Norwegian University of
Science and Technology

Primitive Shape Detection in Point Clouds

Aksel Sveier

Master of Science in Mechanical Engineering
Submission date: June 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

Aksel Sveier

Tittel: Deteksjon av geometriske former i punktskyer

Tittel (engelsk): Primitive shape detection in point clouds

Oppgavens tekst:
3D kamerasyn er blitt praktisk gjennomførbart med nye kamerasystemer som Microsoft Kinect, som gir tilgang på punktskyer med høy tetthet. I denne oppgaven skal bruk av robuste algoritmer som RANSAC studeres for deteksjon av geometriske former i punktskyer.

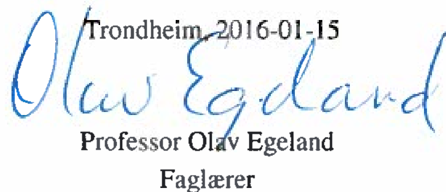
1. Beskriv teknologien bak 3D kameraer og metoder for punktsky prosessering.
2. Presenter RANSAC-algoritmen og andre robuste estimeringsmetoder.
3. Vurder hvordan geometrisk algebra kan brukes for å modellere stilistiske objekter
4. Implementer RANSAC for deteksjon av stilistiske objekter i punktskyer med geometrisk algebra.
5. Vurder metoder for å forbedre RANSAC.
6. Lag programvare for punktsky prosessering og deteksjon av stilistiske objekter
7. Prøv ut systemet i eksperimenter og presenter resultater.
8. Demonstrer mulige applikasjoner i en robotisert plukk-og-plasser operasjon.

Oppgave utlevert: 2016-01-15

Innlevering: 2016-06-10

Utført ved Institutt for produksjons- og kvalitetsteknikk

Trondheim, 2016-01-15



Professor Olav Egeland
Faglærer

Preface

This is the concluding Master's thesis of the study program in Mechanical Engineering at NTNU, Trondheim. The work was carried out between January and June 2016.

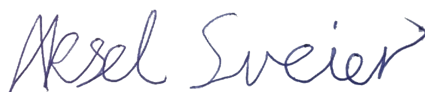
During my specialisation in Production Systems I was introduced to robotic- and computer vision systems. These fields captured my interest because of the many existing and potential applications in industry. After working with 3D cameras and point clouds in my specialization project, it was natural for me to further examine these subjects in this thesis. As a result, the topic of *Primitive Shape Detection in Point Clouds* were formulated in collaboration with my supervising professor Olav Egeland.

I would like to thank my supervisor Olav Egeland for supporting me with this thesis and guiding me along the way. Both the practical and theoretical aspects of this thesis have been very educational and I am grateful for the opportunity to implement experiments in the robotics laboratory at IPK.

Furthermore, I would like to thank PhD candidate Lars Tingelstad for his useful comments and helping me with the software setup. His assistance in the laboratory has also been appreciated.

Lastly, a special thanks to all my fellow students at the "office" for good collaboration and a social working environment.

Trondheim, June 10, 2016



Aksel Sveier

Abstract

Industrial processes often involves handling of objects and surfaces shaped like geometric primitives. This should be taken into consideration when designing computer vision-based systems for such processes. Both pose and parameters of geometric primitives can be established with 3D cameras and simple algorithms. In this thesis, robust estimation algorithms have been considered to detect primitive shapes in point clouds from 3D cameras. The primitives are described with conformal geometric algebra. Possible applications have been suggested and demonstrated through a robotic pick-and-place task solved with data from a 3D camera.

A primitive shape detection algorithm is implemented in a C++ based software. The algorithm is implemented for planes, spheres and cylinders. Results show that the algorithm is able to detect the shapes in data sets containing up to 90% outliers. Furthermore, a real-time tracking algorithm based on the primitive shape detection algorithm is implemented to track primitives in a real-time data stream from a 3D camera. The run-time of the tracking algorithm is well below the required rate for a 60 frames per second data stream. A multiple shape detection algorithm is also developed. The goal is to detect multiple shapes in a point cloud with a single run of the algorithm. The algorithm is implemented for spheres and results show that multiple spheres can be successfully detected in a point cloud.

The accuracy and efficiency of the algorithms is demonstrated in a robotic pick-and-place task. Primitive objects are detected in the robot workspace. These objects are used for robot-camera calibration and data cropping, in addition to the pick-and-place operation. The demonstration show that the algorithms are effective for high accuracy demanding industrial tasks, given raw data from consumer grade 3D cameras.

Sammendrag

Industrielle prosesser involverer ofte håndtering av objekter og overflater formet som stilistiske geometrier. Dette bør tas hensyn til ved design av datasyn-baserte systemer for slike prosesser. Posisjon, orientasjon og parametere av stilistiske geometrier kan identifiseres ved bruk av 3D kameraer og enkle algoritmer. I denne avhandlingen er robuste estimeringsalgoritmer anvendt for deteksjon av geometriske former i punktskyer. De geometriske formene er beskrevet med konform geometrisk algebra. Mulige applikasjoner er blitt foreslått og demonstrert gjennom en robotisert plukk-og-plasser oppgave som er løst med et 3D kamera.

En algoritme for deteksjon av geometriske former er implementert i et C++ basert program. Algoritmen er implementert for plan, kuler og sylindere. Resultatene viser at algoritmen kan detektere geometrier i datasett med opptil 90% støy. I tillegg er en sporingsalgoritme implementert for sporing av geometriske former i sanntid med et 3D kamera. Sporingsalgoritmen yter mer enn raskt nok til å utføre sporing i en bildestrøm på 60 bilder per sekund. En algoritme for deteksjon av flere geometriske former er også utviklet. Målet med denne algoritmen er å detektere alle geometriske former i en punktsky ved å utføre algoritmen én gang. Algoritmen er implementer for kuler og resultatene viser at flere kuler kan detekteres i en punktsky fra et 3D kamera.

De implementerte algoritmene er demonstrert i en robotisert plukk-og-plasser oppgaven. Geometriske former er detektert i robotens arbeidsrom. De detekterte formene brukes til robot-kamera kalibrering og segmentering av data, i tillegg til selve plukk-og-plasser operasjonen. Demonstrasjonen viser at algoritmene er effektive for industrielle oppgaver som krever høy presisjon, gitt rådata fra et lavkost forbrukerklasse 3D kamera.

Table of Contents

Preface v

Abstract vii

Sammendrag ix

Table of Contents xiii

List of Tables xv

List of Figures xviii

Abbreviations xix

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Objectives . . . 2
- 1.3 Approach . . . 2
- 1.4 Structure of the Report . . . 4

2 Literature Review 5
- 2.1 Introduction . . . 5
- 2.2 Robust Estimators for Computer Vision . . . 6
- 2.3 Random Sample Consensus (RANSAC) . . . 7
- 2.4 RANSAC Based Algorithms . . . 10
- 2.5 Summary . . . 12

3 Background Theory 15
- 3.1 Introduction . . . 15
- 3.2 3D Cameras . . . 15
  - 3.2.1 Structured light cameras . . . 16
  - 3.2.2 Time-of-Flight cameras . . . 17

3.3 Point Clouds . . . 19
3.3.1 Octree . . . 19
3.4 Point Cloud Processing . . . 21
3.4.1 Noise filtering . . . 21
3.4.2 Downsampling . . . 21
3.4.3 Surface smoothing . . . 22
3.4.4 Segmentation . . . 22
3.4.5 3D keypoints and descriptors . . . 23
3.4.6 Correspondence matching . . . 23
3.4.7 Alignment . . . 23
3.4.8 Fine registration . . . 24
3.5 Least-Squares Fitting . . . 25
3.6 Geometric Algebra . . . 26
3.6.1 Blades, pseudoscalars and mulitvectos . . . 27
3.6.2 Geometric product . . . 27
3.6.3 Invertibility and Duality . . . 28
3.6.4 Basis Vectors in Conformal Space . . . 29
3.6.5 The Basic Geometric Entities in Conformal Space . . . 29
3.6.6 Least-Squares Fitting in Conformal Space . . . 30
3.6.7 Geometric Algebra vs Vector Algebra . . . 31
3.7 Kinematics . . . 32

4 Solution and Implementation **33**
4.1 Introduction . . . 33
4.2 Software . . . 33
4.2.1 Structure . . . 34
4.2.2 Graphical User Interface . . . 35
4.2.3 External Libraries . . . 35
4.3 Shape modeling . . . 36
4.3.1 Plane . . . 36
4.3.2 Sphere . . . 36
4.3.3 Circle . . . 37
4.3.4 Cylinder . . . 37
4.4 Primitive Shape Detection . . . 39
4.4.1 Initialization and Inlier Classification . . . 39
4.4.2 Termination Strategy . . . 41
4.4.3 Probability . . . 43
4.5 Tracking . . . 43
4.6 Multiple Shape Detection . . . 44
4.6.1 Overview . . . 45
4.6.2 Sampling strategy . . . 45
4.6.3 Performance . . . 46
4.6.4 Scoring . . . 47
4.7 Robotic Pick-and-Place . . . 47
4.7.1 Information Flow and Software . . . 48
4.7.2 Calibration . . . 48

4.7.3 Place Position 50
4.7.4 Pick Position 50
4.7.5 Automatic Execution 50
4.7.6 Video 51

5 Analysis and Discussion **53**
5.1 Introduction 53
5.2 Primitive Shape Detection 53
5.2.1 Sphere 54
5.2.2 Cylinder 56
5.3 Tracking 57
5.3.1 Single Point Cloud 58
5.3.2 Real-Time Static Tracking 58
5.3.3 Real-Time Movement Tracking 59
5.4 Multiple Shape Detection 60
5.5 Scoring Function 63
5.6 Summarizing Remarks 64

6 Concluding Remarks **67**
6.1 Discussion 67
6.2 Conclusion 67
6.3 Further Work 68

Bibliography **71**

Appendices **75**
A Results 77
B C++ Source Code 79
C Digital Appendix 106

List of Tables

2.1 Least-squares procedure 8
2.2 RANSAC procedure 9

3.1 The 32 blades of 5D conformal geometric algebra 27
3.2 The basic geometric entities of 5D conformal space 29

5.1 Results tracking 58
5.2 Results multiple shape detection 62

1 Results for sphere experiment with unknown radius 77
2 Results for sphere experiment with known radius 77
3 Results for cylinder experiment with plane-circle initialization 78
4 Results for cylinder experiment with sphere-sphere initialization 78

List of Figures

2.1 Robust statistical methods 5
2.2 Original data set for RANSAC 8

3.1 Concept of structured light triangulation 16
3.2 Concept of Time-of-Flight 17
3.3 Emitted and received CW ToF signal 18
3.4 Point cloud of a sphere 19
3.5 Octree structure 20
3.6 Rabbit in an octree structure 20
3.7 Voxel grid 22
3.8 Smoothing filter 22
3.9 Alignment of two point clouds 24
3.10 Reference frames for kinematics 32

4.1 Software structure 34
4.2 Developed graphical user interface 35
4.3 Two approaches for defining a cylinder 38
4.4 Point cloud for multiple shape detection 46
4.5 Demonstration setup 47
4.6 Information flow of the demonstrator 48
4.7 Robot cell 49

5.1 Results for the sphere experiments 54
5.2 Evaluation ratio 55
5.3 Results for the cylinder experiments 57
5.4 Single point cloud tracking sequence 59
5.5 Real-time static tracking sequence 60
5.6 Real-time movement tracking sequence 61
5.7 Successful detections of the multiple shape detection algorithm 62
5.8 Distribution of detected radii 63
5.9 Fitted scoring function 64

6.1 Object subdivided to into regions of primitive shapes 69

Abbreviations

PCL	=	Point Cloud Library
CAD	=	Computer Aided Design
RANSAC	=	RANdom SAmple Consensus
IEEE	=	Institute of Eletrical and Electronics Engineers
SRI	=	Stanford Research Institute
LDP	=	Location Determination Problem
RGB(-D)	=	Red, Green, Blue -(Depth)
ToF	=	Time of Flight
CW	=	Continous Wave
FPS	=	Frames Per Second
MLS	=	Moving Least-Squares
ICP	=	Iterative Closest Point
IPNS	=	Inner Product Null Space
OPNS	=	Outer Product Null Space
GUI	=	Graphical User Interface
IDE	=	Integrated Development Environment
ROS	=	Robot Operating System

Chapter 1

Introduction

1.1 Background

Industrial processes often rely on visual data of objects and surroundings for decision-making and inspection. Such processes can be object pose estimation for robotic manipulation tasks, position localization for automated guided vehicles and 3D difference detection for inspections.

Structured light- and time of flight sensors measure depth, either in a point (range sensor), along a line (laser scanner) or in a matrix structure (3D camera). A 3D camera can output depth data of a scene with high frequency, usually in the form of unorganized point clouds consisting of 3D points sampled from the scene. When the Microsoft Kinect 3D camera and the Point Cloud Library (PCL) [1] was launched in 2010, interest in the field of point cloud processing increased as 3D point cloud algorithms for robotics and perception were made easily available. Popular applications are object recognition, surface reconstruction and segmentation. A typical pipeline for object recognition from point clouds consists of filtering, normal computation, keypoint detection, descriptor computation, matching, registration and finally refinement of the registration. These methods have been developed so that shapes with arbitrary geometry and arbitrary pose can be detected in any scene.

Human- and machine-made objects are often shaped like geometric primitives such as planes, spheres, cylinders, cones or torus, or a combination of several. Thus, production-, inspection- and maintenance processes often involves handling of primitives. Considering this when designing computer vision based detection systems, accurate, simple and efficient data treatment can be achieved. Compared to the traditional point cloud processing pipeline, recognition can be simplified and the required level of competence for applying such systems can be reduced, potentially making vision based solutions more attractive for production enterprises.

In [2] it is demonstrated that it is possible to segment both point clouds and CAD models to primitive shapes for object recognition and pose estimation. The focus of this thesis is the primitive shape detection in point clouds, where the *RANdom SAmple Consensus*

(RANSAC) [3] algorithm is one of the first, simplest and most popular methods. Moreover, as geometry of primitive shapes is a central part of this thesis, geometric algebra in conformal space is studied as it offers intuitive and elegant description and handling of geometry. Implementing and testing RANSAC with conformal geometric algebra is prioritized to demonstrate that conformal geometric algebra is applicable to such problems. Furthermore, literature on robust estimators for computer vision is studied for inspiration and ideas to further optimize and improve the algorithmic primitive shape detection. Lastly, a simple robotic system dependent on 3D camera sensor data is implemented and a pick-and-place task is performed. The goal is to demonstrate that the RANSAC-based algorithms are effective for high accuracy demanding tasks, given raw-data from consumer grade 3D cameras.

1.2 Objectives

The objectives of this thesis are

1. Describe the technology behind 3D cameras and common processing steps for point clouds.
2. Present the RANSAC algorithm and other robust estimators.
3. Describe how primitive shapes can be modeled with geometric algebra.
4. Implement the RANSAC algorithm for primitive shape detection in point clouds with geometric algebra.
5. Further optimize and improve the primitive shape detection.
6. Implement software for point cloud processing and primitive shape detection.
7. Test the system in experiments and present the results.
8. Demonstrate applications of primitive shape detection in a robotic pick-and-place task.

1.3 Approach

The objectives of this thesis have been of both theoretical and practical nature. Theoretical in the sense of studying literature, methods, articles and implementations of existing technology. And practical in the sense of acquiring data from a 3D camera, implementing data processing algorithms in the C++ programming language and testing the implemented software.

Objective 1

The technology behind structured light and time of flight 3D cameras are described in Section 3.2. The operational principal of the 3D camera used in this thesis is presented in detail. Point clouds are presented in Section 3.3 and the common processing steps for an object recognition pipeline are presented in Section 3.4

Objective 2

A literature review of robust statistical estimators and RANSAC-based algorithms for computer vision is presented in Chapter 2. A chronological time-line of published estimators is presented. Thereafter, the essence of each estimator is described in relation to its predecessors.

Objective 3

The basis of geometric algebra is presented in Section 3.6 along with a description of the basic geometric entities in conformal space. Furthermore, object modeling with geometric algebra is presented in Section 4.3.

Objective 4

The implemented RANSAC algorithm is presented in Section 4.4. Three different cases for algorithm termination are considered.

Objective 5

In Section 4.5, a RANSAC-based tracking algorithm is presented. The run-time of the algorithm is reduced by applying a tactic for reducing the data size. Thus, tracking is enabled. In Section 4.6, a algorithm for multiple shape detection is presented. A sampling strategy that potentially increases the efficiency of RANSAC is suggested.

Objective 6

The software is implemented in the C++ programming language and relevant source code snippets are appended appendix B. The structure of the software is explained in Section 4.2. The software contains implementations of the suggested algorithms in addition to functionality for point cloud processing and data control.

Objective 7

The suggested algorithms are tested in experiments with the developed software. Result and analysis of the experiments are presented in Chapter 5.

Objective 8

A simple robotic pick-and-place task is defined and described in Section 4.7. A 3D camera-robot demonstration is implemented in the robot laboratory at the Department of Production and Quality Engineering. The goal is to place arbitrarily positioned ping-pong balls in a arbitrarily positioned tube. The objects are detected and positioned by data from a 3D camera. The whole demonstration is documented in a video found in the digital appendix.

Literature and documentation

Articles from the Institute of Electrical and Electronics Engineers (IEEE) journals have been used to a large extend during the project. They are referred to where relevant and listed in the bibliography.

Tutorials, documentation and code snippets from the Point Cloud Library [1] have been used in the implementation of the C++ software.

Textbooks on geometric algebra have been studied to a large extent. The books are:

- *Foundations of geometric algebra computing by Dietmar Hildenbrand* [4].
- *Geometric algebra for computer science: an object-oriented approach to geometry by Leo Dorst, Daniel Fontijne and Stephen Mann* [5].

In addition, relevant scientific articles were studied. Especially important were the article on the RANSAC algorithm [3] and the efficient multiple shape detection [6].

1.4 Structure of the Report

The report is primarily focused on the task of primitive shape detection in point clouds. The literature review, background theory, and solutions are presented for the purpose of this task. In addition, some motivational aspects are established by presenting possible applications and existing methods.

- Chapter 1 present the background and motivation for the work done in this thesis. In addition, the objectives and approach are presented.
- Chapter 2 gives an overview of the statistical estimators for computer vision studied in this thesis. There is a dedicated section to the RANSAC algorithm and a dedicated section to RANSAC-based publications.
- In Chapter 3 the background theory describing 3D camera technologies, point clouds, point cloud processing, least-squares fitting, geometric algebra and kinematics is presented. This is the foundation for the work done. Especially important are the sections about point clouds and geometric algebra, which are the basis for all the algorithms implemented.
- Chapter 4 describes the developed software, shape modeling with conformal geometric algebra, primitive shape detection with RANSAC, a suggested tracking algorithm, a suggested multiple shape detection algorithm and the implementation and execution of the robotic pick-and-place demonstrator.
- In Chapter 5 a description and analysis of the conducted experiments are presented. The performance of the implemented algorithms is mapped and results are presented and discussed.
- In Chapter 6 a discussion regarding practical issues during the work is presented. Furthermore, a conclusion of the thesis is presented and suggestion for further work are discussed.

Chapter 2

Literature Review

2.1 Introduction

In this chapter, a brief overview of robust statistical methods for computer vision will be given in a chronological order and RANSAC will be placed among them. The RANSAC algorithm will be presented in detail, along with a brief description of its descendants. A graphical overview can be seen in Figure 2.1, where each method is listed with its abbreviation and year of publication. A more extensive study on robust statistics in computer vision can be found in [7].

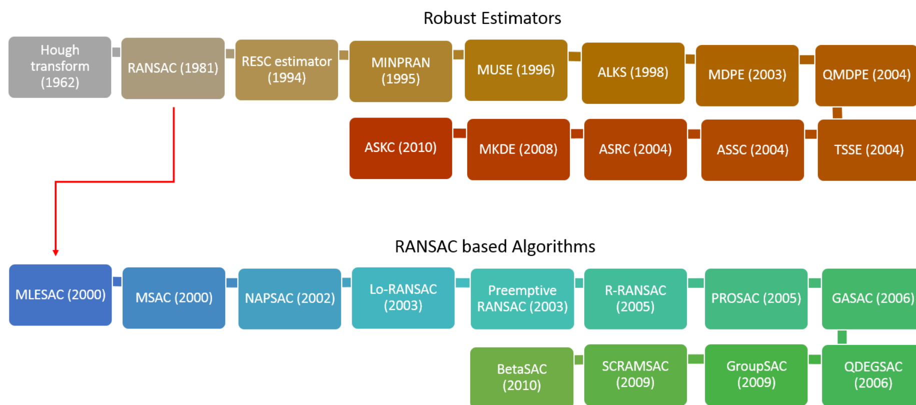


Figure 2.1: A timeline of publications of robust statistical methods for computer vision. The timleine is divided into independent publications and RANSAC-based publications.

It is important to have in mind that RANSAC is only one of many methods for dealing with noisy data. Such methods have been widely studied and developed in the field of robust statistics. Robust statistical methods were adopted to computer vision to improve

performance of various feature extraction algorithms. As the field of computer vision evolved in pace with the computational powers of computers, there was a need for methods that could efficiently handle large data sets with high percentage of noise. In addition, visual data is usually more complex than data analyzed in statistics. The RANSAC algorithm was developed within the computer vision community itself and is one of the first algorithms developed specifically for dealing with vision data.

2.2 Robust Estimators for Computer Vision

Robust statistical methods can be classified as methods that tolerate the presence of data points that do not obey the assumed model. These data points are called *outliers* [7]. The robustness of an estimator can be quantified by the percentage of outliers that can be present without causing arbitrarily bad results.

The *Hough transform* was patented by P.VC Hough (1962) in [8] and has become one of the most applied robust estimators in computer vision. Originally, the Hough transform was a method of fitting lines to a set of noisy data points. It does so by mapping each point to the line-parameter space, where each point will produce a line in the parameter space. If several points lie on the same line, the lines in the parameter space will intersect. A simple voting scheme can then be applied, usually local maximums of intersections in the parameter space are identified as the actual parameters of a line in the xy plane.

The Hough Transform has also been adapted to fit other shapes, like circles and cylinders. There are also a method called *Generalized Hough Transform* that can fit any arbitrary shape, given some training data. However, when the number of parameters increases, the number of votes cast in each local maximum may be small. Thus, the local maximums corresponding to real shapes in the data set may not have more votes than their neighbors. Therefore, the Hough Transform must be used with care when applied to shapes other than lines and circles.

The *Random Sample Consensus* (RANSAC) algorithm was published by Martin A. Fischler and Robert C. Bolles at Stanford Research Institute (SRI) International (1981) in [3]. RANSAC, in its most basic form, is a method of segmenting outliers from a data set given some predefined model. Even though the idea behind RANSAC is simple, it is extremely powerful and have had a big impact on the field of computer vision. The RANSAC algorithm will be presented in detail in the next section.

The *RESidual Consensus* (RESC) algorithm was introduced by Yu et al. (1994) in [9]. RESC is an estimator developed for segmenting primitive shapes in point clouds. It does so by randomly initializing a primitive from a minimum set of points, the rest of the points residuals are calculated and presented in a histogram. After repeating the process K times, the primitive shape is identified as the largest continuous region where the residuals tend to be minimum.

The *Minimum Probability Randomness Estimator* (MINPRAN) algorithm was presented by Charles V. Stewart (1995) in [10]. MINPRAN does not rely on a known error bound for the good data. Instead, it assumes the bad data are randomly distributed within the dynamic range of the sensor. Based on this, MINPRAN uses random sampling to search for the fit and the inliers to the fit that are least likely to have occurred randomly. Using this technique, MINPRAN can deal with noise levels above 50%.

Charles V. Stewart also introduced the *Minimum Unbiased Scale Estimator* (MUSE) algorithm with James V. Miller (1996) in [11]. They observed that random outliers increased in regions of discontinuities in range images and developed the algorithm as a consequence of their observation. MUSE is especially developed to extract multiple primitives from range images by identifying continuous regions.

The *Adaptive Least k^{th} Order Squares* (ALKS) algorithm was proposed by Lee et al. (1998) in [12]. The estimator minimizes the kth order statistics of the squared of residuals. The homogeneous surface patch representing the relative majority of the points is detected by determining the optimal value of k from the data. Thus, the largest continuous surface is detected for each run of the algorithm, which can be fitted to some primitive.

The *Maximum Density Power Estimator* (MDPE) and the *Quick Maximum Density Power Estimator* (QMDPE) was introduced by Wang and Suter (2003) in [13]. MDPE is a parameter estimation algorithm that optimizes an objective function that measures both the density distribution of data points in residual space and the size of the residual corresponding to the local maximum of the density distribution. They claim the algorithm can tolerate 85% outliers. They adapt MDPE to work with segmentation in range images and call this algorithm Quick-MDPE.

In 2004, Wang and Suter introduced two novel estimators called *Two-Step Scale estimatos* (TSSE) and *Adaptive Scale Sample Consensus* (ASSC) in [14]. They divide robust estimation in two task; the task of estimating the model parameters and the task of estimating the scale of the noise in the inlier data. The TSSE estimator is used for estimating the scale of inliers, while ASSC combine TSSE and RANSAC to robustly estimate the model parameters. The advantage of this is that they do not require a prior knowledge of the scale of inliers, as opposite to the RANSAC algorithm (for example).

Wang and Suter (2004) also presented the *Adaptive Scale Residual Consensus* (ASRC) algorithm intended for computer vision tasks in [15]. ASRC scores a model based on both the residuals of inliers and the corresponding scale estimate determined by those inliers. This algorithm is very robust to multiple structure data with high percentage of outliers and requires no predefined inlier threshold.

In 2008, Wang introduced an improved version of QMDPE called *Maximum kernel Density Estimator* (MKDE) in [16]. This algorithm uses the kernel density to evaluate the residuals, which reduces the computational cost compared to QMDPE.

In 2010, Wang, Mirota and Hager presented the *Adaptive-Scale Kernel Consensus* (ASKC) estimator in [17]. This is a generalization of previous published algorithms like RANSAC, ASSC and MKDE that is based on nonparametric kernel density estimation theory. They show that RANSAC, ASSC and MKDE all are special cases of ASKC.

2.3 Random Sample Consensus (RANSAC)

As previously stated, the RANSAC [3] algorithm was introduced by Fishler and Bolles in 1981. They demonstrated the robustness of RANSAC compared to a iterative least-squares fitting with a ”throwing out the worst residual” heuristic. Their original data set and figure for this demonstration can be seen in Figure 2.2. The obvious interpretation of the line is $y = x$, with a small error in point 4 and a gross error in point 7. An iterative least-squares

Table 2.1: Iterative least-squares procedure with a heuristic of throwing out the worst residual. The fitted line will not agree with the actual model, due to noise in the data set.

Iteration	Data set	Line fitted by least-squares
1	$1, 2, 3, 4, 5, 6, 7$	$y = 1.48 + 0.16x$
2	$1, 2, 3, 4, 5, 7$	$y = 1.25 + 0.13x$
3	$1, 2, 3, 4, 7$	$y = 0.96 + 0.14x$
4	$2, 3, 4, 7$	$y = 1.51 + 0.06x$

approach with a heuristic of throwing out the worst residual is tabled in Table 2.1, the noise in the data set causes a bad fitting.

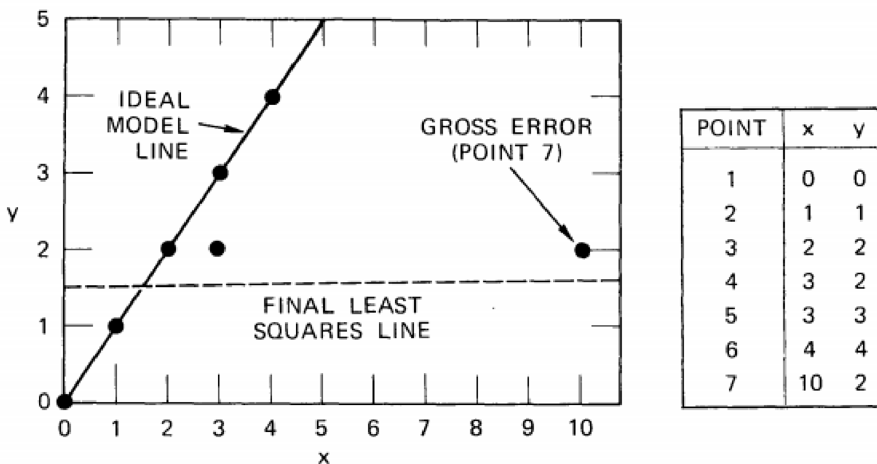


Figure 2.2: Original data set and figure used by Fishler and Bolles to demonstrate the robustness of RANSAC. The data set represent the line $y = x$ with a small error in point 4 and a gross error in point 7. (Adapted from [3])

A RANSAC approach for the line fitting problem is tabled in Table 2.2. Here, a minimal data set is randomly sampled for every iteration and a line is generated from the data set. Each generated line is evaluated by its number of inliers and the line with the most inliers is picked as the final line. Thus, the RANSAC approach will successfully identify the line $x = y$ in this example.

Generally, the RANSAC algorithm fit a predefined model to experimental (noisy) data. Fishler and Bolles studied the effectiveness of the algorithm applied to the Location Determination Problem (LDP) in image analysis. The LDP is the problem of determining in what point in space an image was obtained, given some landmarks with known location in the image scene. However, Fishler and Bolles recognizes the potential of the algorithm when applied to scientific problems in general, where interpreting sensed data in terms of predefined models is a common task.

Conventional fitting methods, like least-squares fitting, considers the whole data set

Table 2.2: A RANSAC procedure for the line-fitting problem. The generated line with the most inliers is identified as the final line.

