	48.3 sec	44.8 sec
Bowl		6.1 sec	5.3 sec
Clutter	20 mm	1 min 8 sec	1 min 3 sec
Bowl		8.6 sec	7.8 sec
Clutter	32 mm	2 min 28 sec	2 min 17 sec
Bowl		17.9 sec	16.8 sec

Chapter 7

Discussion and Future Work

A virtual environment with a robot and a depth camera was built during this project and created point clouds for further evaluation. The following sections in this chapter cover the discussion of the different aspects, values, and further considerations for this project. The last section summarizes shortly for the overall result of the project.

Camera Cut-off Limit

During the capture and creation of point clouds in the virtual environment, some surfaces within a certain reach were not detected by the camera. The points published on the topic $/camera/depth/points$ are a $sensor\_msgs/PointCloud2$ message. This message type is the most used for handling and working with point cloud in ROS.

The problem that occurs for creating point clouds was when objects are too close to the camera. The camera only detects surfaces that are from one given distance and onward. Compared to another topic created by the camera, $/camera/depth/image\_raw$, which publishes a message type $sensor\_msgs/image$, could detect more than what is published on /camera/depth/-points.

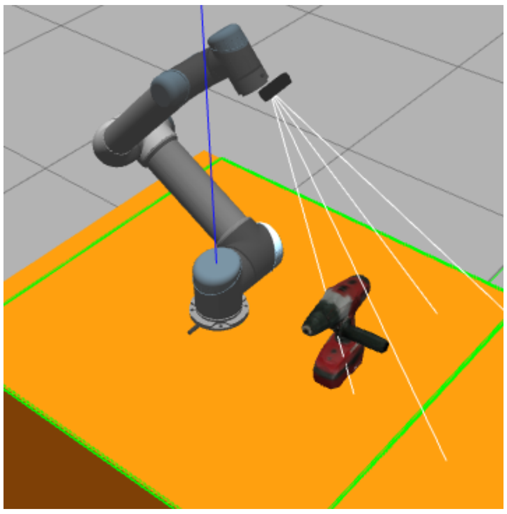


Figure 7.1: Gazebo simulation

One example of this is shown in Figure 7.2 - 7.3, with the simulated environment shown in Figure 7.1. Here was the camera registering the surfaces and publishing data to each topic correspondingly. For the data published to /camera/depth/image_raw, shown in Figure 7.2, it would register surfaces closer to the camera compared to the point cloud published in /camera/depth/points. The result in this scenario was that the top of the drill is not registered in the point cloud, compared to the depth image, which can detect and publish these surfaces.

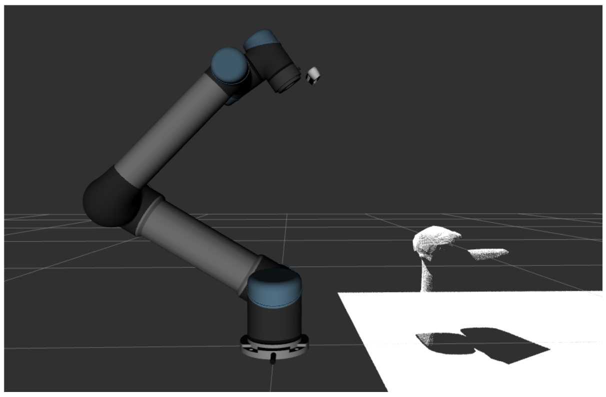


Figure 7.2: Depth Image

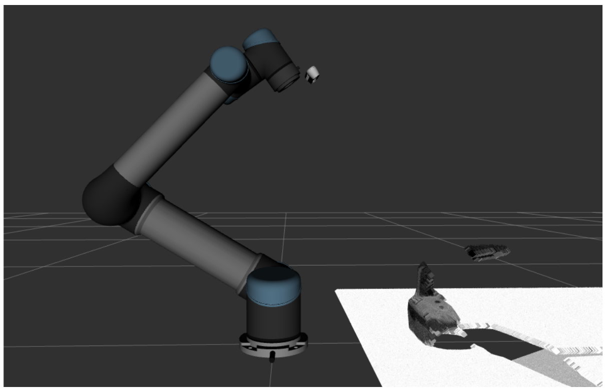


Figure 7.3: PointCloud2

Both of these topics are posted on different message types. Since the libraries and functions used and available for treating point clouds in ROS are aimed towards sensor_msgs/PointCloud2, the data published in /camera/depth/points are more suited.

It is possible to convert the data from sensor_msgs/image to sensor_msgs/PointCloud2 using packages such as *depth_image_proc* or additional software as OpenCV or OpenNI. In that case, one could obtain points closer to the camera in the simulation, thereby gather more data representing the surfaces registered where close placed objects would be occurring. This would increase the data handling and introduce a new lid into the chain of processes, where data would be gathered, converted, and published before being analyzed and used further.

Another solution could be changing the camera's position to be further away from the object during scanning and making sure the objects are within a defined area. Nevertheless, with a wide range of objects in different shapes and sizes, wouldn't this approach be ideal. Also, increasing the robot's range would need consideration regarding the overlapping work area with coworkers and obstacles in its environment to retain the level of safety. Treating, for instance, small objects the same way as big ones considering scanning distance could lead to fewer points and features registered. In this case, would it be more suited to have several scanning trajectories dependent on the size of the objects placed on the workbench. Before a scanning procedure started, could the robot move to an "evaluation pose" to evaluate the registered points and execute the scan trajectory accordingly. This way, one could make a "small, medium, large" object-scan execution. It could also evaluate the height of the object from the highest registered points. If the object is a certain height, which could lead to a collision when the robot would normally move over the object, it could change the planned trajectory to instead move around the object than over.

Regarding the cut-off border, does it seem that the boundary for registered points is set in the package for the camera in the simulated environment, since the topic /camera/depth/image_raw can register surfaces closer than /camera/depth/points. This was also confirmed by testing in a physical setup as shown in Figure 7.4 - 7.6.

A coffee cup was used to confirm the reading distance of detected surfaces from the camera. The data published in both Image and PointCloud2 message are shown in Figure 7.5 - 7.6. Here one can see how the handle is cut off in both topics, showing that both have the same range. In this test were the border 15 cm from the camera, instead of 20 cm as stated in the datasheet [6].

This shows that there is some deviation between the virtual simulation and physical application. Such differences are essential to take into account when designing and fine-tuning a system. During the physical test was the package from the camera, realsense-ros, used. To improve the cut-off deviation in the virtual environment, should one go through the Gazebo-plugin package for the camera and code or find the defined cut-off distance for the simulation and decrease it towards 20

