NTNU

Norwegian University of
Science and Technology

3D robot vision using multiple cameras

Sindre Raknes

Subsea Technology
Submission date: May 2016
Supervisor: Olav Egeland, IPK

Norwegian University of Science and Technology
Department of Production and Quality Engineering

NTNU

Fakultet for ingeniørvitenskap og teknologi
Institutt for produksjons- og kvalitetsteknikk

Vår dato 2016-01-15 Vår referanse OEG

MASTEROPPGAVE 2016

Sindre Raknes

Tittel: 3D-robotsyn ved kombinasjon av flere kamera

Tittel (engelsk): 3D robot vision using multiple cameras

Oppgavens tekst

Ved bestemmelse av geometrien til et objekt fra flere bilder må bildeinformasjonen fra de ulike kameraene kombineres. I denne oppgaven skal dette studeres for robotsyn ved bruk av flere 3D-kamera. Dette skal brukes til gjenkjenning og posisjonsbestemmelse av objekter med kjent 3D geometri. Systemet skal testes ut i instituttets Agilus-lab.

1. Beskriv metoder for å kombinere bildeinformasjon fra ulike kameraer.
2. Beskriv metoder for gjenkjenning av objekter med kjent 3D geometri.
3. Lag et system for å automatisk kalibrere eksterne parameter for kameraer.
4. Bruk dette til gjenkjenning av objekter og til bestemmelse posisjon og orientering, med bildeinformasjon fra flere 3D kamera.
5. Prøv ut systemet i eksperimenter.

Oppgave utlevert: 2016-01-15

Innlevering: 2016-06-11

Utført ved Institutt for produksjons- og kvalitetsteknikk, NTNU.

Trondheim, 2016-01-15

Professor Olav Egeland
Faglærer

Preface

This is the concluding Master's thesis of the 2 year Master study programme Subsea Technology at NTNU. The work was carried out between January and June 2016.

In the last year of this study programme I had the chance to choose specialization in production technology and automation, with focus on robotics. Given my earlier Bachelor degree in automation engineering and my interest in the robotics field, this felt like the right choice for me.

In my previous semester I had the opportunity to start researching in the robotics field together with a fellow student, where we developed robotic welding system without the use of welding fixtures. This was done with the use of force control and 3D-vision. This work caught my interest in researching and testing smarter 3D-vision systems for robotics, and resulted in the topic "3D robot vision using multiple cameras".

Trondheim, May 31, 2016

Sindre Raknes

Acknowledgment

I would like to express my gratitude to Professor Olav Egeland for giving me the opportunity to work on an interesting, innovative and challenging project, as well as helpful supervising during the project work.

I would also like to thank PhD candidate Adam Leon Kleppe for his feedback, cooperation and guidance working with the robot cell. His contribution made it easier to start testing in the thesis.

Furthermore, I would like to thank all my friends and fellow students for creating a very pleasant working environment during our thesis work.

S.R.

Summary

3D computer vision has in recent years become more popular in regards to robotic vision systems. This thesis looks into the possibility of using a multiple 3D camera setup for robotic vision. The approach is to reconstruct a scene in 3D from multiple camera viewpoints, extract an object from the scene and perform object recognition to find the position and orientation of an object with a known 3D geometry.

The output point cloud from the cameras were merged together to reconstruct the entire scene. To be able to reconstruct the scene, the extrinsic parameters for each camera was required. A system was implemented to automatically calibrate the cameras extrinsic parameters, that was further used to transform the point clouds.

A object recognition system was also implemented, based on the Point Cloud Library (PCL). This system supports the required filters and algorithm to be able to detect an object to find its position and orientation in the scene. Object recognition was achieved by implementing a proposed recognition pipeline, based on local feature descriptors. The system is also interfaced to a robot controller via Robot Operating System (ROS).

The experimental results indicate some variation in the objects position and less variation in the objects orientation. This variation appears to originate due to poor repeatability in depth measurements with the Microsoft Kinect™v2. The results are believed to be accurate enough for a grasping task performed by a robot.

Sammendrag

I de seneste årene har bruken av 3D kameraer for robotsyn blitt mer populært. Denne avhandlingen tar for seg robotsyn ved bruk af flere 3D-kamera. Fremgangsmåten går ut på å rekonstruere et objekt og omgivelsene ved å kombinere bildeinformasjonen fra de ulike kameraene. Fra dette skal et objekt med kjent 3D-geometri lokaliseres (posisjon og orientasjon) ut i fra de kombinerte bildedataene.

Punktskyene fra de ulike kameraene ble slått sammen for å rekonstruere omgivelsene. For å oppnå dette, må man vite de eksterne parametrene til kameraene. Et system ble implementert for å automatisk kalibrere desse parametrene for hvert kamera, som videre ble brukt til å transformere punktskyene sammen.

I tillegg til dette ble et system for å gjenkjenne objekter implementert, basert på biblioteket Point Cloud Library (PCL). Dette systemet støtter de nødvendige filter og algoritmer som er nødvendig for å gjenkjenne et objekt med kjent 3D-geometri. Gjenkjenningen ble oppnådd ved å følge en foreslått fremgangsmåte basert på lokale deskriptorer. Systemet er også satt opp ved hjelp av Robot Operating System (ROS), og har mulighet til å kontrollere roboter.

Eksperimentene viser at det forekommer variasjon i objektets posisjon. Derimot er det lite variasjon i objekters orientering. Variasjonen set ut til stamme fra den dårlige repeterbarheten til dybdemålingen til 3D kameraet Kinect™v2. Resultatene antas likevel å være nøyaktige nok for å tas i bruk i en robotisert oppgave.

Contents

List of Figures

List of Tables

Terms and Abbreviations

Manipulator A robot represented as a chain of rigid bodies which are connected by means of revolute or prismatic joints

Revolute joint A joint which provides single-axis rotation

Prismatic joint A joint which provides linear sliding

Base The start point coordinate system of a robot

End-effector The end point coordinate system of a robot

RPY Roll-Pitch-Yaw

Roll Rotation about the z-axis

Pitch Rotation about the y-axis

Yaw Rotation about the x-axis

PCL Point Cloud Library

CAD Computer-aided design

ICP Iterative Closest Point

C++ Programming language

GUI Graphical User Interface

Qt C++ programming environment with GUI builder

XML Extensible Markup Language

ROS Robot Operating System

RViz Visualization tool in ROS

AR Augmented Reality

ToF Time-of-Flight

SL Structured Light

Keypoint A distinctive and repeatable point in a point cloud.

Feature descriptor Description of a point and its surroundings

Cluster A part of a point cloud

PFH Point Feature Histogram

FPFH Fast Point Feature Histogram

SHOT Signature of Histograms of Orientations

VFH Viewpoint Feature Histogram

SIFT Scale Invariant Feature Transform

DoG Difference-of-Gaussian

Chapter 1: Introduction

1.1 Background

In the industry today, there are millions of arm-type robots built and put to work at tasks such as welding, painting, loading and unloading, assembling, packing and palletising. In the recent years the use of robots has led to increased productivity and improved product quality. This is done without harming the job market, and has helped to keep the industry alive in high-labour cost countries (Corke, 2011).

In high-labour cost countries, such as Norway, it is expensive to deal with low batch and high value manufacturing. Thereby, there is a need of reducing manufacturing cost, increase productivity, improve production quality and eliminating dangerous tasks for human operators. This can be achieved by using programmable automation and robotics. A programmable system is more versatile and is capable of manufacturing different items, since it is reprogrammable and can perform multiple functions (Siciliano et al., 2009).

Such solutions are more and more commonly available, and rely on good sensors and sensor technology. One of these sensor technologies is the use of vision based systems. The most common vision systems are based on using 2D cameras, but the increase of low-cost 3D sensors (like the Microsoft Kinect™) has led to an increased work in 3D vision. 3D computer vision offer new and exciting opportunities, like human pose estimation, activity recognition, object and people tracking, 3D mapping and localization, etc.

These types of systems are often based on using a single 3D sensor, that can only capture a scene and its surroundings from one viewpoint. By looking into the concept of combining the output from multiple 3D sensors, to reconstruct a full scene, the quality and efficiency of the vision system can be improved. The main focus in this thesis is to look into methods that can combine the sensor data and use this combined data in an object recognition system.

1.2 Problem description

To reconstruct the geometry of an object from multiple cameras, the output information from the different cameras must be combined. In this thesis, the objective is to study robotic vision by using a multiple 3D camera setup.

The main objectives of this project work are

1. Describe methods to combine the output information from multiple 3D cameras.
2. Describe methods for recognition of objects with known 3D geometry.
3. Implement a system for automatic calibration of extrinsic camera parameters.
4. Use the combined output information from multiple 3D cameras to find an objects position and orientation.

5. Do experiments on the system at the laboratory at the *Department of Production and Quality Engineering (IPK), NTNU*.

1.3 Structure of the report

The structure of this report is as follows:

- The relevant theory and background information on 3D computer vision required for this thesis is given in Chapter 2. Chapter 3 introduces some basic theory behind robotics, kinematics and the Robot Operating System (ROS).
- In Chapter 4, it follows a description of how the background and theory can be applied to achieve the goal of this project.
- Chapter 5 presents results from different experiments carried out during the project work.
- Chapter 6 summarizes and discusses the results from Chapter 5. This chapter is based on opinions and give reasoning for the results and what could have been done differently.
- In Chapter 7, a brief conclusion of this project work is presented, along with recommendations for further work.
- The Appendix contains source code from the project work.

Chapter 2: 3D Computer Vision

2.1 Introduction

In recent years several range sensors (3D cameras) have been developed and introduced at a reasonable price. The most common 3D camera in the market is the Microsoft Kinect™, which has been developed in conjunction with the video game console Microsoft Xbox. The first Kinect™was introduced with the Xbox 360 and is based on the range sensing principle called Structured Light (SL), whereas the second Kinect™v2 (which is used in this thesis) is based on Time-of-Flight (ToF).

SL is an active stereo-vision technique, where a known pattern (grids, laser, horizontal bars) is projected towards a scene which is observed from a camera from a different direction. These patterns deform when striking a surface, allowing extraction of depth information (Sarbolandi et al., 2015).

ToF on the other hand, is based on measuring the time that light emitted by an illumination unit requires to travel to a scene and back to the sensor (Grzegorzek et al., 2013).

2.2 Point Clouds

The output from a 3D camera is called a *point cloud*. A point cloud is described as a set of data points defined in a coordinate system as

$$P = \{p_1, ..., p_M\} \quad p^T \in R^N \tag{2.1}$$

where M is the number of points and N is the dimensionality of the space (Shao et al., 2014).

Equation 2.1 is known as a *depth cloud*, and in addition we are often provided with additional information regarding each point in the cloud. This additional information (feature) is described as

$$F = \{f_1, ..., f_M\} \quad f^T \in R^K \tag{2.2}$$

and is often called a *feature cloud*. By combining equation 2.1 and 2.2 we can acquire a point cloud with XYZ-data together with a feature, for example RGB-data (color). This yields additional information for further use in point cloud processing (Shao et al., 2014).

The resulting point cloud from a ToF 3D camera is classified as a *organized point cloud*, which is a point cloud data-set organized by rows and columns in a logical manner. In contrast, an unorganized point cloud have no organized structure. The advantages of a organized point cloud is knowing the relationship between adjacent points, most point cloud processing operations will be much more efficient and effective (PCL, 2012).

2.3 Point Cloud Processing

Point Cloud Processing is the process of preparing a point cloud for more advanced algorithms. This typically involves filtering, keypoint extraction and feature estimation. The Kinect™v2 has a depth resolution of $512x424$, which gives a total of 217088 points in the raw point cloud data. This raw point cloud can be subjected to noise measurements and consists of a massive amount of data, which increases computation time and can reduce the effect of recognition algorithms.

This Section (2.3) will give an introduction to the most common point cloud processing methods used in this thesis.

2.3.1 Voxel Grid filter

A Voxel Grid filter is a down-sampling filter used to reduce the amount of points in a point cloud. A reduced amount of points improves the computation time and the efficiency of other algorithms. A *voxel* can be thought of as a tiny box in 3D space, whereas a *voxel grid* is a set of tiny boxes in 3D space. Each voxel has a specified size (volume) which contains a certain amount of points. These points present in each voxel will be approximated with their centroid, and thereby reducing the amount of points in the cloud. Choosing the voxel centroid instead of the voxel center as an average gives a more accurate representation. There are other alternatives for down-sampling a point cloud, for example random sampling of points, but those methods to not give the accuracy benefit like the voxel grid filter does (PCLvoxelgrid).

2.3.2 Passthrough filter

A passthrough filter is a very simple filter which cut off values along a specified dimension (according to the camera coordinate system), inside or outside a given range. This is a very fast filter that can remove large outlier noise as well as removing for example the floor from a point cloud.

2.3.3 Outlier removal filter

3D point clouds are often prone to noise measurements, especially outliers which corrupt the results. A version of an outlier removal filter is the statistical outlier removal filter, which performs a statistical analysis on each points neighborhood, and trimming the points that do not meet a certain criteria. The sparse outlier removal is based on the computation of the distribution of point to neighbor distances in the point cloud. For each point, the mean distance from the point to all neighbors are computed. It is assumed that the resulted distribution is Gaussian, with a mean and standard deviation. All points whose mean distances are outside a predefined interval defined by the global distances mean and standard deviation are removed and considered as outliers (PCLoutlier).

2.3.4 Fast Bilateral filter

The bilateral filter is a nonlinear filter that smooths a signal (removes noise) while preserving strong edges and is commonly used in 2D image processing. Paris and Durand (2006) proposed

a fast bilateral filter in 2006 in higher dimensional space (3D point clouds) expressed as a convolution followed by nonlinear operations:

$$linear: \quad (w^{bf} i^{bf}, w^{bf}) = g_(\sigma_s, \sigma_r) \otimes (wi, w) \tag{2.3}$$

$$nonlinear: \quad I_{\boldsymbol{p}}^{bf} = \frac{w^{bf}(\boldsymbol{p}, \boldsymbol{I_p}) i^{bf}(\boldsymbol{p}, \boldsymbol{I_p})}{w^{bf}(\boldsymbol{p}, \boldsymbol{I_p})} \tag{2.4}$$

The parameter σ_s is the size of the spatial neighborhood per point, and σ_r controls how much an near lying point is down-weighted due to intensity difference. The functions $w^{bf} i^{bf}$ and w^{bf} (equation 2.3) is evaluated at a point $(\boldsymbol{p}, \boldsymbol{I_p})$, and that operation is called *slicing*. The second equation (2.4) is the division, which in this case slicing and dividing commutes since the result is independent of their order due to $g_(\sigma_s, \sigma_r)$ is positive and w values are 0 and 1, which ensures that w^{bf} is positive.

2.3.5 Segmentation and Clustering

Processing and storing large point cloud data is often the biggest bottleneck of a 3D processing system. In a general manner, given a point cloud P_1 with less points than a point cloud P_2, processing P_1 will take less time and be more efficient.

The concepts segmentation and clustering are very similar, especially when used together to achieve a goal (hereby referred to as segmentation). Segmentation is the process of breaking apart a point cloud into two or more groups of points, which is called clusters. This gives the advantage of processing each cluster independently of each other, reducing processing time and ignoring other parts of the cloud. A simple example of how segmentation works, given a cloud containing a table with 2 different parts on top, the result will give 3 different clusters: the table, part 1 and part 2.

Segmentation can be achieved by a variety of methods:

- Euclidean: using distance between points.
- Conditional euclidean: using distance between points including custom requirement.
- Region growing: using normals and curvatures.
- Color: using RGB data.
- Model fitting: Random Sample Consensus (RANSAC).

In this thesis, the object to be detected is located on top of a table and the goal of the segmentation is to create a cluster of the object and remove the surroundings. The most common way of creating a cluster of a table is using model fitting. Most model fitting algorithms are based on RANSAC (Martin A. Fischler, 1981), which is an iterative method to determine if a part (in this case, something in the cloud) fits a certain mathematical model, for example a plane, sphere or circle. The plane fitting RANSAC algorithm is as follows (Konstantinos G., 2010):

Algorithm 1 RANSAC plane fitting

1: Select randomly three non-collinear unique points (needed for mathematical plane model).
2: Use the three points to solve the parameters for the plane model ($a\boldsymbol{x} + b\boldsymbol{y} + c\boldsymbol{z} + d = 0$).
3: Compute the distances from all $\boldsymbol{p} \in P$ to the plane model.
4: Check if the distance d on all points $\boldsymbol{p}$ falls between a specified threshold $0 \leq |d| \leq |d_t|$, and keep those points.
5: Repeat steps 1 through 4 until the best match of a plane is found (N times).

Once the table has been segmented through RANSAC, we can remove it from the point cloud, making way for simple euclidean segmentation for the object. Euclidean segmentation checks the distance d between two points and checks if the distance is within a specified maximum distance d_m. If $d \leq d_m$, both points are considered to be part of the same cluster. This process is repeated until no new point can be added to the cluster.

2.4 Keypoints

Computing features (see Section 2.5) for every point in the point cloud requires a lot of computation time, so it makes little sense to compute it for every point. Keypoints are distinctive and repeatable points in the point cloud that are likely to be present in a point cloud regardless of viewpoint, noise and time. Thus, computing features on keypoints allows efficient object description and reduced error when estimating corresponding points.

A good keypoint detector should have the following properties:

- Sparseness - a small amount of the points in the cloud are keypoints.
- Repeatability - a keypoint should be found on multiple point clouds at a corresponding location.
- Distinctiveness - the surrounding area around a keypoint should have a unique shape that can be described by a feature descriptor.

An evaluation of keypoint detectors, from Filipe and Alex (2014), concluded that the Scale Invariant Feature Transform (SIFT) yield the best scores in term of repeatability. SIFT was originally developed for 2D images by Lowe (2004) and further derived for 3D in Point Cloud Library (PCL) (Rusu and Cousins, 2011). The derivation to a 3D detector has been based on replacing the role of the intensity of an pixel in the original 2D algorithm by the principal curvature of a point within the 3D cloud (Hansch et al., 2014). Algorithm 2 gives a brief overview over the major steps in the SIFT keypoint detector. SIFT keypoints are positioned at the scale-space extrema of the Difference-of-Gaussian (DoG) function (equation 2.5).

$$D(x, y, z, \sigma_j) = G(x, y, z, \sigma_{j+1}) - G(x, y, z, \sigma_j) \tag{2.5}$$

The Gaussian Scale-Space (first step of the algorithm) is created by filtering with a voxel grid filter of different sizes and a blur filter by a radius search for each point, and then computing the intensity as weighted average of the found neighbors. Further on, for each two adjacent point clouds, a new DoG point cloud is computed. In this DoG point cloud, all of the resulting points have the same position as in the involved point cloud, but their intensity value represent

the difference of the intensity values of the original points. A point is marked as a keypoint candidate if it has the lowest or highest DoG value among all its neighboring points. The keypoint candidates are then examined for possible elimination if the local principal curvatures of the intensity profile around the keypoint exceeds a specified threshold value.

Algorithm 2 SIFT Keypoint Detection

1: Create Gaussian Scale-Space.
2: Compute Difference-of-Gaussians.
3: Find keypoint candidates.
4: Remove keypoints in low curvature areas.

There is also a large amount of other notable keypoint detectors, such as:

- Harris (Harris and Stephens, 1988)
- Smallets Univalue Segment Assimilating Nucleus (SUSAN) (Smith and Brady, 1997)
- Intrinsic Shape Signatures (ISS) (Zhong, 2009)
- Uniform Sampling (Voxel Grid, see Section 2.3.1)
- Normal Aligned Radial Feature (NARF) (Steder et al., 2010)

2.5 Features

Applications that need to compare points in a point cloud require some characteristics and metrics to be able to distinguish between geometric surfaces. This is where the concept of features (also known as descriptors) has its role. To determine a feature of a query point in a cloud $\boldsymbol{p}_q$, the information about the neighboring points P^k can be used to estimate a local feature representation that captures the geometry of the underlying sampled surface around $\boldsymbol{p}_q$ (Rusu, 2010). Given a query point $\boldsymbol{p_q}$, a set of points nearby the query point $P^k = \{\boldsymbol{p}_1^k ... \boldsymbol{p}_2^k\}$, the concept of a neighbor is given as:

$$||\boldsymbol{p_i^k} - \boldsymbol{p_q}||_x \leq d_m \tag{2.6}$$

where d_m is a specified maximum allowed distance from the query point to a neighbor, and $||\cdot||_x$ is an example L_x Minkowski norm (other distance norms can be used). The number of neighbors in P^k can be limited to a value k, and the point feature can be described as a vector function F that describes the local geometric information captured by P^k, around $\boldsymbol{p_q}$:

$$F(\boldsymbol{p_q}, P^k) = \{x_1, x_2, x_3 ... x_n\} \tag{2.7}$$

where $x_i, i \in \{1...n\}$ is the dimension i of the resultant feature vector representation. Comparing two different points $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ results in comparing the point feature vectors F_1 and F_2 in some measure. Let Γ be the comparing measure describing the difference between $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$, and d the distance measure, then:

$$\Gamma = d(F_1, F_2) \tag{2.8}$$

The two points are considered to be similar if the distance d is close to 0, and considered distinct if d is large.

A good point feature has the following properties:

- Rotation and translation in the point cloud should not influence the resultant feature vector F.
- The point feature should be resistant to varying sampling density.
- The point feature must retain the same or very similar values in its feature vector F in the presence of moderate noise.

The next sections will introduce some of the most common features used in registration (Section 2.6.1) and object detection (Section 2.7). An evaluation of feature descriptors were conducted by Hansch et al. (2014), concluded that the Fast Point Feature Histogram (FPFH) and the Signature of Histograms of Orientations (SHOT) are the best feature candidates available at the time being.

2.5.1 Normals

The most common feature in 3D processing is the normal, which is used in most advanced algorithms like rendering, making visibility computation, answering inside-outside queries, surface reconstruction, etc (Mitra et al., 2003). Many different normal estimation methods exist, but the simplest and most common one was proposed by Berkmann and Caelli (1994), which is based on the first order 3D plane fitting. The normal estimation of a point on the surface is approximated by the problem of estimating the normal of a plane tangent to the surface, which in turn becomes a least-square plane fitting estimation problem in P^k (Shakarji, 1998). The plane is defined by two parameters,

$$\begin{aligned} \boldsymbol{x} &\quad - \quad \text{a point on the plane} \\ \vec{n} &\quad - \quad \text{normal vector} \end{aligned}$$

where the distance from a point $\boldsymbol{p_i} \in P^k$ is defined as $d_i = (\boldsymbol{p_i} - \boldsymbol{x}) \cdot \vec{n}$. The values of $\boldsymbol{x}$ and $\vec{\boldsymbol{n}}$ are computed in a least-square sense, so that $d_i = 0$. Using

$$\boldsymbol{x} = \overline{\boldsymbol{p}} = \frac{1}{k} \cdot \sum_{i=1}^{k} \boldsymbol{p_i} \tag{2.9}$$

as the centroid of P^k, the solution for $\vec{n}$ is found by analyzing the eigenvalues and eigenvectors of the covariance matrix $C \in R^{3x3}$ of P^k. The covariance matrix is expressed as

$$C = \frac{1}{k} \sum_{i=1}^{k} \xi_i \cdot (\boldsymbol{p_i} - \overline{\boldsymbol{p}}) \cdot (\boldsymbol{p_i} - \overline{\boldsymbol{p}})^T, \quad C \cdot \vec{v}_j = \lambda_j \cdot \vec{v}_j, \quad j \in \{0, 1, 2\} \tag{2.10}$$

where ξ_i represents a possible weight for $\boldsymbol{p_i}$ (usually equals 1). C is symmetric and positive semi-definite, and its eigenvalues are real numbers ($\lambda_j \in \mathbb{R}$). The eigenvectors $\vec{\boldsymbol{v}}_{\boldsymbol{j}}$ form an

orthogonal frame corresponding to the principal components of P^k. If $0 \leq \lambda_0 \leq \lambda_1 \leq \lambda_2$, the eigenvector $\vec{\boldsymbol{v}}_{\boldsymbol{0}}$ corresponding to the smallest eigenvalue λ_0 is therefore the approximation of $+\vec{\boldsymbol{n}} = (n_x, n_y, n_z)$ or $-\vec{\boldsymbol{n}}$ (Rusu, 2009).

Even though estimating normals is extremely fast and simple to compute, they lack enough detail to be used as a feature for matching since most point clouds will have many similar normals regardless of neighboring points. Even though, they are often the foundation for or a part of more advanced feature representations.

2.5.2 Point Feature Histograms

Point Feature Histogram (PFH) features are based on the relationship between k-neighboring points and their estimated surface normals. In general, it attempts to estimate the best possible surface variations by taking into consideration all the interactions between the direction of all the k-neighboring estimated normals. The resulted hyperspace is thereby dependent on the quality of the normal estimation step at each point (Rusu, 2009).

The first step in estimating the PFH is as mentioned estimating the surface normals. This step can either be computed on demand, or as a prior on all points $\boldsymbol{p_i} \in P$ before estimating the PFH. After the normals have been estimated, the next step is to compute the relative distance between two points. Given two points, $\boldsymbol{p_i}$ and $\boldsymbol{p_j}$ along with their associated normals $\vec{\boldsymbol{n}}_{\boldsymbol{i}}$ and $\vec{\boldsymbol{n}}_{\boldsymbol{j}}$, a fixed Darboux coordinate frame is defined at one of the points (see Figure 2.1). For the frame to be defined uniquely, given that $\boldsymbol{p_s}$ is defined as the source point and $\boldsymbol{p_s}$ as the target point, we have that (Rusu et al., 2008):

$$\text{if: } \arccos(\vec{\boldsymbol{n}}_{\boldsymbol{i}} \cdot \vec{\boldsymbol{p}}_{\boldsymbol{ji}}) \leq \arccos(\vec{\boldsymbol{n}}_{\boldsymbol{i}}, \vec{\boldsymbol{p}}_{\boldsymbol{ji}}), \boldsymbol{p_{ji}} = \boldsymbol{p_j} - \boldsymbol{p_i}, \boldsymbol{p_{ij}} = \boldsymbol{p_i} - \boldsymbol{p_j}$$

$$\textit{then} \begin{cases} \boldsymbol{p_s} = \boldsymbol{p_i}, \vec{\boldsymbol{n}}_{\boldsymbol{s}} = \vec{\boldsymbol{n}}_{\boldsymbol{i}} \\ \boldsymbol{p_t} = \boldsymbol{p_j}, \vec{\boldsymbol{n}}_{\boldsymbol{t}} = \vec{\boldsymbol{n}}_{\boldsymbol{j}} \end{cases} \tag{2.11}$$

$$\textit{else} \begin{cases} \boldsymbol{p_s} = \boldsymbol{p_j}, \vec{\boldsymbol{n}}_{\boldsymbol{s}} = \vec{\boldsymbol{n}}_{\boldsymbol{j}} \\ \boldsymbol{p_t} = \boldsymbol{p_i}, \vec{\boldsymbol{n}}_{\boldsymbol{t}} = \vec{\boldsymbol{n}}_{\boldsymbol{i}} \end{cases}$$

The selection of the source point $\boldsymbol{p_s}$ is based on having the minimal angle between its associated normal and the line connecting the two point. The Darboux frame is then defined as:

$$\begin{aligned} u &= \boldsymbol{n_s} \\ v &= u \times \frac{(\boldsymbol{p_t} - \boldsymbol{p_s})}{||\boldsymbol{p_t} - \boldsymbol{p_s}||_2} \\ w &= u \times v \end{aligned} \tag{2.12}$$

Using the Darboux frame, we can express the difference between the two normals $\boldsymbol{n_s}$ and $\boldsymbol{n_t}$

by a set of angles as:

$$\begin{aligned} \alpha &= v \cdot \boldsymbol{n_t} \\ \phi &= u \cdot \frac{(\boldsymbol{p_t} - \boldsymbol{p_s})}{d} \\ \theta &= \arctan(w \cdot \boldsymbol{n_t}, u \cdot \boldsymbol{n_t}) \end{aligned} \tag{2.13}$$

where the Euclidean distance d between the two points $\boldsymbol{p_s}$ and $\boldsymbol{p_t}$ is given as $d = ||\boldsymbol{p_t} - \boldsymbol{p_s}||_2$. This quadruplet, $< \alpha, \phi, \theta, d >$, is calculated for each point pairs in the k-neighborhood. In a k-neighborhood P^k, the number of quadruplets formed is $k\frac{k-1}{2}$ with an theoretical computational complexity of $O(k^2)$ (Rusu et al., 2008). This yields that the complexity of estimating PFH on a point cloud dataset P with n points is $O(nk^2)$.

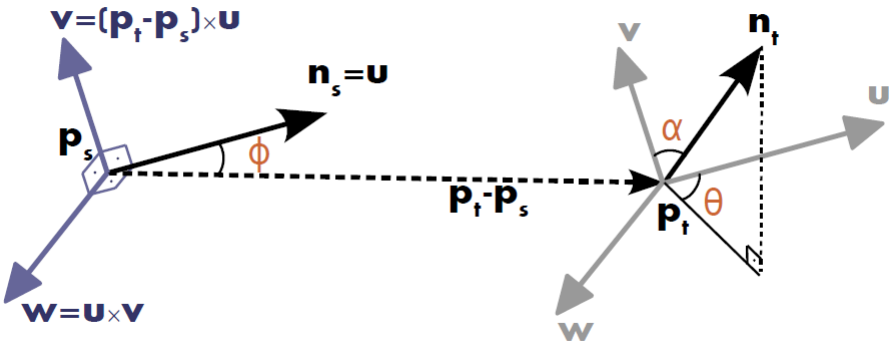


Figure 2.1: A representation of the Darboux frame. (Rusu, 2009)

Figure 2.2 illustrates an influence region diagram of the PFH computation for a query point (marked red) placed in the middle of a circle (sphere in 3D) with radius r. The points $\boldsymbol{p_{k1}}$ to $\boldsymbol{p_{k5}}$ are the k-neighbors and are located within the radius r. These points are fully interconnected in a mesh structure.

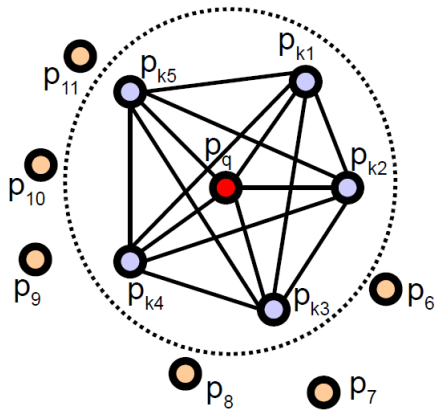


Figure 2.2: Influence region diagram for PFH. (Rusu et al., 2009)

The final PFH representation for the query point $\boldsymbol{p_q}$ is a set of all quadruplets binned into a histogram. Binning is done by dividing each features value range into subdivisions, counting the number of occurrences in each sub-interval. Figure 2.3 shows an example of the PFH feature at corresponding points, along with an comparison of histograms (Rusu et al., 2008).

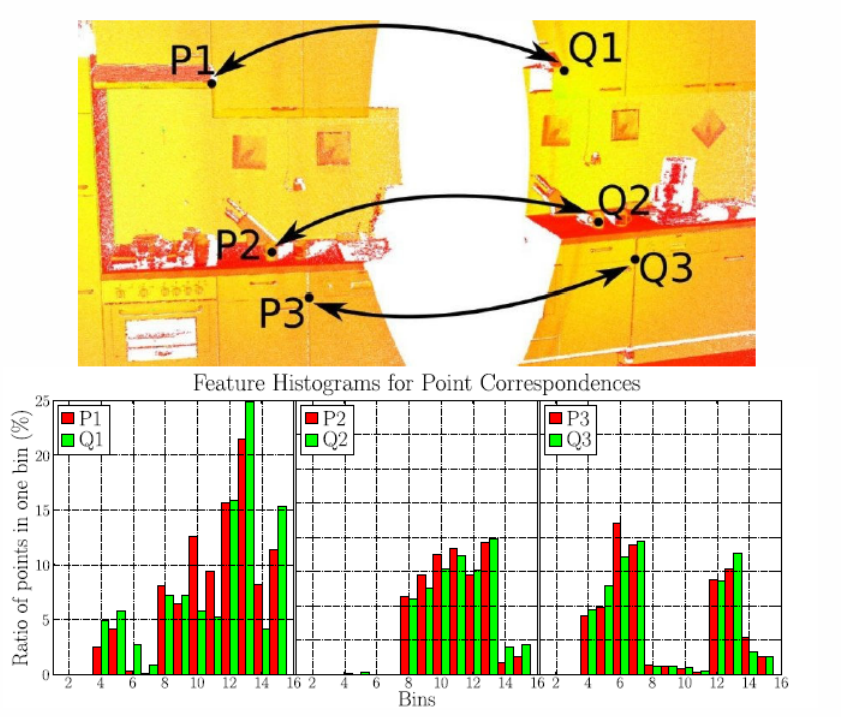


Figure 2.3: Point correspondences and histograms. (Rusu et al., 2008)

2.5.3 Fast Point Feature Histograms

From the previous Section, we have that the the theoretical computational complexity of the PFH for a point cloud P with n points where k is the number of neighbors for each point $p \in P$ is $O(nk^2)$. For applications that require a fast computational time, computing PFH can represent one of the major bottlenecks (Rusu et al., 2009).

Fast Point Feature Histograms (FPFH) is a simplified version of PFH that reduces the computational complexity of the algorithm to $O(nk)$, while preserving the power of PFH. The simplification is done as following:

- For each query point $\boldsymbol{p_q}$, a set of tuples $< \alpha, \phi, \theta >$ is computed between the query point and its neighbors (in the same way as in PFH). This is called the Simplified Point Feature Histogram (SPFH).

- For each point, its k neighbors are re-determined, and the neighboring SPFH values is used to weight the final histogram of the query point (FPFH, see equation 2.14).

$$FPFH(\boldsymbol{p_q}) = SPFH(\boldsymbol{p_q}) + \frac{1}{k}\sum_{i=1}^{k}\frac{1}{\omega_k}\cdot SPFH(\boldsymbol{p_k}) \tag{2.14}$$

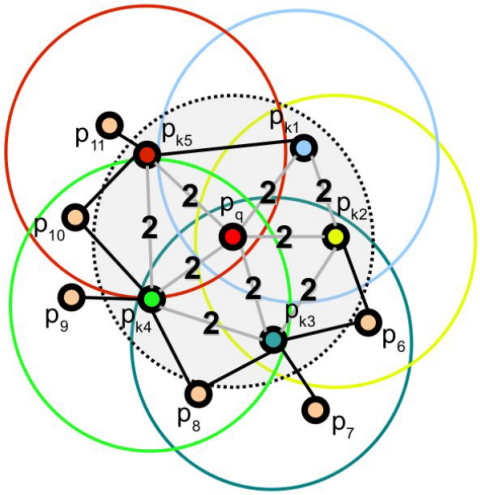


Figure 2.4: Influence region diagram for PFH. (Rusu et al., 2009)

In equation 2.14, ω_k is the distance between the query point $\boldsymbol{p_q}$ and a neighboring point $\boldsymbol{p_k}$ in a given metric space. Figure 2.4 presents the influence region diagram for the FPFH, and illustrates the weighting and how the simplification has affected the diagram compared to Figure 2.2.

Differences between PFH and FPFH (Rusu, 2009):

- FPFH does not create a fully interconnected mesh structure of all neighbors, and thereby miss some value pairs that could contribute to capture the surrounding geometry more accurate.
- PFH models a precisely determined surface around the query point $\boldsymbol{p_q}$ inside the sphere radius r. FPFH includes additional point pairs outside this radius.
- Due to the re-weighting, FPFH combines SPFH values and re-captures some of the point neighboring value pairs.
- The computing complexity is greatly reduced with FPFH, making it viable for use in real-time applications.

2.5.4 Signature of Histograms of Orientations

An evaluation of existing feature descriptors by Tombari et al. (2010) led to the conclusion that one of the major problems of the ones evaluated is the definition of a single, unambiguous and stable local coordinate system at each reference point. Based on this evaluation, the authors suggested a new local coordinate system along with the Signature of Histograms of Orientations (SHOT).

Algorithm 3 gives an overview over the computational steps for the SHOT feature at a reference point p_r. The computation of the local coordinate system at p_r is done through steps 1 to 3. Given n neighbors p_i around a reference point p_r, the weighted covariance matrix C is given as:

$$C = \frac{1}{n}\sum_{i=1}^{n}(r - ||p_i - p_r||) \cdot (p_i - p_r) \cdot (p_i - p_r)^T \tag{2.15}$$

where r is the spherical neighborhood radius.

The local coordinate system at p_r is then defined by decomposition of eigenvalues to create three orthogonal eigenvectors from the covariance matrix. The eigenvectors, v_1, v_2 and v_3, are sorted in an decreasing order by their corresponding eigenvalue, representing the $x-$, $y-$ and $z-$axis. The direction of the X-axis is given by the orientation of the vectors from p_r to the neighboring points p_i, as:

$$X = \begin{cases} v_1, & \text{if} \quad |S_x^-| \leq |S_x^+| \\ -v_1, & \text{otherwise} \end{cases} \tag{2.16}$$

where

$$\begin{aligned} S_x^+ &= \{p_i | (p_i - p_r) \cdot v_1 \geq 0\} \\ S_x^- &= \{p_i | (p_i - p_r) \cdot v_1 < 0\} \end{aligned} \tag{2.17}$$

The direction of the Z-axis is determined in the similar way as the X-axis, and the Y-axis is determined by the cross product between X and Z ($Z \times X$).

The final local coordinate system is used as a basis to divide the spatial environment of p_r with an isotropic spherical grid (Step 4 in Algorithm 3). Further on, on each point p_i in a cell, the angle $\zeta_i = p_i \cdot p_r$ is computed between the normals of the point p_i and p_r. The local distribution of angles is subsequently described by one local histogram for each cell (Hansch et al., 2014). Given a spherical grid that contains k different cells with local histograms and each histogram contains b bins, the final histogram will contain $k \cdot b$ values. Figure 2.5 illustrates the structure for the SHOT feature.