Iteration	Data set	Generated line	Inliers	# inliers
1	$1, 2$	$y = x$	$1, 2, 3, 5, 6$	5
2	$1, 3$	$y = x$	$1, 2, 3, 5, 6$	5
3	$1, 4$	$y = 0.67x$	$1, 2, 4$	3
4	$1, 7$	$y = 0.2x$	$1, 7$	2
5	$3, 4$	$y = 2$	$3, 4, 7$	3

when fitting a model. A gross error in the data set will result in a model that represent the reality poorly. The uniqueness of RANSAC is that it is capable of detecting these gross errors as outliers and completely neglect them when fitting the final model. Data points that lies within the model tolerance is classified as inliers. Depending on the complexity of the model (the size of random samples) RANSAC can handle contamination levels well above 50%, which is commonly assumed to be a practical limit in robust statistics [18].

RANSAC as formally stated by Fishler and Bolles:

> Given a model that requires a minimum of n data points to instantiate its free parameters, and a set of data points P such that the number of points in P is greater than n $[\#(P) \geq n]$, randomly select a subset $S1$ of n data points from P and instantiate the model. Use the instantiated model $M1$ to determine the subset $S1^*$ of points in P that are within some error tolerance of $M1$. The set $S1^*$ is called the consensus set of $S1$.
>
> If $\#(SI^*)$ is greater than some threshold t, which is a function of the estimate of the number of gross errors in P, use $S1^*$ to compute (possibly using least-squares) a new model $M1^*$.
>
> If $\#(S1^*)$ is less than t, randomly select a new subset $S2$ and repeat the above process. If, after some predetermined number of trials, no consensus set with t or more members has been found, either solve the model with the largest consensus set found, or terminate in failure.

In addition to the data set and the specified model, the RANSAC algorithm requires three parameters:

1. The error tolerance T, used to determine whether or not a point is compatible with a model
2. The number of subsets to try, k
3. The threshold t, which is the number of compatible points used to imply that the correct model has been found.

Fishler and Bolles suggest methods for determining reasonable values for these parameters. These parameters can also be determined experimentally, and by allowing k to be sufficiently large, good results for the final model are in most cases certain. The downside

of letting k be large is the computational cost, as k represents the number of times the algorithm iterates. Hence, the challenge when applying RANSAC is to find a good trade-off between the accuracy of the final model and the number of iterations. The tolerance and threshold parameters are dependent on both the model and data set, but should be picked so convergence is reached as fast as possible.

By defining these three user-dependent parameters, Fishler and Bolles encouraged (consciously or not) several researches to contribute in the work of reducing the dependency of such parameters. This have led to the publication of several RANSAC algorithmic techniques.

2.4 RANSAC Based Algorithms

Since the publication of the algorithm in 1981, RANSAC has grown to become a standard tool in computer vision and image processing. Numerous contributions have been made to improve speed, robustness, accuracy and dependency of user specified parameters.

The *Maximum Likelihood Estimation Sample Consensus* (MLESAC) estimator was introduced by Torr and Zisserman (1996) in [19]. This is a generalization of RANSAC, that adopt the same sampling strategy as RANSAC to generate putative solutions. It chooses the solution that maximizes the likelihood of the solution, instead of just evaluating the number of inliers. MLESAC shows superior results to RANSAC for the test data used by Torr and Zisserman. MLESAC was introduced as a part of a new method for robustly estimating multiple view relations from point correspondences.

The *M-estimator Sample Consensus* (MSAC) was also introduced by Torr and Zisserman (2000) in [19]. MSAC was introduced as a "first enhancement" of RANSAC, before they moved on to introducing MLESAC. The only difference between RANSAC and MSAC is the evaluation of inliers and outliers. In RANSAC, each model is scored with its number of inliers. In MSAC, the score of each inlier is weighted, based on how well it fit the model. Torr and Zisserman argue that the computational cost of doing this evaluation is very small and that "there is no reason to use RANSAC in preference to this method".

As Fishler and Bolles stated, a deterministic selection process that reduces the number of subset k to try, and consequently the number of iterations of the algorithm, is advantageous. *N Adjacent Points SAmple Consensus* (NAPSAC) was introduced by Myatt et al. (2002) in [20]. They show that picking data points with proximity, for the model instantiation, increases the probability of selecting a minimal set consisting of inliers. Taking a noisy point cloud of a scene containing a sphere as an example, intuition argues that the probability of 4 point representing the surface of the sphere is greater when the points are selected based on locality, rather than completely random. The proximity criteria is integrated with the original RANSAC algorithm in NAPSAC. NAPSAC shows superior behavior to the original algorithm in high noise and higher dimension data.

Locally Optimized RANSAC (Lo-RANSAC) was introduced by Chum et al. (2003) in [21]. They argue that the underlying assumption of RANSAC, that a model with parameters computed from an outlier-free sample is consistent with all inliers, does not hold in practice. Therefore they propose and test several local optimization techniques that is applied every time that RANSAC finds a model with more than t inliers. In a standard

RANSAC implementation, optimization is usually done after a run of the algorithm, typically the model is fitted to its inliers with least-squares methods. Chum et al. find that their optimization scheme produces results that agree with the underlying assumption of RANSAC mentioned above.

A preemptive strategy was proposed by Nistér (2003) in [22]. Nistér apply RANSAC to real-time computation problems and argue for a strategy that can keep the computation time of the algorithm more or less constant. His solution is to generate a predefined number of models and use the highest scored model in this model-pool. The output of the algorithm does not satisfy an absolute quality metric; the output is now dependent of the quality of the generated models.

The evaluation of the instantiated model is an obvious time consuming step of the RANSAC algorithm. Here, every data point have to be evaluated against the model, and this is repeated for every iteration of the algorithm. The time consumed for evaluating a model is proportional to the number of data points N. *R-RANSAC*, or Randomized RANSAC, was introduced by Chum and Matas (2005) in [23]. They show that it is sufficient to only test the model against a small number, d, of data points from the total of N data points ($d << N$). They implement this in a two-step procedure:

1. Evaluate model against d randomly selected data points
2. Only evaluate against all N data points if the first evaluation scores above a certain threshold

They show that this evaluation scheme improve the speed of the algorithm considerably compared to the original evaluation scheme. The conclusion of their research is that their evaluation scheme obtains the same solution as the original algorithm, in a shorter time.

The *Progressive Sample Consensus* (PROSAC) algorithm was proposed by Chum and Matas (2005) in [24]. They focus their work on matching corresponding points of interest between two images. They apply RANSAC to this problem and then develop an improved version of RANSAC for this task. Instead of a random sampling strategy, they rank each correspondence for each iteration and pick their samples form a progressively larger set of top-ranked correspondences. They find that their sampling scheme can be up to a hundred times faster than RANSAC for this application.

An evolutionary sampling strategy for RANSAC is proposed by Rodehorst and Hellwich (2006) in [25]. They call their modified algorithm for *Genetic Algorithm SAmple Consensus* (GASAC), as they model and modify the minimal samples following the evolution of a gene. Each minimal sample is classified as a gene and can be mutated or crossed with other genes. For every iteration, each gene is scored with some fitness function (eg. number of inliers). High scoring genes are mutated or crossed with other genes and rescored in the next iteration. This evolutionary sampling strategy tends to converge against a global maximum of the fitness score, as local maximums are avoided through mutation and crossing of the genes. Rodehors and Hellwich conclude that significant acceleration can be achieved using a systematic sampling strategy rather than a random one.

Frahm and Pollefeys (2006) propose in [26] a RANSAC-based algorithm for handling (quasi-)degenerate data. Their estimator is called *(Quasi-)DEGenerate data SAmple Consensus* (QDEGSAC). Degeneracy means that the data do not provide enough constraints to compute the relation uniquely, but up to a family of relations that all explain the data.

QDEGSAC is especially developed for computation of the relation from a number of potential matches in image analysis, where degenerate data often is generated from homogeneous regions in the image (like plane surfaces). They concluded that their method performs as well as the state of the art while being more generally applicable.

Ni, Jin and Dellaert (2009) introduce a estimator in [27] that takes advantage of the assumptions that inliers often comes in groups of some form. They propose the *Group SAmple Consensus* (GroupSAC) estimator which uses a grouping sampling strategy. They also point out that the grouping of data can be based on other properties than locality. The algorithm is applied to the image-matching problem and they show that grouping based on the optical flow of the image produce good results. They conclude that their algorithm performs better than RANSAC on data sets with high percentage of noise, given that the inliers can be grouped by some property.

RANSAC with Spatial Consistency Check (SCRAMSAC) is proposed by Sattler, Leibe and Kobbelt (2009) in [28]. They use a spatial consistency check (SCC) to evaluate the set of inliers. The fraction of inliers is measured in a specific region and all further processing is limited to the part of the data set where the fraction surpasses some threshold θ. This results in a reduced data set of higher quality. They show that this simple idea has important consequences for the runtime and robustness of the estimated results. Compared to RANSAC their method improve runtime by a large factor while yielding similar results.

Another sampling strategy based on the beta-probability distribution is introduced by Meler, Decroues and Crowley (2010) in [29], namely the *Beta SAmple Consensus* (BetaSAC). After initializing a minimal set and scoring all points with respect to this set, the points are sorted according to the beta-distribution and a new minimal sample is drawn from this distribution. They demonstrate the benefits of their method on the homography estimation problem and concludes that BetaSAC is dozens of times faster than PROSAC in some cases.

RANSAC have also been developed and adapted to work with data sets containing multiple primitives. Schnabel, Whal and Klein introduced, in [6], an algorithm that can fit several primitive geometric shapes to a point cloud using the RANSAC algorithm. They do this by constructing all their predefined models (plane, sphere, cylinder, cone and torus) for every subset of data points. Each model is scored and evaluated against the other models and some threshold. They also introduce a novel sampling scheme that require the point cloud to be organized in an octree structure [30], this allows a more rapid and efficient search for models in the point cloud.

2.5 Summary

In this literature study, it is found that many robust statistical estimators exist. RANSAC is one of the first robust estimator developed from within the computer vision community. The random sampling scheme and iterative approach of RANSAC is the basis for many algorithms developed for computer vision. Thus, RANSAC is also the basis for the work done in this thesis. However, a lot of inspiration is sourced from the other methods presented in this chapter.

Some summarizing points:

- Robust statistical methods were adopted to computer vision to handle large, noisy data sets
- Robust estimators developed for computer vision are usually applied to problems like image matching and point cloud processing, but can be applied to any statistical parameter estimation problem.
- RANSAC is the first of many robust estimators developed for computer vision.
- RANSAC uses a random sampling scheme and requires three user-specified parameters.
- Numerous contributions to RANSAC have been made to improve speed, robustness, accuracy and dependency of user-specified parameters.

Chapter 3

Background Theory

3.1 Introduction

This chapter gives insight to the theory and concepts studied for fulfilling the objectives of this thesis.

Section 3.2 gives a description of the technology behind the two most common types of 3D cameras today. The goal is to give an understanding of how point clouds of a scene can be generated from such sensors.

Point clouds and their structure are presented in Section 3.3. In Section 3.4 the methods of point cloud processing mentioned in the introducing chapter are described. Noise filtering, smoothing, segmentation and general object recognition are all problems that can be solved with primitive shape detection. The complexity of many of these methods, and the required level of competence for applying such methods, can be reduced. Thus, this section also works as a motivation for developing primitive shape detection algorithms.

Section 3.5 gives a presentation of linear least-squares fitting, which is a central tool in many of the robust estimators discussed in the previous chapter. Furthermore, Section 3.6 gives a brief introduction to geometric algebra and conformal space. Lastly, Section 3.7 show the basic relations of kinematics applied in the robotic pick-and-place demonstrator.

3.2 3D Cameras

Optical 3D data acquisition is the acquiring of the distance from the measuring device to the scene, also called depth or z-coordinate. 3D cameras are a special type of cameras that maps the depth of a scene. In a traditional camera, color information is stored in pixels organized in a matrix structure. Likewise, depth is stored in pixels organized in a matrix structure in 3D cameras. Each pixel can be assigned an x- and y-value based on the intrinsic camera parameters. Consequently, each pixel in a 3D camera represents a 3D point in space.

3D cameras, as different from laser scanners and range finders, can capture the depth

of an entire scene at a single instance. Modern technology makes it possible to capture and output this depth several times per second. Today, cheap, consumer grade cameras can output high-density depth streams at high frame rates.

Consumer grade 3D cameras (also called depth cameras) normally comes with a built-in RGB color camera. The RGB camera and 3D camera can be calibrated such that colour and depth coincide. Such cameras are also called RGB-D cameras, and the information outputted from such cameras can be visualized as coloured 3D points or surfaces.

There exist primarily two types of 3D cameras based on the technology used for obtaining 3D information from the scene.

3.2.1 Structured light cameras

Structured light cameras consists of a light projector and an image sensor. Light is projected in a pattern on to the scene and this pattern is recognized in the image. When the pattern is recognized, it is possible to triangulate the 3D position of every point in the pattern by knowing the position and orientation of the projector relative to the image frame. This method can create dense point clouds that can be obtained under various illumination conditions at high frame rates. A structured light camera is classified as an active type camera, because it actively projects light on to the scene to be mapped.

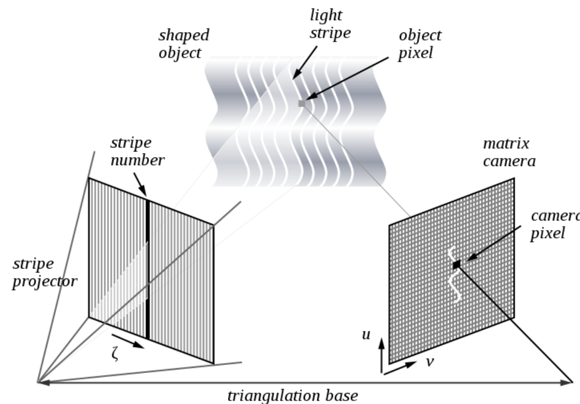


Figure 3.1: Concept of structured light triangulation. The structured light emitted on the scene is identified in the camera matrix. By knowing the transformation between the projector and the camera, the position of each identified light structure can be identified. (Adapted from http://www.extremetech.com/)

Figure 3.1 demonstrates the triangulation procedure in a structured light setup. Structured light is emitted from the projector in a vertical stripe pattern. A traditional digital image sensor observes the projected light pattern and by counting the number of stripes, it is possible to identify each stripe and the angle of transmittance from the projector. Using the intrinsic camera parameters, the angle from the camera frame to the point can be determined. By using the angle from the projector, the angle from the camera and the translation between them, this becomes a simple geometry problem.

Structured light cameras represents one of the main group of 3D cameras today. They offer real-time 3D data streams and can operate in various lighting conditions. They also eliminate the problem of ambiguity in homogeneous surfaces by projecting light. Some cameras also use infrared light projectors and sensors so that the scene is not disturbed by projected visual light.

3.2.2 Time-of-Flight cameras

Time-of-Flight (ToF) cameras also fall under the class of active cameras as they project light on to the scene. The light is reflected from the scene back to the projector, where the receiver is located. The idea of ToF cameras is to measure the time from when the light is emitted from the projector to when the light is received by the receiver. Using the speed of light c, it is possible to calculate the distance to the point of reflection. Figure 3.2 and equation (3.1) gives a simple description of the concept, here is ρ the distance to the scene and τ is the measured time of flight.

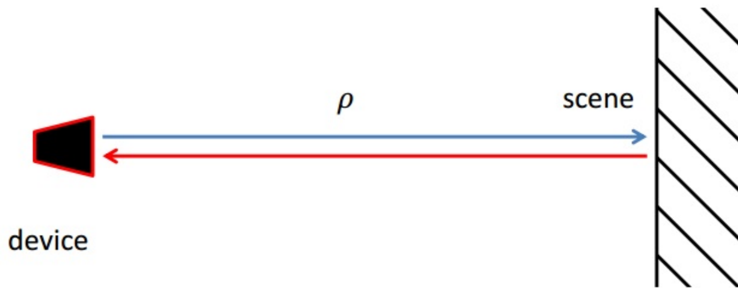


Figure 3.2: Concept of Time-of-Flight. The time of flight is measured from the signal is emitted to it is received. (Adapted from [31])

$$\rho = \frac{c\tau}{2} \tag{3.1}$$

Direct measurement of travel time of light require fast electronics that can measure time spans in the picoseconds range. To avoid the cost and challenges of fast electronics, different clock technologies have been introduced and this have led to several different ToF camera types [31]. The most common type is the continuous wave (CW) intensity modulation approach; this approach is also the basis in the Kinect for Xbox One camera used in this thesis.

CW ToF cameras apply a modulation frequency to a light-wave so that the light becomes a carrier signal for the modulation frequency. The problem of distance estimation is shifted from observing the travel time of light to observing the phase shift of the emitted modulation frequency. This concept is illustrated by assuming a transmitter and receiver in the same position. The transmitter emits a sinusoidal wave with modulation frequency f_{mod} in the direction of the scene, the modulated wave is reflected by the scene and read

by the receiver. An illustration of the two signals is given in Figure 3.3. The emitted signal can be written as

$$s_E(t) = A_E[1 + \sin(2\pi f_{\text{mod}} t)], \tag{3.2}$$

where $s_E(t)$ is the emitted sinusoidal signal, A_E is the amplitude and f_{mod} is the modulation frequency. The received signal can be written as

$$s_R(t) = A \sin(2\pi f_{\text{mod}} t + \Delta\phi) + B, \tag{3.3}$$

where $s_R(t)$ is the received sinusoidal signal, A is the amplitude, $\Delta\phi$ is the phase shift from the emitted signal and B is the noise term of interfering radiation. [31]

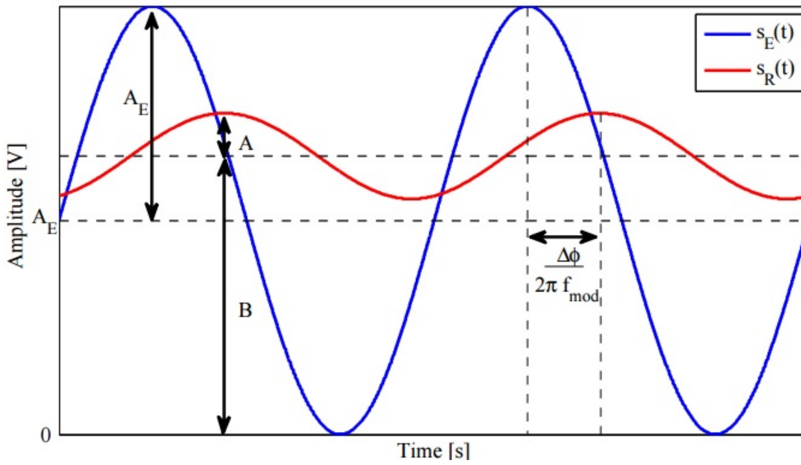


Figure 3.3: Emitted and received CW ToF signal. The blue wave is the emitted signal while the red wave is the received reflected signal. The received signal is identified by its modulation frequency and the phase shift between the signals is used to determine distance to the point of reflection. (Adapted from [31])

The phase shift is in particular interest as this is used to calculate the distance to the object through the relation

$$\rho = \frac{c}{4\pi f_{\text{mod}}} \Delta\phi. \tag{3.4}$$

In practice, the phase shift, amplitude and noise term of the received signal from equation (3.3) is estimated by sampling the received signal 4 times for every period. Thus, the sampling frequency f_{samp}, has to be 4 times the modulation frequency f_{mod}.

In a camera sensor, like in the Kinect for Xbox One, there is an array of receivers that are sampling reflected light from a set of points in the scene. There is not a projector for every receiver, but the projector is constructed in a way that the virtual center lies in the

middle of the array of receivers. By synchronizing the sampling frequency of the receivers, it is possible to capture a number of points in a scene simultaneously, and reconstruct the sampled scene in a point cloud.

The Kinect for Xbox One offer a depth sensor with a 512×424 pixel array with a pixel size of 10×10 μm. This sensor is able to give a $70°$ horizontal and $60°$ vertical field of view with a depth range between 0.8 - 4.2 meters. The sensor can produce a 30 frames per second (FPS) depth image stream where the accuracy error is 1% of the range for an average of 100 images. [32]

3.3 Point Clouds

A point cloud is a set of data points in a coordinate system. In Euclidean space, a point cloud consists of 3D points. Point clouds can be sampled from CAD-models, generated by computer software or created from 3D scanners or cameras. Point clouds can be used for reconstructing surfaces, quality inspection, animation and visualization. Figure 3.4 shows a point cloud of a sphere containing $10\,000$ points.

Point clouds usually comes in an unorganized form, which means that the points are not ordered in any form. Ordering or grouping the points can be advantageous for systematically searching through a point cloud.

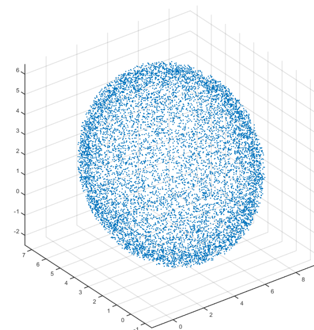


Figure 3.4: Point cloud of a sphere consisting of $10\,000$ points. (Created with MATLAB)

3.3.1 Octree

Octrees are a popular and effective way of organizing a point cloud. Octrees are used to recursively subdividing 3D space into eight octants. Each octant is called a *node* and each node can be subdivided into exactly eight nodes at the lower level of the octree. The root node is the largest and contain all other nodes. The *depth* d of the octree is the number of levels of the octree. The *resolution* d_{res} of an octree is the length of the sides of the nodes at the lowest level of the octree. An octree is fully defined when the length l of the sides of the root node is spesified along with the the depth or resolution. The relation between the root node, depth and resolution can be expressed as:

$$\frac{l}{2^d} = d_{\text{res}}. \tag{3.5}$$

The number of nodes present in an octree is:

$$1 + 8^1 + 8^2 + \ldots + 8^{d-1}. \tag{3.6}$$

Figure 3.5 show the structure of an octree.

By placing the root node of an octree over the space spanned by a point cloud, the points in the point cloud can be organized into the nodes of the octree. The multiple levels

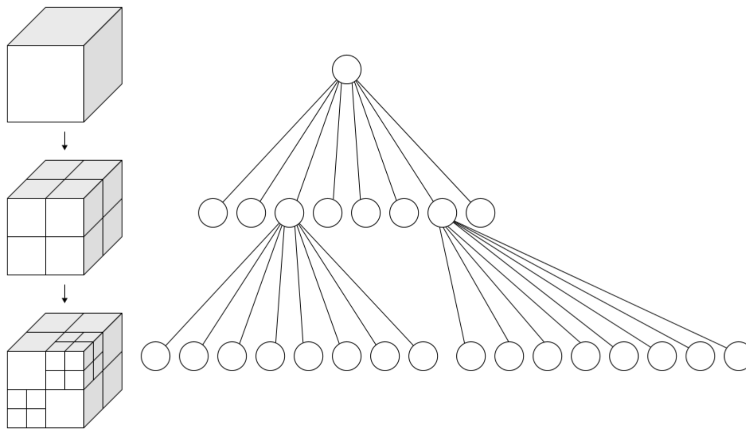


Figure 3.5: Octree structure. Every node is subdivided into 8 nodes in the level below, this is repeated until a desired resolution of the octree is achieved. (Adapted from https://en.wikipedia.org/wiki/Octree)

of an octree allows for systematic searching of different sized point clusters in the point cloud. Figure 3.6 show a 3D model of a rabbit organized in an octree. Only the nodes that contain some part of the rabbit are visualized. Note that an arbitrary point on the rabbit is present at all the levels of the octree.

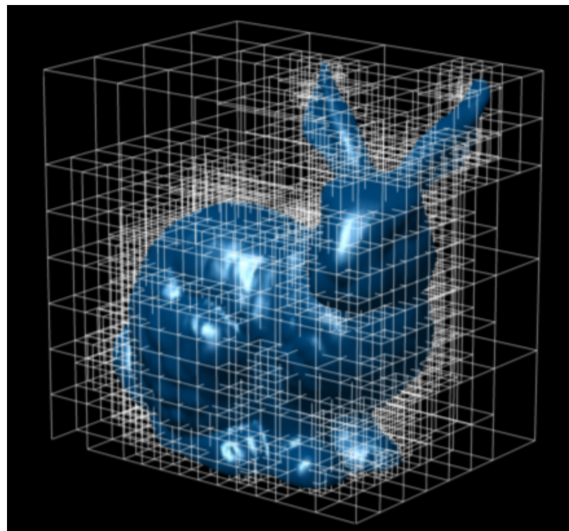


Figure 3.6: A 3D model of a rabbit organized in an octree. Only populated nodes are visualized. (Adapted from http://www.mathworks.com/)

3.4 Point Cloud Processing

Point clouds often contain noise and unwanted elements that needs to be filtered away. It is also desirable to extract information about specific objects, or regions in a point cloud. Several methods and algorithms have been developed for these purposes. In this section a brief summary of some concepts will be given. These concepts will be presented in a sequence following an object recognition pipeline, where the ultimate goal is to recognize and describe a specific object's position and orientation in the scene. This is usually done by matching and fitting some model of the object to the point cloud. This model can be a polygonal mesh, point cloud or a mathematical description of a primitive.

3.4.1 Noise filtering

To decide whether a point is noise or not, its neighbors are considered. One can use simple strategies, like removing a point if its geometrical distance to its nearest neighbor is larger than some threshold. However, methods that are more sophisticated are used for robustness.

Statistical outlier removal is a general method that assumes that the Euclidean distances between neighboring points can be represented by some statistical distribution (eg. the normal distribution). First, all the neighboring distances is considered to establish the parameters of the distribution. Then, all the neighboring distances are tested with this model to decide whether the point is rejected as noise or not.

3.4.2 Downsampling

Downsampling is a simple method for reducing the size and complexity of a point cloud. Ideally, a downsampled point cloud will contain as much information about the scene as the original. The motivation for reducing the complexity of a point cloud is to reduce the calculation load in the later steps of the processing pipeline. Downsampling can be performed by simply removing a number of random points, but more sophisticated methods that preserves the information in the cloud exists.

One of the most common methods is called voxel grid sampling. Voxels (small 3D boxes) are placed over the whole point cloud. The mean, or the centroid, of all the points that lie inside each voxel is computed and replaces the original points in the voxels. By keeping the voxel size small, sufficient detail in the point cloud is preserved. Voxel grid sampling is also a great tool for making two different point clouds equal in density for better comparison. Figure 3.7 shows how a voxel grid is applied over a point cloud and how the points are replaced by the centroid in the voxel.

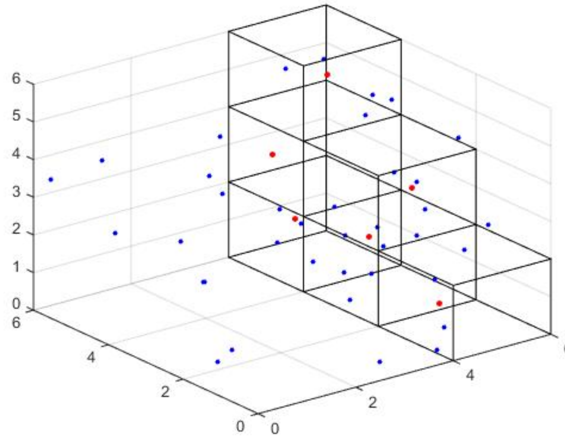


Figure 3.7: Voxel grid in a point cloud. The blue points are the original points while the red points represent the centroid of the blue points within each voxel. (Created with MATLAB)

3.4.3 Surface smoothing

Even though the point cloud is downsampled and noise filtered, it can still contain disturbances like awkward edges, surface inconsistencies and roughness. This can affect later steps in the point cloud-processing pipeline and should be handled to avoid significant errors in the end-result. Different smoothing techniques handles this problem while keeping the shape features of the point cloud intact.

A general smoothing technique called Moving Least Squares (MLS) [33] fits a high order polynomial to the neighboring points of a point. Then, the point is fitted to the polynomial by moving it in the direction of the smallest Euclidean distance from the point to the polynomial. This process is repeated for all the points in a point cloud to obtain a smoothing effect. Figure 3.8 demonstrates the power of surface smoothing. Smoothing algorithms can also be adopted to fit certain shapes, like cylinders or spheres. This makes the smoothing more accurate, given that the point cloud is acquired from a scene with known geometry.

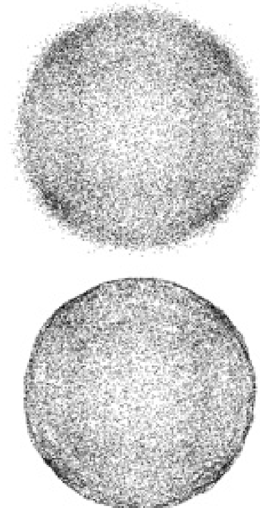


Figure 3.8: Point cloud of a sphere before and after surface smoothing. (Adapted from http://doc.cgal.org/)

3.4.4 Segmentation

Segmentation is simply a term for grouping points based on some property. Such properties can be specific values of coordinates, points that belong to an object or primitive or

points that are classified as noise. When the points are grouped their features can be more easily examined, or they can be filtered and erased from the point cloud.

3.4.5 3D keypoints and descriptors

Keypoints and descriptors are tools used for creating a compact description of a point cloud. A keypoint is simply a point of interest, while a descriptor is a compact description of a region around a keypoint that is assigned to the keypoint.

The motivation and underlying goal of keypoint detection and description is to identify unique points in the point cloud. The unique points are crucial for matching correspondences between two point clouds, alignment of point clouds and object recognition.