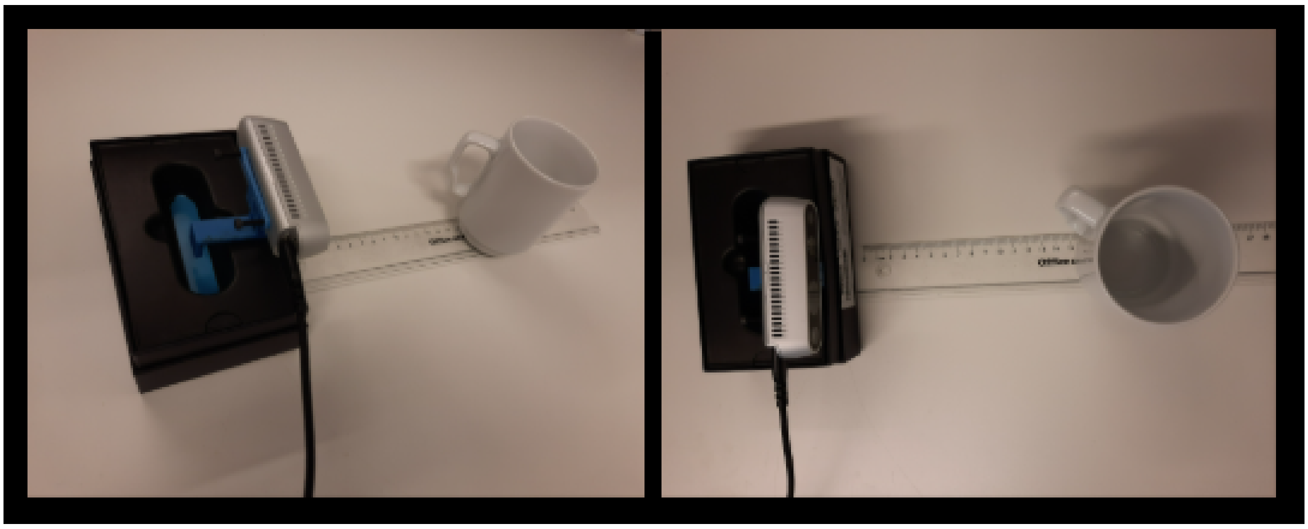
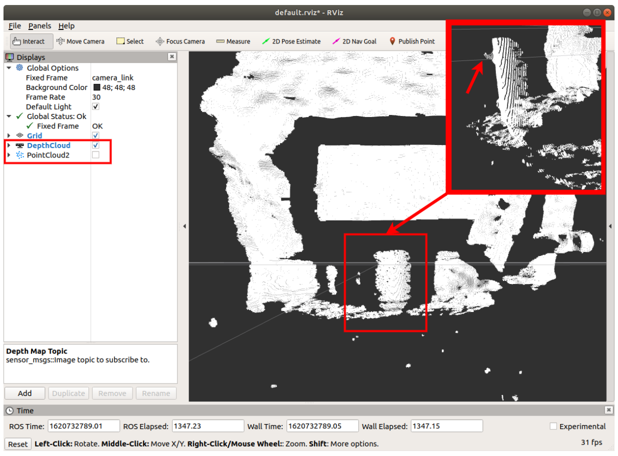


Figure 7.4: Camera test setup

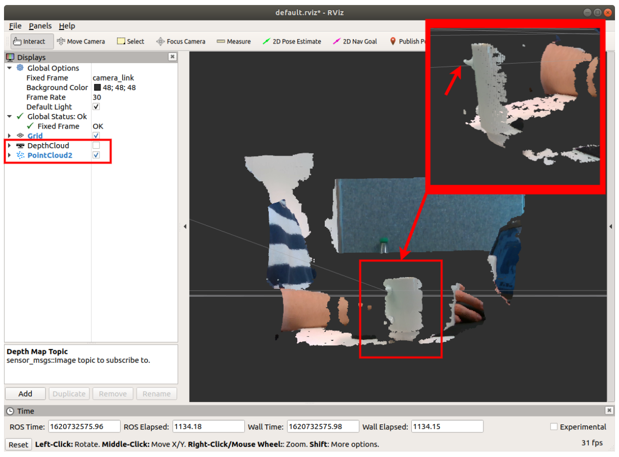


Global Options
Fixed Frame camera_link
Background Color 48; 48; 48
Frame Rate 30
Default Light ✓
Global Status: Ok
Fixed Frame OK
Grid ✓
DepthCloud ✓
PointCloud2 ☐

Depth Map Topic
sensor_msgs::Image topic to subscribe to.

Time
ROS Time: 1620732789.01 ROS Elapsed: 1347.23 Wall Time: 1620732789.05 Wall Elapsed: 1347.15 ☐ Experimental
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options. 31 fps

Figure 7.5: Rviz DepthImage



Global Options
Fixed Frame camera_link
Background Color 48; 48; 48
Frame Rate 30
Default Light ✓
Global Status: Ok
Fixed Frame OK
Grid ✓
DepthCloud ☐
PointCloud2 ✓

Depth Map Topic
sensor_msgs::Image topic to subscribe to.

Time
ROS Time: 1620732575.96 ROS Elapsed: 1134.18 Wall Time: 1620732575.98 Wall Elapsed: 1134.15 ☐ Experimental
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options. 31 fps

Figure 7.6: Rviz PointCloud

cm as stated in the datasheet from the producer. The image topic has a cut-off limit of 20 cm compared to the PointCloud2 topic, which has 50 cm in the virtual environment. This leaves a 30 cm blind spot, meaning the simulated robot would need to move about 30 cm further away from the object to register the same surfaces than the depth image in the virtual environment or the physical setup would need. By changing this would improve the cloud capture and help prevent achieving results like shown in Figure 6.13.

Filter

The result from Table 6.1 shows the effect of adjusting the grid size and the number of points remaining in the cloud. One could refine the raw point cloud given and its distributed points for further processing and analyses by applying such a filter. Only the second voxel grid filter was adjusted during the tests, and the first filter was set to 0.5 mm. This was to keep the test more consistent, the level of data available, prevent affecting or decrease the quality and data for the process downstream from the first filter. For the single object, one could notice that the execution time is consistent around 2 min 17 sec, where only a minor deviation of ± 0.6 seconds was detected during testing.

For the clutter of objects was there more variation in time execution. Every scan executing for clutter was quicker than with a single object given from the calculated time, despite the fact that more data were processed. This could result from the compilation and high process load since the time recording of the execution was included in the scan_trajectory script. Even though the clutter at, for instance, 7 mm grid size produced a similar amount of points as the average single object with 2 mm as an end result, would the clutter need to process and filter more data. Another aspect that could be affecting this result is the number of objects in the Gazebo simulation. When more objects were included was the processing load higher on the computer.

Compared to [3], their approach of executing grasp was using around 31k points on a clutter consisting of 13 objects. This would, for this project, correspond to a voxel grid size of approximately 4 mm, achieving an average of 36 680 points on a clutter of 15 objects. By setting a similar foundation of data amount to assess and act from, would the further implementation of a LoCoMo algorithm be more comparable to see how performance and success rates correlate to each other.

The adjustability of the filter grid size leaves flexibility regarding data included in the final point cloud. This leaves room to set the size suited for including the data level needed for different algorithms and task executions.

Zero Moment Shift

To better illustrate the ZMS calculation was the point cloud of the object and the calculated ZM stored in a pcd-file. Here one can see the effects of ρ for the point cloud and how the curvature is interpreted. A greater value for ρ would result in the edges and highly curved areas looking more defined in the illustrations. This aspect also relies on an evenly distributed point cloud, as shown in Figure 6.13, which could lead to a misinterpreted surface when calculating a surface fit onto a gripper. This shows one effect compared to a well-defined evenly distributed point cloud to interpret. Further testing with a finished system is recommended to see the effect from different quality of point clouds in assessment and action since this could affect the process in a later stage, as well as the level of processing needed for the execution. The finished algorithm should grasp the area with the highest likelihood of a good grasp, which could be in other regions of the point cloud. Its effect should be considered and tested with a system able to grasp and lift objects to see its impact on the execution.

In a setup with a LoCoMo algorithm, would the ZMS not need to store or create such a pcd-file, if not of interest for further assessment. The ZMS calculation could be set up as a function, similarly to how the transformation and point cloud merging script has been set up in this project. This could increase the efficiency, as shown in Table 6.4. Here was the script launched to compare the execution time, illustrating the difference when handling a point cloud and array for the ZMS to only handling n_ρ. The results show fairly similar execution time, both when creating and storing an additional point cloud, and when only calculating the ZMS. The 4 mm voxel grid size was chosen for this test since it produced a similar amount of points as [3]. With a foundation of data consisting of a similar amount of points describing the detected surfaces would make the further testing and development more comparable to see how the project would perform compared to the setup [3].