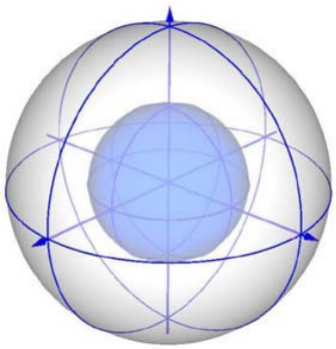


Figure 2.5: Signature structure for SHOT. (Tombari et al., 2010)

There also exist a SHOT-Color version (Tombari et al., 2011), that combines the SHOT feature with color information. In this version, each cell in the spherical grid contains two local histograms, one for the angle between the normals (as explained above) and one histogram which consists of the sum of absolute differences of the RGB values between the points.

Algorithm 3 SHOT feature

1: Create a weighted covariance matrix of neighboring points.
2: Extract the eigenvectors of the weighted covariance matrix.
3: Reorient the eigenvectors to build a local coordinate system.
4: Create a spherical grid.
5: Create histograms for the grid cells.
6: Group the cell histograms to point histograms.

2.5.5 Viewpoint Feature Histogram

The Viewpoint Feature Histogram (VFH) is related to the FPFH feature, and was introduced by Rusu et al. (2010). The main difference between the VFH feature compared to the ones mentioned above, is that VFH is known as a *global* feature. A global feature is estimated for a whole cluster to represent an object, not for individual points. The VFH feature contains two main parts:

- A viewpoint direction component.
- An extended FPFH component.

The viewpoint direction component is computed by finding the objects centroid, which is the point that results from averaging the X-, Y- and Z-coordinates of all the points. The second step is to compute a vector between the centroid point and the viewpoint (position of the 3D camera), and normalize it. The final step is, for all the points in the cluster, to calculate the angle between this vector and their normal. The final result is binned into an histogram. When computing the angle, the vector is translated to each point to make the feature scale invariant.

The extended FPFH component is computed in the same way as explained in Section 2.5.3, with some variations. It is not computed for all the points in the cluster, but only for the centroid. The computed viewpoint direction vector is used as the normal, with all the surrounding points in the cluster as neighbors.

Figures 2.6 and 2.7 illustrates the VFH cluster as well as the extended FPFH representation.

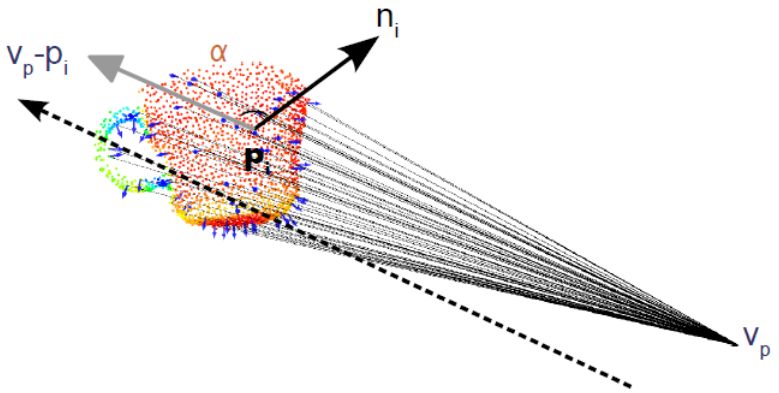


Figure 2.6: The Viewpoint Feature Histogram cluster representation. (Rusu et al., 2010)

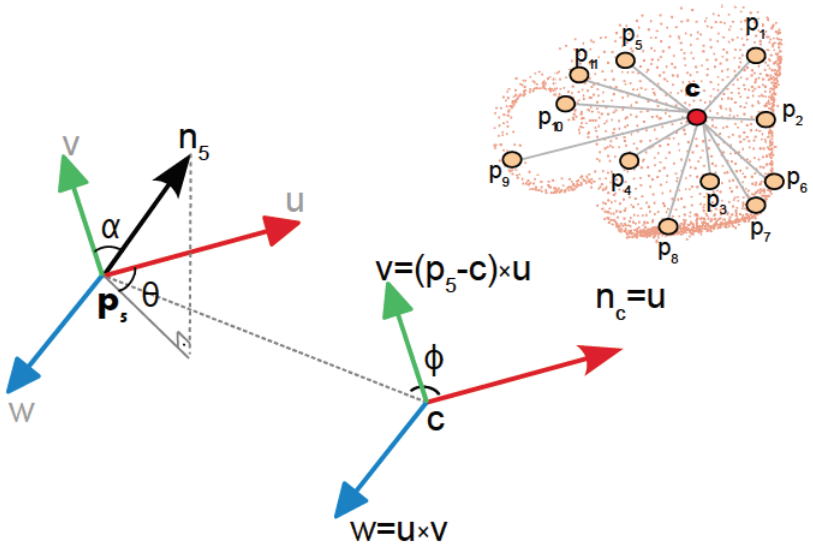


Figure 2.7: The extended Fast Point Feature Histogram representation. (Rusu et al., 2010)

2.6 Alignment

This Section will give an introduction of how to take advantage of processing methods (Section 2.3, keypoints (Section 2.4) and features (Section 2.5) for aligning point clouds.

2.6.1 Registration

The problem of aligning two or more clouds to create a complete model is known as registration. The goal of registration is to find the relative positions and orientations of the separately acquired views in a global coordinate framework, such that the intersecting areas between then overlap perfectly (Rusu et al., 2008). Figure 2.9 gives an rough overview of how to register two

clouds, *Cloud 1* and *Cloud 2*. The first three steps involve the acquiring of point cloud data, estimation of keypoints and feature descriptors (Explained in the previous sections).

The next step is then to find correspondences from the set of feature descriptors together with their XYZ-positions in the two clouds, based on the similarities between the features and positions. The set of correspondences is found by performing a k-nearest neighbor search, from a kD-tree search algorithm. In an ideal world with a 3D-camera that will give perfect data each time, estimation of correspondences will always give point-to-point correspondences. This is unfortunately not the case, and correspondence estimation in noisy data will give false correspondences. False correspondences contribute in a bad way towards transformation estimation and thereby needs to be handled in some way.

Rejection of false correspondences can be done in multiple ways, where 4 of them are illustrated in Figure 2.8. A list of available rejection methods is as follows:

- Rejection based on distance.
- Rejection based on median distance.
- Rejection based on pairs with duplicate target matches.
- Rejection based on normals.
- Rejection based on surface boundaries.
- Rejection based on RANSAC.

Most of these rejection methods are trivial and thereby it is advised to use a combination of at least two methods to yield a good result (Holz et al., 2015).

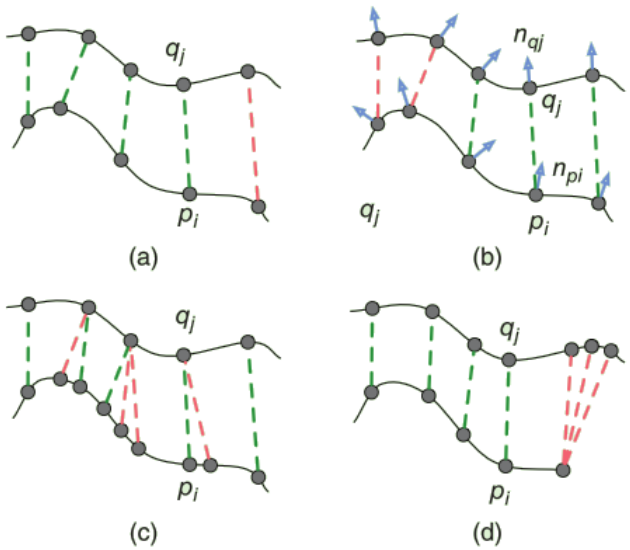


Figure 2.8: Correspondence rejection methods. Rejection based on distance between points (a). Rejection based on normals (b). Rejection based on duplicate matches (c). Rejection based on boundary points (d). (Holz et al., 2015)

The final step in the registration process is to estimate a rigid transformation based on the remaining good set of correspondences between the two clouds. The two common methods to estimate a transformation is:

- Singular Value Decomposition (SVD)
- Levenberg-Marquardt (LM)

The big difference between these estimation methods is that the LM estimation use an iterative approach, whereas the SVD method use a closed-form solution based on the singular value decomposition of a covariance matrix of the data. This means that unlike the iterative approach (LM), SVD provides the best possible transformation estimation in a single step, and is thereby preferred.

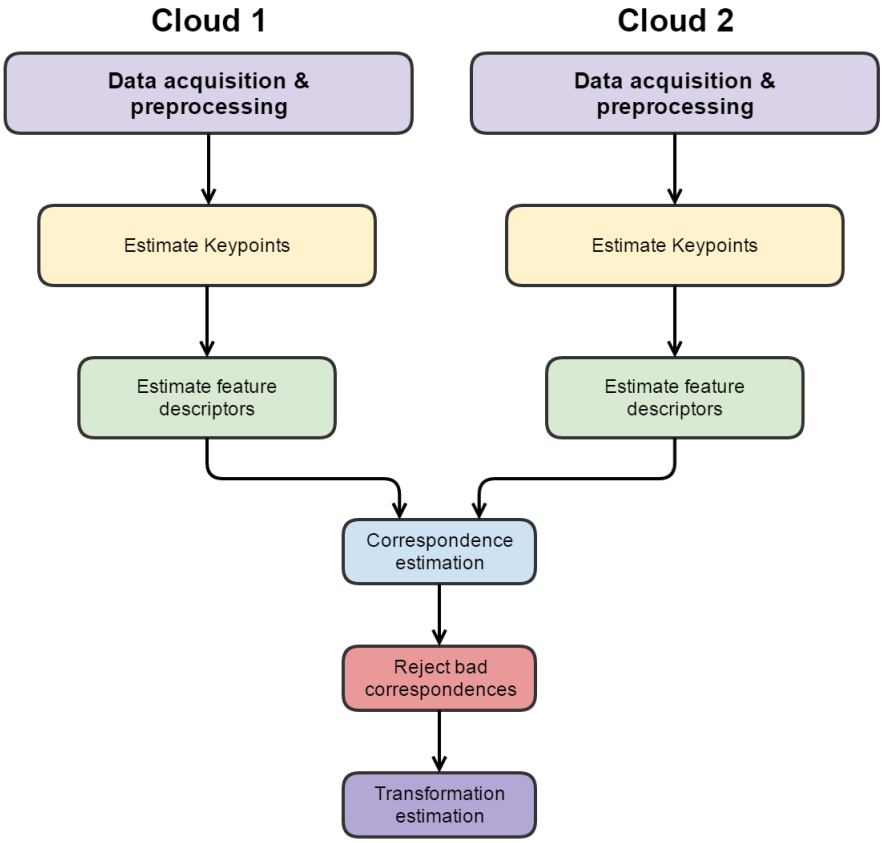


Figure 2.9: The registration process.

2.6.2 Iterative Closest Point

In most cases, the registration process explained in Section 2.6.1 require some form of refined alignment after the transformation estimation step. We often denote the transformation estimation as the initial alignment, followed by a method for final alignment. The Iterative Closest Point (ICP) method is an iterative approach for aligning data and was introduced by Chen and Medioni (1992). The ICP algorithm has two data inputs (point clouds), *target* and *source*,

which has either been initially transformed (from Section 2.6.1) or has an initial guess for the transformation between them. The aim for ICP is to find the best transformation between two datasets by minimizing the distance between corresponding points. This is done by iteratively estimating the optimal transformation by generating pairs of corresponding points and minimizing an error metric. Since the process is iterative, this procedure is continuously repeated until its error metric reaches a local user chosen minimum (Rusinkiewicz and Levoy, 2001).

The ICP algorithm have many different variants, but the main approach for every variant can be summarized in six steps (Rusinkiewicz and Levoy, 2001):

1. **Selection** of some set of points in one or both datasets.
2. **Matching** these points to samples in the other dataset.
3. **Weighting** the corresponding pairs appropriately.
4. **Rejecting** certain pairs based on looking at each pair individually or considering the entire set of pairs.
5. Assigning an **error metric** based on the point pairs.
6. **Minimizing** the error metric.

Given the two datasets, *target* (D) and *source* (M), the ICP algorithm aims to find the transformation consisting of a rotation $\mathbf{R}$ and a translation $\boldsymbol{t}$, which minimizes a cost function. The cost function is given as (Nuchter et al., 2006):

$$E(\mathbf{R}, \boldsymbol{t}) = \sum_{i=1}^{|M|} \sum_{j=1}^{|D|} \omega_{i,j} ||\boldsymbol{m_i} - (\boldsymbol{R d_j} + \boldsymbol{t})||^2 \tag{2.18}$$

where $\omega_{i,j}$ is assigned 1 if the i-th point of M describes the same point in space as the j-th point of D, otherwise $\omega_{i,j}$ is assigned 0.

In every iteration, the optimal transformation ($\mathbf{R}$,$\boldsymbol{t}$ has to be computed. Equation 2.18 can be reduced to:

$$E(\mathbf{R}, \boldsymbol{t}) \propto \frac{1}{N} \sum_{i=1}^{N} ||\boldsymbol{m_i} - (\boldsymbol{R d_i} + \boldsymbol{t})||^2 \tag{2.19}$$

where $N = \sum_{i=1}^{|M|} \sum_{j=1}^{|D|} \omega_{i,j}$, since the correspondence matrix can be represented by a vector containing the point pairs (Nuchter et al., 2006).

One of the most common variant of the ICP algorithm is the one based on Singular Value Decomposition, based on the work of Arun et al. (1987). This variant minimizes equation 2.19, and the difficulty of the minimization is to enforce the orthonormality of the matrix $\mathbf{R}$. The new error function, E($\mathbf{R}$,$\boldsymbol{t}$), is written as:

$$E(\mathbf{R}, \boldsymbol{t}) \propto \sum_{i=1}^{N} ||\boldsymbol{m_i'} - \boldsymbol{R d_i'}||^2 \tag{2.20}$$

where $\boldsymbol{t} = \boldsymbol{c_m} - \boldsymbol{R}\boldsymbol{c_t}$ and

$$\boldsymbol{c_m} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{m_i}, \quad \boldsymbol{c_t} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{d_j} \tag{2.21}$$

The idea in this version is to decouple the calculation of the rotation $\boldsymbol{R}$ and the translation t, using the centroids of the points belonging to the matching (equation 2.21).

2.7 Object detection

The previous sections has given an introduction into how simple point cloud processing can be achieved, how features (descriptors) can be estimated on keypoints and how the registration process can assemble a complete point cloud from multiple viewpoints. This Section will explain the typical steps required to detect an object in a point cloud. These steps are often referred to as the object detection *pipeline*.

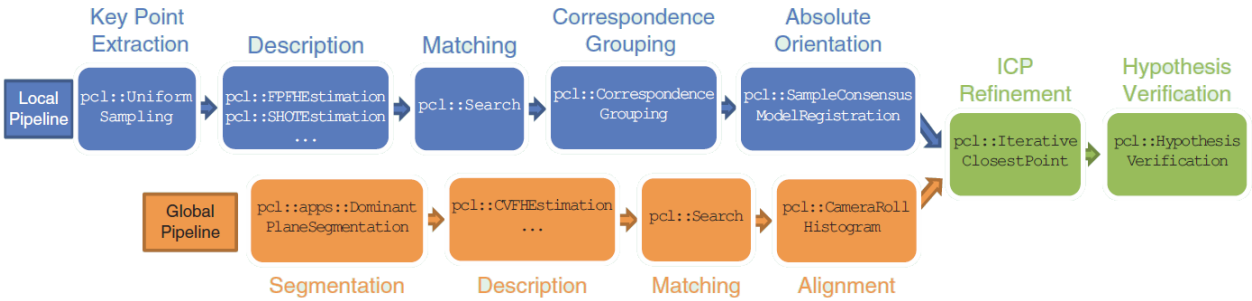


Figure 2.10: Proposal of a local and global recognition pipeline based on Point Cloud Library (Aldoma et al., 2012).

There is no unique solution to a pipeline that will be the best solution in all scenarios, because the pipeline will depend on the problem to be solved as well as the tools that are used. Aldoma et al. (2012) suggested the pipeline in Figure 2.10, which is divided in two different pipelines, *local* and *global*. The main difference between these pipelines is the descriptor used for matching, where the local pipeline use local descriptors and the global pipeline use global descriptors.

Most object recognition systems use one 3D camera to detect an object. This means that the object to be detected is observed only from one viewpoint. The first step is then to *train* the system, which in this case means creating a database of all the objects that are to be detected. Each object in the database also needs to be captured from different viewpoints. This can be done either by taking snapshots of the physical model from different viewpoints, from using a rotating table, or by performing ray-tracing of a CAD-model. Ray-tracing is basically the same as moving the camera around the object and taking snapshots, but it is done with a virtual camera placed around the CAD-model.

The big difference between local and global descriptors are:

- **Local** - Computed for individual points (keypoints).
- **Global** - Not computed for individual points, but for a whole cluster that represents an object.

Looking at the global pipeline in Figure 2.10, the first step is segmentation. This is because every object present in the captured scene must be split up in clusters. For each cluster, a global descriptor has to be computed, to represent the entire object. The idea is then to match each cluster towards every trained model in the system, and give that cluster a score. The cluster with the best matching score will then most likely be the object that is present in the scene.

In the local pipeline the process is almost identical, but it is instead based on local feature descriptors. The last steps in both pipelines are somewhat similar, containing initial and final alignment of the object model to the scene.

Chapter 3: Robotics

This chapter will give an introduction to basic kinematics. This includes how to achieve transformation (rotation and translation), and different representation of orientation. The chapter also explains the basics behind Robot Operating System (ROS).

3.1 Kinematics

A manipulator (robot) can be represented as a chain of rigid bodies (*links*) connected by means of revolute or prismatic *joints*. The start point of the chain is constrained to a *base*, while the *end-effector* is mounted on the end. The motion of the structure is obtained by composition of the elementary motions of each link with respect to the previous one. In order to manipulate an object in space, it is necessary to have a way of describing the *end-effector* position and orientation (Siciliano et al., 2009).

3.1.1 Rotation matrix

A rotation matrix is a way of describing rotation (orientation) about an arbitrary axis in space with respect to a reference frame. The 3x3 rotation matrices about axis X, Y and Z is given below (Siciliano et al., 2009).

$$R_x(\gamma) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos\gamma & -sin\gamma \\ 0 & sin\gamma & cos\gamma \end{pmatrix} \tag{3.1}$$

$$R_y(\beta) = \begin{pmatrix} cos\beta & 0 & sin\beta \\ 0 & 1 & 0 \\ -sin\beta & 0 & cos\beta \end{pmatrix} \tag{3.2}$$

$$R_z(\gamma) = \begin{pmatrix} cos\alpha & -sin\alpha & 0 \\ sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.3}$$

Consider a point in vector coordinates

$$p = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \tag{3.4}$$

which is located in a Cartesian coordinate system with respect to a reference frame. This point can be rotated about an arbitrary axis by multiplying a rotation matrix with the point. Consider the point p to be rotated by 45 degrees by its Z-axis (Siciliano et al., 2009).

$$p_r = \begin{pmatrix} cos45 & -sin45 & 0 \\ sin45 & cos45 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \tag{3.5}$$

3.1.2 Euler angles

Euler angles is a way of representing orientation of a rigid body. This is done by using a set of *three* angles (Siciliano et al., 2009).

$$\phi = \begin{bmatrix} \varphi & \vartheta & \psi \end{bmatrix}^T \tag{3.6}$$

The two most common sets of Euler angles are ZYZ angles and ZYX (roll, pitch, yaw) angles. The RPY angle set originates from the nautical field. The resulting rotation from a RPY set is obtained by the following sequence of rotation

- Rotate the reference frame by the angle ψ about x-axis (yaw). Rotation described by equation 3.1.
- Rotate the reference frame by the angle ϑ about y-axis (pitch). Rotation described by equation 3.2.
- Rotate the reference frame by the angle φ about z-axis (roll). Rotation described by equation 3.3.

which can be written as.

$$R(\phi) = R_z(\varphi)R_y(\vartheta)R_x(\psi) \tag{3.7}$$

$R(\phi)$ is a 3x3 matrix, which can be written as

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \tag{3.8}$$

The resulting inverse solution from $R(\phi)$ is given in two different sets of range for ϑ.

ϑ in the range $(-\pi/2, \pi/2)$:

$$\begin{aligned} \varphi &= Atan2(r_{21}, r_{11}) \\ \vartheta &= Atan2(-r_{31}, \sqrt{r_{32}^2 + r_{33}^2} \\ \psi &= Atan2(r_{32}, r_{33}) \end{aligned} \tag{3.9}$$

ϑ in the range $(\pi/2, 3\pi/2)$:

$$\begin{aligned} \varphi &= Atan2(-r_{21}, -r_{11}) \\ \vartheta &= Atan2(-r_{31}, -\sqrt{r_{32}^2 + r_{33}^2}) \\ \psi &= Atan2(-r_{32}, -r_{33}) \end{aligned} \tag{3.10}$$

3.1.3 Quaternions

Another way to represent orientation is the use of *quaternions*. The unit quaternion corresponding to a rotation matrix $\boldsymbol{R}(\theta)$ can be represented as

$$\boldsymbol{q} = \begin{bmatrix} \eta \\ \boldsymbol{\epsilon} \end{bmatrix} \in \mathbb{R}^4 \tag{3.11}$$

where

$$\eta = \cos(\frac{\theta}{2}) \quad \text{and} \quad \boldsymbol{\epsilon} = \boldsymbol{k}\sin(\frac{\theta}{2}) \tag{3.12}$$

η is called the scalar part of the quaternion, while $\boldsymbol{\epsilon} = [\epsilon_x \quad \epsilon_y \quad \epsilon_z]^T$ is called the vector part of the quaternion. They are constrained by the condition $\eta^2 + \epsilon_x^2 + \epsilon_y^2 + \epsilon_z = 1$, hence the name *unit* quaternion (Siciliano et al., 2009).

The rotation matrix corresponding to a given quaternion is as follows:

$$\boldsymbol{R}(\eta, \boldsymbol{\epsilon}) = \begin{bmatrix} 2(\eta^2 + \epsilon_x^2) - 1 & 2(\epsilon_x\epsilon_y - \eta\epsilon_z) & 2(\epsilon_x\epsilon_z + \eta\epsilon_y) \\ 2(\epsilon_x\epsilon_y + \eta\epsilon_z) & 2(\eta^2 + \epsilon_y^2) - 1 & 2(\epsilon_y\epsilon_z - \eta\epsilon_x) \\ 2(\epsilon_x\epsilon_z - \eta\epsilon_y) & 2(\epsilon_y\epsilon_z + \eta\epsilon_x) & 2(\eta^2 + \epsilon_z^2) - 1 \end{bmatrix} \tag{3.13}$$

The solution to the inverse problem to compute the quaternion corresponding to a given rotation matrix (same as in equation 3.8 above) is:

$$\eta = \frac{1}{2}\sqrt{r_{11} + r_{22} + r_{33} + 1} \tag{3.14}$$

$$\boldsymbol{\epsilon} = \begin{bmatrix} sgn(r_{32} - r_{23})\sqrt{r_{11} - r_{22} - r_{33} + 1} \\ sgn(r_{13} - r_{31})\sqrt{r_{22} - r_{33} - r_{11} + 1} \\ sgn(r_{21} - r_{12})\sqrt{r_{33} - r_{11} - r_{22} + 1} \end{bmatrix} \tag{3.15}$$

where $sgn(x) = 1$ for $x \geq 0$ and $sgn(x) = -1$ for $x < 0$. In equation 3.14 it is assumed that $\eta \geq 0$, which corresponds to an angle $v \in [-\pi \quad \pi]$, thus any rotation can be described (Siciliano et al., 2009).

3.1.4 Homogeneous transformation matrix

As explained above, it is necessary to describe both position and orientation for the *end-effector* with respect to a reference frame. The position (translation) and orientation can be described by the use of the 4x4 *homogeneous transformation matrix*. Consider two *joints*, *joint* 0 and *joint* 1. The translation and rotation between the two joints is given as

$$A_1^0 = \begin{bmatrix} R_1^0 & p_1^0 \\ 0^T & 1 \end{bmatrix} \tag{3.16}$$

where R_1^0 is the rotation from *joint* 0 to *joint* 1 and p_1^0 is the translation (Siciliano et al., 2009).

3.2 Robot Operating System

Robot Operating System (ROS) is an open-source, meta-operating system for robots. It provides the common services of an operating system, including:

- Hardware abstraction
- Low-level device control
- Implementation of commonly-used functionality
- Message-passing between processes
- Package management

ROS also provides the tools and libraries necessary for obtaining, building, writing and running code across multiple computers. Currently, ROS only runs on Unix-based platforms, and is primarily tested on Ubuntu and MAC OS X systems.

The ROS runtime *graph* is a peer-to-peer network of processes that are loosely coupled using the communication infrastructure in ROS. The processes can either be running on a single machine, or distributed across multiple machines (ROS). In every ROS system, one machine has to run the ROS Master. The Master provides name registration and lookup to the rest of the graph, and without the master, nodes are not able to find each other, exchange messages or invoke services.

3.2.1 Nodes

A process in ROS is called a *node*, and serves a purpose of performing a task or computation. Nodes are combined in the ROS graph and communicate with one another using *topics* and *services*. A typical robot control system will consist of many different nodes. For example, one node controls camera data acquisition, one node processes the camera data, one node controls the robot motors, etc. This way of programming simple nodes that perform unique tasks give the advantage of re-using nodes in multiple projects. Code complexity is also reduced in comparison to a monolithic system.

Each node have a unique graph *resource name* to identify them to the rest of the system. For example, */camera_pointcloud* could be the name of the node that acquire a point cloud from

the camera. A ROS node can either be programmed in C++ or python. Running a node can be done in the command line, for example as:

rosrun example_package example_node

where *rosrun* is the command for running a node, *example_package* is the package where the node is located, and *example_node* is the name of the node to be run.

It is also possible to start one or multiple nodes using the *roslaunch* tool. Roslaunch is specified by a XML file that describe the nodes that should be run, parameters that should be set, and other attributes. Starting a launch file can be done as follows:

roslaunch example_package example_xml.launch

where *example_xml.launch* is the launch file. The rest of the command is the same as in the rosrun explanation.

3.2.2 Messages

Nodes communicate with each other by publishing *messages* to *topics*. A message is a simple data structure, containing typed fields. Messages support the primitive data types (integer, float, boolean) etc., as well as arrays of these types (ROSmessage).

3.2.3 Topics

Topics in ROS are named buses that nodes use to exchange messages. They have anonymous publish/subscribe semantics, meaning that topics are decoupled from the nodes with regards of information. Nodes are thereby not aware of who they are communicating with. Instead, nodes that are interested in data *subsribe* to the relevant topic, and nodes that generate data *publish* to the relevant topic. A topic can have multiple subscribers and publishers (ROStopic).

3.2.4 Services

The communication in nodes with the publish/subscribe model is a flexible communication method, but its many-to-many one-way transport is not appropriate for request/reply interactions. This is done in ROS by using *services*. A service is defined by a pair of messages, one for the request and one for the reply. (ROSservice)

3.2.5 RViz

RViz is a powerful 3D visualization tool for ROS. In RViz, you can visualize robots, point clouds and many other ROS related utilities. It is possible to tweak the standard version of RViz to fit the users robot cell setup (ROSrviz).

Chapter 4: Method

4.1 Programming

Some of the questions when starting this thesis was:

1. What programming language to use?
2. What kind of libraries are available?
3. Which operating system to use?
4. What kind of programming environment is best suited for this thesis?

When dealing with data from one or more 3D-cameras, most of the available libraries are based on programming with C++. The most common and most developed library is the Point Cloud Library (PCL). PCL is a very large open project, supported by engineers and scientists from many different organizations all around the world. It is release under the terms of the 3-clause BSD license and is open source software, meaning it is free for commercial and research use.

Regarding the operating system, this thesis focuses on two main parts, computer vision in 3D and robotics. The small robot cell at NTNU IPK contains two KR 6 R900 sixx (KR Agilus) robots. This robot cell is set up to be compatible with ROS. Thus, the natural choice became combining PCL with ROS, by programming in C++ and using Ubuntu as the operating system.

There are many different programming environments that handles C++ development, but since the work in this thesis would end up in a lot of testing and developing, creating a graphical user interface (GUI) would make this easier. The most common C++ GUI development environment is Qt Creator, which is a combination of a standard programming environment together with a drag-and-drop GUI builder.

The one thing that really distinguishes developing in Qt compared to other environments is the *signals* and *slots*. Signals and slots are used for communication between objects, which can be somewhat compared with *listeners* in other frameworks. GUI components in Qt are called *widgets*. Widgets have two main components, a C++ class and a graphical design. The C++ class can contain both signals and slots (both in widgets and in a normal class). Signals and slots are exactly what they are named, meaning that a signal can be connected to a slot. In fact, a signal can be connected to multiple slots in different classes. The big advantage with signals and slots compared to listeners is dealing with GUI programming. Figure 4.1 illustrates an example of signals connected to slots in a clock. The blue connection shows that the *updated(QTime)* signal from the clock is connected to the *setTime(QTime)* slot on the current time box. The red connection shows the *valueChanged(int)* signal from the set time zone box connected to the *setTimeZone(int)* in the clock. Using this type of technique requires zero lines of programming, as long as the widgets are implemented.

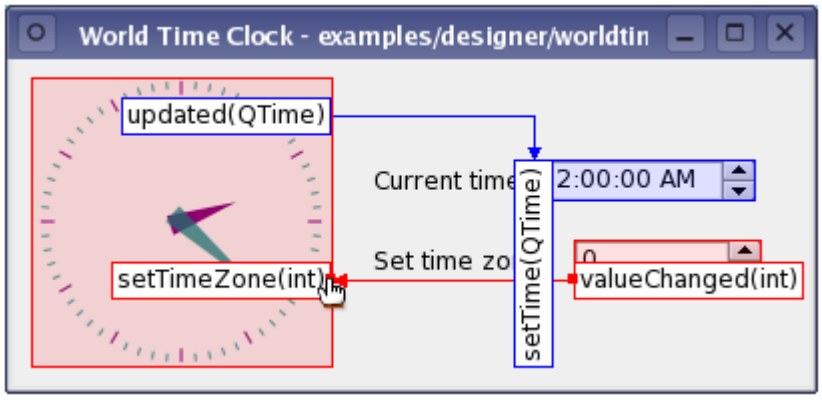


Figure 4.1: Example of signal and slot between widgets (QtSignalSlot).

The idea was then to combine the powerful tools in PCL with ROS, and develop a GUI in Qt allowing real-time testing of point cloud processing tools.

4.2 Practical setup

The small robot cell at NTNU IPK contains the following:

- 2x KUKA KR 6 R900 sixx (KR Agilus) robots
- 2x KUKA KR C4 compact robot controller
- 2x Stationary computers running Ubuntu
- 1x Siemens PLC

In between the two robots, there is a table, as can be seen in Figure 4.2. On top of the table is where the object to be detected will be placed.

Both of the computers are running ROS on Ubuntu 14.04. One of them is running the ROS Master and is connected to the robot controllers. This computer runs the node that handles the communication between the robot controller and the ROS Master, and requires a high priority from the underlying kernel in Ubuntu. Thereby, this computer runs a real-time kernel where you can set the priority of the nodes to real-time. Programs running on real-time priority will always have the highest priority in the scheduler.

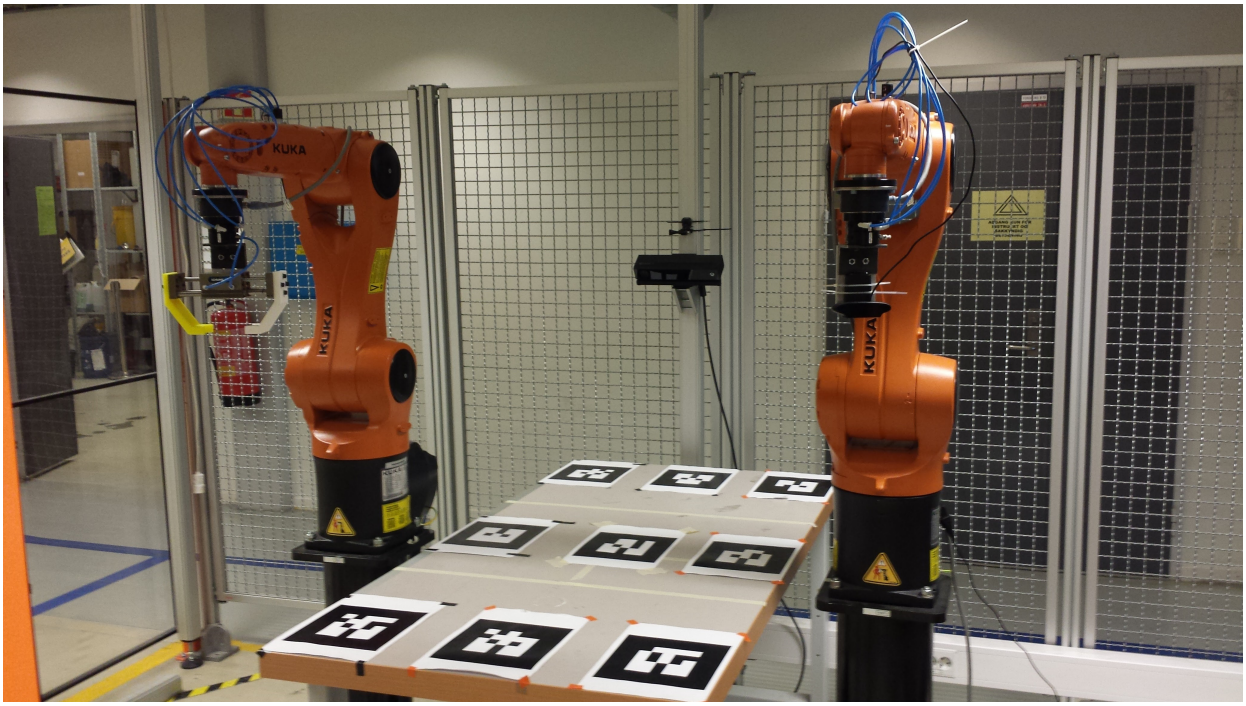


Figure 4.2: The small robot cell at NTNU IPK.

To be able to detect an object from all directions and reconstruct a point cloud of the entire model, at least three 3D-cameras were required. The initial setup is illustrated in Figure 4.3, which illustrates roughly where the cameras could be placed in the robot cell with regards to the robots and the table.

Inside the robot cell there is mounted a rail bar system, in which the cameras could be mounted. To get the cameras to point downwards with a desired angle, a sketch for a bracket to fit the Kinects was made in SolidWorks. This sketch was further on 3D-printed and the cameras could be mounted. Figure 4.4 shows how the model of the bracket. Notice the incline on the mounting surface and the bottom of the bracket is drawn to fit a rail system.

The final placement of the cameras mounted on the custom brackets can be seen in Figure 4.5. The locations correspond to what is shown in Figure 4.3.

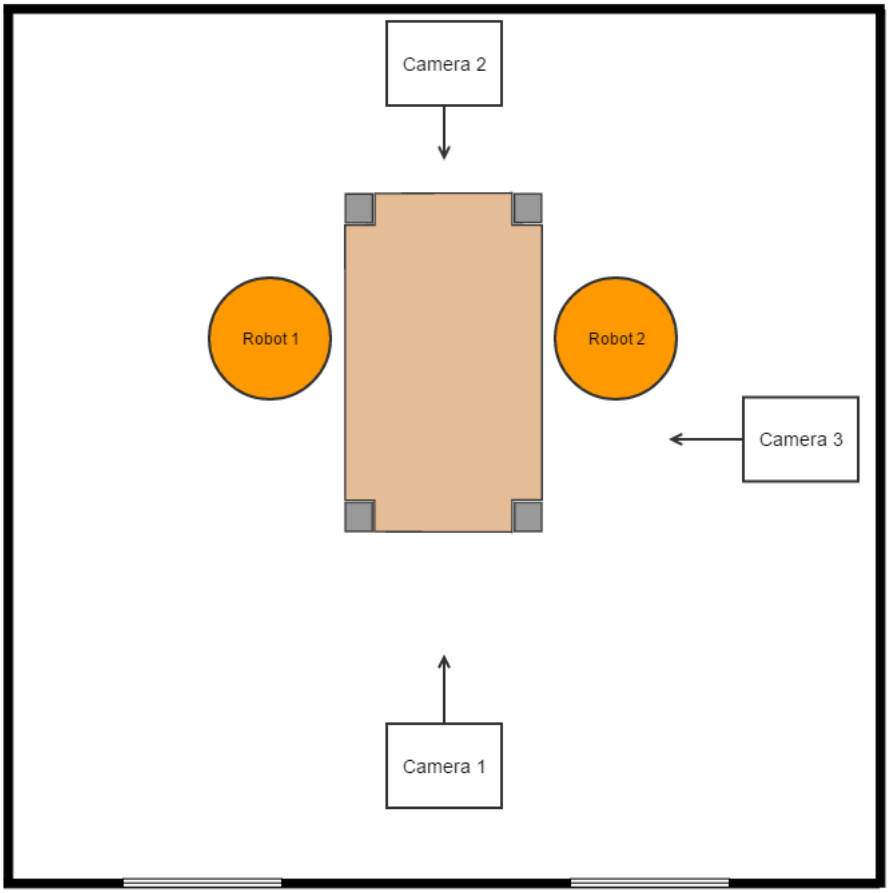


Figure 4.3: Illustration of the placement of the cameras in the robot cell.

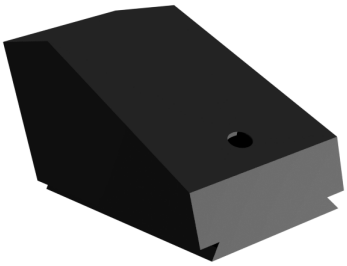


Figure 4.4: Drawing of the bracket to mount the cameras on.