Good keypoints are the basis of good descriptors. *Frederico Tombari* gives a nice classification of 3D keypoint properties in his research paper *Performance Evaluation of 3D Keypoint Detectors* [34]:

3D keypoints are

- **Distinctive**, i.e. suitable for effective description and matching
- **Repeatable** with respect to point-of-view variations, noise and scale

Usually these keypoints are found to lie along peaks and edges in the point cloud. These are regions where the distance to some of the point's neighbors is large, e.g. the distance gradient is large. The 3D keypoint detectors that exists also consider factors such as keypoint density, scale and color, if such data is available.

There exists a number of different types of descriptors that extract different type of information from the underlying surface of a point. The descriptors are usually represented by a n-dimensional vector so they can be easily compared. The choice of which descriptor to pick depends very much on the application. The descriptor type described above is a local descriptor that represent information about a point. Other types are global descriptors, that represents information about an entire shape or object. These descriptors are useful when identifying objects in a scene. An overview of the different descriptor types and properties can be found in [35].

3.4.6 Correspondence matching

As a descriptor is a n-dimensional vector that describe the features at a certain point, the corresponding descriptor in another point cloud can be determined by simply comparing the length between descriptor vectors.

Given a set of descriptors for two point clouds $(\mathbf{d}_i, \mathbf{d'}_j)$, $i = 1,..,n$ $j = 1,..,m$, lengths between $\mathbf{d}_i$ and $\mathbf{d'}_{1,..,m}$ are calculated. Then, all lengths are compared and the two points with the smallest descriptor length are identified as correspondences. This is repeated for all the descriptors $\mathbf{d}_{i,..,n}$ and a set of corresponding points are obtained.

3.4.7 Alignment

Alignment is the process of aligning two different point clouds together. In order to perform the alignment, the two point clouds must contain some common scene. This common

scene is identified from the descriptors, and the point clouds are aligned using their common scene as reference. Figure 3.9 show two point clouds of the same face from different views. The point clouds are aligned to create a better representation of the face. This can be used to reconstruct 3D objects, or it can be used to identify an object's position and orientation.

The alignment process begins with matching correspondences from the two point clouds. These correspondences are used to perform an initial alignment before the fine registration is carried out. The fine registration is simply the process of fine-tuning the initial alignment; this is usually done with an iterative algorithm.

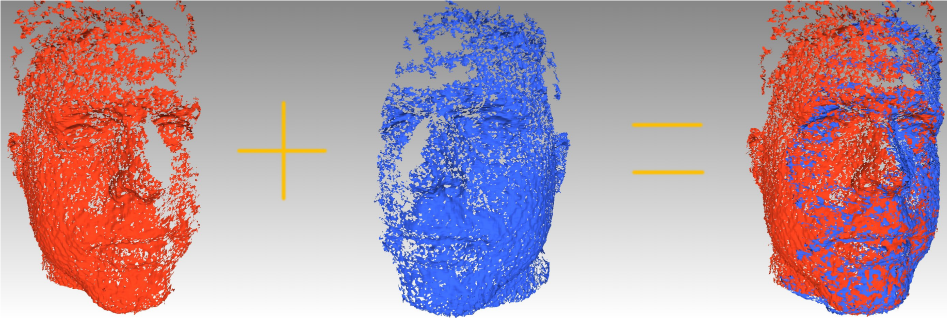


Figure 3.9: Two point clouds of a face aligned. (Adapted from http://dynface4d.isr.uc.pt/)

From the correspondences the homogeneous 4×4 transformation matrix, $\mathbf{T}$, between the two point clouds can be calculated from the relation:

$$\tilde{\mathbf{p}}_k = \mathbf{T}\tilde{\mathbf{p}}'_k, \tag{3.7}$$

where $\tilde{\mathbf{p}}_k$ and $\tilde{\mathbf{p}}'_k$ are the homogeneous corresponding points. As this equation is overdetermined for more than 4 correspondences, the $\mathbf{T}$ matrix is usually calculated by minimizing the error function:

$$E(\mathbf{T}^*) = \sum_k || \tilde{\mathbf{p}}_k - \mathbf{T}\tilde{\mathbf{p}}'_k ||^2 \tag{3.8}$$

3.4.8 Fine registration

As the alignment procedures discussed above can be inaccurate due to mismatches or noise in the data sets, a method for fine tuning the alignment is needed. The Iterative Closest Point (ICP) algorithm was presented by Chen and Medioni (1991) in [36]. This method iteratively refines the alignment by generating correspondences and minimizing an error metric between them. The correspondences are usually matched by picking the nearest neighboring points between the two point clouds. This is considered a brute force method and it may find a solution that lie in a local minimum. Therefore, the initial alignment of the two point clouds is crucial for the result. The ICP algorithm can be summarized in the following steps:

1. Pick correspondences
2. Minimize the cost function

$$E(\mathbf{R}^*,\mathbf{t}^*) = \sum_{i=1}^{n}\sum_{j=1}^{m} w_{i,j} \mid\mid \mathbf{m}_i - (\mathbf{R}\mathbf{d}_j + \mathbf{t}) \mid\mid^2, \tag{3.9}$$

   where $\mathbf{R}$ and $\mathbf{t}$ are the rotation and translation between the two clouds and the * denotes the entities that minimizes $E(\mathbf{R}^*,\mathbf{t}^*)$. $\mathbf{m}_i$ and $\mathbf{d}_j$ are the points in the two point clouds and $w_{i,j}$ is 1 if $\mathbf{m}_i$ and $\mathbf{d}_j$ are correspondences and 0 otherwise.

3. Apply the transformation

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0\,0\,0 & 1 \end{bmatrix}. \tag{3.10}$$

For every iteration a transformation is applied which will shift the relative position between the points in the two point clouds. Therefore, new correspondences have to be picked for every iteration. If the transformation applied is significant, the new correspondences will not be the same as for the previous iteration. Hence, the tactic of the ICP algorithm is simply to minimize the total Euclidean distance between two point clouds. If these two clouds are equal, or similar, they will appear aligned when the total Euclidean distance between them is minimized.

3.5 Least-Squares Fitting

The Least-Squares fitting method is a method for fitting a model to a set of data points. A simple data set consist of n points $(\mathbf{x_i},\mathbf{y_i})$,$i = 1,..,n$. $\mathbf{x_i}$ is independent, while $\mathbf{y_i}$ is dependent of $\mathbf{x_i}$ and is found by observation. The model is described by the function $f(\mathbf{x},B)$, where B is the vector containing the parameters. For example the equation of a line:

$$y = mx + b. \tag{3.11}$$

Here, $y = f(\mathbf{x},B)$, $B = \begin{bmatrix} m & b \end{bmatrix}$ and $\mathbf{x} = x$.

The residual r_i is defined as the difference between the actual value of the dependent variable and the value predicted by the model:

$$r_i = y_i - f(x_i,B). \tag{3.12}$$

The Least-Squares method find the minimum sum of all the squared residuals. This sum is defined as:

$$S = \sum_{i=1}^{n} r_i^2. \tag{3.13}$$

The minimum of the sum is found by setting the gradient to zero:

$$\frac{\partial S}{\partial B_j} = 2\sum_i r_i \frac{\partial r_i}{\partial B_j} = 0, \quad j = 1, \ldots, m, \tag{3.14}$$

where each parameter in B will produce one gradient equation. Equation (3.14) can be reduced to

$$\frac{\partial S}{\partial B_j} = 2\sum_i r_i \frac{\partial f(x_i, B)}{\partial B_j} = 0, \quad j = 1, \ldots, m, \tag{3.15}$$

because y_i is independent of B_j. Setting

$$X_{ij} = \frac{\partial f(x_i, B)}{\partial B_j}, \tag{3.16}$$

the overdetermined system can be defined as:

$$\sum_{j=1}^{n} X_{ij} B_j = y_i, \quad i = 1, \ldots, m. \tag{3.17}$$

Rewriting to matrix form:

$$\mathbf{XB} = \mathbf{y}, \tag{3.18}$$

where

$$\mathbf{X} = \begin{bmatrix} X_{11} & X_{12} & \ldots & X_{1n} \\ X_{21} & X_{22} & \ldots & X_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ X_{m1} & X_{m2} & \ldots & X_{mn} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}.$$

The Least-Squares estimate of B is then given by:

$$\hat{B} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \tag{3.19}$$

3.6 Geometric Algebra

This section gives a brief introduction to geometric algebra. The focus will be on 5D conformal geometric algebra because it allows for easy and intuitive handling of geometric entities such as planes and spheres. The introduction is based of Dietmar Hildenbrand's book *Foundations of Geometric Algebra Computing* published by Springer (2013) [4]. Only material relevant for this thesis will be presented.

Table 3.1: The 32 blades of 5D conformal geometric algebra

Grade	Term	Blades	No.
0	Scalar	1	1
1	Vector	$\mathbf{e}_1, \quad \mathbf{e}_2, \quad \mathbf{e}_3, \quad \mathbf{e}_\infty, \quad \mathbf{e}_0$	5
2	Bivector	$\mathbf{e}_1 \wedge \mathbf{e}_2, \quad \mathbf{e}_1 \wedge \mathbf{e}_3, \quad \mathbf{e}_1 \wedge \mathbf{e}_\infty,$ $\mathbf{e}_1 \wedge \mathbf{e}_0, \quad \mathbf{e}_2 \wedge \mathbf{e}_3, \quad \mathbf{e}_2 \wedge \mathbf{e}_\infty,$ $\mathbf{e}_2 \wedge \mathbf{e}_0, \quad \mathbf{e}_3 \wedge \mathbf{e}_\infty, \quad \mathbf{e}_3 \wedge \mathbf{e}_0,$ $\mathbf{e}_\infty \wedge \mathbf{e}_0$	10
3	Trivector	$\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3, \quad \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_\infty, \quad \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_0,$ $\mathbf{e}_1 \wedge \mathbf{e}_3 \wedge \mathbf{e}_\infty, \quad \mathbf{e}_1 \wedge \mathbf{e}_3 \wedge \mathbf{e}_0, \quad \mathbf{e}_1 \wedge \mathbf{e}_\infty \wedge \mathbf{e}_0,$ $\mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \mathbf{e}_\infty, \quad \mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \mathbf{e}_0, \quad \mathbf{e}_2 \wedge \mathbf{e}_\infty \wedge \mathbf{e}_0,$ $\mathbf{e}_3 \wedge \mathbf{e}_\infty \wedge \mathbf{e}_0$	10
4	Quadvector	$\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \mathbf{e}_\infty,$ $\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \mathbf{e}_0,$ $\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_\infty \wedge \mathbf{e}_0,$ $\mathbf{e}_1 \wedge \mathbf{e}_3 \wedge \mathbf{e}_\infty \wedge \mathbf{e}_0,$ $\mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \mathbf{e}_\infty \wedge \mathbf{e}_0$	5
5	Pseudoscalar	$\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \mathbf{e}_\infty \wedge \mathbf{e}_0$	1

3.6.1 Blades, pseudoscalars and mulitvectos

The basis vectors $\mathbf{e}_1$, $\mathbf{e}_2$,...,$\mathbf{e}_n$ are the basic elements of vector algebra, but only a part of geometric algebra. The basic elements of geometric algebra are called **blades**, where each blade has a **grade**. A scalar is a blade of grade 0 (0-blade) and the basis vectors are blades of grade 1 (1-blade). The blades spanned by two 1-blades are called 2-blades, and so on. The 2-blade

$$\mathbf{A}_2 = \mathbf{e}_1 \wedge \mathbf{e}_2 \tag{3.20}$$

describes a oriented surface area. In this case it is the area spanned by basis vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ which simply is a oriented area in the xy plane. The maximum grade blade n,

$$\mathbf{I} = \mathbf{e}_1 \wedge \mathbf{e}_2 ... \wedge \mathbf{e}_n \tag{3.21}$$

is called the **pesudoscalar** because it only exist one such element. A linear combination of k-blades is called a k-vector and a linear combination of blades with different grades is called a **multivector**, which are the general elements of geometric algebra.

3D Euclidean geometric algebra consists of the scalar, three basis vectors, three bivectors and the pseudo scalar, summing up to a total of 8 blades. 5D conformal geometric algebra consists of the scalar, five basis vectors, ten bivectors, ten trivectors, five quadvectors and the pseudosclar, summing up to a total of 32 blades (see Table 3.1 [4]).

3.6.2 Geometric product

Two vectors in 2D space are given as:

$$\mathbf{u} = u_1\mathbf{e}_1 + u_2\mathbf{e}_2, \quad \mathbf{v} = v_1\mathbf{e}_1 + v_2\mathbf{e}_2. \tag{3.22}$$

Their geometric product is:

$$\mathbf{uv} = \underbrace{\mathbf{u} \wedge \mathbf{v}}_{\text{Outer product}} + \underbrace{\mathbf{u} \cdot \mathbf{v}.}_{\text{Inner product}} \tag{3.23}$$

The inner product of two vectors is a scalar:

$$\begin{aligned} \mathbf{u} \cdot \mathbf{v} &= (u_1\mathbf{e}_1 + u_2\mathbf{e}_2) \cdot (v_1\mathbf{e}_1 + v_2\mathbf{e}_2) \\ &= u_1v_1\mathbf{e}_1 \cdot \mathbf{e}_1 + u_1v_2\mathbf{e}_1 \cdot \mathbf{e}_2 + u_2v_1\mathbf{e}_1 \cdot \mathbf{e}_2 + u_2v_2\mathbf{e}_2 \cdot \mathbf{e}_2 \\ &= u_1v_1 + u_2v_2, \end{aligned} \tag{3.24}$$

where

$$\mathbf{e}_i \cdot \mathbf{e}_j = 0, \quad \mathbf{e}_i \cdot \mathbf{e}_i = 1, \quad i \neq j$$

for basis vectors $\mathbf{e}_{1\ldots n} \in \mathbb{R}^n$. The outer product is:

$$\begin{aligned} \mathbf{u} \wedge \mathbf{v} &= (u_1\mathbf{e}_1 + u_2\mathbf{e}_2) \wedge (v_1\mathbf{e}_1 + v_2\mathbf{e}_2) \\ &= u_1v_1\mathbf{e}_1 \wedge \mathbf{e}_1 + u_1v_2\mathbf{e}_1 \wedge \mathbf{e}_2 + u_2v_1\mathbf{e}_1 \wedge \mathbf{e}_2 + u_2v_2\mathbf{e}_2 \wedge \mathbf{e}_2 \\ &= (u_1v_2 - u_2v_1)\mathbf{e}_1 \wedge \mathbf{e}_2, \end{aligned} \tag{3.25}$$

where

$$\mathbf{e}_i \wedge \mathbf{e}_i = 0, \quad \mathbf{e}_i \wedge \mathbf{e}_j = -\mathbf{e}_j \wedge \mathbf{e}_i, \quad i \neq j$$

for basis vectors $\mathbf{e}_1, ..., \mathbf{e}_n \in \mathbb{R}^n$.

3.6.3 Invertibility and Duality

The inverse of a blade $\mathbf{A}$ is defined by:

$$\mathbf{A}\mathbf{A}^{-1} = 1 \tag{3.26}$$

The inverse of the Euclidean pseudoscalar:

$$\begin{aligned} \mathbf{II} &= -1 \\ \rightarrow \mathbf{II}(\mathbf{I}^{-1}) &= -\mathbf{I}^{-1} \\ \rightarrow \mathbf{I}^{-1} &= -\mathbf{I} \end{aligned} \tag{3.27}$$

The dual of an algebraic expression is calculated by dividing by the pseudoscalar. For example the dual of the plane $\mathbf{A} = \mathbf{e}_2 \wedge (\mathbf{e}_1 + \mathbf{e}_3)$ is:

$$\begin{aligned} \mathbf{A}^* &= \mathbf{e}_2 \wedge (\mathbf{e}_1 + \mathbf{e}_3)(\mathbf{e}_1\mathbf{e}_2\mathbf{e}_3)^{-1} \\ &= \mathbf{e}_2 \wedge (\mathbf{e}_1 + \mathbf{e}_3)(-\mathbf{e}_3\mathbf{e}_2\mathbf{e}_1) \\ &= -\mathbf{e}_3 + \mathbf{e}_1. \end{aligned} \tag{3.28}$$

Table 3.2: The basic geometric entities of 5D conformal space

Entity	IPNS representation	OPNS representation
Point	$P = \mathbf{p} + \frac{1}{2}\mathbf{p}^2\mathbf{e}_\infty + \mathbf{e}_0$	
Sphere	$S = P - \frac{1}{2}r^2\mathbf{e}_\infty$	$S^* = P_1 \wedge P_2 \wedge P_3 \wedge P_4$
Plane	$\pi = \mathbf{n} + d\mathbf{e}_\infty$	$\pi^* = P_1 \wedge P_2 \wedge P_3 \wedge \mathbf{e}_\infty$
Circle	$Z = S_1 \wedge S_2$	$Z^* = P_1 \wedge P_2 \wedge P_3$
Line	$L = \pi_1 \wedge \pi_2$	$L^* = P_1 \wedge P_2 \wedge \mathbf{e}_\infty$
Point pair	$Pp = S_1 \wedge S_2 \wedge S_3$	$Pp^* = P_1 \wedge P_2$

3.6.4 Basis Vectors in Conformal Space

Conformal space consist of the three Euclidean basis vectors $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$ and two additional basis vectors $\mathbf{e}_+$, $\mathbf{e}_-$.

$$\mathbf{e}_+^2 = 1, \quad \mathbf{e}_-^2 = -1, \quad \mathbf{e}_+ \cdot \mathbf{e}_- = 0. \tag{3.29}$$

These basis vectors are combined to give more intuitive meaning:

$$\mathbf{e}_0 = \tfrac{1}{2}(\mathbf{e}_- - \mathbf{e}_+), \quad \mathbf{e}_\infty = \mathbf{e}_- + \mathbf{e}_+. \tag{3.30}$$

$\mathbf{e}_0$ represent the 3D origin and $\mathbf{e}_\infty$ represents infinity. The new basis vectors are null vectors:

$$\mathbf{e}_0^2 = \mathbf{e}_\infty^2 = 0. \tag{3.31}$$

Their inner product is

$$\mathbf{e}_\infty \cdot \mathbf{e}_0 = -1. \tag{3.32}$$

3.6.5 The Basic Geometric Entities in Conformal Space

The basic geometric entities in conformal space (point, sphere, plane, circle, line and point pair) has two representations. The IPNS (inner product null space) representation and the OPNS (outer product null space) representation. These two representations are dual of each other, which makes it easy to go from one to the other. In the OPNS representation, the outer product indicates the construction of a geometric object with help of the points that lie on it. In the IPNS representation, the outer product indicates the intersection of geometric entities. These representations are great when working with point clouds. The OPNS representation allow for easy construction of geometric entities from points, while the IPNS representation allows for easy extraction of parameters.

Both the IPNS and OPNS representations of all the basic geometric entities in conformal space are listed in Table 3.2 [4]. In Table 3.2 $\mathbf{x}$ and $\mathbf{n}$ represent 3D entities obtained by linear combinations of the 3D basis vectors:

$$\mathbf{x} = x_1\mathbf{e}_1 + x_2\mathbf{e}_2 + x_3\mathbf{e}_3 \tag{3.33}$$

3.6.6 Least-Squares Fitting in Conformal Space

This section is a summary of the least-squares approach described in [37] and shows the least-squares fitting procedure of points to spheres and planes in conformal space.

Let P_i describe a point and S a sphere or a plane:

$$P_i = \mathbf{p}_i + \frac{1}{1}\mathbf{p}_i^2\mathbf{e}_\infty + \mathbf{e}_0 \tag{3.34}$$

$$S = \mathbf{s} + s_4 e_\infty + s_5 e_0. \tag{3.35}$$

S is to be fitted to the set of points P by minimizing the error function

$$\min \sum_{i=1}^{n} (P_i \cdot S)^2, \tag{3.36}$$

where $P_i \cdot S$ is a distance measure between point P_i and the sphere/plane S. This can be written in bilinear form as

$$\min(s^T \mathbf{B} s), \tag{3.37}$$

where

$$s^T = (s_1, s_2, s_3, s_4, s_5),$$

and the 5×5 matrix

$$\mathbf{B} = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} & b_{1,5} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} & b_{2,5} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} & b_{3,5} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} & b_{4,5} \\ b_{5,1} & b_{5,2} & b_{5,3} & b_{5,4} & b_{5,5} \end{pmatrix}$$

has entries

$$b_{j,k} = \sum_{i=1}^{n} w_{i,j} w_{i,k},$$

where

$$w_{i,k} = \begin{cases} p_{i,k} & \text{if } k \in \{1,2,3\} \\ -1 & \text{if } k = 4 \\ -\frac{1}{2}\mathbf{p}_i^2 & \text{if } k = 5. \end{cases}$$

By introducing a Lagrangian, it can be shown that the solution of the minimization problem is given by the eigenvector of $\mathbf{B}$ which corresponds to the smallest eigenvalue. The eigenvector can be found by singular value decomposition of the matrix $\mathbf{B}$.

$$\mathbf{B} = UDV^T \tag{3.38}$$

U is a matrix where each column represents an eigenvector of $\mathbf{B}$ and D is a diagonal matrix where each diagonal value represent the corresponding eigenvalue of the eigenvectors in U.

3.6.7 Geometric Algebra vs Vector Algebra

For comparison, an example of a sphere constructed from 4 points are presented. First, the parameters of the sphere are obtained by using vector algebra in Euclidean space. Then, the parameters of the sphere are obtained by using geometric algebra in conformal space.

Sphere Example with Vector Algebra

Given four non-coplanar points in euclidean space:

$$\begin{gathered} x_1\mathbf{e}_1 + y_1\mathbf{e}_2 + z_1\mathbf{e}_3 \\ \vdots \\ x_4\mathbf{e}_1 + y_4\mathbf{e}_2 + z_4\mathbf{e}_3 \end{gathered}$$

The equation of the sphere that passes through the points is found by solving the determinant of the following matrix:

$$\begin{vmatrix} (x^2+y^2+z^2) & x & y & z & 1 \\ (x_1^2+y_1^2+z_1^2) & x_1 & y_1 & z_1 & 1 \\ (x_2^2+y_2^2+z_2^2) & x_2 & y_2 & z_2 & 1 \\ (x_3^2+y_3^2+z_3^2) & x_3 & y_3 & z_3 & 1 \\ (x_4^2+y_4^2+z_4^2) & x_4 & y_4 & z_4 & 1 \end{vmatrix} = 0 \tag{3.39}$$

The parameters are extracted by writing the result on standard form:

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2 \tag{3.40}$$

Sphere Example with Geometric Algebra

Given four non-coplanar points in conformal space:

$$\begin{gathered} P_1 = \mathbf{p}_1 + \tfrac{1}{2}\mathbf{p}_1^2\mathbf{e}_\infty + \mathbf{e}_0 \\ \vdots \\ P_4 = \mathbf{p}_4 + \tfrac{1}{2}\mathbf{p}_4^2\mathbf{e}_\infty + \mathbf{e}_0. \end{gathered}$$

The equation of the sphere that passes through the points is found by taking the wedge product of the points:

$$S^* = P_1 \wedge P_2 \wedge P_3 \wedge P_4.$$

Taking the dual of the sphere and normalizing gives:

$$S = S^*\mathbf{I}^{-1} = \mathbf{x}_c + \frac{1}{2}(\mathbf{x}_c^2 - r^2)\mathbf{e}_\infty + \mathbf{e}_0.$$

The sphere center $\mathbf{x}_c$ and radius r can be extracted from the conformal vector S.

3.7 Kinematics

A vector with coordinates $\mathbf{v}^A$ in reference frame A can be represented with coordinates $\mathbf{v}^B$ in reference frame B

$$\mathbf{v}^A = \mathbf{R}_B^A \mathbf{v}^B + \mathbf{t}^A \tag{3.41}$$

where $\mathbf{R}_B^A$ is the rotation matrix and $\mathbf{t}^A$ is the translation from $\mathbf{A}$ to $\mathbf{B}$. The entities are illustrated in Figure 3.10

The columns of rotation matrix $\mathbf{R}_B^A$ describe the direction of the axes of reference frame $\mathbf{B}$ in the coordinates of reference frame $\mathbf{A}$.

$$\mathbf{R}_B^A = \begin{bmatrix} \mathbf{x}_B^A & \mathbf{y}_B^A & \mathbf{z}_B^A \end{bmatrix}. \tag{3.42}$$

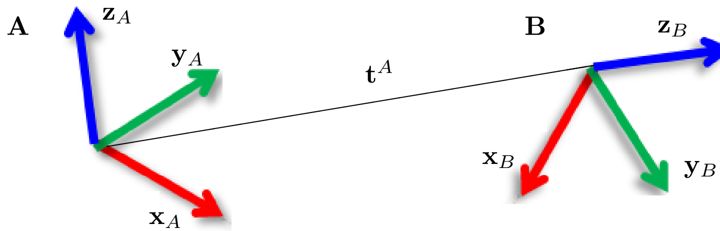


Figure 3.10: The entities of reference frame $\mathbf{A}$ and $\mathbf{B}$ in a 3D space.

Rotation and translation can be combined in a homogeneous transformation matrix

$$\mathbf{T}_B^A = \begin{bmatrix} \mathbf{R}_B^A & \mathbf{t}^A \\ 0 & 1 \end{bmatrix}. \tag{3.43}$$

A homogeneous vector $\tilde{\mathbf{v}}^A = \begin{bmatrix} \mathbf{v}^A \\ 1 \end{bmatrix}$ in reference frame $\mathbf{A}$ can be represented with coordinates $\tilde{\mathbf{v}}^B = \begin{bmatrix} \mathbf{v}^B \\ 1 \end{bmatrix}$ in reference frame $\mathbf{B}$

$$\tilde{\mathbf{v}}^A = \mathbf{T}_B^A \tilde{\mathbf{v}}^B. \tag{3.44}$$

Chapter 4

Solution and Implementation

4.1 Introduction

Much effort has been made in the process of implementing and adapting a robust RANSAC algorithm in this thesis. In order for development and testing to be efficient, a C++ software application with a Graphical User Interface (GUI) was developed in the Qt Integrated Development Environment (IDE). Setting up the software project with relevant libraries for visualization, communication with sensors, point cloud processing and geometric algebra, has been time consuming. Visualization was prioritized when developing the GUI, as visual confirmation of results is a helpful tool when implementing. A lot of informal testing was done from an office desk, as data could be obtained instantly with a Kinect for Xbox One depth sensor. Common objects like ping-pong balls, coffee mugs and the floor could be used to represent primitive shapes.

In this chapter developed software and the RANSAC-based algorithm for primitive shape detection will be presented. Shape modeling and fitting is done in conformal space with geometric algebra. Different strategies for terminating the algorithm are considered. Tactics for real-time tracking are suggested. In addition, a more refined algorithm for detection of several primitives is developed. This algorithm combines several concepts presented in Section 2.4. Lastly, a simple robotic pick-and-place task of primitive objects is implemented for the purpose of demonstrating the algorithms and possible applications.

4.2 Software

The C++ software combines geometric algebra operations, developed algorithms, point cloud capturing and visualization into a GUI program. The program is appended as a digital appendix and the source code for some of the main functions can be found in Appendix B. The program requires a computer running on the *Ubuntu 14.04 LTS* operating system and a working installation of the *Point Cloud Library* (PCL) with all its dependencies installed. Also, the *Versor* open source library for geometric algebra computations and the

libfreenect2 open source library for the Kinect for Xbox One camera is required.

4.2.1 Structure

Figure 4.1 shows the structure of the developed software. Boxes in red indicate the external libraries and where they are used. Boxes in blue indicate features that are controlled by the user through the GUI, these boxes also interacts with the user through the visualization window. The boxes in white are underlying functions in the program.

Visualization is prioritized so that effects of different operations can be understood. It is also useful for verifying the result of shape detection, tracking and segmentation. As the depth sensor of the Kinect for Xbox One output 217 088 3D points, a preprocessing step that reduces the data size is handy such that only relevant data are considered. This preprocessing step mainly consist of cropping point data that fall outside some specified bounds.

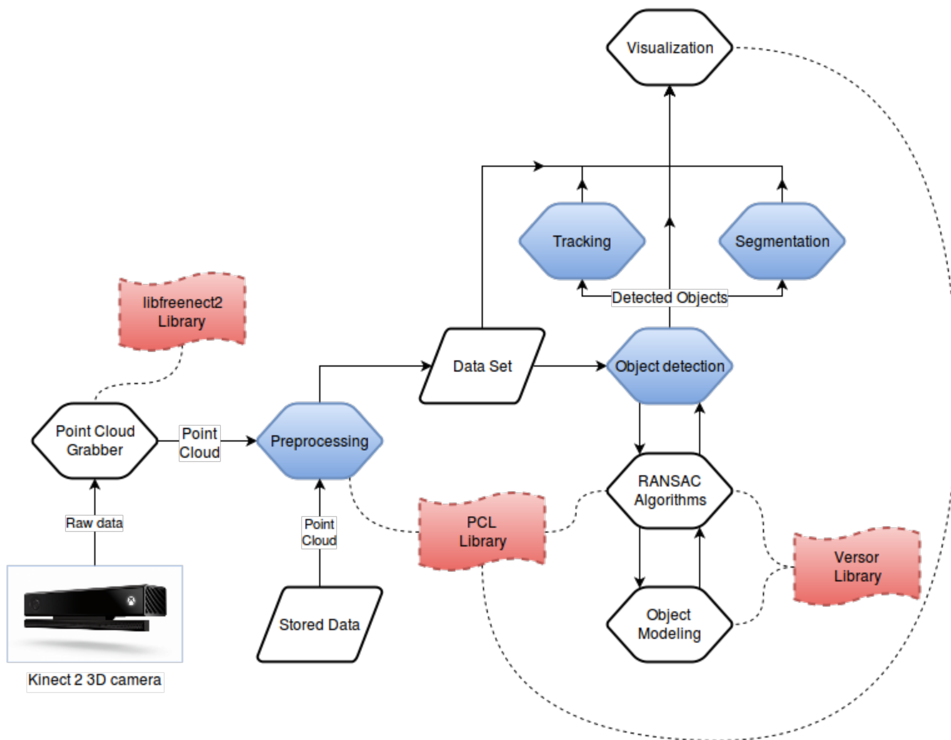


Figure 4.1: Software structure of the developed software. The software takes data from a 3D camera or disk for manipulation. The results are visualized in a visualization window. The red boxes are external software libraries that enables various functionality.