The suitable ρ size applied for calculation of n_ρ on a given cloud should be looked into in conjunction with the gripper used further in the system since this could potentially affect the surface fit assessment between the object and the gripper, as well as the size range of the objects.

LoCoMo Implementation

The virtual environment created and setup is a good foundation for implementing the LoCoMo algorithm. The dependencies for implementing LoCoMo would be getting and integrating a gripper into the virtual environment and calculation. The grippers available during the project execution had limited to no support and models included for ROS. With the limited time for the project was the process of modeling and creating controls for a new gripper from scratch to implement into the virtual environment not executed. For proceeding with the LoCoMo algorithm is the gripper a crucial part since it is used in the calculation and interaction with the object to find a suitable grasping location on the object from the point cloud.

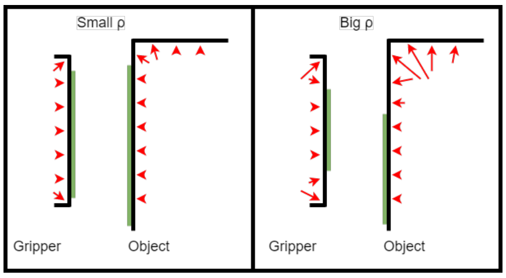


Figure 7.7: Grasp illustration example

Since the LoCoMo is valuating the surface fit between the object and the gripper based on $|n_\rho|$ from ZMS as L_1, could the setup for the ZMS affect the grasp evaluation. An illustration is shown in Figure 7.7, where the green area is where the calculation C_ρ would find in this scenario a surface fit for the gripper and object with a high score. This could be a potential challenge in an environment with vast, varying objects to interact with, like small, highly curved objects and large, low curved objects. An example of this could be for the gear shown in Figure 6.12, where its surface around the teeth is considered highly curved, in comparison with the disk in Figure 6.10, which are several times larger. This should be further evaluated and tested to ensure its effect with the use of the LoCoMo algorithm and the range of objects this could affect.

The LoCoMo are considering the grasp out from the point cloud of the object but not considering other aspects, such as the weight distribution. This problem was experienced by [3]. By including other assessment factors in addition to the geometry computation, could the algorithm be improved. One proposal could accordingly be considered for this enhancement. To keep the system learning-free could an approach by finding the object's approximate center be used. Creating a cutting plane perpendicular to the object's central axis or an approach by computing the centroid of all the points in the point cloud could be considered. This could calculate the approximate center of a single object and be used to weigh the grasps closer to the center of the object higher than those on the outer edge of the cloud. This would assume an even weight distribution of the object and would be more suited for interaction and handling homogeneous objects. This would be dependent on the objects that the setup would interact with and should be considered accordingly.

Equipment Load

During the execution of the scanning procedure and creation of the point clouds were there a high load on the CPU for the computer running the environment, as shown in Figure 7.8. In this case, there were 15 objects placed on the workbench in Gazebo. Here one can see the load on the CPU is running at full capacity during execution. Similar loads were seen during the scanning of single objects.

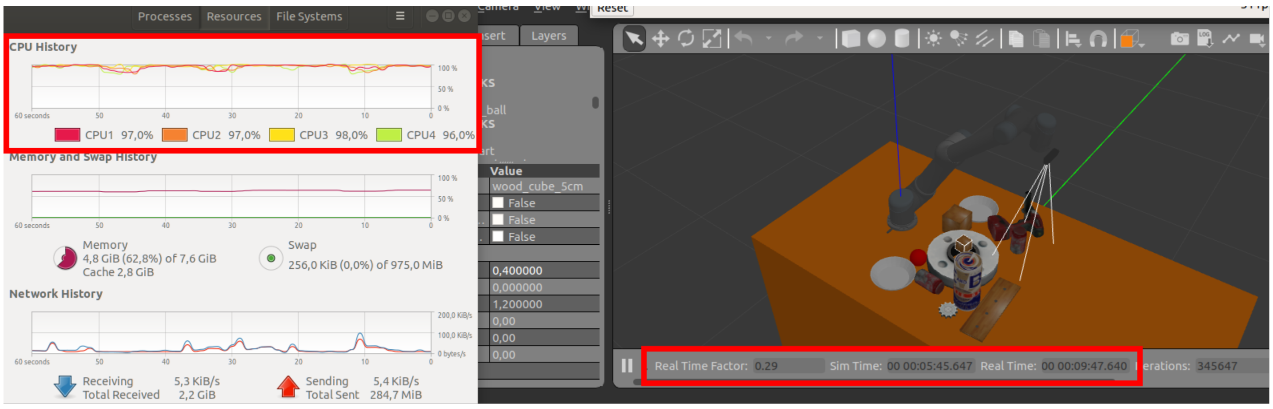


Figure 7.8: System load

During scanning of single objects compared to clutter of objects would the Real-Time Factor vary, which is a factor dependent on Real-Time compared to Simulation Time. While scanning a single object, could the factor go towards 0.6, compared to when scanning a clutter where it could go down to 0.2, meaning the time in the simulation passes slower than the time in Real-Time. This could be caused by the load and step time necessary to simulate the environment with the given objects. This could also be affecting the time calculation for execution, shown in Table 6.2.

Equipment Setup

The execution of creating and assessing a point cloud of an object is set up in a partially automated configuration: The scanning and creation of a point cloud, the saving of the cloud to a pcd-file, and the calculation and illustration by the ZMS done on that cloud. Each by themselves proves the different concepts, but should be executed fully autonomously for ease of use in a final setup. The reason for splitting the execution up was for the python version needed and supported, since ROS has support for python 2.7, while for the library used for handling and treating point clouds, pclpy, needed python 3.6-8. Also, by splitting the process up, could adaption, evaluation, and fine adjustment of the different sections be more easily made to correct and improve the setup in a modular fashion before including and building up the whole system. To make the process fully automated, could one use ROS services to launch external files and environments. This would be necessary to create a user-friendly and efficient final system that could run continuously at a plant or production line.

During the project, GitLab was used to backup and store code externally from the computer as a safety measure. If something happened to the computer, would the code still be reachable. During the setup and development phase in this project were other python libraries considered to integrate into the ROS environment, such as $python-pcl$, but this resulted in an incompatible system during integration. The result was to reinstall the entire environment to ensure that the system would be operative without any incompatibilities. The solution for this project was to use miniconda and python 3.7 with the library pclpy external from ROS to handle and use pcd-files. Therefore, it is recommended if the project is continued to frequently create image backups of the computer during the setup and development phase in addition with the use of GitLab as a safety measure and to prevent compatibility issues with packages and the ROS environment. This could limit the time spent for troubleshooting since the computer could be rolled back to an image before an unsuccessful installation, for instance.

Safety

The robot movement giving the point clouds in the result follows one particular predefined trajectory. There is no visible indication for other people or coworkers of what state the robot is executing and moving accordingly. This could be visualized by including a stack light like shown in Figure 7.9. A stack light could be used to visualize the robot's state to warn or tell nearby coworkers what to expect of the robot. By including a red-yellow-green light, could it become more intuitive for people to know what to expect. For instance, could the green light mean the robot is at a standstill or has reached the home position, a flashing yellow meaning it is moving or operating, and a flashing red meaning the robot is at a halt needing assistance. In addition, should a button be added to stop the robot as an additional safety measure if an event should occur that requires an immediate stop of the process. This button could tell the robot to move back to the home position or stop at its current position. These things could be included and tested before implementing the setup in a physical application. Additional safety measures, like marking the work area and signs, for instance, should be considered accordingly to ensure a safe work environment and prevent potential risks.