(a) Camera number one.

(b) Camera number two.

(c) Camera number three.

Figure 4.5: Overview of where the cameras are placed.

4.3 Point cloud acquisition

Since the Kinect™is created by Microsoft, who also produce the operating system Windows, the Kinect™software development kit is only available on Windows. Libfreenect2 is a driver for the Kinect™v2, created by Xiang et al. (2016), that runs on Windows, MACOS X and Linux (Ubuntu). The installation steps for this driver is found at https://github.com/OpenKinect/libfreenect2. To make libfreenect2 compatible with ROS, Thiemo Wiedemeyer from the University of Bremen has created a bridge between the libfreenect2 and ROS, called Kinect2 Bridge. Installation notes for this is found at https://github.com/code-iai/iai_kinect2.

Kinect2 Bridge creates a ROS node that publishes the output point cloud to a topic, so that other nodes can subscribe to the camera output. Running the node is done by using the *roslaunch* command in ROS, where you can set a number of parameters.. To be able to distinguish the output from all three cameras, the node name has to be unique. This is done by:

roslaunch kinect2_bridge kinect2_bridge.launch base_name:="camera1"

where *base_name* is the unique name for the node. This will give the output topic the name */camera1/sd/points*. One notable aspect with the Kinect2 Bridge is that it has the option to use an built in bilateral filter as well as a edge awareness filter. This means that the output point cloud will have less noise and sharper edges.

4.3.1 Subscribing to a topic

Subscribing to a topic is done using the *ros::NodeHandle* class. As the class name indicates, the *ros::NodeHandle* handles nodes with regards to communication. Subscribing to a topic requires a *callback* method. This method runs each time the node handler is aware that new data has been published on the topic. An example of how to subscribe to the point cloud topic is as following:

```
ros::NodeHandle n;
n.subscribe("/camera1/sd/points", 1, &QNode::callbackMethod, this);
```

where *"/camera1/sd/points"* is the name of the topic to be subscribed to, 1 is the incoming message queue size, *QNode* is the class name of the node and *callbackMethod* is the name of the callback method.

4.4 Main program

The objectives in this thesis requires an implementation of different algorithms and filtering methods. A main program (ROS node) was created with the following properties:

- A graphical user interface.
- Being able to visualize a point cloud.
- Being able to show a resulted point cloud after filtering/manipulation.
- Saving and loading point clouds to the visualizer.

- Easy testing of filtering parameters.
- Easy testing of algorithms and objectives of this thesis.
- ROS communication, controlling robots.
- Logging

To distinguish the ROS part, the GUI part and the filtering/manipulation part, three different classes were implemented, one for each part. This was done by following the typical object oriented programming rules, where each class has its own responsibility. The ROS class and the GUI class is also running as separate threads, so that ROS communication can run in the background, not being disturbed by other computational tasks. The coupling between the classes were also created in such a way that they are easy expandable, especially the filtering class.

4.5 Camera calibration

One of the objectives in this thesis was to create a system for calibration of the extrinsic parameters of a multiple camera setup. This calibration system should be flexible, meaning that a camera could be placed at any location inside the robot cell. This is one of the requirements to be able to reconstruct a full model from different viewpoints in a point cloud. The extrinsic parameters of a camera is a homogeneous transformation matrix, which usually is the transformation from the camera frame to the world coordinates. In the camera setup show in Figure 4.3, *Camera 1* is chosen as the *master* camera. The master camera is used as the main reference to the other cameras. This means that the calibration system needs to find the transformation matrices from *Camera 2* and *Camera 3* to *Camera 1*.

The extrinsic parameters can be found in different ways. One way is measuring the distance and rotation between the cameras manually, but this is a very primitive method and will probably give wrong transformations. Another way is translating and rotating the output point clouds manually to fit each other, taking advantage of a visualizer to see when the clouds fit each other. This is also a primitive approach and it is not easy to achieve a good result. These two methods can instead be used to estimate a rough transformation between the cameras as the first step in an more advanced approach.

For this objective, the goal is to create a ROS node that performs calculation for the extrinsic parameters for one or many cameras. Two main approaches will be tested:

1. Calibrate using the theory behind *registration* from Section 2.6.1.
2. Calibrate using *Alvar*, an open source AR tag tracking library.

4.5.1 Approach 1

From Figure 4.3, we see that the viewpoint of camera 2 is rotated 180 degrees and camera 3 is rotated 90 degrees with respect to camera 1. This means that the surroundings in the points clouds from all the cameras differ. This led to the idea that to be able to calibrate the cameras using registration, the point clouds would have to be more comparable than the raw output, by segmenting out the tabletop and use that part for the registration process. If the tabletop is perfectly aligned, then the remaining parts of the clouds will also align.

The steps to prepare the point clouds for the registration process is given in Algorithm 4. From the main program (see Section 4.4) it is possible to subscribe to a point cloud topic from the Kinect2 Bridge and save it to a file. The first step of Algorithm 4 is to capture three point clouds from each of the cameras. The next three steps are the filtering and segmentation steps to create point clouds of only the tabletop from the raw data clouds. The last steps is estimating the features on the tabletops to prepare the point clouds for further registration.

Algorithm 4 Preparation for calibration with registration

1: Read three saved point clouds from the different cameras.
2: Filter all point clouds with a PassThrough filter for a rough filtering.
3: Down-sample all point clouds with a Voxel Grid filter.
4: Segment the tabletops from the filtered cloud.
5: Estimate normals for the tabletops.
6: Estimate keypoints for the tabletops.
7: Estimate descriptors (features) for the tabletops.

Algorithm 5 gives an overview over the next steps after the preparations for registration has been done. This algorithm provides the extrinsic parameters from a camera n to camera 1. The first step in the algorithm is to compute the correspondences between the point cloud from camera 1 (master camera) and camera n (camera 2 or camera 3 in this case). This set of correspondences will have false correspondences and therefore one or more rejection methods are used. After rejection, the set of good correspondences are used to estimate a transformation from camera n to camera 1. This estimated transformation is used to transform the point cloud so that both point clouds are initially aligned. The final step is to refine the alignment process using Iterative Closest Point (ICP).

Algorithm 5 Calibration through registration

1: Compute correspondences between the point cloud from camera 1 and camera n.
2: Reject bad correspondences.
3: Estimate a transformation based on the good correspondences.
4: Transform the point cloud based on the estimated transformation.
5: Perform a refined alignment using ICP.
6: Save the transformation matrix (extrinsic parameters).

4.5.2 Approach 2

The second approach to calibrate the extrinsic parameters of the cameras is based on taking use of a ROS package called *ar_track_alvar*, which is a ROS wrapper for Alvar, an open source Augmented reality (AR) tag tracking library. An AR tag can be related to qr-codes. The AR tag tracking library has 4 main functions:

1. Generate AR tags.
2. Identify and track the pose of individual AR tags.
3. Identify and track the pose of a bundle of AR tags.

4. Using camera image to automatically calculate spatial relationships between tags in a bundle.

The idea in this approach is to attach AR tags to the tabletop and track them from each camera. Tracking can either be done using an individual tag, or a bundle of tags. A bundle of tags can allow for more stable pose estimates, is robust to occlusions and can track multi-sided objects.

Generating an AR tag is done by running a node called *createMarker*. In this node you can specify size, resolution and give the tag a unique ID (to be able to distinguish them). Figure 4.6 shows 3 different AR tags generated by the *createMarker* node, with ID 0, 1 and 2.

Figure 4.6: Examples of different AR tags.

This library also works with a 2D camera, but the depth information from a 3D camera is integrated for better pose estimates.

Individual tracking

Tracking an individual AR tag only requires one AR tag to be attached to the table. This approach is susceptible of occlusion errors and it can be difficult for a camera to detect the tag at large distances. Tracking the tag is done by running a user specified ROS launch file, for example:

roslaunch ar_track_alvar camera1.launch

where *camera1.launch* is the ROS launch file. This file needs to be specified to subscribe to the correct camera topic and what size the AR tag to be tracked is.

Bundle tracking

Tracking a bundle of AR tags requires two or more tags to be attached to the table. Tracking the bundle of tags is done in the same way as for individual tracking, with one exception. Together with the launch file, the tag bundle has to be specified by an XML file that lists a set of tag IDs and their positions relative to the *master* tag. The first tag in the XML file is chosen as the master tag. The master tag defines the coordinate system for the rest of the tags. This means that if a camera is able to track any of the tags in the bundle, the tracking system knows where the master tag is located. Figure 4.7 shows an example of a XML bundle file.

```
<multimarker markers="4">
    <marker index="8" status="1">
        <corner x="-2.2" y="-2.2" z="0" />
        <corner x="2.2" y="-2.2" z="0" />
        <corner x="2.2" y="2.2" z="0" />
        <corner x="-2.2" y="2.2" z="0" />
    </marker>
    <marker index="9" status="1">
        <corner x="6.8" y="-2.2" z="0" />
        <corner x="11.2" y="-2.2" z="0" />
        <corner x="11.2" y="2.2" z="0" />
        <corner x="6.8" y="2.2" z="0" />
    </marker>
    <marker index="10" status="1">
        <corner x="-2.2" y="-11.2" z="0" />
        <corner x="2.2" y="-11.2" z="0" />
        <corner x="2.2" y="-6.8" z="0" />
        <corner x="-2.2" y="-6.8" z="0" />
    </marker>

    <marker index="16" status="1">
        <corner x="-2.2" y="6.8" z="0" />
        <corner x="2.2" y="6.8" z="0" />
        <corner x="2.2" y="11.2" z="0" />
        <corner x="-2.2" y="11.2" z="0" />
    </marker>


</multimarker>
```

Figure 4.7: XML file that specifies a bundle of tags with ID 8, 9, 10 and 16, where 8 is the master tag.

Extrinsic parameters

The *ar_track_alvar* topic (both individual and bundle) outputs a topic named *ar_pose_marker*. This topic contains a message with information about which camera the output comes from, as well as a pose defined by the X-, Y- and Z-coordinates and the orientation given in quaternion of the tag. The full content of the message can be seen in Figure 4.8.

To decode this message, a ROS node was implemented that subscribes to this topic, extracts the data from the published message and creates a homogeneous transformation matrix. Since the message has the pose orientation given in quaternions, the quaternions were converted into roll, pitch and yaw values to create a rotation matrix. This has to be done for each camera, which will result in 3 different transformation matrices, from camera n to the tag. Knowing this transformation for all cameras, we can also find the transformations between the cameras.