4.2.2 Graphical User Interface

The GUI can be seen in Figure 4.2. It shows a black space for visualization to the right and the program control panel to the left. The implementations of the buttons and visualization window are all declared in the same header file and implemented in the corresponding *.cpp* file. It is under the implementations of the buttons that different functions of the program are called. Furthermore, a callback function for updating the visualization window runs every 20ms, as long as the program is not busy with other processes.

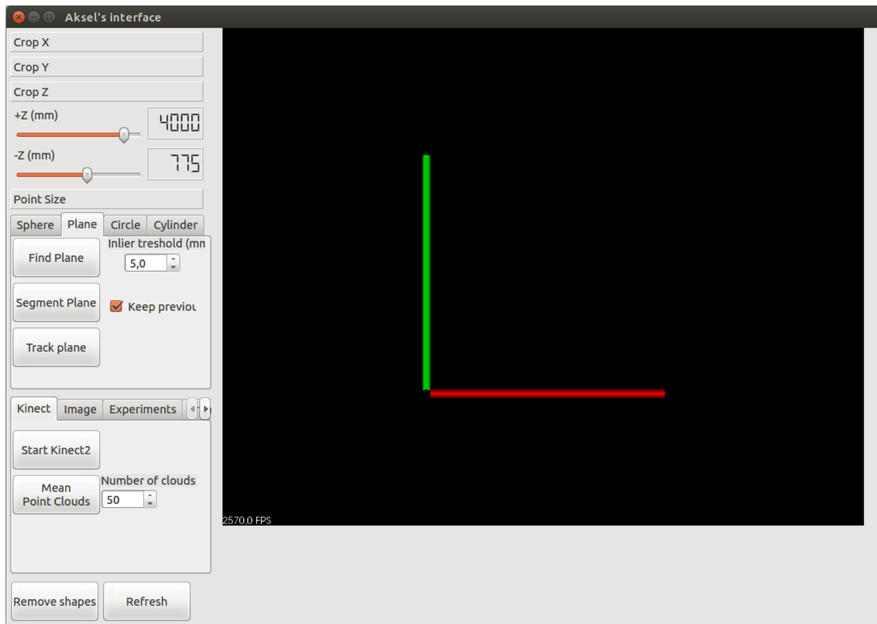


Figure 4.2: The GUI developed in the Qt IDE for the software. The GUI enables interaction with the visualized data set through push-buttons, sliders and editable number-boxes.

4.2.3 External Libraries

The C++ based *Versor* library is developed for geometric algebra computations and formulations. It is an open source project developed by Pablo Colapinto [38]. The features of this library is mainly used for shape modeling, which will be described later in this chapter.

The *Point Cloud Library* is a open source library developed for image- and point cloud processing and visualization. It contains implementations of all the features presented in Section 3.3 and many more. There are many contributors and sponsors to this project. The library's founders, Radu Bogdan Rusu and Steve Cousins, gives a description of the motivation and features of the project in [1]. The library is mainly used for handling point clouds and visualization.

The *libfreenect2* library is an open source grabber for the Kinect for Xbox One camera. It is developed by Lingzhu Xiang and can be found online [https://github.com/OpenKinect/libfreenect2]. The library allows for control of the camera and access to the data streams.

4.3 Shape modeling

One of the focuses of this thesis is to explore the use of geometric algebra when RANSAC is applied to geometrical problems. Planes and spheres are described by vectors in conformal space, and are easily manipulated with geometric operations. The use of geometric algebra is therefor considered superior to vector algebra when working with these objects. More intricate shapes, like cylinder, cone and torus, cannot be described by a single vector in conformal space and are therefore not necessarily easier to model with geometric algebra. However, a novel description of a cylinder is suggested that is based on two spheres in conformal space. All the presented primitives are implemented in the **objects.h** file found in Appendix B.

4.3.1 Plane

A plane is described in conformal space by the vector:

$$\pi = \mathbf{n} + d\mathbf{e}_\infty \tag{4.1}$$

$\mathbf{n}$ is the normal vector of the plane in Euclidean space and d is the distance from the plane to the origin in the direction of the normal vector.

A plane can be constructed from three points in conformal space by taking the wedge product:

$$\pi^* = P_1 \wedge P_2 \wedge P_3 \wedge \mathbf{e}_\infty. \tag{4.2}$$

Taking the dual and normalizing gives the plane on the IPNS form:

$$\pi = \pi^* I^{-1} = \mathbf{n} + d\mathbf{e}_\infty \tag{4.3}$$

4.3.2 Sphere

A sphere is described in conformal space by the vector:

$$S = \mathbf{x}_c + \frac{1}{2}(\mathbf{x}_c^2 - r^2)\mathbf{e}_\infty + \mathbf{e}_0 \tag{4.4}$$

$\mathbf{x}_c$ is the sphere center in Euclidean space and r is the sphere radius.

A sphere can be constructed from four non-coplanar points in conformal space by taking the wedge product:

$$S^* = P_1 \wedge P_2 \wedge P_3 \wedge P_4 \tag{4.5}$$

Taking the dual and normalizing gives the sphere on the IPNS form:

$$S = S^* I^{-1} = \mathbf{x}_c + \frac{1}{2}(\mathbf{x}_c^2 - r^2)\mathbf{e}_\infty + \mathbf{e}_0 \tag{4.6}$$

4.3.3 Circle

A circle is described in conformal space as the intersection of two spheres

$$Z = S_1 \wedge S_2 \tag{4.7}$$

or, by the outer product of three points that lie on it

$$Z^* = P_1 \wedge P_2 \wedge P_3 \tag{4.8}$$

The center of the circle can be found by the product:

$$P_c = Z\mathbf{e}_\infty Z \tag{4.9}$$

Normalization gives:

$$P_c = \mathbf{x}_c + \frac{1}{2}\mathbf{x}_c^2\mathbf{e}_\infty + \mathbf{e}_0 \tag{4.10}$$

The radius of the circle is the distance between the center and a point on the circle. This distance can be found by using the inner product in conformal space:

$$r = \sqrt{-2(P_c \cdot P_1)}. \tag{4.11}$$

4.3.4 Cylinder

Unlike spheres and planes, a cylinder cannot be uniquely defined from points on its surface without some additional assumptions or constraints. A line and a radius can define a cylinder. The line can be infinite, giving the cylinder infinite length, or it can have some absolute length. For obtaining the line and the radius from point-data, two approaches were considered.

Approach 1: Circle-Plane

The centerline of a cylinder is defined as the normal of a plane going through the center of a circle lying on the plane. The cylinder radius is defined as the radius of the circle. Figure 4.3a shows a circle on a plane, the normal of the plane is placed in the center of the circle, defining the position and direction of the cylinder. The same three points can define the plane and the circle, this is easily done in conformal space.

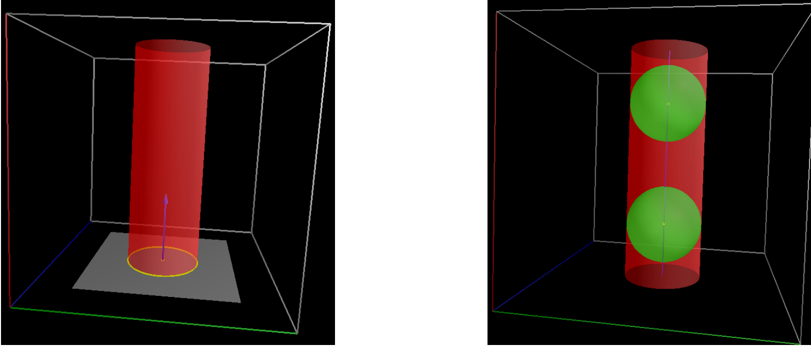
Given three points:

$$\begin{aligned} P_1 &= \mathbf{x}_1 + \tfrac{1}{2}\mathbf{x}_1^2\mathbf{e}_\infty + \mathbf{e}_0 \\ &\vdots \\ P_3 &= \mathbf{x}_3 + \tfrac{1}{2}\mathbf{x}_3^2\mathbf{e}_\infty + \mathbf{e}_0, \end{aligned}$$

the circle and plane are constructed

$$\begin{aligned} Z^* &= P_1 \wedge P_2 \wedge P_3 \\ \pi^* &= P_1 \wedge P_2 \wedge P_3 \wedge \mathbf{e}_\infty. \end{aligned}$$

The normal of the plane, the center of the circle and the radius are found as described in Section 4.3.1 and 4.3.3.



(a) Approach 1: Circle-plane representation of a cylinder **(b)** Approach 2: Sphere-sphere representation of a cylinder

Figure 4.3: The two approaches considered for defining a cylinder. Approach 1 is based on a circle and a plane in conformal space. Approach 2 is based on two spheres in conformal space.

Approach 2: Sphere-Sphere

For cylinders that have no thickness, a sphere with the same radius as the cylinder would fit inside the cylinder with the sphere center coinciding with the centerline of the cylinder. By placing two such spheres inside the same cylinder, the centerline of the cylinder can be defined as the line going through the two sphere centers. Figure 4.3b shows two spheres with equal radius encapsuled by a cylinder, the cylinder-axis goes through the two sphere centers.

Given eight points:

$$P_1 = \mathbf{x}_1 + \tfrac{1}{2}\mathbf{x}_1^2\mathbf{e}_\infty + \mathbf{e}_0$$
$$\vdots$$
$$P_8 = \mathbf{x}_8 + \tfrac{1}{2}\mathbf{x}_8^2\mathbf{e}_\infty + \mathbf{e}_0,$$

two spheres can be constructed

$$S_1^* = P_1 \wedge P_2 \wedge P_3 \wedge P_4$$
$$S_2^* = P_5 \wedge P_6 \wedge P_7 \wedge P_8.$$

Their dual are found to extract parameters:

$$S_1 = \mathbf{x}_{c1} + \tfrac{1}{2}(\mathbf{x}_{c1}^2 - r^2)\mathbf{e}_\infty + \mathbf{e}_0$$
$$S_2 = \mathbf{x}_{c2} + \tfrac{1}{2}(\mathbf{x}_{c2}^2 - r^2)\mathbf{e}_\infty + \mathbf{e}_0,$$

where r is the cylinder radius and $\mathbf{x}_{c1}$ and $\mathbf{x}_{c2}$ are the sphere centers. The centerline of the cylinder in Euclidean space is defined in parametric form by:

$$\mathbf{l}(t) = \mathbf{x}_{c1} + (\mathbf{x}_{c2} - \mathbf{x}_{c1})t, \tag{4.12}$$

where t is the parameter along the line.

4.4 Primitive Shape Detection

RANSAC-based algorithms searches for, and scores, a model within a noisy data set until some stopping criteria is satisfied. As described in Chapter 2, several approaches exist. Which approach to pick depends on what is known about the data set and the model. The focus of this thesis is manipulation of point clouds from 3D camera sensors. Given that the data obtained are noisy point clouds, three cases were considered:

- **Case 1**: The number of inliers for a primitive is known.
- **Case 2**: Nothing is known about the primitives represented in the point cloud.
- **Case 3**: The point cloud contains a representation of a specific primitive with one (or more) known parameter(s).

Each case demand different criteria to be fulfilled to terminate the algorithm. However, the initialization and inlier classification of each primitive are equal in all the cases. The implementation of this will be presented first in this section. Thereafter, the termination strategy for each case is presented. Lastly, some probabilistic relations for detecting a primitive in a point cloud are presented. The presented algorithms are implemented in the **ransac.h** file found in Appendix B.

4.4.1 Initialization and Inlier Classification

Planes, spheres and cylinders have been implemented. They are all initialized in conformal space with geometric algebra. The inlier classification is done in conformal space for planes and spheres as geometric algebra offers easy computation of distances between points and these objects. However, since cylinders cannot be represented as a single vector in conformal space, there is no simple way of calculating distance between a point and a cylinder with geometric algebra. Therefore, this calculation is done in Euclidean space with vector algebra.

Plane

Given a point cloud $\mathscr{P}$ of $N > q$ points in 3D Euclidean space, a plane is to be identified as a subset of $\mathscr{P}$. The error tolerance T is given.

1. Randomly pick a subset of $q = 3$ points from $\mathscr{P}$ and project them to conformal space:

$$\begin{aligned} P_1 &= \mathbf{x}_1 + \tfrac{1}{2}\mathbf{x}_1^2\mathbf{e}_\infty + \mathbf{e}_0 \\ &\vdots \\ P_q &= \mathbf{x}_q + \tfrac{1}{2}\mathbf{x}_q^2\mathbf{e}_\infty + \mathbf{e}_0. \end{aligned}$$

2. Define a plane using the approach described in Section 4.3.1.

3. Calculate the distance d between the plane π and each point P_i in $\mathscr{P}$, using the inner product in conformal space:

$$d = \pi \cdot P_i. \tag{4.13}$$

P_i is classified as a inlier point if the distance d between the point and the plane satisfies the condition:

$$-T \leq d \leq T \tag{4.14}$$

Sphere

Given a point cloud $\mathscr{P}$ of $N > q$ points in 3D Euclidean space, a sphere is to be identified as a subset of $\mathscr{P}$. The error tolerance T is given.

1. Randomly pick a subset of $q = 4$ points from $\mathscr{P}$ and project them to conformal space:

$$P_1 = \mathbf{x}_1 + \tfrac{1}{2}\mathbf{x}_1^2\mathbf{e}_\infty + \mathbf{e}_0$$
$$\vdots$$
$$P_q = \mathbf{x}_q + \tfrac{1}{2}\mathbf{x}_q^2\mathbf{e}_\infty + \mathbf{e}_0.$$

2. Define a sphere using the approach described in Section 4.3.2.
3. Define the center of the sphere as a point:

$$P_c = \mathbf{x}_c + \frac{1}{2}\mathbf{x}_c^2\mathbf{e}_\infty + \mathbf{e}_0 \tag{4.15}$$

4. Calculate the distance d between the sphere center P_c and each point P_i in the data set using the inner product in conformal space:

$$d = \sqrt{-2(P_c \cdot P_i)}. \tag{4.16}$$

P_i is classified as a inlier point if the distance d between the point and the sphere center satisfies the condition:

$$r - T \leq d \leq r + T \tag{4.17}$$

Cylinder

Given a point cloud $\mathscr{P}$ of $N > q$ points in 3D Euclidean space, a cylinder is to be identified as a subset of $\mathscr{P}$. The error tolerance T is given.

1. Randomly pick a subset of q ($q = 3$ for approach 1 and $q = 8$ for approach 2) points from $\mathscr{P}$ and project them to conformal space:

$$P_1 = \mathbf{x}_1 + \tfrac{1}{2}\mathbf{x}_1^2\mathbf{e}_\infty + \mathbf{e}_0$$
$$\vdots$$
$$P_q = \mathbf{x}_q + \tfrac{1}{2}\mathbf{x}_q^2\mathbf{e}_\infty + \mathbf{e}_0.$$

2. Define a cylinder using one of the approaches described in Section 4.3.4.
3. Find two points on the center line of the cylinder in Euclidean space. If approach 1:

$$\begin{aligned} \mathbf{c}_1 &= \mathbf{x}_c \\ \mathbf{c}_2 &= \mathbf{c}_1 + \mathbf{n}. \end{aligned} \tag{4.18}$$

   If approach 2:

$$\begin{aligned} \mathbf{c}_1 &= \mathbf{x}_{c1} \\ \mathbf{c}_2 &= \mathbf{x}_{c2}. \end{aligned} \tag{4.19}$$

4. Calculate the distance d between the line that runs through $\mathbf{c}_1$ and $\mathbf{c}_2$ and each point $\mathbf{x}_i$ in the data set using:

$$d = \frac{\mid (\mathbf{x}_i - \mathbf{c}_1) \times (\mathbf{x}_i - \mathbf{c}_2) \mid}{\mid \mathbf{c}_1 - \mathbf{c}_2 \mid} \tag{4.20}$$

   $\mathbf{x}_i$ is classified as a inlier-point if the distance between the point and the line satisfies the condition: $r - T \leq d \leq r + T$.

5. The cylinder edges are found by picking the two inlier points that lies farthest away from another, $\mathbf{x}_b$ and $\mathbf{x}_t$, and projecting them to the centerline of the cylinder. First the vector from a point $\mathbf{c}_1$ on the center-axis to $\mathbf{x}_b$ and $\mathbf{x}_t$ are found:

$$\begin{aligned} \mathbf{cx}_b &= \mathbf{x}_b - \mathbf{c}_1 \\ \mathbf{cx}_t &= \mathbf{x}_t - \mathbf{c}_1. \end{aligned} \tag{4.21}$$

   Projecting onto the cylinder center-axis, $\mathbf{n}$:

$$\begin{aligned} \mathbf{x}_{eb} &= (\mathbf{cx}_b \cdot \mathbf{n})\mathbf{n} \\ \mathbf{x}_{et} &= (\mathbf{cx}_t \cdot \mathbf{n})\mathbf{n} \end{aligned} \tag{4.22}$$

6. The length of the cylinder is the distance between the cylinder edges:

$$l = \mid \mathbf{x}_{et} - \mathbf{x}_{eb} \mid \tag{4.23}$$

4.4.2 Termination Strategy

Case 1: Known number of inliers for a primitive

In this case, the score is simply the number of inliers. A primitive is accepted and iteration is terminated if the number of inliers I is greater than some threshold t.

Case 2: Nothing is known about the primitive

In this case a predetermined number of iterations k is defined. After k iterations, the highest scoring primitive is accepted and the algorithm is terminated.

For a plane, it is sufficient to define the score as the number of inliers. However, for a sphere and a cylinder this approach would always return the larges primitive in the point

cloud. To avoid this, the density of inliers are considered when scoring the primitives. The density ρ is defined as the number of inliers I per surface area A of the primitive. For a sphere:

$$\rho = \frac{I}{A_s} = \frac{I}{4\pi r^2}, \tag{4.24}$$

and for a cylinder:

$$\rho = \frac{I}{A_c} = \frac{I}{2\pi r l} \tag{4.25}$$

Furthermore, if unprocessed point clouds from a 3D camera are considered, the density of inliers will also depend on the distance from the sensor. This is because the point density is higher closer to the sensor. The number of points n_A per unit area in relation to the z-distance from the sensor is defined as:

$$n_A = az^b, \tag{4.26}$$

where a and b are some sensor-dependent constants which can be found experimentally. A scoring function, which takes into account both the surface area of the primitive and its distance from the sensor, is defined:

$$f(\rho, z) = \frac{\rho}{n_A} = \frac{I}{Aaz^b}. \tag{4.27}$$

If only the surface area that is visible from the sensor is taken into account (for spheres and cylinder this is approximately equal to half their surface area), the denominator for this scoring function approximates the expected number of inliers for a primitive. This means that the functions returns values between 0 and 1.

Case 3: A parameter of the primitive is known

When the point cloud contains a specific primitive with known parameters, the iteration continues until the primitive is detected. The number of iterations k will in this case be undefined. Using the same scoring function as for case 2 implies the need of defining a threshold t for deciding whether the primitive is accepted or not. In addition, after the initialization of a primitive, its parameters can be examined before counting inliers and scoring. A primitive is only scored if its parameters satisfies the threshold $t \in [0, 1]$. Thus, scoring is only done for a fraction of the initialized primitives with the intention of computational savings.

Algorithm for a sphere or a cylinder with specified radius r_t:

1. Initialize the primitive as described in Section 4.4.1.
2. Classify inliers if:

$$r_t(1 - t) \leq r \leq r_t(1 + t), \tag{4.28}$$

repeat step 1 if not.

3. Calculate the score

$$f(\rho, z) = \frac{\rho}{n_A}$$

and terminate if:

$$f(\rho, z) \geq 1 - t. \tag{4.29}$$

Repeat step 1-2 if not.

4.4.3 Probability

Given a point cloud $\mathscr{P}$ of N points containing a shape $\mathscr{S}$ of n points, a minimal subset of q points required to uniquely define the shape is randomly picked. The probability of detecting the shape $\mathscr{S}$ in a single pass of the algorithm is:

$$P(n) = \frac{\binom{n}{q}}{\binom{N}{q}} \approx \left(\frac{n}{N}\right)^q. \tag{4.30}$$

The probability of detecting the shape after k iterations is:

$$P(n, k) = 1 - (1 - P(n))^k. \tag{4.31}$$

The number of iterations K required to detect a shape of size n with a probability $P(n, K) \geq p_t$:

$$K \geq \frac{\ln(1 - p_t)}{\ln(1 - P(n))}. \tag{4.32}$$

The denominator can be approximated by its Taylor series if $P(n)$ is small:

$$K \approx \frac{-\ln(1 - p_t)}{P(n)}. \tag{4.33}$$

Thus, the number of iterations required to detect a shape is directly correlated to the fraction $\frac{n}{N}$. This confirms the intuition that higher percentage of outliers requires higher number of iterations.

4.5 Tracking

Tracking an object in a data stream from a 3D camera is in this thesis simplified to the problem of detecting the same primitive in multiple point clouds. In order for the tracking to be performed in real-time, the detection of a primitive have to be successful before a new point cloud is received form the 3D camera. As 3D cameras can deliver point clouds at rates of 30-60 frames per second (FPS), detections have to be completed in the 17-33ms range. The time of detection depends on the computational power of the computer and computational cost of the algorithm.

In order to keep the computational cost of the algorithm somewhat constant, a preemptive strategy is applied. The stopping criteria of Case 2 in Section 4.4.2 is chosen, which keeps the number of iterations k constant. The size of the point cloud is also reduced to improve the probability of detection and keep the computational cost of every iteration as low as possible. Only points within a certain distance d to the previous known position of the primitive are considered. Thus, the change in position, in two consecutive point clouds in the stream, have to be within the bounds of d for tracking to be successful. Hence, there is a trade-off between computational cost and maximum allowed change in position when choosing the value of d.

For a plane, only points P_i that satisfies the condition

$$\pi \cdot P_i \leq d \tag{4.34}$$

are considered. π is the plane in the previous point cloud. For a sphere the condition is

$$\sqrt{-2(P_c \cdot P_i)} \leq r_t + d, \tag{4.35}$$

where P_c and r_t is the center and radius of the sphere in the previous point cloud. For a cylinder the condition is

$$\frac{\mid (\mathbf{x}_i - \mathbf{c}_1) \times (\mathbf{x}_i - \mathbf{c}_2) \mid}{\mid \mathbf{c}_1 - \mathbf{c}_2 \mid} \leq r_t + d, \tag{4.36}$$

where $\mathbf{c}_1$ and $\mathbf{c}_2$ are two points on the centerline of the cylinder in the previous point cloud and r_t is the previous radius.

The tracking algorithm:

1. Discard all point that do not satisfy the proximity condition
2. Initialize the primitive as described in Section 4.4.1.
3. Classify inliers and compute the score. For spheres and cylinders, only if
$$r(1-t) < r_t < r(1+t).$$
4. Terminate and return the highest scoring primitive if the maximum number of iterations k is reached or if a primitive with a score
$$f(\rho, z) > 1 - t$$
is found. If not, repeat step 2 and 3.

4.6 Multiple Shape Detection

The goal of this algorithm is to detect multiple primitives in a point cloud with a single run of the algorithm. Efforts have been made to make the algorithm robust and computationally cheap. The sampling strategy is inspired by Schnabel et al. in [6], where the

point cloud is organized in an octree structure. However, while the suggested sampling strategy aims to iterate through all nodes of the octree, their sampling strategy is more sophisticated as it keeps track of which levels of the octree is more probable to produce good detections. The shape modeling and scoring is performed as described in previous sections. The algorithm is only implemented for spheres due to the limited time available for this project, but additional primitives are possible. The presented algorithm is implemented in the **ransac.h** filer found in Appendix B.

Even though the motivation for the algorithm is to establish a robust way to detect several primitives in a point cloud, it is discovered that the suggested sampling strategy can in some cases drastically improve the performance of detection. An example is presented and discussed in Section 4.6.3.

4.6.1 Overview

Given a point cloud $\mathcal{P}$ of points $\{P_1, \dots, P_N\}$, the point cloud is organized in an octree structure. The size of the octree is set such that the root node of the octree spans over the whole point cloud. The root node is subdivided into smaller nodes until nodes with sides of length d_{res} are obtained. The resolution d_{res} of the octree should be chosen such that there exist a node that contains only inlier points of the smallest shape in the point cloud.

A minimal subset of q points is picked within a node in the octree. The size of q is determined by the largest q for the primitives considered. All the considered primitives $\mathcal{S}$ are initialized from the minimal subset and their inliers I are classfied as discussed in Section 4.4.1. The primitives $\mathcal{S}$ are scored by evaluating the scoring function $f(E(I)) = \frac{I}{E(I)}$, where $E(I)$ is the expected number of inliers for a primitive as discussed in Case 2 in Section 4.4.2. The subset of inliers for a primitive $\mathcal{P}_{\mathcal{S}_i} \subset \mathcal{P}$ are extracted if the score is larger than some threshold t.

By systematically repeating the initialization and scoring procedure for every node at every level of the octree all shapes are detected. Furthermore, by starting the procedure at the highest level of the octree, large shapes are detected and extracted from the point cloud first. When the procedure is conducted for all the nodes in the octree, the extracted set of primitives $\mathcal{S} = \{\mathcal{S}_1, \dots, \mathcal{S}_m\}$ are returned with their corresponding subsets of inliers $\mathcal{P}_{\mathcal{S}_1}, \dots, \mathcal{P}_{\mathcal{S}_m} \subset \mathcal{P}$.

4.6.2 Sampling strategy

Shapes are a local phenomenon, therefore the probability that two points belong to the same shape is higher the smaller the distance between the points. By organizing the points in an octree, spatial proximity between points are established very effectively [6]. If a shape is present in a point cloud, one can by observation argue that in some level of the octree there exists a node that only contain points that belong to that shape. This is true when the resolution of the octree is sufficiently small. By randomly picking a minimal subset of points from a node that only contain points from a single shape, this minimal subset will represent the shape. However, since sensed data are considered, some additional consideration have to be taken. Even though all the points within a node fall between the inlier bounds of a shape, there is no guaranty that every minimal combination of these points will give an accurate description of the shape. Therefore, the number of iterations k for each

node will be in the range of $2 - 4$ instead of 1 to ensure that no shape is overlooked. By starting at the highest level of the octree, the largest shapes will be detected and extracted first, which reduces the number of points to consider in later iterations.

The number of nodes present in the octree dictates the number of iterations. The number of nodes are dependent of the size of the root node and the resolution of the octree, as shown in Section 3.3.1. When deciding the depth of the octree the smallest shapes present in the point cloud are considered. Setting the resolution such that the smallest node will contain $\leq \frac{1}{4}$ of the volume of the shape is considered sufficient for obtaining a node where most of the points belong to the shape.

When a point cloud is organized in an octree, many of the nodes will be empty or consist of few points. This is the case because the root node will in many cases span over large regions that does not contain any points. By only considering nodes that contain more than q points, the number of iterations can be reduced considerably.

4.6.3 Performance

The power of this sampling strategy is best illustrated with an example. A point cloud of $49\,396$ points can be seen in Figure 4.4. The point cloud contains a sphere with radius $r = 0.02$m and 109 inlier points, which is to be detected. The point cloud fits inside an octree with a root node with sides of 2.56m. According to Section 3.3.1 a resolution of $d_{\text{res}} = 0.01$m suggest a total of $2\,396\,745$ nodes, and with $k = 2$ results in $4\,793\,490$ iterations. However, in this case only $10\,118$ nodes are occupied with more than $q = 4$ points, resulting in a total of $20\,236$ iterations. Assuming that at least one of the nodes has a inlier ratio ≥ 0.95, detection of the sphere is achieved with a probability of 0.966. Detecting the sphere with the same probability using a random sampling scheme as presented in Section 4.4.3, would require $142\,612\,308\,700$ iterations. In this particular case the new sampling scheme is more than $7\,000\,000$ times more effective than a random sampling scheme. However, this measure is highly dependent on the point density and the required depth of the octree.

Figure 4.4: A point cloud of $49\,396$ points showing a ping-pong ball placed on the floor. For a random sampling strategy it would require approximately $7\,000\,000$ times more iterations to detect the sphere, compared to the suggested sampling strategy. (Obtained with the Kinect for Xbox One)

4.6.4 Scoring

Inspired by the R-RANSAC [23] algorithm described in Section 2.4, the cost of scoring is reduced by only scoring against a subset $\mathcal{P}_{sub}$ of points from the point cloud $\mathcal{P}$. The original point cloud $\mathcal{P}$ is randomly divided into 4 disjoint subsets of equal size and point density. When a primitive is to be scored, it is scored against one of these subsets. Only if it the score is above a threshold $t_0 = \frac{t}{4}$ the primitive will be scored against the whole point cloud $\mathcal{P}$. The primitive is accepted if this score is $\geq t$.

4.7 Robotic Pick-and-Place

In this robotic pick-and-place task, a 3D camera is used for primitive shape detection and localisation. The primitives positions and parameters are used for robot-camera calibration and robot positioning. The goal is to pick-and-place arbitrarily positioned ping-pong balls inside an arbitrarily positioned hollow tube. The ping-pong balls are detected as spheres, while the tube is detected as a cylinder.