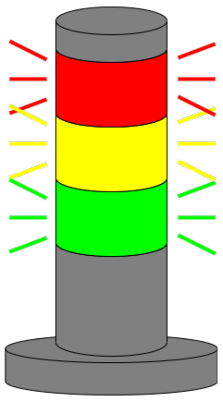


Figure 7.9: Stack light

General

The result is good for the creation of point cloud, giving a detailed representation of the surfaces detected from the object and with the flexibility to adjust the point density in the cloud. The drawback seen during this test is where objects are beyond the cut-off border, which would cause uneven distribution or blind spots on the object. The cut-off border in the simulated environment is further from the camera than stated from the camera producer. This cause is most likely in the realsense_gazebo_plugin since the test on a physical setup, using the package realsense-ros, shows this cut-off limit is closer to the camera, resulting in the ability to register surfaces as close to the camera as its producer states.

The Zero Moment Shift calculation shows how the structure of a point cloud could be interpreted and the effects of having an even and unevenly distributed point cloud of the object. The edges and regions of high curvature are clearly illustrated and distinguishable from the flat and slightly curved surfaces, which gives a foundation for further applications and algorithms to act upon the object's overall structure and surface.

The virtual environment gives a good base for future development and implementation of a Local Contact Moment algorithm. To integrate and develop this project further should a gripper be required to implement into the environment and calculation for further executing grasps of the objects in the environment.

Future Work

Regarding time is there some deviation in the execution of scanning and simulation of the Gazebo environment. This could be due to the limited processing power and capabilities of the computer running the environment. For further development of the project, should a computer with greater performance be considered to improve execution and simulation overall.

To decrease the time for calculation and execution for both the scanning procedure and ZMS calculation and additional python scripts, could parts of the python code be compiled to a machine-executable code. One compiler that could be used for a python setup is $Numba$, which uses JIT ($Just - In - Time$) to compile python code into machine code, resulting in faster execution time [13]. This has the potential to increase the efficiency of the project further.

For making the robot able to execute grasps should a gripper be implemented. A gripper from Robotiq, for instance, either 2F-85 or 2F-140, could be used. These are a two-fingered parallel-jaw gripper and have packages with ROS support, making it simpler to implement and control than creating a package from scratch.

Applying LoCoMo onto the point cloud could be done in an additional python script. Calling the ZMS-function for calculating the curvature of the cloud. The LoCoMo algorithm could since calculate the most feasible area on the point cloud with a low error in the surface fit between the gripper and object. With this and grasp ranking evaluation, would the system be ready to execute learning-free grasps in the virtual environment, ready for further testing in a physical setting.

To prevent collisions with objects or structures in the work environment should path-planning be integrated. One way this could be done is by integrating moveIT into the ROS environment. MoveIT is software that could be run in addition to ROS, and could be used to check, plan and execute a path for a robot. For this setup, could it be used to prevent a collision and plan and execute trajectories for grasping objects.

Following would the next step be to test the setup on a physical robot. This could be done by connecting the computer to a local network with a UR5 robot and configuring the environment to control the UR5 by IP.

Chapter 8

Conclusion

This project aimed to create a virtual environment in ROS as a foundation for further development of a model-free and learning-free setup. The end goal for future projects is to be able to grasp previously unseen objects without any predefined values of the object or any pre-recorded teaching data on how to execute a grasp.

A virtual environment based on ROS has been created during this project. This virtual environment consists of a robot arm, UR5, equipped with a depth camera, Intel RealSense D435i. This setup is able to construct and interpret a partial point cloud of objects placed on a workbench. The setup is able to create these point clouds without any pre-defined values for the object.

To assess the point clouds and their curvature has scripts been made to calculate the Zero Moment Shift (ZMS). With applying ZMS onto the cloud, attributes of its surface structure could be computed to prepare for further calculation and evaluation of the objects point cloud for interaction. The work done lays a foundation for developing further and integrating a learning-free algorithm, Local Contact Moment (LoCoMo), to interact and grasp previously unseen objects.

The result for this project is a virtual model-free environment for observing previously unseen objects as point clouds and evaluating their structural curvature. An algorithm for further enhancement and development has also been evaluated for implementation into the environment to achieve a learning-free grasping application. This application environment could further be used and implemented onto a physical setup for further development and testing.

The work done in this project provides a foundation for future development to reach a fully model-free and learning-free grasping system able to pick up previously unseen objects.

Bibliography

[1] A. Bicchi. “On the form-closure property of robotic grasping.” In: *IFAC Proceedings Volumes* 27.14 (1994), pp. 219–224. DOI: https://doi.org/10.1016/S1474-6670(17)47318-6.

[2] Jean Ponce. Steven Sullivan. Jean-Daniel Boissonnat and Jean-Pierre Merlet. “On Characterizing and Computing Three- and Four-Finger Force-Closure Grasps of Polyhedral Objects.” In: *IEEE International Conference on Robotics and Automation* (1993), pp. 821–827.

[3] Maxime Adjigble. Naresh Marturi. Valerio Ortenzi. Vijaykumar Rajasekaran. Peter Corke and Rustam Stolkin. “Model-free and learning-free grasping by Local Contact Moment matching.” In: (2018). DOI: https://www.researchgate.net/publication/327118653_Model-free_and_learning-free_grasping_by_Local_Contact_Moment_matching.

[4] Brayan S Zapata-Impata. Carlos M Mateo. Pablo Gil and Jorge Pomares. “Using Geometry to Detect Grasping Points on 3D Unknown Point Cloud.” In: *International Conference on Informatics in Control, Automation and Robotics* (2017), pp. 1–8. DOI: 10.5220/0006470701540161.

[5] *Intel RealSense Gazebo ROS plugin.* URL: https://github.com/pal-robotics/realsense_gazebo_plugin. (accessed: 30.03.2021).

[6] *Intel® RealSense*^{TM} *ProductFamilyD400Series.* URL: https://www.intelrealsense.com/wp-content/uploads/2020/06/Intel-RealSense-D400-Series-Datasheet-June-2020.pdf. (accessed: 25.02.2021).

[7] *ISO 10218 standards.* URL: https://www.iso.org/obp/ui/#iso:std:iso:10218:-1:ed-2:v1:en. (accessed: 03.02.2021).

[8] *ISO TS 15066 standards.* URL: https://www.iso.org/obp/ui/#iso:std:iso:ts:15066:ed-1:v1:en. (accessed: 03.02.2021).

[9] Danica Kragic and Markus Vincze. “Vision for Robotics.” In: (2015), p. 41. DOI: https://www.researchgate.net/publication/220666540_Vision_for_Robotics.

[10] Dmitry Kalashnikov. Alex Irpan. Peter Pastor. Julian Ibarz. Alexander Herzog. Eric Jang. Deirdre Quillen. Ethan Holly. Mrinal Kalakrishnan. Vincent Vanhoucke. Sergey Levine. “QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation.” In: *2nd Conference on Robot Learning* (2018), pp. 1–23. DOI: arXiv:1806.10293v3[cs.LG].

[11] H W Liu and C Q Cao. “Grasp Pose Detection Based On Point Cloud Shape Simplification.” In: *IOP Conference Series: Materials Science and Engineering* (2019), pp. 1–10. DOI: 10.1088/1757-899X/717/1/012007.