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 id
uint32 confidence
geometry_msgs/PoseStamped pose
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
```

Figure 4.8: The *ar_track_alvar* message declaration. The bottom of the message contains the pose information, position and orientation.

4.6 Registration

As explained in Section 4.5.1, *registration* can be used as a tool for extrinsic camera calibration. Registration in general, can also be a very powerful tool in other types of applications, like indoor and outdoor mapping of large areas. For example, reconstructing a point cloud of an entire factor by capturing point clouds from a large amount of viewpoints. This can be a tool for experimenting with new machinery, designing new solutions etc. without the need of measuring and drawing the entire factory.

For a large-scale application like the example given, the Kinect (or in general, a 3D-camera) may not be the suited sensor. A large amount of 3D-scanners are available with a very large range and little noise compared to the Kinect. Regardless of the sensor, the registration process is still the same. The set of algorithms presented in Section 4.5.1 are created with the intention on solving one specific problem. Therefore, a more general approach were implemented in the main program. The general approach is given in Algorithm 6. The tests were done with one Kinect placed in the middle of the office room, taking many snapshots while rotating the Kinect.

Algorithm 6 Registration in general

1: Down-sample with a Voxel grid filter.
2: Estimate normals.
3: Estimate keypoints.
4: Estimate descriptors (features).
5: Compute correspondences between the n-th and (n+1)-th point cloud.
6: Reject bad correspondences.
7: Estimate a transformation based on the good correspondences.
8: Transform the (n+1)-th point cloud based on the estimated transformation.
9: Perform a refined alignment using ICP.

4.7 Object detection

Section 2.7 introduced two well known object detection pipelines, based on local and global feature description. In both pipelines, it is assumed that the point cloud is captured from a single viewpoint. From a single viewpoint, only a excerpt of the entire model is shown. Detecting that excerpt part requires a training database, often up to 100 different models, from different viewpoints to be able to match the output of the camera. Training can be done either by ray-tracing or taking multiple pictures using a rotating table.

The objective in this thesis however, is to use the output from multiple cameras. By combining the information from multiple cameras, it is possible to remove the downside of only being able to see an excerpt of the entire model. Since the final output contains the entire model, there is no longer the need of a training database. Only one model is needed to match the output from the cameras.

As mentioned in Section 2.5, we categorize feature descriptors as local or global, where the local is estimated for each point in the cloud and the global is estimated for a whole cluster. The global feature descriptor also has a viewpoint direction component, derived from ray-tracing. Since the combined data output from the cameras yield a fully reconstructed model, the viewpoint component of the feature is not needed. This led to the proposed object recognition pipeline for a multiple camera setup being based on using local feature descriptors. This type of feature descriptors does not have a viewpoint component.

Figure 4.9 gives an overview over the major steps in the proposed object recognition pipeline.

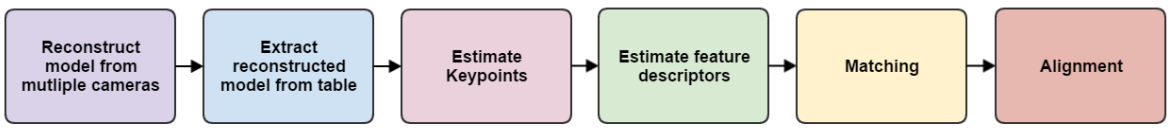


Figure 4.9: Proposed object recognition pipeline for a multiple 3D-camera setup.

The first step of the proposed object recognition pipeline is to use the results from the extrinsic camera calibration (from Section 4.5) to reconstruct an entire model for the entire scene from multiple camera viewpoints. This is done by transforming the point clouds from camera 2 and camera 3 to the same viewpoint as in camera 1.

The next step is to extract the object cluster from the reconstructed scene. This can be achieved by a combination of filtering methods. First, a passthrough filter is used to remove most points of the point cloud besides the table and its content. The next step is to use planar segmentation to remove the tabletop, and use cluster extraction to extract the object to be detected.

After the object cluster is extracted from the full scene, we want to match that cluster towards the an imported CAD model that has been converted to a point cloud. To be able to match the two clouds, local feature descriptors are estimated at keypoints in both clouds.

The matching process is similar to the one in registration. First, correspondences are found between all feature descriptors. A combination of correspondence rejection methods are used to reject bad correspondences between points. Finally, alignment is achieved by estimating a transformation based on the good correspondences. A refined alignment is done using ICP.

4.7.1 Experimental setup

To be able to test the proposed object recognition pipeline using multiple cameras, the object to be detected was placed on measured locations on the tabletop. These measured locations were measured relative to the master AR tag, described in Section 4.5.2. Regardless of the extrinsic calibration method, the AR tracking method has to be used to know the transformation from the main camera to the table. When moving the robot to the desired location, we need to find the location of the object relative to the world or robot coordinate frame.

4.8 Robotics

As mentioned in Section 4.2, the real-time computer in the robot cell is running as the ROS master. Here, the ROS node that handles the communication towards the robot controllers are running. This node provides three different ROS services (for each robot):

- Service for planning a robot pose.
- Service for moving the robot to a pose.
- Service for opening and closing the end effector gripper.

These services were implemented as an extension on the previously developed node in collaboration with fellow students working in the same robotic cell. Creating services for planning and moving the robot allows faster and easier testing, and can easily be reused in future projects at NTNU. There was also no current implementation of gripper manipulation available in the robot cell in ROS. The services for planning and moving the robot requires pose information, which is the X-, Y-, and Z-coordinates as well as rotation in roll, pitch and yaw relative to the world coordinates. The ROS class in the main program sets up a connection to each of the total six services at start-up. To specify the pose values to send to the service, input boxes were implemented in the GUI class, as well as buttons to plan a pose and move to a pose.

When starting the ROS node that communicates with the robot controller, a predefined setup of RViz starts up, that visualize the robots position in real-time. In this RViz window it is possible to see where the robots position will be when calling the planning robot pose service.

Visualizing robot movements is a good tool to prevent the robots to go to a pose that can be dangerous for the surrounding environment and itself.

Chapter 5: Result

5.1 Main program

The final main program graphical user interface (GUI) is shown in Figure 5.1. It is divided into two main parts, the visualization part (left side) and the manipulation panel (right side). The left side contains three parts, on top it is possible to subscribe to a published point cloud topic, and the two large visualizers are point cloud visualizers. The visualizer on the left side is the "main" visualizer that shows raw-data, and the visualizer on the right side shows a point cloud that has been manipulated in some matter (filtering, normal estimation, etc.).

The right side of the GUI consists of a detachable manipulation panel. Detaching the panel increases the size of the visualizers, which was a nice gimmick to have when using two monitors. The manipulation panel consists of four tabs:

- Filters
- Log
- Tester
- Move Robots

In the filter tab it is possible to:

- Add a saved point clouds to the main visualizer.
- Save a filtered point clouds.
- Reload a filtered point cloud to the main visualizer.
- Filter the current point cloud in the main visualizer.

Selecting a filter from the drop-down list will update which parameters are available on that current filter. In Figure 5.1, raw-data from the robot cell (from one camera) is shown in the main visualizer, and the same point cloud filtered by a Voxel Grid filter is shown in the right visualizer. There is also implemented a "Auto Filter" check-box. When this check-box is checked, the currently selected filter will automatically run and update point cloud on the right visualizer when adjusting the filter parameters. Figure 5.2a shows an excerpt of the filter tab with the current filter being the passthrough filter.

When testing new filter combinations, it is often common to forget which parameters the last filtering methods had. The log tab (shown in Figure 5.2b) creates a log of which filter was used and what parameters were used. This log updates each time the filtered cloud is reloaded to the main visualizer.

The tester tab in Figure 5.2c was created as a free space in the GUI to add tester buttons for testing different approaches for objectives for this thesis.

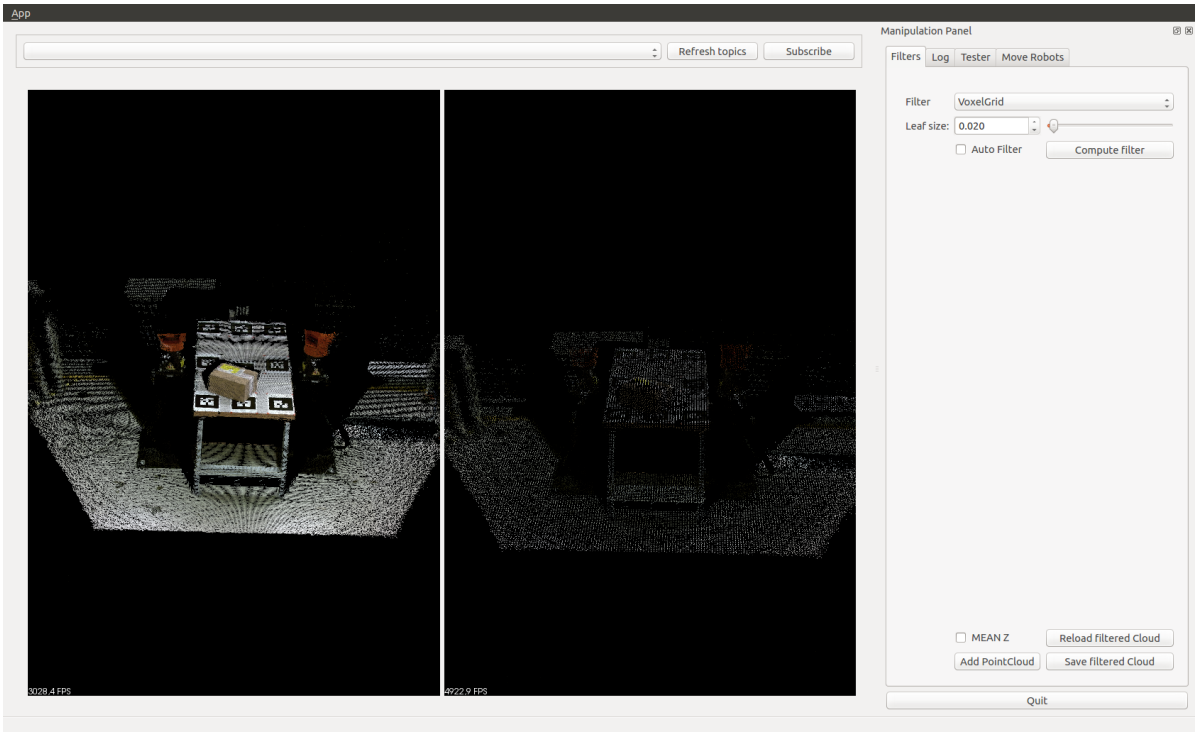
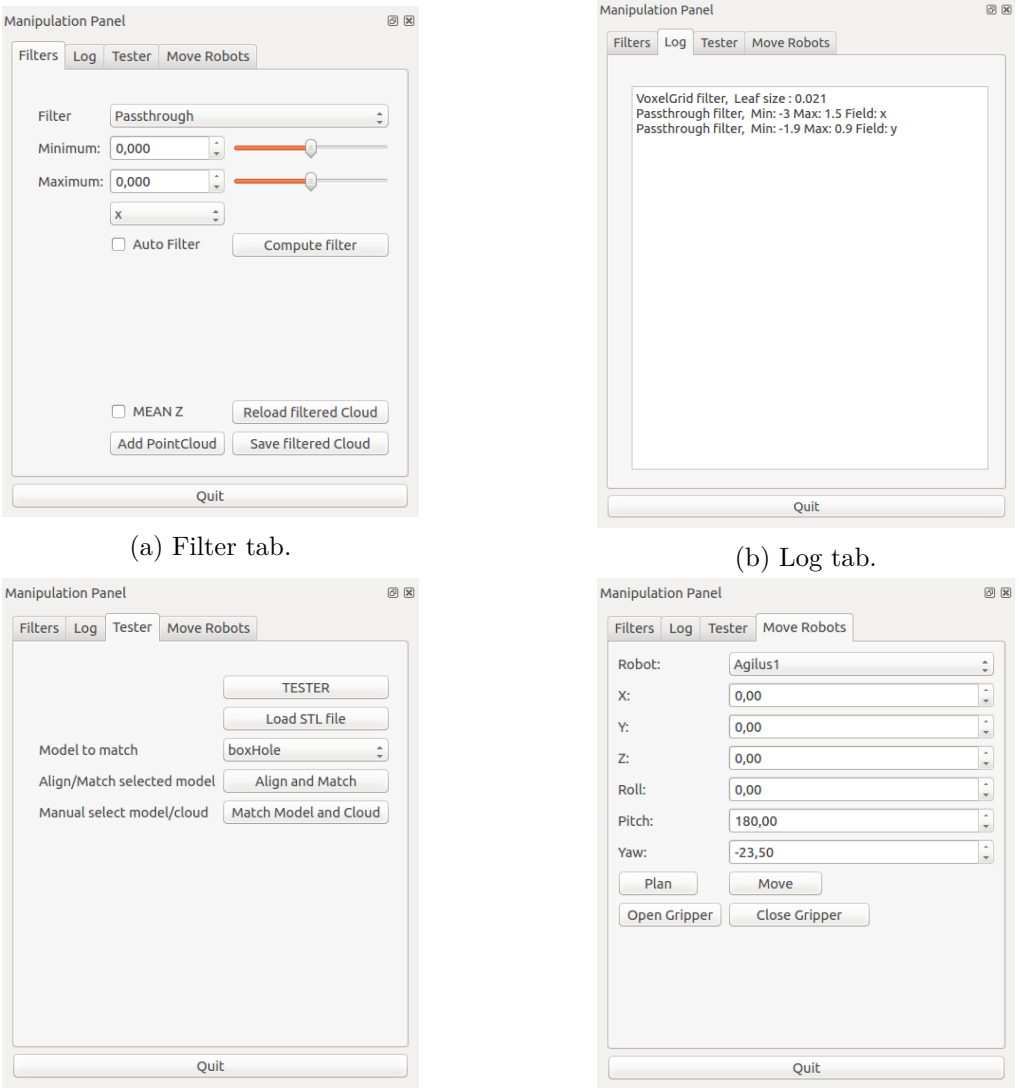


Figure 5.1: The graphical user interface of the main program.

The last tab, shown in Figure 5.2d, is the move robots tab. As you can see, there are input boxes for the robots position and orientation. At the top it is possible to choose between the two robots in the robot cell, *Agilus 1* and *Agilus 2*. Pressing the *Plan* button will call the planning service and visualize the planned motion in RViz. The *Move* button will move the robot to the selected position and orientation.

The source code for the main program can be found in Appendix C, and on GitHub at `https://github.com/sindreraknes/qt_master`. In Appendix C, the graphical user interface class, *main_window.cpp* and *main_window.hpp*, is not included, but can be found in the digital appendix.



(a) Filter tab.

(b) Log tab.

(c) Tester tab.

(d) Move Robots tab.

Figure 5.2: Overview of all four different tabs in the manipulation panel in the main program GUI.

5.2 Camera calibration

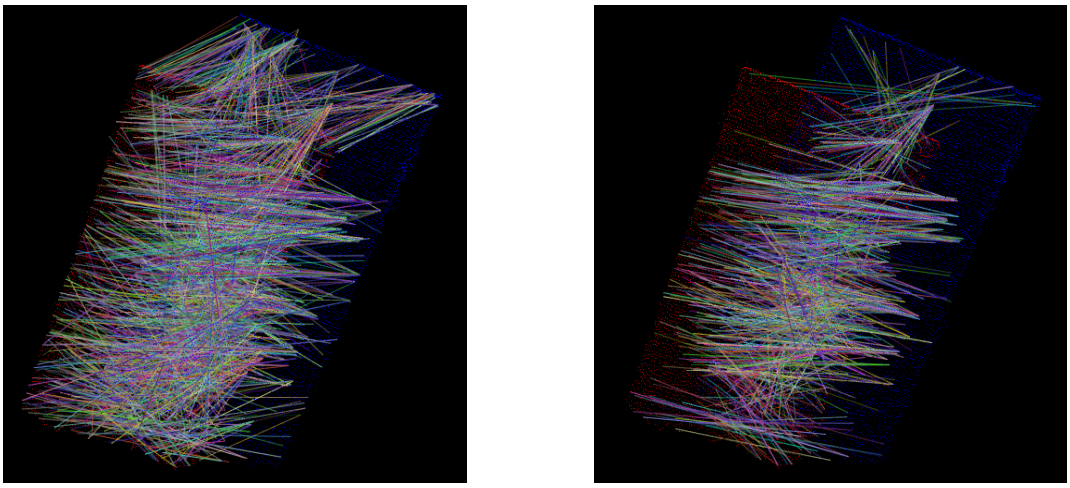
5.2.1 Approach 1

The major steps for this approach, using registration, was explained in detail in Section 4.5.1. This approach was the first test conducted in this thesis, and at the time, the table was in front of the two robots in the robot cell. One of the cameras was also placed on another location. The idea of using the tabletop as the part to use registration on, required all the cameras to see the whole tabletop. This can be seen in Figure 5.6.

The initial tests were done with a clean table with no objects on top. It was quickly discovered that this approach led to a major problem. The registration algorithm were not able to distinguish if the table was upright or upside down. This led to poor and unstable results, which is not something that is wanted for a calibration program.

The next tests were performed with 3 sections of a 60mm outer diameter pipe with the same length, placed on top of the table. The intention with using these pipes was to lure the system into knowing the difference between a upright and upside down tabletop. Most extrinsic camera parameter calibration tools use some kind of known object or feature. This approach led to more stable results, but in some cases the results were the same as in the initial tests.

From Section 4.5.1, we introduced Algorithm 5. The first step in the algorithm was to estimate the correspondences between the feature descriptors. The results are shown in Figure 5.3. The figure illustrates the correspondences by drawing lines between the corresponding points. Figure 5.3a shows the correspondences before a rejection method has been applied, and Figure 5.3b shows the correspondences after.

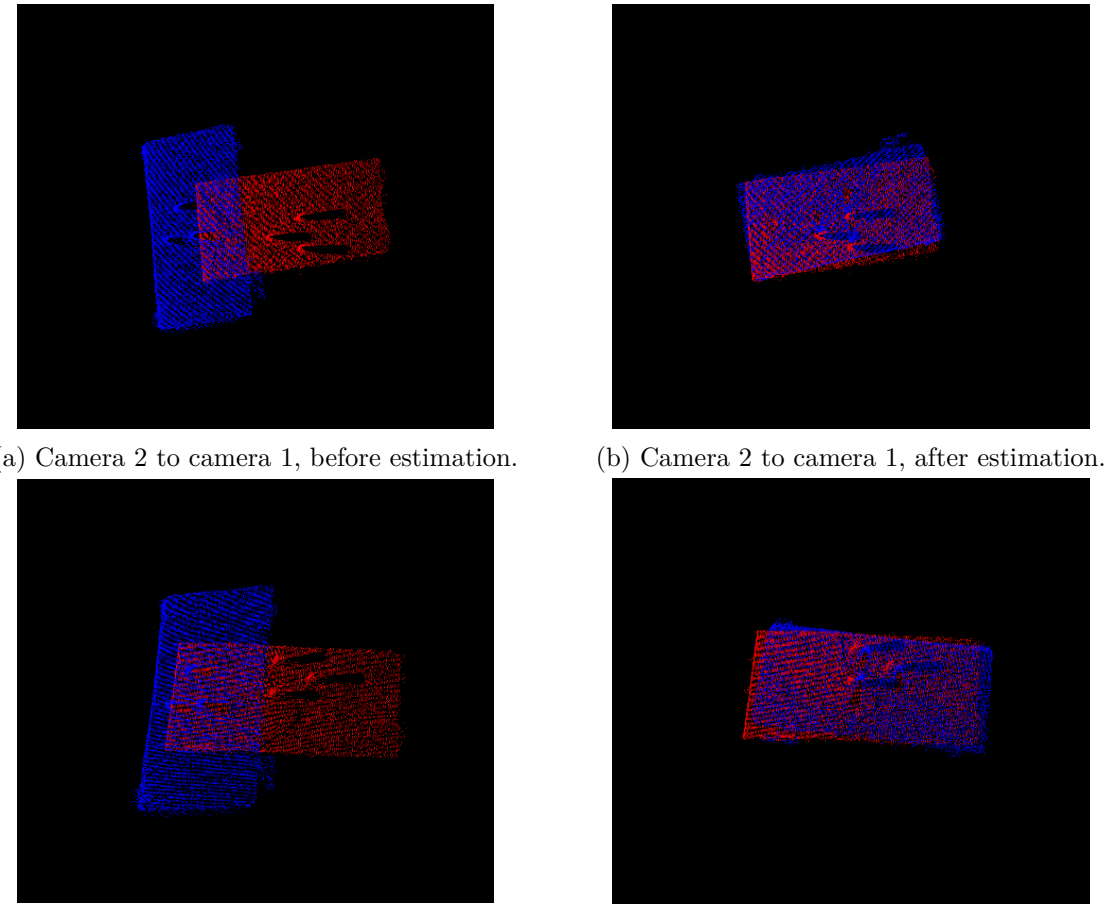


(a) Before correspondence rejection.

(b) After correspondence rejection.

Figure 5.3: Correspondences before and after rejection between two tabletops.

The next step in the algorithm is to estimate a transformation based on the good correspondences, shown in Figure 5.3b. The results of the transformation estimation for camera 2 and camera 3 to camera 1 is shown in Figure 5.4. The red point cloud in the figures is the tabletop seen from camera 1.



(a) Camera 2 to camera 1, before estimation.

(b) Camera 2 to camera 1, after estimation.

(c) Camera 3 to camera 1, before estimation.

(d) Camera 3 to camera 1, after estimation.

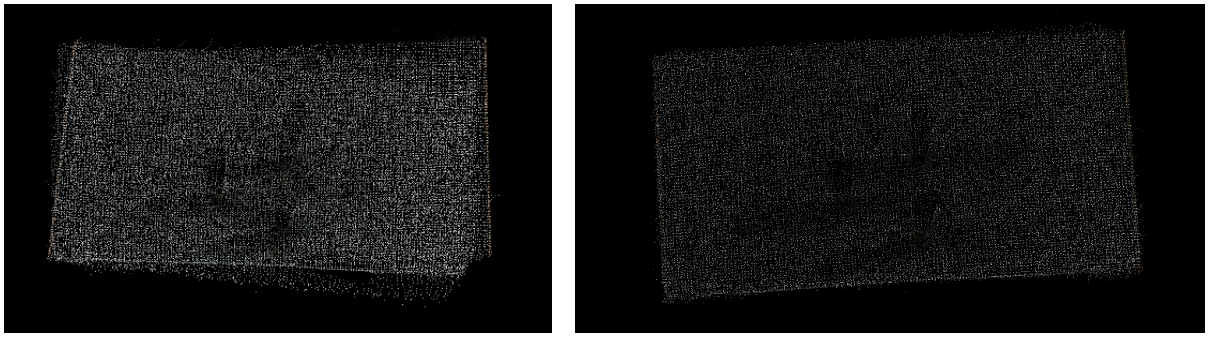
Figure 5.4: Transformation estimation based on good correspondences.

As you can see in Figure 5.4, the transformation resulted from the correspondences is not adequate. The final step in Algorithm 5 is to use Iterative Closest Point (ICP) to find a refined alignment. Figure 5.5 shows the three tabletops before (Figure 5.5a) and after (Figure 5.5b) refined alignment.

The final extrinsic parameters (transformation matrices to camera 0) for camera 2 and camera 3 were found to be:

$$T_{cam1}^{cam2} = \begin{bmatrix} 0.0282982 & -0.663595 & 0.747609 & -1.78142 \\ 0.710821 & 0.539209 & 0.451709 & -0.752916 \\ -0.70284 & 0.518611 & 0.486936 & 0.616017 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.1}$$

$$T_{cam1}^{cam3} = \begin{bmatrix} 0.0354387 & 0.469684 & -0.88216 & 1.7342 \\ -0.702385 & 0.639638 & 0.31234 & -0.67222 \\ 0.710945 & 0.608531 & 0.352557 & 0.816263 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.2}$$



(a) Before refined alignment. (b) After refined alignment.

Figure 5.5: Refined alignment after transformation estimation.

These matrices were then used to reconstruct the entire scene, shown in Figure 5.6. The coordinate systems in the figure is the camera optical frame. The Z-axis is blue (optical axis), Y-axis is green and the X-axis is red.

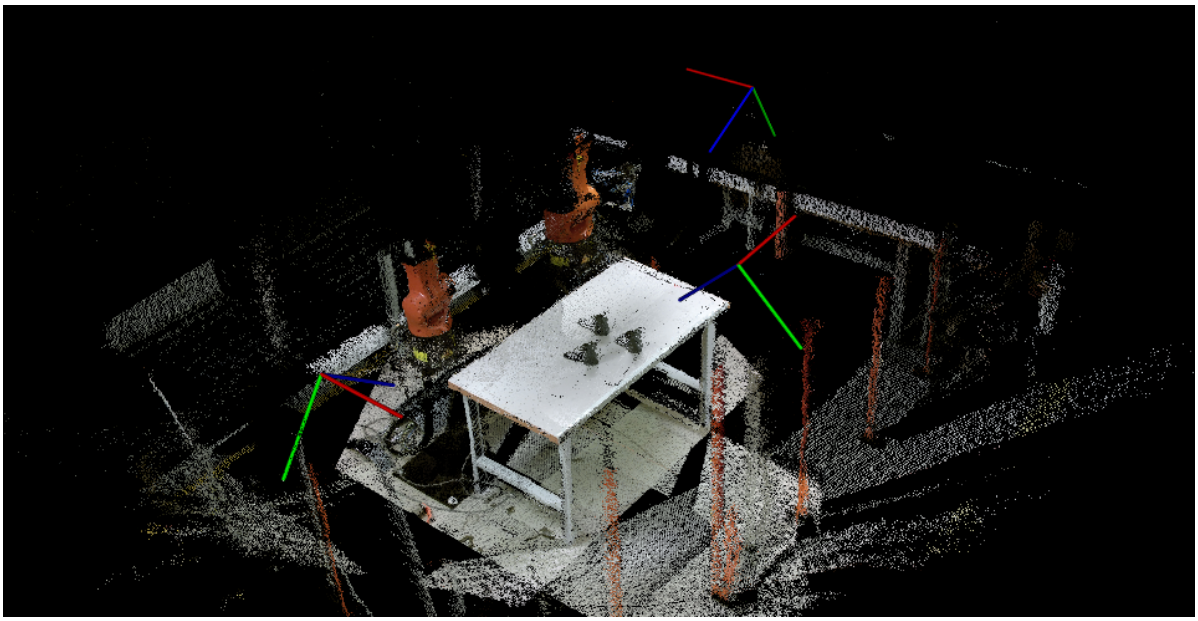


Figure 5.6: The reconstructed scene after camera calibration with registration.

5.2.2 Approach 2

The second calibration approach, explained in Section 4.5.2, is based on tracking known features (AR tags) in the point cloud. The table was moved in between the robots before the testing of this approach were conducted. It was moved to give the robots a larger workspace. One of the cameras, the one to the far left in Figure 5.6, was also moved to a location behind the two robots.

Tracking the AR tags could be done either individually, or as a bundle. After the table and one camera was moved, tracking a individual tag that could be seen from all cameras were

challenging due to occlusion. Thereby, tracking a bundle was chosen as the main approach here. A bundle of nine AR tags of the same size with different ID were placed on the table. Figure 5.7 shows the AR tags placed at the table. The AR tags were place according to a predefined XML file. The predefined XML file can be seen in Appendix A.

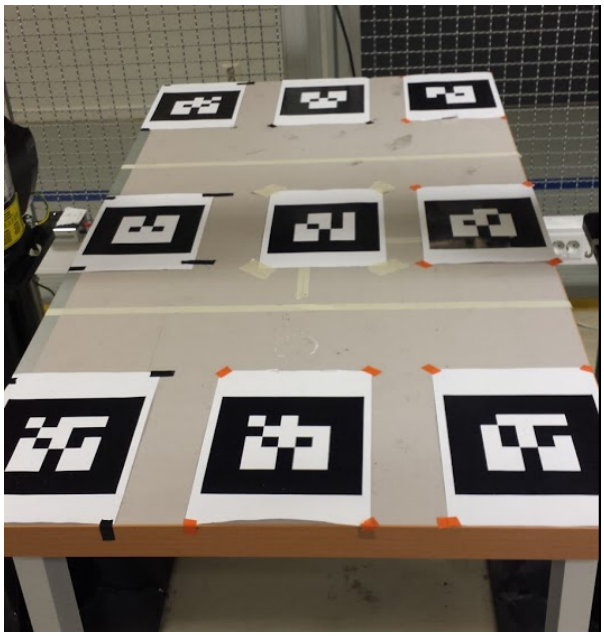


Figure 5.7: AR tag placement on the top of the table.

It was early found out that the published topic by *ar_track_alvar* sometimes published a position and orientation that were not optimal to use. To deal with this problem, the ROS node created for calibration of extrinsic parameters took the average of a user chosen number of measurements. This led to more stable results. Tracking the master tag and averaging 100 measurements for each camera, gave the following results:

$$T_{tag}^{cam1} = \begin{bmatrix} -0.0324064 & 0.999472 & 0.00236665 & -0.236723 \\ 0.701194 & 0.0244224 & -0.712552 & -0.589319 \\ -0.712233 & -0.0214317 & -0.701615 & 1.82195 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.3}$$

$$T_{tag}^{cam2} = \begin{bmatrix} -0.0369295 & -0.99916 & 0.0177825 & 0.174928 \\ -0.592998 & 0.00758757 & -0.805168 & 0.016518 \\ 0.8043570 & -0.0402794 & -0.59278 & 0.945164 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.4}$$

$$T_{tag}^{cam3} = \begin{bmatrix} -0.999718 & -0.0224201 & -0.007764 & 0.604796 \\ -0.0040468 & 0.483571 & -0.875296 & -0.256756 \\ 0.0233787 & -0.875018 & -0.483525 & 2.14104 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.5}$$

These transformation matrices were then used to reconstruct the scene, which is shown in Figure 5.8, along with the camera positions.

The source code for the entire calibration program can be found in Appendix B, and on GitHub at https://github.com/sindreraknes/calib_robot_cell.

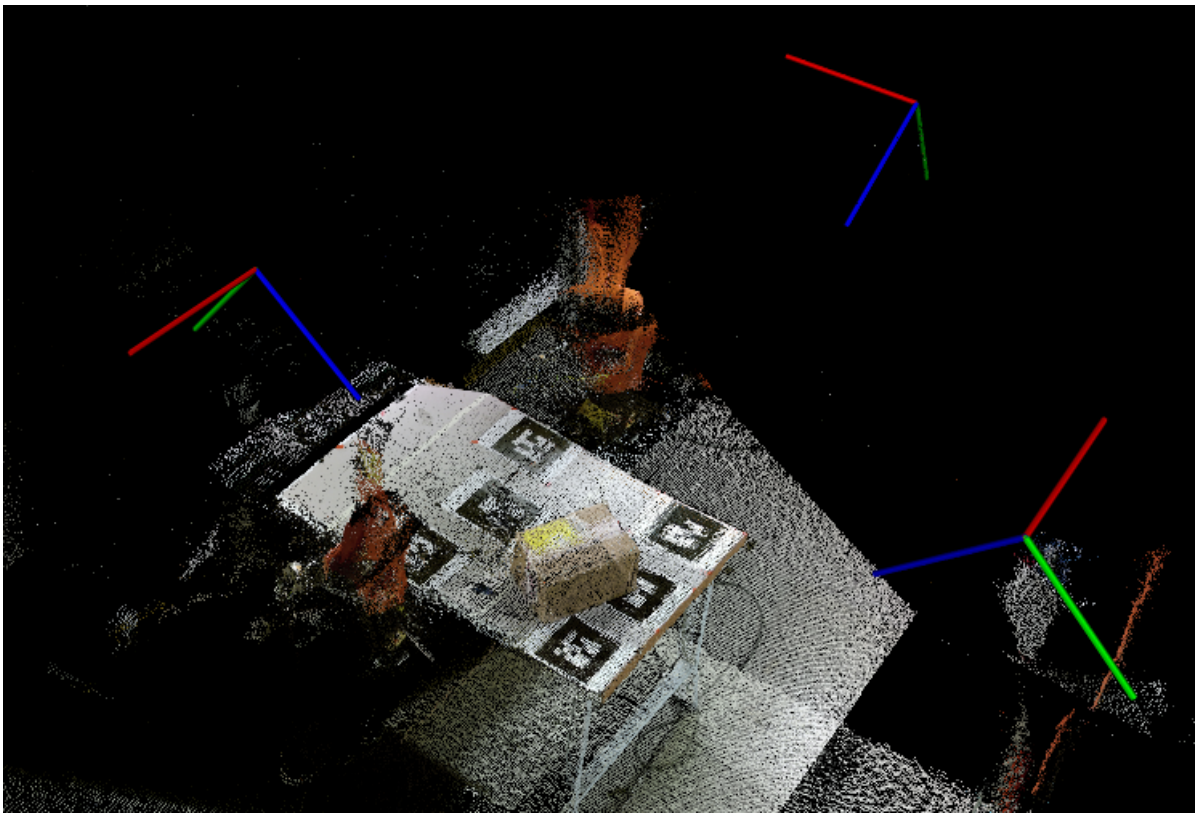


Figure 5.8: The reconstructed scene after camera calibration with AR tags.

5.3 Registration

The result from the general registration approach can be seen in Figure 5.9. One of the things discovered by testing the general approach, is that the point clouds to be pairwise registered need to overlap around 50%. If the point clouds have little overlap, the general registration approach often find very few good correspondences. This leads to poor transformation estimations, which again can alter the results of ICP. Running ICP on two point clouds that are distant from each other will in most cases fail. This is mainly due to the parameters set for the algorithm, such as maximum distance between points. Taking into account the problem on the other hand, gave good results.

Figure 5.9: The reconstructed office room using a general registration approach with multiple point clouds.

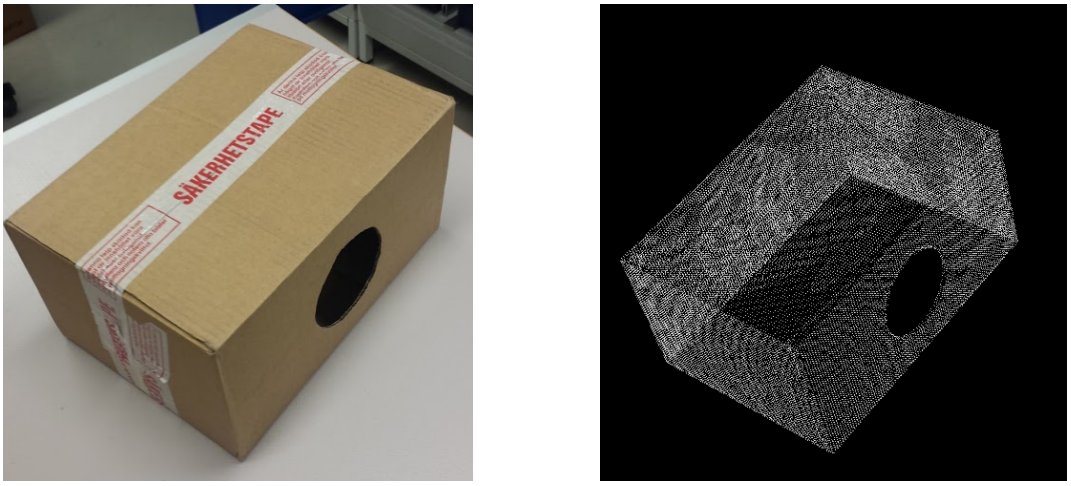
5.4 Object detection

Section 4.7 proposed a object recognition pipeline based on local feature descriptors. This pipeline is somewhat similar to the local pipeline mentioned in Section 2.7. The experiments were carried out by placing a box with a hole in one of the sides on known locations on the tabletop. The box to be detected is shown in Figure 5.10, both the real box and the point cloud version. These locations were relative to the master AR tag. The three locations used for testing were (The Z-coordinate is known):

- Position 1: X:61,7cm Y:29,0cm
- Position 2: X:61,7cm Y:58,6cm
- Position 3: X:47,7cm Y:29,0cm

The following is the walk-through of one of the conducted experiments, with more detailed results in the end of this Section. The first step of the pipeline was to reconstruct the scene (model) from multiple camera viewpoints. This was, in the end, done by using the AR tracking calibration system. A top view of the reconstructed model from the calibration system is shown in Figure 5.11. As can be seen in the figure, all four sides of the box is reconstructed to a

complete model, including the top of the box.



(a) Real box. (b) CAD model of box converted to a point cloud.

Figure 5.10: The box to be detected.

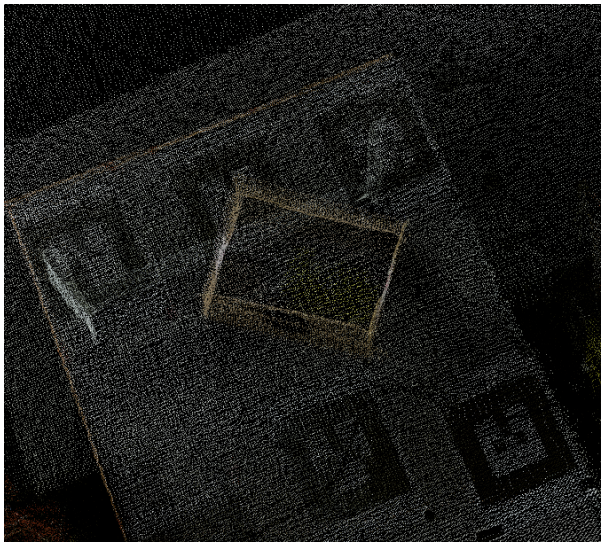


Figure 5.11: Top view of the reconstructed model.

The next step is to extract the object from the full scene. The resulted model from the cluster extraction and plane segmentation combination is shown in Figure 5.12 (The points are enlarged for visualization purposes).

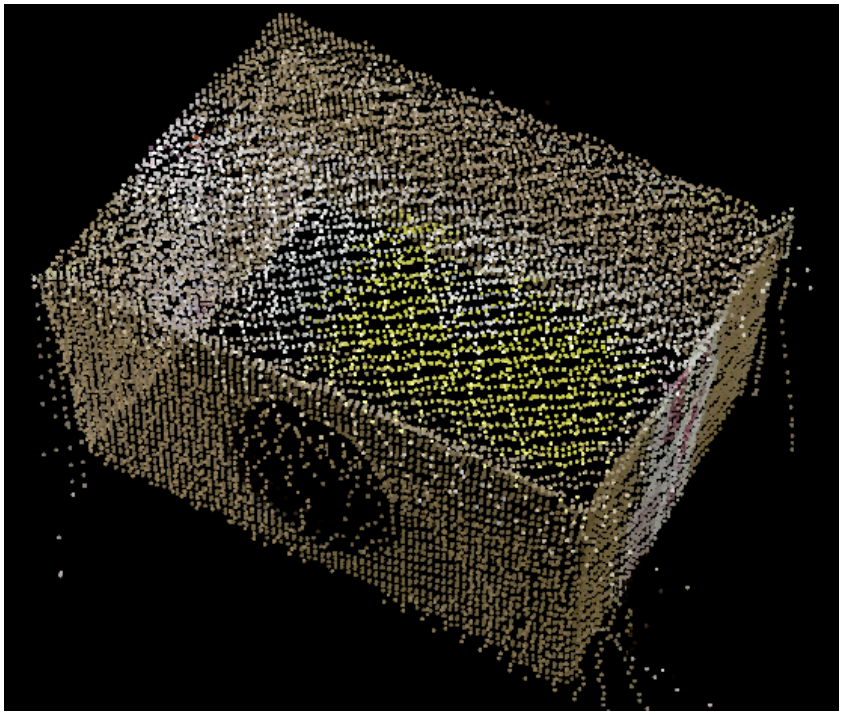
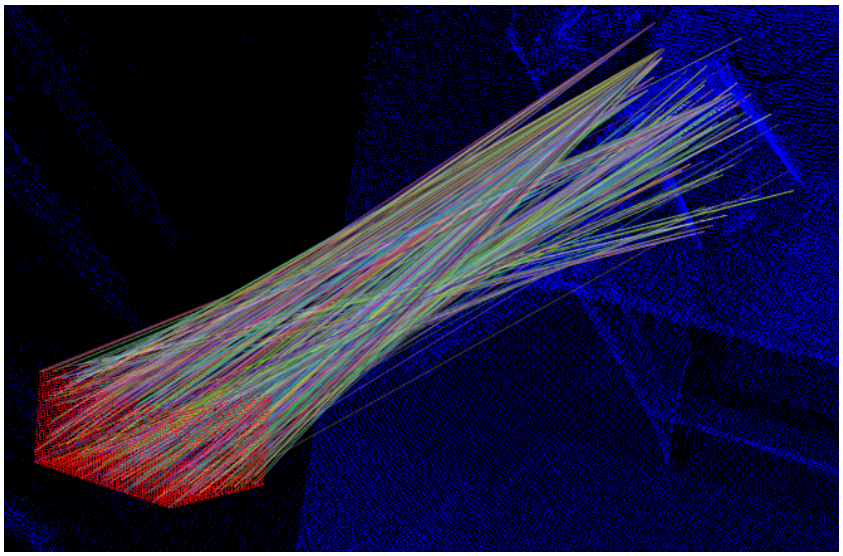
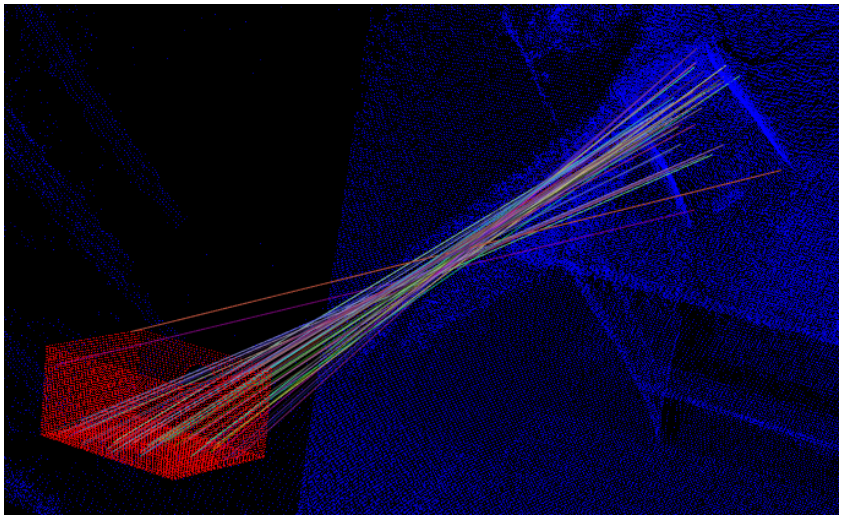


Figure 5.12: The extracted model from the full scene.

When the model is extracted from the scene, keypoints and local feature descriptors are estimated on the CAD model point cloud (in Figure 5.10b) and the extracted point cloud model. As in the registration approach, correspondences are estimated between the local feature descriptors on each keypoint. Figure 5.13 shows the estimated correspondences before and after correspondence rejection. The red part of the figure is the CAD model point cloud, and the blue part is the full scene (the full scene is showing for visualization purposes).



(a) Correspondences before rejection.



(b) Correspondences after rejection.

Figure 5.13: Correspondences before and after rejection.

After the good correspondences has been found, the next step is to use the correspondences to estimate a initial transformation from the camera frame (camera 1) to the object on the table. The initial transformation was found to be (All translation vectors are given in meters):

$$T_{object}^{cam1} = \begin{bmatrix} -0.777476 & 0.103337 & 0.620365 & 0.0462748 \\ -0.488611 & -0.720298 & -0.492372 & -0.178851 \\ 0.395967 & -0.685925 & 0.610506 & 1.39579 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.6}$$

Transforming the CAD model point cloud using matrix 5.6 is shown in Figure 5.14. The CAD model point cloud is colored red. As you can see in the figure, the initial transformation does not give a perfect match.

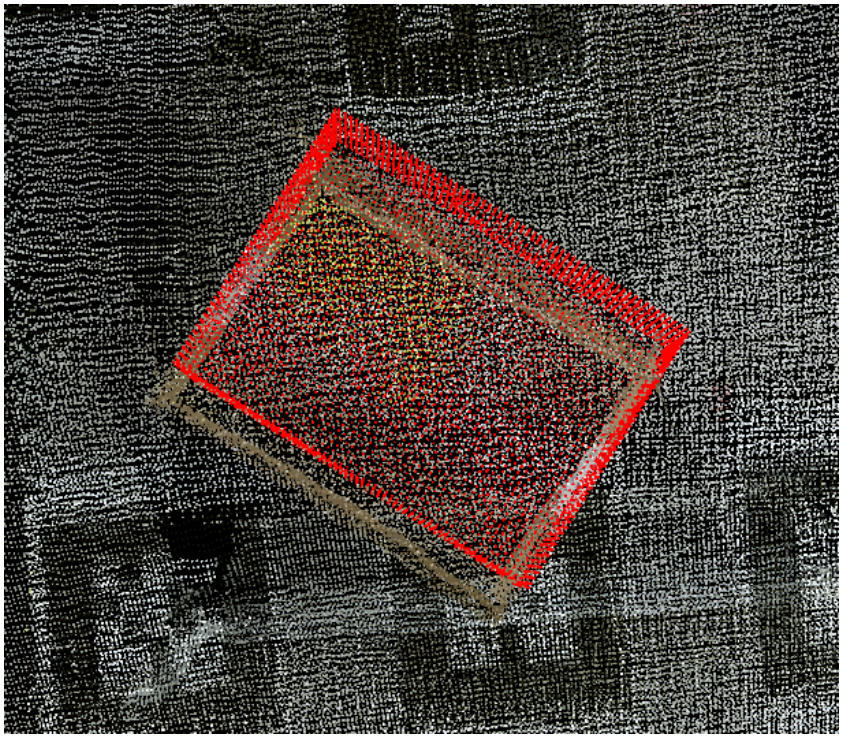


Figure 5.14: Initial alignment based on the good correspondences.

Since the initial transformation does not match the CAD model point cloud to the reconstructed model, an iterative approach is used to refine the alignment to a perfect match. The results from running ICP after the initial alignment is shown in Figure 5.15. The refined alignment adjusted the initial transformation from camera 1 to the object into:

$$T_{object}^{cam1} = \begin{bmatrix} -0.778044 & 0.0223296 & 0.627831 & 0.0337993 \\ -0.469738 & -0.684281 & -0.557794 & -0.161939 \\ 0.417148 & -0.728898 & 0.542885 & 1.39309 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.7}$$

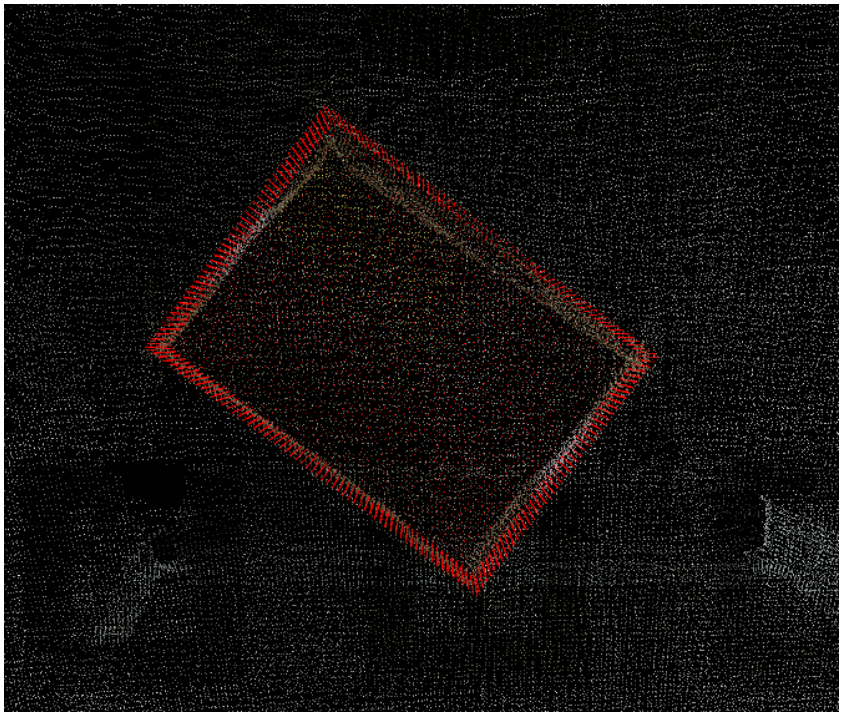


Figure 5.15: Refined alignment with Iterative Closest Point.

Now that the final transformation from the camera to the object is found, we want to find the position in table coordinates (for the experiments) and in world coordinates (for the robots). From the AR tag tracking, the transformation matrix from camera 1 to the master tag was found to be:

$$T_{tag}^{cam1} = \begin{bmatrix} -0.0324064 & 0.999472 & 0.00236665 & -0.236723 \\ 0.701194 & 0.0244224 & -0.712552 & -0.589319 \\ -0.712233 & -0.0214317 & -0.701615 & 1.82195 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.8}$$

By using basic kinematic knowledge, the transformation from the master AR tag to the object is:

$$T_{object}^{tag} = inverse(T_{tag}^{cam1}) * T_{object}^{cam1} \tag{5.9}$$

$$T_{object}^{tag} = \begin{bmatrix} -0.601271 & 0.0386085 & -0.798129 & 0.596359 \\ -0.798045 & 0.0212276 & 0.602241 & 0.290008 \\ 0.0401938 & 0.999045 & 0.0180465 & -0.002995 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.10}$$

To be able to find the objects coordinates in the world coordinate system, the transformation from the world coordinate system to the master AR tag was needed. The world coordinate

system is located on ground level in the middle of the robots. Measuring the distances from the world coordinate system to the master AR tag gave the following transformation matrix (No rotation, meaning identity matrix for the rotation part):

$$T_{world}^{tag} = \begin{bmatrix} 1 & 0 & 0 & -0.084 \\ 0 & 1 & 0 & -0.292 \\ 0 & 0 & 1 & 0.87 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.11}$$

The objects position in the world coordinate is then given by:

$$T_{object}^{world} = T_{tag}^{world} * T_{object}^{tag} \tag{5.12}$$

$$T_{object}^{world} = \begin{bmatrix} -0.601271 & 0.0386085 & -0.798129 & 0.512359 \\ -0.798045 & 0.0212276 & 0.602241 & -0.00199163 \\ 0.0401938 & 0.999045 & 0.0180465 & 0.867005 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.13}$$

The final results is shown in Figure 5.16. In this figure you can see the CAD model object placed in the correct position, and coordinate systems for the object origin, AR master tag and world (seen under the table).

The final pose message that the robot needs to go to a position requires as explained earlier, X-, Y- and Z-coordinates along with angles described by roll, pitch and yaw. From the theory in Section 3.1.2, the roll, pitch and yaw angles from matrix 5.13 becomes:

$$\begin{aligned} \varphi &= 89.91° && (roll) \\ \vartheta &= -2.3° && (pitch) \\ \psi &= -127.06° && (yaw) \end{aligned} \tag{5.14}$$

and the positions extracted from matrix 5.13:

$$\begin{aligned} X &= 0.512m \\ Y &= 0.002m \\ Z &= 0.867m \end{aligned} \tag{5.15}$$

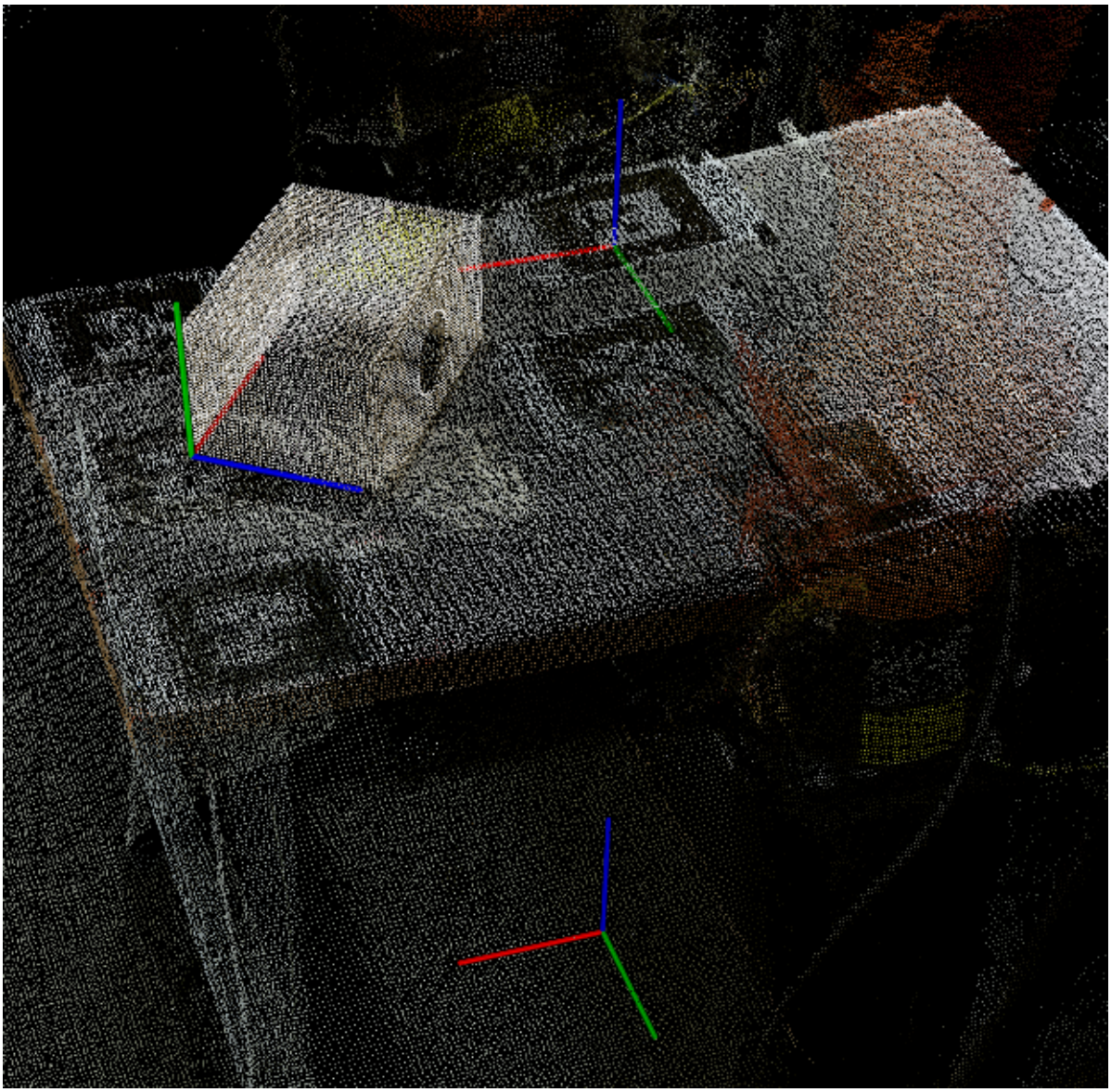


Figure 5.16: Final visualization with detected object and coordinate system at object, AR master tag and world.

To test the robustness and accuracy of the system, experiments were conducted at the positions mentioned at the start of this section. The experiments were carried out by placing the object (the box) at the measured position, and running the proposed pipeline 10 times at the same location. The experimental results are shown in Table 5.1, 5.2 and 5.3.

Table 5.1: Experiments in position 1.

Position 1						
Test nr.	**X [cm]**	**Y [cm]**	**Z [cm]**	**Roll [°]**	**Pitch [°]**	**Yaw [°]**
1	59.64	29	0	89.91	-2.3	-127.06
2	60.98	28.51	0	89.74	-2.5	-127.2
3	59.85	29.47	0	89.71	-2.45	-127.09