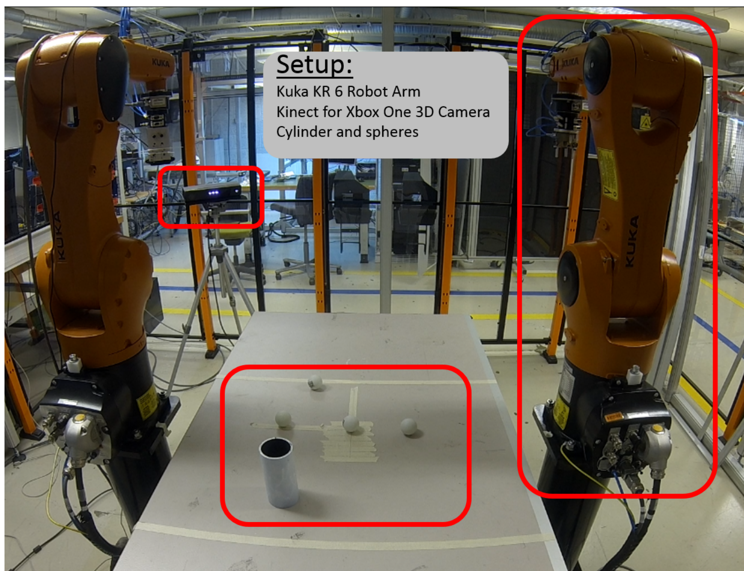


Figure 4.5: The setup of the robotic pick-and-place demonstration. The 3D camera is used for detecting the spheres and the cylinder. The information is sent to the robot arm, which is used to place the spheres in the cylinder.

Figure 4.5 show the setup of the demonstration. A Kinect for Xbox One 3D camera is used to obtain 3D point clouds of the scene. A *KUKA KR 6* robot arm is used for pick-and-place operations. In addition, the robot is controlled by a *KUKA KR CR4* controller. The communication between the robot controller and the software described in Section 4.2 is done through the *Robot Operating System* (ROS) on a computer running the *Ubuntu 14.04 LTS* operating system.

4.7.1 Information Flow and Software

Figure 4.6 show the information flow in the setup. The operations in the purple box represent the software and GUI described in Section 4.2. When an object is detected, the position of the object is transformed to the robot coordinate frame. This position is sent to ROS, which sends necessary move commands to the KR C4 robot controller. Commands for opening and closing the gripper is also sent from ROS. The KR C4 robot controller does the trajectory planning and execution of the robot movements. When a move is successful, this is communicated back to ROS, which sends the next move command, if there are any.

The structure- and information flow of ROS and the robot controller is rather complicated and is not the scope of this thesis. Due to the work of fellow students Asgeir Bjørkedal and Kristoffer Larsen at the Institute of Production and Quality Engineering, easy control of the robot is enabled. Once connected to ROS, their work enable manipulation of the robot with two simple commands. One *move* command, which moves the end effector of the robot to a desired position with a desired orientation, and a command for opening and closing the gripper. These were the only two commands used for controlling the robot. Thanks to Bjørkedal and Larsen, not much time or effort were spent on robot control.

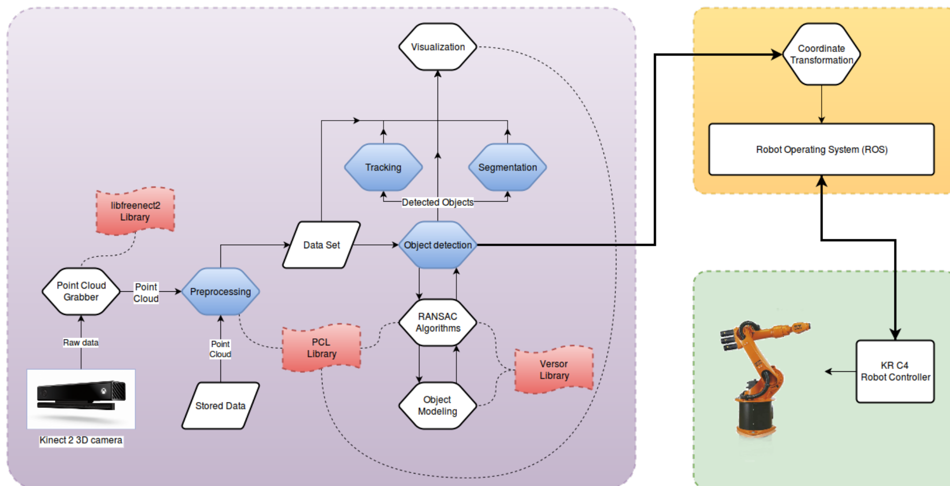


Figure 4.6: Information flow of the demonstrator. Positions of detected shapes are sent to ROS which gives commands to the robot controller.

4.7.2 Calibration

Locating the robot coordinate frame in the camera coordinate frame is necessary in order to transform camera coordinates to robot coordinates. This is done in a calibration step that is necessary to perform every time the camera is repositioned. A standard method of doing this is identifying the position and orientation of a specific object in both coordinate

systems, and then calculate the transformation between the two coordinate systems based on the pose of the object.

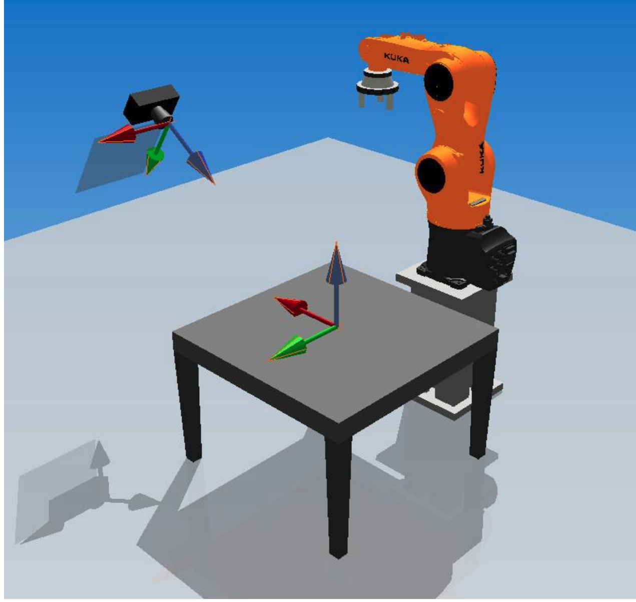


Figure 4.7: The robot cell used for the demonstrator. The robot coordinate frame is placed in the middle of the table and have to be defined in the camera frame.

Figure 4.7 show how the camera can be positioned in the robot cell. It also show how the robot coordinate frame is defined on the work-table. The robot origin O^R is located at the center of the table, the $\vec{z}^R$-axis is normal to the table and the $\vec{x}^R$ and $\vec{y}^R$-axis are parallel with the table. Because the axes are perpendicular to each other, this coordinate system can be fully defined in the camera frame by locating the origin and the direction of two of the axes. The origin is found by placing a small sphere with radius r in the robot origin and detecting the center position $\mathbf{x}_1$ of the sphere in the camera frame:

$$O_c^R = \mathbf{x}_1.$$

The robot z-axis is defined by detecting the plane that represent the table and using the normal $\mathbf{n}$ of the plane.

$$\vec{z}_c^R = \mathbf{n}.$$

The x-axis is found by placing a small sphere with radius r on the robot x-axis and detect the center position $\mathbf{x}_2$ of the sphere in the camera frame:

$$\vec{x}_c^R = \frac{\mathbf{x}_2 - \mathbf{x}_1}{\mid \mathbf{x}_2 - \mathbf{x}_1 \mid}.$$

Finally, the y-axis is defined as the cross product between the x-axis and the z-axis:

$$\vec{y}_c^R = \vec{x}_c^R \times \vec{z}_c^R.$$

Thus, the homogeneous transformation matrix between the camera frame and the robot frame is defined as:

$$\mathbf{T}_c^R = \begin{bmatrix} \vec{x}_c^R & \vec{y}_c^R & \vec{z}_c^R & O_c^R \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -r \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.37}$$

Here, the radius r of the sphere is compensated for.

4.7.3 Place Position

The place position $\mathbf{x}_{\text{place}}^R$ for this demonstration is right above a cylinder placed somewhere on the work-table. This means that the ”top” end of the cylinder have to be located in the robot frame. In order to define the place position, the cylinder edges is located in the camera. This is done as described in Section 4.4. Once the cylinder edges $\mathbf{x}_{eb}$ and $\mathbf{x}_{et}$ are located in the camera frame, they are transformed to the robot frame:

$$\tilde{\mathbf{x}}_{eb}^R = (\mathbf{T}_c^R)^{-1}\tilde{\mathbf{x}}_{eb}$$

$$\tilde{\mathbf{x}}_{et}^R = (\mathbf{T}_c^R)^{-1}\tilde{\mathbf{x}}_{et}$$

The edge with the larges z-value is recognized as the ”top” of the cylinder. Thus, the place position for the robot is defined.

4.7.4 Pick Position

The robot is to pick ping-pong balls that are arbitrarily located on the work-table. Hence, the center positions of the spheres have to be located in the robot frame. The center position of the spheres are located in the camera frame by using the algorithm for multiple shape detection described in Section 4.6. Once the sphere centers $\mathbf{x}_{c,i}$, $i = 1 \ldots n$ are located in the camera frame they are transformed to the robot frame:

$$\tilde{\mathbf{x}}_{c,i}^R = (\mathbf{T}_c^R)^{-1}\tilde{\mathbf{x}}_{c,i}, \quad i = 1 \ldots n$$

4.7.5 Automatic Execution

The calibration and the definition of place position is two rather manual operations, as these require human interaction to be performed. However, in a repetitive pick-and-place application these steps only need to be performed once. After this, objects can be detected, picked and placed in an automatic continuous manner. This is implemented as illustrated in the pseudocode below.

while Not terminated **do**
  Detect spheres $\mathbf{x}_{c,i}, \quad i = 1 \ldots n$
  Transform coordinates $\tilde{\mathbf{x}}_{c,i}^{R} \leftarrow (\mathbf{T}_{c}^{R})^{-1}\tilde{\mathbf{x}}_{c,i}$
  while $i \leq n$ **do**
    Move robot to $\mathbf{x}_{c,i}^{R}$
    Close robot gripper
    Move robot to $\mathbf{x}_{\text{place}}^{R}$
    Open robot gripper
  end while
end while

4.7.6 Video

A video showing the demonstration is produced and added to the digital appendix. The video start with calibration. During this step, multiple shape detection of spheres is demonstrated along with shape detection of a plane. After the transformation between the camera coordinate frame and robot coordinate frame is established, a tube and some ping-pong balls are placed arbitrarily on the table. In the following sequence, the place position is defined. Here, a plane is segmented and removed from the data set before demonstrating shape detection of a cylinder. Next, the spheres on the table are detected and the robot executes the pick-and-place operation.

In the next part of the video, tracking of objects is demonstrated. Two spheres and a plane are tracked simultaneously by using the procedure presented in Section 4.5. The goal is to track the robot coordinate frame by tracking the objects and perform the calculation as described in Section 4.7.2. The camera is moved while tracking and the robot frame is constantly updated. By executing a new pick-and-place operation based on the new transformation matrix between the camera and robot, it is demonstrated that the tracking was successful.

In the last part of the video, the whole process of calibration, place position definition and pick-and-place operation is demonstrated one more time. In addition, it is shown that the pick-and-place operation can be performed continuously once the calibration is done and place position is defined.

Chapter 5

Analysis and Discussion

5.1 Introduction

In this chapter, the conducted experiments and results are presented. The goal of the experiments is to map the performance of the implemented RANSAC-based algorithms described in the previous chapter.

The primitive shape detection algorithm is tested on CAD-models that are sampled to point clouds. The shape detection is performed with different degrees of outliers that are randomly added to the point cloud. During these experiments, it is established that shape detection can be achieved with repetitive precision. It is also seen that specifying one or more parameters of the shape reduces the computational cost. Moreover, it is found that the suggested sphere-sphere approach for initializing a cylinder produces higher accuracy detection and lower computational costs compared to the circle-plane approach.

Tracking of a sphere is tested with point clouds obtained from a 3D camera. Here, results show that the algorithm is able to track a sphere in real-time data streams in the $30 - 60$ FPS range. The observed precision is 3.33mm for both static and movement tracking. However, it is recognized that the precision is dependent on both the size of the shape and the distance from the sensor.

Lastly, the multiple shape detection algorithm is tested on a point cloud obtained from a 3D sensor. It is observed that the detection rates depends on shape size and distance from the camera. This is also true for the observed precision.

5.2 Primitive Shape Detection

The implemented algorithm is tested for spheres and cylinders. The stopping criteria of Case 1 presented in Section 4.4.2 is applied. Hence, the number of inlier points I for both the sphere and the cylinder is known. The algorithm is set to terminate when a primitive consisting of minimum $t \times I$ points is detected, where $t = 0.9$.

5.2.1 Sphere

Two experiments were performed for detection of a sphere. In the first experiment, the sphere radius is considered unknown, meaning that all candidates have to be evaluated. In the second experiment, the sphere radius is known, meaning that only candidates that satisfy the sphere radius is evaluated. The results are presented in Figure 5.1. The figure show logarithmic plots of the total number of iterations before successful detection is achieved, the number of evaluated primitives, the run-time of the algorithm and the position error of the detected sphere center. All the metrics are plotted against the percentage of outliers. The presence of outliers is also graphically illustrated below the plots, where green points are inlier points and blue point are outlier points. For every level of noise considered, 10 test were performed and their average is presented. This is necessary as the algorithm is based on a random sampling strategy.

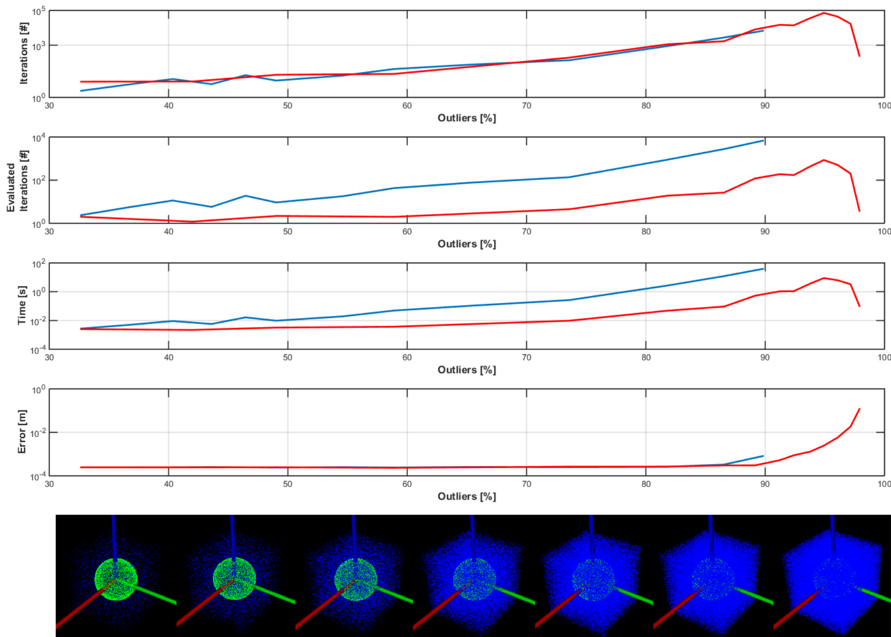


Figure 5.1: Results for the sphere experiments. The blue line represents the experiment where all initialized spheres are evaluated. The red line represents the experiment where only initialized spheres that satisfies the radius threshold is evaluated. The performance of the algorithms is plotted logarithmically for an increasing outlier percentage. As the outlier percentage increases the performance metrics increases exponentially. The outlier percentage is graphically illustrated below the plots.

The result for the first experiment are plotted as the blue line in Figure 5.1. The figure shows how the total number of iterations, total time of detection and error in position of the detected sphere increases exponentially with the outlier percentage. The results are also presented in Table 1 found in Appendix A, along with the calculated probability of

detection according to equation (4.31). The experiment was concluded at a 90% outlier percentage, as average run-time of the algorithm exceeded 38s and the time required for conducting experiments at higher outlier percentages was considered unpractical.

By examining the blue line in Figure 5.1, it can be seen that the error metric is constant and small for outlier percentages up to approximately 90%. At this level, the outlier density is so high that it affects the accuracy of the detection. The outlier density causes spheres that are initialized from outlier points to satisfy the termination criteria of $t \times I$. This causes erroneous detections. As predicted by the probabilistic relations presented in section 4.4.3, the number of iterations increases with the outlier percentage. There is a direct correlation between run-time and number iterations, which is reflected in the "Time" graph in Figure 5.1.

The results for the second experiment are plotted as the red line in Figure 5.1. The results are also presented in Table 2 found in Appendix A, along with the calculated probability of detection according to equation (4.31). It can be seen that both the number of iterations and error in position are approximately the same for the two experiments. This indicates that the evaluation strategy applied in the second experiment does not affect the algorithm's ability to detect shapes, or the algorithm's accuracy. However, the time of detection is reduced considerably as the initialized spheres are only evaluated for a fraction of the iterations. This reveals that evaluating primitives, by classifying inliers and scoring, is computationally costly. The number of evaluated spheres is plotted in the "Evaluated Iterations" graph in Figure 5.1. The ratio between the number of iterations and the number of evaluated iterations is plotted in Figure 5.2. It can be seen that this ratio move toward 0.01 when the outlier percentage increases. For the first experiment, this ratio is always 1 as every iteration is scored, which means that the blue lines plotted in the "Iterations" and "Evaluated Iterations" graph in Figure 5.1 are equal.

Due to the computational savings of the strategy tested in the second experiment, it was possible to test the algorithm for higher outlier percentages without the practical restrictions faced in the first experiment. For outlier percentages above 90%, the error metric increases exponentially. Because of the high outlier density, spheres initialized arbitrarily in the point cloud will contain a sufficient amount of inliers to satisfy the termination criteria of $t \times I$. Thus, it is not possible to separate the actual sphere from the outliers and detection is unsuccessful. At a outlier percentage of approximately 95% the number of iterations peaks. Above this level a sphere initialized anywhere in the point cloud will satisfy the termination criteria, resulting in premature termination.

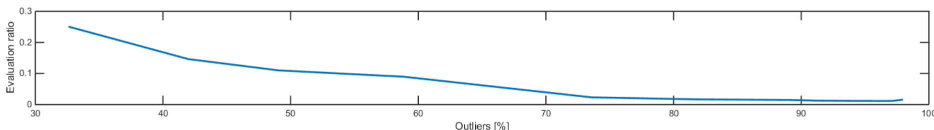


Figure 5.2: The ratio between the number of iterations and the number of evaluated iterations. Major computational savings is achieved as only a fraction of the iterations are evaluated.

5.2.2 Cylinder

The two approaches for initializing a cylinder, discussed in Section 4.3.4, were tested for detection. The cylinder radius is known. Thus, only primitives that satisfy the radius parameter are evaluated, which causes computational savings as established in the sphere experiment. The results are presented in Figure 5.3, which has generally the same setup as Figure 5.1 for the sphere experiments. The figure shows logarithmic plots of the total number of iterations before successful detection is achieved, the number of evaluated primitives, the total time of detection and the angle error of the cylinder center axis. All the metrics are plotted against the percentage of outliers. For every level of noise considered, 10 test were performed and their average is presented.

In the first experiment the plane-circle initialization were tested. The results are plotted as the blue line in Figure 5.3. The results are also presented in Table 3 found in Appendix A. It can be seen that the error metric is small with small variations up to a outlier percentage of approximately 80%. Above 80% outliers, the error metric increases exponentially due to the high outlier density. The number of iterations increases with the outlier percentage. This is intuitively expected, but can not be predicted by the probabilistic relations presented in section 4.4.3. These relations only yield for shapes that are uniquely defined by the points they are initialized from. Unlike spheres and planes, a cylinder cannot be uniquely defined from points on its surface without some additional assumptions or constraints. From Figure 5.3 it can be seen that both the run-time and evaluated iterations are correlated to the number of iterations, which agrees with what is found in the sphere experiments.

In the second experiment the sphere-sphere initialization were tested. The results are plotted as the red line in Figure 5.3. The results are also presented in Table 4 found in Appendix A. Since this initialization is based on spheres, the number of iterations presented is the total number of iterations executed for detecting two spheres that satisfies the radius criteria and inlier termination criteria $t \times I$. The number of inliers that falls on a sphere that is placed inside a point cloud of a cylinder depends on the point density and the error tolerance T, used to determine whether a point is compatible with the sphere. The number of inliers I that falls on the sphere is established by initializing a sphere with the same radius as the cylinder and the center at the cylinder center-axis. The sphere is initialized in a outlier free point cloud of the cylinder and the number of points that falls within the bounds of the sphere surface is counted.

The cylinder is only evaluated when two spheres that satisfies radius and inliers are identified. By examining the "Evaluated Iterations" graph in Figure 5.3, it can be seen the the number of evaluated cylinders is low and almost constant for outlier percentages up to 80%. This indicates that the two spheres gives a good indication of the actual cylinder and that the sphere-sphere approach gives robust results. However, above 80% outliers the number of evaluated cylinders increases exponentially. This means that spheres that do not represent the cylinder are accepted due to the high outlier density.

It can be seen from Figure 5.3 that the sphere-sphere approach performs considerably better than the plane-circle approach. Below outlier percentages up to 80%, the sphere-sphere approach requires fewer iterations than the plane-circle approach. In addition, more accurate detection is performed with the sphere-sphere initialization, as both the error metric and number of scored iterations are lower. By comparing Table 3 and Table 4 in Ap-

pendix A, it is found that the sphere-sphere approach requires approximately 4 times less iterations, uses approximately 32 times less computation time and produces approximately 1.75 times more accurate detections.

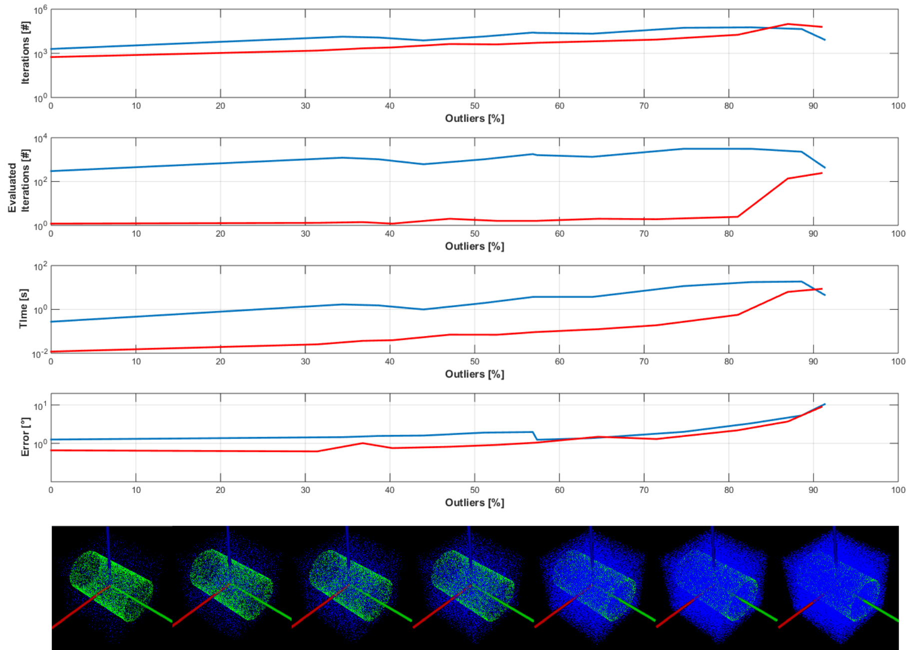


Figure 5.3: Results for the cylinder experiments. The blue line represents the circle-plane initialization while the red line represents the sphere-sphere initialization. The performance of the algorithms is plotted logarithmically for an increasing outlier percentage. As the outlier percentage increases the performance metrics increases exponentially. The outlier percentage is graphically illustrated below the plots.

5.3 Tracking

In order to analyse the performance of the tracking algorithm, real sensor data from a Kinect for Xbox One 3D camera was used. A sphere with radius $r = 33$mm was placed in the sensor field of view and tracking was executed. The parameters of the tracking, as presented in Section 4.5, were set to $d = 5$cm, $t = 0.9$ and $k = 10000$. Three different experiments were conducted. First, tracking was performed on a single point cloud captured by the sensor. The goal is to map the uncertainties that is caused by noise in the data. Secondly, tracking was performed in a real-time data stream. The sphere and the sensor were held at fixed positions in order to map the uncertainties that is caused by variations in the data stream. Lastly, it was attempted to track the sphere during a linear movement. The results of the experiments are presented in Table 5.1 and described in the subsections below.

Table 5.1: Resulting averages for the tracking experiments. Performance for single point cloud tracking, real-time static tracking and real-time movement tracking is tabled.

	Single Point Cloud	Real-Time Static	Real-Time Movement
Accuracy [mm]	2.84	3.33	3.33
Uncertainty [mm]	1.51	1.86	1.72
Points	559	559	1590
Outlier Ratio	0.72	0.72	0.79
Time [ms]	0.23	0.47	2.8
Iterations	757	950	6532
Probability of Detection	0.99	0.99	1

5.3.1 Single Point Cloud

The center position of the sphere was detected 1000 times in a tracking sequence. The detected positions are plotted in Figure 5.4, the origin in the plot is placed at the centroid of the detected positions. The average distance from the centroid of the detected positions is 2.84mm with a standard uncertainty of 1.51mm. The variance in position implies that inliers are defined differently for every detection, which again implies that the points representing the sphere does not lie on a perfect sphere surface. The accuracy for a single point obtained by the sensor is 1% of the range (z-distance) for an average of 100 frames [32]. The average z-position of the sphere is measured at 1.75m, which means that a single point on the sphere have a maximum accuracy of 17.5mm. This explain the variation of the detected center position. However, the accuracy of the sphere center is better than for a single point, implying that the uncertainty in position of a point is normal distributed.

An average of 559 points satisfied the proximity condition explained in Section 4.5. The average outlier ratio was 0.72, which resulted in an average of 757 iterations with a 0.23ms average time of detection. This is well below the limit of 16.67ms for a 60 FPS data stream. According to equation (4.31) presented in Section 4.4.3, the sphere was detected with an average probability of ≈ 0.99.

5.3.2 Real-Time Static Tracking

The sensor was set to stream point clouds at a rate of 30 FPS. The tracking algorithm was run every 40ms, insuring that a new point cloud was received every time the tracking was run. The center position of the sphere was detected 1000 times in a tracking sequence. The sphere and the sensor was held at the same positions as for the single point cloud experiment. The detected sphere center positions are plotted in Figure 5.5, the origin in the plot is placed at the centroid of the detected positions. The average distance from the centroid of the detected positions is $3.33mm$ with a standard uncertainty of 1.86mm. Compared to the accuracy and uncertainty of the single point cloud tracking experiment, the deviance caused by disturbances in the data stream are 0.4mm with a 0.35mm uncertainty. Such disturbances can be caused by small vibrations, changes in lighting conditions or accuracy of the electrical components of the sensor.

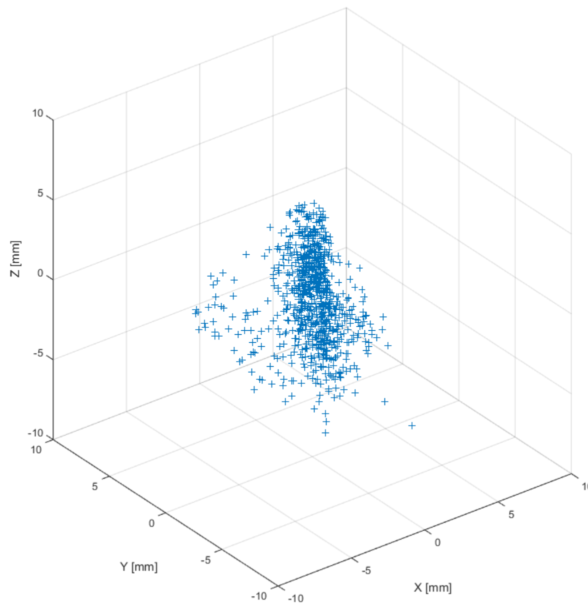


Figure 5.4: Detected center position of a sphere in a single point cloud tracking sequence.

The average time of detection was 0.47ms with an average of 950 iterations. The data set contained an average of 559 points with an average outlier ratio of 0.72. According to equation (4.31) the sphere were detected with an average probability of ≈ 0.99. Compared to the single point cloud experiment, the data set size and outlier ratio are approximately equal. However, the number of iterations and time of detection is higher for a real-time data stream. A possible explanation can be that these metrics already are very small and that the degree of "luck" during the random sampling plays a significant role. Thus, more than 1000 detections are needed to find a statistical significant average.

5.3.3 Real-Time Movement Tracking

To establish whether the tracking algorithm is able to track moving objects, the sphere was tracked while rolling on the floor in an approximate straight path. For tracking to be successful, the sphere must not move more than $d = 5$cm before the sensor acquires a new frame. Thus, when deciding d, both movement speed of the object and frame rate of the sensor should be considered. However, as the goal of this experiment were to establish whether a moving object could be tracked or not, a low movement speed were applied and $d = 5$cm were considered sufficient.