[12] Jia-Wei Li. Hong Liu. and He-Gao Cai. “On Computing Three-Finger Force-Closure Grasps of 2-D and 3-D Objects.” In: *IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION* 19.1 (2003), pp. 155–161.

[13] *Numba.* URL: http://numba.pydata.org/. (accessed: 21.04.2021).

[14] Andreas ten Pas and Robert Platt. “Using Geometry to Detect Grasp Poses in 3D Point Clouds.” In: *Con: ISRR 2015* (2015), pp. 1–16. DOI: https://www.researchgate.net/publication/281377520_Using_Geometry_to_Detect_Grasp_Poses_in_3D_Point_Clouds.

[15] *pclpy: PCL for python.* URL: https://github.com/davidcaron/pclpy. (accessed: 18.03.2021).

[16] Brayan S Zapata-Impata. Pablo Gil. Jorge Pomares and Fernando Torres. “Fast geometry-based computation of grasping points on three-dimensional point clouds.” In: *International Journal of Advanced Robotic Systems* (2019), pp. 1–18. DOI: https://journals.sagepub.com/doi/10.1177/1729881419831846.

[17] Mario Richtsfeld and Markus Vincze. “Robotic Grasping of Unknown Objects.” In: *Robot Arms* (2011), pp. 123–136. DOI: 10.5772/16799.

[18] M´aximo A. Roa and Ra´ul Su´arez. “Computation of Independent Contact Regions for Grasping 3-D Objects.” In: *IEEE TRANSACTIONS ON ROBOTICS* 25.4 (2009), pp. 839–850.

[19] *Robot Operating System*. URL: https://www.ros.org/about-ros/. (accessed: 20.01.2021).

[20] *Robot Operating System wiki*. URL: http://wiki.ros.org/. (accessed: 20.01.2021).

[21] *ROS Wrapper for Intel® RealSense™ Devices*. URL: https://github.com/IntelRealSense/realsense-ros. (accessed: 25.02.2021).

[22] Radu Bogdan Rusu and Steve Cousins. “3D is here: Point Cloud Library (PCL).” In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China, May 2011.

[23] *Universal Robot Git*. URL: https://github.com/ros-industrial/universal_robot. (accessed: 25.02.2021).

[24] *Universal Robot UR5*. URL: https://www.universal-robots.com/products/ur5-robot/. (accessed: 25.02.2021).

[25] Dan Ding. Yun-Hui Liu. Shuguo Wang. “The Synthesis of 3-D Form-Closure Grasps.” In: *2000 IEEE International Conference on Robotics & Automation* (2015), pp. 3579–3584.

[26] Jonathan Weisz and Peter K. Allen. “Pose Error Robust Grasping from Contact Wrench Space Metrics.” In: *2012 IEEE International Conference on Robotics and Automation* (2012), pp. 557–562.

[27] Xian-Feng Han. Jesse S. Jin. Ming-Jie Wang. Wei Jiang. Lei Gao. Liping Xiao. “A review of algorithms for filtering the 3D point cloud.” In: *Signal Processing: Image Communication* 57 (2017), pp. 103–112. DOI: https://doi.org/10.1016/J.IMAGE.2017.05.009.

Appendix A

Virtual Environment Builder

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://wiki.ros.org/xacro" name="ur5env">

  <link name="world"/>
  <link name="workbench">

<!-- Creating workbench -->
    <inertial>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <mass value="20"/>
      <inertia ixx="200" ixy="200" ixz="200" iyy="200" iyz="200" izz="200"/>
    </inertial>
    <visual>
      <origin xyz="0.5 0 0.5" rpy="0 0 0"/>
      <geometry>
        <box size="2 1 1"/>
      </geometry>
      <material name="Gray">
        <color rgba="0.5 0.5 0.5 0" />
      </material>
    </visual>
    <collision>
      <origin xyz="0.5 0 0.5" rpy="0 0 0"/>
      <geometry>
        <box size="2 1 1"/>
      </geometry>
    </collision>
  </link>

  <gazebo reference="workbench">
    <mu1>0.2</mu1>
    <mu2>0.2</mu2>
    <material>Gazebo/Orange</material>
  </gazebo>
  <joint name="world_joint" type="fixed">
    <parent link="world" />
    <child link="workbench" />
    <origin xyz="0 0 0" rpy="0.0 0.0 0.0"/>
  </joint>

  <xacro:arg name="transmission_hw_interface" ...
     default="hardware_interface/PositionJointInterface"/>
  <!-- Gazebo description -->
  <xacro:include filename="$(find ...
     ur_description)/urdf/common.gazebo.xacro" />
  <!-- UR5 -->
  <xacro:include filename="$(find ur_description)/urdf/ur5.urdf.xacro" />

  <xacro:ur5_robot prefix="" joint_limited="true" ...
     transmission_hw_interface="$(arg transmission_hw_interface)" />

  <!-- Camera -->
  <xacro:include filename="$(find ...
     realsense2_description)/urdf/_d435.urdf.xacro"/>
  <sensor_d435 parent="tool0">
    <origin xyz="0 -0.07 0.01" rpy="0 -1.55 -1.55"/>
  </sensor_d435>

  <joint name="base_joint" type="fixed">
    <parent link="workbench" />
    <child link="base_link" />
    <origin xyz="0 0 1" rpy="0.0 0.0 0.0"/>
  </joint>
</robot>
```

Appendix B

Virtual Environment Launch

```
<?xml version="1.0"?>
<launch>
  <arg name="limited" default="true"  doc="If true, limits joint range ...
     [-PI, PI] on all joints." />
  <arg name="paused" default="true" doc="Starts gazebo in paused mode" />
  <arg name="gui" default="true" doc="Starts gazebo gui" />
  <arg name="transmission_hw_interface" ...
     default="hardware_interface/PositionJointInterface" />

  <!-- startup simulated world -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" default="worlds/empty.world"/>
    <arg name="paused" value="$(arg paused)"/>
    <arg name="gui" value="$(arg gui)"/>
  </include>

  <param name="robot_description" command="$(find xacro)/xacro '$(find ...
     ur5_env_description)/urdf/ur5env.urdf.xacro' ...
     transmission_hw_interface:=$(arg transmission_hw_interface)" />

  <!-- push robot_description to factory and spawn robot in initial pose ...
     in gazebo -->
  <node name="spawn_gazebo_model" pkg="gazebo_ros" type="spawn_model" ...
     args="-urdf -param robot_description -model robot
          -z 0.0
          -J shoulder_pan_joint 0.0
          -J shoulder_lift_joint -1.5
          -J elbow_joint 1.6
          -J wrist_1_joint -1.5
          -J wrist_2_joint -1.5
          -J wrist_3_joint 0.0
          -unpause"
          respawn="false" output="screen"/>

  <include file="$(find ur_gazebo)/launch/controller_utils.launch"/>

  <!-- start arm controller -->
  <rosparam file="$(find ur_gazebo)/controller/arm_controller_ur5.yaml" ...
     command="load"/>
  <node name="arm_controller_spawner" pkg="controller_manager" ...
     type="controller_manager" args="spawn arm_controller" ...
     respawn="false" output="screen"/>

  <!-- Starting laserscan assembler service for construction of object ...
     point cloud-->
  <node type="point_cloud2_assembler" pkg="laser_assembler" ...
     name="my_assembler">
    <remap from="cloud" to="/voxel_grid/output"/>
    <param name="max_scans" type="int" value="400" />
    <param name="fixed_frame" type="string" value="base_link" />
  </node>