4	60.45	28.91	0	89.76	-2.45	-127.03
5	60.87	29.16	0	89.95	-2.42	-127.1
6	59.84	29	0	89.77	-2.42	-127.08
7	60.57	29.07	0	90.1	-2.45	-126.98
8	60.76	28.31	0	89.72	-2.36	-127.04
9	60.78	28.46	0	90.02	-2.17	-126.99
10	60.06	29.21	0	89.89	-2.39	-126.89

Table 5.2: Experiments in position 2.

Position 3						
Test nr.	**X [cm]**	**Y [cm]**	**Z [cm]**	**Roll [°]**	**Pitch [°]**	**Yaw [°]**
1	60.08	59.99	0	88.14	0.62	-65.93
2	61.6	59	0	88.18	0.54	-65.86
3	61.05	59.41	0	88.26	0.77	-66.13
4	61.68	58.45	0	88.17	0.44	-65.84
5	60.61	59.45	0	88.03	0.73	-65.74
6	60.71	58.9	0	88.04	0.6	-66
7	62.02	59.47	0	88.04	0.76	-65.89
8	60.87	59.02	0	88.12	0.56	-66.13
9	61.53	59.46	0	88.33	0.7	-66.1
10	61.67	58.03	0	88.22	0.46	-66.12

Table 5.3: Experiments in position 3.

Position 3						
Test nr.	**X [cm]**	**Y [cm]**	**Z [cm]**	**Roll [°]**	**Pitch [°]**	**Yaw [°]**
1	46.68	28.2	0	94.82	-0.86	126.13
2	47.93	29.01	0	94.68	-0.82	126.28
3	47.25	28.21	0	94.8	-1	126.1
4	48.03	29.64	0	94.77	-1.04	126.22
5	46.75	28.56	0	94.69	-0.95	126.32
6	46.69	29.67	0	94.99	-0.93	125.94
7	46.74	28.2	0	94.82	-0.86	126.29
8	46.75	30.12	0	95	-0.79	125.93
9	48.54	29.95	0	94.68	-0.86	126.06
10	47.63	28.42	0	94.72	-0.9	126.01

As you can see in the tables, the results show poor repeatability in regards to the positions X, Y and Z, which can be a problem since robotic grasping requires high accuracy poses. The orientation (roll, pitch and yaw) is not very affected by variation. Since the experiments were carried out by not moving the object, these variations should not appear, and might originate from the sensors. This led to the idea of looking into the repeatability of the Kinect™, by measuring the Z-coordinate (depth) of a known point for a large number of point clouds, to see if the sensor was causing the bad repeatability, which again will lead to false reconstruction of the scene.

For each camera, 1000 measurements of the same point were made. The results for camera 1 are shown in Figure 5.17 (the other two results are somewhat the same). Table 5.4 summarizes important data from the 3 cameras. The table shows the maximum measured, minimum measured, the difference between maximum and minimum, and the average of the Z-values at the same point. What is interesting to see in the table, is that there is a connection between the average distance and the difference between maximum and minimum measurement. The larger the average distance, the larger difference between maximum and minimum. For camera 2, this means that the Z-coordinate for two consecutively acquired point clouds can have a variation up to $2.3cm$.

Another aspect noticed from Figure 5.17 is that it seems like the measured points follow a typical result from a step response of some kind of regulator. It is known from Grzegorzek et al. (2013) that ToF sensors can be affected by temperature variations, which might cause this problem. In the first 500 points you can see that the Z-value is increasing in the start and then decreasing to a more stable area. To check if this stable area really is stable for more points, 20000 measurements on the same point were conducted for camera 3. The results are shown in Figure 5.18. What is interesting to see in this figure is that the first 1000 points look identical to the points in Figure 5.17. After this section, the Z-value increases and seems to become stable around 10000 measurements. Even though it seems to be stable, there still is a variation up to $1.5cm$.

The consequences of these findings, that the Kinect™has poor repeatability, is that it influences the extrinsic parameters found in the camera calibration methods. This affects the

reconstruction of the scene and model from the different cameras.

Table 5.4: Repeatability measurements of the Kinect™.

Camera	Max measured [m]	Min measured [m]	Difference [m]	Average [m]
1	1.149	1.138	0.011	1.144
2	2.012	1.989	0.023	1.999
3	1.914	1.897	0.017	1.905

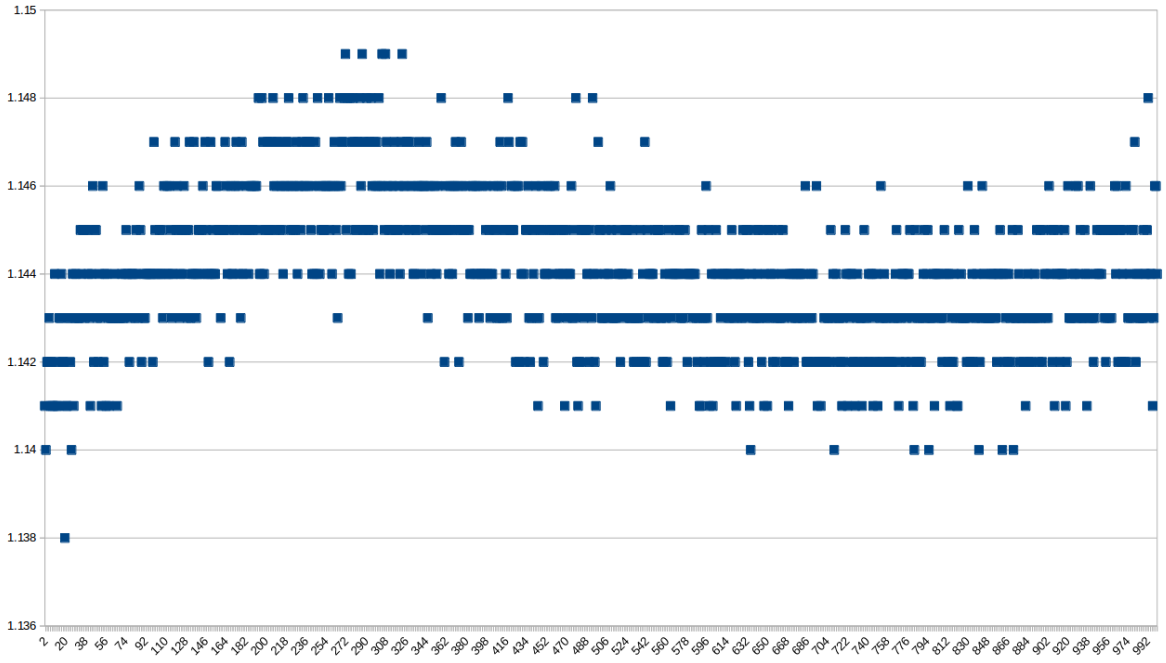


Figure 5.17: 1000 measurements from camera 1.

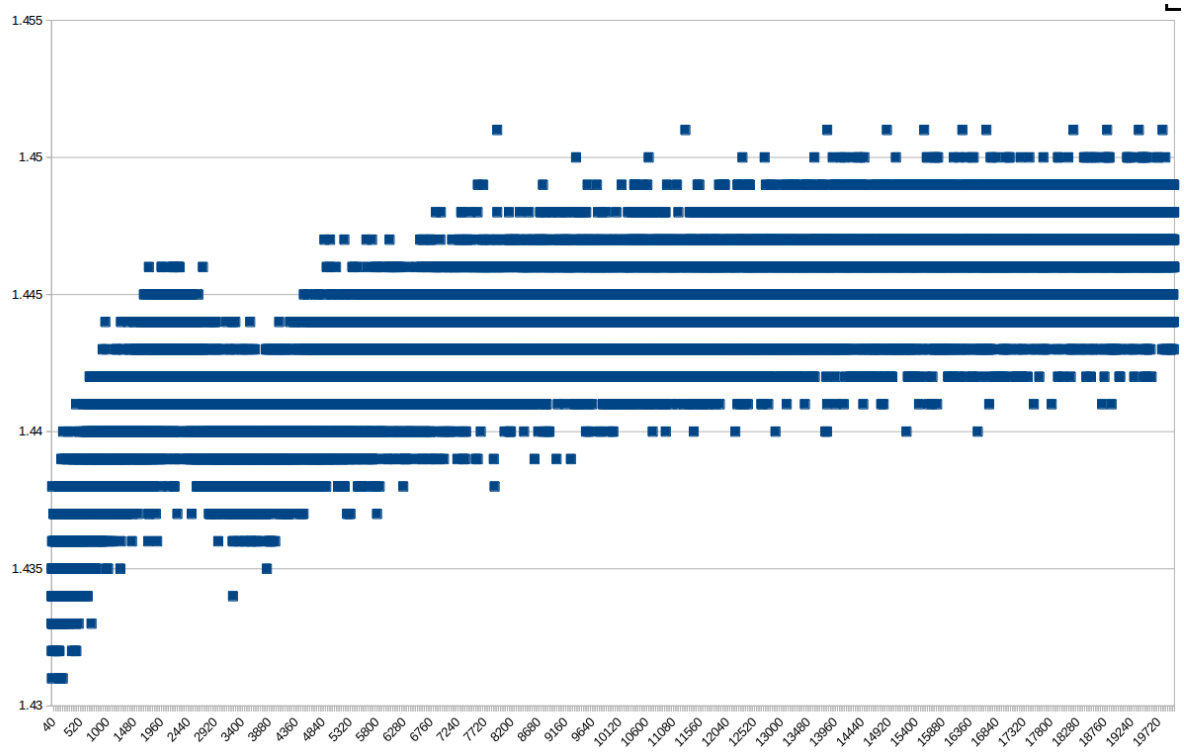


Figure 5.18: 20 000 measurements from camera 3.

5.5 Robotics

From Section 5.4, we saw that the Kinect™had poor repeatability. This led to problems reconstructing a whole model for a small enough object that could be grasped by the robot. Using a larger object (like the box) is not as much affected by the poor repeatability, but this object could not be grasped due to its size. Due to this, the robot that was supposed to grasp an object was moved above the object pose found by the object recognition system. This was mainly to test if the results from the object detection system could be used to move the robots to the correct pose.

Figure 5.19 shows the robot placed above the pose found by the object detection system in position 1. As you can see in the figure, the center of the tool is located directly above the objects origin coordinate system (as shown in the previous Section). The tool also has the correct orientation according to the object coordinate system. This means that the transformation from the object to the world coordinates are correct.

Figure 5.19: Robot manipulator placed directly above the found objects origin.

Chapter 6: Discussion

6.1 Camera calibration

Calibrating the extrinsic parameters for the cameras was a vital part in this thesis, since reconstructing the scene laid the basis for the object recognition. Reconstructing the scene was also more time consuming than initially planned. The first approach, using registration, was found to be unstable and it was difficult to achieve good results. The large angular difference between the cameras led to using an approach that was based on extracting the tabletop from each cameras point cloud. When this approach gave bad results, it was usually due to the matching of feature descriptors for the tabletops. This often led to one of the cameras output were transformed upside-down.

Using an object or multiple objects along with the tabletop somewhat solved this problem, but it still occurred on some occasions. Even though, this approach, using the tabletop, is not very suitable for a general approach. This approach required the use of a passthrough filter to remove the floor planes. If the floor planes are located in the point cloud, the plane extraction algorithm might extract the floor instead of the tabletop, since the floor often is the largest plane present. This means that moving one of the cameras will require manual passthrough filtering beforehand. This approach is more of a situational approach, that needs some kind of manual pre-processing before it can be used.

The second approach, using the AR Alvar library gave much more stable results. This approach, compared to the first, can be used as a general approach. The only requirement if a camera is moved is that it has to be able to locate one of the AR tags. It is also easier to locate a known position on the table or in the world coordinates, relative to the camera. The downside with this approach is that the table is full of printed AR tags. If one of them are damaged or moved slightly, wrong results can occur.

Other approaches

The practical setup in this thesis, using 3 Kinect™3D cameras, is not necessarily the best solution to reconstruct a scene in 3D. The concept of registration is easier to achieve with overlapping point clouds, rather than very unlike point clouds (which we had in this thesis). Acquiring a large number of point clouds to reconstruct the scene can be achieved in two different approaches:

- Using more cameras.
- Use one camera attached to a moving manipulator.

Using more cameras has a big downside, since it requires more hardware and occupies space, and is not a preferred solution. The idea of using one camera, attached to for example a robot, has less downsides. One of the big advantages with this approach, is that you can find very

precise transformations between the different acquired point clouds. This is of course since you always know the pose of the end-effector of a robot. By using this known information, it is not needed to perform registration or a registration like method to reconstruct the scene and the model. An effective approach to this idea is to create a virtual sphere (top of the sphere) over the table, and use this to automatically create poses for the robot. In this way you can *scan* the table and its content from different angles and positions, at a closer distance. This might not be as much affected by the Kinect™repeatability issues (see Section 6.3).

6.2 Registration

Scanning the office (shown in Section 5.3) was a small escapade in this thesis. The idea was to test if the powerful tools in 3D computer vision along with a low cost camera can be used to reconstruct an entire room, office, factory, etc. Scanning a whole factory can help engineers during product development, quality inspection and in general construct the entire factory environment. Scanning has become more and more popular and is usually done with a 3D scanner instead of a 3D camera. 3D scanners have a much larger range (up to kilometers) and normally has a very high repeatability. These scanners are more expensive than a Kinect™by a large margin, and also come with finished software packages.

The increased popularity of the 3D scanners will probably mean that cheaper versions will come along in the near future. A 3D scanner also outputs a point cloud, meaning you could still use the tools of registration to perform reconstruction. Scanning a large factory can reduce the time drastically compared to measuring every part of the factory.

Another thing that is quite interesting for the future to come, is to combine 3D point cloud data with virtual reality (VR). VR is the *big* new thing at the moment, and it will become easier to develop programs for it in the coming years. VR can be used to show customers or colleagues the new ideas while virtually walking through it.

6.3 Object detection

Reconstructing the model from the scene was the most time consuming and problematic part of this thesis. Initially, the idea was set up a pick-and-place or assembly task based on the computer vision part. Due to the problems with the Kinect™, regarding the repeatability, reconstructing objects that are small enough to be grasped in the robot cell was not achieved in a good enough way that it was possible to detect it with acceptable precision. Since all the cameras has this uncertainty, the reconstruction system has an accumulating error. It was especially challenging to reconstruct a cylinder-like object, that lack sharp edges and corners. Placing the cameras very close to the object could maybe have solved this issue, but choosing a bit larger object still proved the point of this thesis. The results with the box are believed to be accurate enough.

One aspect with converting a CAD model to a point cloud is that the resulted point cloud will have perfectly uniform point density. This is not the case of the reconstructed model. This can lead to a challenge when matching feature descriptors. To gain better results with regards to matching, it could be an idea to scan the object and use that model in the object database. This can be achieved with for example a rotating table. This will give more equal point clods to for matching and give better results.

An evaluation of the Kinect™depth data was conducted by Khoshelham and Elberink (2012). This paper used the first Kinect™, based on structured light. The paper concluded that the random error of depth measurements increases quadratically with increasing distance from the sensor, and reaches $4cm$ at the maximum range of $5m$. Also, the depth resolution decreases quadratically with increasing distance from the sensor. These results are similar to the ones conducted in this thesis, as shown in Section 5.4. The results in this thesis also show that the depth repeatability issue increases with the distance from the measured location. This also means that the Kinect™may not be the best 3D sensor for very accurate and precise robotic vision systems.

A 3D object detection system is often combined by both pipelines mentioned in Section 2.7, using both global and local feature descriptors. Since these system normally consist of one camera, the global feature descriptors are used to find the correct model from the database, while the local feature descriptors are used to estimate a transformation to the detected object. Using a multiple camera setup and being able to reconstruct a object model has some benefits compared to a single camera approach. The main advantages with a multiple camera setup is:

- Small object database compared to a ray-tracing database of objects. Only need one model stored in the database, instead of a large number of models that contain only a portion of the entire object.
- Global feature descriptors are not needed.
- Reduced computation time due to the two statements above.
- Possible to locate an object in an occluded scene.

One other aspect with a multiple camera setup is that it is possible to find the exact pose of a cylinder-like object, for example a tunnel thruster welding job. Using a single camera setup to detect the correct orientation of a pipe can be troublesome, since every ray-traced model will be almost identical. In a case like this there has to be taken compromises, for example always place the pipe in a specified pose so the camera can see the hole in the pipe. A multiple camera setup is able to detect the features of an object that can help to identify a correct orientation, regardless of which pose it initially has.

6.4 Features

Section 2.4 and 2.5 contained an introduction to keypoints and features. The choice of a keypoint detector was based on Filipe and Alex (2014), which vouched for the SIFT 3D detector. During the work on this thesis, the SIFT keypoint detector always gave good keypoint estimations and is recommended to use in other future work.

The feature descriptors on the other hand, was based on Hansch et al. (2014). This paper concluded that the FPFH and SHOT feature give the most reliable feature information. Both variants were tested in registration and object recognition. After some testing, it turned out that FPFH was best suited for registration, and SHOT was best suited for object recognition.

It was not allocated too much time during this thesis to test many parameters and different keypoint detectors and feature descriptors. Basing your choices of previous thorough work is a good approach for a large project work. It would be interesting to test other keypoint and feature descriptor combinations and compare the results.

Chapter 7: Conclusion

In this thesis, a solution was developed to perform object recognition from a multiple 3D camera setup. The thesis resulted in two different programs, one for calibrating the extrinsic parameters to reconstruct an object, and one to perform object recognition.

The experiments conducted shows that it is possible to reconstruct an object from multiple camera viewpoints, and detect the objects position and orientation using a object recognition pipeline based on local feature descriptors.

The results show some variation in the objects position and little variation in the objects orientation. Tests were made regarding the repeatability of the Kinect™, and it shows that the depth measurements can have a variation up to $2.3cm$ from one point cloud to another, if the camera is $2m$ from the measured point. The variation of the objects position in the experiments appear to originate from the poor repeatability of the sensor.

Even though the results have some variation, it is believed that the results are accurate enough for a grasping task performed by a robot. Another sensor with higher repeatability would improve the system and increase the accuracy.

7.1 Further work

The experiments conducted were based on a single object to detect. It is yet to be implemented more functionality to the object recognition system regarding multiple object detection. This can be achieved by implementing grouping of correspondences and adding functionality that matches the correct model to the right cluster. If this is achieved, it would also be interesting to see how the system would perform in an occluded scene.

The reconstructed scene and object in this thesis is not considered to be perfect, and testing another approach without updating the hardware should be studied. A promising approach could be to mount the Kinect™on the end-effector of the robot and perform a *scan* close to the object. This might remove most of the repeatability issues as occurred in this thesis, and thereby improve the entire system. If this approach is not sufficient, trying a new 3D sensor would be interesting.

Bibliography

Aldoma, A., Marton, Z., Tombari, F., Wohlkinger, W., Potthast, C., Zeisl, B., Rusu, R. B., Gedikli, S., and Vincze, M. (2012). Tutorial: Point cloud library: Three-dimensional object recognition and 6 dof pose estimation. *Robotics & Automation Magazine, IEEE*, 19(3):80–91.

Arun, K. S., Huang, T. S., and Blostein, S. D. (1987). Least-squares fitting of two 3-d point sets. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-9(5):698–700.

Berkmann, J. and Caelli, T. (1994). Computation of surface geometry and segmentation using covariance techniques. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 16(11):1114–1116.

Chen, Y. and Medioni, G. (1992). Object modeling by registration of multiple range images. *Image Vis. Comput.*, 10(3):145–155.

Corke, P. (2011). *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, volume 73 of *Fundamental Algorithms in MATLAB*. Springer Berlin Heidelberg, Berlin, Heidelberg, Berlin, Heidelberg.

Filipe, S. and Alex, L. A. (2014). A comparative evaluation of 3d keypoint detectors in a rgb-d object dataset.

Grzegorzek, M., Kolb, A., Koch, R., and Theobalt, C. (2013). *Time-of-Flight and Depth Imaging. Sensors, Algorithms, and Applications : Dagstuhl 2012 Seminar on Time-of-Flight Imaging and GCPR 2013 Workshop on Imaging New Modalities*, volume 8200 of *Time-of-flight and depth imaging*. Springer Berlin Heidelberg : Imprint: Springer.

Hansch, R., Weber, T., and Hellwich, O. (2014). Comparison of 3d interest point detectors and descriptors for point cloud fusion. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, II-3(September):57–64.

Harris, C. and Stephens, M. (1988). A combined corner and edge detector. In *In Proc. of Fourth Alvey Vision Conference*, pages 147–151.

Holz, D., Ichim, A. E., Tombari, F., Rusu, R. B., and Behnke, S. (2015). Registration with the point cloud library: A modular framework for aligning in 3-d. *Robotics & Automation Magazine, IEEE*, 22(4):110–124.

Khoshelham, K. and Elberink, S. O. (2012). Accuracy and resolution of kinect depth data for indoor mapping applications. *Sensors*, 12(2):1437.

Konstantinos G., D. (2010). Overview of the ransac algorithm.

Lowe, D. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110.

Martin A. Fischler, R. C. B. (1981). Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395.

Mitra, N. J., Nguyen, A., and Fortune, S. (2003). Estimating surface normals in noisy point cloud data. *SCG '03*, pages 322–328.

Nuchter, A., Lingemann, K., and Hertzberg, J. (2006). Extracting drivable surfaces in outdoor 6d slam. In *IN PROC. OF THE 37ND INT. SYMP. ON ROBOTICS (ISR '06*.

Paris, S. and Durand, F. (2006). A fast approximation of the bilateral filter using a signal processing approach.

PCL (2012). http://pointclouds.org/documentation/tutorials/basic_structures.php.

PCLoutlier. http://pointclouds.org/documentation/tutorials/statistical_outlier.php.

PCLvoxelgrid. http://pointclouds.org/documentation/tutorials/voxel_grid.php.

QtSignalSlot. http://doc.qt.io/qt-4.8/qt-designer-worldtimeclockplugin-example.html.

ROS. http://wiki.ros.org/ROS/Introduction.

ROSmessage. http://wiki.ros.org/Messages.

ROSrviz. http://wiki.ros.org/rviz.

ROSservice. http://wiki.ros.org/Services.

ROStopic. http://wiki.ros.org/Topics.

Rusinkiewicz, S. and Levoy, M. (2001). Efficient variants of the icp algorithm.

Rusu, R. B. (2009). *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*. PhD thesis, Computer Science department, Technische Universitaet Muenchen, Germany.

Rusu, R. B. (2010). Semantic 3d object maps for everyday manipulation in human living environments. *KI - Künstliche Intelligenz*, 24(4):345–348.

Rusu, R. B., Blodow, N., and Beetz, M. (2009). Fast point feature histograms (fpfh) for 3d registration. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 3212–3217.

Rusu, R. B., Blodow, N., Marton, Z. C., and Beetz, M. (2008). Aligning point cloud views using persistent feature histograms. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3384–3391.

Rusu, R. B., Bradski, G., Thibaux, R., and Hsu, J. (2010). Fast 3d recognition and pose using the viewpoint feature histogram. pages 2155–2162.

Rusu, R. B. and Cousins, S. (2011). 3d is here: Point cloud library (pcl). pages 1–4.

Sarbolandi, H., Lefloch, D., and Kolb, A. (2015). Kinect range sensing: Structured-light versus time-of-flight kinect. *Computer Vision and Image Understanding*.

Shakarji, C. (1998). Least-squares fitting algorithms of the nist algorithm testing system. *Journal of Research of the National Institute of Standards and Technology*, 103(6).

Shao, L., Han, J., Kohli, P., and Zhang, Z. (2014). *Computer Vision and Machine Learning with RGB-D Sensors*. Advances in Computer Vision and Pattern Recognition. Springer International Publishing, Cham, Cham.

Siciliano, B., Sciavicco, L., Villani, L., Oriolo, G., and SpringerLink (2009). *Robotics : Modelling, Planning and Control*. Springer London, London.

Smith, S. M. and Brady, J. M. (1997). Susan—a new approach to low level image processing. *International Journal of Computer Vision*, 23(1):45–78.

Steder, B., Bogdan, R., Kurt, R., and Burgard, K. W. (2010). Narf: 3d range image features for object recognition.

Tombari, F., Salti, S., and Di Stefano, L. (2010). *Unique Signatures of Histograms for Local Surface Description*, pages 356–369. Springer Berlin Heidelberg, Berlin, Heidelberg.

Tombari, F., Salti, S., and Stefano, L. D. (2011). A combined texture-shape descriptor for enhanced 3d feature matching. In *2011 18th IEEE International Conference on Image Processing*, pages 809–812.

Xiang, L., Echtler, F., Kerl, C., Wiedemeyer, T., Lars, hanyazou, Gordon, R., Facioni, F., laborer2008, Wareham, R., Goldhoorn, M., alberth, gaborpapp, Fuchs, S., jmtatsch, Blake, J., Federico, Jungkurth, H., Mingze, Y., vinouz, Coleman, D., Burns, B., Rawat, R., Mokhov, S., Reynolds, P., Viau, P., Fraissinet-Tachet, M., Ludique, Billingham, J., and Alistair (2016). libfreenect2: Release 0.2.

Zhong, Y. (2009). Intrinsic shape signatures: A shape descriptor for 3d object recognition.

Appendix A: Bundle XML file

XML bundle

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<multimarker markers="9">
    <marker index="0" status="1">
        <corner x="-9.5" y="-9.5" z="0.0" />
        <corner x="9.5" y="-9.5" z="0.0" />
        <corner x="9.5" y="9.5" z="0.0" />
        <corner x="-9.5" y="9.5" z="0.0" />
    </marker>

    <marker index="1" status="1">
        <corner x="-9.5" y="52.1" z="0.0" />
        <corner x="9.5" y="52.1" z="0.0" />
        <corner x="9.5" y="71.1" z="0.0" />
        <corner x="-9.5" y="71.1" z="0.0" />
    </marker>
    <marker index="2" status="1">
        <corner x="-9.5" y="-70.8" z="0.0" />
        <corner x="9.5" y="-70.8" z="0.0" />
        <corner x="9.5" y="-51.8" z="0.0" />
        <corner x="-9.5" y="-51.8" z="0.0" />
    </marker>

    <marker index="3" status="1">
        <corner x="-68.1" y="-70.8" z="0.0" />
        <corner x="-49.1" y="-70.8" z="0.0" />
        <corner x="-49.1" y="-51.8" z="0.0" />
        <corner x="-68.1" y="-51.8" z="0.0" />
    </marker>
    <marker index="4" status="1">
        <corner x="-68.0" y="52.1" z="0.0" />
        <corner x="-49.0" y="52.1" z="0.0" />
        <corner x="-49.0" y="71.1" z="0.0" />
        <corner x="-68.0" y="71.1" z="0.0" />
    </marker>

    <marker index="5" status="1">
        <corner x="-38.8" y="-70.8" z="0.0" />
```

```
        <corner x="-19.8" y="-70.8" z="0.0" />
        <corner x="-19.8" y="-51.8" z="0.0" />
        <corner x="-38.8" y="-51.8" z="0.0" />
    </marker>
    <marker index="6" status="1">
        <corner x="-38.8" y="-9.5" z="0.0" />
        <corner x="-19.8" y="-9.5" z="0.0" />
        <corner x="-19.8" y="9.5" z="0.0" />
        <corner x="-38.8" y="9.5" z="0.0" />
    </marker>
    <marker index="7" status="1">
        <corner x="-38.75" y="52.1" z="0.0" />
        <corner x="-19.75" y="52.1" z="0.0" />
        <corner x="-19.75" y="71.1" z="0.0" />
        <corner x="-38.75" y="71.1" z="0.0" />
    </marker>
    <marker index="8" status="1">
        <corner x="-68.1" y="-9.5" z="0.0" />
        <corner x="-49.1" y="-9.5" z="0.0" />
        <corner x="-49.1" y="9.5" z="0.0" />
        <corner x="-68.1" y="9.5" z="0.0" />
    </marker>


</multimarker>
```