For the experiment, the center position of the sphere was detected 74 times while the sphere rolled a total distance of 1.57m. The measured accumulated time was 2.85s, which means that the average frame rate was 25.97 FPS and that the sphere had an average speed of $0.55\frac{\text{m}}{\text{s}}$. The detected sphere centers have an average distance of 3.33mm to a fitted line, with a standard uncertainty of 1.7mm. The detected sphere centers is plotted in Figure 5.6

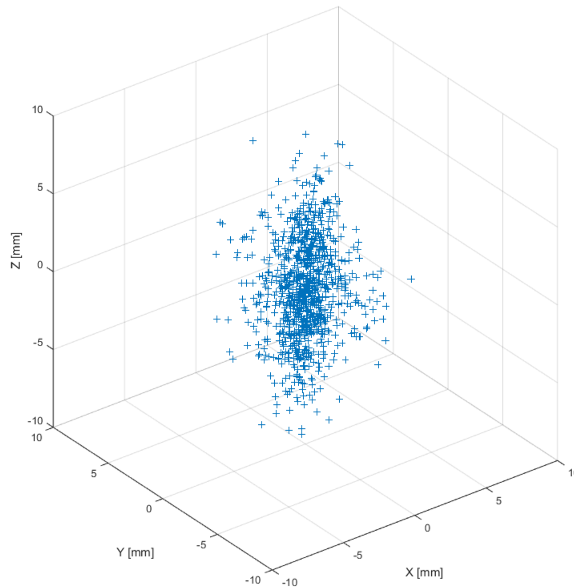


Figure 5.5: Detected center position of a static sphere in a real-time tracking sequence.

along with the fitted line.

The data set contained an average of 1590 points with an average outlier ratio of 0.79. This resulted in an average of 6532 iterations with an average time of detection of 2.8ms. Compared to the two other tracking experiments, the time of detection is higher due to the number of points considered. Nevertheless, the time of detection were below the limit of 16.67ms for a 60 FPS data stream. According to equation (4.31) the sphere was detected with an average probability of ≈ 1.

By examining the result in Table 5.1, it is observed that the measured accuracy and uncertainty of the real-time static- and movement tracking are approximately equal. However, the movement tracking experiment were conducted closer to the sensor. This is reflected in the number of inlier points for the two experiments, as more points fall on the sphere closer to the sensor. Because the accuracy of a single point is dependent on the distance from the sensor, higher accuracy data where used for the movement tracking. Thus, it can be concluded that the accuracy and uncertainty of movement tracking is not as good as for static tracking.

5.4 Multiple Shape Detection

The multiple shape detection algorithm is only implemented for spheres. In order to analyse the performance of the algorithm, real sensor data from a Kinect for Xbox One 3D camera was used. Three spheres were placed in the sensor field of view and a single point cloud of 22225 points was captured. The multiple shape detection was run 1000 times on

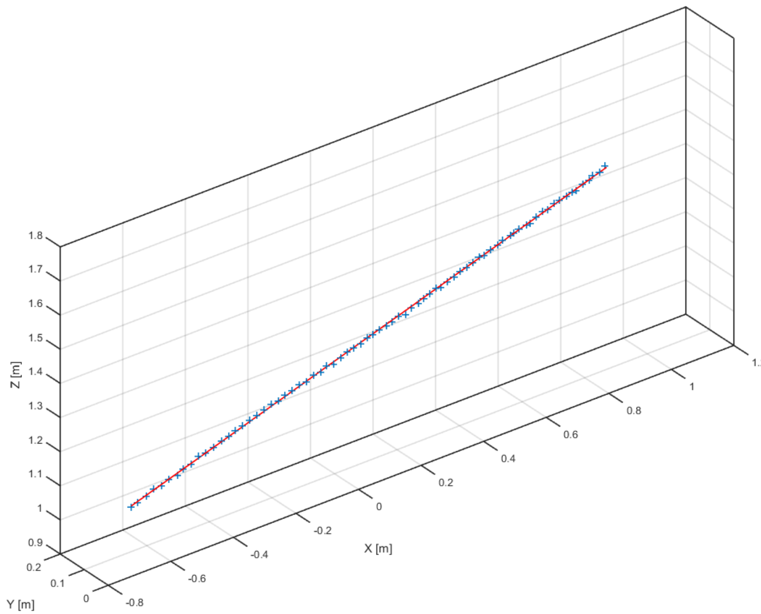


Figure 5.6: Detected center position of a moving sphere in a real-time tracking sequence.

this data set. The data set were organized in an octree with a root node of length $1.28\mathrm{m}$ and a resolution of $0.01\mathrm{m}$. The octree consisted of 3675 populated nodes and the number of iterations per node were set to $k = 4$, resulting in a total of 14700 iterations per run of the algorithm. The average run-time of the algorithm was $121\mathrm{ms}$.

The results of the experiment is presented in Table 5.2. The detection rate describes the ratio of successful detections of a sphere. The distance metric is presented as the spheres absolute distance from the sensor. It can be seen that the detection rate is higher for the largest sphere. In addition, for the two smaller spheres the detection rate is highest for the sphere with the most inlier points, which is proportional to the object's distance from the sensor. The actual radii of the spheres are presented along with the average radii of the detections.

The positions of the successful detections are plotted in Figure 5.7, which gives a visual impression of the accuracy of the algorithm. The uncertainty in position and radius for the largest sphere is higher than expected compared to the two smaller spheres. This occurs because small spheres that are initialized from the surface of the large sphere are accepted. The phenomenon is also reflected in the distribution of the detected sphere radii, which is presented in Figure 5.8. This could have been avoided with a more robust scoring scheme.

For the experiment, the algorithm detected exactly three spheres correctly 56.2% of the attempts. For 42.8% of the attempts, one or more spheres were not detected. Lastly, for 8.8% of the attempts false detections were made. These are not robust results. However, if all three spheres were to be detected with a 95% probability using the single shape detection algorithm, it would require a total of $\approx 28\,025\,543\,170$ iterations. This is $1\,906\,567$

Table 5.2: Resulting averages for the multiple shape detection experiment. The table shows performance metrics of the algorithm for the detection of three different spheres in the point cloud.

	Sphere 1	Sphere 2	Sphere 3
Detection rate	0.989	0.872	0.707
Distance [m]	1.626	1.272	1.697
Accuracy [mm]	6.81	4.79	5.74
Uncertainty [mm]	6.26	4.16	4.89
Inliers	202	141	73
Actual radius [mm]	33	20	20
Detected radius [mm]	28.94	18.97	19.85
Uncertainty [mm]	5.1	2.41	2.34

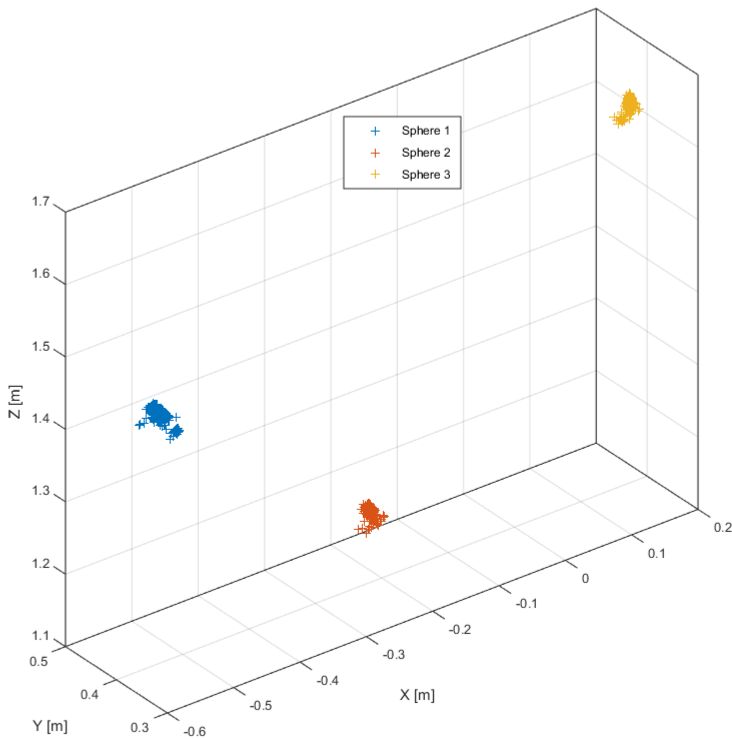


Figure 5.7: Successful detections of the multiple shape detection algorithm. The color of the plotted positions indicate which sphere is detected.

times more iterations compared to the number of iterations conducted in this experiment.

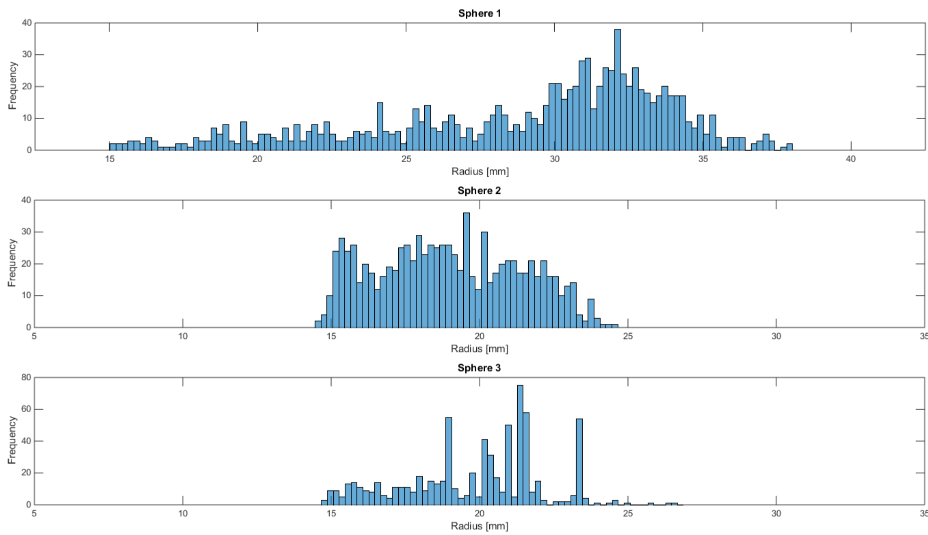


Figure 5.8: Distribution of detected radii for the multiple shape detection algorithm. It can be seen that the distribution for the largest sphere have a long tail of smaller radii. This is a result of the algorithm accepting smaller spheres that are initialized on the surface of the largest sphere.

5.5 Scoring Function

In the experiments for tracking and multiple shape detection, the scoring function described in Section 4.4.2 was used to evaluate the spheres. The denominator of the scoring function describes the expected number of inliers for a sphere depending on its radius and z-distance from the sensor. The parameters of the function is dependent on the sensor and were found experimentally for a Kinect for Xbox One 3D camera. Taking into account that only spheres were to be considered, a small modification was done to the denominator:

$$Aaz^b = 4\pi r^2 az^b = \hat{a}z^b r^2. \tag{5.1}$$

Thus, the parameter $\hat{a}$ were found instead of a.

Three spheres of different radii were placed in the field of view of the sensor. The z-distance, radius and inlier points for each sphere was measured a total of 10628 times from multiple positions. Finally, the function was fitted to the measurements by adjusting the $\hat{a}$ and b parameters using the "problem solver"-tool in Excel. The result is plotted in Figure 5.9, where the blue line represent the measured inliers and the orange line represent the estimated inliers with $\hat{a} = 445471.914$ and $b = -2.202$. The different levels of inliers in the plot reflect the radii of the spheres.

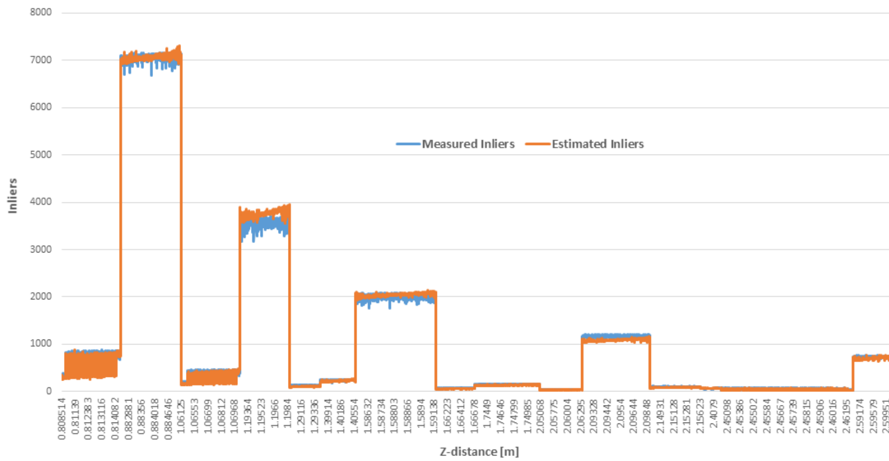


Figure 5.9: Fitted scoring function. The parameters of the scoring function is adjusted in Excel by minimizing the residual between the scoring function and actual measurements. The different levels of inliers reflects the radius of the spheres used in the measurements.

5.6 Summarizing Remarks

This chapter presented performance results for the implemented algorithms. The random sampling scheme of the original RANSAC algorithm is the base of all the implemented algorithms. The performance of this sampling scheme is predicted by the probabilistic relations presented in Section 4.4.3 and the results presented in [3]. This agrees with the relation between outlier ratio and iterations tabled in Table 1 and 2 found in Appendix A. In the tables it can be seen that the calculated probabilities for successful detections lie in the range between $34,1\%$ and 68.2%, which is the range of 1 standard deviation in a standard normal distribution.

In the primitive shape detection experiments, it is discovered that the outlier density is decisive for successful detections. When the outlier density is too high, the algorithm cannot separate the shape from the rest of the point cloud. If the outliers where distributed over a larger volume, it is expected that successful detections could have been achieved. Consequently, the outlier percentage can be close to 100% and successful detections can be made, as along as the outliers are spread over a large volume. However, the number of iterations increases exponentially with outlier percentage, which limits the applications for detections in point clouds with high percentage of outliers.

The single point cloud tracking experiment is simply a primitive shape detection procedure executed multiple times on a point cloud obtained from a sensor. This experiment shows that uncertainties in the points that falls on the sphere causes inaccuracies in the detected center position of the sphere. However, the accuracy for a shape consisting of multiple points is higher than for a single point obtained by the sensor. In addition, it is found that the algorithm performs sufficiently fast for real-time tracking. The real-time static tracking experiment shows that real-time tracking is achieved successfully. However, the

accuracy is slightly reduced due to the variations in the data stream. The real-time movement tracking experiment showed that the tracking algorithm were able to track moving objects, even though the observed accuracy were degraded compared to static tracking. Nevertheless, the experiments showed promising results, considering the accuracy of the sensor. The low computational cost and computation time of the tracking algorithm is a benefit of the data limitation strategy.

The performance of the multiple shape detection algorithm can be compared with the single point cloud tracking performance. The largest sphere is placed at approximately the same distance from the sensor in the two experiments. It can be seen that the accuracy of the primitive shape detection algorithm is about 4 times better than the multiple shape detection algorithm. In addition, the detection rate of the multiple shape detection algorithm is well below 1. However, the multiple shape detection uses fewer iterations than the primitive shape detection algorithm. Thus, computational savings and the convenience of detecting multiple shapes with a single algorithm compensate for the loss in robustness and accuracy.

The octree structure and sampling strategy of the multiple shape detection algorithm ensures that minimal sub-samples are picked from the surfaces of the spheres. Even though spheres are initialized from these sub-samples the spheres are sometimes rejected, which can be seen from the detection rate. This is due to the noise in the point cloud and low accuracy of the sensor. This can be compensated for by trying more minimal subsets in every node, at the cost of computational efficiency. Alternatively, spheres can be fitted to its inliers and re-scored before rejection or acceptance. This could reduce the influence of noisy variations. In addition, the algorithm sometimes produces false detections. This can be avoided by further improving the scoring function.

Chapter 6

Concluding Remarks

6.1 Discussion

Starting with an idea of how conformal geometric algebra could be suited for point cloud processing, and ending up with the results and software presented in this thesis, have been a highly educational journey. Many hours of self-education have been spent with subjects like geometric algebra, point cloud processing, C++ object oriented programming, Linux and ROS, robust statistics and kinematics.

The programming aspect of this thesis have been especially demanding, due to limited programming skills. An experienced programmer would probably criticize a lot of the methods and structure in the software. However, as a GUI with a lot of functionality is successfully implemented, the author is very satisfied with the result.

The initial objectives of this thesis are fulfilled and the author believes the project is successfully completed. In addition to the initial objectives, tracking and multiple shape detection are considered. The computational efficiency of the studied algorithms was the inspiration for the tracking implementation, while methods studied in the literature were the inspiration for multiple shape detection.

6.2 Conclusion

Industrial processes often involves handling of objects and surfaces shaped like geometric primitives. This should be taken into consideration when designing computer vision-based systems for such processes. Both pose and parameters of geometric primitives can be established with 3D cameras and simple algorithms. In this thesis, robust estimation algorithms have been considered to detect primitive shapes in point clouds from 3D cameras. The primitives are described with conformal geometric algebra. Possible applications have been presented.

The technology behind 3D cameras have been described and it is explained how such sensors can output high density point clouds. Existing methods and concepts of point

cloud processing have been presented.

A literature review of robust statistical estimators for computer vision have been conducted. A detailed review of the RANSAC algorithm is given, along with a overview of RANSAC-based algorithms. The objective of these methods and algorithms is to classify a data set into inliers and outliers. Outliers are discarded and inliers can be fitted to some predefined model. When the data set is a point cloud and the model is some geometric primitive, these methods can be applied to detect pose and parameters of primitive shapes.

The primitive shape initialization and inlier classification is implemented in conformal space with geometric algebra. Two approaches for modeling a cylinder is suggested. Conformal geometric algebra is not applied anywhere in the literature of robust estimators studied. Thus, this thesis works as a proof of concept.

A RANSAC-based algorithm is implemented in a C++ based software and the system is tested in experiments and results are presented. Furthermore, additional methods for improving the computational cost and number of iterations of RANSAC are implemented in a tracking algorithm and a multiple shape detection algorithm. The inspiration for these improvements were found in the literature studied.

The results of the conducted experiments shows that primitive shapes can be detected in point clouds with up to 90% outliers. It is also argued that detections can be made in point clouds with close to 100% outliers, given that the outlier density is lower than the inlier density. In point clouds obtained from a Kinect for Xbox one 3D camera, results shows that the accuracy of detected primitives are better than the accuracy for the sensor. This is due to the normalizing effect when considering all the points that describe a shape. However, accuracy is highly dependent on the distance from the sensor and the number of points that falls on the detect shape. Furthermore, the tracking algorithm can achieve real-time tracking due to the low computational cost of the algorithm.

The sphere-sphere approach for detecting a cylinder proved to be superior to the circle-plane approach. In general, the sphere-sphere approach requires fewer iterations and produces more accurate results.

The sampling scheme of the multiple shape detection algorithm were found to be $1\,906\,567$ times more effective than a random one, for the conducted experiment. However, this metric is highly dependent on point density and octree resolution. The multiple shape detection algorithm was set to detect 3 spheres in a noisy point cloud and was executed 1000 times. The algorithm detected all three objects correctly 56.2% of the attempts, which is not a robust result. In addition, it is found that the accuracy for the primitive shape detection algorithm is approximately 4 times better than for the multiple shape detection. Nevertheless, suggestions for improvements have been proposed and the author believes that a powerful multiple shape detection algorithm can be achieved.

In the robotic pick and place demonstrator, some possible applications of primitive shape detection are suggested. The video clearly demonstrates the accuracy of the algorithms, as the robot successfully pick-and-place repeatedly.

6.3 Further Work

Only planes, spheres and cylinders are considered in this thesis. A natural candidate for further work is to define several primitives with conformal geometric algebra and develop

methods for inlier classification. The suggested sphere-sphere approach for detecting a cylinder in a point cloud can be extended to cones and torus, as spheres would fit in the same manner for these shapes.

Another candidate for further work is to implement several primitives for the multiple shape detection algorithm. In addition, further improving the scoring and fitting of detected primitives can achieve robust detection. The ultimate goal would be to successfully detect and extract all primitives present in a point cloud. This can be used for efficient surface reconstruction, meshing or object detection.

Lastly, some comparative work regarding conformal geometric algebra could be performed. In [6] they present performance result of their suggested algorithm. The same algorithm could be implemented with conformal geometric algebra for comparison, as a contribution to their work. In this thesis, it is concluded that conformal geometric algebra is suited for such applications, but no performance advantages are investigated.

The primitive shape detection algorithms suggested in this thesis can also be compared to a traditional point cloud processing pipeline. In cases were the objective is to estimate pose of primitive objects, it is expected that the suggested methods will outperform the complexity of the point cloud processing pipeline presented in Section 3.4. It would also be interesting to examine if a fully functional multiple shape detection algorithm could work with objects of complex geometry. Such objects can be subdivided into regions of primitive shapes like illustrated in Figure 6.1.

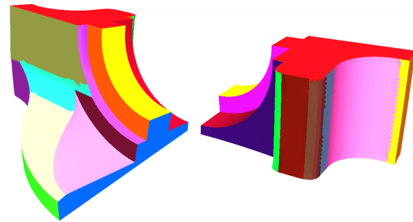


Figure 6.1: An object subdivided to into regions of primitive shapes. (Adapted from [6])

Bibliography

[1] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[2] Jaewoong Kim, Huu Hung Nguyen, Yeonho Lee, and Sukhan Lee. Structured light camera base 3d visual perception and tracking application system with robot grasping task. In *Assembly and Manufacturing (ISAM), 2013 IEEE International Symposium on*, pages 187–192. IEEE, 2013.

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

[4] Dietmar Hildenbrand. *Foundations of geometric algebra computing*. Springer, 2013.

[5] Leo Dorst, Daniel Fontijne, and Stephen Mann. *Geometric algebra for computer science: an object-oriented approach to geometry*. Morgan Kaufmann Publishers Inc., 2009.

[6] Ruwen Schnabel, Roland Wahl, and Reinhard Klein. Efficient ransac for point-cloud shape detection. In *Computer graphics forum*, volume 26, pages 214–226. Wiley Online Library, 2007.

[7] R MUTHUKRISHNAN et al. Contributions to a study on robust statistics and its applications in computer vision. 2015.

[8] Paul VC Hough. Method and means for recognizing complex patterns. Technical report, 1962.

[9] Xinming Yu, TD Bui, and A Krzyżak. Robust estimation for range image segmentation and reconstruction. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 16(5):530–538, 1994.

[10] Charles V Stewart. Minpran: A new robust estimator for computer vision. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 17(10):925–938, 1995.

[11] James V Miller and Charles V Stewart. Muse: Robust surface fitting using unbiased scale estimates. In *Computer Vision and Pattern Recognition, 1996. Proceedings CVPR'96, 1996 IEEE Computer Society Conference on*, pages 300–306. IEEE, 1996.

[12] Kil-Moo Lee, Peter Meer, and Rae-Hong Park. Robust adaptive segmentation of range images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(2):200–205, 1998.

[13] Hanzi Wang and David Suter. Mdpe: A very robust estimator for model fitting and range image segmentation. *International Journal of Computer Vision*, 59(2):139–166, 2004.

[14] Hanzi Wang and David Suter. Robust adaptive-scale parametric model estimation for computer vision. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(11):1459–1474, 2004.

[15] Hanzi Wang and David Suter. Robust fitting by adaptive-scale residual consensus. In *Computer Vision-ECCV 2004*, pages 107–118. Springer, 2004.

[16] Hanzi Wang. Maximum kernel density estimator for robust fitting. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, pages 3385–3388. IEEE, 2008.

[17] Hanzi Wang, Daniel Mirota, and Gregory D Hager. A generalized kernel consensus-based robust estimator. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(1):178–184, 2010.

[18] Peter J Rousseeuw and Annick M Leroy. *Robust regression and outlier detection*, volume 589. John Wiley & Sons, 2005.

[19] Philip HS Torr and Andrew Zisserman. Mlesac: A new robust estimator with application to estimating image geometry. *Computer Vision and Image Understanding*, 78(1):138–156, 2000.

[20] DRMPHSTS Nasuto and JM Bishop R Craddock. Napsac: High noise, high dimensional robust estimation-its in the bag. 2002.

[21] Ondřej Chum, Jiří Matas, and Josef Kittler. Locally optimized ransac. In *Pattern recognition*, pages 236–243. Springer, 2003.

[22] David Nistér. Preemptive ransac for live structure and motion estimation. *Machine Vision and Applications*, 16(5):321–329, 2005.

[23] Ondřej Chum and Jiří Matas. Randomized ransac with td, d test. In *Proc. British Machine Vision Conference*, volume 2, pages 448–457, 2002.

[24] Ondřej Chum and Jiří Matas. Matching with prosac-progressive sample consensus. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 220–226. IEEE, 2005.

[25] Volker Rodehorst and Olaf Hellwich. Genetic algorithm sample consensus (gasac)-a parallel strategy for robust parameter estimation. In *Computer Vision and Pattern Recognition Workshop, 2006. CVPRW'06. Conference on*, pages 103–103. IEEE, 2006.

[26] Jan-Michael Frahm and Marc Pollefeys. Ransac for (quasi-) degenerate data (qdegsac). In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 1, pages 453–460. IEEE, 2006.

[27] Kai Ni, Hailin Jin, and Frank Dellaert. Groupsac: Efficient consensus in the presence of groupings. In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 2193–2200. IEEE, 2009.

[28] Torsten Sattler, Bastian Leibe, and Leif Kobbelt. Scramsac: Improving ransac's efficiency with a spatial consistency filter. In *Computer vision, 2009 ieee 12th international conference on*, pages 2090–2097. IEEE, 2009.

[29] Antoine Meler, Marion Decrouez, and James Crowley. Betasac: A new conditional sampling for ransac. In *British Machine Vision Conference 2010*, 2010.

[30] Donald JR Meagher. *Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer*. Electrical and Systems Engineering Department Rensseiaer Polytechnic Institute Image Processing Laboratory, 1980.

[31] Carlo Dal Mutto, Pietro Zanuttigh, and Guido M Cortelazzo. *Time-of-flight cameras and microsoft Kinect*. Springer Science & Business Media, 2012.

[32] Annette Payne, Andy Daniel, Anik Mehta, Bradley Thompson, Cyrus S Bamji, Dane Snow, Hirotaka Oshima, Larry Prather, Mike Fenton, Lou Kordus, et al. 7.6 a 512× 424 cmos 3d time-of-flight image sensor with multi-frequency photo-demodulation up to 130mhz and 2gs/s adc. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 134–135. IEEE, 2014.

[33] Peter Lancaster and Kes Salkauskas. Surfaces generated by moving least squares methods. *Mathematics of computation*, 37(155):141–158, 1981.

[34] Federico Tombari, Samuele Salti, and Luigi Di Stefano. Performance evaluation of 3d keypoint detectors. *International Journal of Computer Vision*, 102(1-3):198–220, 2013.

[35] Johan WH Tangelder and Remco C Veltkamp. A survey of content based 3d shape retrieval methods. *Multimedia tools and applications*, 39(3):441–471, 2008.

[36] Yang Chen and Gérard Medioni. Object modelling by registration of multiple range images. *Image and vision computing*, 10(3):145–155, 1992.

[37] Dietmar Hildenbrand. *Foundations of geometric algebra computing*, chapter 5.3 Fitting of Planes or Spheres to Sets of Points, Least-Squares Approach. In [4], 2013.

[38] Pablo Colapinto. Versor: Spatial computing with conformal geometric algebra. Master's thesis, University of California at Santa Barbara, 2011. Available at `http://versor.mat.ucsb.edu`.

Appendices

A Results

This appendix contains tabled measurements for the experiments presented in Section 5.2. Table 1 and 2 are graphically presented in Figure 5.1. Table 3 and 4 are graphically presented in Figure 5.3.

Table 1: Results for shape detection of a sphere with unknown radius. The probability of detection is calculated using the probabilistic relations discussed in Section 4.4.3.

Outliers	Iterations	Evaluated Iterations	Time [ms]	Error [mm]	Probability of Detection
33%	2	2	2.7	0.25	42.4%
37%	6	6	5.0	0.25	62.9%
40%	11	11	9.1	0.25	78.6%
44%	6	6	5.7	0.25	46.1%
46%	19	19	16.4	0.25	80.2%
49%	9	9	9.6	0.24	47.9%
55%	18	18	19.3	0.25	54.2%
59%	43	43	48.9	0.24	71.0%
65%	77	77	107.1	0.26	67.2%
74%	137	137	261.6	0.25	48.7%
82%	864	864	2588.4	0.26	62.2%
87%	2749	2749	12009.5	0.33	59.5%
90%	6762	6762	38634.9	0.81	51.8%

Table 2: Results for shape detection of a sphere with known radius. The probability of detection is calculated using the probabilistic relations discussed in Section 4.4.3.

Outliers	Iterations	Evaluated Iterations	Time [ms]	Error [mm]	Probability of Detection
33%	8	2	2.5	0.25	84.1%
42%	8	1	2.2	0.25	62.7%
49%	20	2	3.2	0.25	75.2%
59%	22	2	3.7	0.23	47.7%
74%	193	5	9.6	0.26	61.1%
82%	1135	19	48.1	0.27	71.2%
87%	1694	26	92.5	0.30	42.6%
89%	8089	120	526.3	0.30	67.4%
91%	14849	189	1058.6	0.52	58.7%
92%	13851	172	1090.5	0.89	37.0%
94%	35838	430	3632.0	1.29	42.0%
95%	71751	856	8853.3	2.46	37.9%
96%	44012	506	6231.4	5.78	10.0%
97%	17114	202	3301.3	18.35	1.1%
98%	243	4	100.9	118.32	0.0%

Table 3: Results for shape detection of a cylinder. The cylinder is initialized using the circle-plane approach described in Section 4.3.4.