  <!-- Start voxel grid filters -->
  <include file="$(find ur5_env_description)/launch/voxelgrid.launch"/>
  <include file="$(find ...
     ur5_env_description)/launch/assembled_voxelgrid_filtering.launch"/>

</launch>
```

Appendix C

Scan Trajectory

```
#!/usr/bin/env python
# license removed for brevity
import rospy
import std_msgs.msg
import time
from sensor_msgs.msg import PointCloud2
from trajectory_msgs.msg import JointTrajectory
from trajectory_msgs.msg import JointTrajectoryPoint

import camera_PC_transform as camera_pc_tf #Imports function from python ...
   script
import pc_assembler as pc_assemble #Imports function from python script

#-----------------Variables -------------------

        #----- Scanning pose positions (given in radians) -------
home_pos =      [0, -1.5, 1.5, -1.5, -1.5, 0]#           Home/start position

first_pos =     [1.2, -1.5, 1.5, -1.0, -2.3, 0, 0]#     1. scanning position
second_pos =    [0, -2.6, 1.8, -1.4, -1.5, 0, 0]#       2. scanning position
third_pos =     [-1.5, -1.4, 1.3, -0.8, -0.9, 0, 0]#    3. scanning position
fourth_pos =    [-0.6, -1.1, 0.4, -0.7, -1.2, 0, 0]#    4. scanning position
fifth_pos =     [0, -1.0, 0.3, -0.7, -1.5, 0, 0]#       5. scanning position
sixth_pos =     [0.5, -1.0, 0.3, -0.7, -1.9, 0, 0]#     6. scanning position

pos_nr = 8 #Counter for loop. Nr of poses to visit (visit home in start ...
   and end)
sleepy_time = 4#seconds waited afther each position initiated.




#----------------- Workbench scanning script  -----------------

def scan_trajectory():
    pub = rospy.Publisher('/arm_controller/command', JointTrajectory, ...
       queue_size=10)
    rospy.init_node('scan_trajectory_pub', anonymous=True)

    take_picture = False #If position are reach for taking picture or ...
       returning to home.

    print(rospy.Time.now()) #Initiate time to get update for first ...
       position. Needs interaction/use to collect correct time
    time.sleep(1)#give one second to be able to update clock/time until ...
        next time initiated, if not arent the first position initiated
    startTime = rospy.Time.now()#used for calculation of execution time

    for i in range(pos_nr): #loop for movements
        if i==7:
            new_pos = home_pos
            pos_message = "Move robot back to home position."
            take_picture = False
        elif i==6:
            new_pos = sixth_pos
            pos_message = "Move robot to sixth position."
            take_picture = True
        elif i==5:
            new_pos = fifth_pos
            pos_message = "Move robot to fifth position."
            take_picture = True
        elif i==4:
            new_pos = fourth_pos
            pos_message = "Move robot to fourth position."
            take_picture = True
        elif i==3:
            new_pos = third_pos
            pos_message = "Move robot to third position."
            take_picture = True
        elif i==2:
            new_pos = second_pos
            pos_message = "Move robot to second position."
            take_picture = True
        elif i==1:
            new_pos = first_pos
            pos_message = "Move robot to first position."
            take_picture = True
        elif i==0:
            new_pos = home_pos
            pos_message = "Move robot to home before scan patteren initiated."
            take_picture = False
        else:
            new_pos = home_pos
            pos_message = "No position available, move to home..."
            take_picture = False

        joints_str = JointTrajectory()
        joints_str.header = std_msgs.msg.Header()
        joints_str.header.stamp = rospy.Time.now()
        joints_str.joint_names = ['shoulder_pan_joint', ...
            'shoulder_lift_joint', 'elbow_joint', 'wrist_1_joint', ...
            'wrist_2_joint', 'wrist_3_joint']
        point=JointTrajectoryPoint()
        point.positions = new_pos
        point.time_from_start = rospy.Duration(1,0)
        joints_str.points.append(point)

        pub.publish(joints_str) #gives command message to arm_controller
        rospy.loginfo(pos_message)

        time.sleep(sleepy_time)

        if take_picture == True:
            camera_pc_tf.Tf_camera_PC()
            rospy.wait_for_message('/camera_PC_transposed', PointCloud2)
            pc_assemble.assemble_voxel_pc()
            #rospy.wait_for_message('/assembled_pc_scan', PointCloud2)
            take_picture = False


    rospy.loginfo("The robot are done with the scanning of the object. ...
       Script will stop execution.")

    executionTime = rospy.Time.now() - startTime
    xTime_seconds = executionTime.to_sec()

    print("The execution time for current point cloud collection used ", ...
       xTime_seconds, " seconds.")

if __name__ == '__main__':
    try:
        scan_trajectory()
    except rospy.ROSInterruptException:
        pass
```

Appendix D

Voxel Grid Filter

```
<launch>
  <node pkg="nodelet" type="nodelet" name="pcl_manager" args="manager" ...
     output="screen" />

  <!-- Run a VoxelGrid filter to clean NaNs and downsample the data -->
  <node pkg="nodelet" type="nodelet" name="voxel_grid" args="load ...
     pcl/VoxelGrid pcl_manager" output="screen">
    <remap from="~input" to="/camera_PC_transposed" />
    <rosparam>
      filter_field_name: z <!-- Axis/direction of filtering-->
      filter_limit_min: 0.002 <!-- Filter out top of workbench -->
      filter_limit_max: 1.0
      filter_limit_negative: False
      leaf_size: 0.0005 <!-- Grid size in meters-->
    </rosparam>
  </node>
</launch>
```

Appendix E

Point Cloud Transformation

```
#!/usr/bin/env python
import tf
import rospy
import std_msgs
import numpy as np
from roslib import message
from tf import transformations
import sensor_msgs.point_cloud2 as p_c2
from sensor_msgs.msg import PointCloud2
from sensor_msgs import point_cloud2 as pc2


class Tf_camera_PC():

    def __init__(self):

        self.pc_pub = rospy.Publisher('/camera_PC_transposed', ...
           PointCloud2, queue_size=1)
        self.pc_sub = rospy.Subscriber('/camera/depth/points', ...
           PointCloud2, self.pc_callback, queue_size=1)

    def pc_callback(self,pc_data):
        assert isinstance(pc_data, PointCloud2)
        listener = tf.TransformListener()

        pc = list([x for x in pc2.read_points(pc_data, skip_nans=True, ...
           field_names=["x", "y", "z"])]) #Get points from gathered topic

        listener.waitForTransform("/base_link", ...
           "/camera_depth_optical_frame", rospy.Time(), rospy.Duration(4.0))

        (trans, quat) = listener.lookupTransform("/base_link", ...
           "/camera_depth_optical_frame", rospy.Time(0)) #Get ...
           transformation from camera frame to base frame