Appendix B: Camera calibration

calibCell.hpp

```
#include <ros/ros.h>
#include <string>
#include <sensor_msgs/PointCloud2.h>
#include <ar_track_alvar_msgs/AlvarMarkers.h>
#include <ar_track_alvar_msgs/AlvarMarker.h>
#include <geometry_msgs/PoseStamped.h>
#include <tf/tf.h>
#include <eigen3/Eigen/Core>
#include <iostream>
#include <fstream>

/*!
 * \brief The CalibrateCell class averages a number of pose messages
 * from ar_track_alvar in ROS to find the extrinsic parameters from the
 * camera to an AR tag.
 */
class CalibrateCell{
public:
    /*!
     * \brief Constructor for CalibrateCell class.
     * \param argc Initialization argument
     * \param argv Initialization argument
     */
    CalibrateCell(int argc, char** argv);

    /*!
     * \brief Deconstructor for CalibrateCell class.
     */
    virtual ~CalibrateCell();

    /*!
     * \brief Initialization method for the ROS node.
     * \return Returns true if initialization is OK. Returns false if
     * initialization fails.
     */
    bool init();

    /*!
     * \brief Starts the ROS node and loops.
     */
    void run();

    /*!
     * \brief Callback method from ar_track_alvar
     * \param cam_pos position of the AR tag relative to the camera
     */
```

```
47      void camPosCallback1(const ar_track_alvar_msgs::AlvarMarkersConstPtr&
            cam_pos);
48
49  private:
50      int init_argc; //!< Initialization arguments
51      char** init_argv; //!< Initialization arguments
52      ros::Subscriber subPose; //!< ROS subscriber to the pose message
53      int nrOfMsgs; //!< Number of messages to average
54      int msgs; //!< Current messages averaged
55      std::vector<tf::Vector3> positions; //!< Vector containing the pose
            position (X,Y,Z)
56      std::vector<tf::Quaternion> quaternions; //!< Vector containing the pose
            orientation in quaternions
57      std::string cameraName; //!< Name of the camera
58      std::vector<Eigen::Matrix4f> transformationMatrices; //!< Vector containing
             the pose in transformation matrices
59
60      /*!
61       * \brief Converts a position and quaternion vector to a 4x4 transformation
             matrix.
62       * \param cameraID the name of the camera
63       * \param position the pose position vector
64       * \param quaternion the pose orientation vector in quaternions
65       * \return
66       */
67      Eigen::Matrix4f calcTransformationMatrix(std::string cameraID, tf::Vector3
            position, tf::Quaternion quaternion);
68
69      /*!
70       * \brief Adds a new transformation (position and orientation) to the
             private vectors positions and quaternions.
71       * \param cameraID the name of the camera
72       * \param position the pose position vector
73       * \param quaternion the pose orientation vector in quaternions
74       */
75      void addNewTransformation(std::string cameraID, tf::Vector3 position, tf::
            Quaternion quaternion);
76
77      /*!
78       * \brief Takes the average of the vector of position and orientation and
             writes the
79       * result as a 4x4 transformation matrix to a .txt file.
80       */
81      void averageRotations();
82  };
```

calibCell.cpp

```
#include "../include/calibCell.hpp"
#include <iostream>

CalibrateCell::CalibrateCell(int argc, char **argv):
    init_argc(argc),
    init_argv(argv)
{}

CalibrateCell::~CalibrateCell()
{
    if(ros::isStarted()) {
      ros::shutdown();
      ros::waitForShutdown();
    }
}

bool CalibrateCell::init()
{
    std::cout << "Initing" << std::endl;
    nrOfMsgs = 100;
    msgs=0;
    tf::Quaternion q(0, 0, 0, 0);
    tf::Vector3 v(0,0,0);
    positions.push_back(v);
    quaternions.push_back(q);
    cameraName = "";

    ros::init(init_argc,init_argv,"calib_cell");
    if ( ! ros::master::check() ) {
        return false;
    }
    ros::start();

    ros::NodeHandle n;
    subPose = n.subscribe<ar_track_alvar_msgs::AlvarMarkers, CalibrateCell>("/
        ar_pose_marker", 10, &CalibrateCell::camPosCallback1, this);

    return true;
}

void CalibrateCell::run()
{
    std::cout << "Running" << std::endl;
    ros::Rate loop_rate(200);
    while ( ros::ok() ){
        ros::spinOnce();
        loop_rate.sleep();
    }
}

void CalibrateCell::camPosCallback1(const ar_track_alvar_msgs::AlvarMarkers
    ConstPtr &cam_pos)
{
    if(cam_pos->markers.size() == 0){
        return;
```

```
54      }
55      // Camera ID
56      std::string cameraID = cam_pos->markers.at(0).header.frame_id;
57      // Position XYZ
58      geometry_msgs::Point positionMsg = cam_pos->markers.at(0).pose.pose.
            position;
59      tf::Vector3 position(positionMsg.x, positionMsg.y, positionMsg.z);
60      // Rotation
61      geometry_msgs::Quaternion quatMsg = cam_pos->markers.at(0).pose.pose.
            orientation;
62      tf::Quaternion q(quatMsg.x, quatMsg.y, quatMsg.z, quatMsg.w);
63
64      if(msgs == nrOfMsgs){
65          std::cout << "Im done" << std::endl;
66          subPose.shutdown();
67          averageRotations();
68          ros::shutdown();
69
70      }
71      addNewTransformation(cameraID,position,q);
72  }
73
74  Eigen::Matrix4f CalibrateCell::calcTransformationMatrix(std::string cameraID,
        tf::Vector3 position, tf::Quaternion quaternion)
75  {
76      tf::Matrix3x3 rotation(quaternion);
77      Eigen::Matrix4f transTmp;
78      transTmp <<     rotation.getColumn(0).getX(), rotation.getColumn(1).getX(),
            rotation.getColumn(2).getX(),  position.getX(),
79                      rotation.getColumn(0).getY(), rotation.getColumn(1).getY(),
                            rotation.getColumn(2).getY(),  position.getY(),
80                      rotation.getColumn(0).getZ(), rotation.getColumn(1).getZ(),
                            rotation.getColumn(2).getZ(),  position.getZ(),
81                      0,          0,          0,       1;
82
83      double roll, pitch, yaw;
84      std::cout << "Rotation RPY: " << std::endl;
85      rotation.getRPY(roll, pitch, yaw);
86      std::cout << roll << std::endl;
87      std::cout << pitch << std::endl;
88      std::cout << yaw << std::endl;
89
90      return transTmp;
91  }
92
93  void CalibrateCell::addNewTransformation(std::string cameraID, tf::Vector3
        position, tf::Quaternion quaternion)
94  {
95      if( msgs < nrOfMsgs){
96          std::cout << "Added from: ";
97          std::cout << cameraID << std::endl;
98          std::cout << "Number: ";
99          std::cout << msgs << std::endl;
100         positions[0] += position;
101         quaternions[0] += quaternion;
102         msgs++;
103         calcTransformationMatrix(cameraID, position, quaternion);
```

```
104             cameraName = cameraID;
105         }
106     }
107
108     void CalibrateCell::averageRotations()
109     {
110         std::vector<float> x(quaternions.size());
111         std::vector<float> y(quaternions.size());
112         std::vector<float> z(quaternions.size());
113         std::vector<float> w(quaternions.size());
114
115         std::vector<float> xPos(quaternions.size());
116         std::vector<float> yPos(quaternions.size());
117         std::vector<float> zPos(quaternions.size());
118
119         float div = 1.0f/(float)nrOfMsgs;
120
121         for(int i=0; i<quaternions.size(); i++){
122             x[i] = quaternions.at(i).getX()*div;
123             y[i] = quaternions.at(i).getY()*div;
124             z[i] = quaternions.at(i).getZ()*div;
125             w[i] = quaternions.at(i).getW()*div;
126
127             xPos[i] = positions.at(i).getX()*div;
128             yPos[i] = positions.at(i).getY()*div;
129             zPos[i] = positions.at(i).getZ()*div;
130         }
131
132         for(int k=0; k<quaternions.size(); k++){
133             tf::Quaternion quat(x.at(k), y.at(k), z.at(k), w.at(k));
134             quat.normalize();
135             tf::Vector3 position(xPos.at(k), yPos.at(k), zPos.at(k));
136             Eigen::Matrix4f tmp = calcTransformationMatrix(cameraName,position,
                    quat);
137             transformationMatrices.push_back(tmp);
138         }
139
140         char chars[] = "/";
141
142         for (unsigned int i = 0; i < strlen(chars); ++i)
143         {
144             cameraName.erase (std::remove(cameraName.begin(), cameraName.end(),
                    chars[i]), cameraName.end());
145         }
146
147         std::string tmp = "/home/minions/";
148         tmp.append(cameraName);
149         tmp.append(".txt");
150         std::ofstream file(tmp.c_str());
151
152         if (file.is_open())
153         {
154             for(int i=0; i<transformationMatrices.size(); i++){
155                 file << "Matrix for: " << cameraName << '\n';
156                 file << transformationMatrices.at(i) << '\n' << '\n';
157             }
158
```

```
159     }
160     std::cout << "Done calibrating, wrote matrix to file: ";
161     std::cout << tmp << std::endl;
162 
163 }
164 
165 int main(int argc, char **argv){
166     CalibrateCell cell(argc, argv);
167     cell.init();
168     cell.run();
169 }
```

Appendix C: Main program

main.cpp

```
1  #include <QtGui>
2  #include <QApplication>
3  #include "../include/qt_master/main_window.hpp"
4
5  /*!
6   * \brief Main method for the program. Starts the application.
7   * \param argc Arguments.
8   * \param argv Arguments.
9   * \return Returns 1 if shut down correct, 0 otherwise.
10  */
11 int main(int argc, char **argv) {
12     QApplication app(argc, argv);
13     qt_master::MainWindow w(argc,argv);
14     w.show();
15     app.connect(&app, SIGNAL(lastWindowClosed()), &app, SLOT(quit()));
16     int result = app.exec();
17
18         return result;
19 }
```

qnode.hpp

```
#ifndef qt_master_QNODE_HPP_
#define qt_master_QNODE_HPP_

#include <ros/ros.h>
#include <string>
#include <QThread>
#include <QStringListModel>
#include <sensor_msgs/PointCloud2.h>
#include <pcl_conversions/pcl_conversions.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <agilus_planner/Pose.h>
#include <kuka_rsi_hw_interface/write_8_outputs.h>
#include <math.h>

namespace qt_master {
/*!
 * \brief The QNode class is a ROS node that deals with the communication
 * between the robots, camera outputs etc.
 */
class QNode : public QThread {
    Q_OBJECT
public:
    /*!
     * \brief Constructor for QNode class.
     * \param argc Initialization argument
     * \param argv Initialization argument
     */
        QNode(int argc, char** argv );

    /*!
     * \brief Deconstructor for QNode class.
     */
        virtual ~QNode();

    /*!
     * \brief Initialization method for the ROS node.
     * \return Returns true if initialization is OK. Returns false if
     * initialization fails.
     */
    bool init();

    /*!
     * \brief Starts the ROS node and loops.
     */
        void run();

    /*!
     * \brief Callback method for point cloud from camera 1.
     * \param cloud_msg point cloud message from camera 1.
     */
    void cloudCallback1(const sensor_msgs::PointCloud2ConstPtr& cloud_msg);

    /*!
     * \brief Callback method for point cloud from camera 2.
```

```
56          * \param cloud_msg point cloud message from camera 2.
57          */
58         void cloudCallback2(const sensor_msgs::PointCloud2ConstPtr& cloud_msg);
59
60         /*!
61          * \brief Callback method for point cloud from camera 3.
62          * \param cloud_msg point cloud message from camera 3.
63          */
64         void cloudCallback3(const sensor_msgs::PointCloud2ConstPtr& cloud_msg);
65
66         /*!
67          * \brief Asks the ROS master for every available point cloud topic name.
68          * \return Returns a QStringList of available point cloud topics.
69          */
70         QStringList getTopics();
71
72         /*!
73          * \brief Sets the parameters to the pose object.
74          * \param x position in x (meters)
75          * \param y position in y (meters)
76          * \param z position in z (meters)
77          * \param roll orientation in roll (radians)
78          * \param pitch orientation in pitch (radians)
79          * \param yaw orientation in yaw (radians)
80          */
81         void setPose(double x, double y, double z, double roll, double pitch,
               double yaw);
82
83         /*!
84          * \brief Calls the service for planning a pose with given parameters.
85          * \param x position in x (meters)
86          * \param y position in y (meters)
87          * \param z position in z (meters)
88          * \param roll orientation in roll (radians)
89          * \param pitch orientation in pitch (radians)
90          * \param yaw orientation in yaw (radians)
91          * \param robot selected robot, 0 is "Agilus 1" and 1 is "Agilus 2"
92          */
93         void planPose(double x, double y, double z, double roll, double pitch,
               double yaw, int robot);
94
95         /*!
96          * \brief Calls the service for moving to a pose with given parameters.
97          * \param x position in x (meters)
98          * \param y position in y (meters)
99          * \param z position in z (meters)
100         * \param roll orientation in roll (radians)
101         * \param pitch orientation in pitch (radians)
102         * \param yaw orientation in yaw (radians)
103         * \param robot selected robot, 0 is "Agilus 1" and 1 is "Agilus 2"
104         */
105        void movePose(double x, double y, double z, double roll, double pitch,
               double yaw, int robot);
106
107        /*!
108         * \brief Opens the gripper at the end effector of the selected robot.
109         * \param robot selected robot, 0 is "Agilus 1" and 1 is "Agilus 2"
```

```
110         */
111         void openGripper(int robot);
112 
113         /*!
114          * \brief Closes the gripper at the end effector of the selected robot.
115          * \param robot selected robot, 0 is "Agilus 1" and 1 is "Agilus 2"
116          */
117         void closeGripper(int robot);
118 
119         /*!
120          * \brief Subscribes to all 3 point cloud topics (camera1, camera2 and
                camera3).
121          */
122         void subscribe3Clouds();
123 
124 
125     Q_SIGNALS:
126         /*!
127          * \brief Shuts down the ROS node in the correct manner.
128          */
129         void rosShutdown();
130 
131         /*!
132          * \brief QtSignal for sending 3 aquired point clouds to another class (GUI
                class)
133          * \param clouds a list of 3 point clouds
134          */
135         void send3Clouds(std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr> clouds
                );
136 
137     public Q_SLOTS:
138 
139     private:
140         int init_argc; //!< Initialization arguments
141         char** init_argv; //!< Initialization arguments
142         ros::Subscriber pointCloudSub1; //!< ROS subscriber to the point cloud
                message from camera 1
143         ros::Subscriber pointCloudSub2; //!< ROS subscriber to the point cloud
                message from camera 2
144         ros::Subscriber pointCloudSub3; //!< ROS subscriber to the point cloud
                message from camera 3
145         bool gotCloud1; //!< Flag to check if a point cloud from camera 1 is
                recieved
146         bool gotCloud2; //!< Flag to check if a point cloud from camera 2 is
                recieved
147         bool gotCloud3; //!< Flag to check if a point cloud from camera 3 is
                recieved
148         pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud1; //!< Point cloud for camera
                1
149         pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud2; //!< Point cloud for camera
                2
150         pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud3; //!< Point cloud for camera
                3
151         agilus_planner::Pose pose; //!< Pose object for the robot, contains xyz and
                 orientation in rpy
152         ros::ServiceClient planAg1; //!< ROS service for planning motion for Agilus
                 1
```

```
153     ros::ServiceClient moveAg1; //!< ROS service for moving motion for Agilus 1
154     ros::ServiceClient planAg2; //!< ROS service for planning motion for Agilus
            2
155     ros::ServiceClient moveAg2; //!< ROS service for moving motion for Agilus 2
156     kuka_rsi_hw_interface::write_8_outputs gripperState; //!< Digital Output
            object to open/close gripper
157     ros::ServiceClient gripperAg1; //!< ROS service for gripper handling for
            Agilus 1
158     ros::ServiceClient gripperAg2; //!< ROS service for gripper handling for
            Agilus 2
159 };
160
161 }
162 #endif
```

qnode.cpp

```
#include <ros/ros.h>
#include <ros/network.h>
#include <string>
#include <std_msgs/String.h>
#include <sstream>
#include "../include/qt_master/qnode.hpp"

namespace qt_master {

QNode::QNode(int argc, char** argv ) :
    init_argc(argc),
    init_argv(argv),
    cloud1(new pcl::PointCloud<pcl::PointXYZRGB>()),
    cloud2(new pcl::PointCloud<pcl::PointXYZRGB>()),
    cloud3(new pcl::PointCloud<pcl::PointXYZRGB>())
{
}

QNode::~QNode() {
    if(ros::isStarted()) {
        ros::shutdown();
        ros::waitForShutdown();
    }
    wait();
}

bool QNode::init() {
    ros::init(init_argc,init_argv,"qt_master");
    if ( ! ros::master::check() ) {
        return false;
    }
    ros::start();
    ros::NodeHandle n;
    planAg1 = n.serviceClient<agilus_planner::Pose>("/robot_service_ag1/
        plan_pose");
    moveAg1 = n.serviceClient<agilus_planner::Pose>("/robot_service_ag1/
        go_to_pose");
    planAg2 = n.serviceClient<agilus_planner::Pose>("/robot_service_ag2/
        plan_pose");
    moveAg2 = n.serviceClient<agilus_planner::Pose>("/robot_service_ag2/
        go_to_pose");
    gripperAg1 = n.serviceClient<kuka_rsi_hw_interface::write_8_outputs>("/ag1/
        kuka_hardware_interface/write_8_digital_outputs");
    gripperAg2 = n.serviceClient<kuka_rsi_hw_interface::write_8_outputs>("/ag2/
        kuka_hardware_interface/write_8_digital_outputs");
    gotCloud1 = false;
    gotCloud2 = false;
    gotCloud3 = false;
    start();
    return true;
}

void QNode::run() {
    ros::Rate loop_rate(10);
    int count = 0;
```

```
50      while ( ros::ok() ) {
51          if(gotCloud1 && gotCloud2 && gotCloud3){
52              std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr> clouds;
53              clouds.push_back(cloud1);
54              clouds.push_back(cloud2);
55              clouds.push_back(cloud3);
56              Q_EMIT send3Clouds(clouds);
57              pointCloudSub1.shutdown();
58              pointCloudSub2.shutdown();
59              pointCloudSub3.shutdown();
60              std::cout << "Done taking pictures" << std::endl;
61              gotCloud1 = false;
62              gotCloud2 = false;
63              gotCloud3 = false;
64          }
65          ros::spinOnce();
66          loop_rate.sleep();
67          ++count;
68      }
69      std::cout << "Ros shutdown, proceeding to close the gui." << std::endl;
70      Q_EMIT rosShutdown(); // used to signal the gui for a shutdown (useful to
            ros launch)
71  }
72
73  QStringList QNode::getTopics()
74  {
75      QStringList list;
76      QString tmp;
77      ros::master::V_TopicInfo master_topics;
78      ros::master::getTopics(master_topics);
79      for (ros::master::V_TopicInfo::iterator it = master_topics.begin() ; it !=
            master_topics.end(); it++) {
80          const ros::master::TopicInfo& info = *it;
81          tmp = QString::fromUtf8(info.datatype.c_str());
82          if(QString::compare(tmp, "sensor_msgs/PointCloud2", Qt::CaseInsensitive
                ) == 0){
83              list.append(QString::fromUtf8(info.name.c_str()));
84          }
85      }
86      return list;
87  }
88
89  void QNode::setPose(double x, double y, double z, double roll, double pitch,
        double yaw)
90  {
91      pose.request.header.frame_id = "/world";
92      pose.request.set_position = true;
93      pose.request.set_orientation = true;
94      pose.request.position_x = x;
95      pose.request.position_y = y;
96      pose.request.position_z = z;
97      pose.request.orientation_r = roll*M_PI/180.0;
98      pose.request.orientation_p = pitch*M_PI/180.0;
99      pose.request.orientation_y = yaw*M_PI/180.0;
100 }
101
102 void QNode::planPose(double x, double y, double z, double roll, double pitch,
```

```
    double yaw, int robot)
{
    setPose(x,y,z,roll,pitch,yaw);
    if(robot == 0){
        //Agilus1
        planAg1.call(pose);
    }
    else if(robot == 1){
        //Agilus2
        planAg2.call(pose);
    }
}

void QNode::movePose(double x, double y, double z, double roll, double pitch,
    double yaw, int robot)
{
    setPose(x,y,z,roll,pitch,yaw);
    if(robot == 0){
        //Agilus1
        moveAg1.call(pose);
    }
    else if(robot == 1){
        //Agilus2
        moveAg2.call(pose);
    }
}

void QNode::openGripper(int robot)
{
    gripperState.request.out1 = false;
    gripperState.request.out4 = true;
    gripperState.request.out2 = true;
    if(robot == 0){
        gripperAg1.call(gripperState);
    }
    else if(robot == 1){
        gripperAg2.call(gripperState);
    }
}

void QNode::closeGripper(int robot)
{
    gripperState.request.out1 = true;
    gripperState.request.out4 = false;
    gripperState.request.out2 = true;
    if(robot == 0){
        gripperAg1.call(gripperState);
    }
    else if(robot == 1){
        gripperAg2.call(gripperState);
    }
}

void QNode::subscribe3Clouds()
{
    ros::NodeHandle n;
    pointCloudSub1 = n.subscribe<sensor_msgs::PointCloud2, QNode>("/NUC1/sd/
```

```
        points", 10, &QNode::cloudCallback1, this);
158     pointCloudSub2 = n.subscribe<sensor_msgs::PointCloud2, QNode>("/NUC2/sd/
            points", 10, &QNode::cloudCallback2, this);
159     pointCloudSub3 = n.subscribe<sensor_msgs::PointCloud2, QNode>("/PC/sd/
            points", 10, &QNode::cloudCallback3, this);
160 }
161
162 void QNode::cloudCallback1(const sensor_msgs::PointCloud2ConstPtr &cloud_msg){
163     pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmp(new pcl::PointCloud<pcl::
            PointXYZRGB>());
164     pcl::fromROSMsg(*cloud_msg, *tmp);
165     pcl::copyPointCloud(*tmp, *cloud1);
166     gotCloud1 = true;
167 }
168
169 void QNode::cloudCallback2(const sensor_msgs::PointCloud2ConstPtr &cloud_msg){
170     pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmp(new pcl::PointCloud<pcl::
            PointXYZRGB>());
171     pcl::fromROSMsg(*cloud_msg, *tmp);
172     pcl::copyPointCloud(*tmp, *cloud2);
173     gotCloud2 = true;
174 }
175
176 void QNode::cloudCallback3(const sensor_msgs::PointCloud2ConstPtr &cloud_msg){
177     pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmp(new pcl::PointCloud<pcl::
            PointXYZRGB>());
178     pcl::fromROSMsg(*cloud_msg, *tmp);
179     pcl::copyPointCloud(*tmp, *cloud3);
180     gotCloud3 = true;
181 }
182
183 }
```