Outliers	Iterations	Evaluated Iterations	Time [ms]	Error [°]
0%	1988	299	273.8	1.257
34%	13505	1222	1683.9	1.452
39%	11853	1038	1516.8	1.557
44%	7521	615	990.8	1.595
51%	14003	1038	1961.6	1.906
57%	25826	1784	3689.4	1.972
57%	24548	1609	3715.8	1.240
64%	21598	1340	3706.7	1.369
75%	54311	3122	11592.0	1.996
83%	58215	3111	17595.1	3.329
89%	44719	2291	18693.8	5.316
91%	8404	436	4539.8	10.498

Table 4: Results for shape detection of a cylinder. The cylinder is initialized using the sphere-sphere approach described in Section 4.3.4.

Outliers	Iterations	Evaluated Iterations	Time [ms]	Error [°]
0%	557	1	11.9	0.657
31%	1536	1	25.4	0.618
37%	2171	1	36.7	1.012
40%	2530	1	39.2	0.752
47%	4282	2	70.1	0.816
53%	4035	2	69.5	0.913
57%	5175	2	92.3	1.042
65%	6682	2	125.0	1.483
71%	8567	2	188.5	1.298
81%	18192	2	562.5	2.187
87%	98748	137	6286.9	3.703
91%	63190	243	8652.8	8.949

B C++ Source Code

This appendix contain the source code for primitive shape modeling with conformal geometric algebra and the implemented RANSAC-based algorithms.

- **objects.h** is the header file for the primitive shapes in conformal space. It contains declarations for planes, spheres, circles and cylinders.
- **objects_body.cpp** is the implementation of the **objects.h** header file. It contains the implementations for all the functions and variables declared in **objects.h**.
- **ransac.h** is the header file for the implemented algorithms. It contains declarations of the primitive shape detection algorithm for planes, spheres and cylinders. In addition, it contains a declaration of the multiple shape detection algorithm for spheres. The classes and methods declared in **objects.h** are included.
- **ransac_body.cpp** is the implementation of the **ransac.h** header file. It contains the implementations of all the functions and variables declared in **ransac.h**.

objects.h

```
//
// Created by Aksel Sveier. Spring 2016
//

#ifndef RANSACGA_OBJECTS_H
#define RANSACGA_OBJECTS_H

#endif //RANSACGA_OBJECTS_H

#include <vsr/space/vsr_cga3D_op.h>
#include <pcl/point_types.h>
#include <pcl/common/common_headers.h>

using namespace vsr;
using namespace vsr::cga;
using namespace pcl;

//Class that creates and handles planes with CGA
class plane
{
public:
    Pnt dualPlane; //Stores the IPNS representation
    Pnt normDualPlane; //Stores the normalized IPNS representation

    void defineDual(Pnt p1, Pnt p2, Pnt p3); //Creates a plane from 3
    points in conformal space
    void defineDual(PointCloud<PointXYZRGB>::Ptr cl, int *index); //
    Creates a plane from 3 indexed points in a point cloud
private:
    void normalize(Pnt dPln); //Function that normalizes a conformal
    vector
};
```

```
31
32 //Class that creates and handles spheres with CGA
33 class sphere{
34 public:
35     float radius, x, y, z; //Floats for storing the parameters of the
       sphere
36     Pnt dualSphere; //Stores the IPNS representation
37     Sph sphe; //Stores the OPNS representation
38
39     void defineDual(float x, float y, float z, float r); //Creates a
       sphere from center(x,y,z) coordinates and radius
40     void defineDual(Pnt p1, Pnt p2, Pnt p3, Pnt p4); //Creates a sphere
       from 4 points in conformal space.
41     void defineDual(PointCloud<PointXYZRGB>::Ptr cl1, PointCloud<
       PointXYZRGB>::Ptr cl2, PointCloud<PointXYZRGB>::Ptr cl3, PointCloud<
       PointXYZRGB>::Ptr cl4); //Creates a sphere from 4 points from a
       pointcloud.
42     void defineDual(PointCloud<PointXYZRGB>::Ptr cl, vector<int> index);
       //Creates a sphere in from 4 indexed points in a point cloud
43 private:
44     float calcRadius(Pnt dSph); //Function that calculates radius
45     Pnt normalize(Pnt dSph); //Function that normalizes a conformal vector
46 };
47
48 //Class that creates and handles circles with CGA
49 class circle
50 {
51 public:
52     float radius = 0; //Float for storing the circle radius
53     Pnt circleCenter; //Point in conformal space for storing the circle
       center
54     Par circle; //Store the OPNS representation
55     Cir dualCircle; //Stores the IPNS representation
56     Pnt plane; //Stores the plane the the circle lie on
57
58     void defineCircle(Pnt p1, Pnt p2, Pnt p3); //Creates circle from 3
       points in conformal space
59     void defineCircle(PointXYZRGB p1, PointXYZRGB p2, PointXYZRGB p3); //
       Creates a circle from 3 point from a point cloud
60     void defineCircle(PointCloud<PointXYZRGB>::Ptr cl, int *index); //
       Creates circle from 3 indexed points in a point cloud
61
62 private:
63 void calcRadius(Pnt p); //Function that finds the cricle radius
64 void findNormal(Pnt pln); //Creates the plane that the cricle lie on
65 };
66
67 //Class that creates and handles cylinders with CGA
68 class cylinder{
69 //Since there is no representation of a cylinder in CGA a cylinder will be
        represente by a circle on a plane.
70 //The direction of the normal of the plane in the circle center will be
       the center-axis of the cylinder
71 public:
72     Pnt plan; //Store the IPNS representation of a plane
73     Pnt center; //Stores the center in a conformal vector
74     float radius = 0; //Float for storing the radius
```

```
    void defineCylinder(Pnt p1, Pnt p2, float rad); //Creates a cylinder from two points in conformal space on the center axis and a radius
    void defineCylinder(Pnt p1, Pnt p2, Pnt p3); //Creates a cylinder from 3 points in conformal space
    void defineCylinder(PointCloud<PointXYZRGB>::Ptr cl, int *index); // Creates the cylinder from 3 indexed points in a point cloud
};
```

objects_body.cpp

```
//
// Created by Aksel Sveier. Spring 2016
//

#include "objects.h"
#include <vsr/space/vsr_cga3D_op.h>
#include <pcl/point_types.h>
#include <pcl/common/common_headers.h>

using namespace vsr;
using namespace vsr::cga;
using namespace pcl;

//Creates a sphere from center(x,y,z) coordinates and radius
void sphere::defineDual(float x, float y, float z, float r){
    dualSphere = normalize(Round::dls( Vec( x,y,z ), r ));
    radius = r;
}

//Creates a sphere from 4 points in conformal space.
void sphere::defineDual(Pnt p1, Pnt p2, Pnt p3, Pnt p4){
    dualSphere = normalize((p1^p2^p3^p4).dual());
    radius = calcRadius(dualSphere);
}

//Creates a sphere from 4 points from a pointcloud.
void sphere::defineDual(PointCloud<PointXYZRGB>::Ptr cl1, PointCloud<PointXYZRGB>::Ptr cl2, PointCloud<PointXYZRGB>::Ptr cl3, PointCloud<PointXYZRGB>::Ptr cl4){
    dualSphere = normalize((Vec(cl1->points[0].x,cl1->points[0].y,cl1->points[0].z).null()
     ^ Vec(cl2->points[0].x,cl2->points[0].y,cl2->points[0].z).null()
     ^ Vec(cl3->points[0].x,cl3->points[0].y,cl3->points[0].z).null()
     ^ Vec(cl4->points[0].x,cl4->points[0].y,cl4->points[0].z).null()).dual());
    radius = calcRadius(dualSphere);
}

//Creates a sphere in from 4 indexed points in a point cloud
void sphere::defineDual(PointCloud<PointXYZRGB>::Ptr cl, vector<int> index){
    dualSphere = normalize((Vec(cl->points[index[0]].x,cl->points[index[0]].y,cl->points[index[0]].z).null()
                 ^ Vec(cl->points[index[1]].x,cl->points[index[1]].y,cl->points[index[1]].z).null()
```

```
39                      ^ Vec(cl->points[index[2]].x,cl->points[index[2]].y,cl->
    points[index[2]].z).null()
40                      ^ Vec(cl->points[index[3]].x,cl->points[index[3]].y,cl->
    points[index[3]].z).null()).dual());
41      radius = calcRadius(dualSphere);
42      x = dualSphere[0]; y = dualSphere[1]; z = dualSphere[2];
43  }
44
45  //Function that calculates radius
46  float sphere::calcRadius(Pnt sph){
47      return sqrt((1/pow(sph[3],2))*(pow(sph[0],2)+pow(sph[1],2)+pow(sph
    [2],2)) - (2*sph[4]/sph[3]));
48  }
49
50  //Function that normalizes a conformal vector
51  Pnt sphere::normalize(Pnt dSph){
52      Pnt ret;
53      for(int i = 0; i<5;i++){
54          ret[i] = dSph[i]/dSph[3];
55      }
56      return ret;
57  }
58
59  //Creates a plane from 3 points in conformal space
60  void plane::defineDual(Pnt p1, Pnt p2, Pnt p3){
61      dualPlane = (p1^p2^p3^Inf(1)).dual();
62      normalize(dualPlane);
63  }
64
65  //Creates a plane from 3 indexed points in a point cloud
66  void plane::defineDual(PointCloud<PointXYZRGB>::Ptr cl, int *index){
67      dualPlane = (Vec(cl->points[index[0]].x,cl->points[index[0]].y,cl->
    points[index[0]].z).null()
68                              ^ Vec(cl->points[index[1]].x,cl->points[index
    [1]].y,cl->points[index[1]].z).null()
69                              ^ Vec(cl->points[index[2]].x,cl->points[index
    [2]].y,cl->points[index[2]].z).null()
70                            ^ Inf(1)).dual();
71      normalize(dualPlane);
72  }
73
74  //Function that normalizes a conformal vector
75  void plane::normalize(Pnt dPln){
76      for(int i = 0; i < 5; i++){
77          normDualPlane[i] = dPln[i]/sqrt(pow(dPln[0],2)+pow(dPln[1],2)+pow(
    dPln[2],2));
78      }
79  }
80
81  //Creates circle from 3 points in conformal space
82  void circle::defineCircle(Pnt p1, Pnt p2, Pnt p3){
83      dualCircle = p1 ^ p2 ^ p3;
84      circle = (p1 ^ p2 ^ p3).dual();
85      Pnt temp = ((p1 ^ p2 ^ p3).dual()) * Inf(1) * ((p1 ^ p2 ^ p3).dual());
86      for(int i= 0; i<5; i++){
87          circleCenter[i] = temp[i]/temp[3];
88      }
```

```
89      findNormal((p1 ^ p2 ^ p3 ^ Inf(1)).dual());
90      calcRadius(p1);
91  }
92
93  //Creates a circle from 3 point from a point cloud
94  void circle::defineCircle(PointCloud<PointXYZRGB>::Ptr cl, int *index){
95      dualCircle = Vec(cl->points[index[0]].x, cl->points[index[0]].y, cl->
        points[index[0]].z).null()
96              ^ Vec(cl->points[index[1]].x, cl->points[index[1]].y, cl->
        points[index[1]].z).null()
97              ^ Vec(cl->points[index[2]].x, cl->points[index[2]].y, cl->
        points[index[2]].z).null();
98      circle = dualCircle.dual();
99      Pnt temp = circle * Inf(1) * circle;
100     for(int i= 0; i<5; i++){
101         circleCenter[i] = temp[i]/temp[3];
102     }
103     findNormal((Vec(cl->points[index[0]].x, cl->points[index[0]].y, cl->
        points[index[0]].z).null() ^ Vec(cl->points[index[1]].x, cl->points[
        index[1]].y, cl->points[index[1]].z).null() ^ Vec(cl->points[index
        [2]].x, cl->points[index[2]].y, cl->points[index[2]].z).null() ^ Inf
        (1)).dual());
104     calcRadius(Vec(cl->points[index[0]].x, cl->points[index[0]].y, cl->
        points[index[0]].z).null());
105 }
106
107 //Creates circle from 3 indexed points in a point cloud
108 void circle::defineCircle(PointXYZRGB p1, PointXYZRGB p2, PointXYZRGB p3){
109     dualCircle = Vec(p1.x,p1.y,p1.z).null()
110             ^ Vec(p2.x,p2.y,p2.z).null()
111             ^ Vec(p3.x,p3.y,p3.z).null();
112     circle = dualCircle.dual();
113     Pnt temp = circle * Inf(1) * circle;
114     for(int i= 0; i<5; i++){
115         circleCenter[i] = temp[i]/temp[3];
116     }
117     findNormal((Vec(p1.x,p1.y,p1.z).null() ^ Vec(p2.x,p2.y,p2.z).null() ^
        Vec(p3.x,p3.y,p3.z).null() ^ Inf(1)).dual());
118     calcRadius(Vec(p1.x,p1.y,p1.z).null());
119 }
120
121 //Function that finds the cricle radius
122 void circle::calcRadius(Pnt p){
123     radius = sqrt(pow(p[0] - circleCenter[0],2) + pow(p[1] - circleCenter
        [1],2) +pow(p[2] - circleCenter[2],2) );
124 }
125
126 //Function that creates the plane that the cricle lies on
127 void circle::findNormal(Pnt pln){
128     for(int i = 0; i < 5; i++){
129         plane[i] = pln[i]/sqrt(pow(pln[0],2)+pow(pln[1],2)+pow(pln[2],2));
130     }
131 }
132
133 //Creates a cylinder from two points in conformal space on the center axis
        and a radius
134 void cylinder::defineCylinder(Pnt p1, Pnt p2, float rad){
```

```
135     radius = rad;
136     center = p1;
137     for(int i = 0; i < 3; i++){
138         plan[i] = (p1[i] - p2[i])/sqrt(pow(p1[0] - p2[0],2) + pow(p1[1] -
        p2[1],2) + pow(p1[2] - p2[2],2));
139     }
140     plan[3] = 0;
141 }
142 
143 //Creates a cylinder from 3 points in conformal space
144 void cylinder::defineCylinder(Pnt p1, Pnt p2, Pnt p3){
145     plane pln;
146     pln.defineDual(p1,p2,p3);
147     plan = pln.normDualPlane;
148     circle cir;
149     cir.defineCircle(p1, p2, p3);
150     center = cir.circleCenter;
151     radius = cir.radius;
152 }
153 
154 //Creates the cylinder from 3 indexed points in a point cloud
155 void cylinder::defineCylinder(PointCloud<PointXYZRGB>::Ptr cl, int *index)
        {
156     plane pln;
157     pln.defineDual(cl, index);
158     plan = pln.normDualPlane;
159     circle cir;
160     cir.defineCircle(cl, index);
161     center = cir.circleCenter;
162     radius = cir.radius;
163 }
```

ransac.h

```
1  //
2  // Created by Aksel Sveier. Spring 2016
3  //
4  
5  #ifndef RANSACGA_RANSAC_H
6  #define RANSACGA_RANSAC_H
7  
8  #endif //RANSACGA_RANSAC_H
9  
10 #include <vsr/space/vsr_cga3D_op.h>
11 #include <pcl/point_types.h>
12 #include <pcl/common/common_headers.h>
13 #include <pcl/filters/passthrough.h>
14 #include <pcl/octree/octree.h>
15 #include <pcl/octree/octree_impl.h>
16 #include <pcl/filters/extract_indices.h>
17 #include "objects.h"
18 
19 using namespace vsr;
20 using namespace vsr::cga;
21 using namespace pcl;
22 
23 ///////////////////////////////////////////////////////////////////////
```

```
//////////////////////// PLANE ///////////////////////////

//////////////////////////////////////////////////////////////////

//Class for detecting a plane in a point cloud with RANSAC
class ransacPlane{
public:
    ransacPlane(); //Constructor
    float planetol; //Tolerance used for deciding wether a point is
    compatable with a plane
    PointCloud<PointXYZRGB>::Ptr cloud; //Point cloud for holding the data
     set
    PointCloud<PointXYZRGB>::Ptr segment; //Point cloud for holding the
    inlier points
    std::vector<int> *indexlist; //Vector for holding the index of inlier
    points

    Pnt ranPln; //Plane object
    int candidates; //Number of candidates to detect

    void setData(PointCloud<PointXYZRGB>::Ptr cl, float tol, int cand);
    void compute(); //does all computation, and eventually stores the
    plane with the most inliers as a Pnt type;
    void segmentPlane(PointCloud<PointXYZRGB>::Ptr seg); //Function that
    creates and stores the indexlist of the current plane
    Pnt fit(Pnt pln); //Plane fitting in conformal space
private:
    bool isInlier(Pnt point, Pnt plane); //Bool for determining weather a
    point is a inlier
};

//////////////////////////////////////////////////////////////////

//////////////////////// SPHERE ///////////////////////////

//////////////////////////////////////////////////////////////////
//Class for detecting a sphere in a point cloud with ransac
class ransacSphere{
public:
    ransacSphere(); //Constructor
    float radiusTolerance;
    float inlierTreshold = 0.005; //For the kinect the accuracy is 1 % of
    the distance from the sensor for 100 images.
    int iterations = 1; //Maximum allowed iterations
    int actualIt; //Integer for storing the number of iterations performed
    float radius = 0; // float for storing the raduis
    PointCloud<PointXYZRGB>::Ptr cloud; //Point cloud for storing the data
     set

    float time; //Float for storing the computation time of the algorithm

    Pnt ranSph; //Sphere object
    int candidates = 1; //Number of candidates to detect
    std::vector<int> *indexlist; //Vector for storing inliers
    int numInliers; //Integer for storing the number of inliers

```

```
72      void setData(PointCloud<PointXYZRGB>::Ptr cl, float tol, float rad);
73      void setData(PointCloud<PointXYZRGB>::Ptr cl, float tol, float rad,
        int cand);
74      bool compute(); //does all computation, and eventually stores the
        sphere with the most inliers as a Pnt type;
75      Pnt fit(sphere sph); //Sphere fitting in conformal space.
76
77  private:
78      bool isInlier(float x, float y, float z, sphere can); //Bool for
        determining whether a point is a inlier
79      bool sphereCheck(sphere sph); //Check if the sphere radius is inside
        tolerance limits
80      int estimateInliers(sphere sph); //Function for estimating the inliers
         for a specifc sphere.
81  };
82
83  ///////////////////////////////////////////////////////////////////
84
85  ////////////////////        CYLINDER        ///////////////////////////
86
87  ///////////////////////////////////////////////////////////////////
88  //Class for detecting a cylinder in a point cloud  using the plane-circle
        approach.
89  class ransacCylinder{
90  public:
91      PointCloud<PointXYZRGB>::Ptr cloud; //Point cloud for storing the data
         set
92      PointCloud<PointXYZRGB>::Ptr inlierCloud; //Point cloud for storing
        the inliers
93      float tol = 0; //Float for the radius tolerance
94      float searchRadius = 0; //Float for storing the radius
95      int numberInliers; //Float for storing the number of inliers
96      cylinder ranCyl; //cylinder object
97      int iterations = 1; //Maximum number of iterations
98      int candidates; //Number of candidates to detect
99
100     //Length
101     Eigen::Vector4f centroid; //Centroid of inliers
102     Eigen::Vector3f projL; //Point at the edge of cylinder on the cylinder
         axis
103     Eigen::Vector3f projS; //Point at the edge of cylinder on the cylinder
         axis
104     int largestInd = 0; //Index of point at cylinder edge
105     int smallestInd = 0; //Index of point at cylinder edge
106
107
108     void setData(PointCloud<PointXYZRGB>::Ptr cl, float tolerance, float
        rad, int inliers, int cand);
109     void compute(); //does all computation, and eventually stores the
        cylinder with most inliers
110     void cylinderLength(); //Function for determining the cylinder length
111 private:
112     bool isInlier(float x, float y, float z, Pnt p1, Pnt p2); //Bool for
        determining weather a point is a inlier
113     bool radiusCheck(float rad); //Check if the cylinder radius is inside
        tolerance limits
114 };
```

```
//Class for detecting a cylinder in a point cloud using the sphere-sphere
    approach.
class ransacCylinder2{
public:
    ransacCylinder2(); //Constructor
    PointCloud<PointXYZRGB>::Ptr cloud; //Point cloud for storing the data
     set
    PointCloud<PointXYZRGB>::Ptr inlierCloud; //Point cloud for storing
    the inliers
    float tol = 0; //Float for the radius tolerance
    float searchRadius = 0; //Float for storing the radius
    int inliers; //Integer for storing the number of inliers
    cylinder ranCyl; //Cylinder object
    ransacSphere ball; //Sphere detection object

    //Length
    Eigen::Vector4f centroid; //Centroid of inliers
    Eigen::Vector3f projL; //Point at the edge of cylinder on the cylinder
     axis
    Eigen::Vector3f projS; //Point at the edge of cylinder on the cylinder
     axis
    int largestInd = 0; //Index of point at cylinder edge
    int smallestInd = 0; //Index of point at cylinder edge

    void setData(PointCloud<PointXYZRGB>::Ptr cl, float tolerance, float
    rad);
    void compute(); //does all computation, and eventually stores the
    cylinder with most inliers
    void cylinderLength(cylinder cyl); //Function for determining the
    cylinder length
private:
    bool isInlier2(float x, float y, float z, Pnt p1, Pnt p2); //Bool for
    determining weather a point is a inlier
};

/////////////////////////////////////////////////////////////////

//////////////////     MULTIPLE SPHERES    /////////////////////////

/////////////////////////////////////////////////////////////////
//Multiple shape detection algorithm implemented for spheres
class ransacSpheres{
public:
    ransacSpheres(); //Constructor
    PointCloud<PointXYZRGB>::Ptr cloud; //Point cloud for storing the data
     set
    PointCloud<PointXYZRGB>::Ptr cloud_filtered; //Point cloud for storing
     the data set after a sphere is extracted
    vector<PointCloud<PointXYZRGB>::Ptr> *subClouds; //Vector for storing
    the subclouds
    std::vector<pcl::PointIndices::Ptr> *indexlist; //Vector for storing
    the inlier indices
    std::vector<sphere> *foundSpheres; //Vector for storing the detected
    spheres

```

```
157     int totalNumberOfIterations; //Integer for storing the total number of
         iterations accumulated
158     int totalNumberOfPopulatedNodes; //Integer for storing the total
        number of populated nodes
159     int estimatedInliers; //Integer for storing the estimated inliers
160
161     //Default octree bounds
162     double x_min = -1, y_min = -1, z_min = 0, x_max = 1, y_max = 1, z_max
        = 2;
163     double rootLength = 0;
164     float res = 0.02;
165     int depth;
166
167
168     void setData(PointCloud<PointXYZRGB>::Ptr cl, float resolution);
169     void setOctreeBounds(double xmin, double ymin, double zmin, double
        xmax, double ymax, double zmax); //Function for setting the octree
        bounds
170     void setOctree(float resolution); //Function for creating the point
        picking octree with a desired resolution
171     void setSearchOctree(float resolution); //Function for creating the
        search octree with a desired resolution
172     void compute(); //Function that runs the algorithm
173     sphere fit(sphere sph, pcl::PointIndices::Ptr inliers); //Function for
         fitting a sphere to its inliers
174     void divideCloud();
175
176 private:
177     pcl::octree::OctreePointCloudSearch<pcl::PointXYZRGB>::Ptr octSearch;
        //Search octree object
178     pcl::octree::OctreePointCloud<pcl::PointXYZRGB>::Ptr oct; //Point
        picking octree object
179
180     vector<int> shuffle(vector<int> list); //Function for shuffeling a
        index vector
181     int getInliers(sphere can); //Function for counting the inliers for a
        sphere from a subcloud
182     bool isInlier(float x, float y, float z, sphere can); //Function for
        deremining whether a point is a inlier
183     int estimateInliers(sphere sph); //Function for estimating the inliers
         of a sphere
184 };
```