        Tra = transformations.translation_matrix(trans)
        Rot = transformations.quaternion_matrix(quat)
        RT = Rot + Tra - np.identity(4)

        points_transposed = np.zeros((len(pc),3)) #Create empthy list to ...
           store transposed points

        for i, point in enumerate(pc):
            point_setup = np.array((point[0],point[1],point[2], 1))
            point_new = np.matmul(RT, point_setup)
            points_transposed[i] = np.array([point_new[0], point_new[1], ...
                point_new[2]])

        header = std_msgs.msg.Header()
        header.stamp = rospy.Time.now()
        header.frame_id = "base_link"
        PointCloud_transposed = p_c2.create_cloud_xyz32(header, ...
            points_transposed)

        rospy.loginfo("Point Cloud are transposed. Ready for publication ...
            to topic.")
        self.pc_pub.publish(PointCloud_transposed)
        self.pc_sub.unregister() #Stops subscription for not continue ...
            publishing several points

if __name__ == '__main__':

    rospy.init_node('camera_PC_transform', anonymous=True)
    tf_pc_cam = Tf_camera_PC()
    rospy.spin()
```

Appendix F

Point Cloud Assembler

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('laser_assembler')
import rospy; from laser_assembler.srv import *
from sensor_msgs.msg import PointCloud2


def assemble_voxel_pc(): #Assembles point clouds recived from voxal grid ...
    filter
    rospy.wait_for_service("assemble_scans2")
    assemble_PC = rospy.ServiceProxy('assemble_scans2', AssembleScans2)
    pub = rospy.Publisher('/assembled_pc_scan', PointCloud2, queue_size=2)

    try:
        resp = assemble_PC(rospy.Time(0,0), rospy.get_rostime())
        pub.publish(resp.cloud)
        rospy.loginfo('New cloud added to object point cloud')
    except rospy.serviceException, e:
        rospy.loginfo('Didnt manage to assemble cloud...')

    return
```

Appendix G

Assembly Cloud Voxel Grid Filter

```
<launch>
  <node pkg="nodelet" type="nodelet" name="voxel_assembled_cloud" ...
      args="manager" output="screen" />

  <!-- Run a VoxelGrid filter to clean NaNs and downsample the data -->
  <node pkg="nodelet" type="nodelet" ...
      name="assebled_voxelgrid_applied_pointcloud" args="load ...
      pcl/VoxelGrid pcl_manager" output="screen">
    <remap from="~input" to="/assembled_pc_scan"/>
    <remap from="~output" to="/voxel_assembled_cloud" />
    <rosparam>
      filter_field_name: x <!-- Axis/direction of filtering-->
      filter_limit_min: 0.09 <!-- Filter out points from robot base -->
      filter_limit_max: 1.5 <!-- End of workbench -->
      filter_limit_negative: False
      leaf_size: 0.001 <!-- Grid size in meters-->
    </rosparam>
  </node>
</launch>
```

Appendix H

Zero Moment Shift Algorithm

```
import pclpy
import math
import array
from sensor_msgs.msg import PointCloud2
from pclpy import pcl
import numpy as np

# ------------- Variables -----------------------

rho = 0.004 #radius for search-sphere in meter

random_point_index = 7 #choose a random point in point cloud for ...
   illustrating ZMS

#--------------- Files --------------------

pcd_file_in = 'zms_070521/CokeCan.pcd'
zms_array_list_file = "zms_array.txt" #create file before running
zms_demo_list_file = "zms_demo.txt" #create file before running

pcd_file_out = ...
   "/home/tommen/Desktop/cloud_handling/zms_070521/ZMS_onCokeCan_withZMSarray_testtest.
   #creates new file
pcd_demo_file_out = ...
   "/home/tommen/Desktop/cloud_handling/zms_070521/ZMS_pointCloud_demo.pcd" ...
   #creates new file



# -------------- ZMS algorithm --------------------

point_cloud = pclpy.pcl.PointCloud.PointXYZ()
point_cloud_rgb = pclpy.pcl.PointCloud.PointXYZRGB()
calc_point_cloud = pclpy.pcl.PointCloud.PointXYZRGB()
zms_demo_point_cloud = pclpy.pcl.PointCloud.PointXYZRGB() #For ...
   demonstration of ZMS in dataset

pcl.io.loadPCDFile(pcd_file_in, point_cloud)
zms_array_file = open(zms_array_list_file, "w")
zms_demo_file = open(zms_demo_list_file, "w") #File for demonstration of ZMS

point_cloud_rgb.resize(point_cloud.size())
calc_point_cloud.resize(point_cloud.size())

pointIndexRadiusSearch = pclpy.pcl.vectors.Int()
```

```
pointRadiusSquaredDistance = pclpy.pcl.vectors.Float()

resolution = 1.0 #Octree resolution
octree = pclpy.pcl.search.Octree.PointXYZ(resolution)
octree.setInputCloud(point_cloud)

ZMS_vectors = np.zeros([point_cloud.size(),3])
ZMS_demo_vectors = np.zeros([3,3])

for i in range (0,point_cloud.size()):#Loop for calculating ZM points and ZMS
    octree.radiusSearch(point_cloud,i,rho, pointIndexRadiusSearch, ...
        pointRadiusSquaredDistance)

    point_cloud_rgb.points[i].x = point_cloud.points[i].x
    point_cloud_rgb.points[i].y = point_cloud.points[i].y
    point_cloud_rgb.points[i].z = point_cloud.points[i].z
    point_cloud_rgb.points[i].rgb =  255 << 16 | 255 << 8 | 255

    if i == random_point_index:
        point_cloud_rgb.points[i].rgb =  255 << 16 | 0 << 8 | 0 #mark ...
            search point red in object point cloud

        zms_demo_point_cloud.resize(len(pointIndexRadiusSearch)+1)
        print(len(pointIndexRadiusSearch), " number of points in sphere.")

        ZMS_demo_vectors[0][0] = point_cloud.points[i].x#store "search ...
            point" in first row
        ZMS_demo_vectors[0][1] = point_cloud.points[i].y
        ZMS_demo_vectors[0][2] = point_cloud.points[i].z

    x_cor = 0.0
    y_cor = 0.0
    z_cor = 0.0

    for j in range (0,len(pointIndexRadiusSearch)):
        x_cor = x_cor + point_cloud.points[pointIndexRadiusSearch[j]].x
        y_cor = y_cor + point_cloud.points[pointIndexRadiusSearch[j]].y
        z_cor = z_cor + point_cloud.points[pointIndexRadiusSearch[j]].z

        if i == random_point_index:#used to store and show specific search ...
            sphere and illustrate ZMS calculation
             zms_demo_point_cloud.points[j].x = ...
                 point_cloud.points[pointIndexRadiusSearch[j]].x
             zms_demo_point_cloud.points[j].y = ...
                 point_cloud.points[pointIndexRadiusSearch[j]].y
             zms_demo_point_cloud.points[j].z = ...
                 point_cloud.points[pointIndexRadiusSearch[j]].z
             zms_demo_point_cloud.points[j].rgb =  255 << 16 | 255 << 8 | 255


             if (point_cloud.points[pointIndexRadiusSearch[j]].x ==  ...
                 point_cloud.points[i].x) and ...
                 (point_cloud.points[pointIndexRadiusSearch[j]].y ==  ...
                 point_cloud.points[i].y) and ...
                 (point_cloud.points[pointIndexRadiusSearch[j]].z ==  ...
                 point_cloud.points[i].z):
                  zms_demo_point_cloud.points[j].rgb =  255 << 16 | 0 << 8 | ...
                      0#colour the search point red

    #Calculating Zero Moment
    try:
        x_cor = x_cor/len(pointIndexRadiusSearch)
        y_cor = y_cor/len(pointIndexRadiusSearch)
        z_cor = z_cor/len(pointIndexRadiusSearch)

    except ZeroDivisionError:
        print("Division by Zero.")
        x_cor = point_cloud.points[i].x
        y_cor = point_cloud.points[i].y
        z_cor = point_cloud.points[i].z

    calc_point_cloud.points[i].x = x_cor
    calc_point_cloud.points[i].y = y_cor
    calc_point_cloud.points[i].z = z_cor
    calc_point_cloud.points[i].rgb =  0 << 16 | 0 << 8 | 0

    if i == random_point_index:
        zms_demo_point_cloud.points[j+1].x = x_cor
        zms_demo_point_cloud.points[j+1].y = y_cor
        zms_demo_point_cloud.points[j+1].z = z_cor
        zms_demo_point_cloud.points[j+1].rgb = 0 << 16 | 0 << 8 | 0

        ZMS_demo_vectors[1][0] = x_cor#store ZM in second row
        ZMS_demo_vectors[1][1] = y_cor
        ZMS_demo_vectors[1][2] = z_cor

        ZMS_demo_vectors[2][0] = calc_point_cloud.points[i].x - ...
           point_cloud.points[i].x#store ZMS in third row
        ZMS_demo_vectors[2][1] = calc_point_cloud.points[i].y - ...
           point_cloud.points[i].y
        ZMS_demo_vectors[2][2] = calc_point_cloud.points[i].z - ...
           point_cloud.points[i].z
        np.savetxt(zms_demo_file,ZMS_demo_vectors)


    #Calculate Zero Moment Shift
    ZMS_vectors[i][0] = calc_point_cloud.points[i].x - point_cloud.points[i].x
    ZMS_vectors[i][1] = calc_point_cloud.points[i].y - point_cloud.points[i].y
    ZMS_vectors[i][2] = calc_point_cloud.points[i].z - point_cloud.points[i].z

    pointIndexRadiusSearch.clear()
    pointRadiusSquaredDistance.clear()


np.savetxt(zms_array_file,ZMS_vectors)

#create a common point cloud to write the original and the ZM-calculation
#wrappers do not consist with manual/web showcase, therefor using a ...
   for-loop instead of storing directly
zms_calc_point_cloud  = pclpy.pcl.PointCloud.PointXYZRGB()
zms_calc_point_cloud.resize(calc_point_cloud.size()+point_cloud_rgb.size())

for i in range (calc_point_cloud.size()+point_cloud_rgb.size()):

    if (i < calc_point_cloud.size()):
        zms_calc_point_cloud.points[i].x = calc_point_cloud.points[i].x
        zms_calc_point_cloud.points[i].y = calc_point_cloud.points[i].y
        zms_calc_point_cloud.points[i].z = calc_point_cloud.points[i].z
        zms_calc_point_cloud.points[i].rgb = calc_point_cloud.points[i].rgb
    else:
        zms_calc_point_cloud.points[i].x = ...
           point_cloud_rgb.points[(i-calc_point_cloud.size())].x
        zms_calc_point_cloud.points[i].y = ...
           point_cloud_rgb.points[(i-calc_point_cloud.size())].y
        zms_calc_point_cloud.points[i].z = ...
           point_cloud_rgb.points[(i-calc_point_cloud.size())].z
        zms_calc_point_cloud.points[i].rgb = ...
           point_cloud_rgb.points[(i-calc_point_cloud.size())].rgb

zms_array_file.close()
zms_demo_file.close()
pcl.io.savePCDFileASCII(pcd_file_out, zms_calc_point_cloud)
pcl.io.savePCDFileASCII(pcd_demo_file_out, zms_demo_point_cloud)
```