PointCloudManipulator.hpp

```
#ifndef qt_master_POINT_CLOUD_MANIPULATOR_H
#define qt_master_POINT_CLOUD_MANIPULATOR_H

#include <QStringList>
#include <pcl/visualization/pcl_visualizer.h>
#include <pcl/visualization/histogram_visualizer.h>
#include <pcl/filters/passthrough.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/filters/median_filter.h>
#include <pcl/filters/statistical_outlier_removal.h>
#include <pcl/filters/fast_bilateral.h>
#include <pcl/filters/shadowpoints.h>
#include <pcl/features/normal_3d.h>
#include <pcl/features/integral_image_normal.h>
#include <pcl/common/transforms.h>
#include <pcl/keypoints/iss_3d.h>
#include <pcl/keypoints/narf_keypoint.h>
#include <pcl/range_image/range_image_planar.h>
#include <pcl/features/range_image_border_extractor.h>
#include <pcl/visualization/range_image_visualizer.h>
#include <pcl/features/narf_descriptor.h>
#include "pcl/point_types.h"
#include "pcl/point_cloud.h"
#include "pcl/io/pcd_io.h"
#include "pcl/kdtree/kdtree_flann.h"
#include "pcl/features/normal_3d.h"
#include "pcl/features/normal_3d_omp.h"
#include "pcl/features/pfh.h"
#include "pcl/features/fpfh.h"
#include "pcl/features/fpfh_omp.h"
#include "pcl/keypoints/sift_keypoint.h"
#include "pcl/keypoints/iss_3d.h"
#include <pcl/registration/transforms.h>
#include <pcl/registration/ia_ransac.h>
#include <pcl/registration/correspondence_estimation.h>
#include <pcl/registration/correspondence_estimation_normal_shooting.h>
#include <pcl/registration/correspondence_rejection.h>
#include <pcl/registration/correspondence_rejection_distance.h>
#include <pcl/registration/correspondence_rejection_sample_consensus.h>
#include <pcl/registration/correspondence_rejection_one_to_one.h>
#include <pcl/registration/correspondence_rejection_organized_boundary.h>
#include <pcl/registration/correspondence_rejection_median_distance.h>
#include <pcl/features/shot.h>
#include <pcl/features/shot_omp.h>
#include <pcl/registration/transformation_estimation_svd.h>
#include <pcl/registration/transformation_estimation_lm.h>
#include <pcl/registration/transformation_estimation_point_to_plane_lls.h>
#include <pcl/registration/transformation_estimation_point_to_plane_weighted.h>
#include <pcl/registration/transformation_estimation_point_to_plane.h>
#include <pcl/registration/icp.h>
#include <pcl/registration/icp_nl.h>
#include <pcl/registration/lum.h>
#include <pcl/registration/elch.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/filters/extract_indices.h>
```

```
56 #include <pcl/segmentation/extract_clusters.h>
57 #include <pcl/surface/vtk_smoothing/vtk_utils.h>
58 #include <pcl/io/ply_io.h>
59 #include <pcl/io/vtk_lib_io.h>
60 #include <pcl/recognition/cg/hough_3d.h>
61 #include <pcl/recognition/cg/geometric_consistency.h>
62
63 namespace qt_master {
64
65 /*!
66  * \brief The PointCloudManipulator class contains a number of filtering
        methods,
67  * algorithms and matching methods.
68  */
69 class PointCloudManipulator : public QObject
70 {
71     Q_OBJECT
72 public:
73     /*!
74      * \brief Constructor for PointCloudManipulator class.
75      * \param parent ui parent
76      */
77     explicit PointCloudManipulator(QObject *parent = 0);
78
79     /*!
80      * \brief Deconstructor for PointCloudManipulator class.
81      */
82     ~PointCloudManipulator();
83
84     /*!
85      * \brief Aquires names the available filters in the class.
86      * \return a QStringList of available filters
87      */
88     QStringList getFilters();
89
90     /*!
91      * \brief Runs the selected filter on a point cloud.
92      * \param selectedFilter the selected filter
93      * \param inCloud input point cloud object
94      * \param outCloud output point cloud object
95      * \param d1 parameter 1
96      * \param d2 parameter 2
97      * \param d3 paramter 3
98      * \param xyz parameter for passthrough filter
99      */
100    void runFilter(int selectedFilter,pcl::PointCloud<pcl::PointXYZRGB>::Ptr
           inCloud,pcl::PointCloud<pcl::PointXYZRGB>::Ptr outCloud, double d1,
           double d2, double d3, QString xyz);
101
102    /*!
103     * \brief Creates the required indexes for the selected filter, and
104     * sends them to the GUI. (label text, scaling etc)
105     * \param selectedFilter the selected filter
106     */
107    void getNewIndexInfo(int selectedFilter);
108
109    /*!
```

```
110         * \brief Implementation of the PassThrough filter for a point cloud.
111         * \param inCloud input point cloud object
112         * \param outCloud output point cloud object
113         * \param limitMin minimum limit (in meters)
114         * \param limitMax maximum limit (in meters)
115         * \param field the axis to be cut (x, y or z)
116         */
117        void filterPassThrough(pcl::PointCloud<pcl::PointXYZRGB>::Ptr inCloud, pcl
               ::PointCloud<pcl::PointXYZRGB>::Ptr outCloud, double limitMin, double
               limitMax, QString field);
118
119        /*!
120         * \brief Implementation of the VoxelGrid filter for a point cloud.
121         * \param inCloud input point cloud object
122         * \param outCloud output point cloud object
123         * \param leafSize the specified leaf size (volume) for the filter
124         */
125        void filterVoxelGrid(pcl::PointCloud<pcl::PointXYZRGB>::Ptr inCloud, pcl::
               PointCloud<pcl::PointXYZRGB>::Ptr outCloud, double leafSize);
126
127        /*!
128         * \brief Implementation of the median filter for a point cloud.
129         * \param inCloud input point cloud object
130         * \param outCloud output point cloud object
131         * \param windowSize window size of the median
132         * \param maxMovement maximum allowed movement for the median
133         */
134        void filterMedian(pcl::PointCloud<pcl::PointXYZRGB>::Ptr inCloud, pcl::
               PointCloud<pcl::PointXYZRGB>::Ptr outCloud, double windowSize, double
               maxMovement);
135
136        /*!
137         * \brief Implementation of normal estimation for a point cloud.
138         * \param inCloud input point cloud object
139         * \param radius radius of search in normal estimation
140         * \param nrToDisplay nr. of normals to return (for example 1/10)
141         */
142        void filterNormal(pcl::PointCloud<pcl::PointXYZRGB>::Ptr inCloud, double
               radius, double nrToDisplay);
143
144        /*!
145         * \brief Implementation of the bilateral filter for a point cloud.
146         * \param inCloud input point cloud object
147         * \param sigmaS sigma S value in the filter
148         * \param sigmaR sigma R value in the filter
149         */
150        void filterBilateral(pcl::PointCloud<pcl::PointXYZRGB>::Ptr inCloud, double
                sigmaS, float sigmaR);
151
152        /*!
153         * \brief Transforms a point cloud with given parameters.
154         * \param inCloud input point cloud object to transform
155         * \param rX rotation in x
156         * \param rY rotation in y
157         * \param rZ rotation in z
158         * \param tX translation in x
159         * \param tY translation in y
```

```
160         * \param tZ translation in z
161         */
162        void translateCloud(pcl::PointCloud<pcl::PointXYZRGB>::Ptr inCloud, double
               rX, double rY, double rZ, double tX, double tY, double tZ);
163
164        /*!
165         * \brief Resets a visualizer and send it to the GUI.
166         * \param selectedFilter selected filter to use
167         */
168        void getNewVisualizer(int selectedFilter);
169
170        /*!
171         * \brief Get the last used filter.
172         * \return Returns a QString name of the last filter used
173         */
174        QString getLastFiltered();
175
176        /*!
177         * \brief Alignes point clouds using registration. Visualizes the process.
178         * \param fileNames filenames of saved point clouds
179         */
180        void alignClouds(QStringList fileNames);
181
182        /*!
183         * \brief Alignes the robot cell at NTNU IPK using registration. Visualizes
                the process.
184         * \param fileNames filenames of saved point clouds
185         */
186        void alignRobotCell(QStringList fileNames);
187
188        /*!
189         * \brief Refines the alignment of the robot cell at NTNU IPK using
                Iterative Closest Point.
190         * \param fileNames filenames of saved point clouds
191         */
192        void refineAlignment(QStringList fileNames);
193
194        /*!
195         * \brief Implementation of the VoxelGrid filter for a point cloud.
196         * \param inCloud input point cloud object
197         * \param leafSize the specified leaf size (volume) for the filter
198         * \return Returns a point cloud object that has been filtered
199         */
200        pcl::PointCloud<pcl::PointXYZRGB>::Ptr filterVoxel(pcl::PointCloud<pcl::
               PointXYZRGB>::Ptr inCloud, double leafSize);
201
202        /*!
203         * \brief Implementation of the PassThrough filter for a point cloud.
204         * \param inCloud input point cloud object
205         * \param outCloud output point cloud object
206         * \param limitMin minimum limit (in meters)
207         * \param limitMax maximum limit (in meters)
208         * \param field the axis to be cut (x, y or z)
209         * \return Returns a point cloud object that has been filtered
210         */
211        pcl::PointCloud<pcl::PointXYZRGB>::Ptr filterPassThrough(pcl::PointCloud<
               pcl::PointXYZRGB>::Ptr inCloud, double limitMin, double limitMax,
```

```
        QString field);
212
213     /*!
214      * \brief Implementation of the ShadowPoint removal filter for a point
           cloud.
215      * \param inCloud input point cloud
216      * \param normals normals of the point cloud
217      * \param threshold threshold to remove points
218      * \return Returns a point cloud object that has been filtered
219      */
220     pcl::PointCloud<pcl::PointXYZRGB>::Ptr filterShadowPoint(pcl::PointCloud<
        pcl::PointXYZRGB>::Ptr inCloud,pcl::PointCloud<pcl::Normal>::Ptr
        normals, double threshold);
221
222     /*!
223      * \brief Implementation of the Outlier removal filter for a point cloud.
224      * \param inCloud input point cloud
225      * \return Returns a point cloud object that has been filtered
226      */
227     pcl::PointCloud<pcl::PointXYZRGB>::Ptr filterOutlier(pcl::PointCloud<pcl::
        PointXYZRGB>::Ptr inCloud);
228
229     /*!
230      * \brief Implementation of normal estimation for a point cloud.
231      * \param inCloud input point cloud object
232      * \param radius radius of search in normal estimation
233      * \param nrToDisplay nr. of normals to return (for example 1/10)
234      * \return Returns a point cloud normal object containing the normals
235      */
236     pcl::PointCloud<pcl::Normal>::Ptr computeSurfaceNormals(pcl::PointCloud<pcl
        ::PointXYZRGB>::Ptr input, float radius);
237
238     /*!
239      * \brief Implementation of surface point normal estimation for a point
           cloud
240      * \param input input point cloud object
241      * \param surface surface of point cloud object
242      * \param radius radius of search in normal estimation
243      * \return Returns a point normal object containing the normals
244      */
245     pcl::PointCloud<pcl::PointNormal>::Ptr computeSurfacePointNormals(pcl::
        PointCloud<pcl::PointXYZRGB>::Ptr input,pcl::PointCloud<pcl::PointXYZ
        RGB>::Ptr surface, float radius);
246
247     /*!
248      * \brief Implementation of cluster extraction.
249      * \param input input point cloud object
250      * \param distance maximum distance allowed to be outside a cluster
251      * \return Returns a list of point cloud clusters that are found
252      */
253     std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr> extractClusters (pcl::
        PointCloud<pcl::PointXYZRGB>::Ptr input, double distance);
254
255     /*!
256      * \brief Implementation of SIFT keypoint detector.
257      * \param points input point cloud object
258      * \param minScale minimum scale in SIFT
```

```
259         * \param nrOctaves number of octaves in SIFT
260         * \param nrScalesPerOctave number of scales per octave in SIFT
261         * \param minContrast minimum allowed contrast in SIFT
262         * \return Returns a point cloud object containing the SIFT keypoints
263         */
264        pcl::PointCloud<pcl::PointXYZRGB>::Ptr detectSIFTKeyPoints(pcl::PointCloud<
               pcl::PointXYZRGB>::Ptr points, float minScale,int nrOctaves, int
               nrScalesPerOctave, float minContrast);
265
266        /*!
267         * \brief Implementation of FPFH feature descriptor estimation.
268         * \param points input point cloud object
269         * \param normals normals of input point cloud object
270         * \param keyPoints keypoints of point cloud object
271         * \param featureRadius radius to search in FPFH estimation
272         * \return Returns a FPFHSignature33 histogram of the estimated FPFH
                feature descriptors
273         */
274        pcl::PointCloud<pcl::FPFHSignature33>::Ptr computeLocalDescriptorsFPFH(pcl
               ::PointCloud<pcl::PointXYZRGB>::Ptr points, pcl::PointCloud<pcl::Normal
               >::Ptr normals ,pcl::PointCloud<pcl::PointXYZRGB>::Ptr keyPoints, float
                featureRadius);
275
276        /*!
277         * \brief Implementation of the SHOTColor feature descriptor estimation.
278         * \param points input point cloud object
279         * \param normals normals of input point cloud object
280         * \param keyPoints keypoints of point cloud object
281         * \param featureRadius radius to search in SHOT estimation
282         * \return Returns a SHOT1344 histogram of the estimated SHOTColor feature
                descriptors
283         */
284        pcl::PointCloud<pcl::SHOT1344>::Ptr computeLocalDescriptorsSHOTColor(pcl::
               PointCloud<pcl::PointXYZRGB>::Ptr points, pcl::PointCloud<pcl::Normal
               >::Ptr normals ,pcl::PointCloud<pcl::PointXYZRGB>::Ptr keyPoints,float
               featureRadius);
285
286        /*!
287         * \brief Implementation of estimating an initial alignment of FPFH feature
                descriptors
288         * \param sourcePoints input point cloud of cloud 1
289         * \param sourceDescriptors input FPFH descriptors for cloud 1
290         * \param targetPoints input point cloud of cloud 2
291         * \param targetDescriptors input FPFH descriptors for cloud 2
292         * \param minSampleDistance minimum sample distance in alignment estimation
293         * \param maxCorrespondenceDistance maximum correspondence distance in
                alignment estimation
294         * \param nrIterations nr of iterations to run before "giving up"
295         * \return Returns a 4x4 transformation matrix of the estimated
                transformation
296         */
297        Eigen::Matrix4f computeInitialAlignmentFPFH(pcl::PointCloud<pcl::PointXYZ
               RGB>::Ptr sourcePoints, pcl::PointCloud<pcl::FPFHSignature33>::Ptr
               sourceDescriptors,pcl::PointCloud<pcl::PointXYZRGB>::Ptr targetPoints,
               pcl::PointCloud<pcl::FPFHSignature33>::Ptr targetDescriptors,float
               minSampleDistance, float maxCorrespondenceDistance, int nrIterations);
298
```

```
299     /*!
300      * \brief Implementation of estimating an initial alignment of SHOTColor
             feature descriptors
301      * \param sourcePoints input point cloud of cloud 1
302      * \param sourceDescriptors input SHOTColor descriptors for cloud 1
303      * \param targetPoints input point cloud of cloud 2
304      * \param targetDescriptors input SHOTColor descriptors for cloud 2
305      * \param minSampleDistance minimum sample distance in alignment estimation
306      * \param maxCorrespondenceDistance maximum correspondence distance in
             alignment estimation
307      * \param nrIterations nr of iterations to run before "giving up"
308      * \return Returns a 4x4 transformation matrix of the estimated
             transformation
309      * \return
310      */
311     Eigen::Matrix4f computeInitialAlignmentSHOTColor(pcl::PointCloud<pcl::
            PointXYZRGB>::Ptr sourcePoints, pcl::PointCloud<pcl::SHOT1344>::Ptr
            sourceDescriptors,pcl::PointCloud<pcl::PointXYZRGB>::Ptr targetPoints,
            pcl::PointCloud<pcl::SHOT1344>::Ptr targetDescriptors,float
            minSampleDistance, float maxCorrespondenceDistance, int nrIterations);
312
313     /*!
314      * \brief Implementation of plane segmentation.
315      * \param inCloud input point cloud object
316      * \param radius radius to search
317      * \return Returns the point cloud without the plane
318      */
319     pcl::PointCloud<pcl::PointXYZRGB>::Ptr extractPlane(pcl::PointCloud<pcl::
            PointXYZRGB>::Ptr inCloud, double radius);
320
321     /*!
322      * \brief Implementation of plane segmentation.
323      * \param inCloud input point cloud object
324      * \param radius radius to search
325      * \return Returns the plane
326      */
327     pcl::PointCloud<pcl::PointXYZRGB>::Ptr extractPlaneReturnPlane(pcl::
            PointCloud<pcl::PointXYZRGB>::Ptr inCloud, double radius);
328
329     /*!
330      * \brief The PointCloudFeatures struct, contains the total points,
331      * normals, keypoints and feature descriptor for one point cloud. Reduces
332      * the amount of total code for each operation.
333      */
334     struct PointCloudFeatures{
335         pcl::PointCloud<pcl::PointXYZRGB>::Ptr points;
336         pcl::PointCloud<pcl::Normal>::Ptr normals;
337         pcl::PointCloud<pcl::PointXYZRGB>::Ptr keyPoints;
338         pcl::PointCloud<pcl::FPFHSignature33>::Ptr localDescriptorsFPFH;
339         pcl::PointCloud<pcl::SHOT1344>::Ptr localDescriptorsSHOTColor;
340     };
341
342     /*!
343      * \brief Computes the selected feature descriptor and keypoints for a
             point cloud
344      * \param points input point cloud object
345      * \param keyPoints name of the keypoint detector (for example "SIFT")
```

```
346         * \param descriptors name of the feature descriptor estimator (for example
                "FPFH")
347         * \return
348         */
349        PointCloudFeatures computeFeatures(pcl::PointCloud<pcl::PointXYZRGB>::Ptr
               points, QString keyPoints, QString descriptors);
350
351        /*!
352         * \brief Estimate correspondences between two feature descriptor
                histograms (FPFH).
353         * \param sourceDescriptors source cloud feature descriptors
354         * \param targetDescriptors target cloud feature descriptor
355         * \param correspondencesOut output correspondences
356         * \param correspondenceScoresOut output correspondences score
357         */
358        void findFeatureCorrespondences(pcl::PointCloud<pcl::FPFHSignature33>::Ptr
               sourceDescriptors, pcl::PointCloud<pcl::FPFHSignature33>::Ptr
               targetDescriptors,std::vector<int> &correspondencesOut, std::vector<
               float> &correspondenceScoresOut);
359
360        /*!
361         * \brief Visualizes the correspondences between two point clouds.
362         * \param points1 input point cloud 1
363         * \param keyPoints1 input keypoints from point cloud 1
364         * \param points2 input point cloud 2
365         * \param keyPoints2 input keypoints from point cloud 2
366         * \param correspondences correspondences between feature descriptors
367         * \param correspondenceScores correspondence scores between feature
                descriptors
368         * \param maxToDisplay maximum correspondences to display
369         */
370        void visualizeCorrespondences(pcl::PointCloud<pcl::PointXYZRGB>::Ptr
               points1, pcl::PointCloud<pcl::PointXYZRGB>::Ptr keyPoints1,pcl::
               PointCloud<pcl::PointXYZRGB>::Ptr points2, pcl::PointCloud<pcl::
               PointXYZRGB>::Ptr keyPoints2,std::vector<int> &correspondences, std::
               vector<float> &correspondenceScores, int maxToDisplay);
371
372        /*!
373         * \brief Estimates correspondences between FPFH feature descriptors.
374         * \param sourceDescriptors source FPFH feature descriptors
375         * \param targetDescriptors target FPFH feature descriptors
376         * \return Returns a Correspondences object with the correspondences found.
377         */
378        pcl::CorrespondencesPtr findCorrespondences(pcl::PointCloud<pcl::
               FPFHSignature33>::Ptr sourceDescriptors, pcl::PointCloud<pcl::
               FPFHSignature33>::Ptr targetDescriptors);
379
380        /*!
381         * \brief Estimates correspondences between SHOT feature descriptors.
382         * \param sourceDescriptors source SHOT feature descriptors
383         * \param targetDescriptors target SHOT feature descriptors
384         * \return Returns a Correspondences object with the correspondences found.
385         */
386        pcl::CorrespondencesPtr findCorrespondencesSHOT(pcl::PointCloud<pcl::
               SHOT1344>::Ptr sourceDescriptors, pcl::PointCloud<pcl::SHOT1344>::Ptr
               targetDescriptors);
387
```

```
388     /*!
389      * \brief Implementation of a correspondence rejector based on distance.
390      * \param correspondences input correspondences to reject
391      * \param sourceKeyPoints source cloud keypoints
392      * \param targetKeyPoints target cloud keypoints
393      * \param maximumDistance maximum allowed distance before rejection
394      * \return Returns a Correspondences object with the good correspondences.
395      */
396     pcl::CorrespondencesPtr rejectCorrespondencesDistance(pcl::Correspondences
         Ptr correspondences, pcl::PointCloud<pcl::PointXYZRGB>::Ptr
         sourceKeyPoints,pcl::PointCloud<pcl::PointXYZRGB>::Ptr targetKeyPoints,
          float maximumDistance);
397     /*!
398      * \brief Implementation of a correspondence rejector based on RANSAC.
399      * \param correspondences input correspondences to reject
400      * \param sourceKeyPoints source cloud keypoints
401      * \param targetKeyPoints target cloud keypoints
402      * \param inlierTreshold treshold for inliers
403      * \param maxIterations maximum allowed iterations
404      * \return Returns a Correspondences object with the good correspondences.
405      */
406     pcl::CorrespondencesPtr rejectCorrespondencesSampleConsensus(pcl::
         CorrespondencesPtr correspondences, pcl::PointCloud<pcl::PointXYZRGB>::
         Ptr sourceKeyPoints,pcl::PointCloud<pcl::PointXYZRGB>::Ptr
         targetKeyPoints, float inlierTreshold, int maxIterations);
407     /*!
408      * \brief Implementation of a correspondence rejector based on one-to-one.
409      * \param correspondences input correspondences to reject
410      * \return Returns a Correspondences object with the good correspondences.
411      */
412     pcl::CorrespondencesPtr rejectCorrespondencesOneToOne(pcl::Correspondences
         Ptr correspondences);
413
414     /*!
415      * \brief Implementation of a correspondence rejector based on median
          distance.
416      * \param correspondences input correspondences to reject
417      * \param meanDistance mean distance to reject
418      * \return Returns a Correspondences object with the good correspondences.
419      */
420     pcl::CorrespondencesPtr rejectCorrespondencesMedianDistance(pcl::
         CorrespondencesPtr correspondences, double meanDistance);
421
422     /*!
423      * \brief Visualizes the correspondences between two point clouds.
424      * \param sourcePoints input source point cloud
425      * \param targetPoints input target point cloud
426      * \param sourceKeyPoints input source keypoints on point cloud
427      * \param targetKeyPoints input target keypoints on point cloud
428      * \param correspondences total correspondences
429      * \param goodCorrespondences good correspondences after rejection
430      */
431     void visualizeCorrespondences(pcl::PointCloud<pcl::PointXYZRGB>::Ptr
         sourcePoints, pcl::PointCloud<pcl::PointXYZRGB>::Ptr targetPoints,pcl::
         PointCloud<pcl::PointXYZRGB>::Ptr sourceKeyPoints, pcl::PointCloud<pcl
         ::PointXYZRGB>::Ptr targetKeyPoints,pcl::CorrespondencesPtr
         correspondences, pcl::CorrespondencesPtr goodCorrespondences);
```

```
432
433         /*!
434          * \brief Implementation of transformation estimation based on Singular
                 Value Decomposition.
435          * \param sourcePoints input source point cloud
436          * \param targetPoints input target point cloud
437          * \param correspondences correspondences between the two point clouds
438          * \return Returns a 4x4 transformation matrix with the estimated
                 transformation
439          */
440         Eigen::Matrix4f estimateTransformationSVD(pcl::PointCloud<pcl::PointXYZRGB
                >::Ptr sourcePoints, pcl::PointCloud<pcl::PointXYZRGB>::Ptr
                targetPoints,pcl::CorrespondencesPtr correspondences);
441
442         /*!
443          * \brief Implementation of transformation estimation based on
                 LevenbergMarquardt.
444          * \param sourcePoints input source point cloud
445          * \param targetPoints input target point cloud
446          * \param correspondences correspondences between the two point clouds
447          * \return Returns a 4x4 transformation matrix with the estimated
                 transformation
448          */
449         Eigen::Matrix4f estimateTransformationLM(pcl::PointCloud<pcl::PointXYZRGB
                >::Ptr sourcePoints, pcl::PointCloud<pcl::PointXYZRGB>::Ptr
                targetPoints,pcl::CorrespondencesPtr correspondences);
450
451         /*!
452          * \brief Visualizes a transformation between two point clouds.
453          * \param sourcePoints input source point cloud
454          * \param targetPoints input target point cloud
455          * \param transform a 4x4 transformation matrix
456          */
457         void visualizeTransformation(pcl::PointCloud<pcl::PointXYZRGB>::Ptr
                sourcePoints, pcl::PointCloud<pcl::PointXYZRGB>::Ptr targetPoints,
                Eigen::Matrix4f transform);
458
459         /*!
460          * \brief Samples a .STL file to a point cloud.
461          * \param path path to point cloud file
462          * \param resolution resolution of sampling
463          * \param tess_level tessalation level of sampling
464          * \return Returns a point cloud object of the sampled .STL file
465          */
466         pcl::PointCloud<pcl::PointXYZRGB>::Ptr sampleSTL(QString path, int
                resolution, int tess_level);
467
468         /*!
469          * \brief Performs object recognition between a model and a scene.
470          * \param model the model point cloud
471          * \param scene the scene point cloud
472          */
473         void matchModelCloud(pcl::PointCloud<pcl::PointXYZRGB>::Ptr model, pcl::
                PointCloud<pcl::PointXYZRGB>::Ptr scene);
474
475         /*!
476          * \brief Automatic object recognition by subscribing to the point cloud
```

```
         topics.
477      * \param clouds input point clouds, scene and model
478      */
479     void alignAndMatch(std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr>
            clouds);
480
481     /*!
482      * \brief Refines the alignment of 3 point clouds in the robot cell.
483      * \param cloudsIn list of 3 point clouds
484      * \return Returns the reconstructed point cloud scene
485      */
486     pcl::PointCloud<pcl::PointXYZRGB>::Ptr alignCloudsRefined(std::vector<pcl::
            PointCloud<pcl::PointXYZRGB>::Ptr> cloudsIn);
487
488
489 Q_SIGNALS:
490     /*!
491      * \brief Signal to send new information to the GUI
492      * \param labels label text for the GUI
493      * \param show which labels are showing in the GUI
494      * \param stepsAndRange step and range for input box/slider
495      */
496     void sendNewIndexInfo(QStringList labels, QList<bool> show, QList<double>
            stepsAndRange);
497
498     /*!
499      * \brief Signal to send a new visualizer to the GUI.
500      * \param vis the visualizer to send
501      */
502     void sendNewVisualizer(boost::shared_ptr<pcl::visualization::PCLVisualizer>
             vis);
503
504     /*!
505      * \brief Signal to send a new point cloud to the GUI.
506      * \param cloud the cloud to send
507      * \param name name of the cloud in the visualizer (needs to be unique)
508      */
509     void sendNewPointCloud(pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud,
            QString name);
510
511 public Q_SLOTS:
512
513 private:
514     QStringList filterList; //!< List containing available filters
515     pcl::PassThrough<pcl::PointXYZRGB> passThroughFilter; //!< PassThrough
            filter object
516     pcl::PassThrough<pcl::PointXYZRGB> passThroughFilterRGB; //!< PassThrough
            filter object
517     pcl::VoxelGrid<pcl::PointXYZRGB> voxelGridFilter; //!< VoxelGrid filter
            object
518     pcl::VoxelGrid<pcl::PointXYZRGB> voxelGridFilterRGB; //!< VoxelGrid filter
            object
519     pcl::ShadowPoints<pcl::PointXYZRGB, pcl::Normal> shadowPointsFilter; //!<
            ShadowPoint removal filter object
520     pcl::MedianFilter<pcl::PointXYZRGB> medianFilter; //!< Median filter object
521     pcl::StatisticalOutlierRemoval<pcl::PointXYZRGB> statOutlierFilter; //!<
            Outlier removal filter object
```

```
522     pcl::FastBilateralFilter<pcl::PointXYZRGB> bilateralFilter; //!< Bilateral
            filter object
523     boost::shared_ptr<pcl::visualization::PCLVisualizer> visualizer; //!<
            Visualizer object
524     QString lastFiltered; //!< Last used filter
525 };
526
527 }
528
529 #endif
```