ransac_body.cpp

```
1  ////////////////////////////////////////////////////////////////////
2
3  //////////////////////          PLANE       ///////////////////////////
4
5  ////////////////////////////////////////////////////////////////////
6
7  //Constructor
8  ransacPlane::ransacPlane(){
9      indexlist = new std::vector<int>;
10 }
11
12 //Setting the data and parameters of the algoritm
```

```
13 void ransacPlane::setData(PointCloud<PointXYZRGB>::Ptr cl, float tol, int
   cand){
14     cloud = cl;
15     planetol = tol;
16     candidates = cand;
17 }
18 //Runs the primitive shape detection for a plane
19
20 void ransacPlane::compute() {
21     //Variables to keep track of the algorithm
22     int ran[3];
23     int count = 0;
24     Pnt cand[candidates];
25     plane aplane; //Create a plane object from object.h
26
27     //Algorithm
28     while(count < candidates){
29
30         //Generate random indexesx
31         for(int i = 0; i<3;i++){
32             ran[i] = rand() % cloud->points.size();
33         }
34         //Create dual plane with indexed points from point cloud, using GA
35         aplane.defineDual(cloud, ran); //Creates a dual plane in confromal
     space from 3 indexed points in a point cloud
36         cand[count] = aplane.normDualPlane;
37         count++;
38     }
39
40     //Find number of inlier points for each candidate
41     int inPoints[candidates];
42     for(int i = 0; i < candidates; i++){
43         count = 0;
44         for(int j = 0; j < cloud->points.size(); j++)
45             if (isInlier(Vec(cloud->points[j].x,cloud->points[j].y,cloud->
    points[j].z).null(), cand[i])){
46                 count++;
47             }
48         inPoints[i] = count;
49     }
50
51     //Find candidate with most inliers
52     int best = 0;
53     int numBest = 0;
54     for(int i=0; i < candidates; i++){
55         if(inPoints[i] > numBest){
56             best = i;
57             numBest = inPoints[i];
58         }
59     }
60
61     //Fit the plane with the most inilers to its inliers
62     Pnt fittedPln = fit(cand[best]);
63
64     //Finally normalize
65     for(int i = 0; i < 5; i++){
```

```
66              ranPln[i] = fittedPln[i]/sqrt(pow(fittedPln[0],2)+pow(
    fittedPln[1],2)+pow(fittedPln[2],2));
67          }
68
69  }
70
71  //Function to check if a point is classified as a inlier
72  bool ransacPlane::isInlier(Pnt point, Pnt plane){
73      float distance = sqrt(pow((point <= plane)[0],2));
74      if(distance < planetol){
75          return true;
76      }else{
77          return false;
78      }
79  }
80
81  //Function that creates and stores the indexlist of the current plane
82  void ransacPlane::segmentPlane(PointCloud<PointXYZRGB>::Ptr seg){
83      delete indexlist;
84      indexlist = new std::vector<int>;
85      for(int j = 0; j < seg->points.size(); j++)
86          if (isInlier(Vec(seg->points[j].x,seg->points[j].y,seg->points[j].
    z).null(), ranPln)){
87              indexlist->push_back(j);
88          }
89  }
90
91  //Fitting of a plane to its inlier point using geometric algebra
92  Pnt ransacPlane::fit(Pnt pln){
93
94      delete indexlist;
95      indexlist = new std::vector<int>;
96      for(int j = 0; j < cloud->points.size(); j++){
97          if (isInlier(Vec(cloud->points[j].x,cloud->points[j].y,cloud->
    points[j].z).null(), pln)){
98              indexlist->push_back(j);
99          }
100     }
101
102     //Put inlierpoints into matrix for easier handling
103     Eigen::MatrixXf P(indexlist->size(),5);
104
105     for(int i=0; i < indexlist->size(); i++){
106         P(i,0) = cloud->points[indexlist->at(i)].x; //x
107         P(i,1) = cloud->points[indexlist->at(i)].y; //y
108         P(i,2) = cloud->points[indexlist->at(i)].z; //x
109         P(i,4) = -0.5*(pow(P(i,0),2) + pow(P(i,1),2) + pow(P(i,2),2));
110         P(i,3) = -1;
111     }
112
113     //Compute SVD of matrix
114     Eigen::JacobiSVD<Eigen::MatrixXf> USV(P.transpose()*P, Eigen::
    ComputeFullU | Eigen::ComputeFullV);
115
116     for(int i = 0; i < 3; i++){
117         pln[i] = USV.matrixU()(i,4); //USV.matrixU()(3,0);
118     }
```

```
119     pln[3] = USV.matrixU()(4,4); //USV.matrixU()(4,0);
120     pln[4] = USV.matrixU()(3,4); //USV.matrixU()(4,0);
121 
122     return pln;
123 }
124 
125 ///////////////////////////////////////////////////////////////////////
126 
127 /////////////////////      SPHERE      ///////////////////////////
128 
129 ///////////////////////////////////////////////////////////////////////
130 
131 //Class constructor
132 ransacSphere::ransacSphere(){
133     indexlist = new std::vector<int>;
134 }
135 
136 //Setting the data and parameters of the algoritm
137 void ransacSphere::setData(PointCloud<PointXYZRGB>::Ptr cl, float tol,
        float rad){
138     cloud = cl;
139     radius = rad;
140     radiusTolerance = tol;
141     candidates = 1;
142 }
143 
144 //Setting the data and parameters of the algoritm
145 void ransacSphere::setData(PointCloud<PointXYZRGB>::Ptr cl, float tol,
        float rad, int cand){
146     cloud = cl;
147     radius = rad;
148     radiusTolerance = tol;
149     candidates = cand;
150 }
151 
152 //Runs the primitive shape detection for a sphere
153 bool ransacSphere::compute() {
154     //Variables to keep track of the algorithm
155     int it = 0;
156     vector<int> ran(4);
157     int can = 0;
158     vector<sphere> cand;
159     sphere asphere; //Create a sphere object from object.h
160     int count;
161     vector<int> inPoints;
162 
163     //Timer
164     std::clock_t t1,t2;
165     t1=std::clock();
166 
167     //Algorithm
168     while(can < candidates && it < iterations){
169 
170         //Generate random indexes
171         for(int i = 0; i<4;i++){
172             ran[i] = rand() % cloud->points.size();
173         }
```

```
174
175         //Create dual sphere with indexed points from point cloud
176         asphere.defineDual(cloud, ran); //Creates a dual sphere in
    confromal space from 4 indexed points in a point cloud
177
178         //Check if sphere is inside tolerance
179         if(sphereCheck(asphere)){
180
181             //Count inliers
182             count = 0;
183             for(int j = 0; j < cloud->points.size(); j++) {
184                 if (isInlier(cloud->points[j].x,cloud->points[j].y,cloud->
    points[j].z, asphere)){
185                     count++;
186                 }
187             }
188
189             //Check if sphere is feasable
190             if(count >= estimateInliers(asphere)){
191                 cand.push_back(asphere);
192                 inPoints.push_back(count);
193                 can++;
194             }
195         }
196         it++;
197     }
198     //Store the performance
199     actualIt = it;
200     t2=std::clock();
201     time = ((float)t2-(float)t1)/(CLOCKS_PER_SEC);
202
203     if(can > 0){
204         //Find candidate with most inliers
205         int best = 0;
206         numInliers = 0;
207         for(int i=0; i < cand.size(); i++){
208             if(inPoints[i] > numInliers){
209                 best = i;
210                 numInliers = inPoints[i];
211             }
212         }
213
214         //Fit the sphere to its inliers
215         ranSph = fit(cand[best]);
216         float rad = sqrt((1/pow(ranSph[3],2))*(pow(ranSph[0],2)+pow(ranSph
    [1],2)+pow(ranSph[2],2)) - (2*ranSph[4]/ranSph[3]));
217         //Check if the fit is succesfull, return the unfitted sphere.
218         if (cand[best].radius/rad > 1.1 || cand[best].radius/rad < 0.9){
219             ranSph = cand[best].dualSphere;
220         }
221         //Return true if a sphere was found
222         return true;
223     }else{
224         //Return false if not
225         return false;
226     }
227 }
```

```
228
229 //Function to check if a point is classified as a inlier
230 bool ransacSphere::isInlier(float x, float y, float z, sphere can){
231     float dist = sqrt(pow((can.dualSphere[0]-x),2)+pow((can.dualSphere[1]-
        y),2)+pow((can.dualSphere[2]-z),2));
232     if ((dist > (can.radius - inlierTreshold)) && (dist < (can.radius+
        inlierTreshold))){
233         return true;
234     }else{
235         return false;
236     }
237 }
238
239 //Check if the radius is inside the tolerance limits
240 bool ransacSphere::sphereCheck(sphere sph){
241     if(sph.radius < (radius*(1+radiusTolerance)) && sph.radius > (radius
        *(1-radiusTolerance)))
242     {
243         return true;
244     }else{
245         return false;
246     }
247 }
248
249 //Function that estimates the inliers, sensor dependent
250 int ransacSphere::estimateInliers(sphere sph){
251     //a and b are found by experimental measurments with the kinect2 and
        regression in excel.
252     //y = a*(dist^(b)*rad^2)
253     float a = 439731.61228317, b =-2.2345982262;
254     float rad = sph.radius, dist = sph.dualSphere[2];
255     int estimatedInliers =  a*(pow(dist,b)*pow(rad,2));
256     if (estimatedInliers > 10){
257         return estimatedInliers;
258     }else{
259         return 999999;
260     }
261 }
262
263 //Fitting of a sphere to its inlier point using geometric algebra
264 Pnt ransacSphere::fit(sphere sph){
265     delete indexlist;
266     indexlist = new std::vector<int>;
267     for(int j = 0; j < cloud->points.size(); j++){
268         if (isInlier(cloud->points[j].x,cloud->points[j].y,cloud->points[j
        ].z, sph)){
269             indexlist->push_back(j);
270         }
271     }
272
273     //Put inlierpoints into matrix for easier handling
274     Eigen::MatrixXf P(indexlist->size(),5);
275
276     for(int i = 0; i < indexlist->size(); i++){
277         P(i,0) = cloud->points[indexlist->at(i)].x; //x
278         P(i,1) = cloud->points[indexlist->at(i)].y; //y
279         P(i,2) = cloud->points[indexlist->at(i)].z; //x
```

```
280         P(i,4) = -0.5*(pow(P(i,0),2) + pow(P(i,1),2) + pow(P(i,2),2));
281         P(i,3) = -1;
282     }
283 
284     //Compute SVD of matrix
285     Eigen::JacobiSVD<Eigen::MatrixXf> USV(P.transpose()*P, Eigen::
        ComputeFullU | Eigen::ComputeFullV);
286 
287     //Normalizing
288     for(int i = 0; i < 5; i++){
289         sph.dualSphere[i] = USV.matrixU()(i,4)/USV.matrixU()(4,4);
290     }
291     sph.dualSphere[3] = USV.matrixU()(4,4)/USV.matrixU()(4,4);
292     sph.dualSphere[4] = USV.matrixU()(3,4)/USV.matrixU()(4,4);
293     return sph.dualSphere;
294 }
295 
296 //////////////////////////////////////////////////////////////////////
297 
298 /////////////////////          CYLINDER        ///////////////////////////
299 
300 //////////////////////////////////////////////////////////////////////
301 
302 //Setting the data and parameters of the algoritm
303 void ransacCylinder::setData(PointCloud<PointXYZRGB>::Ptr cl, float
     tolerance, float rad, int inliers, int cand){
304     cloud = cl;
305     tol = tolerance;
306     searchRadius = rad;
307     numberInliers = inliers;
308     candidates = cand;
309 }
310 
311 //Runs the primitive shape detection for a cylinder using the plane-circle
      approach
312 void ransacCylinder::compute(){
313     //Variables to keep track of the algorithm
314     int ran[3];
315     int icount = 0;
316     int can = 0;
317     vector<cylinder> cand;
318     vector<int> inPoints;
319     bool terminate = true;
320     cylinder acylinder; //Create a cylinder object from object.h
321     int it = 0;
322 
323     //Algorithm
324     while(can < candidates && it < iterations){
325         it++;
326         //Generate random indexes
327         for(int i = 0; i<3;i++){
328             ran[i] = rand() % cloud->points.size();
329         }
330         //Create cylinder with indexed points from point cloud, using GA
331         acylinder.defineCylinder(cloud, ran);
332 
333         //Check if cylinder is inside radius tolerance
```

```
334         if(radiusCheck(acylinder.radius)){
335 
336             //Find number of inliers for cylinder
337             Pnt p1 = acylinder.center;
338             Pnt p2;
339             p2[0] = p1[0] + acylinder.plan[0];
340             p2[1] = p1[1] + acylinder.plan[1];
341             p2[2] = p1[2] + acylinder.plan[2];
342             icount = 0;
343             for(int i = 0; i < cloud->points.size(); i++){
344                 if(isInlier(cloud->points[i].x,cloud->points[i].y,cloud->
    points[i].z, p1,p2)){
345                     icount++;
346                 }
347             }
348             //Store cylinder and number of inliers
349             cand.push_back(acylinder);
350             inPoints.push_back(icount);
351         }
352     }
353 
354     if(can > 0){
355         //Find candidate with most inliers
356         int best = 0;
357         int numInliers = 0;
358         for(int i=0; i < cand.size(); i++){
359             if(inPoints[i] > numInliers){
360                 best = i;
361                 numInliers = inPoints[i];
362             }
363         }
364         //Store the cylinder with the most inliers
365         ranCyl = cand[best];
366     }
367 }
368 
369 //Check if the radius is inside the tolerance limits
370 bool ransacCylinder::radiusCheck(float rad){
371     if(rad > (searchRadius -(searchRadius*tol)) && rad < (searchRadius+(
    searchRadius*tol))){
372         return true;
373     }else{
374         return false;
375     }
376 }
377 
378 //Function to check if a point is classified as a inlier
379 bool ransacCylinder::isInlier(float x, float y, float z, Pnt p1, Pnt p2){
380     Eigen::Vector3d v(x-p1[0],y-p1[1],z-p1[2]);
381     Eigen::Vector3d w(x-p2[0],y-p2[1],z-p2[2]);
382     Eigen::Vector3d q(p2[0]-p1[0],p2[1]-p1[1],p2[2]-p1[2]);
383     Eigen::Vector3d cross = v.cross(w);
384     float d = sqrt(pow(cross[0],2)+pow(cross[1],2)+pow(cross[2],2))/sqrt(
    pow(q[0],2) + pow(q[1],2) + pow(q[2],2));
385     if(d < (searchRadius*(1+tol)) && d > (searchRadius*(1-tol))){
386         return true;
387     }else{
```

```
388             return false;
389         }
390 }
391
392 //Function that finds the cylinder length from the inlierpoints
393 void ransacCylinder::cylinderLength(){
394     inlierCloud.reset(new PointCloud<PointXYZRGB>);
395     //First find all inliers and store in a point cloud
396     cylinder acylinder = ranCyl;
397     Pnt p1 = acylinder.center;
398     Pnt p2;
399     p2[0] = p1[0] + acylinder.plan[0];
400     p2[1] = p1[1] + acylinder.plan[1];
401     p2[2] = p1[2] + acylinder.plan[2];
402     for(int i = 0; i < cloud->points.size(); i++){
403         if(isInlier(cloud->points[i].x,cloud->points[i].y,cloud->points[i
    ].z, p1,p2)){
404             inlierCloud->push_back(cloud->points[i]);
405         }
406     }
407     //Normalize
408     pcl::compute3DCentroid(*inlierCloud, centroid);
409
410     for(int i= 0; i < inlierCloud->size(); i++){
411         inlierCloud->points[i].x -= centroid[0];
412         inlierCloud->points[i].y -= centroid[1];
413         inlierCloud->points[i].z -= centroid[2];
414     }
415
416     //Find point farthest away from origo
417     float largestD = 0;
418     float thisD =0;
419     for(int i = 0; i < inlierCloud->size(); i++){
420         thisD = sqrt(pow(inlierCloud->points[i].x , 2) + pow(inlierCloud->
    points[i].y , 2) + pow(inlierCloud->points[i].z , 2));
421         if(largestD < thisD){
422             largestD = thisD;
423             largestInd = i;
424         }
425     }
426
427     //Find the farthest point in the opposite direction.
428     //This is done by only searching in the opposite octant of the first
    point
429     float smallestD = 0;
430     thisD = 0;
431     for(int i = 0; i < inlierCloud->size(); i++){
432         if((inlierCloud->points[i].x * inlierCloud->points[largestInd].x)
    < 0 && (inlierCloud->points[i].y * inlierCloud->points[largestInd].y)
    < 0 && (inlierCloud->points[i].z * inlierCloud->points[largestInd].z)
    < 0){
433             thisD = sqrt(pow(inlierCloud->points[i].x , 2) + pow(
    inlierCloud->points[i].y , 2) + pow(inlierCloud->points[i].z , 2));
434             if(smallestD < thisD){
435                 smallestD = thisD;
436                 smallestInd = i;
437             }
```

```
438            }
439        }
440
441        //Translate back to original position
442        for(int i= 0; i < inlierCloud->size(); i++){
443            inlierCloud->points[i].x += centroid[0];
444            inlierCloud->points[i].y += centroid[1];
445            inlierCloud->points[i].z += centroid[2];
446        }
447
448        //Find lines from "center" to largest and smallest
449        Eigen::Vector3f cl(inlierCloud->points[largestInd].x - ranCyl.center
           [0], inlierCloud->points[largestInd].y - ranCyl.center[1], inlierCloud
           ->points[largestInd].z - ranCyl.center[2]);
450        Eigen::Vector3f cs(inlierCloud->points[smallestInd].x - ranCyl.center
           [0], inlierCloud->points[smallestInd].y - ranCyl.center[1],
           inlierCloud->points[smallestInd].z - ranCyl.center[2]);
451
452
453        //Project the lines onto the centerline
454        Eigen::Vector3f centerline(ranCyl.plan[0], ranCyl.plan[1], ranCyl.plan
           [2]);
455
456        projL = cl.dot(centerline) * centerline;
457        projS = cs.dot(centerline) * centerline;
458    }
459
460    //Constructor
461    ransacCylinder2::ransacCylinder2(){
462        inlierCloud.reset(new PointCloud<PointXYZRGB>);
463    }
464
465    //Setting the data and parameters of the algoritm
466    void ransacCylinder2::setData(PointCloud<PointXYZRGB>::Ptr cl, float
           tolerance, float rad){
467        cloud = cl;
468        tol = tolerance;
469        searchRadius = rad;
470    }
471
472    //Runs the primitive shape detection for a cylinder using the sphere-
           sphere approach
473    void ransacCylinder2::compute(){
474        //Variables to keep track of the algorithm
475        inlierCloud.reset(new PointCloud<PointXYZRGB>);
476        int icount;
477        Pnt p1, p2;
478        ball.setData(cloud, tol, searchRadius); //Ransac for spheres, data is
           set
479
480        //Find the first sphere that satisfies the radius
481        ball.compute();
482        p1 = ball.ranSph;
483
484        //Find the second sphere that satisfies the radius
485        ball.compute();
486        p2 = ball.ranSph;
```

```
487
488     //Count the inliers for the cylinder and store the inliers in a inlier
         cloud
489     for(int i = 0; i < cloud->points.size(); i++){
490         if(isInlier2(cloud->points[i].x,cloud->points[i].y,cloud->points[i
        ].z, p1, p2)){
491             inlierCloud->push_back(cloud->points[i]);
492             icount++;
493         }
494     }
495     inliers = icount;
496
497     //The cylinder is created and stored.
498     ranCyl.defineCylinder(p1,p2,searchRadius);
499 }
500
501 //Function to check if a point is classified as a inlier
502 bool ransacCylinder2::isInlier2(float x, float y, float z, Pnt p1, Pnt p2)
        {
503     Eigen::Vector3d v(x-p1[0],y-p1[1],z-p1[2]);
504     Eigen::Vector3d w(x-p2[0],y-p2[1],z-p2[2]);
505     Eigen::Vector3d q(p2[0]-p1[0],p2[1]-p1[1],p2[2]-p1[2]);
506     Eigen::Vector3d cross = v.cross(w);
507     float d = sqrt(pow(cross[0],2)+pow(cross[1],2)+pow(cross[2],2))/sqrt(
        pow(q[0],2) + pow(q[1],2) + pow(q[2],2));
508     if(d < (searchRadius*(1+tol)) && d > (searchRadius*(1-tol))){
509         return true;
510     }else{
511         return false;
512     }
513 }
514
515 //Function that finds the cylinder length from the inlierpoints
516 void ransacCylinder2::cylinderLength(cylinder cyl){
517
518     //Normalize
519     pcl::compute3DCentroid(*inlierCloud, centroid);
520
521     for(int i= 0; i < inlierCloud->size(); i++){
522         inlierCloud->points[i].x -= centroid[0];
523         inlierCloud->points[i].y -= centroid[1];
524         inlierCloud->points[i].z -= centroid[2];
525     }
526
527     //Find point farthest away from origo
528     float largestD = 0;
529     float thisD =0;
530     for(int i = 0; i < inlierCloud->size(); i++){
531         thisD = sqrt(pow(inlierCloud->points[i].x , 2) + pow(inlierCloud->
        points[i].y , 2) + pow(inlierCloud->points[i].z , 2));
532         if(largestD < thisD){
533             largestD = thisD;
534             largestInd = i;
535         }
536     }
537
538     //Find the farthest point in the opposite direction.
```

```
539     //This is done by only searching in the opposite octant of the first
        point
540     float smallestD = 0;
541     thisD = 0;
542     for(int i = 0; i < inlierCloud->size(); i++){
543         if((inlierCloud->points[i].x * inlierCloud->points[largestInd].x)
        < 0 && (inlierCloud->points[i].y * inlierCloud->points[largestInd].y)
        < 0 && (inlierCloud->points[i].z * inlierCloud->points[largestInd].z)
        < 0){
544             thisD = sqrt(pow(inlierCloud->points[i].x , 2) + pow(
        inlierCloud->points[i].y , 2) + pow(inlierCloud->points[i].z , 2));
545             if(smallestD < thisD){
546                 smallestD = thisD;
547                 smallestInd = i;
548             }
549         }
550     }
551 
552     //Translate back to original position
553     for(int i= 0; i < inlierCloud->size(); i++){
554         inlierCloud->points[i].x += centroid[0];
555         inlierCloud->points[i].y += centroid[1];
556         inlierCloud->points[i].z += centroid[2];
557     }
558 
559     //Find lines from "center" to largest and smallest
560     Eigen::Vector3f cl(inlierCloud->points[largestInd].x - ranCyl.center
        [0], inlierCloud->points[largestInd].y - ranCyl.center[1], inlierCloud
        ->points[largestInd].z - ranCyl.center[2]);
561     Eigen::Vector3f cs(inlierCloud->points[smallestInd].x - ranCyl.center
        [0], inlierCloud->points[smallestInd].y - ranCyl.center[1],
        inlierCloud->points[smallestInd].z - ranCyl.center[2]);
562 
563 
564     //Project the lines onto the centerline
565     Eigen::Vector3f centerline(ranCyl.plan[0], ranCyl.plan[1], ranCyl.plan
        [2]);
566 
567     projL = cl.dot(centerline) * centerline;
568     projS = cs.dot(centerline) * centerline;
569 }
570 
571 ////////////////////////////////////////////////////////////////////
572 
573 /////////////////////// MULTIPLE SPHERES /////////////////////////
574 
575 ////////////////////////////////////////////////////////////////////
576 //Constructor
577 ransacSpheres::ransacSpheres()
578     :oct (new pcl::octree::OctreePointCloud<PointXYZRGB>(res)),
579     octSearch (new pcl::octree::OctreePointCloudSearch<PointXYZRGB>(res)){
580 }
581 
582 //Setting the data and parameters of the algoritm
583 void ransacSpheres::setData(PointCloud<PointXYZRGB>::Ptr cl, float
        resolution){
584     cloud = cl;
```

```
585     cloud_filtered = cl;
586     res = resolution;
587     divideCloud(); //Divides the cloud to 4 random subsets, used for
        faster scoring
588     indexlist = new std::vector<pcl::PointIndices::Ptr>;
589     foundSpheres = new std::vector<sphere>;
590 }
591
592 //Function for suffeling a list of integers
593 vector<int> ransacSpheres::shuffle(vector<int> list){
594     int ran, temp;
595     for(int i = list.size(); i > 0; i--){
596         ran = rand() % i;
597         temp = list[i-1];
598         list[i-1] = list[ran];
599         list[ran] = temp;
600     }
601     return list;
602 }
603
604 //Function that divides the point cloud to 4 random subsets, used for
        faster scoring
605 void ransacSpheres::divideCloud(){
606     subClouds = new vector<PointCloud<PointXYZRGB>::Ptr>;
607     PointCloud<PointXYZRGB>::Ptr temp(new PointCloud<PointXYZRGB>);
608     vector<int> list (cloud_filtered->points.size());
609     //Store all the indexes in a vector
610     for(int i = 0; i < cloud_filtered->points.size(); i++){
611         list[i] = i;
612     }
613     //Shuffle the vector
614     list = shuffle(list);
615
616     //Split the point cloud into 4 "subclouds"
617     int count = 1;
618     for(int i = 0; i < cloud_filtered->points.size(); i++){
619         temp->push_back(cloud_filtered->points[list[i]]);
620         if(i == ((cloud_filtered->points.size()/4)*count-1)){
621             subClouds->push_back(temp);
622             temp.reset(new PointCloud<PointXYZRGB>);
623             count++;
624         }
625     }
626
627     //Put the "rest" in the first subcloud
628     for(int i = cloud_filtered->points.size()-1; i > cloud_filtered->
        points.size()-1-(cloud_filtered->points.size()%4); i--){
629         subClouds->at(0)->push_back(cloud_filtered->points[i]);
630     }
631 }
632
633 //Function that creates octree for point picking
634 void ransacSpheres::setOctree(float resolution){
635     oct->deleteTree();
636     oct->setResolution(resolution);
637     oct->defineBoundingBox(x_min,y_min,z_min,x_max,y_max,z_max);
638     double xma,yma,zma,xm,ym,zm;
```

```
    oct->getBoundingBox(xm,ym,zm,xma,yma,zma);
    rootLength = sqrt(pow(xma-xm,2));
    oct->setInputCloud(cloud_filtered);
    oct->addPointsFromInputCloud();
}

//Function that creates octree for searching
void ransacSpheres::setSearchOctree(float resolution){
    octSearch->deleteTree();
    octSearch->setResolution(resolution);
    octSearch->defineBoundingBox(x_min,y_min,z_min,x_max,y_max,z_max);
    octSearch->setInputCloud(subClouds->at(0));
    octSearch->addPointsFromInputCloud();
}

//Function that sets the bounds of the octree root node
void ransacSpheres::setOctreeBounds(double xmin, double ymin, double zmin,
     double xmax, double ymax, double zmax){
    x_min = xmin; y_min = ymin; z_min = zmin; x_max = xmax; y_max = ymax;
    z_max = zmax;
    oct->deleteTree();
    oct->setResolution(res);
    oct->defineBoundingBox(x_min,y_min,z_min,x_max,y_max,z_max);
    double xma,yma,zma,xm,ym,zm;
    oct->getBoundingBox(xm,ym,zm,xma,yma,zma);
    rootLength = sqrt(pow(xma-xm,2));

    //Start at rootlength
    oct->deleteTree();
    oct->setResolution(rootLength);
    oct->defineBoundingBox(x_min,y_min,z_min,x_max,y_max,z_max);
    octSearch->deleteTree();
    octSearch->setResolution(rootLength);
    octSearch->defineBoundingBox(x_min,y_min,z_min,x_max,y_max,z_max);
}

//Runs the multipe shape detection algorithm
void ransacSpheres::compute() {
    //Variables to keep track of the algorithm
    pcl::PointIndices::Ptr inliers (new pcl::PointIndices ());
    std::vector<int> indices;
    pcl::ExtractIndices<pcl::PointXYZRGB> extract;
    octree::OctreePointCloud<PointXYZRGB>::LeafNodeIterator it;
    pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_f (new pcl::PointCloud<
    pcl::PointXYZRGB>);
    bool restart = false;
    float thisRes = rootLength;
    totalNumberOfIterations = 0;
    totalNumberOfPopulatedNodes= 0;

    //Iterates through the octree
    while (thisRes >= res) {

        setOctree(thisRes);
        setSearchOctree(thisRes);
        int tempInl = 0;
        int bestInl = 0;
```

```
693
694         //Iterate throug all nodes at present level
695         it.reset();
696         it = oct->leaf_begin();
697         sphere asphere;
698         sphere tempSphere;
699         while(*++it && !restart){
700             bestInl = 0;
701             tempInl = 0;
702             //Only consider populated nodes
703             if(it.isLeafNode()){
704                 indices.resize(0);
705                 it.getLeafContainer().getPointIndices(indices);
706                 //Only consider nodes with more than 4 points
707                 if(indices.size() > 4){
708                     totalNumberOfPopulatedNodes++;
709                     //Initialzes 4 spheres in the current node
710                     for(int i = 0; i < 4; i++){
711                         totalNumberOfIterations++;
712                         indices = shuffle(indices);
713                         tempSphere.defineDual(cloud_filtered, indices);
714                         //Store the sphere with the most inliers
715                         if(tempSphere.z < 3 && tempSphere.z > 0.2 &&
    tempSphere.radius > 0.015 && tempSphere.radius < 0.13){
716                             tempInl = getInliers(tempSphere);
717                             if(bestInl < tempInl){
718                                 bestInl = tempInl;
719                                 asphere = tempSphere;
720                                 tempInl = 0;
721                             }
722                         }
723
724                     }
725                     //Score the sphere, the estimate have to be divided by
     4 because only a quater of the total point is used for scoring
726                     if((bestInl >= estimateInliers(asphere)*0.25)){
727                         //Find and store inliers all the inliers
728                         inliers.reset(new pcl::PointIndices());
729                         for(int i = 0; i < cloud_filtered->points.size();
    i++){
730                             if(isInlier(cloud_filtered->points[i].x,
    cloud_filtered->points[i].y, cloud_filtered->points[i].z, asphere)){
731                                 inliers->indices.push_back(i);
732                             }
733                         }
734
735                         //All of the inliers are checked against the
    estimate
736                         if(inliers->indices.size() >= estimateInliers(
    asphere)){
737                             //Sphere is accepted
738                             indexlist->push_back(inliers);
739
740                             //Fit the sphere
741                             sphere fitted = fit(asphere, inliers);
742                             if (fitted.radius/asphere.radius > 1.1 ||
    fitted.radius/asphere.radius < 0.9){
```

```
743                                 std::cout << "Bad fitting!" << std::endl;
744                                 foundSpheres->push_back(asphere);
745                             }else{
746                                 foundSpheres->push_back(fitted);
747                             }
748 
749                             //Extract inliers from the point cloud
750                             extract.setInputCloud (cloud_filtered);
751                             extract.setIndices (inliers);
752                             extract.setNegative (true);
753                             extract.filter (*cloud_f);
754                             cloud_filtered.reset();
755                             cloud_filtered = cloud_f;
756 
757                             //Reset the octrees and restart the iteration
        at the same resolution
758                             restart = true;
759                         }
760 
761                     }
762                 }
763             }
764         }
765         if(!restart){
766             //No sphere was found at the current level
767             //Continue iteration at the lower level
768             thisRes = thisRes/2;
769         }else{
770             //A sphere was found and the iteration is restarted at the
        current level
771             restart = false;
772             divideCloud();
773         }
774     }
775 }
776 
777 //Function to check if a point is classified as a inlier
778 bool ransacSpheres::isInlier(float x, float y, float z, sphere can){
779     float dist = sqrt(pow((can.dualSphere[0]-x),2)+pow((can.dualSphere[1]-
        y),2)+pow((can.dualSphere[2]-z),2));
780     float inlierTreshold = (7.0f/1800.0f) + (1.0f/18.0f)*can.radius; //
        Interpolation function
781     if ((dist > (can.radius - inlierTreshold)) && (dist < (can.radius +
        inlierTreshold))){
782         return true;
783     }else{
784         return false;
785     }
786 }
787 
788 //Function that counts and return the inliers of a sphere for a subcloud
789 int ransacSpheres::getInliers(sphere can){
790     int inl = 0;
791     pcl:PointXYZRGB searchPoint;
792     std::vector<int> pointIdxVec;
793     std::vector<float> pointNKNSquaredDistance;
794 
```

```
795     searchPoint.x = can.x;
796     searchPoint.y = can.y;
797     searchPoint.z = can.z;
798     octSearch->radiusSearch(searchPoint, can.radius + 0.05, pointIdxVec,
        pointNKNSquaredDistance);
799     float inlierTreshold = (7.0f/1800.0f) + (1.0f/18.0f)*can.radius; //
        Interpolation function
800     for(int i = 0; i < pointIdxVec.size(); i++){
801         if((sqrt(pointNKNSquaredDistance[i]) < (can.radius +
        inlierTreshold)) && (sqrt(pointNKNSquaredDistance[i]) > (can.radius -
        inlierTreshold))){
802             inl++;
803         }
804     }
805     return inl;
806 }
807
808 //Function that estimates the number of inliers, used to decide whether a
        sphere is accepted
809 int ransacSpheres::estimateInliers(sphere sph){
810     //a and b are found by experimental measurments with the kinect2 and
        regression in excel.
811     //y = a*(dist^(b)*rad^2)
812     float a = 439731.61228317, b =-2.2345982262;
813     float rad = sph.radius, dist = sph.dualSphere[2];
814     estimatedInliers =  a*(pow(dist,b)*pow(rad,2));
815     if (estimatedInliers > 10){
816         return estimatedInliers;
817     }else{
818         return 999999;
819     }
820 }
821
822 //Function that fits a sphere to its inlier point using conformal
        geometric algebra
823 sphere ransacSpheres::fit(sphere sph, pcl::PointIndices::Ptr inliers){
824
825     //Put inlierpoints into matrix for easier handling
826     Eigen::MatrixXf P(inliers->indices.size(),5);
827     for(int i = 0; i < inliers->indices.size(); i++){
828         P(i,0) = cloud_filtered->points[inliers->indices.at(i)].x; //x
829         P(i,1) = cloud_filtered->points[inliers->indices.at(i)].y; //y
830         P(i,2) = cloud_filtered->points[inliers->indices.at(i)].z; //x
831         P(i,4) = -0.5*(pow(P(i,0),2) + pow(P(i,1),2) + pow(P(i,2),2));
832         P(i,3) = -1;
833     }
834
835     //Compute SVD of matrix
836     Eigen::JacobiSVD<Eigen::MatrixXf> USV(P.transpose()*P, Eigen::
        ComputeFullU | Eigen::ComputeFullV);
837
838     //Normalize
839     Pnt temp;
840     for(int i = 0; i < 5; i++){
841         temp[i] = USV.matrixU()(i,4)/USV.matrixU()(4,4);
842     }
843     temp[3] = USV.matrixU()(4,4)/USV.matrixU()(4,4);
```

```
844     temp[4] = USV.matrixU()(3,4)/USV.matrixU()(4,4);
845     float radius = sqrt((1/pow(temp[3],2))*(pow(temp[0],2)+pow(temp[1],2)+
        pow(temp[2],2)) - (2*temp[4]/temp[3]));
846     sph.defineDual(temp[0], temp[1], temp[2], radius);
847     return sph;
848 }
```

C Digital Appendix

The digital appendix is submitted with the electronic document as *AkselSveier-Master-DigitalAppendix.zip*. It includes:

- A video showing the robotic pick-and-place demonstration.
- A folder containing the **objects.h**, **objects_body.cpp**, **ransac.h** and **ransac_body.cpp** files.
- A folder containg the developed GUI software, including a **README.md** file and the *vesor* library.