Appendix I

ROS Traffic Nodes

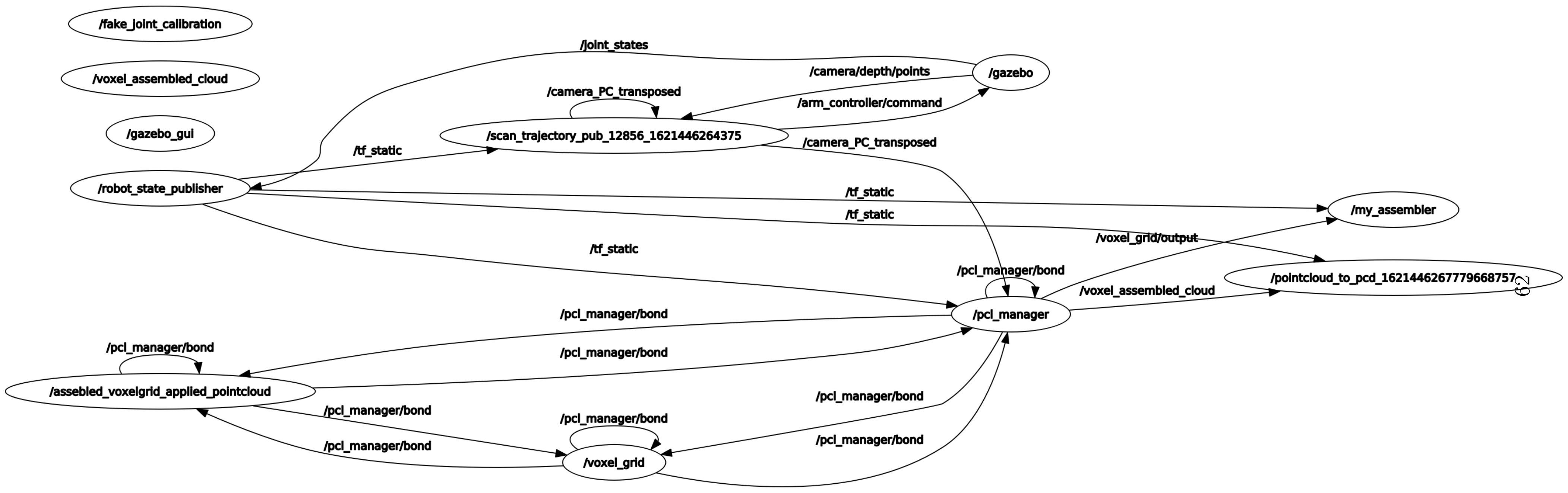


Figure I.1: ROS traffic nodes

Appendix J

ROS Traffic Topics

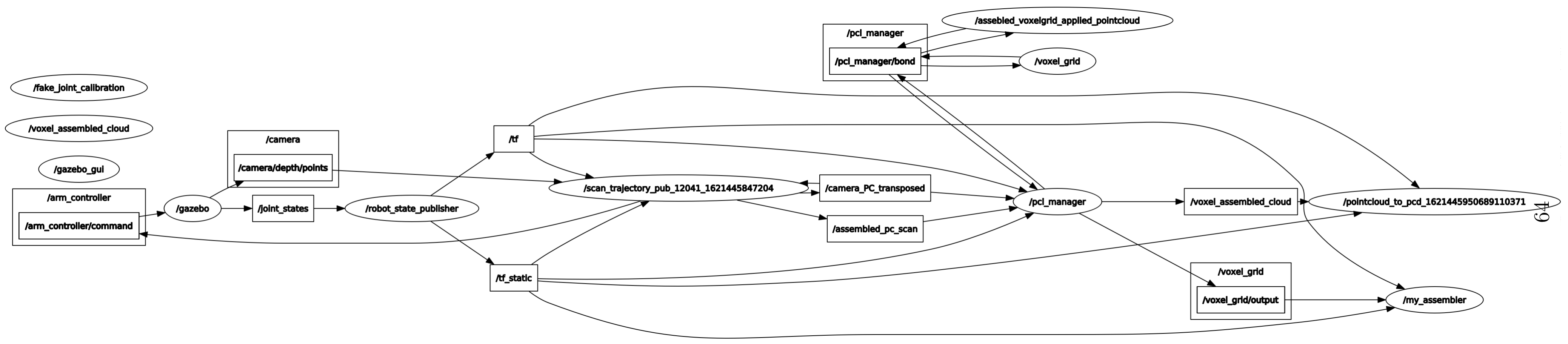


Figure J.1: ROS traffic topics