PointCloudManipulator.cpp

```
1  #include "../include/qt_master/PointCloudManipulator.hpp"
2
3
4  namespace qt_master {
5
6  PointCloudManipulator::PointCloudManipulator(QObject *parent) :
7      QObject(parent)
8  {
9
10 }
11
12 PointCloudManipulator::~PointCloudManipulator(){}
13
14 QStringList PointCloudManipulator::getFilters()
15 {
16     filterList.append("Passthrough");
17     filterList.append("VoxelGrid");
18     filterList.append("Median");
19     filterList.append("Normals");
20     filterList.append("Plane extraction");
21     filterList.append("Bilateral");
22     return filterList;
23 }
24
25 void PointCloudManipulator::runFilter(int selectedFilter,pcl::PointCloud<pcl::
       PointXYZRGB>::Ptr inCloud,pcl::PointCloud<pcl::PointXYZRGB>::Ptr outCloud,
       double d1, double d2, double d3, QString xyz)
26 {
27     pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud (new pcl::PointCloud<pcl::
           PointXYZRGB>);
28     switch(selectedFilter)
29     {
30     case 0:
31         // PASSTHROUGH FILTER
32         filterPassThrough(inCloud, outCloud, d1, d2, xyz);
33         lastFiltered = "Passthrough filter, ";
34         lastFiltered.append(" Min: ");
35         lastFiltered.append(QString::number(d1));
36         lastFiltered.append(" Max: ");
37         lastFiltered.append(QString::number(d2));
38         lastFiltered.append(" Field: ");
39         lastFiltered.append(xyz);
40         break;
41     case 1:
42         // VOXEL GRID FILTER
43         filterVoxelGrid(inCloud, outCloud, d1);
44         lastFiltered = "VoxelGrid filter, ";
45         lastFiltered.append(" Leaf size : ");
46         lastFiltered.append(QString::number(d1));
47         break;
48     case 2:
49         // MEDIAN FILTER
50         filterMedian(inCloud, outCloud, d1, d2);
51         lastFiltered = "Median filter, ";
52         lastFiltered.append(" Window size: ");
```

```
53          lastFiltered.append(QString::number(d1));
54          lastFiltered.append(" Max allowed movement: ");
55          lastFiltered.append(QString::number(d2));
56          break;
57      case 3:
58          // NEW VISUALIZER
59          // Send new vis with normals filtered shit
60          filterNormal(inCloud, d1, d2);
61          lastFiltered = "Normals filter, ";
62          lastFiltered.append(" Radius: ");
63          lastFiltered.append(QString::number(d1));
64          lastFiltered.append(" Nr. to display: ");
65          lastFiltered.append(QString::number(d2));
66          break;
67      case 4:
68          // Plane extraction
69          cloud = extractPlane(inCloud, d1);
70          Q_EMIT sendNewPointCloud(cloud, "filteredCloud");
71          lastFiltered = "Plane extraction, ";
72          lastFiltered.append(" Radius: ");
73          lastFiltered.append(QString::number(d1));
74          break;
75      case 5:
76          filterBilateral(inCloud, d1, d2);
77          lastFiltered = "Bilateral filter, ";
78          lastFiltered.append(" SigmaS: ");
79          lastFiltered.append(QString::number(d1));
80          lastFiltered.append(" SigmaR: ");
81          lastFiltered.append(QString::number(d2));
82          break;
83      default:
84          ;
85      }
86  }
87
88  void PointCloudManipulator::getNewIndexInfo(int selectedFilter)
89  {
90      QList<QString> labels;
91      labels.append("");
92      labels.append("");
93      labels.append("");
94      QList<bool> show;
95      show.append(false);
96      show.append(false);
97      show.append(false);
98      show.append(false);
99      QList<double> stepsAndRange;
100     stepsAndRange.append(0.1);
101     stepsAndRange.append(0.1);
102     stepsAndRange.append(0.1);
103     stepsAndRange.append(-5);
104     stepsAndRange.append(5);
105     stepsAndRange.append(-5);
106     stepsAndRange.append(5);
107     stepsAndRange.append(-5);
108     stepsAndRange.append(5);
109
```

```
110     switch(selectedFilter)
111     {
112     case 0:
113         // PASSTHROUGH FILTER
114         labels.replace(0, "Minimum:");
115         labels.replace(1, "Maximum:");
116         show.replace(0,true);
117         show.replace(1,true);
118         show.replace(3,true);
119         Q_EMIT sendNewIndexInfo(labels, show, stepsAndRange);
120         break;
121     case 1:
122         // VOXEL GRID FILTER
123         labels.replace(0, "Leaf size:");
124         show.replace(0, true);
125         stepsAndRange.replace(0, 0.001);
126         stepsAndRange.replace(3, 0.001);
127         stepsAndRange.replace(4, 0.2);
128         Q_EMIT sendNewIndexInfo(labels, show, stepsAndRange);
129         break;
130     case 2:
131         // MEDIAN FILTER
132         labels.replace(0, "Window size:");
133         labels.replace(1, "Max movement:");
134         show.replace(0, true);
135         show.replace(1, true);
136         stepsAndRange.replace(0, 1);
137         stepsAndRange.replace(3, 0);
138         stepsAndRange.replace(4, 100);
139         stepsAndRange.replace(5, 0);
140         stepsAndRange.replace(6, 10);
141         Q_EMIT sendNewIndexInfo(labels, show, stepsAndRange);
142         break;
143     case 3:
144         // NORMALS
145         labels.replace(0, "Radius:");
146         labels.replace(1, "Nr. Normals");
147         show.replace(0, true);
148         show.replace(1, true);
149         stepsAndRange.replace(0, 0.001);
150         stepsAndRange.replace(1, 1);
151         stepsAndRange.replace(3, 0.001);
152         stepsAndRange.replace(4, 0.5);
153         stepsAndRange.replace(5, 1);
154         stepsAndRange.replace(6, 10);
155         Q_EMIT sendNewIndexInfo(labels, show, stepsAndRange);
156         break;
157     case 4:
158         // TRANSLATION
159         labels.replace(0, "Radius: ");
160         show.replace(0, true);
161         stepsAndRange.replace(0, 0.01);
162         Q_EMIT sendNewIndexInfo(labels, show, stepsAndRange);
163         break;
164     case 5:
165         labels.replace(0, "Sigma S");
166         labels.replace(1, "Sigma R");
```

```
167         show.replace(0, true);
168         show.replace(1,true);
169         stepsAndRange.replace(0, 1);
170         stepsAndRange.replace(1, 0.001);
171         stepsAndRange.replace(3, 0);
172         stepsAndRange.replace(4, 10);
173         stepsAndRange.replace(5, 0);
174         stepsAndRange.replace(6, 0.1);
175 
176         Q_EMIT sendNewIndexInfo(labels, show, stepsAndRange);
177     default:
178         ;
179     }
180 }
181 
182 void PointCloudManipulator::filterPassThrough(pcl::PointCloud<pcl::PointXYZRGB
    >::Ptr inCloud, pcl::PointCloud<pcl::PointXYZRGB>::Ptr outCloud, double
    limitMin, double limitMax, QString field)
183 {
184     passThroughFilter.setInputCloud(inCloud);
185     passThroughFilter.setFilterFieldName(field.toStdString());
186     passThroughFilter.setFilterLimits(limitMin, limitMax);
187     passThroughFilter.setKeepOrganized(true);
188     passThroughFilter.filter(*outCloud);
189     Q_EMIT sendNewPointCloud(outCloud, "filteredCloud");
190 }
191 
192 void PointCloudManipulator::filterVoxelGrid(pcl::PointCloud<pcl::PointXYZRGB>::
    Ptr inCloud, pcl::PointCloud<pcl::PointXYZRGB>::Ptr outCloud, double
    leafSize)
193 {
194     float leaf = leafSize;
195     voxelGridFilter.setInputCloud(inCloud);
196     voxelGridFilter.setLeafSize(leaf, leaf, leaf);
197     voxelGridFilter.filter(*outCloud);
198     Q_EMIT sendNewPointCloud(outCloud, "filteredCloud");
199 }
200 
201 void PointCloudManipulator::filterMedian(pcl::PointCloud<pcl::PointXYZRGB>::Ptr
     inCloud, pcl::PointCloud<pcl::PointXYZRGB>::Ptr outCloud, double
    windowSize, double maxMovement)
202 {
203     int windowSizeTmp = (int)windowSize;
204     medianFilter.setInputCloud(inCloud);
205     medianFilter.setWindowSize(windowSizeTmp);
206     medianFilter.setMaxAllowedMovement(maxMovement);
207     medianFilter.applyFilter(*outCloud);
208     Q_EMIT sendNewPointCloud(outCloud, "filteredCloud");
209 }
210 
211 void PointCloudManipulator::filterNormal(pcl::PointCloud<pcl::PointXYZRGB>::Ptr
     inCloud, double radius, double nrToDisplay)
212 {
213     int tmpDisplay = (int) nrToDisplay;
214     visualizer.reset(new pcl::visualization::PCLVisualizer ("viewer2", false));
215     pcl::PointCloud<pcl::Normal>::Ptr normals_out (new pcl::PointCloud<pcl
        ::Normal>);
```

```
216     pcl::search::KdTree<pcl::PointXYZRGB>::Ptr tree (new pcl::search::KdTree<
            pcl::PointXYZRGB> ());
217     pcl::NormalEstimationOMP<pcl::PointXYZRGB, pcl::Normal> norm_est;
218     norm_est.setSearchMethod(tree);
219     norm_est.setKSearch(10);
220     norm_est.setInputCloud (inCloud);
221     norm_est.compute (*normals_out);
222     visualizer->addPointCloud<pcl::PointXYZRGB> (inCloud, "filteredCloud");
223     visualizer->addPointCloudNormals<pcl::PointXYZRGB, pcl::Normal> (inCloud,
            normals_out, tmpDisplay, 0.05, "normals");
224     visualizer->setPointCloudRenderingProperties(pcl::visualization::
            PCL_VISUALIZER_COLOR, 0.0,0.0,1.0, "normals");
225     Q_EMIT sendNewVisualizer(visualizer);
226 }
227
228 void PointCloudManipulator::filterBilateral(pcl::PointCloud<pcl::PointXYZRGB>::
        Ptr inCloud, double sigmaS, float sigmaR)
229 {
230     pcl::PointCloud<pcl::PointXYZRGB>::Ptr filteredCloud (new pcl::PointCloud<
            pcl::PointXYZRGB>);
231     bilateralFilter.setInputCloud(inCloud);
232     bilateralFilter.setSigmaS(sigmaS);
233     bilateralFilter.setSigmaR(sigmaR);
234     bilateralFilter.filter(*filteredCloud);
235     Q_EMIT sendNewPointCloud(filteredCloud, "filteredCloud");
236 }
237
238 void PointCloudManipulator::translateCloud(pcl::PointCloud<pcl::PointXYZRGB>::
        Ptr  inCloud, double rX, double rY, double rZ, double tX, double tY, double
         tZ)
239 {
240     pcl::PointCloud<pcl::PointXYZRGB>::Ptr translatedCloud (new pcl::PointCloud
            <pcl::PointXYZRGB>);
241     Eigen::Matrix4f transform = Eigen::Matrix4f::Identity();
242     double x = M_PI/180*rX;
243     double y = M_PI/180*rY;
244     double z = M_PI/180*rZ;
245     transform << cos(y) * cos(z), -cos(y) * sin(z), sin(y), tX,
246             (cos(x) * sin(z)) + (cos(z) * sin(x) * sin(y)), (cos(x) * cos(z)) -
                    (sin(x) * sin(y) * sin(z)), -cos(y) *
247             sin(x), tY,
248             (sin(x) * sin(z)) - (cos(x) * cos(z) * sin(y)), (cos(z) * sin(x)) +
                    (cos(x) * sin(y) * sin(z)), cos(x) *
249             cos(y), tZ,
250             0.0, 0.0, 0.0, 1.0;
251
252     pcl::transformPointCloud(*inCloud, *translatedCloud, transform);
253     Q_EMIT sendNewPointCloud(translatedCloud, "translatedCloud");
254
255 }
256
257 void PointCloudManipulator::getNewVisualizer(int selectedFilter)
258 {
259     visualizer.reset(new pcl::visualization::PCLVisualizer ("viewer2", false));
260     Q_EMIT sendNewVisualizer(visualizer);
261 }
262
```

```
263 QString PointCloudManipulator::getLastFiltered()
264 {
265     return lastFiltered;
266 }
267
268 pcl::PointCloud<pcl::Normal>::Ptr PointCloudManipulator::computeSurfaceNormals(
    pcl::PointCloud<pcl::PointXYZRGB>::Ptr input, float radius)
269 {
270     pcl::NormalEstimationOMP<pcl::PointXYZRGB, pcl::Normal> normal_estimation;
271     pcl::PointCloud<pcl::Normal>::Ptr normals (new pcl::PointCloud<pcl::Normal
        >);
272     pcl::search::KdTree<pcl::PointXYZRGB>::Ptr tree (new pcl::search::KdTree<
        pcl::PointXYZRGB> ());
273     normal_estimation.setNumberOfThreads(8);
274     normal_estimation.setSearchMethod (tree);
275     normal_estimation.setRadiusSearch (radius);
276     normal_estimation.setKSearch(10);
277     normal_estimation.setInputCloud (input);
278     normal_estimation.compute (*normals);
279     return (normals);
280 }
281
282 pcl::PointCloud<pcl::PointNormal>::Ptr PointCloudManipulator::
    computeSurfacePointNormals(pcl::PointCloud<pcl::PointXYZRGB>::Ptr input,pcl
    ::PointCloud<pcl::PointXYZRGB>::Ptr surface, float radius)
283 {
284     pcl::NormalEstimationOMP<pcl::PointXYZRGB, pcl::PointNormal>
        normal_estimation;
285     pcl::PointCloud<pcl::PointNormal>::Ptr normals (new pcl::PointCloud<pcl::
        PointNormal>);
286     pcl::search::KdTree<pcl::PointXYZRGB>::Ptr tree (new pcl::search::KdTree<
        pcl::PointXYZRGB> ());
287     normal_estimation.setNumberOfThreads(8);
288     normal_estimation.setSearchSurface(surface);
289     normal_estimation.setSearchMethod (tree);
290     normal_estimation.setRadiusSearch (radius);
291     normal_estimation.setInputCloud (input);
292     normal_estimation.compute (*normals);
293     return (normals);
294 }
295
296 std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr> PointCloudManipulator::
    extractClusters(pcl::PointCloud<pcl::PointXYZRGB>::Ptr input, double
    distance)
297 {
298     pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_f (new pcl::PointCloud<pcl::
        PointXYZRGB>), outcloud(new pcl::PointCloud<pcl::PointXYZRGB>);
299     pcl::PointCloud<pcl::PointXYZRGB>::Ptr incloud (new pcl::PointCloud<pcl::
        PointXYZRGB>);
300     pcl::copyPointCloud(*input,*incloud);
301
302     pcl::SACSegmentation<pcl::PointXYZRGB> seg;
303     pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
304     pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
305     pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_plane (new pcl::PointCloud<pcl
        ::PointXYZRGB> ());
306     seg.setOptimizeCoefficients (true);
```

```
307     seg.setModelType (pcl::SACMODEL_PLANE);
308     seg.setMethodType (pcl::SAC_RANSAC);
309     seg.setMaxIterations (100);
310     seg.setDistanceThreshold (distance);
311     int i=0, nr_points = (int) incloud->points.size ();
312
313     while (incloud->points.size () > 0.3 * nr_points)
314     {
315         seg.setInputCloud (incloud);
316         seg.segment (*inliers, *coefficients);
317         if (inliers->indices.size () == 0)
318         {
319             std::cout << "Could not estimate a planar model for the given
                dataset." << std::endl;
320             break;
321         }
322         pcl::ExtractIndices<pcl::PointXYZRGB> extract;
323         extract.setInputCloud (incloud);
324         extract.setIndices (inliers);
325         extract.setNegative (false);
326         extract.filter (*cloud_plane);
327         extract.setNegative (true);
328         extract.filter (*cloud_f);
329         *incloud = *cloud_f;
330     }
331     pcl::search::KdTree<pcl::PointXYZRGB>::Ptr tree (new pcl::search::KdTree<
        pcl::PointXYZRGB>);
332     tree->setInputCloud (incloud);
333     std::vector<pcl::PointIndices> cluster_indices;
334     pcl::EuclideanClusterExtraction<pcl::PointXYZRGB> ec;
335     ec.setClusterTolerance (0.02);
336     ec.setMinClusterSize (200);
337     ec.setMaxClusterSize (25000);
338     ec.setSearchMethod (tree);
339     ec.setInputCloud (incloud);
340     ec.extract (cluster_indices);
341
342     std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr> clusters;
343     for(int i = 0; i< cluster_indices.size(); i++){
344         pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmpcloud (new pcl::PointCloud<
            pcl::PointXYZRGB>);
345         pcl::copyPointCloud(*incloud,cluster_indices[i],*tmpcloud);
346         clusters.push_back(tmpcloud);
347     }
348
349     pcl::PointCloud<pcl::PointXYZRGB>::Ptr clusterCloud (new pcl::PointCloud<
        pcl::PointXYZRGB>);
350     for(int i = 0; i< clusters.size(); i++){
351         *clusterCloud += *clusters.at(i);
352     }
353
354     Q_EMIT sendNewPointCloud(clusterCloud, "filteredCloud");
355
356     return (clusters);
357 }
358
359 pcl::PointCloud<pcl::PointXYZRGB>::Ptr PointCloudManipulator::
```

```
    detectSIFTKeyPoints(pcl::PointCloud<pcl::PointXYZRGB>::Ptr points, float
    minScale,int nrOctaves, int nrScalesPerOctave, float minContrast)
360 {
361     pcl::SIFTKeypoint<pcl::PointXYZRGB, pcl::PointWithScale> siftDetect;
362     siftDetect.setSearchMethod(pcl::search::Search<pcl::PointXYZRGB>::Ptr (new
            pcl::search::KdTree<pcl::PointXYZRGB>));
363     siftDetect.setScales(minScale,nrOctaves,nrScalesPerOctave);
364     siftDetect.setMinimumContrast(minContrast);
365     siftDetect.setInputCloud(points);
366     pcl::PointCloud<pcl::PointWithScale> tmpKeyPoints;
367     siftDetect.compute(tmpKeyPoints);
368     pcl::PointCloud<pcl::PointXYZRGB>::Ptr keyPoints (new pcl::PointCloud<pcl::
            PointXYZRGB>);
369     pcl::copyPointCloud(tmpKeyPoints, *keyPoints);
370     return keyPoints;
371 }
372
373 pcl::PointCloud<pcl::PointXYZRGB>::Ptr PointCloudManipulator::filterVoxel(pcl::
    PointCloud<pcl::PointXYZRGB>::Ptr inCloud, double leafSize)
374 {
375     float leaf = leafSize;
376     pcl::PointCloud<pcl::PointXYZRGB>::Ptr filteredCloud (new pcl::PointCloud<
            pcl::PointXYZRGB>());
377     voxelGridFilterRGB.setInputCloud(inCloud);
378     voxelGridFilterRGB.setLeafSize(leaf, leaf, leaf);
379     voxelGridFilterRGB.filter(*filteredCloud);
380     return filteredCloud;
381 }
382
383 pcl::PointCloud<pcl::PointXYZRGB>::Ptr PointCloudManipulator::filterShadowPoint
    (pcl::PointCloud<pcl::PointXYZRGB>::Ptr inCloud,pcl::PointCloud<pcl::Normal
    >::Ptr normals, double threshold)
384 {
385     pcl::PointCloud<pcl::PointXYZRGB>::Ptr filteredCloud (new pcl::PointCloud<
            pcl::PointXYZRGB>());
386     shadowPointsFilter.setInputCloud(inCloud);
387     shadowPointsFilter.setKeepOrganized(true);
388     shadowPointsFilter.setNormals(normals);
389     shadowPointsFilter.setThreshold(threshold);
390     shadowPointsFilter.filter(*filteredCloud);
391     return filteredCloud;
392 }
393
394 pcl::PointCloud<pcl::PointXYZRGB>::Ptr PointCloudManipulator::filterOutlier(pcl
    ::PointCloud<pcl::PointXYZRGB>::Ptr inCloud)
395 {
396     pcl::StatisticalOutlierRemoval<pcl::PointXYZRGB> sor;
397     pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_filtered (new pcl::PointCloud<
            pcl::PointXYZRGB>());
398     sor.setInputCloud (inCloud);
399     sor.setMeanK (20);
400     sor.setStddevMulThresh (1.0);
401     sor.filter (*cloud_filtered);
402     return cloud_filtered;
403 }
404
405 pcl::PointCloud<pcl::PointXYZRGB>::Ptr PointCloudManipulator::filterPassThrough
```

```
    (pcl::PointCloud<pcl::PointXYZRGB>::Ptr inCloud, double limitMin, double
    limitMax, QString field)
406 {
407     pcl::PointCloud<pcl::PointXYZRGB>::Ptr filteredCloud (new pcl::PointCloud<
            pcl::PointXYZRGB>());
408     passThroughFilterRGB.setInputCloud(inCloud);
409     passThroughFilterRGB.setFilterLimits(limitMin, limitMax);
410     passThroughFilterRGB.setFilterFieldName(field.toStdString());
411     passThroughFilterRGB.setKeepOrganized(true);
412     passThroughFilterRGB.filter(*filteredCloud);
413     return filteredCloud;
414 }
415
416 pcl::PointCloud<pcl::FPFHSignature33>::Ptr PointCloudManipulator::
    computeLocalDescriptorsFPFH(pcl::PointCloud<pcl::PointXYZRGB>::Ptr points,
    pcl::PointCloud<pcl::Normal>::Ptr normals ,pcl::PointCloud<pcl::PointXYZRGB
    >::Ptr keyPoints, float featureRadius)
417 {
418     pcl::FPFHEstimationOMP<pcl::PointXYZRGB, pcl::Normal, pcl::FPFHSignature33>
            fpfhEstimation;
419     fpfhEstimation.setNumberOfThreads(8);
420     fpfhEstimation.setSearchMethod(pcl::search::Search<pcl::PointXYZRGB>::Ptr (
            new pcl::search::KdTree<pcl::PointXYZRGB>));
421     fpfhEstimation.setRadiusSearch(featureRadius);
422     fpfhEstimation.setSearchSurface(points);
423     fpfhEstimation.setInputNormals(normals);
424     fpfhEstimation.setInputCloud(keyPoints);
425     pcl::PointCloud<pcl::FPFHSignature33>::Ptr localDescriptors (new pcl::
            PointCloud<pcl::FPFHSignature33>);
426     fpfhEstimation.compute(*localDescriptors);
427     return localDescriptors;
428 }
429
430 pcl::PointCloud<pcl::SHOT1344>::Ptr PointCloudManipulator::
    computeLocalDescriptorsSHOTColor(pcl::PointCloud<pcl::PointXYZRGB>::Ptr
    points, pcl::PointCloud<pcl::Normal>::Ptr normals ,pcl::PointCloud<pcl::
    PointXYZRGB>::Ptr keyPoints,float featureRadius)
431 {
432     pcl::SHOTColorEstimationOMP<pcl::PointXYZRGB, pcl::Normal, pcl::SHOT1344>
            shot;
433     pcl::PointCloud<pcl::SHOT1344>::Ptr localDescriptors (new pcl::PointCloud<
            pcl::SHOT1344>);
434     shot.setNumberOfThreads(8);
435     shot.setInputCloud(keyPoints);
436     shot.setSearchSurface(points);
437     shot.setInputNormals(normals);
438     shot.setRadiusSearch(featureRadius);
439     shot.compute(*localDescriptors);
440     return (localDescriptors);
441 }
442
443 Eigen::Matrix4f PointCloudManipulator::computeInitialAlignmentFPFH(pcl::
    PointCloud<pcl::PointXYZRGB>::Ptr sourcePoints, pcl::PointCloud<pcl::
    FPFHSignature33>::Ptr sourceDescriptors,pcl::PointCloud<pcl::PointXYZRGB>::
    Ptr targetPoints, pcl::PointCloud<pcl::FPFHSignature33>::Ptr
    targetDescriptors,float minSampleDistance, float maxCorrespondenceDistance,
     int nrIterations)
```

```
444 {
445     pcl::SampleConsensusInitialAlignment<pcl::PointXYZRGB, pcl::PointXYZRGB,
            pcl::FPFHSignature33> sacInitAlign;
446     sacInitAlign.setMinSampleDistance(minSampleDistance);
447     sacInitAlign.setMaxCorrespondenceDistance(maxCorrespondenceDistance);
448     sacInitAlign.setMaximumIterations(nrIterations);
449     sacInitAlign.setInputSource(sourcePoints);
450     sacInitAlign.setSourceFeatures(sourceDescriptors);
451     sacInitAlign.setInputTarget(targetPoints);
452     sacInitAlign.setTargetFeatures(targetDescriptors);
453     pcl::PointCloud<pcl::PointXYZRGB> regOutput;
454     sacInitAlign.align(regOutput);
455
456     return (sacInitAlign.getFinalTransformation());
457 }
458
459 Eigen::Matrix4f PointCloudManipulator::computeInitialAlignmentSHOTColor(pcl::
        PointCloud<pcl::PointXYZRGB>::Ptr sourcePoints, pcl::PointCloud<pcl::
        SHOT1344>::Ptr sourceDescriptors,pcl::PointCloud<pcl::PointXYZRGB>::Ptr
        targetPoints, pcl::PointCloud<pcl::SHOT1344>::Ptr targetDescriptors,float
        minSampleDistance, float maxCorrespondenceDistance, int nrIterations)
460 {
461     pcl::SampleConsensusInitialAlignment<pcl::PointXYZRGB, pcl::PointXYZRGB,
            pcl::SHOT1344> sacInitAlign;
462     sacInitAlign.setMinSampleDistance(minSampleDistance);
463     sacInitAlign.setMaxCorrespondenceDistance(maxCorrespondenceDistance);
464     sacInitAlign.setMaximumIterations(nrIterations);
465     sacInitAlign.setInputSource(sourcePoints);
466     sacInitAlign.setSourceFeatures(sourceDescriptors);
467     sacInitAlign.setInputTarget(targetPoints);
468     sacInitAlign.setTargetFeatures(targetDescriptors);
469     pcl::PointCloud<pcl::PointXYZRGB> regOutput;
470     sacInitAlign.align(regOutput);
471
472     return (sacInitAlign.getFinalTransformation());
473 }
474
475 pcl::PointCloud<pcl::PointXYZRGB>::Ptr PointCloudManipulator::extractPlane(pcl
        ::PointCloud<pcl::PointXYZRGB>::Ptr inCloud, double radius)
476 {
477     pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_plane (new pcl::PointCloud<pcl
            ::PointXYZRGB>);
478     pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_no_plane (new pcl::PointCloud<
            pcl::PointXYZRGB>);
479     pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
480     pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
481     pcl::SACSegmentation<pcl::PointXYZRGB> seg;
482     seg.setOptimizeCoefficients (true);
483     seg.setModelType (pcl::SACMODEL_PLANE);
484     seg.setMethodType (pcl::SAC_RANSAC);
485     seg.setMaxIterations(100);
486     seg.setDistanceThreshold (radius);
487     seg.setInputCloud (inCloud);
488     seg.segment (*inliers, *coefficients);
489
490     pcl::ExtractIndices<pcl::PointXYZRGB> extract;
491     extract.setInputCloud (inCloud);
```

```
492     extract.setIndices (inliers);
493     extract.setNegative (false);
494     extract.filter (*cloud_plane);
495     extract.setNegative(true);
496     extract.filter(*cloud_no_plane);
497 
498     return (cloud_no_plane);
499 }
500 
501 pcl::PointCloud<pcl::PointXYZRGB>::Ptr PointCloudManipulator::
    extractPlaneReturnPlane(pcl::PointCloud<pcl::PointXYZRGB>::Ptr inCloud,
    double radius)
502 {
503     pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_plane (new pcl::PointCloud<pcl
        ::PointXYZRGB>);
504     pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
505     pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
506     pcl::SACSegmentation<pcl::PointXYZRGB> seg;
507     seg.setOptimizeCoefficients (true);
508     seg.setModelType (pcl::SACMODEL_PLANE);
509     seg.setMethodType (pcl::SAC_RANSAC);
510     seg.setMaxIterations(100);
511     seg.setDistanceThreshold (radius);
512     seg.setInputCloud (inCloud);
513     seg.segment (*inliers, *coefficients);
514 
515     pcl::ExtractIndices<pcl::PointXYZRGB> extract;
516     extract.setInputCloud (inCloud);
517     extract.setIndices (inliers);
518     extract.setNegative (false);
519     extract.filter (*cloud_plane);
520 
521     return (cloud_plane);
522 }
523 
524 PointCloudManipulator::PointCloudFeatures PointCloudManipulator::
    computeFeatures(pcl::PointCloud<pcl::PointXYZRGB>::Ptr inCloud, QString
    keyPoints, QString descriptors)
525 {
526     PointCloudFeatures features;
527     features.points = inCloud;
528 
529     features.normals = computeSurfaceNormals(features.points, 0.005);
530     std::cout << "Found normals: " ;
531     std::cout << features.normals->size() << std::endl;
532 
533     if(QString::compare(keyPoints, "SIFT", Qt::CaseInsensitive) == 0){
534         // CONTRAST IS THE LAST PART (0.01, 3, 3, 0.2)
535         features.keyPoints = detectSIFTKeyPoints(features.points, 0.01, 15, 15,
                0.0);
536         std::cout << "Found SIFT keypoints: " ;
537         std::cout << features.keyPoints->size() << std::endl;
538     }
539     else if(QString::compare(keyPoints, "VOXEL", Qt::CaseInsensitive) == 0){
540         features.keyPoints = filterVoxel(features.points, 0.001);
541         std::cout << "Found VOXEL keypoints: " ;
542         std::cout << features.keyPoints->size() << std::endl;
```

```
543     }
544
545     if(QString::compare(descriptors, "FPFH", Qt::CaseInsensitive) == 0){
546         features.localDescriptorsFPFH = computeLocalDescriptorsFPFH(features.
                points, features.normals, features.keyPoints, 0.55); //0.15
547         std::cout << "Found FPFH descriptors: " ;
548         std::cout << features.localDescriptorsFPFH->size() << std::endl;
549     }
550     else if(QString::compare(descriptors, "SHOTCOLOR", Qt::CaseInsensitive) ==
            0){
551         features.localDescriptorsSHOTColor = computeLocalDescriptorsSHOTColor(
                features.points, features.normals,features.keyPoints,0.55);
552         std::cout << "Found SHOTColor descriptors: " ;
553         std::cout << features.localDescriptorsSHOTColor->size() << std::endl;
554     }
555     return features;
556 }
557
558 void PointCloudManipulator::findFeatureCorrespondences(pcl::PointCloud<pcl::
        FPFHSignature33>::Ptr sourceDescriptors, pcl::PointCloud<pcl::
        FPFHSignature33>::Ptr targetDescriptors,std::vector<int> &
        correspondencesOut, std::vector<float> &correspondenceScoresOut)
559 {
560     correspondencesOut.resize(sourceDescriptors->size());
561     correspondenceScoresOut.resize(sourceDescriptors->size());
562     pcl::KdTreeFLANN<pcl::FPFHSignature33> descriptorKDTree;
563     descriptorKDTree.setInputCloud(targetDescriptors);
564     const int k = 1;
565     std::vector<int> k_indices (k);
566     std::vector<float> k_squared_distances (k);
567     for (size_t i = 0; i < sourceDescriptors->size(); i++){
568         descriptorKDTree.nearestKSearch(*sourceDescriptors, i, k , k_indices,
                k_squared_distances);
569         correspondencesOut[i] = k_indices[0];
570         correspondenceScoresOut[i] = k_squared_distances[0];
571     }
572 }
573
574 void PointCloudManipulator::visualizeCorrespondences(pcl::PointCloud<pcl::
        PointXYZRGB>::Ptr points1, pcl::PointCloud<pcl::PointXYZRGB>::Ptr
        keyPoints1,pcl::PointCloud<pcl::PointXYZRGB>::Ptr points2, pcl::PointCloud<
        pcl::PointXYZRGB>::Ptr keyPoints2,std::vector<int> &correspondences, std::
        vector<float> &correspondenceScores, int maxToDisplay)
575 {
576     pcl::PointCloud<pcl::PointXYZRGB>::Ptr points_left (new pcl::PointCloud<pcl
            ::PointXYZRGB>);
577     pcl::PointCloud<pcl::PointXYZRGB>::Ptr keypoints_left (new pcl::PointCloud<
            pcl::PointXYZRGB>);
578     pcl::PointCloud<pcl::PointXYZRGB>::Ptr points_right (new pcl::PointCloud<
            pcl::PointXYZRGB>);
579     pcl::PointCloud<pcl::PointXYZRGB>::Ptr keypoints_right (new pcl::PointCloud
            <pcl::PointXYZRGB>);
580     const Eigen::Vector3f translate (0.4, 0.0, 0.0);
581     const Eigen::Quaternionf no_rotation (0, 0, 0, 0);
582     pcl::transformPointCloud (*points1, *points_left, -translate, no_rotation);
583     pcl::transformPointCloud (*keyPoints1, *keypoints_left, -translate,
            no_rotation);
```

```
584     pcl::transformPointCloud (*points2, *points_right, translate, no_rotation);
585     pcl::transformPointCloud (*keyPoints2, *keypoints_right, translate,
            no_rotation);
586
587     pcl::visualization::PCLVisualizer vis;
588     vis.addPointCloud (points_left, "points_left");
589     vis.addPointCloud (points_right, "points_right");
590
591     std::vector<float> temp (correspondenceScores);
592     std::sort (temp.begin (), temp.end ());
593     if (maxToDisplay >= temp.size ())
594         maxToDisplay = temp.size () - 1;
595     float threshold = temp[maxToDisplay];
596     for (size_t i = 0; i < keypoints_left->size (); ++i)
597     {
598         if (correspondenceScores[i] > threshold)
599         {
600             continue;
601         }
602         const pcl::PointXYZRGB & p_left = keypoints_left->points[i];
603         const pcl::PointXYZRGB & p_right = keypoints_right->points[
                correspondences[i]];
604         double r = (rand() % 100);
605         double g = (rand() % 100);
606         double b = (rand() % 100);
607         double max_channel = std::max (r, std::max (g, b));
608         r /= max_channel;
609         g /= max_channel;
610         b /= max_channel;
611         std::stringstream ss ("line");
612         ss << i;
613         vis.addLine (p_left, p_right, r, g, b, ss.str ());
614     }
615     vis.resetCamera ();
616     vis.spin ();
617 }
618
619 pcl::CorrespondencesPtr PointCloudManipulator::findCorrespondences(pcl::
        PointCloud<pcl::FPFHSignature33>::Ptr sourceDescriptors, pcl::PointCloud<
        pcl::FPFHSignature33>::Ptr targetDescriptors)
620 {
621     pcl::CorrespondencesPtr correspondences (new pcl::Correspondences);
622     pcl::registration::CorrespondenceEstimation<pcl::FPFHSignature33, pcl::
            FPFHSignature33> est;
623     est.setInputSource(sourceDescriptors);
624     est.setInputTarget(targetDescriptors);
625     est.determineCorrespondences(*correspondences);
626     return correspondences;
627 }
628
629 pcl::CorrespondencesPtr PointCloudManipulator::findCorrespondencesSHOT(pcl::
        PointCloud<pcl::SHOT1344>::Ptr sourceDescriptors, pcl::PointCloud<pcl::
        SHOT1344>::Ptr targetDescriptors){
630     pcl::CorrespondencesPtr correspondences (new pcl::Correspondences);
631     pcl::registration::CorrespondenceEstimation<pcl::SHOT1344, pcl::SHOT1344>
            est;
632     est.setInputSource(sourceDescriptors);
```

```
633     est.setInputTarget(targetDescriptors);
634     est.determineCorrespondences(*correspondences);
635     return correspondences;
636 }
637
638 pcl::CorrespondencesPtr PointCloudManipulator::rejectCorrespondencesDistance(
        pcl::CorrespondencesPtr correspondences, pcl::PointCloud<pcl::PointXYZRGB
        >::Ptr sourceKeyPoints,pcl::PointCloud<pcl::PointXYZRGB>::Ptr
        targetKeyPoints, float maximumDistance)
639 {
640     pcl::CorrespondencesPtr goodCorrespondences (new pcl::Correspondences);
641     pcl::registration::CorrespondenceRejectorDistance rej;
642     rej.setInputSource<pcl::PointXYZRGB> (sourceKeyPoints);
643     rej.setInputTarget<pcl::PointXYZRGB> (targetKeyPoints);
644     rej.setMaximumDistance(maximumDistance); //meters
645     rej.setInputCorrespondences(correspondences);
646     rej.getCorrespondences(*goodCorrespondences);
647     return goodCorrespondences;
648 }
649
650 pcl::CorrespondencesPtr PointCloudManipulator::
        rejectCorrespondencesSampleConsensus(pcl::CorrespondencesPtr
        correspondences,pcl::PointCloud<pcl::PointXYZRGB>::Ptr sourceKeyPoints,pcl
        ::PointCloud<pcl::PointXYZRGB>::Ptr targetKeyPoints,float inlierTreshold,
        int maxIterations)
651 {
652     pcl::CorrespondencesPtr goodCorrespondences (new pcl::Correspondences);
653     pcl::registration::CorrespondenceRejectorSampleConsensus<pcl::PointXYZRGB>
            rej;
654     rej.setInputSource(sourceKeyPoints);
655     rej.setInputTarget(targetKeyPoints);
656     rej.setInlierThreshold(inlierTreshold);
657     rej.setMaximumIterations(maxIterations);
658     rej.setInputCorrespondences(correspondences);
659     rej.getCorrespondences(*goodCorrespondences);
660     return goodCorrespondences;
661
662 }
663
664 pcl::CorrespondencesPtr PointCloudManipulator::rejectCorrespondencesOneToOne(
        pcl::CorrespondencesPtr correspondences)
665 {
666     pcl::CorrespondencesPtr goodCorrespondences (new pcl::Correspondences);
667     pcl::registration::CorrespondenceRejectorOneToOne rej;
668     rej.setInputCorrespondences(correspondences);
669     rej.getCorrespondences(*goodCorrespondences);
670     return goodCorrespondences;
671 }
672
673 pcl::CorrespondencesPtr PointCloudManipulator::
        rejectCorrespondencesMedianDistance(pcl::CorrespondencesPtr correspondences
        , double meanDistance)
674 {
675     pcl::CorrespondencesPtr corrRejectMed (new pcl::Correspondences);
676     pcl::registration::CorrespondenceRejectorMedianDistance rejMed;
677     rejMed.setInputCorrespondences(correspondences);
678     rejMed.setMedianFactor(meanDistance);
```

```
679     rejMed.getCorrespondences(*corrRejectMed);
680     return corrRejectMed;
681 }
682
683 void PointCloudManipulator::visualizeCorrespondences(pcl::PointCloud<pcl::
        PointXYZRGB>::Ptr sourcePoints, pcl::PointCloud<pcl::PointXYZRGB>::Ptr
        targetPoints,pcl::PointCloud<pcl::PointXYZRGB>::Ptr sourceKeyPoints, pcl::
        PointCloud<pcl::PointXYZRGB>::Ptr targetKeyPoints, pcl::CorrespondencesPtr
        correspondences, pcl::CorrespondencesPtr goodCorrespondences)
684 {
685     const Eigen::Vector3f translate (0.4, 0.0, 0.0);
686     const Eigen::Quaternionf no_rotation (0, 0, 0, 0);
687     pcl::PointCloud<pcl::PointXYZRGB>::Ptr left (new pcl::PointCloud<pcl::
            PointXYZRGB>);
688     pcl::PointCloud<pcl::PointXYZRGB>::Ptr leftKey (new pcl::PointCloud<pcl::
            PointXYZRGB>);
689     pcl::transformPointCloud (*sourcePoints, *left, -translate, no_rotation);
690     pcl::transformPointCloud (*sourceKeyPoints, *leftKey, -translate,
            no_rotation);
691
692     pcl::visualization::PCLVisualizer vis;
693     int c = 0;
694     int d = 1;
695     vis.createViewPort(0, 0, 0.5, 1, c);
696     vis.createViewPort(0.5, 0, 1, 1, d);
697     pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZRGB> red (
            left, 255, 0, 0);
698     pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZRGB> blue (
            targetPoints, 0, 0, 255);
699     vis.addPointCloud(left, red, "cloud1",c);
700     vis.addPointCloud(targetPoints, blue, "cloud2",c);
701     vis.addCorrespondences<pcl::PointXYZRGB>(leftKey,targetKeyPoints,*
            correspondences,"Correspondences",c);
702     vis.addPointCloud(left, red, "cloud1b",d);
703     vis.addPointCloud(targetPoints, blue, "cloud2b", d);
704     vis.addCorrespondences<pcl::PointXYZRGB>(leftKey,targetKeyPoints,*
            goodCorrespondences, "Good correspondences",d);
705     vis.spin();
706
707 }
708
709 Eigen::Matrix4f PointCloudManipulator::estimateTransformationSVD(pcl::
        PointCloud<pcl::PointXYZRGB>::Ptr sourcePoints, pcl::PointCloud<pcl::
        PointXYZRGB>::Ptr targetPoints, pcl::CorrespondencesPtr correspondences)
710 {
711     Eigen::Matrix4f transResult = Eigen::Matrix4f::Identity ();
712     pcl::registration::TransformationEstimationSVD<pcl::PointXYZRGB, pcl::
            PointXYZRGB> estTransSVD;
713     estTransSVD.estimateRigidTransformation(*sourcePoints, *targetPoints, *
            correspondences, transResult);
714     return transResult;
715 }
716
717 Eigen::Matrix4f PointCloudManipulator::estimateTransformationLM(pcl::PointCloud
        <pcl::PointXYZRGB>::Ptr sourcePoints, pcl::PointCloud<pcl::PointXYZRGB>::
        Ptr targetPoints, pcl::CorrespondencesPtr correspondences)
718 {
```

```
719     Eigen::Matrix4f transResult = Eigen::Matrix4f::Identity ();
720     pcl::registration::TransformationEstimationLM<pcl::PointXYZRGB, pcl::
            PointXYZRGB> estTransLM;
721     estTransLM.estimateRigidTransformation(*sourcePoints, *targetPoints, *
            correspondences, transResult);
722     return transResult;
723 }
724
725 void PointCloudManipulator::visualizeTransformation(pcl::PointCloud<pcl::
        PointXYZRGB>::Ptr sourcePoints, pcl::PointCloud<pcl::PointXYZRGB>::Ptr
        targetPoints,Eigen::Matrix4f transform)
726 {
727     pcl::visualization::PCLVisualizer vis;
728     int a = 0;
729     int b = 1;
730     vis.createViewPort(0, 0, 0.5, 1, a);
731     vis.createViewPort(0.5, 0, 1, 1, b);
732
733     pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZRGB> red (
            sourcePoints, 255, 0, 0);
734     pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZRGB> blue (
            targetPoints, 0, 0, 255);
735     vis.addPointCloud(sourcePoints, red, "source",a);
736     vis.addPointCloud(targetPoints, blue, "target",a);
737
738     vis.addPointCloud(sourcePoints, red, "source2",b);
739     pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmp2 (new pcl::PointCloud<pcl::
            PointXYZRGB>);
740     pcl::transformPointCloud(*targetPoints, *tmp2, transform);
741     vis.addPointCloud(tmp2, blue, "target2", b);
742     vis.spin();
743 }
744
745 pcl::PointCloud<pcl::PointXYZRGB>::Ptr PointCloudManipulator::sampleSTL(QString
        path, int resolution, int tess_level)
746 {
747     pcl::PointCloud<pcl::PointXYZRGB>::Ptr outCloud (new pcl::PointCloud<pcl::
            PointXYZRGB>);
748     pcl::PolygonMesh mesh;
749     pcl::io::loadPolygonFileSTL(path.toStdString(), mesh);
750     pcl::PointCloud<pcl::PointXYZ> scaled_mesh;
751     Eigen::Matrix4f scaleMatrix = Eigen::Matrix4f::Identity();
752     scaleMatrix(0,0)=0.001f;
753     scaleMatrix(1,1)=0.001f;
754     scaleMatrix(2,2)=0.001f;
755     pcl::fromPCLPointCloud2(mesh.cloud,scaled_mesh);
756     pcl::transformPointCloud(scaled_mesh,scaled_mesh,scaleMatrix);
757     pcl::toPCLPointCloud2(scaled_mesh, mesh.cloud);
758     vtkSmartPointer<vtkPolyData> meshVTK;
759     pcl::VTKUtils::convertToVTK(mesh, meshVTK);
760
761     pcl::visualization::PCLVisualizer generator("Generating traces...");
762     generator.addModelFromPolyData (meshVTK, "mesh", 0);
763     std::vector<pcl::PointCloud<pcl::PointXYZ>, Eigen::aligned_allocator<pcl::
            PointCloud<pcl::PointXYZ> > > clouds;
764     std::vector<Eigen::Matrix4f, Eigen::aligned_allocator<Eigen::Matrix4f> >
            poses;
```

```
765     std::vector<float> enthropies;
766     generator.renderViewTesselatedSphere(resolution, resolution, clouds, poses,
            enthropies, tess_level);
767     pcl::PointCloud<pcl::PointXYZ>::Ptr tmp(new pcl::PointCloud<pcl::PointXYZ>)
            ;
768     pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmprgb(new pcl::PointCloud<pcl::
            PointXYZRGB>);
769     for(int i=0; i<clouds.size(); i++){
770         Eigen::Matrix4f tmpPose;
771         tmpPose = poses.at(i).inverse();
772         pcl::transformPointCloud(clouds.at(i), *tmp, tmpPose);
773         pcl::copyPointCloud(*tmp, *tmprgb);
774         *outCloud += *tmprgb;
775     }
776
777     for (int i = 0; i< outCloud->points.size(); i++){
778         outCloud->points[i].r = 255;
779         outCloud->points[i].g = 255;
780         outCloud->points[i].b = 255;
781     }
782     return outCloud;
783 }
784
785 void PointCloudManipulator::matchModelCloud(pcl::PointCloud<pcl::PointXYZRGB>::
        Ptr model, pcl::PointCloud<pcl::PointXYZRGB>::Ptr scene)
786 {
787     pcl::PointCloud<pcl::PointXYZRGB>::Ptr fullScene(new pcl::PointCloud<pcl::
            PointXYZRGB>());
788     *fullScene = *scene;
789
790     Eigen::Matrix4f cameraToTag = Eigen::Matrix4f::Identity();
791     cameraToTag << -0.0324064,   0.999472, 0.00236665,  -0.236723,
792             0.701194,  0.0244224,  -0.712552,  -0.589319,
793             -0.712233, -0.0214317,  -0.701615,    1.82195,
794             0,          0,          0,          1;
795     Eigen::Matrix4f worldToTag = Eigen::Matrix4f::Identity();
796     worldToTag <<  1,  0, 0, -0.084,
797             0,  1, 0, -0.292,
798             0,  0, 1, 0.87,
799             0,  0, 0, 1;
800
801     scene = filterPassThrough(scene, -0.4, 0.4, "x");
802     scene = filterPassThrough(scene, -0.6, -0.04, "y");
803     scene = filterPassThrough(scene, 1.1, 1.8, "z");
804     model = filterVoxel(model, 0.005);
805     scene = filterVoxel(scene, 0.001);
806     scene = extractPlane(scene,0.02);
807
808     PointCloudFeatures modelFeature = computeFeatures(model, "SIFT", "SHOTCOLOR
            ");
809     PointCloudFeatures sceneFeature = computeFeatures(scene, "SIFT", "SHOTCOLOR
            ");
810
811     pcl::CorrespondencesPtr all_correspondences (new pcl::Correspondences);
812     all_correspondences = findCorrespondencesSHOT(modelFeature.
            localDescriptorsSHOTColor, sceneFeature.localDescriptorsSHOTColor);
813     std::cout << "CorrespondenceEstimation correspondences ALL: ";
```

```
814     std::cout << all_correspondences->size() << std::endl;
815
816     pcl::CorrespondencesPtr corrRejectSampleConsensus (new pcl::Correspondences
            );
817     corrRejectSampleConsensus = rejectCorrespondencesSampleConsensus(
            all_correspondences,modelFeature.keyPoints,sceneFeature.keyPoints
            ,0.03,1000); //Was 0.10, 0.07, 0.05
818     std::cout << "Rejected using sample consensus, new amount is : ";
819     std::cout << corrRejectSampleConsensus->size() << std::endl;
820
821     visualizeCorrespondences(modelFeature.points, fullScene, modelFeature.
            keyPoints, sceneFeature.keyPoints, all_correspondences,
            corrRejectSampleConsensus);
822
823     Eigen::Matrix4f transSVD = Eigen::Matrix4f::Identity ();
824     transSVD = estimateTransformationSVD(modelFeature.keyPoints, sceneFeature.
            keyPoints, corrRejectSampleConsensus);
825     std::cout << "Initial transformation CAMERA to OBJECT: " << std::endl;
826     std::cout << transSVD << std::endl;
827     visualizeTransformation(sceneFeature.points, modelFeature.points, transSVD)
            ;
828     visualizeTransformation(fullScene, modelFeature.points, transSVD);
829
830     Eigen::Affine3f A;
831     A = transSVD;
832     Eigen::Affine3f B2;
833     B2 = cameraToTag;
834     pcl::visualization::PCLVisualizer vis;
835     pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZRGB> red (
            modelFeature.points, 255, 0, 0);
836     pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZRGB> blue (
            fullScene, 0, 0, 255);
837     vis.addPointCloud(fullScene, "fullScene");
838     pcl::PointCloud<pcl::PointXYZRGB>::Ptr model2 (new pcl::PointCloud<pcl::
            PointXYZRGB>());
839     pcl::transformPointCloud(*modelFeature.points,*model2,transSVD);
840     vis.addPointCloud(modelFeature.points,red, "model");
841     vis.addPointCloud(model2,red,"model2");
842     vis.spin();
843
844     pcl::IterativeClosestPoint<pcl::PointXYZRGB, pcl::PointXYZRGB> icp;
845     icp.setMaxCorrespondenceDistance(0.01);
846     icp.setMaximumIterations(1000);
847     icp.setTransformationEpsilon(1e-8 );
848     icp.setEuclideanFitnessEpsilon(0.00001);
849     pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmp (new pcl::PointCloud<pcl::
            PointXYZRGB>());
850     Eigen::Matrix4f aids = transSVD;
851     pcl::transformPointCloud(*modelFeature.points,*tmp,aids);
852     icp.setInputSource(tmp);
853     icp.setInputTarget(sceneFeature.points);
854     pcl::PointCloud<pcl::PointXYZRGB>::Ptr outCloud (new pcl::PointCloud<pcl::
            PointXYZRGB>());
855     icp.align(*outCloud);
856
857     Eigen::Matrix4f final;
858     if(icp.hasConverged()){
```

```
859             std::cout << "Converged! \n";
860             final = icp.getFinalTransformation();
861         }
862         final = final*transSVD;
863 
864         std::cout << "FINAL transformation CAMERA to OBJECT: " << std::endl;
865         std::cout << final << std::endl;
866 
867         std::cout << "CAMERA TO TAG MATRIX: " << std::endl;
868         std::cout << cameraToTag << std::endl;
869 
870         Eigen::Matrix4f initialTable = cameraToTag.inverse()*transSVD;
871         std::cout << "Initial table transform: " << std::endl;
872         std::cout << initialTable << std::endl;
873         Eigen::Matrix4f finalTable = cameraToTag.inverse()*final;
874         std::cout << "Final table transform: " << std::endl;
875         std::cout << finalTable << std::endl;
876 
877         vis.addPointCloud(tmp, "modelMoved");
878         vis.addPointCloud(outCloud, "modelRefined");
879 
880         pcl::PointCloud<pcl::PointXYZRGB>::Ptr finalPointCloud (new pcl::PointCloud
                <pcl::PointXYZRGB>());
881         pcl::transformPointCloud(*modelFeature.points,*finalPointCloud,final);
882         vis.addPointCloud(finalPointCloud, "WHAT");
883 
884         Eigen::Matrix4f test = cameraToTag*worldToTag.inverse();
885         Eigen::Affine3f C;
886         C = test;
887         vis.addCoordinateSystem(0.5, C);
888         vis.spin();
889 
890         std::cout << "World coordinates" << std::endl;
891         Eigen::Matrix4f posInWorld = worldToTag*finalTable;
892         std::cout << posInWorld << std::endl;
893 
894         Eigen::Affine3f A2;
895         float x, y, z, roll, pitch, yaw;
896         A2 = posInWorld;
897         pcl::getTranslationAndEulerAngles(A2, x, y, z, roll, pitch, yaw);
898         std::cout << "X: " << x << ", Y: " << y << ", Z: " << z << std::endl;
899         std::cout << "Roll: " << roll*(180.0/3.14) << ", Pitch: " << pitch
                *(180.0/3.14) << ", yaw: " << yaw*(180.0/3.14) << std::endl;
900 
901         Eigen::Affine3f finalCoordinates;
902         finalCoordinates = final;
903         visualizer.reset(new pcl::visualization::PCLVisualizer ("viewer2", false));
904         visualizer->addCoordinateSystem(0.2, B2);
905         visualizer->addCoordinateSystem(0.2, finalCoordinates);
906         visualizer->addCoordinateSystem(0.2, C);
907         visualizer->addPointCloud(fullScene, "fullScene");
908         visualizer->addPointCloud(heihoo, "model");
909 
910         Q_EMIT sendNewVisualizer(visualizer);
911 
912         pcl::visualization::PCLVisualizer tmpVis;
913         pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZRGB> red2 (
```

```
        heihoo, 255, 0, 0);
914     tmpVis.addPointCloud(heihoo,red2, "model");
915     tmpVis.addPointCloud(fullScene, "fullScene");
916     tmpVis.spin();
917 }
918
919 void PointCloudManipulator::alignAndMatch(std::vector<pcl::PointCloud<pcl::
        PointXYZRGB>::Ptr> clouds)
920 {
921     pcl::PointCloud<pcl::PointXYZRGB>::Ptr scene(new pcl::PointCloud<pcl::
            PointXYZRGB>());
922     pcl::PointCloud<pcl::PointXYZRGB>::Ptr model(new pcl::PointCloud<pcl::
            PointXYZRGB>());
923     scene = alignCloudsRefined(clouds);
924     std::cout << clouds.size() << std::endl;
925     *model = *clouds.at(clouds.size()-1);
926     pcl::visualization::PCLVisualizer vis;
927     vis.addPointCloud(scene, "scene");
928     vis.addPointCloud(model, "model");
929     vis.spin();
930     matchModelCloud(model, scene);
931 }
932
933 pcl::PointCloud<pcl::PointXYZRGB>::Ptr PointCloudManipulator::
        alignCloudsRefined(std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr>
        cloudsIn)
934 {
935     std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr> clouds;
936     std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr> cloudsOriginal;
937     std::vector<Eigen::Matrix4f> cameraPositions;
938     Eigen::Matrix4f cam0 = Eigen::Matrix4f::Identity();
939     cameraPositions.push_back(cam0);
940     Eigen::Matrix4f cam1 = Eigen::Matrix4f::Identity();
941     // This is the matrix from NUC2 (camera2) to table
942     cam1 << -0.0369295,   -0.99916 , 0.0177825,   0.174928,
943             -0.592998, 0.00758757 , -0.805168  , 0.016518,
944             0.804357, -0.0402794 ,  -0.59278  , 0.945164,
945             0,          0 ,         0   ,       1;
946     cameraPositions.push_back(cam1);
947     Eigen::Matrix4f cam2 = Eigen::Matrix4f::Identity();
948     // This is the matrix from PC (camera3) to table
949     cam2 <<   -0.999718 , -0.0224201 ,-0.00776418 ,   0.604796,
950             -0.00404688 ,   0.483571  , -0.875296 ,  -0.256756,
951             0.0233787 ,  -0.875018  , -0.483525  ,   2.14104,
952             0 ,          0    ,       0     ,      1;
953     cameraPositions.push_back(cam2);
954     // This is the matrix from NUC1 (camera1) to table
955     Eigen::Matrix4f cam3 = Eigen::Matrix4f::Identity();
956     cam3 << -0.0324064,   0.999472, 0.00236665,  -0.236723,
957             0.701194,  0.0244224,  -0.712552,  -0.589319,
958             -0.712233, -0.0214317,  -0.701615,    1.82195,
959             0,          0,          0,          1;
960
961     cameraPositions.push_back(cam3);
962
963
964     for(int i = 0; i<cloudsIn.size()-1; i++){
```

```
965         pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmpCloud (new pcl::PointCloud<
                pcl::PointXYZRGB>());
966         std::cout << "Loop nr: " << i << std::endl;
967         *tmpCloud = *cloudsIn.at(i);
968 
969         cloudsOriginal.push_back(tmpCloud);
970         tmpCloud = filterVoxel(tmpCloud, 0.01);
971 
972         switch(i){
973         case 0:
974             tmpCloud = filterPassThrough(tmpCloud, -0.4, 0.39, "x");
975             tmpCloud = filterPassThrough(tmpCloud, -1.1, -0.1, "y");
976             break;
977         case 1:
978             tmpCloud = filterPassThrough(tmpCloud, -0.5, 0.3, "x");
979             tmpCloud = filterPassThrough(tmpCloud, -0.7, 0.3, "y");
980             break;
981         case 2:
982             tmpCloud = filterPassThrough(tmpCloud, -0.5, 0.39, "x");
983             tmpCloud = filterPassThrough(tmpCloud, -0.4, 0.1, "y");
984             tmpCloud = filterPassThrough(tmpCloud, 0.8, 3.1, "z");
985             break;
986         }
987 
988         tmpCloud = filterVoxel(tmpCloud, 0.001);
989         tmpCloud = extractPlane(tmpCloud,0.02);
990 
991         clouds.push_back(tmpCloud);
992     }
993 
994     std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr> roughClouds;
995     roughClouds.push_back(clouds.at(0));
996     std::vector<Eigen::Matrix4f> cameraPositions2;
997     Eigen::Matrix4f tmp = Eigen::Matrix4f::Identity();
998     cameraPositions2.push_back(tmp);
999     for(int i=1; i<clouds.size(); i++){
1000         pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmp (new pcl::PointCloud<pcl::
                PointXYZRGB>());
1001         Eigen::Matrix4f tmpMat2 = cameraPositions.at(i);
1002         Eigen::Matrix4f tmpMat3 = cameraPositions.at(3);
1003         Eigen::Matrix4f final = tmpMat3*tmpMat2.inverse();
1004         cameraPositions2.push_back(final);
1005         pcl::transformPointCloud(*clouds.at(i),*tmp,final);
1006         roughClouds.push_back(tmp);
1007     }
1008 
1009     pcl::IterativeClosestPoint<pcl::PointXYZRGB, pcl::PointXYZRGB> icp;
1010     icp.setMaxCorrespondenceDistance(0.03);
1011     icp.setMaximumIterations(1000);
1012     icp.setTransformationEpsilon(1e-10);
1013     icp.setEuclideanFitnessEpsilon(0.0000001);
1014     icp.setInputTarget(roughClouds.at(0));
1015 
1016     for(int i=1; i<roughClouds.size(); i++){
1017         icp.setInputSource(roughClouds.at(i));
1018         pcl::PointCloud<pcl::PointXYZRGB>::Ptr outCloud (new pcl::PointCloud<
                pcl::PointXYZRGB>());
```

```
1019            icp.align(*outCloud);
1020            Eigen::Matrix4f transICP = Eigen::Matrix4f::Identity();
1021            if(icp.hasConverged()){
1022                std::cout << "Converged! ";
1023                std::cout << i << std::endl;
1024                transICP = icp.getFinalTransformation();
1025                cameraPositions2[i] = transICP*cameraPositions2[i];
1026
1027            }
1028        }
1029
1030        pcl::PointCloud<pcl::PointXYZRGB>::Ptr writeToFileCloud (new pcl::
                PointCloud<pcl::PointXYZRGB>());
1031        for(int i=0; i<roughClouds.size(); i++){
1032            pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmp (new pcl::PointCloud<pcl::
                    PointXYZRGB>());
1033            pcl::transformPointCloud(*cloudsOriginal[i], *tmp, cameraPositions2[i])
                    ;
1034            *writeToFileCloud += *tmp;
1035        }
1036        pcl::io::savePCDFileBinary("/home/minions/alignedWithObject.pcd", *
                writeToFileCloud);
1037        return (writeToFileCloud);
1038    }
1039
1040
1041    void PointCloudManipulator::alignClouds(QStringList fileNames)
1042    {
1043        std::vector<PointCloudFeatures> pointClouds;
1044
1045        for(int i = 0; i<fileNames.size(); i++){
1046            pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmpCloud (new pcl::PointCloud<
                    pcl::PointXYZRGB>());
1047            std::cout << fileNames.at(i).toStdString() << std::endl;
1048            pcl::io::loadPCDFile(fileNames.at(i).toUtf8().constData(), *tmpCloud);
1049            tmpCloud = filterPassThrough(tmpCloud, 0.0, 2.3, "z");
1050            tmpCloud = filterPassThrough(tmpCloud, -0.5, 1.0, "y");
1051            tmpCloud = filterPassThrough(tmpCloud, -1.0, 1.0, "x");
1052            tmpCloud = filterVoxel(tmpCloud, 0.01);
1053            PointCloudFeatures tmpFeature = computeFeatures(tmpCloud, "SIFT", "FPFH
                    ");
1054            pointClouds.push_back(tmpFeature);
1055        }
1056
1057        pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmpAligned (new pcl::PointCloud<pcl
                ::PointXYZRGB>());
1058        pcl::PointCloud<pcl::PointXYZRGB>::Ptr icpCloud (new pcl::PointCloud<pcl::
                PointXYZRGB>());
1059        *tmpAligned = *pointClouds.at(0).points;
1060        *icpCloud = *pointClouds.at(0).points;
1061
1062        pcl::IterativeClosestPoint<pcl::PointXYZRGB, pcl::PointXYZRGB> icp;
1063        icp.setMaxCorrespondenceDistance(0.01);
1064        icp.setMaximumIterations(10000);
1065        icp.setRANSACIterations(0);
1066        icp.setTransformationEpsilon(1e-8 );
1067        icp.setEuclideanFitnessEpsilon(0.00001);
```

```
1068        Eigen::Matrix4f prevTrans = Eigen::Matrix4f::Identity();
1069
1070        for(int k = 0; k<pointClouds.size()-1; k++){
1071            pcl::CorrespondencesPtr all_correspondences (new pcl::Correspondences);
1072            all_correspondences = findCorrespondences(pointClouds.at(k).
                    localDescriptorsFPFH, pointClouds.at(k+1).localDescriptorsFPFH);
1073            std::cout << "CorrespondenceEstimation correspondences ALL: ";
1074            std::cout << all_correspondences->size() << std::endl;
1075
1076            pcl::CorrespondencesPtr corrRejectSampleConsensus (new pcl::
                    Correspondences);
1077            corrRejectSampleConsensus = rejectCorrespondencesSampleConsensus(
                    all_correspondences,pointClouds.at(k).keyPoints,pointClouds.at(k+1)
                    .keyPoints,0.25,1000);
1078            std::cout << "Rejected using sample consensus, new amount is : ";
1079            std::cout << corrRejectSampleConsensus->size() << std::endl;
1080            visualizeCorrespondences(pointClouds.at(k).points, pointClouds.at(k+1).
                    points, pointClouds.at(k).keyPoints, pointClouds.at(k+1).keyPoints,
                     all_correspondences, corrRejectSampleConsensus);
1081
1082            Eigen::Matrix4f transSVD = Eigen::Matrix4f::Identity ();
1083            transSVD = estimateTransformationSVD(pointClouds.at(k).keyPoints,
                    pointClouds.at(k+1).keyPoints, corrRejectSampleConsensus);
1084            std::cout << transSVD << std::endl;
1085            visualizeTransformation(pointClouds.at(k+1).points, pointClouds.at(k).
                    points, transSVD);
1086            pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmp (new pcl::PointCloud<pcl::
                    PointXYZRGB>());
1087            Eigen::Matrix4f trans = prevTrans*transSVD.inverse();
1088            pcl::transformPointCloud(*pointClouds.at(k+1).points,*tmp,trans);
1089
1090            icp.setInputSource(tmp);
1091            icp.setInputTarget(pointClouds.at(k).points);
1092            pcl::PointCloud<pcl::PointXYZRGB>::Ptr outCloud (new pcl::PointCloud<
                    pcl::PointXYZRGB>());
1093            icp.align(*outCloud);
1094
1095            if(icp.hasConverged()){
1096                *icpCloud = *icpCloud + *outCloud;
1097                std::cout << "Converged! ";
1098                std::cout << k << std::endl;
1099                std::cout << icp.getFinalTransformation() << std::endl;
1100            }
1101            *tmpAligned = *tmpAligned + *tmp;
1102            prevTrans = trans;
1103        }
1104
1105        pcl::visualization::PCLVisualizer vis;
1106        vis.addPointCloud(tmpAligned, "alignedcloud");
1107        vis.spin();
1108
1109        pcl::visualization::PCLVisualizer vis2;
1110        vis2.addPointCloud(icpCloud, "icpalignedcloud");
1111        vis2.spin();
1112    }
1113
1114    void PointCloudManipulator::alignRobotCell(QStringList fileNames)
```

```
1115 {
1116     std::vector<PointCloudFeatures> pointClouds;
1117     std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr> originalClouds;
1118     std::vector<Eigen::Matrix4f> cameraPositions;
1119     cameraPositions.push_back(Eigen::Matrix4f::Identity());
1120 
1121     for(int i = 0; i<fileNames.size(); i++){
1122         pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmpCloud (new pcl::PointCloud<
                pcl::PointXYZRGB>());
1123         std::cout << fileNames.at(i).toStdString() << std::endl;
1124         pcl::io::loadPCDFile(fileNames.at(i).toUtf8().constData(), *tmpCloud);
1125         originalClouds.push_back(tmpCloud);
1126 
1127         switch(i){
1128         case 0:
1129             tmpCloud = filterPassThrough(tmpCloud, -0.8, 0.8, "x");
1130             tmpCloud = filterPassThrough(tmpCloud, -0.4, 0.3, "y");
1131             tmpCloud = filterPassThrough(tmpCloud, 1.0, 1.7, "z");
1132             break;
1133         case 1:
1134             tmpCloud = filterPassThrough(tmpCloud, -0.4, 0.5, "x");
1135             tmpCloud = filterPassThrough(tmpCloud, -1.0, 0.1, "y");
1136             tmpCloud = filterPassThrough(tmpCloud, 1.3, 2.6, "z");
1137             break;
1138         case 2:
1139             tmpCloud = filterPassThrough(tmpCloud, -0.6, 0.3, "x");
1140             tmpCloud = filterPassThrough(tmpCloud, -0.6, 0.4, "y");
1141             tmpCloud = filterPassThrough(tmpCloud, 1.2, 2.6, "z");
1142             break;
1143         }
1144         tmpCloud = filterVoxel(tmpCloud, 0.01);
1145         tmpCloud = extractPlaneReturnPlane(tmpCloud, 0.1);
1146 
1147         PointCloudFeatures tmpFeature = computeFeatures(tmpCloud, "SIFT", "FPFH
                ");
1148         pointClouds.push_back(tmpFeature);
1149     }
1150 
1151     Eigen::Matrix4f prevTrans = Eigen::Matrix4f::Identity();
1152     for(int k = 0; k<pointClouds.size()-1; k++){
1153         pcl::CorrespondencesPtr all_correspondences (new pcl::Correspondences);
1154         all_correspondences = findCorrespondences(pointClouds.at(k).
                localDescriptorsFPFH, pointClouds.at(k+1).localDescriptorsFPFH);
1155         std::cout << "CorrespondenceEstimation correspondences ALL: ";
1156         std::cout << all_correspondences->size() << std::endl;
1157 
1158         pcl::CorrespondencesPtr corrRejectSampleConsensus (new pcl::
                Correspondences);
1159         corrRejectSampleConsensus = rejectCorrespondencesSampleConsensus(
                all_correspondences,pointClouds.at(k).keyPoints,pointClouds.at(k+1)
                .keyPoints,0.25,1000);
1160         std::cout << "Rejected using sample consensus, new amount is : ";
1161         std::cout << corrRejectSampleConsensus->size() << std::endl;
1162         visualizeCorrespondences(pointClouds.at(k).points, pointClouds.at(k+1).
                points, pointClouds.at(k).keyPoints, pointClouds.at(k+1).keyPoints,
                 all_correspondences, corrRejectSampleConsensus);
1163 
```

```
1164            Eigen::Matrix4f transSVD = Eigen::Matrix4f::Identity ();
1165            transSVD = estimateTransformationSVD(pointClouds.at(k).keyPoints,
                    pointClouds.at(k+1).keyPoints, corrRejectSampleConsensus);
1166            std::cout << transSVD << std::endl;
1167            visualizeTransformation(pointClouds.at(k+1).points, pointClouds.at(k).
                    points, transSVD);
1168            pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmp (new pcl::PointCloud<pcl::
                    PointXYZRGB>());
1169            Eigen::Matrix4f trans = prevTrans*transSVD.inverse();
1170            pcl::transformPointCloud(*pointClouds.at(k+1).points,*tmp,trans);
1171        }
1172
1173        pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmpAligned (new pcl::PointCloud<pcl
                ::PointXYZRGB>());
1174        pcl::PointCloud<pcl::PointXYZRGB>::Ptr icpCloud (new pcl::PointCloud<pcl::
                PointXYZRGB>());
1175        *tmpAligned = *pointClouds.at(0).points;
1176        *icpCloud = *pointClouds.at(0).points;
1177        pcl::IterativeClosestPoint<pcl::PointXYZRGB, pcl::PointXYZRGB> icp;
1178        icp.setMaxCorrespondenceDistance(0.3);
1179        icp.setMaximumIterations(10000);
1180        icp.setTransformationEpsilon(1e-10);
1181        icp.setEuclideanFitnessEpsilon(0.0000001);
1182
1183        for(int k = 0; k<pointClouds.size()-1; k++){
1184            Eigen::Matrix4f initTrans = Eigen::Matrix4f::Identity ();
1185            Eigen::Matrix4f cameraPos = Eigen::Matrix4f::Identity();
1186            initTrans = computeInitialAlignmentFPFH(pointClouds.at(0).keyPoints,
                    pointClouds.at(0).localDescriptorsFPFH,pointClouds.at(k+1).
                    keyPoints, pointClouds.at(k+1).localDescriptorsFPFH,0.08,1.0,1000);
1187            std::cout << initTrans << std::endl;
1188            visualizeTransformation(pointClouds.at(k+1).points, pointClouds.at(0).
                    points, initTrans);
1189
1190            pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmp (new pcl::PointCloud<pcl::
                    PointXYZRGB>());
1191            Eigen::Matrix4f initTransInv = initTrans.inverse();
1192            pcl::transformPointCloud(*pointClouds.at(k+1).points,*tmp,initTransInv)
                    ;
1193
1194            *tmpAligned = *tmpAligned + *tmp;
1195            cameraPos = initTrans.inverse();
1196            icp.setInputSource(tmp);
1197            icp.setInputTarget(pointClouds.at(0).points);
1198            pcl::PointCloud<pcl::PointXYZRGB>::Ptr outCloud (new pcl::PointCloud<
                    pcl::PointXYZRGB>());
1199            icp.align(*outCloud);
1200            Eigen::Matrix4f transICP = Eigen::Matrix4f::Identity();
1201            if(icp.hasConverged()){
1202                *icpCloud = *icpCloud + *outCloud;
1203                std::cout << "Converged! ";
1204                std::cout << k << std::endl;
1205                transICP = icp.getFinalTransformation();
1206                cameraPos = transICP*cameraPos;
1207                cameraPositions.push_back(cameraPos);
1208            }
1209        }
```

```
1210
1211        pcl::visualization::PCLVisualizer vis;
1212        vis.addPointCloud(tmpAligned, "alignedcloud");
1213        vis.spin();
1214
1215        pcl::visualization::PCLVisualizer vis2;
1216        vis2.addPointCloud(icpCloud, "icpalignedcloud");
1217        vis2.spin();
1218
1219        pcl::visualization::PCLVisualizer vis3;
1220        pcl::PointCloud<pcl::PointXYZRGB>::Ptr originalAligned (new pcl::PointCloud
                <pcl::PointXYZRGB>());
1221        for(int i = 0; i<3; i++){
1222            pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmp (new pcl::PointCloud<pcl::
                    PointXYZRGB>());
1223            QString s = "cloud";
1224            s.append(QString::number(i));
1225            pcl::transformPointCloud(*originalClouds.at(i),*tmp,cameraPositions.at(
                    i));
1226            std::cout << cameraPositions.at(i) << std::endl;
1227            vis3.addPointCloud(tmp,s.toStdString());
1228            *originalAligned += *tmp;
1229        }
1230        vis3.addCoordinateSystem(0.5);
1231        Eigen::Affine3f A;
1232        A = cameraPositions.at(1);
1233        vis3.addCoordinateSystem(0.5, A);
1234        std::cout << "Cam1: " << std::endl;
1235        std::cout <<  cameraPositions.at(1) << std::endl;
1236        A = cameraPositions.at(2);
1237        std::cout << "Cam2: " << std::endl;
1238        std::cout <<  cameraPositions.at(2) << std::endl;
1239        vis3.addCoordinateSystem(0.5, A);
1240        vis3.spin();
1241        pcl::io::savePCDFileBinary("/home/minions/aligned.pcd", *originalAligned);
1242    }
1243
1244    void PointCloudManipulator::refineAlignment(QStringList fileNames)
1245    {
1246        std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr> clouds;
1247        std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr> cloudsOriginal;
1248        std::vector<Eigen::Matrix4f> cameraPositions;
1249        Eigen::Matrix4f cam0 = Eigen::Matrix4f::Identity();
1250        cameraPositions.push_back(cam0);
1251        Eigen::Matrix4f cam1 = Eigen::Matrix4f::Identity();
1252        // This is the matrix from NUC2 (cam2) to table
1253        cam1 << -0.0369295,    -0.99916 , 0.0177825,    0.174928,
1254                -0.592998, 0.00758757 , -0.805168   , 0.016518,
1255                0.804357, -0.0402794 ,  -0.59278   , 0.945164,
1256                0,          0 ,         0    ,       1;
1257        cameraPositions.push_back(cam1);
1258        Eigen::Matrix4f cam2 = Eigen::Matrix4f::Identity();
1259        // This is the matrix from PC (cam3) to table
1260        cam2 <<    -0.999718 , -0.0224201 ,-0.00776418 ,    0.604796,
1261                -0.00404688 ,    0.483571   , -0.875296 ,   -0.256756,
1262                0.0233787 ,   -0.875018   , -0.483525   ,    2.14104,
1263                0 ,          0     ,       0      ,      1;
```

```
1264        cameraPositions.push_back(cam2);
1265        // This is the matrix from NUC1 (cam1) to table
1266        Eigen::Matrix4f cam3 = Eigen::Matrix4f::Identity();
1267        cam3 << -0.0324064,   0.999472, 0.00236665,  -0.236723,
1268                0.701194,  0.0244224,  -0.712552,  -0.589319,
1269                -0.712233, -0.0214317,  -0.701615,    1.82195,
1270                0,          0,          0,          1;
1271        cameraPositions.push_back(cam3);
1272
1273        for(int i = 0; i<fileNames.size(); i++){
1274            pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmpCloud (new pcl::PointCloud<
                pcl::PointXYZRGB>());
1275            std::cout << fileNames.at(i).toStdString() << std::endl;
1276            pcl::io::loadPCDFile(fileNames.at(i).toUtf8().constData(), *tmpCloud);
1277
1278            cloudsOriginal.push_back(tmpCloud);
1279            tmpCloud = filterVoxel(tmpCloud, 0.01);
1280
1281            switch(i){
1282            case 0:
1283                tmpCloud = filterPassThrough(tmpCloud, -0.4, 0.39, "x");
1284                tmpCloud = filterPassThrough(tmpCloud, -1.1, -0.1, "y");
1285                break;
1286            case 1:
1287                tmpCloud = filterPassThrough(tmpCloud, -0.5, 0.3, "x");
1288                tmpCloud = filterPassThrough(tmpCloud, -0.7, 0.3, "y");
1289                break;
1290            case 2:
1291                tmpCloud = filterPassThrough(tmpCloud, -0.5, 0.39, "x");
1292                tmpCloud = filterPassThrough(tmpCloud, -0.4, 0.1, "y");
1293                tmpCloud = filterPassThrough(tmpCloud, 0.8, 3.1, "z");
1294                break;
1295            }
1296            tmpCloud = filterVoxel(tmpCloud, 0.001);
1297            tmpCloud = extractPlane(tmpCloud,0.02);
1298            clouds.push_back(tmpCloud);
1299        }
1300
1301        pcl::visualization::PCLVisualizer vis1;
1302        for(int i=0; i<clouds.size(); i++){
1303            QString s = "cloud";
1304            s.append(QString::number(i));
1305            vis1.addPointCloud(clouds.at(i),s.toStdString());
1306        }
1307        vis1.spin();
1308
1309        pcl::visualization::PCLVisualizer vis2;
1310        std::vector<pcl::PointCloud<pcl::PointXYZRGB>::Ptr> roughClouds;
1311        pcl::PointCloud<pcl::PointXYZRGB>::Ptr initAlignedSave (new pcl::PointCloud
            <pcl::PointXYZRGB>());
1312        roughClouds.push_back(clouds.at(0));
1313        *initAlignedSave = *cloudsOriginal.at(0);
1314        std::vector<Eigen::Matrix4f> cameraPositions2;
1315        Eigen::Matrix4f tmp = Eigen::Matrix4f::Identity();
1316        cameraPositions2.push_back(tmp);
1317        vis2.addPointCloud(clouds.at(0), "cloud0");
1318        for(int i=1; i<clouds.size(); i++){
```

```
1319            pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmp (new pcl::PointCloud<pcl::
                    PointXYZRGB>());
1320            QString s = "cloud";
1321            s.append(QString::number(i));
1322            Eigen::Matrix4f tmpMat2 = cameraPositions.at(i);
1323            Eigen::Matrix4f tmpMat3 = cameraPositions.at(3);
1324            Eigen::Matrix4f final = tmpMat3*tmpMat2.inverse();
1325            cameraPositions2.push_back(final);
1326            pcl::transformPointCloud(*clouds.at(i),*tmp,final);
1327            roughClouds.push_back(tmp);
1328            vis2.addPointCloud(tmp,s.toStdString());
1329
1330            pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmp2 (new pcl::PointCloud<pcl::
                    PointXYZRGB>());
1331            pcl::transformPointCloud(*cloudsOriginal.at(i), *tmp2, final);
1332            *initAlignedSave += *tmp2;
1333        }
1334        pcl::io::savePCDFileBinary("/home/minions/initialAlignedWithObject.pcd", *
                initAlignedSave);
1335        vis2.spin();
1336
1337        pcl::IterativeClosestPoint<pcl::PointXYZRGB, pcl::PointXYZRGB> icp;
1338        icp.setMaxCorrespondenceDistance(0.03); // 0.05
1339        icp.setMaximumIterations(1000);
1340        icp.setTransformationEpsilon(1e-10);
1341        icp.setEuclideanFitnessEpsilon(0.0000001);
1342        icp.setInputTarget(roughClouds.at(0));
1343        pcl::PointCloud<pcl::PointXYZRGB>::Ptr icpCloud (new pcl::PointCloud<pcl::
                PointXYZRGB>());
1344        *icpCloud += *roughClouds.at(0);
1345
1346        for(int i=1; i<roughClouds.size(); i++){
1347            icp.setInputSource(roughClouds.at(i));
1348            pcl::PointCloud<pcl::PointXYZRGB>::Ptr outCloud (new pcl::PointCloud<
                    pcl::PointXYZRGB>());
1349            icp.align(*outCloud);
1350            Eigen::Matrix4f transICP = Eigen::Matrix4f::Identity();
1351            if(icp.hasConverged()){
1352                *icpCloud = *icpCloud + *outCloud;
1353                std::cout << "Converged! ";
1354                std::cout << i << std::endl;
1355                transICP = icp.getFinalTransformation();
1356                cameraPositions2[i] = transICP*cameraPositions2[i];
1357            }
1358        }
1359
1360        pcl::visualization::PCLVisualizer vis3;
1361        vis3.addPointCloud(icpCloud, "icp");
1362        vis3.spin();
1363
1364        pcl::visualization::PCLVisualizer vis4;
1365        pcl::PointCloud<pcl::PointXYZRGB>::Ptr writeToFileCloud (new pcl::
                PointCloud<pcl::PointXYZRGB>());
1366        for(int i=0; i<roughClouds.size(); i++){
1367            Eigen::Affine3f A;
1368            A = cameraPositions2[i];
1369            vis4.addCoordinateSystem(0.5, A);
```

```
1370            QString s = "cloud";
1371            s.append(QString::number(i));
1372            pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmp (new pcl::PointCloud<pcl::
                    PointXYZRGB>());
1373            pcl::transformPointCloud(*cloudsOriginal[i], *tmp, cameraPositions2[i])
                    ;
1374            vis4.addPointCloud(tmp, s.toStdString());
1375            *writeToFileCloud += *tmp;
1376        }
1377        vis4.spin();
1378    }
1379
1380    }
```

Appendix D: Digital Appendix

A *.zip* file is included as the digital appendix. The file contains:

- Source code for the calibration of extrinsic parameters program.
- Source code for the main program.